applied sciences

Article

Malicious JavaScript Detection Based on Bidirectional LSTM Model

Xuyan Song [1,2], Chen Chen [2,3], Baojiang Cui [1,2,*] and Junsong Fu [1,2]

1 School of Cyberspace Security, Beijing University of Posts and Telecommunications, Beijing 100876, China; sungxy@bupt.edu.cn (X.S.); fujs@bupt.edu.cn (J.F.)
2 National Engineering Laboratory for Mobile Network Security, Beijing 100876, China; chencc924@bupt.edu.cn
3 School of Computer Science, Beijing University of Posts and Telecommunications, Beijing 100876, China
* Correspondence: cuibj@bupt.edu.cn

Received: 24 April 2020; Accepted: 10 May 2020; Published: 16 May 2020

Abstract: JavaScript has been widely used on the Internet because of its powerful features, and almost all the websites use it to provide dynamic functions. However, these dynamic natures also carry potential risks. The authors of the malicious scripts started using JavaScript to launch various attacks, such as Cross-Site Scripting (XSS), Cross-site Request Forgery (CSRF), and drive-by download attack. Traditional malicious script detection relies on expert knowledge, but even for experts, this is an error-prone task. To solve this problem, many learning-based methods for malicious JavaScript detection are being explored. In this paper, we propose a novel deep learning-based method for malicious JavaScript detection. In order to extract semantic information from JavaScript programs, we construct the Program Dependency Graph (PDG) and generate *semantic slices*, which preserve rich semantic information and are easy to transform into vectors. Then, a malicious JavaScript detection model based on the Bidirectional Long Short-Term Memory (BLSTM) neural network is proposed. Experimental results show that, in comparison with the other five methods, our model achieved the best performance, with an accuracy of 97.71% and an F1-score of 98.29%.

Keywords: cyber security; malware detection; program slice; deep learning; malicious JavaScript; Bidirectional LSTM

1. Introduction

JavaScript is a lightweight scripting language that is often included in web pages to provide additional dynamic functionality [1]. The study of Bichhawat et al. [2] shows that more than 95% of Web sites choose the JavaScript language for Web front-end development. However, JavaScript's dynamic nature [3] makes it widely abused by malware authors to attack the network users' computers and mobile devices. According to the Internet Security Threat Report [4], in 2018, 1 in 10 analyzed URLs were identified as being malicious, and meanwhile 1 in 16 URLs were malicious in 2017. These malicious codes mainly employ the script language such as JavaScript and VBScript. For example, FormJacking [5] is a JavaScript-based attack that steals credit card details and user privacy on the checkout pages of e-commerce websites. In 2018, an average of 4,800 websites were compromised by FormJacking attacks each month. In addition, there are various types of attacks based on malicious JavaScript, such as the XSS, drive-by download attack [6], and even the Distributed Denial of Service (DDoS) Attack [7].

For security researchers, finding these malicious JavaScripts is usually much more difficult than expected. On the one hand, attackers often use obfuscation techniques to hide their malicious intent to evade the detection of antivirus software. On the other hand, the dynamic nature and flexible syntax of JavaScript further increase the complexity of analysis. Attackers can dynamically generate attack instructions during script execution, which makes traditional static analysis techniques [8,9] fail to perform well.

Traditional solutions for malicious JavaScript detection rely on expert knowledge. But even for experts, determining whether a JavaScript file is malicious is an error-prone and time-consuming task because of the complexity of the problem. To overcome these limitations, many new learning-based methods are being explored. Researchers believe that these methods can better understand the complex malicious JavaScripts and automatically detect and analyze new variants. In recent years, researchers have employed some machine learning algorithms to model and detect JavaScript malware by different intermediate representations. In particular, Fang et al. [10] used the V8 engine to generate JavaScript bytecode and trained deep learning models based on features extracted from the bytecode. Similarly, Stokes et al. [11] proposed a new deep learning model to process the byte sequence of detecting malicious JavaScript. Moreover, Hao et al. [12] extracted API symbol features from the abstract syntax tree and used the Naive Bayes algorithm to detect malicious JavaScript. Based on the N-grams features extracted from the abstract syntax tree, Fass et al. [13] also proposed a fully static analysis method. With JSAC, Liang et al. [14] exacted feature from malicious JavaScript's abstract syntax tree (AST) and control flow graph (CFG) and implemented tree-based convolutional neural networks (CNN) and graph-based CNN to process the feature, respectively.

The above studies treat the program as a special natural language. There are indeed similarities between programs and natural languages. For example, they are both in the form of sequences, and they all conform to specific syntax rules. However, the syntax of the program is more flexible, and the dependency relationship between statements is not determined by their distance, which means that simply traversing the abstract syntax tree or the token sequence cannot effectively capture the semantic information. This means that abstract representations that are more sensitive to program characteristics make malicious JavaScript detection more accurate.

Therefore, we propose a novel abstract code representation for malicious JavaScript detection, which preserve rich semantic information and are easy to transform into vectors. Then, a malicious code detection model based on the Bidirectional Long Short-Term Memory (BLSTM) neural network is proposed. In summary, the contributions made by this paper are as follows:

- We propose a new approach for detecting malicious JavaScript based on the BLSTM neural network. In order to extract semantic information and transform code into vectors we propose the concept of *semantic slice* for malicious JavaScript and leverage specific key functions as the slicing criterion to generate program slices.
- We studied the influence of obfuscation technique on neural classifier, which is one of the key steps in data processing. Through the comparison of multiple sets of experiments, we observe that the obfuscation can reduce the performance of the BSLTM neural network. Then we comprehensively discuss the reasons for this situation.
- We implemented the BLSTM neural network and compared it with four other machine learning-based detection models and a traditional antivirus software. Experimental results show that our model achieves the best performance.

The rest of this paper is organized as follows. We introduce related work in Section 2. The abstract code representation of JavaScript is proposed in Section 3 and the BLSTM neural network for malicious JavaScript detection is introduced in Section 4. In Section 5, we introduce experimental designs and experimental results. We discuss the limitations in Section 6 and the conclusion in Section 7.

2. Related Work

In addition to the previously mentioned malicious JavaScript detection algorithms, some other methods are discussed in this section. We divide the traditional related work into two categories: dynamic analysis and static analysis, and they are presented in the following, respectively.

2.1. Dynamic Analysis of Malicious JavaScript

Dynamic analysis can capture the runtime behavior of JavaScript programs, which is important for detecting runtime attack vectors.

Cova et al. [15] proposed a system for detecting and analyzing malicious JavaScript code, which uses ten characteristics to describe the entire life cycle of a vulnerability attack, from the initial request to the actual exploitation of the vulnerable component. During the detection process, the system can identify the abnormal JavaScript code by simulating the behavior of the abnormal JavaScript code and comparing it with the established profiles.

Simulating the behavior of JavaScript may lead to distortion, and hence some researchers used JavaScript engines to capture dynamic features. Kyungtae et al. [16] proposed JavaScript lightweight execution engine J-Force. When using the engine to execute and traverse code execution paths that may hide malicious programs, the J-Force records and selects different branches to form new paths during execution. The function call parameters, tracked objects, and Document Object Model (DOM) node content are also used to detect malicious and DOM injection attacks. Because J1-Force simulates all possible execution paths of JavaScript code, the detection time of a complex JavaScript with multi-branch and multi-path increases exponentially, so the detection speed is less than that of other methods.

Similarly, Jayasinghe et al. [17] proposed a method for dynamically monitoring web browsers to generate byte streams, which can detect potential drive-by download attacks in the web browser environment.

Considering the JavaScript's support for Android, a method for detecting attacks in cross-platform applications was proposed by Mao et al. [18]. This method extracts runtime information from the application and uses function-level execution information to distinguish benign and malicious behavior.

In conclusion, dynamic analysis plays an important role in detecting malicious JavaScript. However, in order to obtain higher accuracy, dynamic analysis requires more time and resources (e.g., memory or CPU). Sometimes, dynamic analysis fails to obtain test results in an acceptable time.

2.2. Static Analysis of Malicious JavaScript

Static analysis aims to analyze the static behavior features of JavaScript programs. It usually transforms the code into some suitable intermediate representation (e.g., token sequence, AST), and extracts features from them. These features are modeled to determine whether the sample is malware.

Davide et al. [19] extracted features from the JavaScript code and URL associated with the HTML content of the web page, and then used static analysis techniques to analyze these features.

Similarly, Zozzle [20] also extracted features from the AST. Zozzle is a static JavaScript malware detection system deployed in most commercial browsers. It uses Bayesian classification of hierarchical features of the AST of JavaScript to identify syntax elements of highly predictive malware. Experiment results show that Zozzle can effectively detect JavaScript malware through static code analysis.

Some researchers prefer to extract features manually. For example, Likarishet et al. [21] manually selected 65 malicious JavaScript features through statistical analysis of a large number of samples, and then test the effectiveness of these features. However, manual feature selection relies on expertise and extensive experience. This means the method cannot detect new malware variant.

In addition, the authors in [22–24] use lexical analysis to extract code features, and propose detection tools for malicious pdf and SQL injection. Some other methods [25–28] extract syntax information from AST to detect malicious JavaScript.

Compared with dynamic analysis, static analysis requires less resources and has fast detection speed. Through reasonable design, static analysis can also achieve high performance.

3. Abstract Code Representation

Since deep learning algorithms use vectors as input, JavaScript programs need to be transformed into vector representations. However, this transformation process is not arbitrary, because the transformed vector needs to retain the semantic information of the program. This means that we need a suitable abstract code representation to connect programs and vectors.

As shown in Figure 1, we first deobfuscate the raw JavaScript code to expose semantic information. Then, the Program Dependency Graph (PDG) is generated based on the Abstract Syntax Tree (AST) and the Control Flow Graph (CFG). The PDG can help us find the dependencies between statements and facilitate program slicing. Finally, we obtained the *semantic slices* of a JavaScript program, which preserve rich semantic information and are easy to transform into vectors.

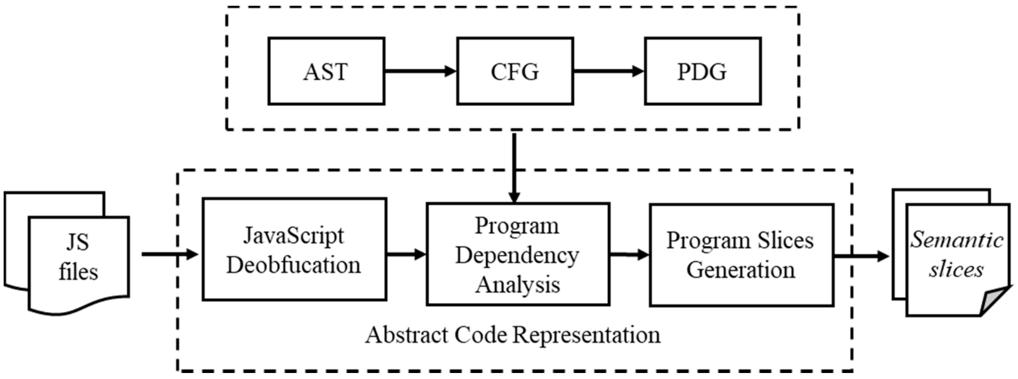

AST
CFG
PDG
JS files
JavaScript Deobfucation
Program Dependency Analysis
Program Slices Generation
Abstract Code Representation
Semantic slices

Figure 1. A brief review of abstract code representation.

3.1. Defining Semantic Slice

In order to extract the semantic features of the program and transform it into the input vector of the neural network, we need a reasonable abstract code representation, just as one would with processing natural language. However, there are many differences between program code and natural language. On the one hand, in natural language, the semantics of a word is limited to a small space, whereas in program code, words cannot be directly mapped to a limited semantic space [11]. On the other hand, the syntax of the program code is more flexible than natural language. In a program, two adjacent statements may have no relationship at all, which is rare in natural languages. This means that program code cannot be viewed as a sequence of words just like natural language. Therefore, we propose a *semantic slice* as an abstract code representation, which is much finer than using words.

The *semantic slice* is a kind of program slice, which consists of a number of statements that have semantical dependencies on each other, such as control dependence or data dependence. In order to generate *semantic slice*, we need to introduce the *key function*, which refer to specific API functions in the program. Intuitively, the *key function* can be seen as an indicator of code snippets implying the existence of a malicious script. For example, malicious code must execute some instructions to achieve their evil intent. This process requires calling specific system API functions, such as the *eval* function. Therefore, we take the *eval* function as a *key function* and get all the statements that are dependent on it. These statements are considered a *semantic slice*. This is inspired by the observation that a large percentage of malicious scripts require specific API function calls to trigger malicious behavior.

In this paper, we leverage the specific *key functions* to generate *semantic slices* of JavaScript programs and demonstrate its effectiveness in malicious JavaScript detection based on the BLSTM neural network.

3.2. JavaScript Deobfuscation

The obfuscation technique is widely used in JavaScript programs. On the one hand, benign obfuscation can protect intellectual property rights and prevent code from being plagiarized. On the other hand, to evade the detection of antivirus software and hide malicious intent, malicious JavaScript uses a variety of obfuscation techniques. This presents a challenge to malicious script detection. Several categories of obfuscation have been collected, as stated by [29–31], and they are discussed as follows:

(1) Randomization Obfuscation includes randomly inserting or modifying script elements without changing the semantics of the script, for example, adding whitespace characters, randomizing variable and function names. These operations invalidate content-based detectors.

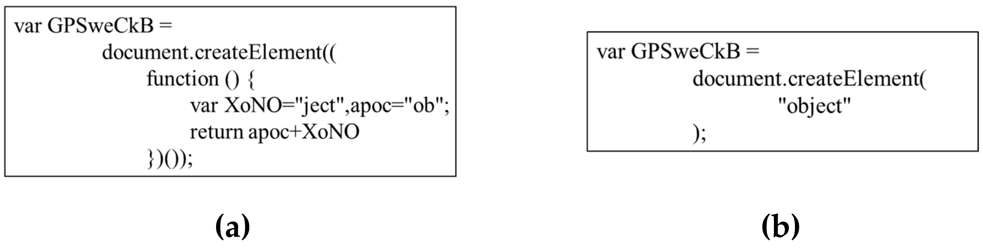
(2) Encoding obfuscation uses standard encoding or custom encoding to encode key variables and strings, making detection tools unable to identify real information.

(3) Data obfuscation uses various operations to modify and regroup strings in programs, including, string concatenation, string splitting, and string replacement.

(4) Logical obfuscation refers to adding functional-independent logical structures to the code, such as a large number of conditional branch statements.

It should be noted that the obfuscation technique is not one technique but a combination of multiple techniques, which brings potential risks to the deep learning-based classifier for detecting malicious JavaScript. For example, the training dataset for the neural network is mainly composed of encoding obfuscated samples, in which there are a large number of encoded variables in the program, then the trained neural network model may only tend to identify samples with abundant number of encoded variables as malicious script. This makes the classifier more prone to false positives when detecting samples processed by other obfuscation techniques.

To solve this problem, we need to deobfuscate the samples. We select JSDetox [32] and other 5 online JavaScript deobfuscation tools to process the samples. JSDetox is a famous JavaScript deobfuscation tool which can analyze the given code and tries to solve calculations through static analysis of the code. As shown in Figure 2, in the original code, the parameter of *document.createElement()* is the result of an anonymous function calculation. Through the processing of JSDetox, the calculation results of the variables *XoNo* and *apoc* are placed in the correct position of the *document.createElement()*.



(a)
```
var GPSweCkB =
        document.createElement((
            function () {
                var XoNO="ject",apoc="ob";
                return apoc+XoNO
            })());
```

(b)
```
var GPSweCkB =
        document.createElement(
                "object"
        );
```

Figure 2. An example of deobfuscation using JSDetox. (**a**) shows the original code; (**b**) shows the analysis result.

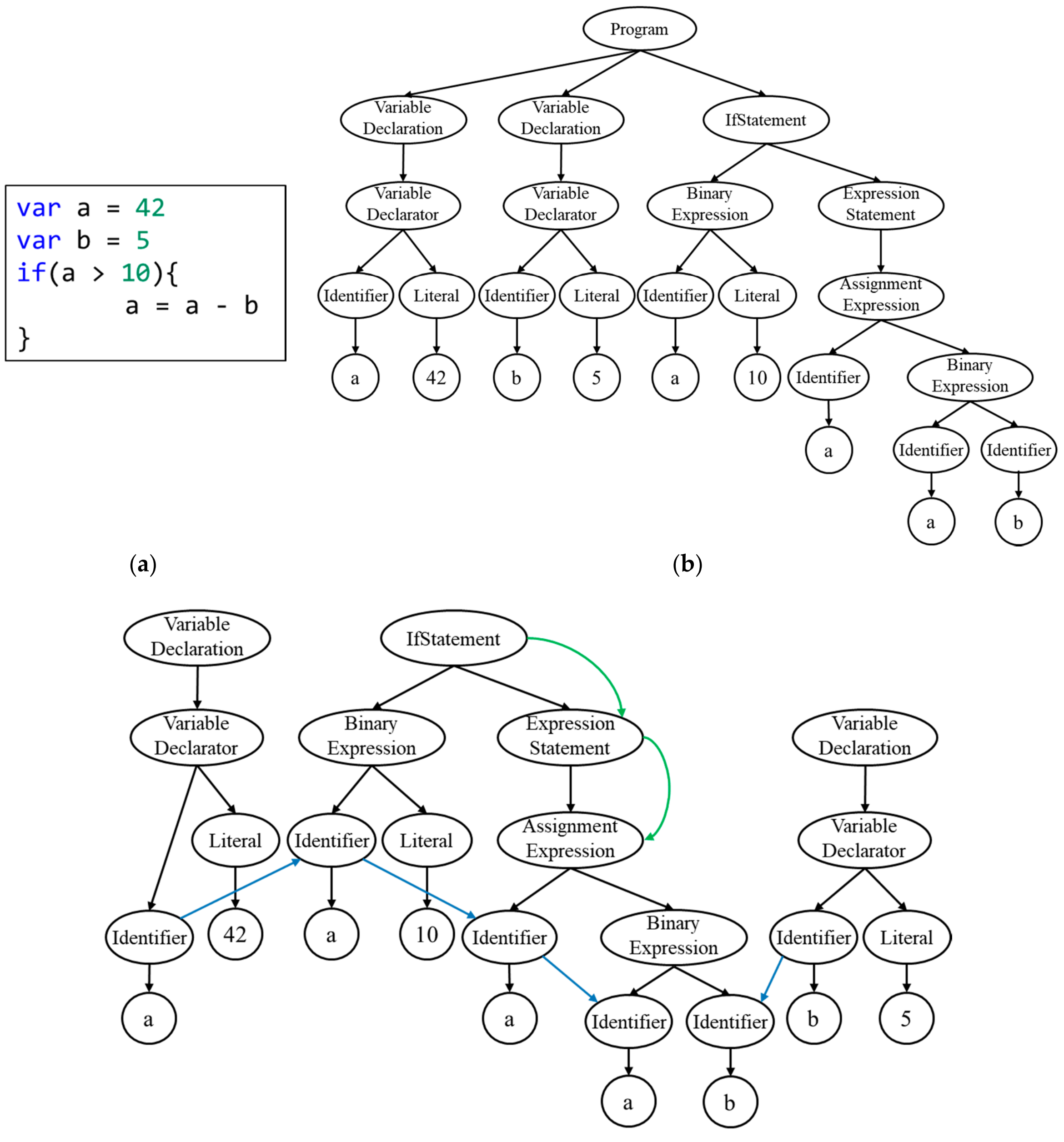
After further analysis, we find that the obfuscation techniques lead to huge changes in the results of static analysis (e.g., lexical analysis, syntactic analysis) of the code. Table 1 shows the statistic result of the code in Figure 2 after lexical analysis and syntactic analysis. The statistic result shows the obfuscation increases the number of tokens and the number of AST nodes by 3 and 2 times, respectively. These changes inevitably distort features extracted at the lexical level and syntactic level. Also, this indicates the necessary to leverage the deobfuscated samples to train a neural network.

Table 1. Statistical results after static analysis of the code in Figure 2.

State	#Token	#AST Node
Original code	30	24
Analysis result	9	10

3.3. *Program Dependency Analysis*

In order to generate *semantic slices*, we need to generate the Program Dependency Graph (PDG) of input programs, which can help us analyze the dependency between statements. For this purpose, we first parse the program to get the Abstract Syntax Tree (AST) of the program. Then, the Control Flaw Graph (CPG) is generated by adding control dependencies to the AST. Finally, the data dependencies are added to the CFG. Figure 3. shows how to generate PDG with an example.



```
var a = 42
var b = 5
if(a > 10){
        a = a - b
}
```

Figure 3. The process from source code to abstract syntax tree to program dependency graph. (**a**) shows the sample code with an *if* statement; (**b**) shows the Abstract Syntax Tree generated by the sample code; (**c**) shows the Program Dependency Graph generated by the Abstract Syntax Tree.

Syntactic Analysis. We leverage Esprima [33], a famous open source JavaScript parser, to take syntactic analysis. It takes a valid JavaScript sample as input and generates the corresponding AST. Overall, Esprima consists of 69 syntax units, which are divided into four categories: Expressions, Patterns, Statements, and Declarations. Figure 3b shows the parsing result, where the sample from Figure 3a is transformed into an AST. In order to make it easy to understand, this paper retains the variable name and constant value in the leaf node, but they are not part of the AST.

The AST retains the structural information of the source code, but does not contain any control dependency and data dependency.

Control Flow Analysis. Compared with the AST, the CFG is able to infer the conditions that must be met when executing a specific program path. To this end, statements are represented by nodes connected by directed edges to represent control flow. As shown in Figure 3c, the CFG is constructed over the previous AST in order to preserve the relationship between nodes. We refer to the extension of the AST with control flow edges as CFG. There are many advantages to a CFG, for example, it can easily summarize the information of each statement, it can easily locate the code that is not accessible in the program, and it can easily find syntax structures such as loops in the CFG.

To add control dependencies in the AST, we first traverse the AST and look for nodes with conditional syntax units, such as *IFStatement* and *SwitchStatement*. Then we traverse the subtree of the node, and add control dependent edges between the parent and child nodes, that is, the green directed edges on Figure 3c.

Data Flow Analysis. Finally, we implemented the PDG, which extends the previous CFG with the data dependency. As shown in Figure 3c, the nodes are connected with the directed blue edge if an element (e.g., variables, objects, function) defined or modified at source node is used at the target node. Because the PDG captures the data and control dependency between the different program components, it is not influenced by the order of statements. In addition, the PDG implementation follows the scoping rules of JavaScript and distinguishes between function declarations and function expressions. Also, the function call node is connected to the corresponding function definition node with data dependency, which define the PDG at the program level.

3.4. Program Slices Generation

Before generating program slices, we need to choose the appropriate *key function*, which can guide the slicing to cover as many possible malicious statements as possible. We selected 2000 malicious scripts from the dataset (detailed in Section 5) and performed statistical analysis to study the features of API function calls. There are many studies that demonstrate the usefulness of API function calls for malicious script detection as well, such as [34,35]. Combining existing research with our statistical results, the *key functions* are divided into 4 categories according to their features. Table 2 shows the API functions corresponding to each category of *key functions*.

Table 2. The *key functions* categories.

No	Categories	API Functions
1	String Operation	*substring(), charAt(), split(), concat(), slice(), substr()*
2	Encoding Operation	*escape(), unescape(), string(), fromCharCode()*
3	URL Redirection	*setTimeout(), location.reload(), location.replace(), ducument.URL(), doucument.location (), document.referrer(),*
4	Specific Behaviors	*eval(), setTime(), setInterval(), ActiveXObject(), createElement(), document.write(), document.writln(), document.replaceChildren()*

String Operations: The API functions regarding string are widely used in malicious JavaScript. These functions are used by attackers to obfuscate code, which allows malicious scripts to evade the detection of most programmers.

Encoding Operations: Similar to string operations, the API functions regarding encoding encode variables and strings, making the attack scripts difficult to understand.

URL Redirection: JavaScript-based redirection technology is widely used in malicious pages to redirect spam.

Specific Behaviors: Malicious scripts frequently execute specific instructions dynamically, for example, use the *document.replaceChildren()* function to dynamically manipulate the document object model tree and plant malicious links into web pages.

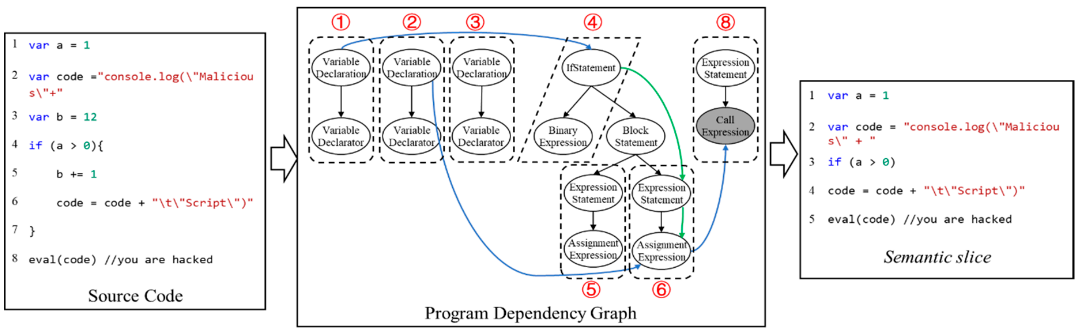
When we implement program slicing, for convenience, all these *key functions* are divided into forward functions and backward functions according to the position of the dependent statement. The forward function receives parameters from an external input, such as a file or HTTP request. For example, the *eval* function is a forward function because its arguments must come from the previous statement. The backward function is that when the program runs, it does not need to receive parameters from the external environment. Program slicing has 3 steps:

(1) Traverse the PDG of the input program to find a specific node. This node can hint that there is a *key function* in the statements, such as *CallExpression*, *FunctionDeclaration*, or *VariableDeclaraion*.

(2) Check the name attribute of the specific node found in step 1. If it is one of the key functions, traverse along the edges of the data dependency and control dependency to find all nodes that have a dependency relationship with the specific node.

(3) Collect statements corresponding to all the nodes found in step 2. Then sort these statements in the order of the input samples to generate program slices.

For ease of understanding, we leverage an example to illustrate the slicing process. The source code in Figure 4 shows an example with an *eval* function. Then, we generate the PDG (for the sake of clarity, we simplified the PDG), where the blue edges indicate data dependency, the green directed edges indicate control dependency, and the number indicates the line number of the node corresponding statement.



```
1 var a = 1
2 var code ="console.log(\"Maliciou
  s\"+"
3 var b = 12
4 if (a > 0){
5     b += 1
6     code = code + "\t\"Script\")"
7 }
8 eval(code) //you are hacked
```

Source Code

```
1 var a = 1
2 var code = "console.log(\"Maliciou
  s\" + "
3 if (a > 0)
4 code = code + "\t\"Script\")"
5 eval(code) //you are hacked
```

Semantic slice

Figure 4. Extracting program slice from a JavaScript sample.

In order to generate program slice, we first traverse the PDG of the source code and find that the gray node is a *CallExpression* node. We further found that the name attribute of this node is an *eval* function, which is one of the *key functions*. Then, we track the data flow edge and control flow edge separately, and get the sequence of nodes as *CallExpression, AssignmentExpression, VariableDeclaration, ExpressionStatement, IfStatement, VariableDeclaration*. Finally, we extract the statements corresponding to these nodes to generate program slices, which we call *semantic slice*.

4. Malicious JavaScript Detection Model

4.1. Model Selection

Although deep learning has been widely used in computer vision [36], natural language processing [37], and image processing [38], these fields are different from malicious JavaScript detection. This means that not all neural networks are suitable for malicious JavaScript detection. This is because whether or not a slice of code is malicious depends on the context, so we need to find a proper neural network which can cope with context. Therefore, we consider that the neural networks applied to natural language processing are potentially suitable for malicious JavaScript detection, as context is also crucial in this domain. However, many kinds of neural networks can be applied to natural language processing. In order to narrow the choice, we focus on the two most widely used algorithms, Convolutional Neural networks (CNNs) and Recurrent Neural Networks (RNNs). Before deciding on the most suitable model for malicious JavaScript detection, we need to analyze these two algorithms and their variants.

CNNs have outstanding performance in the fields of image classification and machine vision. But some CNN variants can be used to process natural language. Text-CNN [39] is a classic CNN variant that uses convolutional operations to process text. It is often used to solve the classification problem of long text sequences. But the convolutional operations ignore the word order of the sequence, and code semantic information might be lost in this process. Therefore, it has limitations in detecting malicious JavaScript code.

RNNs are effective in dealing with long sequence problems [40,41], and they indeed have been used for program analysis [42,43]. However, existing studies demonstrate that RNNs may suffer from vanishing gradients [44] when the input sequence is too long. This means that using RNNs to process JavaScript code may be invalid or even fail to train because the length of commercial JavaScript code is usually very large. To solve the gradient problem, Long Short-Term Memory (LSTM) and the Gated Recurrent Unit (GRU) are proposed [45,46]. LSTM is considered a special RNN because it has special units in addition to standard units. These units include a memory cell that can save information in memory for long periods of time. In order to control the flow of information in the *memory cell*, three gates are designed, i.e., input gate, output gate, and forget gate. These gates allow *memory cells* to learn for a long time. GRU adopts a similar design idea to LSTM but with a simpler structure. It also used a set of gates to control the flow of information, but they did not use separate storage units. Since GRU easily suffers from the overfitting problem, LSTM preforms better than GRU in natural language modeling. Therefore, LSTMs are selected as the basic model.

Through further analysis, we find that the parameters of the function call in the program may be affected by the statements before or after the function call. This means that even LSTM are not enough to detect malicious JavaScript. Therefore, the selected LSTM variant should be able to process the input sequence forward and backward. In the end, we chose the Bidirectional LSTM (BLSTM) neural network [47] for detecting malicious JavaScript in this paper because it can learn input information bidirectionally.

4.2. Structure of BLSTM

In this section, we introduce the structure of the BLSTM neural network for detecting malicious JavaScript (the training parameters of the neural network are discussed in Section 5.3). As shown in Figure 5, this model consists of an output layer, a dense layer, and multiple BLSTM layers. The input of the model requires a fixed-length vector, which is a vector representation corresponding to a *semantic slice*. The dense layer is used to compress the dimension of the output vector of the BLSTM layer. The output layer collects the vector from the dense layer and outputs the probability of classification.

The BLSTM layer is the most critical part of the model. It is established by stacking multiple LSTM cells. These LSTM cells are chain-structured and process input information in both forward and backward directions. Each LSTM defines and maintains a memory cell throughout the whole life cycle, which is the most important element of the LSTM. As Figure 6 shows, the previous cell state c_{t-1} interacts with the previous cell output h_{t-1} and the present input x_t to determine which elements of the internal state vector should be updated, maintained or forgotten. Three gates are defined to control the changes of the memory cell state. The forget gate f_t decides what information to throw away from the cell state, the input gate i_t decides what new information to store in the cell state, and the output gate o_t decides what to output.

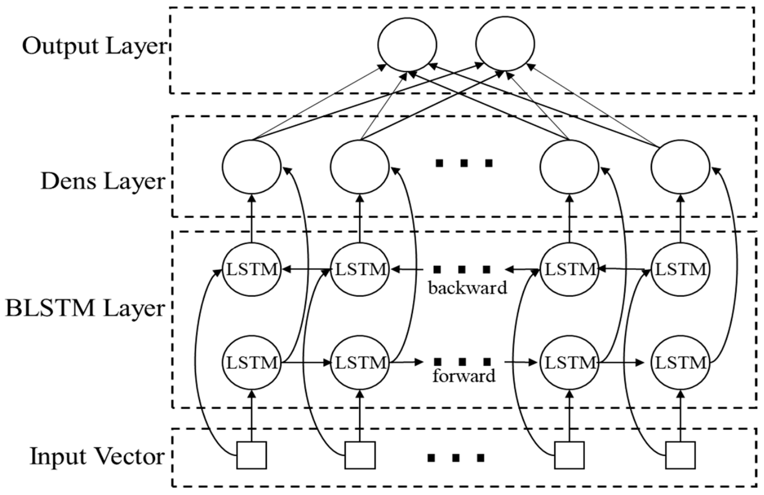


Figure 5. A brief review of the Bidirectional Long Short-Term Memory (BLSTM) neural network.

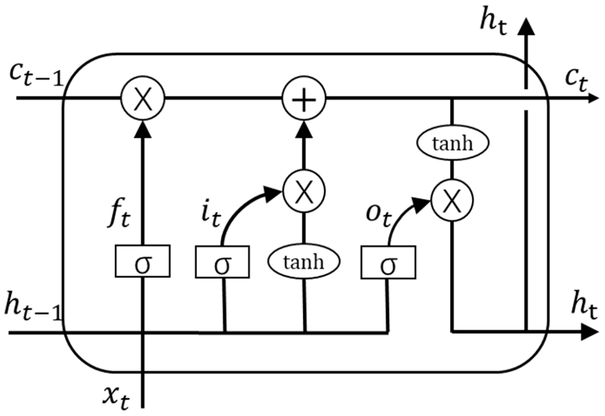


Figure 6. The structure of a typical Long Short-Term Memory (LSTM) cell. The circles represent pointwise operations, e.g., vector addition, and the rectangles are the gates. Lines merging denote concatenation, a line forking denotes its content being copied and the copies going to different locations.

The details of calculation are shown below, where σ is the sigmoid function, *tanh* is the hyperbolic tangent function, c_t is the cell state, and h_t is the cell output. W_{xi}, W_{hi}, W_{xf}, W_{hf}, W_{xc}, W_{hc}, W_{xo} and W_{ho} are weight matrixes for the corresponding input of the network activation functions. b_i, b_f, b_c, and b_o are bias vectors.

$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + b_i) \tag{1}$$

$$f_t = \sigma\left(W_{xf}x_t + W_{hf}h_{t-1} + b_f\right) \tag{2}$$

$$c_t = f_t c_{t-1} + i_t tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c) \tag{3}$$

$$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + b_o) \tag{4}$$

$$h_t = o_t tanh(c_t) \tag{5}$$

Since each BLSTM layer contain a forward LSTM layer and a backward LSTM layer, the calculation of output vector h_t is separated into forward output $\overrightarrow{h}_t$ and backward output $\overleftarrow{h}_t$. The forward output $\overrightarrow{h}_t$ is iteratively calculated using input in a positive sequence from time 0 to time T, while the backward output $\overleftarrow{h}_t$ is calculated using the reversed input from time T to time 0. Equations (6) and (7) give the details, where $W_{x\overrightarrow{h}}$, $W_{\overrightarrow{h}\overrightarrow{h}}$, $W_{x\overleftarrow{h}}$, $W_{\overleftarrow{h}\overleftarrow{h}}$ are weight matrixes and $b_{\overrightarrow{h}}$, $b_{\overleftarrow{h}}$ are bias vectors.

$$\overrightarrow{h}_t = \sigma\left(W_{x\overrightarrow{h}}x_t + W_{\overrightarrow{h}\overrightarrow{h}}\overrightarrow{h}_{t-1} + b_{\overrightarrow{h}}\right) \tag{6}$$

$$\overleftarrow{h}_t = \sigma\left(W_{x\overleftarrow{h}}x_t + W_{\overleftarrow{h}\overleftarrow{h}}\overleftarrow{h}_{t+1} + b_{\overleftarrow{h}}\right) \tag{7}$$

4.3. Transforming Semantic Slice into Vectors

The input of the BLSTM neural network must be in vector form, so we need to transform the *semantic slice* into a vector. First, we delete the blank characters and comments in the slice because they are not useful for detecting malicious JavaScript. Second, we map user-defined variables to symbolic names in a one-to-one manner, noting that the same variable name may be repeatedly defined in different blocks of code in the original program, and these variable names may map to the same symbolic name. For different variables, we use symbol and number to distinguish them, such as Identifier_1, Identifier_2. Third, user-defined functions are mapped to symbolic names in a one-to-one manner, and similar to variables, functions with the same name may be mapped to the same symbolic representation. Fourth, through lexical analysis, we divide the rest of the slice into a token sequence, including identifiers, keywords, operators and symbols. Table 3 shows the token sequence generated by the *semantic slice* shown in the Figure 4.

In order to encode each token into a vector, we choose the word2vec as an embedding tool, which is a shallow, two-layer neural network. Compare with other embedding algorithms, Word2vec has powerful architecture and fast training speed. In this paper, the input of Word2vec is a series of token sequences, and the output is a set of feature vectors representing tokens in the sequence.

BLSTM neural networks require vectors of equal length as input. However, because the number of tokens in slices may be different, the corresponding vectors also have different lengths. In order to solve this problem, the vector corresponding slice needs to be converted into a fixed length, which is represented by the parameter *len*. When the vector length is longer than *len*, we delete the part of the vector that is far from the *key function*. When the length of the vector is shorter than *len*, we pad zeros at the end closer to the key function. The parameter *len* is related to the number of nodes in the input layer in the BLSTM neural network. This parameter will affect the performance of malicious JavaScript detection.

Table 3. Tokens generated from *semantic slice.*

No	Token	Value	No	Token	Value
1	Keyword	var	13	Numeric	0
2	Identifier_1	a	14	Punctuator	)
3	Punctuator	=	15	Identifier_2	code
4	Numeric	1	16	Punctuator	=
5	Keyword	Var	17	Identifier_2	code
6	Identifier_2	code	18	Punctuator	+
7	Punctuator	=	19	String	"\t\"Script\")\""
8	String	"console.log(\"Malicious\"+"	20	Identifier_3	eval
9	Keyword	if	21	Punctuator	(
10	Punctuator	(	22	Identifier_2	code
11	Identifier_1	a	23	Punctuator	)
12	Punctuator	>	24	Numeric	0

5. Experiment and Result

In this section, several research questions on the BLSTM neural network for malicious JavaScript detection are proposed. This can help us objectively evaluate the effectiveness of the method proposed in this paper. To this end, we propose comprehensive experiments to answer these questions:

RQ1: Is the BLSTM neural network effective for malicious JavaScript detection?

RQ2: How effective is the BLSTM neural network compared to other machine learning-based detection methods?

RQ3: What is the influence of obfuscation on the BLSTM neural network and other learning-based detection methods?

5.1. Dataset Preprocessing

We collected 60380 raw benign and malicious samples with a total size of more than 3.1 GB. To prevent duplication of data from multiple data sources, we calculated the MD5 of each sample to ensure that each sample is unique.

As for malicious samples, 29893 files (Table 4) were collected from HynekPatrak [48], geeksonsecurity [49] and Wang Wei's dataset [50]. Most of these malicious samples are JavaScript files. For HTML files, we manually extracted the JavaScript code from the <script> tags and stored them as a JavaScript file. Almost all samples were obfuscated, e.g., through data obfuscation or encoding obfuscation.

Table 4. Overview of malicious JavaScript.

Source	#JS
HynekPatrak	27458
geeksonsecurity	1751
WangWei	684
Total	29893

As for benign samples, we collected JavaScript files in Alexa [51] top sites. We believe that the JavaScript files from these sites are safe, because subjectively these sites have no motive to do evil, and objectively there are a large number of security engineers to protect these sites. For each website, we first visited their homepage, then visited the same domain link on the homepage to collect dynamically generated scripts. For inline scripts in HTML, we extracted the JavaScript code and stored it as a JavaScript file in the original order. In this way, we obtained 30487 unique benign samples. In order to ensure that these JavaScript files are benign, we checked them with anti-virus software. Skolka et al. [52] indicate that more than 30% of developers obfuscate their scripts and more than 55% of third-party scripts are obfuscated. Therefore, there should also be a certain proportion of obfuscated scripts in our benign samples.

We constructed two new datasets, DB_DeOb and DB_Ob, based on the collected raw data (called DB_Raw). The dataset DB_DeOb contains only deobfuscated scripts. We combined deobfuscation tools and manual analysis to deobfuscate 1227 malicious samples and 1516 benign samples. It should be noted that there is currently no strict definition of obfuscated code and deobfuscated code, so we consider the code that meets the following two conditions to be successful in deobfuscating:

Condition 1: The variables and strings in the code are not encoded by Unicode, ASCII, etc.

Condition 2: Compared with before processing, the clarity of the processed code logic has been significantly improved.

The dataset DB_Ob consists only of obfuscated codes. In order to construct DB_Ob, we collected 9892 benign samples and 9721 malicious samples from DB_Raw with conspicuous feature of obfuscation. Table 5 gives the details of these datasets.

Table 5. The datasets for BLSTM neural network.

Name	Type	#JS
DB_Raw	benign	30487
	malicious	29893
DB_Ob	benign	9892
	malicious	9721
DB_DeOb	benign	1516
	malicious	1227

5.2. Measurement Metrics

We leverage accuracy, precision, recall, and F1-score to evaluate the effectiveness of BLSTM neural network in detecting malicious JavaScript. In general, the effectiveness of a model is evaluated by the accuracy rate, i.e.,

$$accuracy = \frac{N_{true}}{N_{totle}} \tag{8}$$

where N_{true} represents the number of samples that predict correctly, and N_{totle} represents the total number of samples in the dataset. However, it is not enough to evaluate the performance of the model based on accuracy alone, so we choose precision and recall as supplements. The precision refers to the ratio of true positive malicious JavaScript to the total number of samples detected as malicious, i.e.,

$$precision = \frac{TP}{TP + FP} \tag{9}$$

where TP is the number of samples with malware JavaScript detected correctly, and FP is the number of samples with false malicious JavaScript detected. The recall measures the ratio of true positive malware JavaScript to the entire population of malicious JavaScript samples, i.e.,

$$recall = \frac{TP}{TP + FN} \tag{10}$$

where FN is the number of samples with true malicious JavaScript undetected. F1-score takes into account both precision and recall, that is:

$$F1 - score = \frac{2 \times precision \times recall}{precision + recall} \tag{11}$$

Ideally, our detector should not miss any malicious code and not trigger false alarms, that is, it both has high precision and recall at the same time, but these two indicators are mutually exclusive in some scenarios. For example, if only one sample is taken, and the sample is also a positive sample, then the precision = 1.0, and recall may be lower (because there may be multiple samples in the dataset). As a comprehensive index, the F1-score is to evaluate a classifier comprehensively in order to balance the influence of precision and recall.

5.3. Learning the BLSTM Neural Network

In this part, we introduce the hyper-parameter selection of the BLSTM neural network. The BLSTM neural network is implemented in Python with TensorFlow 2.0 and Keras. We experiment on a workstation with NVIDIA Quadro P4000 and Intel Xeon E5-2630 CPU operating at 2.50GHz. Since malicious JavaScript detection is a binary classification task, we use cross-entropy as the loss function, i.e.,

$$loss = -ylog\hat{y} + (1 - y)log(1 - \hat{y}) \tag{12}$$

where y (0 for benign and 1 for malicious) is the label of the sample and $\hat{y}$ is the output probability of the sample being malicious. We adopt Adam [53] as an optimizer, which is one of the most efficient optimizers at present. Then, we experiment with different combinations of hyper-parameters and compare the accuracy by 10 cross-validation. We will fix others when testing the influence of a hyper-parameter.

(1) Determined Input Vector Dimensions: First, we test the influence of different input vector dimensions. We fixed the number of BLSTM layers to 2 and the epoch to 10. Table 6 shows how the model's performance is affected by the different input vector dimensions. As we can see, the accuracy increases obviously when the dimension increases from 60 to 80, the bug gets lower when the dimension is set to 100. We also tested combinations of bigger dimensions and smaller dimensions, but their performances are poorer. This is because too big a dimension gives the sample pad too many zeros, which is useless for detecting malicious JavaScript, and too small a dimension makes the samples lose too many program features, which reduces the performance of the model. Therefore, the input vector dimension is set to 80.

Table 6. The influence of the input vector dimensions.

Vector Dimensions	Accuracy
60	78.49%
70	80.68%
80	**92.86%**
90	89.13%
100	87.57%

(2) Determined the number of epochs: Then we fixed the input vector dimension to 80 and test the influence of the number epoch. An epoch refers to one cycle through the full training dataset. An appropriate epoch can improve the performance of the model, but too large an epoch will only waste training time. As shown in Figure 7, when the number of epochs is from 1 to 6, the accuracy of the model continues to increase. When the number of epochs is greater than 6, the accuracy rate hardly changes. This means that when the number of epochs is 6, we can get better performance and less training time at the same time. Therefore, this hyper- parameter is set to 6.

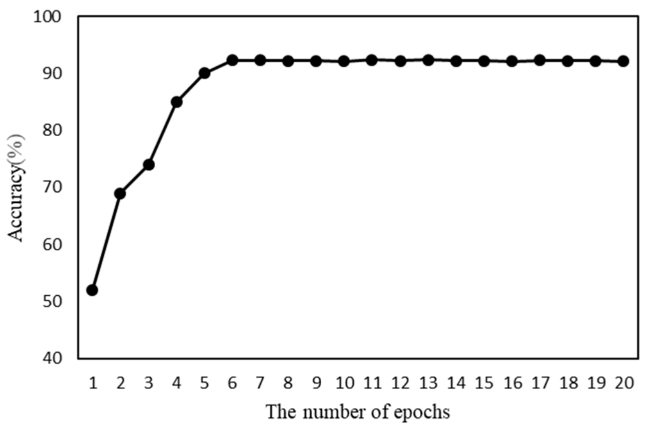


Figure 7. The influence of the number of epochs.

(3) *Determined the Number of BLSTM Layers:* After setting the number of epochs to 10, we test the influence of the number of the BLSTM layers. As Figure 8 shows, when the number of BLSTM

layers is from 1 to 3, the accuracy of the model continues to increase. When the number of BLSTM layers is greater than 3, the performance of the model shows a downward trend. Therefore, the number of the BLSTM layers is set to 3.

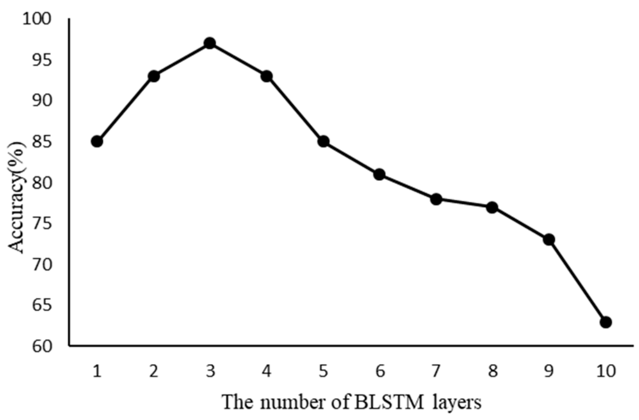


Figure 8. The influence of the number of BLSTM layers.

5.4. Runtime Performance

In order to comprehensively evaluate the performance of BLSTM neural network, we gather statistics on the time consumption in each of the stages. We randomly select 2000 unique samples, half of which are benign and half of which are malicious and that recorded the time consumption. As for abstract code representation, Table 7 shows the minimum, maximum, average, and median time consumption at each stage, for one file. As we can see, deobfuscation takes the least amount of time, and a file takes only 18 ms on average. The generation of AST is divided into two steps, one is to parse JavaScript files with Esprima [33], the other is to construct the AST, which makes the process take a little longer, with an average of 96 ms per file. The more time-consuming stage is to generate the PDG, which largely depends on the size of the AST, because we have to traverse it several times to add data dependencies. The generation of *semantic slices* takes the most time, with an average of 233 ms, because this process requires traversing the PDG and finding the corresponding source code statements at the same time. In general, the more complex code representation causes higher overhead, but this also makes the code abstract representation preserve more semantic information.

Table 7. Time consumption to generate abstract code representations.

Stage	**Min(ms)**	**Max(s)**	**Mean(ms)**	**Median(ms)**
deobfuscation	0.012	0.761	18.214	9.001
AST	0.138	2.414	96.231	20.587
CFG	0.013	1.211	37.091	4.635
PDG	0.142	12.261	210.258	9.621
semantic slice	0.271	13.957	232.712	11.243

Since the input vector dimensions of BLSTM neural networks are fixed, the time consumption per sample is similar. Table 8 shows the average time of each stage of learning the BLSTM neural network, for one file. As we can see, in the learning phase, the model spends an average of 0.241 ms processing a file, and in the classification phase, the model spends an average of 2.765 ms process a file.

Table 8. The time consumption to learn the BLSTM neural network.

Stage	Time Consumption (ms)
Learner	0.241
Classifier	2.763

For one JavaScript sample, the detection time is approximately the sum of the average time for generating the abstract code representation (from deobfuscation to *semantic slice* generation) and the average time for the classifier. Based on the data in Tables 7 and 8, the detection time is 595 ms. Considering that the average size of JavaScript samples is 32KB, we think this overhead is reasonable.

5.5. *Detection Performance*

Answer to RQ1: As shown in Table 9, the BLSTM neural network achieves a good performance. The accuracy of the BLSTM neural network reached 97.71%, which indicates that most malicious samples can be accurately found during the test. At the same time, the model also achieved a recall rate of 97.91%, which means that it caused very few false positives. The F1-score is the harmonic mean of the precision and recall, where an F1 score reaches its best value at 1. It is one of the most important measurements for evaluating a model's performance. The F1-score of the BLSTM neural network reached a very high value of 98.29%, which means our model achieves higher rates of true positives with lower rates of false positives.

Table 9. The performance of the models on DB_DeOb.

Methods	Accuracy	Precision	Recall	F1-Score
BLSTM	**97.71%**	**98.68%**	**97.91%**	**98.29%**
Naive Bayes	64.61%	67.91%	95.31%	79.31%
SVM	81.62%	93.71%	72.39%	81.68%
Random forest	90.01%	92.83%	89.53%	91.15%
JaSt	93.00%	96.14%	92.03%	94.04%
ClamAV	84.21%	95.01%	68.30%	79.47%

Therefore, we conclude that the BLSTM neural network is extremely effective in detecting malicious JavaScript.

Answer to RQ2: In order to compare with other malicious code detection approaches, we chose three widely used machine learning models, a malicious JavaScript detection tool and an antivirus software. As shown in Table 9, although the Naive Bayes algorithm achieves a relatively high recall rate, its accuracy is very low, only 64.61%. This means that a large number of benign samples are misclassified. If the Naive Bayes model is used to detect malicious JavaScript on the Internet, then much manpower must be invested to manually audit the detection results of the model to reduce the impact on normal websites. Contrary to the Naive Bayes model, the accuracy rate of the Support Vector Machine (SVM) model is higher than the recall rate, and the F1-score of 81.68% seems to perform well. However, we noticed that the recall rate of the SVM model is only 72.39%, which means that 27.61% of the malicious JavaScript evade the detection of the model. Therefore, such a model is not capable of detecting malicious JavaScript. We consider that the reason for the poor performance of these two models is that the models are too simple to fully describe the malicious features. The performance of random forest is much better than the previous two models, but it is still worse than the BLSTM neural network with 7.7% lower accuracy, 5.88% lower recall rate, and 7.17% lower F1-Score. We need to emphasize that malicious JavaScript detection is different from other neural network applications, such as speech recognition and emotion classification. Undetected malicious JavaScript may cause huge losses to Internet users, such as the stealing of credit card information. Therefore, any increase in accuracy will protect more Internet users from attackers.

JaSt was proposed by Fass et al. [13], which extracts features from the AST and uses the frequency analysis of specific patterns to detect malicious JavaScript. Since JsSt is open source, we use it to compare with our method. Table 9 shows the experimental results, the perform of the BLSTM neural network is better than JaSt with 4.71% higher accuracy, 5.88% higher recall rate, and 4.25% higher F1-score. This means that the program slices can extract more semantic information than simply traversing the AST, and the increase in semantic information improves the performance. The results in Table 9 are all experimentally obtained on the deobfuscated dataset. But we noticed that the authors of JaSt do not process the samples collected from the Internet, but directly use them to train the model. In order to demonstrate the robustness of our model, we use the same data processing method as JaSt to experiment again. The experimental results are shown in Table 10. The accuracy rate of the BLSTM neural network is 2.26% higher than that of JaSt, the recall rate is 2.99% higher, and the F1-score is 1.08% higher. This means that even with unprocessed samples to train the model, the BLSTM neural network still performs better than JaSt.

Table 10. The performance of the models on DB_Raw.

Methods	Accuracy	Precision	Recall	F1-Score
BLSTM	**94.51%**	**95.67%**	**94.74%**	**95.20%**
JaSt	92.25%	96.61%	91.75%	94.12%

In addition, in order to demonstrate that the performance of the learning-based method is better than the pattern-based method, we chose a traditional antivirus software for comparison. ClamAV [54] is a famous open source antivirus software used in a variety of situations, including email scanning, web scanning, and endpoint security. We used the latest stable release version (0.102.2) of ClamAV to compare with our method. As shown in Table 9, ClamAV has an accuracy of 84.21%, which is significantly lower than that of the BLSTM neural network and even lower than that of JaSt and random forest. ClamAV's 68.3% recall rate means that 31.7% of malicious samples have not been found, which is lower than our expectations for antivirus software. Upon further analysis, we found that the precision of ClamAV reached 95.01%, combined with a low recall rate, which shows that ClamAV has very low false positives. This means that ClamAV may tend to classify the samples as benign to avoid the bad experience that false positives bring to users. In summary, learning-based detection methods perform better than pattern-based detection methods. This is because the learning-based method can capture high-level semantic information and describe attack vectors from hundreds of dimensions, while the pattern-based method can only define rules to describe syntactic-level features. Therefore, we conclude that the BLSTM neural network performs better than other methods in detecting malicious JavaScript.

Answer to RQ3: Many existing studies assume that neural networks can extract high-level semantic features of programs, so obfuscation techniques have no influence on neural network-based detectors. But they do not prove it experimentally. In order to study the influence of obfuscation techniques on the BLSTM neural network and other machine learning models, we experiment again on obfuscated dataset. As shown in Tables 9 and 11, the performance of the BLSTM neural network decreases significantly on the obfuscation dataset, the accuracy, decrease from 97.71% to 92.01%, the recall rate decrease from 97.91% to 93.03%, and the F1-score decrease from 98.29% to 93.63%. These measurements decrease by an average of 5.08%. This means that the obfuscation techniques have a relatively significant negative impact on the BLSTM neural network. For the Naive Bayes model, SVM model, and random forest model, the accuracy decreases by an average of 2.32% and F1-socre decreases by an average of 2.4%. This means that deep learning models seem to be more sensitive to obfuscation than traditional machine learning models. This may be due to the obfuscation techniques make the semantic information in the *semantic slices* incorrect. These errors continue to accumulate between different layers of the neural network, which ultimately leads to a decline in the performance of the BLSTM neural network.

Table 11. The performance of the models on DB_Ob.

Methods	Accuracy	Precision	Recall	F1-Score
BLSTM	**92.01%**	**94.23%**	**93.03%**	**93.63%**
Naive Bayes	62.73%	65.32%	93.16%	76.79%
SVM	80.22%	90.82%	71.48%	80.00%
Random forest	86.32%	89.12%	87.20%	88.15%
JaSt	92.14%	95.91%	92.45%	94.15%

For JaSt, obfuscation has little effect on it. The reason for this may be the choice of program features. JaSt extracts syntax units from JavaScript programs, which is a syntactic level of program features, and then uses them to train the model. This feature extraction method also captures the syntax changes caused by obfuscation. In other words, JaSt considers that obfuscation is one of the characteristics of malicious code. The author of JaSt also confirmed this in the paper. However, this features extracting method has obvious limitations. On the one hand, benign obfuscation is widely used to protect the intellectual property rights of program developers. Using obfuscation as a feature of malicious code leads to false positives. On the other hand, the syntactic level program features have limited expression power to malicious code. Experienced attackers can still evade the detection of JaSt by adjusting the syntax structure, which makes it difficult to improve JaSt's performance.

In summary, the assumptions about the effects of obfuscation in existing studies are not entirely correct. The impact of obfuscation is related to the abstract representation of the program and the choice of model. Semantic level program features are more sensitive to obfuscation than syntactic level program features. Deep learning models are more sensitive to obfuscation than traditional machine learning models.

6. Limitations

The abstract code representation is based on a static analysis of JavaScript to generate both the control and data flow of an input program. Therefore, a complete code coverage based on code analysis is constructed. In turn, it is subject to the defects brought about by the dynamics of JavaScript. Specifically, JavaScript can generate code at runtime, for example, the *eval* function interprets a dynamically constructed string as a program fragment and executes it in the current scope. If all the code is generated dynamically (we only encounter it in conditional compilation), static analysis cannot extract the correct program features. However, this does not mean static analysis is useless. Static analysis requires fewer computing resources and has fast detection speed. With a reasonable design, static analysis can also achieve high performance in most situations.

In addition, all learning-based malicious JavaScript detectors will fail to detect some attacks, such as malicious scripts not containing any features in the current training dataset. In this paper, the malicious JavaScript samples come from the Internet from 2015 to 2017. These samples cover most of the JavaScript-based attacks. But the dataset still lacks samples of new attacks appeared after 2017, which makes our model unable to detect these new attacks. We plan to continually add new malicious JavaScript samples in the follow-up study to improve the performance of the model.

7. Conclusion

With the widespread use of JavaScript on the Internet, a large number of JavaScript-based attacks have been developed, such as XSS and drive-by-download attack. This poses a serious threat to Internet users. At present, the detection method based on expert knowledge cannot alleviate the growth rate of malicious code on the Internet. Fortunately, the latest studies demonstrate that the deep learning model shows encouraging results in detecting malicious scripts.

Based on the existing research, we proposed a novel malicious JavaScript detection approach based on BSLTM neural network. In order to transform JavaScript programs to vectors, we proposed the concept of *semantic slices*, which preserve rich semantic information and are easy to transform into vectors. Then we trained the BLSTM neural network to classify JavaScript samples. Experimental results show that our model achieves the best performance compared with other four machine learning-based models and a traditional antivirus software, with 97.71% accuracy and 98.29% F1-score. At the same time, we preliminarily studied the influence of obfuscation on the performance of the learning-based detector, and corrected the relevant assumptions of the existing research.

Future work will be carried out from two aspects. On the one hand, static analysis cannot detect the malicious JavaScript code generated dynamically at present. We hope to alleviate the negative impact of JavaScript dynamics through more sophisticated string processing. On the other hand, researchers have proposed many new models, such as tree structure neural networks and graph structure neural networks. We hope to study the effectiveness of more types of neural networks in malicious JavaScript detection.

Author Contributions: Conceptualization, X.S. and C.C.; Methodology, X.S.; Software, X.S.; Validation, X.S., C.C. and B.C.; Data Curation, C.C.; Writing—Original Draft Preparation, X.S.; Writing—Review and Editing, J.F.; Visualization, J.F.; Supervision, B.C.; Project Administration, B.C.; Funding Acquisition, B.C. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by Beijing Natural Science Foundation(4204107), the Fundamental Research Funds for the Central Universities (No. 500419810), Funds of "YinLing" and in part by the China Postdoctoral Science Foundation under Grant 2019M650020.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. JavaScript. Available online: https://developer.mozilla.org/en-US/docs/Web/JavaScript (accessed on 29 December 2019).
2. Bichhawat, A.; Rajani, V.; Garg, D.; Hammer, C. Information flow control in WebKit's JavaScript bytecode. In Proceedings of the International Conference on Principles of Security and Trust, Grenoble, France, 5–13 April 2014.
3. Zhou, Y.; Evans, D. Understanding and monitoring embedded web scripts. In Proceedings of the IEEE Symposium on Security and Privacy, San Jose, CA, USA, 17–21 May 2015.
4. Symantec Security Center. Available online: https://www.symantec.com/security-center/threat-report (accessed on 19 November 2019).
5. Tanaka, Y.; Kashima, S. SeedsMiner: Accurate URL Blacklist-Generation Based on Efficient OSINT Seed Collection. In Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence, Thessaloniki, Greece, 14–17 October 2019.
6. Egele, M.; Wurzinger, P.; Kruegel, C.; Kirda, E. Defending Browsers against Drive-by Downloads: Mitigating Heap-Spraying Code Injection Attacks. In Proceedings of the Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Como, Italy, 9–10 July 2009.
7. Sachin, V.; Chiplunkar, N.N. SurfGuard JavaScript instrumentation-based defense against Drive-by downloads. In Proceedings of the International Conference on Recent Advances in Computing and Software Systems, Chennai, India, 25–27 April 2012.
8. Hallaraker, O.; Vigna, G. Detecting malicious JavaScript code in Mozilla. In Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems, Shanghai, China, 16–20 June 2005.
9. Xu, W.; Zhang, F.; Zhu, S. Jstill: Mostly static detection of obfuscated malicious JavaScript code. In Proceedings of the Malicious JavaScript Detection using Statistical Language Model, San Antonio, TX, USA, 18–20 February 2013.
10. Fang, Y.; Huang, C.; Liu, L.; Xue, M. Research on malicious JavaScript detection technology based on LSTM. *IEEE Access* **2018**, *6*, 59118–59125. [CrossRef]

11. Stokes, J.W.; Agrawal, R.; McDonald, G.; Hausknecht, M. ScriptNet: Neural Static Analysis for Malicious JavaScript Detection. *arXiv* **2019**, arXiv:1904.01126.
12. Hao, Y.; Liang, H.; Zhang, D.; Zhao, Q.; Cui, B. JavaScript Malicious Codes Analysis Based on Naive Bayes Classification. In Proceedings of the International Conference on P2P, Parallel, Grid, Cloud and Internet Computing, Guangdong, China, 8–10 November 2014.
13. Fass, A.; Krawczyk, R.P.; Backes, M.; Stock, B. JaSt: Fully Syntactic Detection of Malicious (Obfuscated) JavaScript. In Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Saclay, France, 28–29 June 2018.
14. Liang, H.; Yang, Y.; Sun, L.; Jiang, L. JSAC: A Novel Framework to Detect Malicious JavaScript via CNNs over AST and CFG. In Proceedings of the 2019 International Joint Conference on Neural Networks, Budapest, Hungary, 14–19 July 2019.
15. Cova, M.; Kruegel, C.; Vigna, G. Detection and analysis of driveby-download attacks and malicious javascript code. In Proceedings of the 19th International Conference on World Wide Web, Raleigh, NC, USA, 26–30 April 2010.
16. Kyungtae, K.; Kim, I.L.; Kim, C.H.; Kwon, Y.; Zheng, Y.; Zhang, X.; Xu, D. J-Force: Forced Execution on JavaScript. In Proceedings of the 26th international conference on World Wide Web, Perth, Australia, 3–7 April 2017.
17. Jayasinghe, G.K.; Culpepper, J.S.; Bertok, P. Efficient and effective realtime prediction of drive-by download attacks. *Network Computer Appl.* **2014**, *38*, 135–149. [CrossRef]
18. Mao, J.; Bian, J.; Bai, G.; Wang, R.; Chen, Y.; Xiao, Y.; Liang, Z. Detecting Malicious Behaviors in JavaScript Applications. *IEEE Access* **2018**, *6*, 12284–12294. [CrossRef]
19. Canali, D.; Cova, M.; Vigna, G.; Kruegel, C. Prophiler: A Fast Filter for the Large-scale Detection of Malicious Web Pages. In Proceedings of the International Conference on World Wide Web, Hyderabad, India, 28 March–1 April 2011.
20. Curtsinger, C.; Livshits, B.; Zorn, B.G.; Seifert, C. ZOZZLE: Fast and precise in-browser JavaScript malware detection. In Proceedings of the 20th USENIX Security Symposium, San Francisco, CA, USA, 8–12 August 2011.
21. Likarish, P.; Jung, E.; Jo, I. Obfuscated malicious JavaScript detection using classification techniques. In Proceedings of the Proceedings of the Third ACM conference on Data and Application Security and Privacy, Quebec, AB, Canada, 13–14 October 2009.
22. Laskov, P.; Šrndić, N. Static Detection of Malicious JavaScript-Bearing PDF Documents. In Proceedings of the 27th Annual Computer Security Applications Conference, Orlando, FL, USA, 5–9 December 2011.
23. Stock, B.; Livshits, B.; Zorn, B. Kizzle: A SignatureCompiler for Detecting Exploit Kits. In Proceedings of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Toulouse, France, 28 June–1 July 2016.
24. Kar, D.; Panigrahi, S.; Sundararajan, S. 2016. SQLiGoT: Detecting SQL injection attacks using graph of tokens and SVM. *Computer Secur.* **2016**, *60*, 206–225. [CrossRef]
25. Kapravelos, A.; Shoshitaishvili, Y.; Cova, M.; Kruegel, C.; Vigna, G. Revolver: An Automated Approach to the Detection. In Proceedings of the 22nd USENIX Conference on Security, Berkeley, CA, USA, 14–16 August 2013.
26. He, X.; Xu, L.; Cha, C. Malicious JavaScript Code Detection Based on Hybrid Analysis. In Proceedings of the 25th Asia-Pacific Software Engineering Conference, Nara, Japan, 4 December 2018.
27. Ndichu, S.; Ozawa, S.; Misu, T.; Okada, K. A machine learning approach to malicious JavaScript detection using fixed length vector representation. In Proceedings of the 2018 International Joint Conference on Neural Networks, Rio de Janeiro, Brazil, 8–13 July 2018.
28. Fass, A.; Backes, M.; Stock, B. JStap: A static pre-filter for malicious JavaScript detection. In Proceedings of the 35th Annual Computer Security Applications Conference, San Juan, PR, USA, 9–13 December 2019.
29. Fass, A.; Backes, M.; Stock, B. Hidenoseek: Camouflaging malicious javascript in benign ASTs. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, London, UK, 11–15 November 2019.
30. Howard, F. Malware with your Mocha? Obfuscation and antiemulation tricks in malicious JavaScript. *Sophos Tech. Papers* **2010**, *14*, 1–18.
31. Xu, W.; Zhang, F.; Zhu, S. The power of obfuscation techniques in malicious JavaScript code: A measurement study. In Proceedings of the 7th International Conference on Malicious and Unwanted Software, Fajardo, PR, USA, 16–18 October 2012.
32. JSDetox-A Javascript Malware Analysis Tool Using Static Analysis. Available online: http://www.relentless-coding.org/projects/jsdetox (accessed on 19 November 2019).

33. ECMAScript Parsing Infrastructure for Multipurpose Analysis. Available online: https://esprima.org/index.html (accessed on 19 November 2019).
34. Rieck, K.; Krueger, T.; Dewald, A. Cujo: Efficient detection and prevention of drive-by-download attacks. In Proceedings of the 26th Annual Computer Security Applications Conference, Austin, TX, USA, 6–10 December 2010.
35. Xu, Q. Research on the Methods for Javascript Malicious Code Detection. Master's Thesis, Southwest Jiaotong University, Chengdu, China, 2014.
36. Chen, Y.R.; Chao, K.; Kim, M.S. Machine vision technology for agricultural applications. *Comput. Electron. Agric.* **2002**, *36*, 173–191. [CrossRef]
37. Hinton, G.; Deng, L.; Yu, D.; Dahl, G.E.; Mohamed, A.R.; Jaitly, N.; Kingsbury, B. Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal Process. Mag.* **2012**, *29*, 82–97. [CrossRef]
38. Krizhevsky, A.; Sutskever, I.; Hinton, G.E. Imagenet classification with deep convolutional neural networks. In Proceedings of the Advances in Neural Information Processing Systems, Lake Tahoe, NV, USA, 3–6 December 2012.
39. Gibert, L.D. Convolutional Neural Networks for Malware Classification. Master's Thesis, Universitat Politècnica de Catalunya, Campus Nord, Carrer de Jordi Girona, 2016.
40. Mikolov, T.; Karafiát, M.; Burget, L.; Černocký, J.; Khudanpur, S. Recurrent neural network based language model. In Proceedings of the 11th annual Conference of the International Speech Communication Association, Makuhari, Chiba, Japan, 26–30 September 2010.
41. Cho, K.; Van Merriënboer, B.; Gulcehre, C.; Bahdanau, D.; Bougares, F.; Schwenk, H.; Bengio, Y. Learning phrase representations using RNN Encoder-Decoder for statistical machine translation. In Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, Doha, Qatar, 25–29 October 2014.
42. Li, Z.; Zou, D.; Xu, S.; Ou, X.; Jin, H.; Wang, S.; Zhong, Y. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. *arXiv* **2018**, arXiv:1801.01681.
43. Guo, N.; Li, X.; Yin, H.; Gao, Y. VulHunter: An Automated Vulnerability Detection System Based on Deep Learning and Bytecode. In Proceedings of the International Conference on Information and Communications Security, Beijing, China, 15–17 December 2019.
44. Kolen, J.F.; Kremer, S.C. Gradient flow in recurrent nets: The difficulty of learning long-term dependencies. *Field Guide Dyn. Recurr. Netw.* **2001**, 237–243. [CrossRef]
45. Cho, K.; Van Merriënboer, B.; Bahdanau, D.; Bengio, Y. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv* **2014**, arXiv:1409.1259.
46. Hochreiter, S.; Schmidhuber, J. Long short-term memory. *Neural Comput.* **1997**, *9*, 1735–1780. [CrossRef] [PubMed]
47. Graves, A.; Schmidhuber, J. Framewise phoneme classification with bidirectional LSTM and other neural network architectures. *Neural Netw.* **2005**, *18*, 602–610. [CrossRef] [PubMed]
48. Hynek Petrak. Available online: https://github.com/HynekPetrak/javascript-malware-collection (accessed on 19 November 2019).
49. Geeksonsecurity. Available online: https://github.com/geeksonsecurity/js-malicious-dataset (accessed on 19 November 2019).
50. Wang Wei's Home Page. Available online: http://infosec.bjtu.edu.cn/wangwei/?page_id=85 (accessed on 19 November 2019).
51. Alexa. Available online: https://www.alexa.com/ (accessed on 19 November 2019).
52. Skolka, P.; Staicu, C.A.; Pradel, M. Anything to Hide? Studying Minified and Obfuscated Code in the Web. In Proceedings of the World Wide Web Conference, San Francisco, CA, USA, 13–17 May 2019.
53. Kingma, D.; Ba, J. Adam: A method for stochastic optimization. *arXiv* **2014**, arXiv:1412.6980.
54. ClamAV. Available online: https://www.clamav.net/ (accessed on 21 February 2020).