Hindawi
Scientific Programming
Volume 2022, Article ID 4442374, 8 pages
https://doi.org/10.1155/2022/4442374

Research Article

Research on Software Vulnerability Detection Method Based on Improved CNN Model

Gao Qiang

Shandong Management University, School of Labor Relations, Jinan, Shandong 250357, China

Correspondence should be addressed to Gao Qiang; 14438120210041@sdmu.edu.cn

Received 16 April 2022; Revised 26 April 2022; Accepted 8 May 2022; Published 12 July 2022

Academic Editor: Jie Liu

Copyright © 2022 Gao Qiang. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

A software construction detection algorithm based on improved CNN model is proposed. Firstly, extract the vulnerability characteristics of the software, extract the characteristics from the static code by using the program slicing technology, establish the vulnerability library, standardize the vulnerability language, and vectorize it as the input data. Gru model is used to optimize CNN neural network. The organic combination of the two can quickly process the feature data and retain the calling relationship between the codes. Compared with single CNN and RNN model, it has stronger vulnerability detection ability and higher detection accuracy. In contrast, the software algorithm of the improved CNN model has strong vulnerability detection ability and higher detection accuracy. In terms of training loss rate, the DNN + Gru model is 17.2% lower than the single RNN model, 10.5% lower than the single CNN model, and 7% lower than the VulDeePecker model.

1. Introduction

Software systems are widely used in various production and life fields. The primary issue to be considered in the development process is security. Software vulnerabilities will not only cause unnecessary consumption of resources, but also seriously damage the economic property of the application industry. Traditional vulnerability analysis is divided into three methods: static analysis, dynamic analysis, and combined dynamic and static analysis [1]. Xia [2] compared the static analysis method with other program analysis methods and found that the static analysis method has a higher degree of automation and faster speed in detecting software vulnerabilities, but the static analysis method generally has the problem of high false positive rate. Lu [3] proposes a vulnerability detection technology based on dynamic taint analysis, which realizes the taint propagation process based on control flow and data flow, but frequent taint mark detection takes up a lot of memory and reduces system performance. Pan and Zhou [4] propose a method of combining static code analysis of pollution propagation model and dynamic detection of purification units to discover vulnerabilities in web applications, but this method is only used for cross-site scripting attacks and is used to detect other vulnerabilities, such as poor ability. Perl et al. [5] proposed a tool VccFinder that uses SVM classifier to mark suspicious codes. Although this tool reduces the false positive rate, it needs to reextract features and perform model training every time when detecting codes in different languages. Li et al. [6] developed the VulPecker tool, which has a very low false positive rate when detecting vulnerabilities in code clones, but is not suitable for dealing with other types of vulnerabilities.

With the continuous development of the deep learning discipline, the use of machine learning to achieve software vulnerability detection has gradually emerged. A deep learning-based Android malicious application detection is proposed, and a recurrent neural network is used to detect Smali static code, but this method is only aimed at malicious application attack problems and cannot find vulnerabilities in the code itself [7]. Li et al. [8] proposed an improved long short-term memory network (LSTM) model, which is

applied to the vulnerability detection problem of open source code, but this model is only for C/C++ source code problems and can only handle API and library function calls question.

On the basis of the above method, this paper proposes a software vulnerability detection method of deformable convolutional neural network, relying on the activation function and residual unit to improve the stability of the training gradient, because the convolution kernel can be shared in the convolutional neural network, and the network depth determines the length of the back propagation path, so it can greatly reduce the algorithm's time when detecting software vulnerabilities memory consumption.

2. Software Code Feature Extraction

Feature extraction of software code is the key to vulnerability detection. Firstly, program slicing is performed with key points in the vulnerability library as entry points, and code fragments containing vulnerability features are extracted from open source code, and these code fragments are called "code unit sets" [7]. Secondly, the set of code units containing vulnerabilities is vectorized, and the features are represented in a vector form that can be processed by the deep learning model. The feature processing flow of open source code is shown in Figure 1.

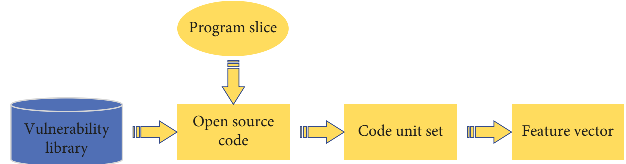

Figure 1: Feature code feature processing flow.

2.1. Establish a Vulnerability Library. In order to ensure that the slicing tool can accurately locate the code part containing the vulnerability features, a vulnerability library needs to be designed, and the key points of program slicing are defined in the vulnerability library. Taking API misuse as an example, the calling function of the API in the program is the key point of API misuse in the vulnerability library. Using the calling function as the entry point, the parameters, statements, and expressions related to the key points in the code are extracted. Therefore, the design of open source software vulnerability library is an indispensable link in static code vulnerability detection.

The open source software vulnerability library designed in this paper is mainly based on the CVE vulnerability database. CVE is compatible with 28 communities and institutions and contains about 6,500 entries. It is currently the authoritative standard vulnerability library for vulnerability scanning and evaluation. In addition, this paper also combines other large vulnerability information bases, such as CWE, NVD, and CNNVD. Through comparative analysis, the vulnerability library is roughly divided into seven categories: input validation, buffer overflow, memory management, API misuse, error handling, information leakage, and cross-site scripting [9]. Some key points are shown in Table 1.

2.2. Program Slicing. Program slicing is used to implement static code vulnerability feature extraction and process static code into a code unit set containing features. In the slicing process, the key points in the vulnerability library are used as entry points, and the control flow graph and data flow graph are constructed according to the order of mutual calls in the program and the flow of data parameters, so as to extract the expressions related to the key points. Formulas and statements and code statements and comments that are not related to features are removed [10]. There are many algorithms and tools related to program slicing. This paper uses LLVM to complete static code slicing.

2.3. Feature Vectorized Expression. After the program is sliced, a code unit set containing vulnerability features is obtained. The code unit set cannot be directly used as the input of the deep learning model and needs to be quantized into a fixed-length vector. In this paper, the word vectorization model word2vec is used to complete the vectorization of features. The word2vec model processes the code unit set by constructing a multilayer neural network. During the processing, the parameters of the neural network are continuously corrected and a series of linear and nonlinear operations are performed. Finally, we get the required word vectors. Before vectorizing the code unit set, the code unit set should be regularized, and the user-defined variables and function names in the code should be replaced with standard symbolic names in one-to-one correspondence. In this paper, the static analysis tool cppcheck is used to traverse line by line code and completes the substitution of user-defined variables and standardized names.

3. Improve the CNN Vulnerability Detection Model

3.1. CNN Model. The basic structure of CNN consists of an input layer, a convolution layer, a pooling layer, a fully connected layer, and an output layer. Generally, several convolution layers and pooling layers are used, and the convolution layers and pooling layers are alternately set; that is, one convolutional layer is connected to a pooling layer and so on. Since each neuron of the output feature surface in the convolutional layer is locally connected to its input, and the corresponding connection weights and local inputs are weighted and summed together with the bias value to get the input value of the neuron, this process is equivalent to the convolution process.

The convolution layer consists of multiple feature surfaces, each feature surface consists of multiple neurons, and each neuron is connected to the local area of the feature surface of the previous layer through a convolution kernel, which is a weight matrix (such as for two-dimensional avatars, it can be a 3×3 or 5×5 matrix) [11], the convolutional layer of CNN extracts input features through convolution operations, the first convolutional layer extracts

Table 1: Program vulnerabilities and key points.

Program vulnerabilities	Key points
Input validation problem	insect, create, select, alter, update, order, cookie, subject, system, command, open, close, getProperty, getRuntime
Buffer overflow problem	Strcpy, strlen, stract, strchr, scanf, sprintf, sterror, strcoll, sbumpc, strcnpy, cin, gets, fgets, getchat, getc,getpass, malloc, istream, printf
Misuse of API	Cin, gets, fgets, getchat, getc, getpass, memcpy, malloc, getParameter, equals, getProperty, read, gethostbyaddr
Content management issues	Malloc, calloc, realloc, alloca, free, new, delete, memcpy, memmove, memcmp, memchr, memset, mmap, munmap, memccpy, getpagesize
Error handling issues	-Alloca, catch, throw, EnterCriticalSection
Cross site scripting problem	URL, submit, cookie
Information leakage problem	Malloc, calloc, realloc, alloca, memcpy, memmove

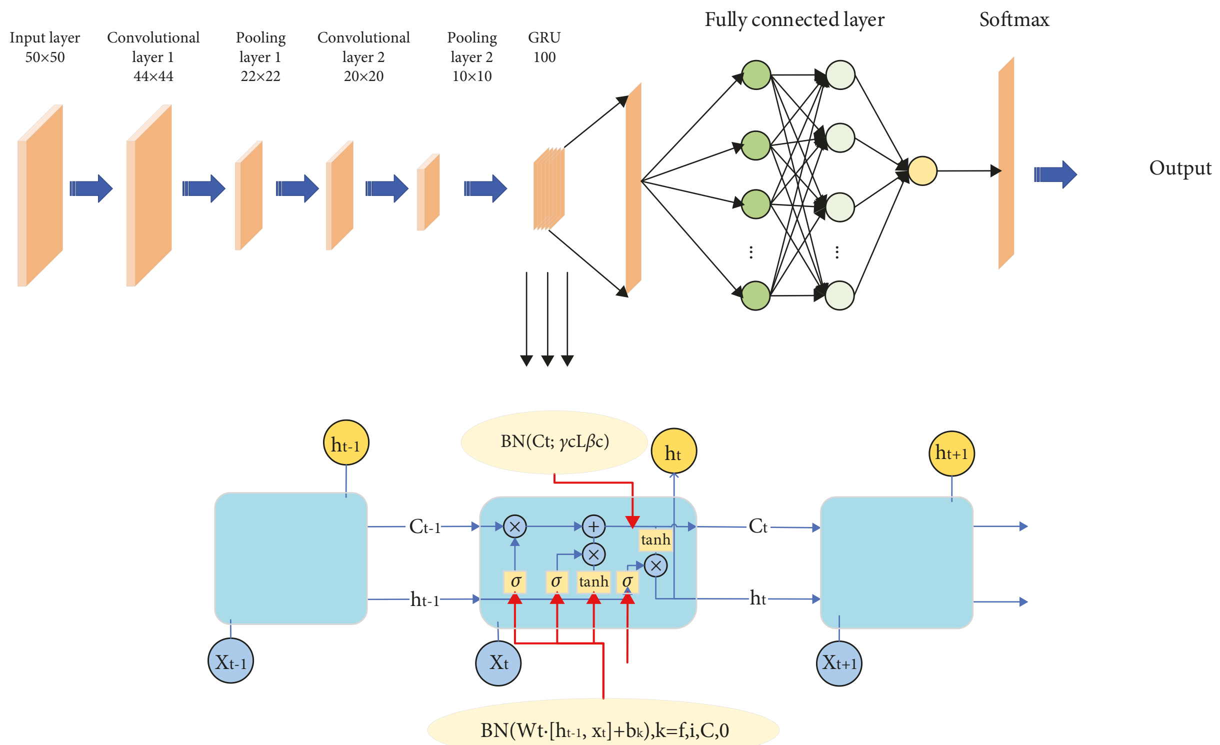

Figure 2: DNN + GRU model structure diagram.

low-level features, and the convolutional layer of higher layers extract higher-level features. Figure 2 shows a schematic diagram of the convolutional layer and pooling layer structure of a one-dimensional CNN.

The pooling layer follows the convolutional layer and is also composed of multiple feature surfaces, each of which uniquely corresponds to a feature surface of the previous layer and does not change the number of feature surfaces. As shown in Figure 2, the convolutional layer is the input layer of the pooling layer. A feature surface of the convolutional layer uniquely corresponds to a feature surface in the pooling layer, the neurons of the pooling layer are also connected to the local receptive field of the input layer, and the local receptive fields of different neurons do not overlap. The pooling layer aims to obtain spatially invariant features by reducing the resolution of feature surfaces [12].

The pooling layer plays the role of secondary feature extraction, and each neuron performs a pooling operation on the local receptive field. In the CNN structure, one or more fully connected layers are connected after multiple convolutional layers and pooling layers. Each neuron in the fully connected layer is fully connected to all neurons in the previous layer. The fully connected layer can integrate the class-discriminative local information in the convolutional layer or the pooling layer [13].

It can be seen from Figure 3 that the neurons of the convolution layer are tissue into each feature, and each neuron is connected to the local region of the upper layer, that is, the gland in the convolution layer. The feature in the input layer performs local connection [14]. The local connection weighted and passed to a nonlinear function such as the RELU function to obtain an output value of each neuron

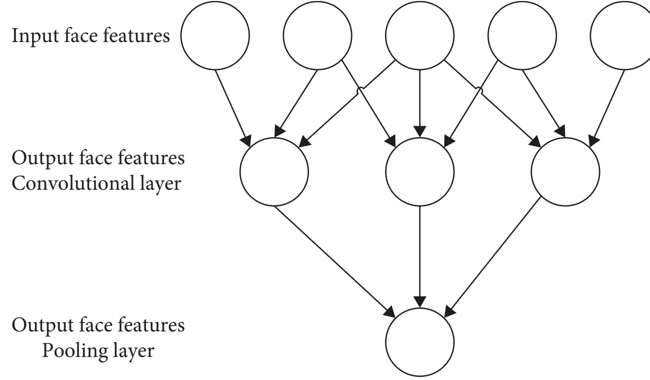

Figure 3: Schematic diagram of convolutional layer and pooling layer.

in the convolution layer. In the same input feature and the same output feature, the weight sharing of the CNN can reduce the model complexity by weight sharing, making the network easier to train.

3.2. CNN + GRU Model. Although CNN has good classification ability in vulnerability detection, it cannot well preserve the contextual relationship between code statements, and the overly complex neural network structure will have the problem of gradient disappearance as the number of layers increases [15]. RNN is often used to deal with time series problems and can better express the contextual calling relationship between codes, but RNN also has the problem of gradient disappearance. GRU is an effective variant of LSTM network. It has simpler structure and better effect than LSTM network. Therefore, it is also a very manifold network at present. Since Gru is a variant of LSTM, it can also solve the long dependency problem in RNN networks [16]. GRU introduces three gate functions into LSTM: input gate, forgetting gate, and output gate to control input value, memory value, and output value. In GRU model, there are only two doors: update door and reset door.

This paper proposes to combine CNN and GRU, organically integrate the advantages of the two models, and build a new model that is more suitable for open source software vulnerability detection. The CNN is used as the interface for interacting with the feature vector, and the GRU is used as the gating mechanism to deal with the relationship between the code statements, which constitutes the CNN + GRU model. The efficiency of CNN in processing data is higher and faster than GRU, and the automatic learning ability of convolution kernel is also stronger than GRU [17], and GRU model not only solves the problem of gradient disappearance in CNN, but also captures CNN Call information between code functions is ignored. The structure of the CNN + GRU model is shown in Figure 4.

In Figure 2, the first is the convolution and pooling processing of CNN. CNN can quickly process high-dimensional data and ensure the invariance of feature data to the greatest extent during dimensionality reduction [18]. Second, the GRU is embedded between the pooling layer and the fully connected layer, and the GRU is used to preserve the up-down calling

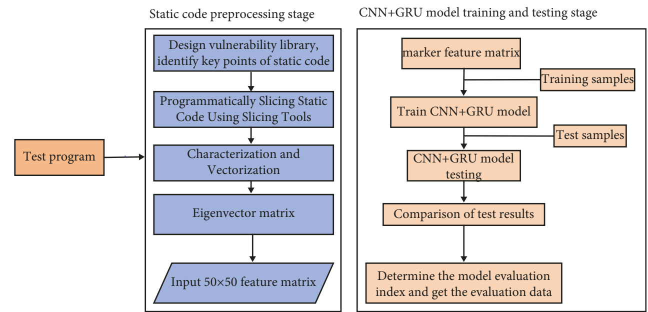

Figure 4: The process of CNN + GRU model detecting software vulnerabilities.

relationship between code data. Finally, the fully connected layer is used to complete the normalization process, and the processed output value is sent to the SoftMax classifier for classification and detection, and the classification result is obtained [19].

The input layer is a preprocessed 50×50-dimensional feature matrix. The red square in the figure represents the convolution kernel with a size of 7×7. The convolution kernel is the weight matrix in the perception field. The scan pitch for input data is set to 1. There may be out-of-bounds phenomenon when scanning to the boundary, the boundary needs to be expanded, and the value of the out-of-bounds part is set to 0. The input of the convolutional layer is the 50×50 feature matrix in the input layer, and the output matrix dimension is determined by

$$\begin{cases} \text{height}_{\text{out}} = \dfrac{\text{height}_{\text{in}} - \text{height}_{\text{kemel}} + 2 \times \text{padding}}{\text{stride}} + 1, \\ \\ \text{width}_{\text{out}} = \dfrac{\text{width}_{\text{in}} - \text{width}_{\text{kemel}} + 2 \times \text{padding}}{\text{stride}} + 1. \end{cases} \quad (1)$$

In formula (1), height and width represent the length and width of the matrix, padding is the padding mode, and stride is the step size. To be precise, each convolution kernel also contains a bias parameter, but the formula omits bias. In the CNN + GRU model, padding is 0, stride is 1, and the length and width of the output matrix are $(50 - 7 + 2)/1 + 1 = 44$; that is, the input matrix of pooling layer 1 is 44×44

dimensions. The pooling layer is mainly to compress and reduce features and prevent overfitting. In pooling layer 1, a filter of size 2×2 is used, and the stride is chosen to be 2. It can be concluded that the output of pooling layer 1 is 22×22. The processing of convolutional layer 2 and pooling layer 2 is similar to convolutional layer 1 and pooling layer 1.

GRU is embedded between the pooling layer and the fully connected layer. Since CNN uses filters and windows of different sizes to process data, it often loses the up-down calling and transfer relationship between these code data. In addition, too many neural network layers will also have the problem of gradient disappearance, so it is necessary that GRU acts as a storage timing information and control gate in the whole model. In the GRU structure diagram, x is the input, h is the output, f_i is the forgotten part of the input information, and r_i is the memorized part of the input information. The calculation in GRU is shown in

$$\begin{cases} r_t = \sigma(W_r \times [h_{t-1}, x_t]), \\ f_t = \sigma\left(W_f \times [h_{t-1}, x_t]\right), \\ \widetilde{h}_t = \mathrm{Relu}(W \times [f_t \times h_{t-1}, x_t]), \\ h_t = (1 - r_t) \times h_{t-1} + r_t \times \widetilde{h}_t. \end{cases} \quad (2)$$

In formula (2), w represents the weight, and relu is the activation function. In order to make CNN + GRU have nonlinear modeling ability, an activation function is added to GRU. The relu function, which is faster to calculate and can alleviate gradient disappearance, is selected as the activation function, as shown in

$$f(x) = \max(0, x), \quad (3)$$

where x is the input, $f(x)$ is the output, and relu can keep the gradient from decaying when $x > 0$, alleviating the problem of gradient disappearance.

The problem of program vulnerability detection is actually a two-category problem, with or without loopholes. So you need to add a fully connected layer and a SoftMax layer at the end of the model. The fully connected layer is responsible for further dimensionality reduction and purification of the features, and the classifier is responsible for whether the final sample contains vulnerabilities. Through filters (also called convolution kernels), the fully connected layer connects the input and output together, and the fully connected part is shown in

$$W * x + b = z. \quad (4)$$

Among them, $x = [x_0, x_1, x_2, \ldots, x_n]_T$ is the input vector; $y = [y_0, y_1, y_2, \ldots, y_n]_T$ is the output vector, then the filter part is a matrix of size $m \times n$, and b is a partial set of the term, $b = [b_0, b_1, b_2, \ldots, b_n]_T$.

SoftMax classifiers are widely used to solve multi-classification problems in various domains. The input feature of the SoftMax function is set to x, and the probability value is $p(y = j|x)$, assuming the function is as follows:

$$h_\theta(x) = \begin{bmatrix} p\left(y^{(i)} = 1|x^{(i)}; \theta\right) \\ p\left(y^{(i)} = 2|x^{(i)}; \theta\right) \\ \vdots \\ p\left(y^{(i)} = k|x^{(i)}; \theta\right) \end{bmatrix} = \frac{1}{\sum_{j=1}^{k} e^{\theta_j^T x(i)}} \begin{bmatrix} e^{\theta_1^T x(i)} \\ e^{\theta_2^T x(i)} \\ \vdots \\ e^{\theta_k^T x(i)} \end{bmatrix}. \quad (5)$$

The parameter θ is obtained through training, the setting of θ needs to minimize the regression cost function, k is the dimension of the vector, and the regression cost function of SoftMax is

$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{j=1}^{k} 1\left\{y^{(i)} = j\right\}|b \frac{e^{\theta_j^T x(i)}}{\sum_{i=1}^{k} e^{\theta_j^T x(i)}}\right] + \frac{\lambda}{2}\sum_{i=1}^{k}\sum_{j=0}^{n} \theta_{ij}^2. \quad (6)$$

$(\lambda/2)(\sum_{i=1}^{k}\sum_{j=0}^{n}\theta_{ij}^2)$ is the weight failure; in order to minimize the value of $J(\theta)$, use iterative optimal algorithm. By seeking, gradient formulas can be obtained:

$$\nabla_{\theta_j} J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[1\{x_i(y_i = j\} - p(y_i = j|x_i; \theta)\right] + \lambda\theta_j. \quad (7)$$

$\nabla_{\theta_j} J(\theta)$ represents a vector, and a SoftMax model can be implemented by minimizing $J(\theta)$.

3.3. Model Assessment Indicator

3.3. Model Assessment Indicator. Before training and testing models, you need to give an evaluation indicator of the vulnerability detection model, the accuracy (ACC), and the loss rate (LOSS), which is often used.

Refer to the mainstream assessment index system, according to the difference between the prediction results and the real results, divided into the following four cases [20]:

TP: the prediction result is positive and the real results are positive.

FP: the prediction result is positive, and the real results are negative.

FN: the forecast results are negative, and the real results are positive.

TN: the forecast results are negative, and the real results are negative. The calculation of the accuracy ACC is as shown in

$$\mathrm{ACC} = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{TP} + \mathrm{FP} + \mathrm{FN} + \mathrm{TN}}. \quad (8)$$

The loss rate of the CNN + GRU model is calculated by the cross-entropy loss function and reflects the gap between the prediction results and the real results by calculating the cross entropy. The collection of predicted results is used to represent the collection of real results, and the cross-entropy of the two sets can be defined as follows:

Table 2: Simulation experiment environment.

Name	Parameter
RAM	16 GB
HD	512 GB
CPU	Intel® Core™ i5-3515M
OS	64-bit Win10 physical machine
Python	3.7.1

$$H(p,r) = E_p[-|br] = H(p) + D_{KL}(pr). \tag{9}$$

$H(p)$ represents the entropy of P, and $D_{KL}(pr)$ is KL distance to measure the distance of two collections.

4. Simulation Experiment

When using the software vulnerability detection algorithm of the CNN model, the corresponding indicators are mainly the accuracy rate of training and the loss rate of training as the main basis for judging the detection of software vulnerabilities of the CNN model.

4.1. Experimental Environment. The training and testing environment of the vulnerability detection model is 16 GB memory, and the processor is Intel® Core™ i5-3515M, 64-bit Win10 physical machine. Open source data often have class imbalance problems; that is, most of the data are positive samples (samples without vulnerabilities), while the number of negative samples is low (samples with vulnerabilities). Such imbalance problems will affect the vulnerability detection model performance, causing vulnerability cases in the standard library. This dataset is preprocessed, including feature extraction, normalization, and vectorization of features. Finally, more than 300 samples were formed as experimental data (171 positive samples and 144 negative samples), and 215 samples were taken as training data (117 positive samples and 98 negative samples). The remaining 100 samples are used as test samples, 54 positive samples, and 46 negative samples, which are used to verify the vulnerability detection ability of the CNN + GRU model as shown in Table 2.

4.2. Detection Process. The vulnerability detection process can be divided into two stages: the static code preprocessing stage and the vulnerability detection model training and testing stage [21].

In the static code preprocessing stage, first of all, refer to the real vulnerability cases in CVE, extract the API functions with specific errors in the cases, divide the API functions into 7 categories to construct the open source software vulnerability library of this article, and use the API functions in the vulnerability library. It is the key point and the entry point of the program slice. Second, collect the data set, use LLVM to slice the data set program, extract key points from the data set code, and construct the control flow chart of the key point function. In the control flow graph, each node is a basic block. The variables and operations related to the key point function are found through each branch of the basic block. Finally, all the basic blocks related to the key point function are intercepted to form more than 300 code unit sets. In addition, it is necessary to standardize and vectorize the code unit set and use word2vec to vectorize more than 300 code unit sets in batches to obtain training samples. Use the same method to obtain test samples. Finally, normalize all the feature vectors, and process the feature vectors into a 50×50-dimensional feature matrix according to the size of the sample. The size of the ordinate is the dimension of the word vector, and the abscissa is the number of word vectors. If the word vector is less than 50, it is filled with 0. In the training and testing phases of the vulnerability detection model, the training model and the testing model need to be written, and the entire compilation process uses the python language. Add a label to the test sample, set the label of the sample containing the vulnerability to "0," and set the label of the sample without the vulnerability to "1." 215 samples were taken as training data. In the experiment, batch extraction was used to extract a fixed number of samples from the test samples each time, and the model was trained through multiple iterations. In the testing phase, use the trained CNN + GRU model for testing, compare whether the model test results are the same as the actual results, and test the detection ability of the CNN + GRU model. The overall process of the entire static code vulnerability detection is shown in Figure 4.

4.3. Analysis of Experimental Results. In the training phase, iterate 3000 training cycles, and use the minibatch gradient descent algorithm (MBGD) for batch extraction. Every 10 iterations, the current training accuracy rate (training ACC) and training loss rate (training loss) are output, and save the model document. The model document saves the weights that are adjusted and set when training the neural network, so that the model document can run directly. The detection results of part of the model during the training process are shown in Table 3.

During the training process, the accuracy and loss rate after 500 iterations are stable at 0.903 3 and 0.154 1. The weight values of the model at this time are saved in the ckft model document, and the parameters in the document are in the testing phase parameter to use. The changes in the accuracy and loss rates during training are shown in Figure 5.

As can be seen from Figure 5, when the number of iterations is less than 500, the accuracy of the whole curve is significantly improved; when the number of iterations is more than 500, the accuracy curve tends to be stable and remains at about 0.9. When the number of iterations is less than 500, the whole curve shows an obvious downward trend. When the number of iterations is greater than 500, the curve also tends to be stable and remains at about 0.15. After the model training is completed, the test samples are tested. By loading the model documents saved during the training, the model can be directly restored to the state at the end of the training. 100 samples are randomly selected from 315 test samples each time for 5 times. See Table 4 for test accuracy (test ACC) and test loss rate (test loss).

In order to further prove that the CNN + GRU model has high vulnerability detection ability, the CNN + GRU model

Table 3: CNN + GPU model training results.

Number of iterations	Training ACC	Training loss
10	0.3706	0.6635
20	0.6225	0.3782
30	0.8742	0.2375
60	0.9017	0.1581
90	0.9032	0.1543
120	0.9033	0.1541
150	0.9033	0.1541

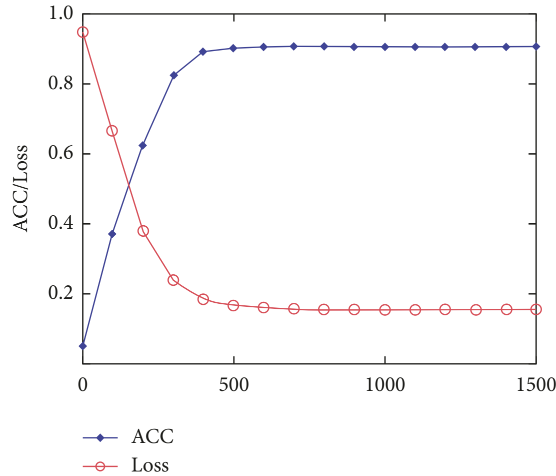

Figure 5: CNN + GRU model test results.

Table 4: CNN + GRU Model test results.

Sample	Test ACC	Test loss
1	0.8800	0.1617
2	0.8700	0.1674
3	0.8700	0.1707
4	0.8600	0.1972
5	0.8700	0.1591

Table 5: Data comparison of deep learning models.

Model	Training ACC	Training loss	Test ACC	Test loss
RNN	0.8372	0.1819	0.8000	0.2068
CNN	0.8810	0.1702	0.8400	0.1918
VulDeePecker	0.8791	0.1613	0.8300	0.1844
CNN + GPU	0.9033	0.1541	0.8700	0.1713

is compared with a single CNN, RNN, and the existing vulnerability detection model VulDeePecker [22]. The CNN, RNN, and VulDeePecker models were trained and tested using the same dataset, and the experimental results of the four models were compared. The specific results are shown in Table 5. The information in the table is the average of the training data and test data of the four models. It can be seen that the experimental results of CNN + GRU are the best.

The experimental results show that it is feasible to nest GRU into the pooling layer and fully connected layer of CNN. The CNN + GRU model proposed in this paper can not only ensure the invariance of feature vectors to the greatest extent during dimension reduction, but also preserve the invariance between codes. Call relationship has stronger vulnerability detection ability, and compared with CNN, RNN, and VulDeePecker models, CNN + GRU has higher accuracy and lower loss rate.

5. Conclusion

This paper proposes a software building detection algorithm based on an improved CNN model. Firstly, extract the vulnerability features of the software, use program slicing technology to extract features from static code, establish a vulnerability library, and standardize and vectorize the vulnerability library as input data. GRU model is used to optimize CNN neural network. The organic combination of the two can quickly process the feature data and retain the calling relationship between the codes. The improved CNN model is better than the single CNN and RNN model in vulnerability detection ability and detection accuracy. Compared with single CNN model and VulDeePecker model, the training loss rate is 4.25% higher. On the contrary, compared with single RNN model and VulDeePecker model, the training loss rate is 17.2% and 7% lower, respectively.

Compared with other single algorithms, the improved CNN algorithm has relatively high requirements for data and needs to be further optimized in the future.

Data Availability

The dataset can be accessed upon request.

Conflicts of Interest

The author declares that there are no conflicts of interest.

References

[1] H. Shahriar and M. Zulkernine, “Mitigating program security vulnerabilities,” *ACM Computing Surveys*, vol. 44, no. 3, pp. 1–46, 2012.

[2] Y. Xia, “Research on security vulnerability detection technology based on static analysis,” *Computer Science*, vol. 33, no. 10, pp. 279–282, 2006.

[3] K. Lu, *Research and Implementation of Vulnerability Attack Detection Technology Based on Dynamic Taint Analysis*, University of Electronic Science and Technology Press, Chengdu, China, 2013.

[4] G. Pan and Y. Zhou, “XSS vulnerability discovery based on static analysis and dynamic detection,” *Computer Science*, vol. 39, no. s1, pp. 51–53, 2012.

[5] H. Perl, S. Dechand, and M. Smith, “VccFinder: finding potential vulnerabilities in opensource projects to assistcode audits,” in *Proceedings of the 22nd ACM SIGSACConference on Computer and Communications Security*, pp. 426–437, Denver, CO, USA, October 12-16, 2015.

[6] Z. Li, D. Zou, and S. Xu, “VulPecker : an automated vulnerabilitydetection system based on code similarity analysis,”

in *Proceedings of the Conference on Computer Security Applications*, pp. 201–213, Angeles, CA, USA, December 2016.
[7] S. Chen, *Research and Implementation of Android Malicious Application Detection Technology Based on Deep Learning algorithm*, Beijing University of Posts and Telecommunications Press, Beijing, China, 2016.
[8] Z. Li, D. Zou, and S. Xu, "VulDeePecker: a deep learning-based system for vulnerability detection," in *Proceedings of the Network and Distributed Systems Security (NDSS) Symposium*, Diego, CA, USA, February 2018.
[9] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE Transactions on Software Engineering*, vol. 37, no. 6, pp. 772–787, 2011.
[10] B. Chernis and R. Verma, "Machine learning methods for softwarevulnerability detection," in *Proceedings of the ACM International Workshop*, pp. 31–39, Tokyo, Japan, November 2018.
[11] R. Grosu and S. A. Smolka, *Monte Carlo Model checking// Tools and Algorithms Forthe Construction and Analysis of Systems*, Springer Berlin Heidelberg, Berlin, Germany, 2005.
[12] M. D. Zeiler and R. Fergus, "Stochastic pooling for regularzition of deep convolutional neural networks," 2013, https://arxiv.org/abs/1301.3557.
[13] Y.-L. Boureau, J. Ponce, and Y. LeCun, "A theoretical analusis of feature pooling in visual recognition," *International Conference on Machine Learning*, vol. 32, no. 4, pp. 111–118, 2010.
[14] D. Erhan, Y. Bengio, and A. Courville, "Why does unsupervised pre-training help deep learning?" *Journal of Machine Learning Research*, vol. 11, no. 3, pp. 625–660, 2010.
[15] J. Saxe and K. Berlin, "Xpose : a character-level convolutionalneural network with embeddings for detecting maliciousURLs, file paths and registry keys," 2017, https://www.arxiv-vanity.com/papers/1702.08568/.
[16] Z. Qu, L. Su, and X. Wang, "A unsupervised learningmethod of anomaly detection using GRU," in *Proceedings of the IEEE International Conference on Big Data & Smart Computing*, IEEE, Shanghai, China, January 2018.
[17] F. Wu, J. Wang, and J. Liu, "Vulnerability detection withdeep learning," in *Proceedings of the IEEE International Conference onComputer and Communications*, pp. 1298–1302, IEEE, Chengdu, China, December 2017.
[18] J. Su, Z. Tan, and D. Xiong, "Lattice-based recurrent neuralnetwork encoders for neural machine translation," in *Proceedings of the 31st AAAI Conference on ArtificialIntelligence*, pp. 3302–3308, February 2017.
[19] V. Nair, G. E. Hinton, and C. Farabet, "Rectified linear units implement restored boltzmann machines," in *Proceedings of the 27th International Conference on Machine Learning*, pp. 807–814, Haifa, Israel, July 2010.
[20] D. Silver, A. Huang, and C. J. Maddison, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.
[21] S. Lawrence, C. L. Giles, A. C. Ah Chung Tsoi, and A. Back, "Face recognition: a convolutional neural-network approach," *IEEE Transactions on Neural Networks*, vol. 8, no. 1, pp. 98–113, 1997.
[22] C. Neubauer, "Evaluation of convolutional neural networks for visual regression," *IEEE Transactions on Neural Networks*, vol. 9, no. 4, pp. 685–696, 1998.