Detecting Vulnerability in Source Code Using CNN and LSTM Network

Junjun Guo (✉ guojunjun@xatu.edu.cn)

Xi'an Technological University

Zhengyuan Wang

Xi'an Technological University

Haonan Li

Xi'an Technological University

Yang Xue

Xi'an Technological University

Research Article

Keywords: Vulnerability Detection, Code Metrics, Convolutional Neural Network, Long Short-Term memory

Posted Date: April 5th, 2021

DOI: <https://doi.org/10.21203/rs.3.rs-174281/v1>

License: © ⓘ This work is licensed under a Creative Commons Attribution 4.0 International License.

[Read Full License](#)

Detecting Vulnerability in Source Code Using CNN and LSTM Network

Junjun Guo^{a*}, Zhengyuan Wang^b, Haonan Li^c and Yang Xue^d

*School of Computer Science and Engineering, Xi'an Technological
University, Xi'an, Shanxi, China*

^aguojunjun@xatu.edu.cn

^b1906210486@st.xatu.edu.cn

^c1906310531@st.xatu.edu.cn

^dxueyang@st.xatu.edu.cn

*Corresponding author

Abstract

Vulnerabilities can have very serious consequences for information security, with huge implications for economic, social, and even national security. Automated vulnerability detection has always been a keen topic for researchers. From traditional manual vulnerability mining to static detection and dynamic detection, all rely on human experts to define features. The rapid development of machine learning and deep learning has alleviated the tedious task of manually defining features by human experts while reducing the lack of objectivity caused by human subjective awareness. However, we still need to find an objective characterization method to define the features of vulnerabilities. Therefore, we use code metrics for code characterization, which are sequences of metrics that represent code. To use code metrics for vulnerability detection, we propose VulnExplore, a deep learning-based vulnerability detection model that uses a composite neural network of CNN+LSTM for feature extraction and learning of code metrics. Experimental results show that VulnExplore has a lower false positive rate, a lower miss rate, and a better accuracy rate compared to other deep learning-based vulnerability detection models.

Index Terms: Vulnerability Detection, Code Metrics, Convolutional Neural Network, Long Short-Term memory

1. Introduction

In the context of information technology, software are used in large quantities in various industries. Unfortunately, however, the problem of vulnerabilities cannot be completely avoided due to the vulnerabilities caused by programmers, either intentionally or unintentionally, and the high complexity of software.

Considering that vulnerabilities cannot be prevented from the root of generation, the main defense method is to detect vulnerabilities and patch them as early as possible, thus giving rise to the field of software vulnerability detection [1]. Due to the importance of the vulnerability issue, many researchers have conducted a lot of research on vulnerability detection.

The current solutions for vulnerability detection are mainly static detection, dynamic detection, machine learning, and deep learning approaches [2-8]. Many mature methods have been developed for static and dynamic detection, but these methods require a large amount of a priori expertise. Machine learning and deep learning-based methods have emerged in recent years and are rapidly evolving with the support of cloud computing and high computing power, making automated vulnerability detection a reality.

Recent research has used deep learning on code features to detect vulnerabilities in code. Unlike traditional detection methods, deep learning does not require researchers to manually perform extensive processing of code features and can automatically determine the relationship of features in the sample data from the training samples [9-16].

In order to explore the correlation between vulnerability code features and vulnerabilities, we propose a code metric-based vulnerability detection method VULNEXPLORE. VULNEXPLORE uses a code metric dataset to implement vulnerability detection using a long short-term memory network (LSTM), with the aim of exploring the implied relationship between different code metrics, while practically verifying that there is a positive relation between code metrics and the generation of vulnerabilities. Previous researchers have used CNNs or DNNs on code metrics for vulnerability detection, but these schemes only process and learn each feature and do not explore the relationship between each code metric.

As mentioned previously, code metrics can quantify the characteristics of vulnerabilities quite effectively, so in this study, we hope to explore the implicit relationship between different code metrics by building a deep learning model for the

purpose of vulnerability detection.

The contribution of our study is twofold:

As an important step in the study, we constructed a code metric dataset and make public this function granularity-based code metric dataset.

Aiming at the code metric dataset, we constructed a deep learning detection model to explore the relationship between code metrics and vulnerabilities, and also improved previous work of the researchers to make our vulnerability detection model better in terms of accuracy, recall, and F-measure of vulnerability detection.

Paper Organization. The rest of this paper is organized as follows.

Section II presents the relevant background and the most relevant studies, Section III describes the problem studied and the methodology of the study, Section IV describes the design of the experiment, Section V presents the results and a discussion of the experiments, Section VI discusses the limitations of the study methodology, and Section VII presents future work on the subject.

2. Background and related works

In this section, the background and other related work are briefly described. The background of the research is first presented, then the concepts related to our research (neural networks, vulnerability detection, and code metrics) are introduced, and in the end, other related articles are cited.

2.1 Deep Learning Neural Networks

Deep learning is a sub-branch of machine learning. There are well-established deep learning methods in image processing, medicine, natural language processing, and several other fields. Most deep learning models are now based on artificial neural networks (ANN), which are complex neural networks composed of artificial neurons (ANs) inspired by neurons in the human brain cortex [17]. The information processing of individual artificial neurons is quite simple, but the global behavior caused by the interaction of individual neurons in a neural network allows solving complex problems.

While neural networks have been very successful in such areas, it is not the same as in the vulnerability detection area. Source code is different from image processing and natural language processing, and many deep learning networks are not applicable to vulnerability detection, leading to the need to select an appropriate neural network for

the source code. In this paper, considering the use of code metrics to characterize source code, we use the convolutional neural network (CNN) for feature extraction of code metrics and long short-term memory network (LSTM) for vulnerability detection to explore the relationship between different code metric attributes and whether contextual semantic information is reflected between attributes.

Convolutional neural networks have excelled in the field of image processing. Hubel and Wiesel's research from the 1850s to the 1860s found that the visual cortex of monkeys and cats contains neurons that correspond separately to a small visual area [18]. When their eyes are fixed in an area that excites a single neuron, and adjacent neurons have similar receptive intervals. In order to form a complete image, the size and location of the receptive range of neurons on the whole visual cortex show systematic variations. With the above concept, convolutional neural networks are proposed. In the field of image processing, features of images such as edges, lines, and corners of images can be extracted. For one-dimensional features such as code metrics, we use asymmetric convolutional kernels (e.g., 1*2 or 1*3 convolutional kernel size) for feature extraction of the input data and then perform feature sampling through the pooling layer [19, 20].

Long Short-term Memory network (LSTM) is a variant of Recurrent Neural Network (RNN). In order to overcome the problem of gradient disappearance and gradient explosion during the training of long sequences of RNNs, LSTMs have better performance in training long sequences compared to normal RNNs. Since LSTM is explicitly designed to avoid the long-term dependency problem, remembering information for a long time is the default behavior of LSTM rather than the behavior that needs to be learned during the training process, and the transmission state is controlled by gating the state to selectively remember information.

2.2 Vulnerability Detection

In this paper, we choose C/C++ as the target languages for vulnerability detection. Since many of the underlying software and systems are written in C/C++, once these software vulnerabilities are exploited they can have very serious consequences.

Our vulnerability detection model is built on code metrics and deep learning. The model extracts the code metric of the target function, extracts the features of the code metric by a convolutional neural network, and then learns the feature vector by LSTM. The features of the source code are presented in the form of code metrics, and different code metrics have a different granularity of abstraction generalization of the

source code. The code metrics are further extracted by CNN so that these features can effectively characterize the not provided between the vulnerability code and the optimistic code. The LSTM can learn these features, and the long-term memory behavior of the LSTM network for features can bring out the features of positive and negative samples more to achieve the purpose of vulnerability detection.

Unlike the previous studies, our CNN+LSTM can better capture the relationship between individual code metrics and significantly improve the detection accuracy of vulnerabilities. We will describe the vulnerability detection model in Section 3.

2.3 Code Metrics

A code metric is a set of software metrics that characterize the nature and specification of the source code with the goal of obtaining objective, quantitative metrics. Some basic code metrics are the number of lines of code, the number of blank lines, the number of comment lines, the number of words, etc [21].

The McCabe metric is a complexity metric based on the program control flow proposed by Thomas McCabe, also known as the loop metric [22]. He argues that the complexity of a program depends heavily on the complexity of the program control flow graph. A single sequential structure is the simplest, and the more loops formed by loops and selections, the more complex the program is, and the higher the likelihood of vulnerabilities. After the problem of loop complexity is depicted as a control flow graph, the loop complexity can be calculated using any of the following three.

- (1) The circle complexity of the control flow graph $V(G) = R$, where R is the number of regions in the control flow graph.
- (2) The circle complexity of the control flow graph $V(G) = E - N + 2$, where E is the number of edges in the graph, and N is the number of nodes.
- (3) The circle complexity of the control flow graph $V(G) = P + 1$, where P is the number of decision nodes in the graph.

The Halstead complexity measures is a software metric method proposed by Halstead in 1977, who observed that software metrics should reflect the way algorithms are implemented in different software yet be independent of the platform and language used. These metrics can be calculated from the base metric of the source code. Based on the base metrics of the source code (number of operators η_1 , number of operators η_2 , number of occurrences of all operators N_1 , number of occurrences of all operators N_2), the code metrics that can be calculated are shown in Table 1 [23, 24].

Table 1 Halstead Metrics

Basic Metrics	Calculated Metrics
Operators	Program vocabulary
Operands	Program length
total number of distinct operators	Calculated program length
total number of distinct operands	Volume
	Difficulty
	Effort
	Time required to program
	Number of delivered bugs

3 Methodology

Our target is to design a vulnerability detection system based on the designed vulnerability detection model that can automatically determine the existence of vulnerabilities in software based on the source code, by simply doing code metric calculations on the given source code. In this section, we first describe the design of the model, then we present the main research questions based on our final goal to achieve vulnerability detection, and in the end, we elaborate on the modules in the model based on these questions.

3.1 Overview of the Model

This subsection is an overview of the model. The model has two phases: a training phase and a testing phase. In the learning phase, we construct the code metric dataset by extracting code metrics from a large number of source code files, some of which are vulnerable and some of which are clean. Our CNN+LSTM network is trained by labeling the vulnerable and clean data. At the same time, using cross-validation to evaluate the model.

3.2 Research questions and proposed methods

RQ1: Can code metrics be used as input features to deep learning models for vulnerability detection?

To answer this question, we will first describe the preparation of the dataset. The data used for training and testing comes from the public dataset of labeled code slices proposed by Li et al. This dataset mainly covers library/API calls, incorrect use of arrays, improper use of pointers, and incorrect use of arithmetic expressions (e.g., integer overflow vulnerabilities). The specific number of these four sections as well as the vulnerable and clean code pieces are shown in Table 2. In this dataset, the label '1' indicates that the code slices are vulnerable, and '0' indicates that the code slices

are clean. With this labeled code slice dataset, we selected 65513 of them and constructed a dataset containing 20 code metrics and 1 vulnerability annotation from these raw code slices data. Since the code slices in this original dataset are composed of only vulnerability-related code statements, we extracted the code metrics related to the lines of code metric as well as the maintainability metric and the number of vulnerabilities committed per line metric, while the code metrics related to the class granularity could not be computed [25].

The preparation of our dataset has two steps. First, parsing the public dataset of labeled code slices, computing the code metric for each code slice separately and labeling it as vulnerable or not. Then, eliminate the redundant data. We end up with a dataset that includes 65513 instances, using a CSV comma-separated value file.

Since many vulnerability detections are based on the assumption that the more complex the code, the more vulnerabilities, the metrics used to measure software quality can also be used as a representation of code characteristics. Younis et al. selected eight code metrics such as lines of code, circle complexity, degree of nesting, information flow, and call function to characterize the code and predict the exploitability of vulnerabilities by machine learning methods. The experimental results show that the selected code metrics can provide an effective feature representation of whether the vulnerability is exploited or not, and the F-Measure reaches 84% when wrapped hair is used for feature selection. Perl et al. propose the VCCFinder model, which uses the SVM model to identify vulnerabilities based on code metrics and metadata collected from the Github code repository. The detection results were improved by 59.9% when compared with the representative Flawfinder. In our study by selecting 20 code metrics to form a sequence of metrics that can summarize the code information, the various information of the code is represented by quantified program attributes, and the sequence of quantified representations is well suited for statistical analysis, and the features of the code can be significantly extracted by subsequent convolution, which is good news for deep learning networks.

Table 2 Description of the original dataset

Types	Number of vulnerable slices (1)	Number of Clean slices (0)
Library/API calls	13603	50800
Incorrect use of arrays	10926	31303
Improper use of pointers	28391	263450
Arithmetic	3475	18679

RQ2: Which model can use code metrics for feature extraction and vulnerability detection?

The purpose of this study is to investigate the detection accuracy when using code metrics as input to

a vulnerability detection model. Previous research methods based on code metrics or other forms of code representations such as Token, abstract syntax trees, and graphs have only used a single one such as DNN, LSTM, BiLSTM, etc. in the selection of neural networks, and some require the source code to be converted to a form acceptable to neural networks after extensive preprocessing such as through word embedding methods like word2vec.

We use the neural network structure of CNN+LSTM. Specifically, it includes an input layer, a convolutional layer, a pooling layer, an LSTM layer and a dense layer. Due to the good performance of CNN in image processing and natural language processing, its local perception and weight sharing can greatly reduce the number of parameters and thus improve the learning efficiency of the model. Since our code metrics dataset is a sequence of individual code metrics, with the good ability of CNN to abstract features, the information implied on the code metrics can be extracted and higher quality and high concentration features can be passed to the LSTM for vulnerability detection.

Our CNN consists of two main components: convolutional layer and pooling layer. Figure 1 shows the details of CNN.

Figure 1 CNN Model.

Since each instance in the code metric dataset is a one-dimensional sequence, we choose an asymmetric convolutional kernel to perform feature extraction on the sequence. After the convolutional layer performs the convolutional operation, although the features of the data are

extracted, the dimensionality of the extracted features is pretty high, so in order to solve the problem and reduce the cost of training the network, a pooling layer is added after convolution to select the features. Its calculation formula is as in Equation:

$$l_t = \tanh(x_t * w_t + b_t),$$

where l_t represents the output value after convolution, $\tanh$ as the activation function, x_t is the input sequence, w_t is the weight of the convolution kernel, and b_t is the bias of the convolution kernel. The pooling layer selects the maximum pooling. The output vector after convolution and pooling is as in Equation:

$$X_t = f_{\max\text{pooling}}\{\tanh(x_t * w_t + b_t)\},$$

Where X_t is the output after the CNN layer and $f_{\max\text{pooling}}$ is the max-pooling operation.

Then the output of the CNN is passed to the LSTM layer. the LSTM unit consists of forgetting gate, input gate, and output gate. As shown in Figure 2.

(a) Input the output value of the last moment and the input value of the current time to the forgetting gate, and calculate the output of the forgetting gate, as shown in the following formula:

$$f_t = \sigma(W_f \cdot [h_{t-1}, X_t] + b_f),$$

where the value of f_t is in the range (0, 1), W_f is the weight of the forgetting gate, h_{t-1} is the bias of the forgetting gate, X_t is the output value at the current time, and b_f is the input value at the last moment.

(b) Input the last output value and the input value at the current time to the input gate, and calculate to obtain the output value and the candidate cell state of the input gate, as shown in the following equation.

$$\begin{aligned} i_t &= \sigma(W_i \cdot [h_{t-1}, X_t] + b_i), \\ \tilde{C}_t &= \tanh(W_c \cdot [h_{t-1}, X_t] + b_c), \end{aligned}$$

where i_t has values in the range (0, 1), W_i is the weight of the input gates, b_i is the bias of the input gates, W_c is the weight of the candidate input gates, and b_c is the bias of the candidate input gates.

(c) Update the status of the current unit as follows:

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t,$$

where C_t is in the range (0, 1).

(d) Accept the output and input as the input value of the output gate at time t and obtain the output of the output gate as follows.

$$o_t = \sigma(W_o \cdot [h_{t-1}, X_t] + b_o),$$

where o_t is in the range (0, 1), W_o is the weight of the output gate, and b_o is the bias of the output gate.

(e) The final output value is obtained by calculating the output of the outputs and the state of the cell, as shown in the following equation.

$$h_t = o_t * \tan(C_t),$$

The output of LSTM is obtained by the above computational process, and finally, the output of the whole neural network is obtained by a layer of the dense layer.

Figure 2 LSTM unit

We can find that the features are extracted by the convolution and pooling of CNN, and the extracted features are transferred to LSTM for learning. With the feature of local feature attention of CNN and long short-term memory for solving the problem of gradient disappearance and gradient explosion, CNN+LSTM neural network model can be good for the extraction and learning of code metric features.

The experiments and results will be discussed in Section 4.

4 Experiments and Results

In this section, we describe the steps and results of the experiment in detail. To verify the validity of the model, we give the evaluation metrics. Then, we describe the process of data pre-processing and how to train the model. Finally, we use the model for vulnerability detection and compare the results with other methods and tools.

4.1 Evaluative indicators

A good vulnerability detection model should make as many correct detections as possible within the detection range and miss as few vulnerabilities as possible. Given the above purpose of vulnerability detection, we use five general and well-known evaluation metrics: accuracy, recall, false-negative rate, false-positive rate, and F-measure.

4.2 Data Preparation

As described in Subsection 3.2, the original dataset comes from the public dataset proposed by Li et al. The code slices in this dataset are from NVD [26] and SARD [27], where NVD contains flaws in real-world software applications and may also contain diff files before and after patching of the vulnerable code slices. SARD contains samples of vulnerabilities in real-world software applications and artificially constructed vulnerabilities and is classified as positive, negative, and mixed, i.e., patched [28]. Here we give a piece of code slice to demonstrate the exact process of computing the code metric.

```
struct task_struct *me = current;
static char lastcomm[sizeof(me->comm)];

if (strncmp(lastcomm, me->comm, sizeof(lastcomm))) {
    printk(KERN_INFO "IA32 syscall %d from %s not implemented\n", call, me->comm);
    strncpy(lastcomm, me->comm, sizeof(lastcomm));
}
return -ENOSYS;
```

A snippet of sliced code

This code slice is a fragment of code that has been processed to retain only the statements related to the vulnerability. As mentioned in RQ1 of subsection 3.2, in this study we propose 20 code metrics to characterize a piece of code, among which 8 code metrics can be extracted directly from the code fragment, namely: Empty lines, Lines of comments, lines of programs, physic lines, number of distinct operators (η_1), number of distinct operands (η_2), total number of distinct operators (N_1), total number of distinct operands (N_2). The above code snippet gives $\eta_1=21$, $\eta_2=8$, $N_1=45$, and $N_2=20$. The rest of the code metrics can be obtained by the above calculation. The formulae for these code metrics are as follows:

- Program vocabulary: $\eta = \eta_1 + \eta_2$,
- Program length: $N = N_1 + N_2$,
- Calculated estimated program length: $\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$,
- Volume: $V = N * \log_2 \eta$
- Difficulty: $D = (\frac{\eta_1}{2}) * (\frac{N_2}{\eta_2})$
- Effort: $E = D * V$
- Time required to program: $T = \frac{E}{18}$
- Number of delivered bugs: $B = \frac{V}{3000}$

·Maintainability: $M = \max(0, (171 - 5.2 * \ln(V) - 0.23 * CC - 16.2 * \ln(LOC)) * 100/171)$

100/171)

·Operation per Line: $O = \frac{N}{LOC} * \frac{1}{B}$

·And the Cyclomatic complexity(CC): $CC = e - n + 2p$, where e denotes the number of edges in the control flow graph, n denotes the number of nodes in the control flow graph, and p denotes the number of connected components of the graph (Usually, since control flow graphs are all connected, $p = 1$).

Through the above code metric formula, we can get a complete code representation to build a code metric dataset.

4.3 Experiments

We use the code metric dataset constructed in subsection 4.2 to train the neural network and find the best network model parameters. To evaluate the effectiveness of the model, we used a k-fold cross-validation technique to evaluate the model on the same benchmark. The dataset is divided equally and randomly into k subsets of the same size, so that each subset can be used as a test set in each iteration round, and the other subsets are used as training sets for training the model. For our dataset, 3-fold cross-validation works best.

4.4 Results and Discussion

In this section, we validate the effectiveness of the model by comparing it with other vulnerability detection methods. We choose the commercial detection tool Checkmarx, the open-source static analysis tool Flawfinder, the VCCFinder proposed by Perl et al, the VPM proposed by M. Zagane et al, and the state-of-the-art VulDeePecker for comparison. Checkmarx and Flawfinder are widely used in real-world applications as enterprise software security solutions. VCCFinder, VPM, and VulDeePecker represent the most advanced static analysis techniques for vulnerability detection in the current vulnerability detection field in terms of using code characterization as a data source. Tables 3 and 4 show the detection results and the results of whether our model has used the balanced dataset.

Table 3 Result of Prediction using different Models

Models	P (%)	R (%)	FN (%)	FP (%)	F1 (%)
Checkmarx	39.6	58.9	41.1	43.1	47.3
Flawfinder	25.0	31.0	69.0	44.7	27.7
VCCFinder	59.55	24.1	75.80	40.04	34.31
VPM	76.9	76.6	23.36	23.36	76.57

VulDeePecker	76.43	95.47	4.53	8.19	84.89
VULNEXPLORE	80.62	82.2	17.82	19.39	81.4

Table 4 Result of Prediction using balanced dataset or not

VULNEXPLORE	P (%)	R (%)	FN (%)	FP (%)	F1 (%)
Balanced	80.62	82.2	17.82	19.39	81.4
Unbalanced	80.8	85.8	82.1	85.5	79.3

We found that when using the unbalanced dataset, Precision and Recall were not very different from those using the balanced dataset, but the FPR and FNR were unusually high. We learned from the analysis that in the unbalanced dataset, the difference between the number of positive and negative samples is huge, with positive samples accounting for only 14.2% of the total samples, and such a large difference leads to very large detection indicators in the negative samples, while in the positive samples, except for the accuracy of 50%, all other indicators do not exceed 2%, which leads to higher than expected evaluation indicators after the final weighted average. This reminds us that it is necessary to use a balanced dataset.

Using the four indicators mentioned in subsection 4.1 to draw conclusions, first we find that our model is ahead of the other five detection tools and models in Precision. VULNEXPLORE, VPM and VulDeePecker are above 80% in Precision, Recall and F1-Measure, which is enough to show that the use of deep learning is effective in vulnerability detection. Moreover, the FNR and FPR of both VULNEXPLORE and VulDeePecker are below 20%, and the FNR and FPR of VPM are below 25%, while the FNR and FPR of Checkmarx and Flawfinder even reach 76% and 45%, which are much higher than the deep learning-based vulnerability detection models, due to the fact that Checkmarx and Flawfinder rely mainly on manually defined prior knowledge, which is prone to high false positives when the code features of positive and negative samples do not differ much. This indicates that deep learning is better at noticing details of code features in vulnerability detection than human experts.

5 Limitations

The current VULNEXPLORE has several limitations in the design, experimentation, and evaluation process. This also provides new ideas and directions for our future work. First, our code metrics dataset is based on a publicly available code slicing dataset, and the inability to compute code metrics for class granularity (e.g., class coupling, inheritance depth) due to code slicing makes the experimental dataset inevitably missing some information. In the future, we need to collect a large

amount of vulnerability source code data in order to investigate class-granularity code metrics. Second, we have currently only targeted four types of vulnerabilities in C/C++: library/API calls, incorrect use of arrays, improper use of pointers, and incorrect use of arithmetic expressions, and were unable to verify the generality of the model. The VULNEXPLORE implementation was then written in Python. Future consideration needs to be given to using other languages to improve the operational efficiency of the model.

6 Conclusion

In this study, we propose an improved composite neural network vulnerability detection model-VULNEXPLORE. It utilizes convolutional neural networks for feature extraction of code metrics, followed by long and short-term memory networks for learning the extracted features, and is able to detect vulnerabilities in code metrics. Experiments show that our model is effective, but there is room for further research.

We conclude that while it is possible to characterize vulnerable code using a single granularity or one code metric, this characterization is incomplete. Characterizing vulnerabilities by multi-scale (e.g., different granularity, control flow, code slicing, or even word-by-word comparison using sliding windows) code metrics may provide better detection performance.

We also conclude that the CNN+LSTM neural network model has better results in terms of processing of features than other single networks [29-30]. Combined with the previous conclusion, it would be an interesting topic to introduce scale pyramids in the future by borrowing the concept of image pyramids in the field of target recognition, combined with attention mechanisms or classical ML methods.

7 Future Works

For future work, we will first collect a large amount of vulnerability source code, and on this basis, we will study multi-scale code metrics to characterize the vulnerability source code at both shallow and deep levels, such as slicing the source code in conjunction with control flow to obtain semantic information about the vulnerability and use it as a code metric at the flow scale. On the other hand, we will continue to investigate deep learning-based vulnerability detection. Deep learning has been very effective in image processing, natural language processing, and vulnerability detection, vulnerabilities have many similarities with images and natural language, so we will further investigate neural network models for vulnerability detection to improve the effectiveness of vulnerability detection.

Declaration of Conflicting Interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) received no financial support for the research, authorship, and/or publication of this article.

Declarations

Confirm that all the research meets the ethical guidelines, including adherence to the legal requirements of the study country.

Authorship contributions

Junjun Guo: Conceptualization, Methodology.

Zhengyuan Wang: Data curation, Writing- Original draft preparation.

Haonan Li: Visualization, Investigation.

Yang Xue: Software, Validation

References

- [1] Krsul I V. Software vulnerability analysis [M]. West Lafayette, IN: Purdue University, 1998.
- [2] Checkmarx. <https://www.checkmarx.com>
- [3] Cppchecker. <http://cppcheck.sourceforge.net/>.
- [4] Cvechecker. <https://www.oschina.net/p/cvechecker>.
- [5] Jang J, Agrawal A, Brumley D. ReDeBug: finding unpatched code clones in entire os distributions[C]//2012 IEEE Symposium on Security and Privacy. IEEE, 2012: 48-62.
- [6] Li J, Ernst M D. CBCD: Cloned buggy code detector[C]//2012 34th International Conference on Software Engineering (ICSE). IEEE, 2012: 310-320.
- [7] Yamaguchi F, Wressnegger C, Gascon H, et al. Chucky: Exposing missing checks in source code for vulnerability discovery[C]//Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. 2013: 499-510.
- [8] Sajnani H, Saini V, Svajlenko J, et al. SourcererCC: Scaling code clone detection to big-code[C]//Proceedings of the 38th International Conference on Software Engineering. 2016: 1157-1168.
- [9] Guo N, Li X, Yin H, et al. VulHunter: An Automated Vulnerability Detection System Based on Deep Learning and Bytecode[C]//International Conference on Information and Communications Security. Springer, Cham, 2019: 199-218.
- [10] Li Z, Zou D, Xu S, et al. Sysevr: A framework for using deep learning to detect software vulnerabilities [J]. arXiv preprint arXiv:1807.06756, 2018.
- [11] Lin G, Zhang J, Luo W, et al. POSTER: Vulnerability discovery with function representation learning

from unlabeled projects [C]//Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. 2017: 2539-2541.

[12] Lin G, Zhang J, Luo W, et al. Cross-project transfer representation learning for vulnerable function discovery [J]. IEEE Transactions on Industrial Informatics, 2018, 14(7): 3289-3297.

[13] Russell R, Kim L, Hamilton L, et al. Automated vulnerability detection in source code using deep representation learning [C]//2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA). IEEE, 2018: 757-762.

[14] Perl H, Dechand S, Smith M, et al. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits [C]//Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. 2015: 426-437.

[15] Younis A, Malaiya Y, Anderson C, et al. To fear or not to fear that is the question: Code characteristics of a vulnerable function with an existing exploit [C]//Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy. 2016: 97-104.

[16] Zagane M, Abdi M K, Alenezi M. Deep Learning for Software Vulnerabilities Detection Using Code Metrics [J]. IEEE Access, 2020, 8: 74562-74570.

[17] Schmidhuber J. Deep learning in neural networks: An overview [J]. Neural networks, 2015, 61: 85-117.

[18] Hubel D H, Wiesel T N. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex [J]. The Journal of physiology, 1962, 160(1): 106.

[19] LeCun Y, Bottou L, Bengio Y, et al. Gradient-based learning applied to document recognition [J]. Proceedings of the IEEE, 1998, 86(11): 2278-2324.

[20] Hochreiter S, Schmidhuber J. Long short-term memory [J]. Neural computation, 1997, 9(8): 1735-1780.

[21] Jiang Y, Cuki B, Menzies T, et al. Comparing design and code metrics for software quality prediction [C]//Proceedings of the 4th international workshop on Predictor models in software engineering. 2008: 11-18.

[22] McCabe T J. A complexity measure [J]. IEEE Transactions on software Engineering, 1976 (4): 308-320.

[23] Halstead M H. Elements of software science [M]. New York: Elsevier, 1977.

[24] Wheeler D. Flawfinder home page [J]. Web page: <http://www.dwheeler.com/flawfinder>, 2006.

[25] word2vec. May 11, 2019, <http://radimrehurek.com/gensim/models/word2vec.html>.

[26] NVD. 2018, <https://nvd.nist.gov/>.

[27] Software assurance reference dataset. 2018, <https://samate.nist.gov/SRD/index.php>.

[28] Geisser S. A predictive approach to the random effect model [J]. Biometrika, 1974, 61(1): 101-107.

[29] Burt P J, Adelson E H. The Laplacian pyramid as a compact image code [M]//Readings in computer vision. Morgan Kaufmann, 1987: 671-679.

[30] Lan Z, Lin M, Li X, et al. Beyond gaussian pyramid: Multi-skip feature stacking for action recognition [C]//Proceedings of the IEEE conference on computer vision and pattern recognition. 2015: 204-212.

computer science and
Studies, Guangzhou, China
Xi'an Technological
detection and deep

E-mail:

Junjun Guo was born in Hengshan, Shaanxi, P.R. China, in 1977. He received the Ph.D. from Xidian University, P.R. China. Now, he works in School of computer science and engineering, Xi'an Technological University. His current research interests include program analysis, cyberspace security, machine learning and data mining.

E-mail: guojunjun@xatu.edu.cn

Zhengyuan Wang received the B.E. degree in technology from Guangdong University of Foreign in 2016, and is currently pursuing the master degree at University. Now his is interesting in vulnerability learning.

1906210486@st.xatu.edu.cn

Haonan Li was born in Baotou City, Inner Mongolia Autonomous Region of China. He has graduated from Chongqing University of Posts and Telecommunications in 2018, and is currently studying for his master degree at Xi'an Technological University. Now his is majoring in vulnerability detection and vulnerability risk assessment.

E-mail: 1906310531@st.xatu.edu.cn

Yang Xue was born in Shanxi province of China in 1996. He has graduated from Jilin University of Finance and Economics in 2020, and is currently studying for his master degree at Xi'an Technological University. Now his main interests focus on vulnerability detection, artificial intelligence.

E-mail: xueyang@st.xatu.edu.cn

Figures

Figure 1

CNN Model.

Figure 2

LSTM unit