Hindawi
Security and Communication Networks
Volume 2022, Article ID 4315829, 11 pages
https://doi.org/10.1155/2022/4315829

Research Article

BERT-Embedding-Based JSP Webshell Detection on Bytecode Level Using XGBoost

Ao Pu,[1] Xia Feng,[2] Yuhan Zhang,[1] Xuelin Wan,[3] Jiaxuan Han[1] and Cheng Huang[1]

[1]*School of Cyber Science and Engineering, Sichuan University, Chengdu, China*
[2]*College of Mathematics, Sichuan University, Chengdu, China*
[3]*China Merchants Bank, Shenzhen, China*

Correspondence should be addressed to Cheng Huang; opcodesec@gmail.com

Received 6 April 2022; Accepted 23 July 2022; Published 31 August 2022

Academic Editor: Shudong Li

Webshell is a malicious program that might result in data theft, file modification, or other damaging behaviors once uploaded to a server. Detecting webshells is a key security concern for website administrators. In recent years, techniques such as obfuscation and encryption have been deployed on webshell technology, and classic detection approaches such as static feature matching are gradually underperforming on webshell detection. Meanwhile, there are variations between languages such as JSP and PHP, and researchers have proposed webshell detection methods primarily for languages such as PHP. At the same time, there are fewer detection techniques for JSP webshells. In this case, a detection approach for the JSP webshells is needed. This paper provides a novel webshell detection model for the JSP language. The model's fundamental premise is that it introduces the BERT-based word vector extraction method, which has been shown in experiments to be more effective at detecting obfuscation, encryption, and other means of evading detection than the traditional Word2vec word vector extraction method. Meanwhile, we introduce the XGBoost algorithm as the model classifier. The experimental results reveal that present model has achieved 99.14% accuracy, 98.68% precision, 98.03% recall, and 98.35% f1 score, and the overall effect is better than the already existing JSP webshell detection approaches.

1. Introduction

Numerous web applications have brightened people's lives in recent years as a result of the ongoing growth of web technology. Nonetheless, cyberattacks have the potential to harm the Internet's environment and jeopardize societal security. As a result, preventing web attacks is a critical issue that we must address immediately. Among the numerous web attacks, webshell, a type of backdoor application for websites, is a frequently used attack method that has a significant detrimental effect. According to the report released by CNCERT/CC in 2022 [1], the number of webshell-attacked websites is still on the rise, and even some government websites have been attacked. After deploying a webshell on a web server, hackers can read, alter, delete, and even directly execute system instructions on the web server in order to carry out the next step of the attack.

At this stage, hackers frequently employ obfuscation techniques to avoid webshell detection, granting webshells a measure of invisibility. As a result, we are developing new ways for successfully detecting concealed webshells. The majority of susceptible websites at this level are those that have been online for an extended period of time and heavily utilize PHP and JSP. Current research on PHP-type webshell attack [2–7] is enough, but research on JSP-type webshell detection is much lower than that on PHP-type webshell detection, so we need to propose a detection technique for JSP-type webshell. At the moment, webshell research is concentrated on static and dynamic detection. Static detection is typically accomplished by extracting static information from the file, such as the longest string, which may not be as successful as expected due to the JSP language's specificities and the usage of encryption and other measures. Dynamic detection is typically dependent on network traffic;

however it consumes a large amount of system resources and is therefore inefficient. We offer a detection approach tailored to the JSP webshell that identifies the bytecode extracted from the original JSP file following a series of transformations and for the first time applies BERT to the word vector extraction of JSP bytecode, resulting in good results. The main work of this paper is as follows.

(1) We extract features from JSP compiled bytecodes using a BERT-based word embedding approach and then utilize the machine learning XGBoost method to detect JSP webshells. According to the experimental results, this method performs pretty well, and its detection effect is superior to that of other JSP webshell detection methods now available.

(2) The detection results of the widely used Word2vec word vector extraction approach are compared to those of the BERT-based word vector extraction method. The results indicate that the BERT-based word vector extraction method outperforms the Word2vec extraction method, with an accuracy of 99.14%.

(3) We obtained 2073 benign JSP samples and 830 JSP webshell samples from the Internet, including several obfuscated and encrypted JSP webshells. Additionally, we produced some JSP webshells artificially to augment the real dataset, increase the number of data samples, and boost the detection effect.

(4) We analyze and detect JSP files on the bytecode level to effectively identify obfuscated and encrypted webshells.

Next, we present the related work in Section 2. Section 3 details the whole framework of our model and the main methods used. Section 4 presents our experimental procedure and the results of the experiments. Finally, Section 5 concludes the research of this paper.

2. Background and Related Work

2.1. Outline. Webshell is a typical backdoor program left behind after a web server is compromised, usually in the form of a dynamic web script that is a command execution environment. Concealment and flexibility are the characteristics of webshells, so users may be unaware of hackers using webshells to control the system steadily over time. More importantly, a minimal study has been conducted on JSP webshells.

Zhang et al. [8] proposed a research method for JSP webshells. They used TF-IDF to count the importance of JSP compiled bytecode and used XGBoost to classify them. Their method utilizes statistical features without considering contextual information between texts, so detection is ineffective. Liu et al. [9] directly divided the original JSP code, extracted word vectors using Word2vec, and then used a bidirectional GRU with attention for classification. While some benign JSP code now employs antitheft methods like obfuscation and encryption, this method of recognizing the original JSP code directly has limited efficiency in detecting JSP webshells that also employ obfuscation and encryption. Additionally, we employ a more advanced method of word vector extraction than Word2vec, namely, BERT word embedding.

Since JSP and PHP webshells have distinct properties, we cannot simply apply the PHP webshell detection approach to JSP for the following reasons.

(1) The opcode of the PHP files and the bytecode of the JSP files are compiled differently. Typically, the PHP files' opcode is compiled using the Zend engine, whereas the JSP files' bytecode is compiled using Tomcat.

(2) Compared to standard PHP code samples, PHP webshells often contain shorter text. However, the JSP webshells have the same amount of text as the standard sample but a significantly greater number of words. It indicates that most webshell files of JSP type include malicious code with several functions and complexities.

(3) There are ready libraries to extract opcode for PHP, and many freely available PHP webshell datasets on the Internet exist. Webshells written in the JSP language lack this convenience.

As a result, the next discussion will concentrate on webshell identification for popular programming languages such as PHP.

2.2. Classification of Webshell. There are various ways to classify webshells. According to the programming language used, they can be classified as PHP, JSP, ASP webshells, etc. According to the size of the webshell, they can be classified as big Trojans, small Trojans, and one-word Trojans. Among them, the big Trojan not only connects to the database but also has some graphical operability interface. The code presentation of webshells can be divided into noncoded webshells written directly in ordinary files, coded webshells with encoding such as base64 or some obfuscated encoding, and fileless webshells that launch attacks by directly using dangerous command execution functions in web application code.

2.3. Method of Webshell Detection. In general, webshell detection can be classified into two categories: static detection and dynamic detection.

Dynamic detection is usually based on the detection of web traffic [10]. For instance, Wu et al. [11] employ reinforcement learning to determine whether the web traffic originates from a webshell. However, while dynamic detection can give real-time detection, it also uses a significant amount of system resources. As a result, it is difficult to implement in real applications.

The research is generally based on text features and some static features for static detection. Information entropy, longest string, compression rate, etc. are included in static features. Due of the inherent limits of static features, their

detection methods are frequently integrated with text information [12–14]. In addition, some research approaches for PHP language [15–17] focus on using machine learning methods to classify the opcode obtained after processing the PHP language directly. Yong et al. [18] transform PHP language into opcodes and use statistical methods such as TF-IDF and bag-of-words model to extract features and use deep learning models for classification. Wu et al. [19] perform the detection of logs on the server. Since webshell programs run with different network traffic than normal, webshells can be accurately detected.

Fass et al. [20] build abstract syntax trees from code to extract features. To detect malicious code in Javascript, Ndichu et al. [21] use abstract syntax trees to extract features and exploit the contextual information in the text. Abstract syntax trees [22–24] is an abstract representation of the source code structure to represent the syntactic structure of the programming language in a tree-like structure, which is good at extracting textual information and has many applications in the Javascript language.

Machine learning and deep learning methods [25–27] are used in webshell detection. Among the deep learning methods used, CNN [28–30] is often used for webshell detection. Tian et al. [31] first applied CNN to webshell detection. Recurrent Neural Networks are good at extracting text sequence information, so they are also used for webshell detection [32]. Zhou et al. [33] use a single-layer LSTM to extract text sequence information from PHP opcode and conclude that using a two-layer or more-layer LSTM will reduce the detection effectiveness.

Webshell can be transformed into sequential text information by some means. Word2vec is a tool that can transform text information into word vectors and is used for webshell detection [34]. Tong [35] extracted word vectors from the native webshell using BERT. This method is advantageous since it eliminates the requirement to execute targeted compilation for various programming languages. Nevertheless, it is ineffective against encryption and obfuscation when utilizing the native webshell. Min [36] used BERT to PHP-compiled opcodes and employ a modified Bi-LSTM network for detection. This method yields excellent results, but it continues to prioritize the PHP language above the JSP language. This paper proposes a new method for extracting word vectors from transformed bytecode from JSP webshells. We are the first study to use BERT to extract word vectors from transformed bytecode from JSP webshells to the best of our knowledge.

3. Methodology

3.1. Architecture. In this paper, we propose an effective JSP webshell detection model. Our model is divided into three parts: data preprocessing, word vector extraction, and the machine learning classification model. The architecture designed is shown in Figure 1. In data preprocessing, we convert the original JSP file into a servlet in the form of a Java class through Tomcat Server and then compile it to generate bytecode for extracting text features. In the word vector extraction stage, we use BERT's tokenizer to convert the bytecode into its corresponding token id and mask and then input the obtained token id and mask into the pretrained BERT model for fine-tuning to obtain the word vector. Finally, we input the word vector into our machine learning model, which will be able to determine whether the output word vector belongs to a JSP webshell or not.

3.2. Data Preprocessing. JSP serves user requests using the Java language as a script and can collaborate with other Java programs on the server to address sophisticated business requirements. Java code is embedded in a static JSP page and used as a template to dynamically generate the parts of web pages. Some webshells, in addition to Java code, also contain some HTML code for displaying visual interfaces. Certain JSP webshells may also contain obfuscated or encrypted code, making it difficult to extract functionality straight from the source JSP files. We must process the original JSP file in order to convert it to a text file and then extract the text information directly from the text file.

After deploying the JSP to the server, the user can view the rendered JSP page by navigating to the server's JSP file. When a user first accesses a JSP file, the server converts it to an executable servlet in the form of a Java class. Since Java is a platform-independent language, the same JSP code can be written once and run on a multitude of different operating systems. After the first request, the servlet is saved on the server, and subsequent requests for the JSP file do not require recompilation.

Javap is a Java class file decomposer that is frequently used for decomposition of class files. It can decompile and inspect the bytecode generated by the Java compiler. We can use the command $javap - c$ to get the bytecode in the Java class generated by the servlet after compilation. Following compilation, we can obtain a text file containing bytecode. To extract successive bytecodes from text files, regular expressions are utilized. The bytecode before and after regular expression extraction is shown in Figure 2.

3.2.1. Tomcat Server. We use Tomcat Server to convert raw JSP files into Java class in our model. Tomcat Server is the preferred development and debugging environment for JSP applications, particularly for small and medium-sized systems and applications with few concurrent users. We set up Tomcat Server on port 8080 of the local server and then put the original JSP files into $webapps/jspproject$ under the Tomcat Server directory. Following that, we connect to the local server's port 8080. If the compilation succeeds, we can find the compiled Java class file inside the Tomcat Server directory in the $/work/Catalina/localhost/jspproject/org/apache/jsp$ directory under the Tomcat Server directory.

3.2.2. Bytecode. JSP code usually contains two parts, one is HTML code, which is usually used to display the web page, and the other is Java code contained by <%> tags that can be used to handle the interaction of the web page with the server. The bytecode processing flow is shown in Figure 3. Due to the fact that the original JSP code lacks substantial

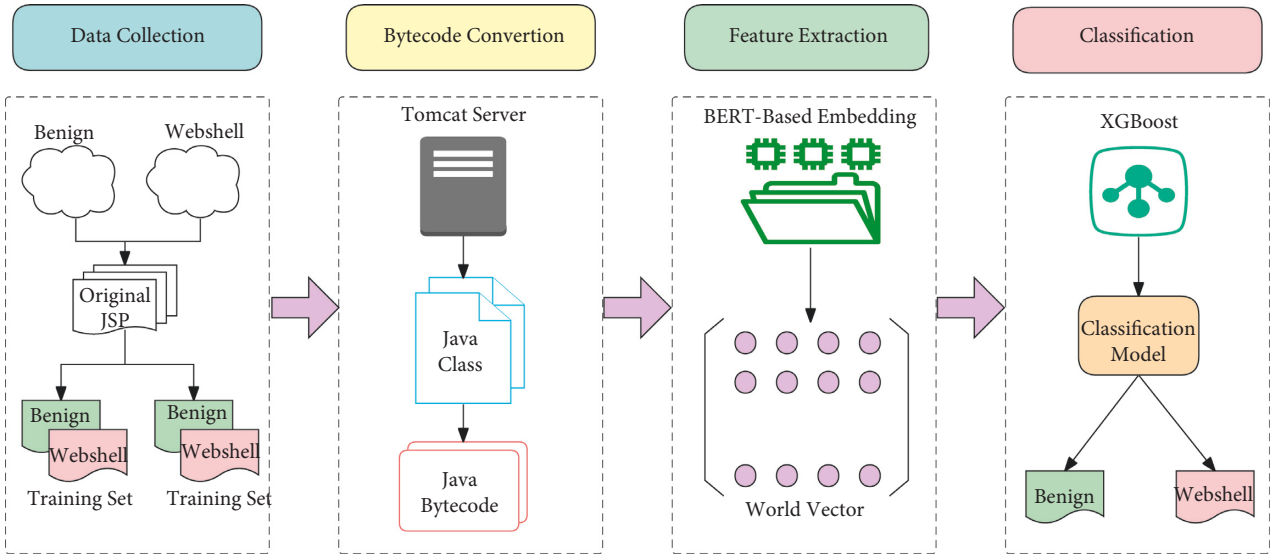

Figure 1: The architecture of our model.

```
Compiled from "_0_jsp.java"
public final class org.apache.jsp._0_jsp extends
org.apache.jasper.runtime.HttpJspBase implements
org.apache.jasper.runtime.JspSourceDependent,org.apache.jasper.runtime.
JspSourceImports{
  static {};
    Code:
       0: invokestatic  #26                 // Method javax/servlet/jsp/JspFactory.
getDefaultFactory:()Ljavax/servlet/jsp/JspFactory;
       3: putstatic     #32                 // Field _jspxFactory:Ljavax/servlet/jsp/JspFactory;
       6: new           #34                 // class java/util/HashSet
       9: dup
      10: invokespecial #36                 // Method java/util/HashSet."<init>":()V
      13: putstatic     #39                 // Field _jspx_imports_packages:Ljava/util/Set;
      16: getstatic     #39                 // Field _jspx_imports_packages:Ljava/util/Set;
      19: ldc           #41                 // String javax.servlet
      21: invokeinterface #43,  2           // InterfaceMethod java/util/Set.add:
(Ljava/lang/Object;)Z
      26: pop
      27: getstatic     #39                 // Field _jspx_imports_packages:Ljava/util/Set;
      30: ldc           #49                 // String javax.servlet.http
      32: invokeinterface #43,  2           // InterfaceMethod java/util/Set.add:
(Ljava/lang/Object;)Z
      37: pop
      38: getstatic     #39                 // Field _jspx_imports_packages:Ljava/util/Set;
      41: ldc           #51                 // String javax.servlet.jsp
      43: invokeinterface #43,  2           // InterfaceMethod java/util/Set.add:
(Ljava/lang/Object;)Z
      48: pop
      49: aconst_null
      50: putstatic     #53                 // Field _jspx_imports_classes:Ljava/util/Set;
      53: return
                                            ...   ...   ...
```

```
invokestatic
putstatic
new
dup
invokespecial
putstatic
getstatic
ldc
invokeinterface
pop
getstatic
ldc
invokeinterface
pop
getstatic
ldc
invokeinterface
pop
aconst_null
putstatic
return
...   ...   ...
```

Figure 2: The file before extraction with regular expressions.

textual features, some transformation of the JSP code is required. Tomcat transforms the Java class file into hexadecimal data that can be translated, and the translated bytecode provides some semantic information. Bytecode is cross-platform because it still needs to be compiled in the JVM (Java Virtual Machine).

3.3. Feature Extraction

3.3.1. Word2vec-Based Word Embedding. Word2vec [37] is a tool for converting words into real-valued vectors and is one of the language models widely used in natural language processing for learning semantic knowledge from large text

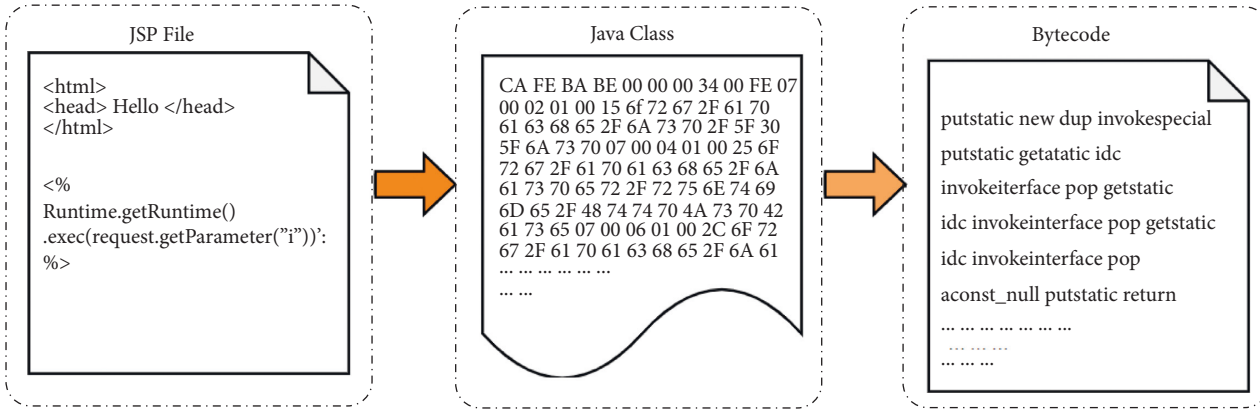

Figure 3: Bytecode processing flow.

corpora unsupervised. Word2vec makes use of both the Skip-Gram and the Continued Bag-of-Words (CBOW) models. The Skip-Gram approach is used to determine the context of the current word, whereas the CBOW model predicts the target value based on the context, which is the method we utilize here. In our experiments, we obtain the bytecode, from which we can extract the word vector. Since the length of bytecode varies, the length of the word vector extracted from bytecode also varies. Therefore, a suitable threshold is chosen as the maximum length of the word vector. If the bytecode is too long, a part of it will be truncated, and conversely, if the bytecode is shorter than the threshold value, it will be filled with 0 to the threshold length. The vector of each word is high-dimensional.

3.3.2. Word Segmentation. Before inputting the BERT model, the bytecode must be converted to token ID, and creating token ID requires word segmentation of the input text. There are numerous algorithms for segmenting words. Wordpiece is employed in BERT as a tool for word segmentation, and its main algorithm is the greedy matching algorithm. The Wordpiece algorithm's fundamental concept is that each word in the bytecode is traversed forward from the last letter, and each traversal determines whether the first letter to the currently traversed position is a word in the vocabulary. If yes, the current word is divided into two subwords from the traversed point, and the preceding procedure is repeated for each subword. If the split subword does not begin the unsplit word, it should be preceded by a "##" indication to aid in assembly during decoding. This algorithm is shown in Algorithm 1.

3.3.3. BERT-Based Word Embedding. Traditional word vector models, such as Word2vec, are ineffective at explaining polysemous words, and it is difficult to distinguish their various meanings in various contexts. In 2018, Google proposed the BERT (Bidirectional Encoder Representation from Transformers) model, a pretrained linguistic representation model that employs encoders in a bidirectional transformer for feature extraction to generate dynamic word vectors [38], which can solve the problem of multiple meanings of a word. Simultaneously, BERT can be pretrained to obtain various levels of features on various network layers in order to better represent the complex properties of words. When applied to some downstream tasks, BERT usually requires simple fine-tuning to obtain good results. This paper proposes a method to apply BERT's word embedding technique to JSP webshell detection. In the BERT structure, the [*CLS*] and [*SEP*] are two unique symbols. In the binary classification task, [*CLS*] is added to the beginning of the sentence to indicate that it is a single-sentence classification task, and the [*SEP*] token will not be added to the sentence in the binary classification task.

After retrieving the bytecode for the original JSP file, we can use it to generate the word vector using BERT. BERT understands the input appropriately by utilizing the unique tokens [*CLS*]. The [*CLS*] token is unique. The output vector corresponding to the [*CLS*] symbols can be used as the semantic representation of the entire text in the classification task, as it contains the semantic information for each word in the text. Token embedding is a term that refers to the addition of tokenized sentences, which is combined with positional and segment embedding as the final input to BERT. After passing the bytecode to BERT, the classifier receives the output vector corresponding to our [*CLS*] symbols. The procedure of word vector extraction is shown in Figure 4. In the figure, the final output T is the feature vector. C represents the output vector corresponding to the input [*CLS*]. The characters on the same line as the [*CLS*] token are the bytecode after word segmentation, and these characters are processed to get the corresponding code; for example, [*CLS*] corresponds to the code 102.

3.4. Classification Model. Since Word2vec and BERT generate word vectors with sequential contextual data, they can be classified using the Bi-LSTM classifier. In comparison, for webshell detection, which is a binary classification problem, XGBoost may be utilized as an excellent classification model. As a result, we compare the performance of the two classifiers.

3.4.1. XGBoost. XGBoost is short for Extreme Gradient Boosting. XGBoost is a toolkit for parallelizing the Boost tree with a focus on speed, effectiveness, and portability. It is an upgrade on the standard GBDT (Gradient Boosting

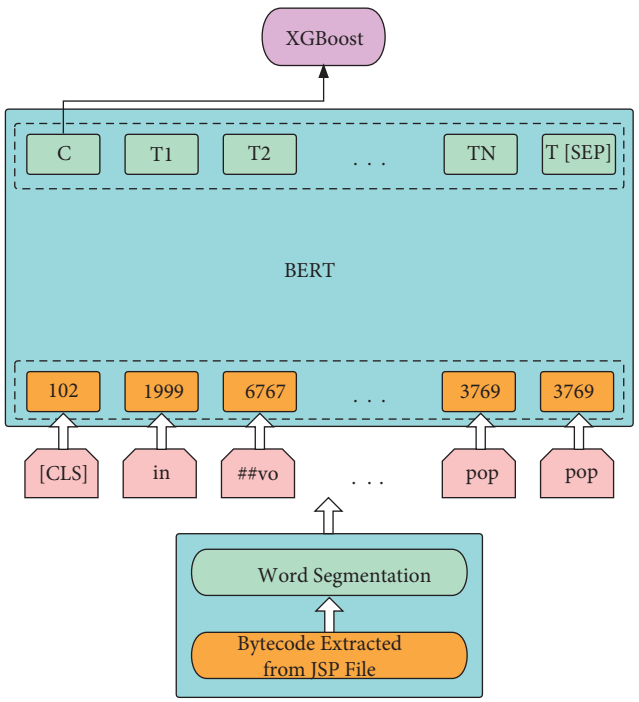

Figure 4: The process of word vector extraction.

Decision Tree) algorithm by using a loss function, regularization, and several other techniques to enhance the result.

XGBoost achieves excellent prediction accuracy by adding a new tree to fit the difference between the previous tree's forecast result and the actual value and establishing a new model as the foundation for subsequent model learning. If we set F to be the set of all decision tree models, y_c to be the c th prediction of the model, and $f_k(u_c)$ to be the output of the k th decision tree model corresponding to the c th input feature u_c, K to be the total number of decision trees, then the prediction of the model is

$$y_c = \sum_{k=1}^{K} f_k(u_c), \quad f_k \in F. \tag{1}$$

Similarly, if we set $Y^{(m)}$ to be the objective function for iterations up to the m th round, $q(y_c, \hat{y}_c^{(m)})$ is the error function between the true value y_c and the predicted value $\hat{y}_c^{(m)}$ of the m th iteration, and the error function gradually makes the prediction result keep approximating the true value by continuous learning. Then $Y(m)$ can be expressed as follows:

$$Y^{(m)} = q\left(y_c, \hat{y}_c^{(m)}\right) + \Omega, \tag{2}$$

where Ω is the penalty term; let γ, λ be the coefficient matrix of the number of leaves and the fraction of leaf nodes, respectively, and let T be the number of leaves in each tree and ω be the set consisting of the fraction of leaf nodes in each tree; then Ω can be expressed as follows:

$$\Omega = \gamma T + \frac{1}{2}\lambda\|\omega\|^2. \tag{3}$$

The word vectors are extracted from the JSP file, and we need a capable classifier with good performance and accuracy. XGBoost is fast, accurate, and adaptable and often performs well in handling binary classification tasks. Therefore, we chose XGBoost as the classifier for word vectors. The word vectors extracted from BERT are used to input to XGBoost. Before we acquire the final prediction results, we need to use the input word vectors to train XGBoost. First, the word vectors are input to the first tree, and after we get the output results of the first tree, we add a new tree, which makes the word vectors obtained from BERT input to these two trees such that the objective function obtained is minimized. And so on, the training of XGBoost can be completed. The word vectors are input to each tree of XGBoost, and each tree is given the probability score that its word vector belongs to a webshell. The predicted probability scores of all the trees are added to yield the probability that the original JSP file is a webshell.

3.5. Bi-LSTM. While RNNs are effective in extracting sequence features, they may encounter issues such as exploding gradient and vanishing gradient. The forgetting gate, input gate, and output gate are all included in the LSTM to successfully solve the above concerns. The following formula is used to calculate a standard LSTM unit:

$$\begin{aligned}
X &= \begin{bmatrix} h_t - 1 \\ \\ w_t \end{bmatrix}, \\
\sigma(z) &= \frac{1}{1 + e^{-z}}, \\
f_t &= \sigma\left(W_f X + b_f\right), \\
i_t &= \sigma\left(W_i X + b_i\right), \\
o_t &= \sigma\left(W_o X + b_o\right), \\
c_t &= f_t \otimes c_{t-1} + i_t \otimes \left(W_c X + b_c\right), \\
h_t &= o_t \otimes \tanh\left(c_t\right),
\end{aligned} \tag{4}$$

where $\otimes$ denotes the dot product of the corresponding elements, σ denotes the sigmoid function in the neural network, and W_f, W_i, W_o, and W_c are the weight matrices of the hidden layers, and b_f, b_i, b_o, and b_c are the bias vectors. f_t, i_t, and o_t are the forgetting gate, input gate, and output gate, respectively, h_t is the output at moment t, and c_t is the state of the final memory unit.

The LSTM has a long-time memory function and is well-suited for processing long sequences. However, since the LSTM is a simple one-way propagation neural network, it is incapable of connecting the information contained in preceding and subsequent texts for the purpose of learning. Bi-LSTM usually has a strong performance in classifying word vectors. When word vectors are input to the Bi-LSTM, the Bi-LSTM can efficiently learn the relationship between word vectors corresponding to operations such as pop from the

Input: A word to be split, W
Output: Lists of subwords, L
$L \leftarrow Empty\ list$
while W *not in vocab* **do**
  $S \leftarrow Find\ the\ longest\ in\ vocab\ from\ W.$
  if L *is empty* **then**
    $L.push(S)$
else
    $L.push(strcat(\text{“\#\#,”}S))$
  end
  $W \leftarrow Get\ the\ remaining\ subword\ from\ W\ after\ removing\ S.$
end
$L.push(W)$

Algorithm 1: Word segmentation algorithm of BERT.

stack and push to the stack. Through the intrinsic connection of these operations, Bi-LSTM can effectively analyze which word vectors belong to the webshell word vectors. Therefore, we employ Bi-LSTM as a control group of the classifier in our experiments to evaluate the effect of our experiments.

4. Evaluation

In this section, we will present the details of our dataset, the experimental setting, the procedure and details of the experiment, and the experiment's conclusions.

4.1. Dataset. Due to the fact that the number of webshell files is significantly less than the number of benign files on a typical website, the number of benign files in our dataset exceeds the number of JSP webshells. We gathered 2903 JSP samples from open-source projects, which included 830 webshells and 2073 benign JSP files. Among them, we collected innocuous JSP samples from Github's open-source projects, and owing to a dearth of JSP samples, we personally generated several JSP webshell samples, which included some JSP webshell samples that were obfuscated and encrypted to verify the correctness of model testing. The JSP webshell projects collected from Github are shown in Table 1. After deduplicating the JSP samples gathered, including benign and webshell samples, we ended up with 2903 samples. In our trials, we converted them to Java classes and then turned them into bytecode files using the Tomcat Server.

4.2. Experiment Design. To examine the effectiveness of our derived methods, we used both Word2vec and BERT word embedding methods for feature extraction. For the classification model, we used Bi-LSTM to compare it against XGBoost, which excels at solving classification issues, in the hope of obtaining the best performing model. In addition, we also used the Support Vector Machine algorithm and the more generalized RandomForest algorithm as comparison experiments. A total of eight sets of experiments were used to test the effectiveness of the model.

Table 1: Github projects lists.

No	Source
#0	https://github.com/ysrc/webshell-sample
#1	https://github.com/xl7dev/WebShell
#2	https://github.com/tutorial0/WebShell
#3	https://github.com/tennc/webshell
#4	https://github.com/tanjiti/webshellSample
#5	https://github.com/oneoneplus/webshell
#6	https://github.com/threedr3am/JSP-WebShells
#7	https://github.com/fr4nk404/Webshell-Collections

Our experiments were all run on Kaggle [39], which offers machine learning competitions and provides a Jupyter environment with GPUs. We used an Intel(R) Xeon(R) CPU @ 2.00 GHz with 16 GB of RAM and a GPU of Tesla P100-PCIE. Pytorch is used to implement Bi-LSTM, and the XGBoost package is used to compute XGBoost in Python.

The experiments were conducted using 5-fold cross validation. The dataset is partitioned into five equal portions, with one serving as the test set and the other as the training set. To reduce the effect of randomness in dataset splitting, we run the experiment five times and take the average of the five results as our experimental result. Each experiment has 2322 JSP samples in the training set and 581 JSP samples in the testing set. To eliminate the randomness effect when the dataset is disrupted, we used a random number seed 42 to ensure that the training and test sets for each division are the same when comparing the trials.

Through the Tomcat Server, the original JSP file samples were converted to Java class files. Following that, the Java class file was compiled into a bytecode-containing text file. Regular expressions were employed to extract the bytecode. Four distinct sets of experiments were used to compare and confirm our model's performance.

4.3. Evaluation Metrics. The model was evaluated using four metrics: accuracy, precision, recall, and f1 score. Additionally, we displayed receiver operating characteristic (ROC) curves to evaluate the model's detection performance on the test set.

After determining the model with the best performance based on the aforementioned measures, we create the model's ROC (receiver operating characteristics) curve to visually assess its performance and determine the AUC value (area under curve). The AUC can be stated as follows:

$$\mathrm{AUC} = \frac{\sum_{i \in \mathrm{positiveclass}} \mathrm{rank}_i - (M(1+M)/2)}{M \times N}, \quad (5)$$

where M denotes the number of positive samples and N denotes the number of negative samples. rank_i denotes the sum of the rankings of the probability of predicting all positive samples as positive (the sample with the highest probability is $\mathrm{rank}_{(M+N)}$).

4.4. Experiment Result. Using the aforementioned methodologies, we designed tests and evaluated each combination of feature vector extraction methods and classification models. The results are shown in Table 2.

The evaluation results in the table represent the maximum values that can be obtained using the model's various parameters. The evaluation results in the table represent the maximum values that can be obtained using the model's various parameters. Model 1, Model 3, Model 5, and Model 7 all use Word2vec for word vector extraction, while Model 2, Model 4, Model 6, and Model 8 employ BERT as a tool for word vector extraction. They are classified by four machine algorithms, including XGBoost, which is strong at binary classification tasks, and Bi-LSTM algorithm, which is good at classifying word vectors, so we can achieve the best classification results. Our model achieves 99.14% accuracy, 98.68% precision, 98.03% for recall, and 98.35% for the final f1 score, all of which are greater than the accuracy, precision, recall, and final f1 score of other models, demonstrating that the BERT word embedding strategy is perfect for webshell detection. The hyperparameters of XGBoost are set as follows: max_depth = 6, min_child_weight = 1, n_estimators = 150, gamma = 0. The rest of the hyperparameters are set by default.

From the above experimental results, we can conclude that BERT performs better than Word2vec as a word vector extraction technique for bytecode of JSP files, allowing for a more accurate classification of the original JSP files as webshell and normal files. We have analyzed the two most crucial points: First, Word2vec has only one representation for each token, so it cannot understand some cases of polysemy. Some words in bytecode do not exist or have incorrect meanings in Word2vec's vocab, which may reduce the accuracy of the word vectors extracted from bytecode by Word2vec. BERT can successfully comprehend polysemous words in context, allowing word vectors derived from bytecode to be comprehended more precisely in contextual relationships and with fewer errors. Second, Word2vec's vocabulary is fairly extensive, containing up to one million words. In this situation, performance and accuracy must be compromised. However, BERT as a pretrained model and the Byte-Pair Encoding tokenizer it possesses contain appropriate levels of vocabulary. Therefore, in most cases, BERT can be used with a high degree of accuracy.

To visualize the classification results, we plotted the receiver operating characteristic (ROC) curves for several feature extraction and classification models, as shown in Figure 5. As can be observed, the model performs best when BERT is used to extract the word vectors and then XGBoost is used as the classification model. Thus, we use this model to recognize JSP webshells.

Zhang et al. extracted statistical features from bytecode using the TF-IDF technique and classified them using the XGBoost classifier. However, this technique does not take into account bytecode context information. Liu et al. extract word vectors from the source JSP files and classify them using Bi-GRU with attention. It is worth noting that this straight word vector extraction of raw JSP files is insufficiently powerful to discover obfuscated and encrypted code files. Due to the possibility that our dataset differs from theirs, we conducted trials using our dataset, the results of which are displayed in Table 3.

4.5. Case Study. Today's webshells employ obfuscation, encryption, and a variety of bypass mechanisms, and conventional webshell detection models are not as efficient as they could be at detecting such webshells. To evaluate our model's effectiveness against such webshells, a total of 393 webshells were used to check whether the obfuscated and encrypted webshells could be detected correctly. The main sources of the encrypted and obfuscated webshells are shown in Table 4 below. The project #0 is a collection of several webshells that can bypass the detection of professional tools. The projects #1 and #2 are generators of JSP webshells. The project #1 can generate different JSP webshells with obfuscation each time. The webshell generator in project #2 has a variety of features, including eleven bypass methods such as dynamic compilation using javac, loading bytecode with ClassLoader, dynamically generating bytecode with ASM, and loading BCEL bytecode, and Unicode encoding of parameters based on any of the bypass methods. In order to check the robustness of our model, we use our model and other models dedicated to detecting JSP webshells mentioned in the references for these 393 special JSP webshell files, respectively.

The original JSP webshell files were translated to bytecode using the same method and then evaluated using our model. Due to the fact that all created files are webshell files, we chose accuracy as the evaluation metric. The results of the test are shown in Table 5. From the experimental results, we see that our model can still have excellent detection results for encrypted and obfuscated JSP webshell files, and we can conclude that the means to combat obfuscation and encryption are weaker when using the source code directly for detection. The experimental results show that our model still works quite well for various encrypted and obfuscated JSP webshells with 97.96% accuracy, indicating our proposed method's good robustness.

Table 2: The evaluation metrics results.

Feature extraction	Classification algorithm	Evaluation metrics			
		Accuracy	Precision	Recall	F1 score
Word2vec	Bi-LSTM	0.9800	0.9627	0.9685	0.9654
BERT	Bi-LSTM	0.9852	0.9703	0.9789	0.9746
Word2vec	XGBoost	0.9859	0.9805	0.9701	0.9753
BERT	*XGBoost*	*0.9914*	*0.9868*	*0.9803*	*0.9835*
Word2vec	RandomForest	0.9844	0.9901	0.9562	0.9729
BERT	RandomForest	0.9842	0.9865	0.9589	0.9724
Word2vec	Support vector machine	0.9852	*0.9926*	0.9561	0.9740
BERT	Support vector machine	0.9828	0.9652	0.9751	0.9700

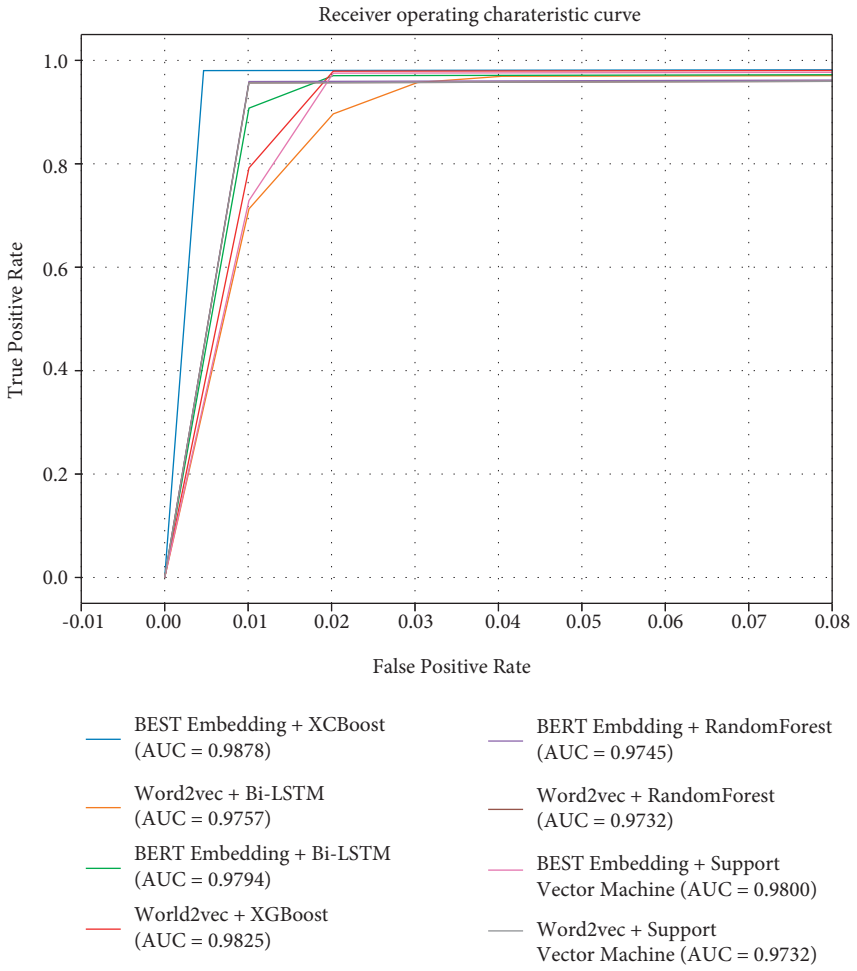

Figure 5: ROC curves of the different models.

Table 3: Model performance with related works.

	Accuracy	Precision	Recall	F1 score
Zhang et al. [8]	0.9848	0.9775	0.9706	0.9739
Liu et al. [9]	0.9873	0.9844	0.9718	0.9780
Our model	*0.9914*	*0.9868*	*0.9803*	*0.9835*

Table 4: The main source of encrypted or obfuscated JSP webshells.

Index	Github source
#0	LandGrey/webshell-detect-bypass
#1	pureqh/webshell
#2	Sec-Fork/JSPHorse

Table 5: Results of webshell detection with encryption and obfuscation.

	TP (true positive)	Accuracy
Zhang et al. [8]	379	0.9644
Liu et al. [9]	356	0.9059
Our model	*385*	*0.9796*

5. Conclusion

In this paper, we present a model for detecting JSP webshells and demonstrate the usage of BERT in transformed bytecode from JSP webshell for the first time. Finally, our model achieves 99.14% accuracy and significantly greater precision, recall, and f1 score than prior JSP webshell identification models. The Tomcat Server is used to convert JSP files to Java class files, which are subsequently converted to bytecode. The bytecode is processed using BERT-based word vector extraction, and the extracted word vectors are classified using XGBoost. Additionally, we compare our model to the conventional webshell detection model. Previously, Word2vec was frequently used to extract word vectors for opcode or bytecode extraction, and the model suggested in this study may provide researchers with a fresh way of thinking.

This study still contains certain limitations. First, there are few publicly available JSP webshell datasets on the Internet. The experimental results could be more convincing if we could get more data. Second, the method in this paper needs to compile JSP webshell into the corresponding bytecode; hence porting this method to other programming languages is challenging. In the future, we can work on the following areas.

(1) We can conduct experiments on the methods mentioned in this manuscript on webshells of other programming languages to check the generality of the method in this manuscript.

(2) The literal meaning of the bytecode we extracted is not obvious, while BERT can understand the context effectively. Therefore, we may be able to find a way to semanticize bytecode to improve the method in this manuscript.

The webshell detection method, which uses BERT to extract word vectors, was demonstrated to be effective experimentally. We can attempt to extend this method in the future to detect webshells in other programming languages.

Data Availability

The data that were utilized to substantiate the study's findings are included within the article.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This research was funded by the National Key Research and Development Program of China (no. 2021YFB3100500).

References

[1] CNCERT, "Weekly report of cncert-issue 9," 2022, https://www.cert.org.cn/publish/main/upload/File/Weekly Report of CNCERT-Issue 9 2022.pdf.

[2] Y. Lawrence, L. Dong Liang, C. Yung-Hui, and L. X. Yann, "Lexical analysis for the webshell attacks," in *Proceedings of the 2016 International Symposium on Computer, Consumer and Control (IS3C)*, pp. 579–582, IEEE, Xi'an, China, 04-06 July 2016.

[3] Y. Li, J. Huang, A. Ikusan, M. Mitchell, J. Zhang, and R. Dai, "Shellbreaker: automatically detecting php-based malicious web shells," *Computers & Security*, vol. 87, Article ID 101595, 2019.

[4] Z. Pan, Y. Chen, Y. Chen, Y. Shen, and X. Guo, "Webshell detection based on executable data characteristics of php code," *Wireless Communications and Mobile Computing*, vol. 2021, Article ID 5533963, 12 pages, 2021.

[5] W. Kang, S. Zhong, K. Chen, J. Lai, and G. Xu, "Rf-adacost: webshell detection method that combines statistical features and opcode," in *International Conference on Frontiers in Cyber Security*, vol. 1286, pp. 667–682, Springer, Singapore, 2020.

[6] M. V. Peter and B. V. W. Irwin, "Towards a php webshell taxonomy using deobfuscation-assisted similarity analysis," in *Proceedings of the 2015 Information Security for South Africa (ISSA)*, pp. 1–8, IEEE, Johannesburg, South Africa, 12-13 August 2015.

[7] A. Zhou, N. Luktarhan, and Z. Ai, "Research on webshell detection method based on regularized neighborhood component analysis (rnca)," *Symmetry*, vol. 13, no. 7, Article ID 1202, 2021.

[8] H. Zhang, M. Liu, Z. Yue, Z. Xue, Y. Shi, and X. He, "A php and jsp web shell detection system with text processing based on machine learning," in *Proceedings of the 2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pp. 1584–1591, Guangzhou, China, December 2020.

[9] Z. Liu, D. Li, and L. Wei, "A new method for webshell detection based on bidirectional gru and attention mechanism," *Security and Communication Networks*, vol. 2022, Article ID 3434920, 11 pages, 2022.

[10] W. Yang, B. Sun, and B. Cui, "A webshell detection technology based on http traffic analysis," in *International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, vol. 773, pp. 336–342, Springer, Singapore, 2018.

[11] Y. Wu, M. Song, Y. Li et al., "Improving convolutional neural network-based webshell detection through reinforcement learning," in *International Conference on Information and Communications Security*, pp. 368–383, Springer, Cham, 2021.

[12] Y. Fang, Y. Qiu, L. Liu, and C. Huang, "Detecting webshell based on random forest with fasttext," in *Proceedings of the 2018 International Conference on Computing and Artificial Intelligence*, pp. 52–56, Association for Computing Machinery, New York, NY, USA, 2018.

[13] Y. Fang, M. Xie, and C. Huang, "Pbdt: Python backdoor detection model based on combined features," *Security and Communication Networks*, vol. 2021, Article ID 9923234, 13 pages, 2021.

[14] H. Cui, D. Huang, Y. Fang, L. Liu, and C. Huang, "Webshell detection based on random forest–gradient boosting decision tree algorithm," in *Proceedings of the 2018 IEEE Third*

International Conference on Data Science in Cyberspace (DSC), pp. 153–160, IEEE, Guangzhou, China, 18-21 June 2018.

[15] Y. Guo, H. Marco-Gisbert, and P. Keir, "Mitigating webshell attacks through machine learning techniques," *Future Internet*, vol. 12, no. 1, Article ID 12, 2020.

[16] Z. Wang, J. Yang, M. Dai, R. Xu, and X. Liang, "A method of detecting webshell based on multi-layer perception," *Academic Journal of Computing & Information Science*, vol. 2, no. 1, pp. 81–91, 2019.

[17] L. Jinping, T. Zhi, M. Jian, G. Zhiling, and Z. Jiemin, "Mixed-models method based on machine learning in detecting webshell attack," in *Proceedings of the 2020 International Conference on Computers, Information Processing and Advanced Education*, pp. 251–259, 2020.

[18] B. Yong, X. Liu, Y. Liu, H. Yin, L. Huang, and Q. Zhou, "Web behavior detection based on deep neural network," in *Proceedings of the 2018 IEEE SmartWorld, Ubiquitous Intelligence Computing, Advanced Trusted Computing, Scalable Computing Communications, Cloud Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*, pp. 1911–1916, Guangzhou, China, 08-12 October 2018.

[19] Y. Wu, Y. Sun, C. Huang, P. Jia, and L. Liu, "Session-based webshell detection using machine learning in web logs," *Security and Communication Networks*, vol. 2019, Article ID 3093809, 11 pages, 2019.

[20] A. Fass, R. P. Krawczyk, M. Backes, and B. Stock, "Jast: fully syntactic detection of malicious (obfuscated) javascript," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 303–325, Springer, Cham, 2018.

[21] S. Ndichu, S. Ozawa, T. Misu, and K. Okada, "A machine learning approach to malicious javascript detection using fixed length vector representation," in *Proceedings of the 2018 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8, IEEE, Rio de Janeiro, Brazil, 08-13 July 2018.

[22] Z. Li, Q. Alfred Chen, C. Xiong, Y. Chen, T. Zhu, and H. Yang, "Effective and light-weight deobfuscation and semantic-aware attack detection for powershell scripts," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1831–1847, New York, NY, USA, November 2019.

[23] A. Fass, M. Backes, and B. Stock, "Hidenoseek: camouflaging malicious javascript in benign asts," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1899–1913, November 2019.

[24] A. Fass, M. Backes, and B. Stock, "Jstap: a static pre-filter for malicious javascript detection," in *Proceedings of the 35th Annual Computer Security Applications Conference*, pp. 257–269, December 2019.

[25] Z. Zhang, M. Li, L. Zhu, and X. Li. Smartdetect, "A smart detection scheme for malicious web shell codes via ensemble learning," in *International Conference on Smart Computing and Communication*, pp. 196–205, Springer, Cham, 2018.

[26] S. Li, Y. Li, W. Han, X. Du, M. Guizani, and Z. Tian, "Malicious mining code detection based on ensemble learning in cloud computing environment," *Simulation Modelling Practice and Theory*, vol. 113, Article ID 102391, 2021.

[27] W. Wang, X. Wang, D. Feng, J. Liu, Z. Han, and X. Zhang, "Exploring permission-induced risk in android applications for malicious application detection," *IEEE Transactions on Information Forensics and Security*, vol. 9, no. 11, pp. 1869–1882, 2014.

[28] N.-H. Nguyen, V.-H. Le, V.-O. Phung, and P.-H. Du, "Toward a deep learning approach for detecting php webshell," in *Proceedings of the Tenth International Symposium on Information and Communication Technology*, pp. 514–521, December 2019.

[29] Z.-H. Lv, H.-B. Yan, and R. Mei, "Automatic and accurate detection of webshell based on convolutional neural network," in *China Cyber Security Annual Conference*, vol. 970, pp. 73–85, Springer, Singapore, 2019.

[30] H. V. Le, T. N. Nguyen, H. N. Nguyen, and L. Le, "An efficient hybrid webshell detection method for webserver of marine transportation systems," *IEEE Transactions on Intelligent Transportation Systems*, pp. 1–13, 2021.

[31] Y. Tian, J. Wang, Z. Zhou, and S. Zhou, "Cnn-webshell: malicious web shell detection with convolutional neural network," in *Proceedings of the 2017 VI International Conference on Network, Communication and Computing*, pp. 75–79, December 2017, https://doi.org/10.1145/3171592.3171593.

[32] Z. Zhao, Q. Liu, T. Song, Z. Wang, and X. Wu, "Wsld: detecting unknown webshell using fuzzy matching and deep learning," in *International Conference on Information and Communications Security*, pp. 725–745, Springer, Cham, 2019.

[33] Z. Zhou, L. Li, and X. Zhao, "Webshell detection technology based on deep learning," in *Proceedings of the 2021 7th IEEE Intl Conference on Big Data Security on Cloud (BigDataSecurity), IEEE Intl Conference on High Performance and Smart Computing(HPSC) and IEEE Intl Conference on Intelligent Data and Security (IDS)*, pp. 52–56, IEEE, NY, USA, 15-17 May 2021.

[34] Z. Ai, N. Luktarhan, AiJ. Zhou, and D. Lv, "Webshell attack detection based on a deep super learner," *Symmetry*, vol. 12, no. 9, Article ID 1406, 2020.

[35] H. Tong, "Research and implementation of webshell detection method based on deep learning," *Master's Thesis*, Beijing University of Posts and Telecommunications, 2021.

[36] J. Min, "Conv-bilstm: a new intelligent webshell detection network based on bi-lstm," *Master's Thesis*, Lanzhou University, 2021.

[37] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013, http://arxiv.org/abs/1301.3781.

[38] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: pre-training of deep bidirectional transformers for language understanding," 2018, http://arxiv.org/abs/1810.04805.

[39] K. Kaggle, https://www.kaggle.com, 2022.