Journal of Intelligent & Fuzzy Systems 41 (2021) 3615–3628
DOI:10.3233/JIFS-211011
IOS Press

Intelligent mining vulnerabilities in python code snippets

Wenbo Guo[a], Cheng Huang[a,*], Weina Niu[b] and Yong Fang[a]
[a]*School of Cyber Science and Engineering, Sichuan University, Chengdu, China*
[b]*Institute for Cyber Security, School of Computer Science and Engineering, University of Electronic Science and Technology of China, Chengdu, China*

Abstract. In the software development process, many developers learn from code snippets in the open-source community to implement specific functions. However, few people think about whether these code have vulnerabilities, which provides channels for developing unsafe programs. To this end, this paper constructs a source code snippets vulnerability mining system named PyVul based on deep learning to automatically detect the security of code snippets in the open source community. PyVul builds abstract syntax tree (AST) for the source code to extract its code feature, and then introduces the bidirectional long-term short-term memory (BiLSTM) neural network algorithm to detect vulnerability codes. If it is vulnerable code, the further constructed a multi-classification model could analyze the context discussion contents in associated threads, to classify the code vulnerability type based the content description. Compared with traditional detection methods, this method can identify vulnerable code and classify vulnerability type. The accuracy of the proposed model can reach 85%. PyVul also found 138 vulnerable code snippets in the real public open-source community. In the future, it can be used in the open-source community for vulnerable code auditing to assist users in safe development.

Keywords: Open-source community, vulnerability mining, content analysis, BiLSTM

1. Introduction

The rapid development of computer technology. On the one hand, it has accelerated the progress of human society, but it also brought many threats. Technology is always a double-edged sword. In recent years, hacker activities are getting more and more frequent. They use the program vulnerabilities to steal personal privacy and even launch attacks to endanger national security. As the foundation of computer systems, software programs have vulnerabilities that allow criminals to take advantage of them.

During the software life cycle, most of the bugs come from the development phase. During the software development process, developers will mostly find code through open-source communities, including StackOverflow and GitHub, but few people consider whether the code on it is safe or not. Suppose some hackers maliciously upload code or program with vulnerabilities, and developers do not carefully review and refer directly to their software. In that case, it is very likely to leave fatal vulnerabilities [1].

GitHub and StackOverflow [2] are the most popular open-source communication communities for software developments. GitHub is a hosting platform for open and private software projects, hosting many program source code. StackOverflow is an IT technology Q&A website. Developers can submit questions, browse questions, find relevant content, and so on for free. Until September 2018, StackOverflow had over 9,400,000 registered users and over 16,000,000 questions, with the most common topics being JavaScript, Java, PHP and Python. Many posts contain much vulnerable code, and research has found that Stack-

*Corresponding author. Cheng Huang, School of Cyber Science and Engineering, Sichuan University, Chengdu 610065, China. E-mail: opcodesec@gmail.com.

Overflow copies the most code ever containing bugs [3], which seriously threatens the development of software programs. Python, one of the most global popular programming languages, ranks among the top three in all major programming charts, and has more than 1,440,000 communication topics on Stack-Oveflow. Well-known Internet companies, including Google and Facebook, are using it. Python also has many vulnerabilities [4]. Rahman et al. [5] found 12 common types of vulnerabilities in Python, including Command Execution, HardCode, and SQL Injection.

There are many tools for vulnerability mining, such as Scan Dal [6], Hybri Droid [7] and PREfix [8]. They are mostly used to detect vulnerabilities in C language or Java, such as Fortify [9], which can detect vulnerabilities in many programming languages. There is also a certain amount of research on Python vulnerability mining. There are mainly the following methods.

Vulnerability detection based on code similarity is mainly realized by similarity code, which may contain the same vulnerability. Li et al. [10] proposed an efficient code similarity detection system, VulPecker. By learning NVD and the data in the vulnerability code instance database (VCID), then generated vulnerability feature, and the patch database (VPD) is constructed to generate diff blocks for vulnerability code block extraction. Although this method is effective, it is only applied to C/C++, and the vulnerability database is not perfect. Karnalim [11] proposed a similarity detection technique, which generates a syntax tree for program code files, extracts directly connected n-gram structure token from them, and uses an information retrieval algorithm to perform a subsequent comparison, namely Cosine correlation in the vector space model. Although the accuracy of this method has been improved to a certain extent, it sacrifices execution efficiency. Peng et al. [12] Proposed a python vulnerability mining method based on similarity. This method realizes vulnerability detection by comparing the similarity between the current file and the original file containing the vulnerability. However, this method only considers the similarity of data flow, and does not consider the similarity of control dependence. At the same time, the type of vulnerabilities that this method can detect depends on the size of the comparison sample set, that is, the smaller the comparison sample set, the worse the detection effect, and it is not suitable for large-scale vulnerability detection in the open source community.

Vulnerability mining for third-party extensions Mahmoud et al. [13] found that there are many vulnerabilities in python third-party libraries. PyXhon [14] is a plug-in for python security programming, which can detect security vulnerabilities and privacy leaks from third-party extensions. PyXhon is a function-oriented analysis. Developers use it to monitor the process of all function calls; dynamic byte instruction tracking analysis, which infers whether the behavior of importing modules and accessing private DLLs conforms to the security policy, and it provides policies to accept or reject extensions. These security mechanisms do not require Python language feature, so they are completely transparent to Python applications. However, this method can only identify security issues introduced by third parties, and cannot identify other types of vulnerabilities.

Rule-based vulnerability mining In the rule-based vulnerability detection method, experts manually analyze all kinds of vulnerabilities to generate vulnerability rules. Based on the lexical and syntactic analysis, source code modeling, data flow analysis, stain analysis, etc. Prabakar et al. [15] proposed a SQL injection detection system based on Aho-Corasick pattern matching. During the detection process, the system maintains a list of known abnormal patterns. Anomaly detection is achieved by applying a pattern matching algorithm to check user-generated SQL query comparison lists for known patterns. The rule-based vulnerability detection needs to rely on experts to customize the rules, but the rules cannot be fully covered, which is only useful for some vulnerabilities.

Vulnerability detection based on deep learning is the current research hotspot. There are many machine learning algorithms with good results. At present, there are many attempts in script language and other programming languages. Vuldeepecker [16] uses the BiLSTM algorithm to construct the detection model, then constructs the code gadgets to extract code feature, and then marks it (0/1). Finally, it vectorizes them for training the bidirectional long short-term memory neural network. The experimental results show that it has a good effect. However, there is a significant problem of information loss in this scheme in converting code gadgets into vectors. Simultaneously, this method only applies to the C/C++ buffer overflow and vulnerability of resource management error type.

We found that the code in some posts on Stack-Overflow contains vulnerabilities. Concurrently, in the context discussion contents of associated threads, many users discuss the details of the vulnerabili-

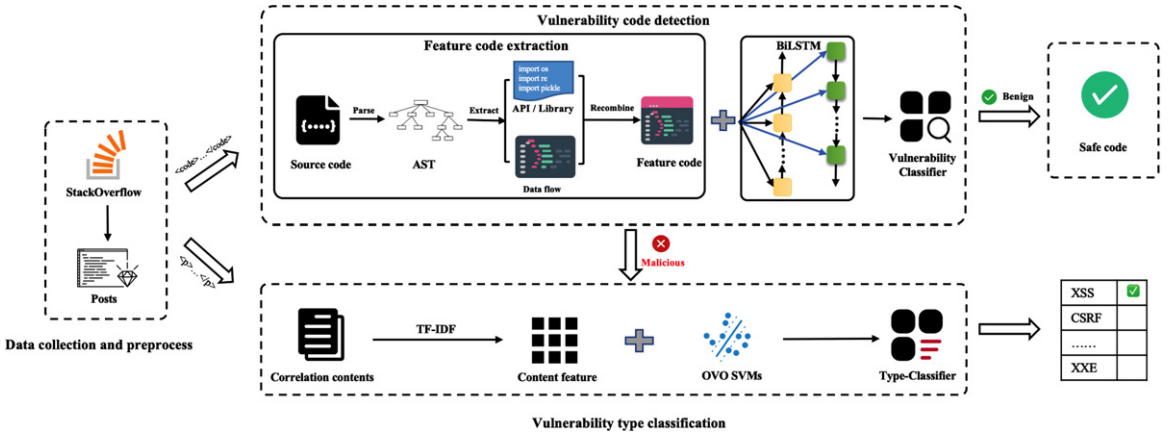


Fig. 1. The framework of PyVul.

ties, including the types of vulnerabilities and repair methods.

For this reason, the paper proposes a vulnerability mining system named PyVul for the source code snippets of the open-source community. The system consists of two parts, a vulnerability mining module for source code snippets and a vulnerability type classification module for context discussion contents in associated threads.

The specific contributions of this work are the following:

- The paper constructs a general code feature extraction method for open-source community. This method can effectively integrate unstructured code and extract code feature while retaining semantic and structural information.
- The paper introduces the BiLSTM algorithm to the vulnerability mining of Python code and achieves good results. The accuracy of the model can reach 85%.
- The paper combines the vulnerability mining of source code snippets with content mining for the first time, which realizes the vulnerability detection of code snippets and uses the OVO SVMs algorithm to identify its vulnerability types. The recognition accuracy can reach 90%.

The rest of the paper is organized as follows: Section 2 is a mathematically formalized problem definition. Section 3 details the implementation process of the PyVul system. Section 4 presents the experiments and analyses. Section 5 summarizes the conclusion and proposes future works.

2. Problem definition

Let $P = \{p_1, p_2 ..., p_n\}$ be a set of post in open-source community, $R(p_i) = \{r_1, r_2 ..., r_n\}$ represents the reply corresponding to the p_i post. $T(r_i) = \{s_i, w_i\}$ represents the source code and text content in each reply.

Our goal is to detect the security of the Python code snippet s_i in the post, and determine the type of vulnerability if it is malicious code. The process is divided into two steps. The first step is vulnerability code detection. We define the problem as a binary classification problem. c indicates the label category, where $c = 1$ indicates that the code is a vulnerable code, and $c = 0$ indicates that it is a secure code. We train a model $f(.)$ to detect the label of a given code snippet s_i, where $f(s_i) = 1$ indicates that s_i is a vulnerable code, and $f(s_i) = 0$ indicates that s_i is a secure code. The second step is the classification of vulnerability types. We define the problem as a multi-classification problem. $c = \{i_1, i_2, ...i_{11}\}$ represents different types of vulnerabilities. We train a multi-classification model $g(.)$ to detect the corresponding post content w_i. where $g(w_i) = i_n$ indicates that vulnerability type of s_i is i_n.

3. Methodology

In this section, the paper will introduce the source code snippets vulnerability mining system PyVul for the open-source community. The system is mainly composed of four modules. The high-level design of PyVul is illustrated in Fig. 1.

Data Collection and Preprocess crawls posts content from the open-source community and cleans them, then uses the web tags to separate the source code and content in the post. *Code Feature Extraction* uses AST and data dependence to extract code feature. *Vulnerability Code Detection* uses code feature and BiLSTM algorithm to build a vulnerability classifier for code detection. If the code is benign, outputs the security code. Otherwise, further, analyzes the associated context. *Vulnerability Type Classification* extracts associated text feature and combine with the OVO SVMs algorithm to build a type classifier for vulnerability type classification. Finally, the system gives the vulnerability types of code snippet to complete vulnerability mining.

3.1. Data collection and preprocess

To collect data from the open-source community, a crawler has been designed and developed in this paper. In practical application, we found that different pages have different structures, so we adapted them. In the open-source community, user interaction is based on threads. Users create topic posts in the community, ask questions, and then other users reply. In data collection, crawling is also carried out according to the above organization, and all threads in the forum are obtained first. Then, according to the thread, collect all the reply information under it. For the next step of the better analysis, We use $< code >< /code >< p >< /p >$ tags to separate the code and content from the post and store them.

The existing open-source community has some specific protection mechanisms. Users who do not login in can only view part of posts. At the same time, to prevent tracking analysis, many websites have some anti-crawler mechanisms. In order to evade these, the crawler adds the mechanisms of simulated login, random request header, and dynamic IP proxy pool.

To better carry out the next step of text analysis, we preprocess the data. Preprocessing includes five aspects: case conversion, word segmentation, stop word removal, punctuation removal, and lemmatization. First of all, all the data are converted to lowercase to keep the data format consistency. NLTK [17] is used to deal with word segmentation and stop words to ensure its accuracy. At the same time, remove the nontext data such as punctuation in the data. Finally, NLTK is used to restore word form to reduce the phenomenon of multiple words with one meaning and reduce text redundancy.

3.2. Code feature extraction

The source code crawled from the open-source community is more chaotic and has no uniform format. There may not be any semantic relationship between these code. If the unprocessed code is directly used in the machine learning model's training, it may increase a lot of overhead. Therefore, we introduce AST technology to obtain intermediate structure to extract code feature. The specific steps are as follows:

3.2.1. Source code normalization

There are many comments in the source code extracted from the open-source community, these are not helpful for vulnerable code mining. Therefore, we use Python regular expressions to remove. Simultaneously, We found that the source code indentation in some posts is incorrect and cannot be analyzed. However, there may be some vulnerabilities in these codes, so it is necessary to format them. We use the Python code formatting tool: Autopep8 to solve this problem. Autopep8 formats Python program code according to the specification and eliminate errors caused by indentation. For code snippets that still cannot construct AST after normalization, such as command line and non-Python code. we remove it without analysis.

3.2.2. API / Library

API and Library refer to the module imported by the source code, which is reflected in the code as *import* and *from.....import* statements. In Python code, many library functions have some vulnerabilities [14]. When developers import these library functions, they may bring these vulnerabilities into their code. Therefore, it is essential to extract the library dependency from the code. Similarly, the order in which libraries are imported may be different. Therefore, we sort the libraries after extracting them to unify them and reduce the inaccuracy of the model caused by code style differences.

3.2.3. Data flow

Code feature extraction is a difficult problem in static analysis. At present, the mainstream and effective method is based on data flow [18]. In the analysis process, we found that the leading cause of source code vulnerability is that malicious parameters are not filtered and directly passed into dangerous functions. Through the data flow method, we can extract the parameters and functions that lead to code defects. Other code statements that are not related to the

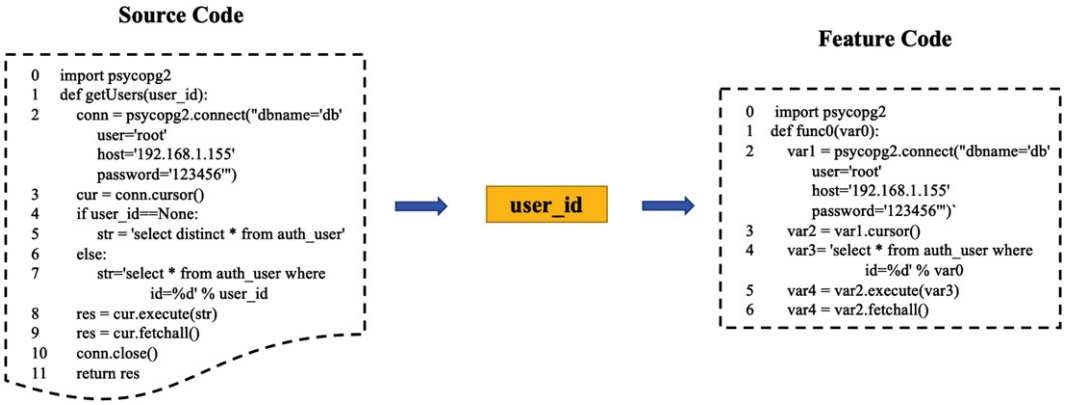


Fig. 2. Comparison of source code snippet and code feature.

vulnerabilities, such as assignments and definitions, will be ignored. When extracting the code's data flow, we mainly use AST [19] (abstract syntax trees). AST can parse program source code into a syntax tree, so developers can modify the syntax tree directly to change code. AST module provides the possibility for Python source code security check, program debugging, and syntax analysis.Data flow is the process of function parameter passing. Function parameters are passed to another variable through assignments and other operations, and so on, and finally output. Through data flow to find the input parameters path, then extract it and recombine a new function. Remove the lines of code without parameter conversion.When there are multiple input parameters in the function body, it may generate multiple data flows. To facilitate analysis, we merge the multiple data flows. In this paper, the AST module is used to construct the abstract syntax tree of the source code snippet. Considering the input as an origin point, tracking the variables and functions entered by the input parameters, and then extracting these parameters trajectory as the suspicious code.

3.2.4. Variable substitution

Since the source code in the post comes from different developers, there are big differences in code habits. If it is directly input into the neural network for training, there will be some errors. So in this step, some user-defined variables need to be replaced. The variables have two main parts. The first part of the variable is the function name: the user-defined function name in the extracted function body, such as *getUsers*. For the replacement of function names, use $funcN(N = 0, 1, ...n)$ to replace; the second part is the replacement of user-defined variables, such as *user_id*, for the replacement of these variables use $varN(N = 0, 1, ...n)$ to replace. In this way, errors caused by differences in user code can be eliminated.

3.2.5. Code recombine

After extracting the import library and data dependent code and completing all variable replacement, the next step is recombining them. This paper uses a unified format to sort the dependencies at the source file's head and then reorganize the code parts according to the original order. The purpose of this is to maintain the structural and semantic relationships between code contexts. As shown in Fig. 2, there is a comparison of source code snippet and code feature.

3.3. Vulnerability code detection

3.3.1. Data annotation

This paper first uses the tool to label once and then manually label and review some samples. We define two labels of source code: "1" (vulnerable code) and "0" (secure code). Here we use the Python code audit tool Bandit to complete automatic annotation. The Bandit [20] tool can find vulnerabilities in Python source code. During the detection process, Bandit parses the source code, generates its abstract syntax tree, and then performs security checks for the abstract syntax tree. After completing the AST scan, Bandit will directly generate code security reports for users. First of all, using Bandit to complete the first annotation will produce many positive samples and negative samples. In the actual labeling process, we found that the positive samples accounted for the majority, and the negative samples were few. For this reason, it is necessary to label the samples marked as

"0" manually and then label them again to find some missing and unrecognized vulnerable code.

3.3.2. Source code vectorization

The input received by the neural network is vectorized data, which cannot directly train the source code. Therefore, it is necessary to vectorize the preprocessed code feature.

Because in vulnerable code, every character may have crucial semantic information. Therefore, we need to divide the source code snippet into tokens, including keywords, identifiers, operators, and symbols. For example, the source code for the symbolic representation is as follows:

$$var4 = var2.execute(var3) \tag{1}$$

Lexical analysis is expressed as tokens as follows:

$$var4, =, var2, ., execute, (, var3,) \tag{2}$$

In order to convert the above token into a vector, the Tokenizer method in Keras is used to vectorize it. We found that the average number of tokens in the training data is 107, and 93% of the data are less than 200 tokens. Therefore, we use 0 to fill in the data less than 200, and truncate the data more than 200.

3.3.3. Vulnerability detection model based on BiLSTM

The neural network is very successful in image processing, speech recognition, and natural language processing [21–23], which are different from vulnerability detection. This means that many neural networks are not suitable for detecting vulnerabilities.

RNN [24] (Recurrent Neural Network) is one of the most commonly used models when using deep learning to deal with sequence problems. The current output of each sequence in the model is related to the previous output. RNN will take the t-1 time slice's hidden node as the input of the current time slice at the time t. Then it is applied to the calculation of the current output, and the nodes between the hidden layers are no longer unconnected but connected. The output of the traditional model's hidden nodes only depends on the input of the current time slice, so RNN has a better effect on sequence data processing. However, RNN can only memorize the last state information and cannot handle long-term dependent information well. Sundermeyer and others [25] proposed the LSTM for language modeling. LSTM (Long Short-Term Memory) is a variant of RNN. It adds a line to the RNN to express the long-term dependence of the input information. Therefore, LSTM has excellent advantages in modeling time sequence data.

The key of the LSTM model is to add the concept of "gate". The gate structure contains a sigmoid function, which takes a value between 0 and 1, as shown in Formula 1. Any value multiplied by 0 is 0. In this way, the value is "forgotten". Any number multiplied by 1 is the original value. Therefore, the value will be "kept". Important information can be retained, and unimportant information can be forgotten through the gate.

$$\sigma(t) = \frac{1}{1 + e^{-t}} \tag{3}$$

However, LSTM model can only carry out forward calculation. Due to the particularity of Python source code, function definition can be pre or post set, so it is necessary to calculate backward propagation. BiLSTM [26] (Bi-directional Long Short-Term Memory) is a combination of forward LSTM and backward LSTM. BiLSTM can learn the characteristics of serialization and long-term dependency and capture the implicit dependency between sequences.

As shown in Fig. 3, in the Forward layer, the forward calculation is performed from time 1 to time t, and the output of the forward hidden layer at each time is obtained and saved. In the Backward layer, the backward calculation is performed from time t to time 1, and the output of the backward hidden layer at each time is obtained and saved. Finally, combine the Forward layer and the Backward layer's output results at the corresponding time to get the final output. The mathematical expression is as follows:

$$h_t = f(w_1 x_t + w_2 h_{t-1}) \tag{4}$$

$$h_t^{'} = f(w_3 x_t + w_5 h_{t+1}^{'}) \tag{5}$$

$$o_t = g(w_4 h_t + w_6 h_t^{'}) \tag{6}$$

In the paper, our model network structure includes the embedding of the Token sequence, the BiLSTM layer, and then use the Dropout layer to randomly disconnect some neurons to prevent overfitting, and the Dense layer to match the previous layer The neural network is fully connected to achieve the nonlinear

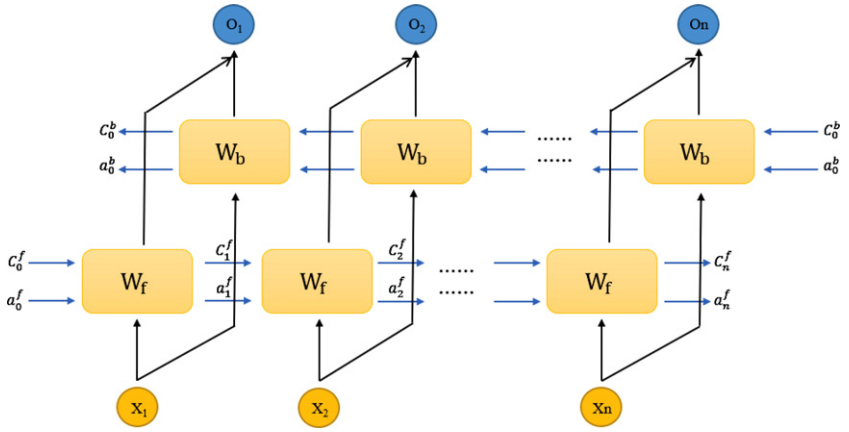


Fig. 3. BiLSTM structure diagram.

combination of features, and then the BatchNormalization layer is added to solve the problem of gradient disappearance and explosion. Next, the Relu function is used to activate the neurons, and after the Drop, Dense and BatchNormalization layers are passed again, since vulnerability detection is essentially a binary classification problem, we use the Sigmoid function for activation output, aiming to realize the detection of vulnerabilities.

3.4. Vulnerability type classification

In source code snippets vulnerabilities mining, we identify whether the code snippet is safe, but we cannot judge the vulnerability type. It is also essential to judge the types of code vulnerabilities. It is helpful to patch the vulnerability further.

In this part, we first preprocess the text of all the posts collected, then extract the text feature to feature representation, and finally use the OVO SVMs algorithm to analyze each post to get its related vulnerability type [27], and then associate the vulnerable code involved in the post with the vulnerability type extracted from the text, so as to obtain the vulnerable code and its vulnerability type.

3.4.1. Text preprocess

Text data can not be directly used for OVO SVMs model training, so it is necessary to extract text feature. At present, when extracting text feature, the commonly used method is to set the feature threshold according to the feature vector of words in the text, select the best feature as the text feature subset, and establish the text feature model.

There are many methods for extracting text feature, and word frequency weight is still the most effective method. There are generally many useless words and symbols in text data. Therefore, it is necessary to preprocess the text, including case conversion, removal of non-ASCII codes, word segmentation, removal of stop words, removal of punctuation, stemming, and Lemmatisation. Then count the word frequency to calculate its weight and sort from the largest to the smallest. Sort out a new sequence, and then extract the first N words with the highest weight.

In summary, the extraction steps of text feature are as follows:

i First, we need to preprocess the text data to remove irrelevant content.
ii Calculate and sort all the words and their frequency in each category document, and then filter and delete the useless words.
iii Calculate the word frequency of the words in each category and use the previous N words with the highest frequency as the category's feature word set.
iv Merge the feature word sets of all categories and delete the duplicate words from them. The final word set is the feature set to be used later, and then select the feature used in the test set.

3.4.2. Feature extraction and representation

TF-IDF [28] (term frequency-reverse document frequency) is a statistical method to evaluate the importance of words in document sets or corpus documents. If a word appears in the corpus with very low frequency, but it appears in the document frequency is very high, it indicates that the word is essential for the

document, has a strong distinction, and can be used for classification. On the contrary, it indicates that the word is not highly differentiated and is not suitable for text classification. TF-IDF algorithm is used to express text feature. The formula is as follows:

$$w_{ik} = \frac{tf_{ik} * \log(\frac{N}{nk} + 0.01)}{\sqrt{\sum_{i=k}[tf_{ik} * \log(\frac{N}{nk} + 0.01)]^2}} \quad (7)$$

It can be seen from the above formula that the higher the frequency of words, the lower the discrimination. On the contrary, the lower the frequency, the greater the discrimination and the weight. Therefore, we need to select words with high discrimination to better the classification effect in selecting text feature.

The primary purpose of normalization is to prevent the model from being biased into files, resulting in model errors. The formula is as follows:

$$\frac{a - min}{max - min} = b \quad (8)$$

In the formula, a is the word frequency of the keyword, min is the minimum word frequency of the word in all texts, and max is the maximum word frequency. This step is standardization. When comparing word frequency, there may be a big deviation. Normalization can make the classification of text data more accurate.

3.4.3. Vulnerability type classification model based on OVO SVMs

SVM [29] algorithm has exceptional advantages in text classification. It maps data to high-dimensional feature space through nonlinear transformation and classifies data in high-dimensional space with linear discriminant function, avoiding dimension disaster. The algorithm's complexity is independent of sample dimension; Its mathematical form is simple, geometric interpretation is intuitive, less manual setting parameters, easy to understand, and use.

Because of the advantages of SVM in performance, SVM is widely used in text classification. In two classifications, the SVM algorithm tries to find a hyperplane to separate two categories. As shown in Fig. 4, red and blue dots represent different categories. A is the hyperplane where they are divided, and the dotted line points are support vectors.

OVO(one-versus-one), referred to as 1-v-1 SVM [30], its core idea is to build multiple SVM classifiers, and any two samples build a classifier, then $K(K-1)/2$ classifiers need to be designed for K categories. When the sample is predicted, each SVM classifier will give a score, and the highest one is the prediction category of the sample.

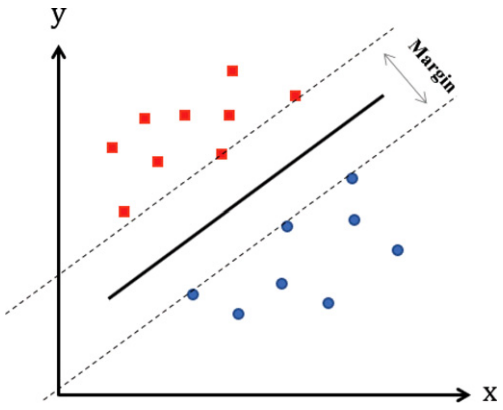


Fig. 4. Maximum interval classification diagram.

4. Experiments

4.1. Datasets

Currently, there is no open dataset for Python malware detection. Therefore, in the experimental process, all the data are collected and constructed by ourselves. The data source of the paper was the StackOverflow open-source community. We collected data through the Security and Python keywords retrieval method and collected 30,924 topic posts and 51,436 replies. Simultaneously, in order to train the vulnerability detection model and the classification model of code vulnerability types, the data are filtered and labeled. There are 5,372 data and 11 categories in text data annotation. There are 5,000 training data labels in the code feature, including 2,300 vulnerable codes and 2,700 security codes.

4.2. Evaluation methods

Evaluation indicators are significant for machine learning evaluation. Different machine learning models generally need to build different evaluation systems, and the same model also has different evaluation indicators. Each indicator has a different focus, such as classification, regression, clustering, topic model, etc. At the same time, various indicators can be used for the comprehensive evaluation of different models. In this paper, the following indicators are used for evaluation.

Table 1
Python vulnerability type and description

Type	Description
Command Execution	The attacker uses the vulnerability to make the system execute malicious commands to achieve the purpose of attack. For example: os.system,os.popen.
SQL Injection	The attacker inserts malicious code into the requested URL or form to trick the server into running the SQL command.
Cross Site Script	The attacker inserts malicious code into the web page containing the vulnerability, and when the user loads the web page, it will execute the Trojan horse program, thus causing the attack.
Cross-Site Request Forgery	Without the user's knowledge, forgery information requests other websites to achieve the purpose of attack.
File Upload	A file upload vulnerability means that an attacker uploads an executable to the server and executes the file. The file can be a Trojan horse, virus or webshell, etc.
Directory Traversal	Using the website security verification flaw or the user request authentication flaw to list the server directory vulnerability utilization way.
HardCode	In the source code use plaintext password or user name and other related privacy information.
Insecure Encryption	Using unsafe MD5, SHA1, ARC, DES and other algorithms in calculating hash or encryption.
Insecure Connection	The insecure connection in Python mainly uses HTTP without TLS.
File Permission	The arbitrary operation of files is mainly caused by the high permission given to the files.
Deserialization	Deserialization converts serialized data back to variables and program objects in memory.

ROC: The receiver operating feature curve is a comprehensive indicator reflecting the machine learning model. The larger the area under the curve is, the better the model is. Each point on the curve reflects the sensitivity to the same signal stimulus.

Confusion Matrix: Also called the possibility table or error matrix. It is a specific matrix used to visualize algorithm performance, usually supervised learning (unsupervised learning, usually using matching matrix). Each column represents the predicted value, and the confusion matrix can be used to represent each category's recognition rate intuitively.

Precision:

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

Recall:

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

F1-Score:The harmonic average of precision and recall, its value approaches the minimum of Precision and Recall, as shown in the formula:

$$F1 = \frac{2 * precision * recall}{precision + recall} \tag{11}$$

True Positive: Positive samples predicted to be positive by the model.

True Negative: Negative samples predicted to be negative by the model.

False Positive: Negative samples predicted to be positive by the model.

False Negative: Positive samples predicted to be negative by the model.

4.3. Experimental setups

To evaluate the vulnerability detection model, we conducted experiments using a Ubuntu server with a 4-core 3.6 GHz Intel Core i7-7700 processor, 6GB GeForce GTX 1060 graphics processing unit (GPU), and 16GB memory.

Vulnerability code detection model: In the vulnerability code detection experiment, we randomly shuffle the dataset, with 50% of the training set, 20% of the test set, and 30% of the verification set. To prevent overfitting during training, we set DROPOUT_RATE to 0.3. Simultaneously, to verify the effect of the model, we use the ten-fold crossover algorithm for verification.

Vulnerability type identification model: in all categories of text data, we compared the Random Forest algorithm[31], which is composed of multiple decision trees and is widely used in the field of multi-classification. In the experiment, we tested 11 common Python vulnerability types, as shown in Table 1.

4.4. Result & analysis

In the vulnerability detection model, to better analyze the model's performance, we build the ROC curve and the curve of accuracy and loss rate. As shown in Fig. 5 is the ROC curve of the vulnerability code detection model. It can be found that the area under the curve reaches 0.9068, which proves that the model has good performance.

It can be seen from Fig. 6 that during the training process, the loss rate continues to drop, from 0.8 to the

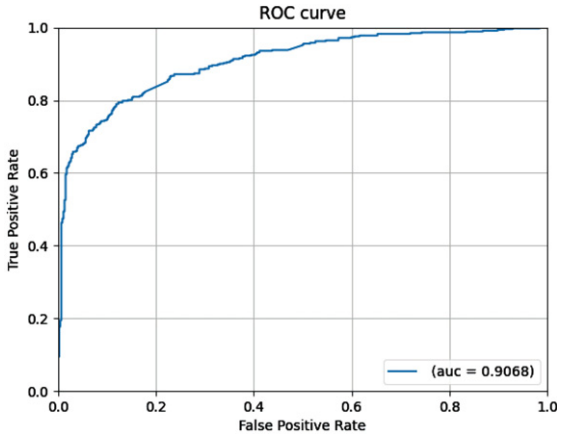
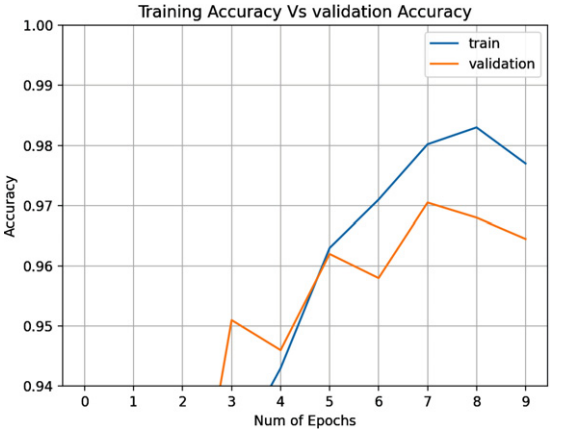
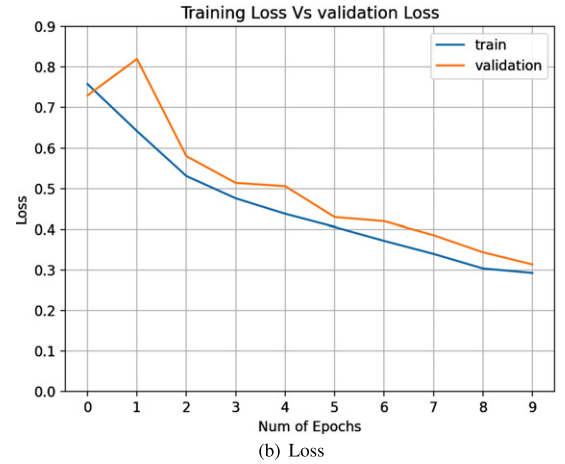


Fig. 5. ROC curve of the vulnerability code detection model.



(a) Accuracy



(b) Loss

Fig. 6. Accuracy and loss rate curve of the training set and validation set.

final 0.3, and the accuracy rate is also rising. Finally, the accuracy and loss rate on the validation set and training set tend to be consistent.

Because whether the code contains vulnerabilities may depend on the context, a neural network that can handle the context may be suitable for vulnerability detection. Many existing papers use RNN and LSTM algorithms for vulnerability detection [32–34]. The experimental results are shown in the Table 2, whether it is Precision, Recall, F1-Score, the BiLSTM algorithm results are optimal. Simultaneously, to verify the performance of the model, we carried out a ten-fold cross validation experiment. The average accuracy of the model was 84.35%. This shows that our model has good performance and strong generalization ability.

In the vulnerability type classification experiment, we compare the Random Forest algorithm with the OVO SVMs algorithm, as shown in Fig. 7 and Fig. 8, respectively, showing the confusion matrix of the two algorithms. The average accuracy of Random Forest is 87.5%, and the accuracy of OVO SVMs is 89.08%.

As can be seen from the figure, both algorithms can achieve good results in identifying vulnerability types. In the classification of XSS, CSRF, Command Execution, SQL injection, Deserialization, File Upload, Insecure Encryption, and Insecure Connection types, the two models can reach about 94%, in Directory Traversal and HardCode two types of classification. The OVO SVMs algorithm is better than the Random Forest algorithm.

To further analyze and compare the model's recognition ability for different vulnerabilities, we made a histogram about the accuracy, recall, and F1-score indicators. It can be seen from Table 3 that the average accuracy of the model can reach about 89%. The classification accuracy of most vulnerability types is above 90%.

In order to verify the actual effect of the model. We deploy the PyVul system to detect Python posts on StackOverflow in real-time. Not surprisingly, we found much vulnerable code. Table 4 shows some examples of vulnerable code that we found from StackOverflow. At the same time, these posts have a great impact. For example, the post with question_id 23739832 contains a command execution vulnerability code. The post has been viewed more than 1,000 times and has been widespread. The post with question_id 49308355 contains the vulnerability of too high file permissions. It has been read more than 8,000 and has been adopted. Meanwhile, we checked and found that these posts are not in our dataset. It

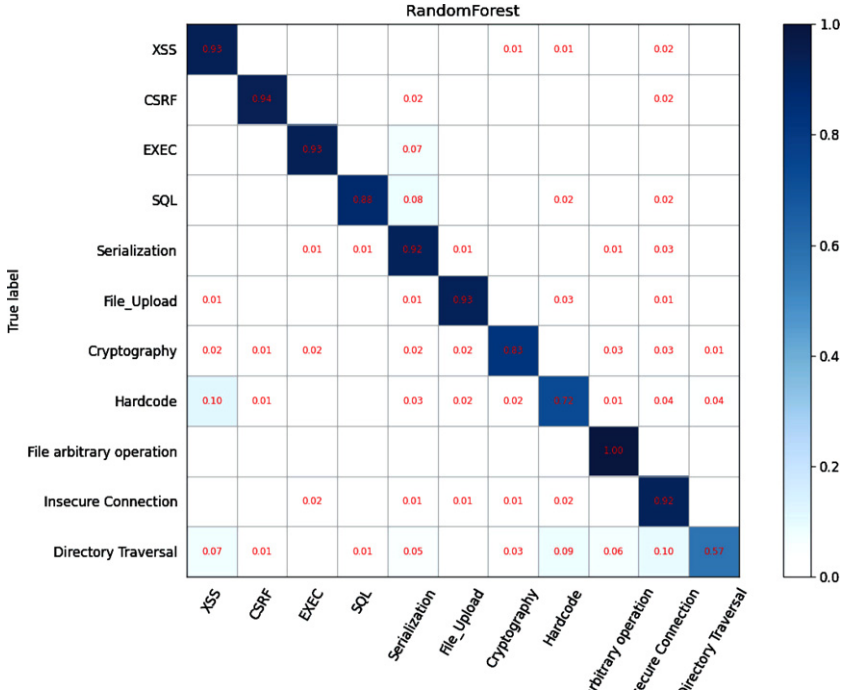


Fig. 7. Confusion matrix of Random Forest algorithm.

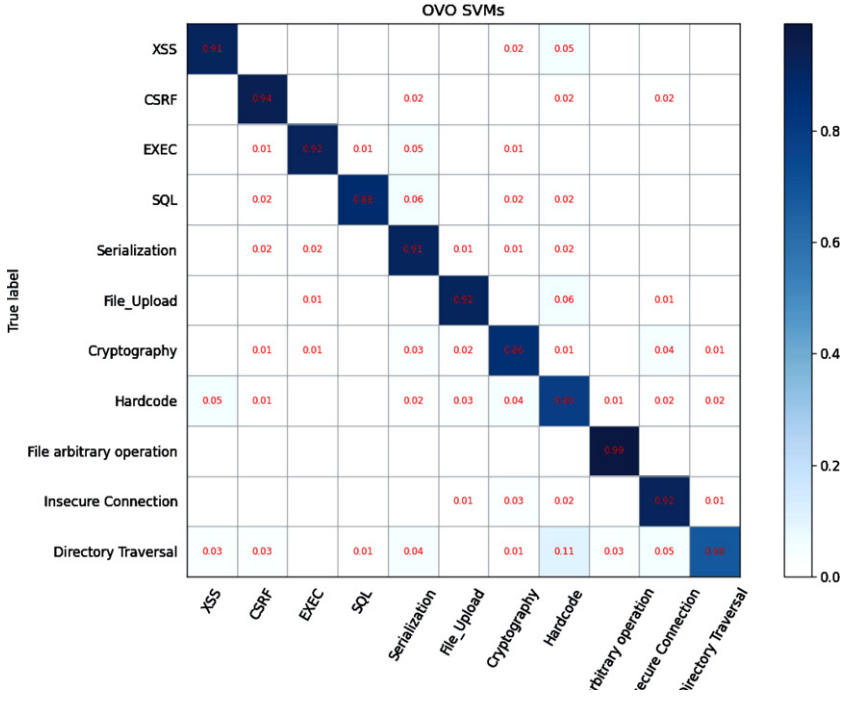


Fig. 8. Confusion matrix of OVO SVMs algorithm.

Table 2
Experiment results of comparison algorithms

Algorithm	Precision	Recall	F1-Score
RNN	0.7734	0.7581	0.7342
LSTM	0.8056	0.8073	0.8061
BiLSTM	**0.8547**	**0.8546**	**0.8546**

Table 3
Vulnerability type classification model evaluation

Type	Precision	Recall	F1-Score
Command Execution	0.94	0.92	0.93
SQL Injection	0.96	0.88	0.91
Cross Site Script	0.94	0.91	0.92
Cross-Site Request Forgery	0.91	0.94	0.93
File Upload	0.90	0.92	0.91
Directory Traversal	0.94	0.68	0.79
HardCode	0.73	0.80	0.76
Insecure Encryption	0.84	0.86	0.85
Insecure Connection	0.85	0.92	0.88
File Permission	0.97	0.99	0.98
Deserialization	0.82	0.91	0.86

shows that our system has excellent accuracy and practical value.

5. Conclusion

This paper builds a vulnerability mining system named PyVul for the open-source community's source code snippets. It includes a vulnerability mining model based on deep learning BiLSTM neural network and OVO SVMs algorithm. The open-source community's source code is relatively complex, and many of them are not complete code. Therefore, the paper first construct a source code preprocessing framework and uses the method based on data flow to extract the code feature. Simultaneously, the differences of user-defined are eliminated based on ensuring the code structure and semantics remain unchanged. In vulnerability mining, A detection model based on the BiLSTM algorithm was constructed. Compared with RNN, this model can memorize more semantic information. Simultaneously,

Table 4
Some unsafe posts detected by PyVul

Question_ID	Type	Key Code
23739832	Command Execution	p = subprocess.Popen(command, shell=True, stdin=subprocess.PIPE) p.communicate(password)
21106177	SQL Injection	username = request.GET('username') connection.execute("SELECT id,name,email FROM user WHERE use rname=%s LIMIT 1", (username,))
26204473	Cross Site Script	xss_url = 'http://www.foo.bar/index.php?ids="><sCrIpT>alert('XSS')< /ScRiPt>' r = requests.get(xss_url);
5100539	Cross-Site Request Forgery	xhr.setRequestHeader("XCSRFToken", getCookie('csrftoken'));
20473572	File Upload	up_file = request.FILES['file'] destination = open('/Users/Username/' + up_file.name, 'wb+')
3964681	Directory Traversal	param = request.GET.get('param') startdir = os.path.abspath(os.curdir) requested_path = os.path.relpath(param, startdir)
56662470	HardCode	APP_GUEST_USERNAME = "ppams.asguest" APP_GUEST_PASSWORD = "ppams123456"
35403878	Insecure Encryption	iv = ciphertext[:AES.block_size] cipher = AES.new(key, AES.MODE_CBC, iv) plaintext = cipher.decrypt(ciphertext[AES.block_size:])
29591313	Insecure Connection	r = requests.get('http://api.steampowered.com/IPlayerService/GetOwned Games/v0001/', params=steaminfo)
49308355	File Permission	for file in files: os.chmod(file, 0o0777)
3006727	Deserialization	zf = zipfile.ZipFile('zipped_pickle.zip', 'r') sd1 = cPickle.loads(zf.open('data.pkl').read())

the paper construct a vulnerability type classification model based on the OVO SVMs algorithm for the context discussion contents in associated threads to determine the vulnerability type of the code snippets. In the experiment, we use ten-fold cross validation to find that the BiLSTM model and OVO SVM have significant advantages in vulnerability mining. At present, The PyVul can only detect whether the code contains vulnerabilities, but it can not locate the vulnerabilities. The main reason is that the granularity of the code is different. In the future, we will also make efforts to achieve locating vulnerabilities.

Acknowledgments

This research is funded by the National Natural Science Foundation of China (No. 61902265), National Key Research and Development Program of China (No. 2016QY13Z2302, No. 2018YFB 0804103), Sichuan Science and Technology Program (No. 2020YFG0047) and Guangxi Key Laboratory of Cryptography and Information Security (No. GCIS201921).

References

[1] F. Fischer, K. Böttinger, H. Xiao, C. Stransky, Y. Acar, M. Backes and S. Fahl, Stack overflow considered harmful? the impact of copy&paste on android application security, in: *2017 IEEE Symposium on Security and Privacy (SP), IEEE*, (2017), 121–136.

[2] D. van der Linden, E. Williams, J. Hallett and A. Rashid, The impact of surface features on choice of (in) secure answers by Stackoverflow readers, *IEEE Transactions on Software Engineering* (2020), 1–1.

[3] S. Baltes and S. Diehl, Usage and attribution of Stack Overflow code snippets in GitHub projects, *Empirical Software Engineering* **24**(3) (2019), 1259–1295.

[4] G. Antal, M. Keleti and P. Hegedűs, Exploring the Security Awareness of the Python and Java Script Open Source Communities, in: *Proceedings of the 17th International Conference on Mining Software Repositories* (2020), 16–20.

[5] M.R. Rahman, A. Rahman and L. Williams, Share, But be Aware: Security Smells in Python Gists, in: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE*, (2019), 536–540.

[6] J. Kim, Y. Yoon, K. Yi, J. Shin and S. Center, ScanDal: Static analyzer for detecting privacy leaks in android applications, *MoST* **12**(110) (2012), 1.

[7] S. Lee, J. Dolby and S. Ryu, HybriDroid: static analysis framework for Android hybrid applications, in: *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE*, (2016), 250–261.

[8] W.R. Bush, J.D. Pincus and D.J. Sielaff, A static analyzer for finding dynamic programming errors, *Software: Practice and Experience* **30**(7) (2000), 775–802.

[9] D. D'Souza, Y.P. Kim, T. Kral, T. Ranade and S. Sasalatti, Tool evaluation report: Fortify, (2014).

[10] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi and J. Hu, VulPecker: an automated vulnerability detection system based on code similarity analysis, in: *Proceedings of the 32nd Annual Conference on Computer Security Applications* (2016), 201–213.

[11] O. Karnalim, Syntax trees and information retrieval to improve code similarity detection, in: *Proceedings of the Twenty-Second Australasian Computing Education Conference*, (2020), 48–55.

[12] S. Peng, P. Liu and J. Han, A Python security analysis framework in integrity verification and vulnerability detection, *Wuhan University Journal of Natural Sciences* **24**(2) (2019), 141–148.

[13] M. Alfadel, D.E. Costa and E. Shihab, Empirical Analysis of Security Vulnerabilities in Python Packages, in: *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, (2021).

[14] M. Sun, D. Gu, J. Li and B. Li, Pyxhon: Dynamic detection of security vulnerabilities in python extensions, in: *2012 IEEE International Conference on Information Science and Technology, IEEE*, (2012), 461–466.

[15] M.A. Prabakar, M. Karthikeyan and K. Marimuthu, An efficient technique for preventing SQL injection attack using pattern matching algorithm, in: *2013 IEEE International Conference ON Emerging Trends in Computing, Communication and Nanotechnology (ICECCN), IEEE*, (2013), 503–506.

[16] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng and Y. Zhong, Vuldeepecker: A deep learning-based system for vulnerability detection, *arXiv preprint arXiv:1801.01681* (2018).

[17] E. Loper and S. Bird, NLTK: the natural language toolkit, *arXiv preprint cs/0205028* (2002).

[18] Y. Fang, S. Han, C. Huang and R. Wu, TAP: A static analysis model for PHP vulnerabilities based on token and deep learning technology, *PloS One* **14**(11) (2019), e0225196.

[19] G. An, J. Kim and S. Yoo, Comparing line and AST granularity level for program repair using PyGGI, in: *Proceedings of the 4th International Workshop on Genetic Improvement Workshop*, (2018), 19–26.

[20] Bandit, https://github.com/PyCQA/bandit.

[21] H. Zheng, J. Fu, T. Mei and J. Luo, Learning multi-attention convolutional neural network for fine-grained image recognition, in: *Proceedings of the IEEE international conference on computer vision* (2017), 5209–5217.

[22] A. Graves, A.-r. Mohamed and G. Hinton, Speech recognition with deep recurrent neural networks, in: *2013 IEEE international conference on acoustics, speech and signal processing, IEEE*, (2013), 6645–6649.

[23] K. Hashimoto, C. Xiong, Y. Tsuruoka and R. Socher, A joint many-task model: Growing a neural network for multiple nlp tasks, *arXiv preprint arXiv:1611.01587* (2016).

[24] J. Schmidhuber, Habilitation thesis: System modeling and optimization, Page 150 ff demonstrates credit assignment across the equivalent of 1,200 layers in an unfolded RNN (1993).

[25] M. Sundermeyer, R. Schlüter and H. Ney, LSTM neural networks for language modeling, in: *Thirteenth annual conference of the international speech communication association*, (2012).

[26] T. Chen, R. Xu, Y. He and X. Wang, Improving sentiment analysis via sentence type classification using

BiLSTM-CRF and CNN, *Expert Systems with Applications* **72** (2017), 221–230.

[27] R. Scandariato, J. Walden, A. Hovsepyan and W. Joosen, Predicting vulnerable software components via text mining, *IEEE Transactions on Software Engineering* **40**(10) (2014), 993–1006.

[28] D. Wijayasekara, M. Manic and M. McQueen, Vulnerability identification and classification via text mining bug databases, in: *IECON 2014-40th Annual Conference of the IEEE Industrial Electronics Society, IEEE*, (2014), 3612–3618.

[29] B. Shuai, H. Li, M. Li, Q. Zhang and C. Tang, Automatic classification for vulnerability based on machine learning, in: *2013 IEEE International Conference on Information and Automation (ICIA), IEEE*, (2013), 312–318.

[30] A. Rocha and S.K. Goldenstein, Multiclass from binary: Expanding one-versus-all, one-versus-one and ecoc-based approaches, *IEEE Transactions on Neural Networks and Learning Systems* **25**(2) (2013), 289–302.

[31] M.S. Alam and S.T. Vuong, Random forest classification for detecting android malware, in: *2013 IEEE international conference on green computing and communications and IEEE Internet of Things and IEEE cyber, physical and social computing, IEEE*, (2013), 663–669.

[32] M. White, M. Tufano, C. Vendome and D. Poshyvanyk, Deep learning code fragments for code clone detection, in: *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE*, (2016), 87–98.

[33] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood and M. McConley, Automated vulnerability detection in source code using deep representation learning, in: *2018 17th IEEE international conference on machine learning and applications (ICMLA), IEEE*, (2018), 757–762.

[34] S. Liu, G. Lin, Q.-L. Han, S. Wen, J. Zhang and Y. Xiang, DeepBalance: Deep-learning and fuzzy oversampling for vulnerability detection, *IEEE Transactions on Fuzzy Systems* **28**(7) (2019), 1329–1343.