Automated Software Vulnerability Detection via Pre-trained Context Encoder and Self Attention

Na Li, Haoyu Zhang, Zhihui Hu, Guang Kou, and Huadong Dai(✉)

Artificial Intelligence Research Center, Defense Innovation Institute, Beijing, China
zhanghaoyu10@nudt.edu.cn, hddai@vip.163.com

Abstract. With the increasing size and complexity of modern software projects, it is almost impossible to discover all software vulnerabilities in time by manual analysis. Most existing vulnerability detection methods rely on manual designed vulnerability features, which is costly and leads to high false positive rates. Pre-trained models for programming language have been used to gain dramatic improvements to code-related tasks, which considers syntactic-level structure of code further. Thus, we propose an automated vulnerability detection method based on pre-trained context encoder as well as self-attention mechanism. Instead of current static analysis approaches, we treat the program source code as natural language and introduce the pre-trained contextualized language model to capture the program local dependencies and learn a better contextualized representation. The extracted source code feature vectors are then fed into a designed Self Attention Networks (SAN) module. We develop the SAN module based on Long-Short Term Memory (LSTM) model and self attention, which learns the long-range dependencies of program vulnerable points more efficiently. We conduct experiments on two source code level C program benchmark datasets, where four different evaluation metrics are applied for comparing the vulnerability detection performances of different systems. Extensive experimental results demonstrate that our proposed model outperforms previous state-of-the-art automated vulnerability detection method by around 7.2% in F1-measure and 2.6% in precision.

Keywords: Automated vulnerability detection · Self attention · Pre-trained language model · Transfer learning

1 Introduction

Software vulnerabilities usually refer to the internal defects of software, and these defects may be used to damage the software systems. To improve the security of software systems, a series of software development security principles have been proposed. With the development of computer technology, the demand and scale

P. Gladyshev et al. (Eds.): ICDF2C 2021, LNICST 441, pp. 248–264, 2022.
https://doi.org/10.1007/978-3-031-06365-7_15

of software are expanding rapidly. Meanwhile, the number and complexity of codes are increasing exponentially. Due to the increasing complexity of modern programs and the wide application of software, the number of malicious software attacks also continues to rise. Thousands of software vulnerabilities are reported by the Common Vulnerabilities and Exposures [1] and National Vulnerability Database [5] each year. Many can propagate quickly owing to the prevalence of code cloning and open-source software. These vulnerabilities pose a serious threat that potentially allows attackers to compromise systems and applications.

Traditional vulnerability detection techniques include static and dynamic analysis of programs [8]. The static analysis which analyzes source code without running the program has high coverage, and the dynamic analysis which discovers the nature of programs by running the software has a low false positive rate. Most of the static or dynamic systems and studies for vulnerability detection are based on pre-defined rules [2,4,6] and code similarity metrics [12], which rely on experts experiences to define features. These existing tools for static or dynamic analysis techniques generated by human experts typically only detect a limited subset of possible errors. And, it is subjective to define the features of software vulnerabilities accurately for experts, which sometimes leads to an error-prone task. In other words, vulnerability detection rules are very difficult to be defined accurately and completely. Therefore, more intelligent and automatic research for vulnerability detection instead of human experts to manually define has been the future trend. As deep learning in natural language processing (NLP) develops rapidly, researchers used neural networks based vulnerability detection technology to achieve vulnerability features automatically. The large amount of open-source code make it possible to learn the patterns of software vulnerabilities intelligently and recent work shows great potential in this field [14,16,24,27]. However, capturing effective and high-quality vulnerability features from the program source code is still a difficult and unsolved problem.

To solve the problem of difficulty in capturing vulnerability features and automatic vulnerability detection, in this paper we propose a novel model that consists of a pre-trained context encoder based vulnerability representation module and a self-attention enhanced Long-Short Term Memory (LSTM) module. Different from traditional static word embeddings, our approach learns word level vulnerability representations by designing a transfer learning framework based on Bidirectional Encoder Representation from Transformers (BERT) model [10,23], where the pre-trained model is first fine-tuned on the vulnerability source code. BERT is a successful pre-trained context language model in NLP, and it can effectively capture syntactic and semantic vulnerability information with the fine-tuning process. Then, for further improving the accuracy of vulnerability detection, LSTM with self attention mechanism is selected for vulnerability classification learning. LSTM model is an extension of Recurrent Neural Network (RNN) [18]. In the SAN module, attention layer is added between the LSTM layer and the output layer for adjusting the weight of each sequence. The LSTM-attention model can extract key information points in vulnerability source code. Compared with state-of-the-art intelligent methods, it can be

seen that our method outperforms them by a large margin in four metrics: false positive rate, false negative rate, precision, and F1-measure.

The rest of this paper is organized as follows: Sect. 2 reviews the related work about detecting software vulnerability automatically. Section 3 presents our method for automatic vulnerability detection on source code. Section 4 describes details of experiment evaluation metrics, corresponding results, and analysis. Finally, Sect. 5 concludes the paper.

Our Contributions. First, we propose the design and implementation of the pre-trained context encoder-based vulnerability representation on source code. We leverage transfer learning that allows us to obtains high-quality deep context-dependent representations through fine-tuning the pre-trained model, which encodes the symbolic form into the real vector value. The experiments show that compared with the different language representation models, our method has given rise to overall task performance. Second, since there are long-range dependencies of program vulnerable points, we present the SAN module for learning the dependency of vulnerability information. Finally, we conduct experiments to confirm the effectiveness of our approach. Meanwhile, through a series of sub-experiments, the method that we proposed can achieve the desired goals.

2 Related Work

To detect vulnerability automatically, the techniques learning patterns in source code that may be related to software vulnerabilities have been proposed by researchers. In [12,15], both methods detect software vulnerabilities based on identifying similar code. Different than machine learning techniques trying to learn patterns from large amounts of vulnerability features, they are limited to vulnerable code clones. In [21], the authors present a vulnerability detection technique based on combining N-gram analysis and feature selection algorithms, which could reduce the feature and search space by machine learning-based feature selection algorithms. In [25], this paper aims at extracting search patterns for taint-style vulnerabilities in C source code, and the programs are represented as code property graphs which are traversed as features for vulnerability classification based on k-means clustering algorithm. Traditional machine learning techniques focus on inferring the features from the pre-classified vulnerabilities.

Deep learning based recent development techniques is capable of learning vulnerability features automatically. In [17], this paper proposes a vulnerability detection system named VulDeePecker. The VulDeePecker obtains vector representation from code gadgets which are small pieces of relevant code sliced by a commercial code analysis product Checkmarx. To acquire vector representation, VulDeePecker uses word2vec [7] to vectorize the code gadgets, then applies the vectors to the Bi-LSTM network to classify vulnerabilities. In this paper, it can be shown that the application of deep learning to vulnerability detection is feasible, however, there are still some defects. With word2vec to vectorize the source code, VulDeePecker may lose important semantic information in context about

the vulnerabilities, which affects the effectiveness of the vulnerability detection model. In [16], to improve VulDeePecker, the researchers propose a framework called SySeVR, which aims at achieving program representations for syntax and semantic information associated with vulnerabilities. In addition, it also adds different types of machine learning models to cover more types of vulnerabilities. In [26], the paper proposes the Devign model based on a graph neural network, which includes a novel Conv module to find features in a graph's heterogeneous node features for graph-level classification. Additionally, Devign also could craft with automatically property graphs to learn vulnerable patterns through using graph neural networks.

3 The Methodology

Figure 1 shows an overview of our proposed method, which aims to automatically detect vulnerabilities in the source code level. It first transforms code gadgets into symbolic representation when the input data flows through the Pre-Processing stage. In the Fine-tuning stage, it fine-turns the released BERT-Base pre-trained model by using the symbolic code gadgets extracted randomly from the training code gadgets. The output of the Fine-tuning stage is used as the model for representation learning in the next two stages. It transforms the symbolic form of source code into a real vector value in the Embedding stage. Finally, in the Classifier training stage, the detection model is trained and will be applied to predict whether test samples are vulnerable or not.

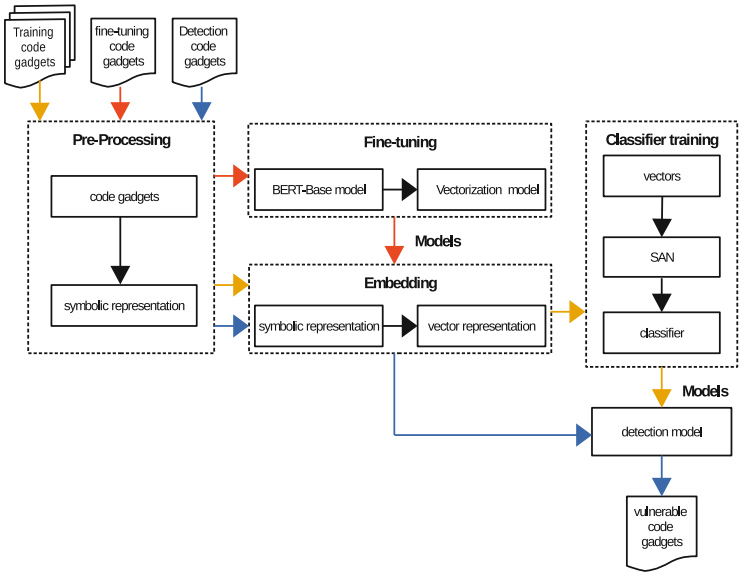


Fig. 1. Overview of the proposed automatic software vulnerability detection method.

3.1 Pre-processing

Instead of taking an entire code file as input, a code gadget is a small code snippet acquired by using the program slicing technique, and it performs slicing based on the risk library/API function calls which are closely related to the vulnerabilities. By encoding code gadgets in symbolic representation, user-defined variables and functions in code gadgets are mapped to the uniform symbolic form. In this step, it can not only retain semantic information [28] but also improve the efficiency of training and reduce unnecessary time and space cost of embeddings.

As shown in Fig. 2, the sample in the form of the source code is transformed into the symbolic representation in the Pre-Processing stage. First, it extracts library/API function calls and assembles slices which are generated from the arguments of the library/API function into code gadgets. Second, it transforms variables into symbolic names (e.g., "V1", "V2") and user-defined functions into symbolic names (e.g., "F1", "F2"). As risk library/API function calls may cause different types of vulnerabilities, multiple functions mapped to the uniform symbolic name can increase the generalization ability of classifier training.

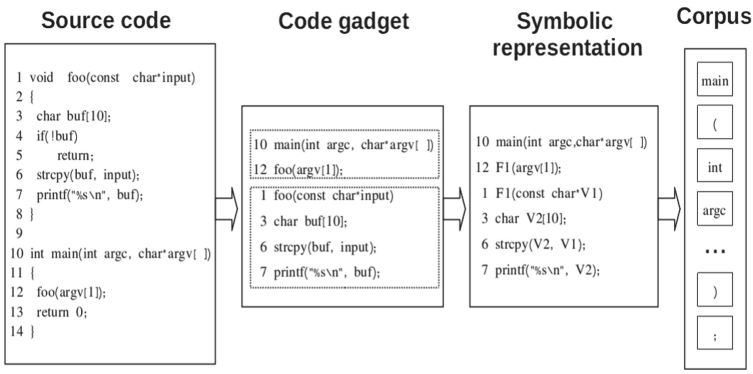


Fig. 2. Transform the source code into the symbolic representation.

3.2 Fine-Tuning

Since deep learning models take vectors as inputs, the symbolic representation needs to be converted to real value vectors. These vectorized representation as a feature extractor to produce the contextual word embeddings are fed into the neural network language models. Word2vec that makes the dense vector representation used for different tasks in NLP is one of the most popular in these embedding models. Word2vec maps words into low-dimensional space vectors, but it cannot learn context-aware word representations. During the same period, coarse-grained language models such as doc2vec [3] and sent2vec [21], have also received some attention. These methods try to encode paragraph, sentence or document into a fixed-dimensional vector representation, but the generalization performance of these sentence embedding models is too poor. Some sentence

embedding models are also proposed at the same time, such as doc2vec [3] and sent2vec [21]. However, these sentence embedding models try to learn a fixed-length feature representation, rather than the contextual representation for each token. Different from traditional word embeddings which only allow a single context-independent representation for each word, BERT language representation [10,23] uses the masked language model for pre-training a deep bidirectional transformer that can fuse the left and the right context. BERT model is the first fine-tuning based representation model that has shown effective performance on a large suite of sentence-level and token-level tasks. BERT which stands for deep bidirectional transformers can not only obtain learn context-aware word representations but also be a context encoder. And, it is also the meaning of the pre-trained context encoder in the title.

In the Fine-tuning stage, the released BERT-Base pre-trained model (Uncased: 12-layer, 768-hidden, 12-heads, 110M parameters) is leveraged to obtain the fine-tuning model named Vectorization. As illustrated in Fig. 3, both pre-training and Vectorization model have the same architectures. The model is first initialized with the same pre-trained parameters, then the symbolized source codes extracted randomly are used to fine-tune BERT. As different layers of the neural network can capture different syntax and semantic information, it requires us to select appropriate outputs used as a reference for fine-tuning. Considering the over-fitting problem, the value of the last layer is too close to the target. Therefore, the outputs of the penultimate layer are taken. During fine-tuning, all parameters are fine-tuned in Vectorization model end-to-end.

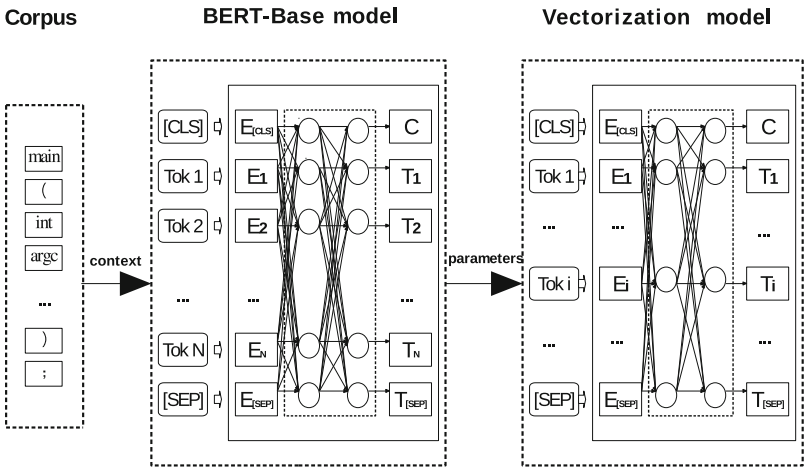


Fig. 3. In the Fine-tuning stage, obtain the fine-tuning model named Vectorization.

3.3 Embedding

Vectorization model is applied to obtain the vector representation of context. Vectorization model takes symbolized source codes as input, then outputs the vector representation after extracting text features. BERT input embeddings are the sum of the token embeddings, the segmentation embeddings, and the position embeddings. These inputs are tokenized before being sent to token embeddings, then two special tokens used to divide sentences are inserted at the beginning ([CLS]) and the end ([SEP]) of the tokenization. In the Embedding stage, each word is transformed into a fixed-dimensional vector representation.

3.4 Classifier Training

Recurrent Neural Network (RNN) [18] is a typical neural network used to process sequence data. RNN is more suitable to deal with long-distance dependence problem, but the problem of gradient disappearance exists in the process of RNN modeling. To solve this problem, many variants of RNN have been proposed, such as LSTM and Gated Recurrent Unit (GRU) [9,11].

The neural network based on attention mechanism [19] can extract important features in the text by training word vector representation. In the SAN module, "key points" in vulnerabilities can be detected through the role of attention mechanism, and they are often important information for vulnerability classification. A typical network structure of the SAN module is shown in Fig. 4. The inputs in the Fig. 4 are the vectors after embeddings, and these inputs will pass through the LSTM layers. Then, the attention layers are introduced into the hidden layer to calculate the attention probability distribution value received from the LSTM layers. The output matrix of LSTM units regarded as the input of the attention mechanism can be expressed by:

$$H = (h_1, h_2, ..., h_T) \tag{1}$$

The final output of the attention unit is computed with multiplying the weight by the input vector, denoted as:

$$v = \sum_{i}^{t} \alpha_i h_i \tag{2}$$

The weight, for each feature vector included in the feature matrix, can be accomplished with the softmax activation function as follows:

$$\alpha_t = \frac{e^{q^T h_t}}{\sum_i^T e^{q^T h_i}} \tag{3}$$

Finally, the softmax function is responsible for representing and formatting the classification result, which provides feedback for updating the neural network parameters in the training phase. The output is the SAN module with fine-tuned model parameters, and the output of the detection phase is the classification results.

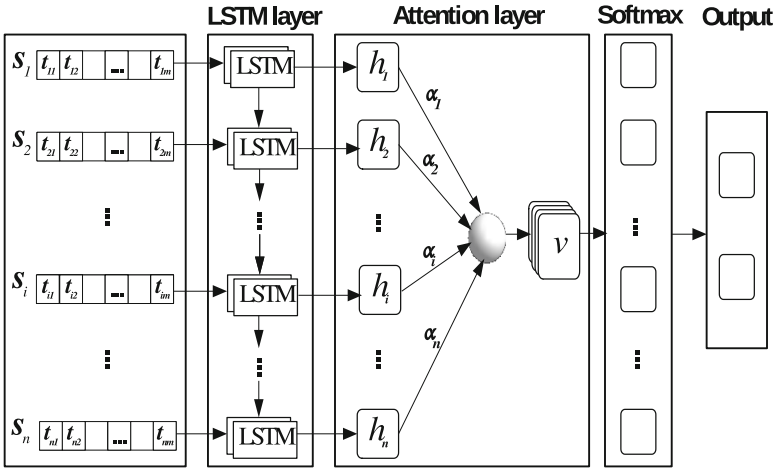


Fig. 4. A typical network structure of the SAN module.

4 Experiment and Results

4.1 Datasets

The datasets used in this paper come from Code Gadget Database (CGD) [17], which include two datasets as the samples: the buffer overflow vulnerabilities (CWE-119) and resource management error vulnerabilities (CWE-399). Each sample in the datasets is small pieces of the C source code with above known vulnerabilities in software products. The number of samples is shown in each dataset as follows in Table 1. Each dataset is divided into two parts by 4:1, where the larger part is used for training and the other one is for detection. Besides, half of each training set is randomly selected from the training set for fine-tuning. Table 2 summarizes the partitioning of each dataset, where the second column represents the number of samples for training, the third column represents the number of samples for fine-tuning and the fourth column represents the number of samples for detection.

Table 1. The datasets in code gadget database (CGD).

Dataset	Samples	Vulnerable	Not vulnerable
CWE-119	39753	10440	29313
CWE-399	21885	7285	14600

Table 2. The partitioning of each dataset.

Dataset	Training	Fine-tuning	Detection
CWE-119	31802	15901	7951
CWE-399	17508	8754	4377

4.2 Evaluation Metrics

In our experiment, we use the False Positive Rate (FPR), False Negative Rate (FNR), Precision (P), and F1-measure (F1) to evaluate the vulnerability detection model [22]. Let True Positive (TP) be the number of vulnerable samples classified correctly, False Positive (FP) be the number of samples with false vulnerabilities classified, False Negative (FN) be the number of falsely detected vulnerable samples, and True Negative (TN) be the number of samples with no vulnerabilities undetected. The false positive rate (FPR) = FP/(FP+TN), measures the proportion of falsely classified positive vulnerabilities in all samples that are not vulnerable. False Negative Rate (FNR) = FN/(FN+TP), measures the proportion of falsely classified negative vulnerabilities in all samples that are vulnerable. Precision (P) = TP/(TP+FP), represents the proportion of correctly classified vulnerable samples in all classified vulnerabilities. F1-Measure(F1) = $2 \cdot \mathrm{P} \cdot \mathrm{R}/(\mathrm{P} + \mathrm{R})$, is the harmonic average of accuracy and recall. The values of the above four indicators range between [0, 1]. In this study, we prefer to achieving FPR and FNR, whose values are closer to 0. For other indicators, the closer their values are to 1, the better the model will achieve.

4.3 Implementation Details

BERT is trained on the common corpus. However, the models trained by this common corpus cannot fully extract the intrinsic meaning of vulnerability source code, so model fine-tuning is required. We focus on fine-tuning the pre-trained BERT model and applying it to vulnerability vector representation. In this work, we choose the BERT-Base (Uncased: 12-layer, 768-hidden, 12-heads, 110M parameters) pre-trained model rather than the BERT-Large, because, in our experiment, the BERT-Base is already sufficient to the dataset. The hyperparameters remain as the default settings and the maximum length of the inputs is 128. After the pre-processing, variable names and function names are converted into symbolic representation, which reduces the number of different tokens and improves the efficiency of fine-tuning. In order to avoid overfitting and improve the generalization ability of the model, the output information of the penultimate hidden layer of the Vectorization model is selected by using embeddings. The vector dimension of Vectorization is 128×768, where the vector dimension of one word is 768 and the number of words to represent a sample is 128.

In this paper, the hyper-parameters tuned for SAN learning are shown in Table 3. The neural networks are implemented in Python using Keras and the distributed embeddings are implemented from Vectorization by fine-tuning BERT. Our experiments were run on a machine with TITAN V GPU.

Table 3. The parameters for SAN learning.

Parameter	Value
Batch size	128
Hidden layers	5
Learning rate	0.001
Sample length	128

4.4 Results Analysis

Select Neural Networks. Our experiments are carried out on two different types of vulnerability datasets: CWE-399 and CWE-119. Two datasets are performed with the same hyper-parameters and experimental procedures. Through the comparison between SAN and LSTM on the results of vulnerability source codes, it is illustrated that attention mechanism makes a positive impact on vulnerability detection. In order to select the best model for classification learning, we compare SAN and Gated Recurrent Unit with self attention (GRU-attention). Then, we apply the traditional machine learning methods naive Bayes (NB) on the same dataset and compare the classification effects with the previous deep learning models to illustrate the advantages of adopting SAN. The comparison results are shown in Table 4.

It's shown that the SAN module proposed in this paper is superior to the other three classification models. By comparing the table from top to bottom, it can be seen that the vulnerability detection effect of deep learning is significantly better than that of traditional machine learning method. FPR and F1-measure of SAN are better than that of LSTM. Both models use LSTM neural network layer to extract key information of vulnerability source codes. But, the difference between the two models is that one is added the attention layer on the basis of the LSTM layer. The comparison between the both shows that the attention layer highlights important information to improve the performance.

The experimental results of SAN are better than that of the GRU-attention. Both of the two experimental models have the attention layer and the main functions of the attention layer are basically the same, in addition, the model structure is basically the same. The difference is that the first layer of the model is LSTM layer, and the other is GRU layer. Through analysis of the evaluation indexes in this experiment, it can be seen that LSTM performs better than GRU in extracting key information of vulnerability characteristics. Among the above four classifiers, it can be concluded that SAN is better than GRU-attention in the dataset. The attention layer optimizes performance by highlighting key information. Therefore, SAN has a more balanced performance.

Table 4. Comparison of different classification models.

Model	CWE-399				CWE-119			
	FPR(%)	FNR(%)	P(%)	F1(%)	FPR(%)	FNR(%)	P(%)	F1(%)
LSTM	22.2	3.6	90.8	93.5	3.7	8.2	85.1	88.3
GRU-attention	2.4	5.2	93.8	94.3	2.7	7.1	88.8	90.9
NB	5.9	18.4	86.7	84.0	7.3	18.1	72.1	76.7
SAN	**1.1**	**4.4**	**95.2**	**95.4**	**1.3**	**6.7**	**94.3**	**93.8**

Table 5. Comparative experimental results of different embedding tools.

Embedding tool	CWE-399				CWE-119			
	FPR(%)	FNR(%)	P(%)	F1(%)	FPR(%)	FNR(%)	P(%)	F1(%)
word2vec	2.8	4.7	94.6	95.0	2.9	18.0	91.7	86.6
doc2vec	3.8	9.7	91.9	91.1	3.9	16.9	88.1	85.5
Vectorization	**1.1**	**4.4**	**95.2**	**95.4**	**1.3**	**6.7**	**94.3**	**93.8**

Effectiveness of Fine-Tuning. In order to verify the effectiveness of Fine-tuning, we conduct comparative experiments with word2vec and doc2vec. Vectorization is the representation model that is obtained from the fine-tuning stage. It can be seen from Table 5 that our method outperforms word2vec and doc2vec for the same dataset. The doc2vec model is an improvement on the basis of the word2vec model. Both of them are the static approach and cannot be dynamically optimized for specific tasks. It can be also observed that the neural network trained from the CWE-399 datasets outperforms the neural network trained from the CWE-119 datasets in terms of all four metrics. This can be explained by the fact that the number of CWE-399 datasets is far smaller than the number of CWE-119 datasets. What is more, CWE-119 which are the buffer overflow vulnerabilities have more complex forms than CWE-399 which are the datasets of resource management error vulnerabilities.

Table 6. Comparative with other pattern-based vulnerability detection systems.

System	FPR(%)	FNR(%)	P(%)	F1(%)
Flawfinder	47.0	66.2	21.8	26.5
VUDDY	4.1	91.0	45.9	15.0
Vuldeepecker	2.9	18.0	91.7	86.6
Our method	**1.3**	**6.7**	**94.3**	**93.8**

Comparison of Different Vulnerability Detection Systems. For examining the effectiveness of our method, we also perform a comparative experiment with other pattern-based vulnerability detection systems. Open source

tool Flawfinder [4] is the traditional vulnerability detection method based on static analysis. VUDDY [12] is elected because it represents the approach based on code similarity. Vuldeepecker and ours are the deep learning-based system for vulnerability detection. Table 6 summarizes the comparison.

Our method outperforms the other pattern-based vulnerability detection systems on the same dataset. Flawfinder has high FNR (47%) and FPR (66.2%), because vulnerability detection method applies the static vulnerability analysis, which leads to high false positive rate and over-reliance on detection rules. VUDDY incurs low FPR (4.1%) but high FNR (91%), which is the result at a very low value for F1-measure (15.1%). It can be interpreted as VUDDY to detect vulnerabilities by code similarity. However, if it is used for vulnerabilities that are not caused by code cloning, a high false negative rate will occur. Compared with Vuldeepecker, our method performs better on all four metrics. Instead of fine-tuning representation model stage, Vuldeepecker directly applies word2vec model for embeddings. Our method improved by 1.6% in FPR, 11.3% in FNR, 2.6% in P, 7.2% in F1-measure, which are respectively better than their counterparts in Vuldeepecker. It indicates that transfer learning and fine-tuning models for representation can effectively improve the effect of vulnerability detection.

To the best of our knowledge, we are the first to use the fine-tune BERT model for vulnerability representation. The above experimental results show that our method can not only learn semantic meanings of vulnerability source codes but also learn higher-level concepts within the vulnerability source codes such as syntactic structures and semantic roles. In this way, it has the ability to capture the features of vulnerabilities from context and improve vulnerability detection.

The Hyper-parameter Values. In the experiment, there are many hyper-parameters that need to be set and adjusted according to the accuracy and loss rate of the experiment after each iteration. Then, we adopt a 10-fold cross validation to learn the classification model, then select the best hyper-parameters values for vulnerability detection. Figure 5 shows the value of Precision of SAN

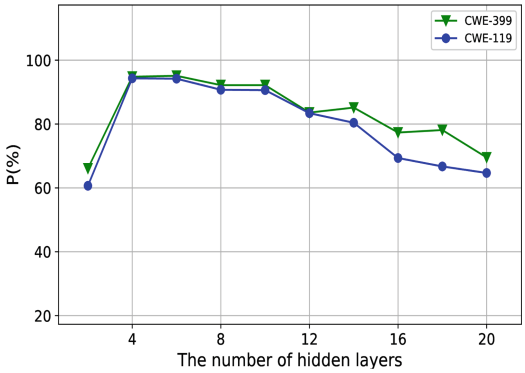


Fig. 5. Precision of SAN for 2 datasets with the different number of hidden layers.

with respect to the 2 datasets affected by the different number of hidden layers. It can be observed that the Precision of both SAN reaches the maximum at the 4 or 5 layers, and the Precision of both SAN declines with the number of layers greater than 7. Note that the other hyper-parameters of SAN can be adjusted in a similar way

The Output of Attention Layer. With the same lines of source code as input, we visualize the output values of the LSTM layer and the attention layer. In the same picture, the depth of color represents the difference of attention. It can be observed in Fig. 6 that the color distribution of LSTM layer is relatively uniform, and that means the difference between output values of LSTM layer is not obvious. The other picture, the difference of color in the attention layer is obvious, which can be considered that more obvious the color difference is, the greater the fluctuation of attention. And it intuitively illustrates that the attention layer can allocate the corresponding weight to highlight the key features of the vulnerability source codes.

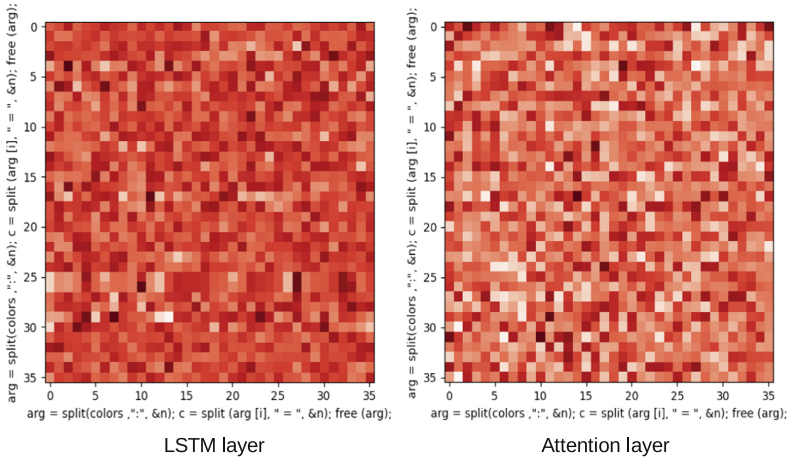
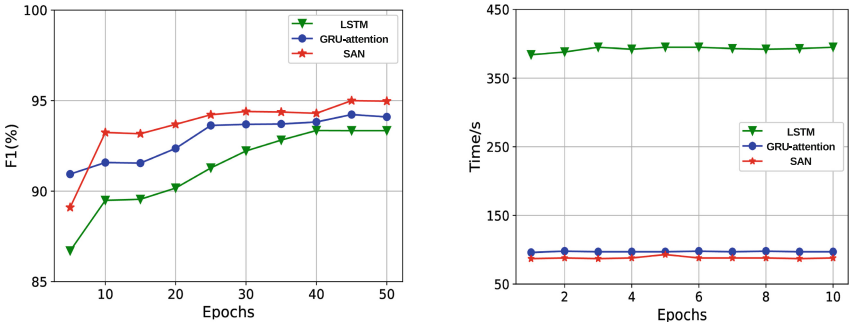


Fig. 6. Compare between LSTM layer and attention layer.

Different Numbers of Epochs. As shown in Fig. 7, three kinds of neural networks: LSTM, SAN, and GRU-Attention are trained on the same training set. In (a), a comparative experiment is performed on the detection set to obtain the relationship between the F1-measures and epochs. The F1-measures of SAN is always higher than other two neural networks. In (b), it shows the trend curve of the time needed to complete the epoch of the three neural networks under the same experimental condition. It can be seen that the epoch time of each neural network does not fluctuate much on the whole, and the overall time tends to be stable. Generally, after the minimum epoch time is passed, the training time will

no longer fluctuate greatly during the retraining. It can be seen that the curves of SAN and GRU-attention model are relatively close, but SAN takes less time to complete an iterative training and its training speed is relatively fast.



(a) F1-measures of three neural networks with the different number of epochs.

(b) Epoch time of three neural networks.

Fig. 7. Iterative experiments across three kinds of neural networks

5 Conclusion

This paper introduces a novel approach for detecting vulnerability from source code and shows effective improvements when learning vulnerable programming patterns automatically. Through fine-tuning the pre-trained model and applying it to embeddings, our proposed method extracts syntactic and semantic information of vulnerabilities in source code context. It alleviates the difficulty in describing the features of software vulnerabilities accurately and provides precise information for vulnerability detection. By transfer learning, we obtain the vulnerability representation model from the pre-trained BERT on large-corpus task. And it is also confirmed that the effectiveness of transfer learning has improved in methodology through experimental results.

The SAN module is used for learning and training vulnerability samples to get the detection model in our methods. Different from general LSTM without attention layer, SAN can capture key points of vulnerabilities and perform better on the precision, where these experimental conclusions aim to provide some guidelines for researchers to choose neural networks in future vulnerability detection studies. And it is shown that our method has been superior to the state-of-the-art methods.

Acknowledgment. This paper is supported by the Major Research Project of National Natural Science Foundation of China (No. 91948303).

References

1. Common vulnerabilities and exposures. https://cve.mitre.org. Accessed 4 Jan 2021
2. Cppcheck: a tool for static C/C++ code analysis. https://cppcheck.sourceforge.net. Accessed 4 Jan 2021
3. Doc2vec. https://radimrehurek.com/gensim/models/doc2vec.html. Accessed 4 Oct 2017
4. Flawfinder software official website. https://www.dwheeler.com/flawfinder/. Accessed 4 Jan 2021
5. National vulnerability database. https://nvd.nist.gov. Accessed 4 Jan 2021
6. Static analysis with CodeSonar. https://www.grammatech.com/products/source-code-analysis. Accessed 4 Jan 2021
7. Word2vec. https://radimrehurek.com/gensim/models/word2vec.html. Accessed 4 Jan 2021
8. Brooks, T.N.: Survey of automated vulnerability detection and exploit generation techniques in cyber reasoning systems. CoRR abs/1702.06162 (2017)
9. Cho, K., van Merrienboer, B., Bahdanau, D., Bengio, Y.: On the properties of neural machine translation: encoder-decoder approaches. In: Wu, D., Carpuat, M., Carreras, X., Vecchi, E.M. (eds.) Proceedings of SSST@EMNLP 2014, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation, Doha, Qatar, 25 October 2014, pp. 103–111. Association for Computational Linguistics (2014). https://doi.org/10.3115/v1/W14-4012
10. Devlin, J., Chang, M., Lee, K., Toutanova, K.: BERT: pre-training of deep bidirectional transformers for language understanding. In: Burstein, J., Doran, C., Solorio, T. (eds.) Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2–7 2019, Volume 1 (Long and Short Papers), pp. 4171–4186. Association for Computational Linguistics (2019). https://doi.org/10.18653/v1/n19-1423
11. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Comput. **9**(8), 1735–1780 (1997). https://doi.org/10.1162/neco.1997.9.8.1735
12. Kim, S., Woo, S., Lee, H., Oh, H.: VUDDY: a scalable approach for vulnerable code clone discovery. In: 2017 IEEE Symposium on Security and Privacy (SP), pp. 595–614 (2017). https://doi.org/10.1109/SP.2017.62
13. Le, Q.V., Mikolov, T.: Distributed representations of sentences and documents. In: Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21–26 June 2014. JMLR Workshop and Conference Proceedings, vol. 32, pp. 1188–1196. JMLR.org (2014)
14. Li, Z., Zou, D., Tang, J., Zhang, Z., Sun, M., Jin, H.: A comparative study of deep learning-based vulnerability detection system. IEEE Access **7**, 103184–103197 (2019). https://doi.org/10.1109/ACCESS.2019.2930578
15. Li, Z., Zou, D., Xu, S., Jin, H., Qi, H., Hu, J.: VulPecker: an automated vulnerability detection system based on code similarity analysis. In: Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, pp. 201–213. Association for Computing Machinery, New York (2016). https://doi.org/10.1145/2991079.2991102

16. Li, Z., et al.: SySeVR: a framework for using deep learning to detect software vulnerabilities. CoRR abs/1807.06756 (2018)
17. Li, Z., et al.: VulDeePecker: a deep learning-based system for vulnerability detection. In: 25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, 18–21 February 2018. The Internet Society (2018)
18. Mikolov, T., Sutskever, I., Chen, K., Corrado, G.S., Dean, J.: Distributed representations of words and phrases and their compositionality. In: Burges, C.J.C., Bottou, L., Ghahramani, Z., Weinberger, K.Q. (eds.) Advances in Neural Information Processing Systems 26: 27th Annualsplncs0 Conference on Neural Information Processing Systems 2013. Proceedings of a Meeting Held 5–8 December 2013, Lake Tahoe, Nevada, United States, pp. 3111–3119 (2013)
19. Mnih, V., Heess, N., Graves, A., Kavukcuoglu, K.: Recurrent models of visual attention. In: Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N.D., Weinberger, K.Q. (eds.) Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, 8–13 December 2014, Montreal, Quebec, Canada, pp. 2204–2212 (2014)
20. Pagliardini, M., Gupta, P., Jaggi, M.: Unsupervised learning of sentence embeddings using compositional n-gram features. In: Walker, M.A., Ji, H., Stent, A. (eds.) Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2018, New Orleans, Louisiana, USA, 1–6 June 2018, Volume 1 (Long Papers), pp. 528–540. Association for Computational Linguistics (2018). https://doi.org/10.18653/v1/n18-1049
21. Pang, Y., Xue, X., Namin, A.S.: Predicting vulnerable software components through N-gram analysis and statistical feature selection. In: Li, T., et al. (eds.) 14th IEEE International Conference on Machine Learning and Applications, ICMLA 2015, Miami, FL, USA, 9–11 December 2015, pp. 543–548. IEEE (2015). https://doi.org/10.1109/ICMLA.2015.99
22. Pendleton, M., Garcia-Lebron, R., Cho, J., Xu, S.: A survey on systems security metrics. ACM Comput. Surv. **49**(4), 62:1–62:35 (2017). https://doi.org/10.1145/3005714
23. Qiu, X., Sun, T., Xu, Y., Shao, Y., Dai, N., Huang, X.: Pre-trained models for natural language processing: a survey. CoRR abs/2003.08271 (2020)
24. Russell, R.L., et al.: Automated vulnerability detection in source code using deep representation learning. In: Wani, M.A., Kantardzic, M.M., Mouchaweh, M.S., Gama, J., Lughofer, E. (eds.) 17th IEEE International Conference on Machine Learning and Applications, ICMLA 2018, Orlando, FL, USA, 17–20 December 2018, pp. 757–762. IEEE (2018). https://doi.org/10.1109/ICMLA.2018.00120
25. Yamaguchi, F., Maier, A., Gascon, H., Rieck, K.: Automatic inference of search patterns for taint-style vulnerabilities. In: 2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, 17–21 May 2015, pp. 797–812. IEEE Computer Society (2015). https://doi.org/10.1109/SP.2015.54
26. Zhou, Y., Liu, S., Siow, J.K., Du, X., Liu, Y.: Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In: Wallach, H.M., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E.B., Garnett, R. (eds.) Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8–14 December 2019, Vancouver, BC, Canada, pp. 10197–10207 (2019)

27. Zou, D., Wang, S., Xu, S., Li, Z., Jin, H.: μvuldeepecker: a deep learning-based system for multiclass vulnerability detection. IEEE Trans. Dependable Secure Comput., 1 (2019). https://doi.org/10.1109/TDSC.2019.2942930
28. Zhang, H., Cai, J., Xu, J., Wang, J.: Complex question decomposition for semantic parsing. In: Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics, pp. 4477–4486. Association for Computational Linguistics, Florence. Italy, July 2019. https://aclanthology.org/P19-1440. https://doi.org/10.18653/v1/P19-1440