IEEE INFOCOM 2022 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS) | 978-1-6654-0926-1/22/$31.00 ©2022 IEEE | DOI: 10.1109/INFOCOMWKSHPS54753.2022.9798297

Hierarchical Attention Network for Interpretable and Fine-Grained Vulnerability Detection

Mianxue Gu*[1,3], Hantao Feng*[2,3], Hongyu Sun[2,3], Peng Liu[4], Qiuling Yue[1],
Jinglu Hu[5], Chunjie Cao[1], Yuqing Zhang[1,2,3]
[1]School of Cyberspace Security, Hainan University, Haikou, China, zhangyq@nipc.org.cn
[2]College of Cyber Engineering, Xidian University, Xi'an, China
[3]National Computer Network Intrusion Protection Center, University of Academy of Sciences, Beijing, China
[4]College of Information Sciences and Technology, The Pennsylvania State University, United States
[5]Graduate School of Information, Production and Systems, Waseda University, Japan

***Abstract*—With the rapid development of software technology, the number of vulnerabilities is proliferating, which makes vulnerability detection an important topic of security research. Existing works only focus on predicting whether a given program code is vulnerable but less interpretable. To overcome these deficits, we first apply the hierarchical attention network into vulnerability detection for interpretable and fine-grained vulnerability discovery. Especially, our model consists of two level attention layers at both the line-level and the token-level of the code to locate which lines or tokens are important to discover vulnerabilities. Furthermore, in order to accurately extract features from source code, we process the code based on the abstract syntax tree and embed the syntax tokens into vectors. We evaluate the performance of our model on two widely used benchmark datasets, CWE-119 (Buffer Error) and CWE-399 (Resource Management Error) from SARD. Experiments show that the F1 score of our model achieves 86.1% (CWE-119) and 90.0% (CWE-399) on two datasets, which is significantly better than the-state-of-the-art models. In particular, our model can directly mark the importance of different lines and different tokens, which can provide useful information for further vulnerability exploitation and repair.**

***Index Terms*—vulnerability detection, abstract syntax tree, hierarchical attention network, deep learning**

I. Introduction

Software vulnerability detection is an important topic in software security research. There are tens of thousands of software vulnerabilities reported each year in the CVE (Common Vulnerabilities and Exposures) database[1] or other vulnerability databases, and many of them pose a massive risk to cyberspace security. There are currently more than 28 million open-source software projects published on Github[2]. The popularity of open-source makes it possible to fully share technology between different projects, but it may also lead to new security problems due to code copying, code cloning, or import of dependent libraries. Therefore, discovering vulnerabilities from program source code has become a hot topic in current security vulnerability research.

In order to improve the efficiency of vulnerability analysis, there already exist some approaches to analyze the program from source code or binary files. The dynamic analysis extracts information by running the programs, such as fuzzing [1], symbolic execution [2] and taint analysis [3]. These methods can effectively extract vulnerability information, but bring a lot of resource overhead, and also require experts to analyze the vulnerability manually. The static analysis extracts information from program source code or bytecode, such as rule-based analysis[3] and code-clone detection [4] [5] [6]. The static analysis does not require running programs, so it is suitable to be combined with machine learning to analyze a large number of programs. Combining static and dynamic analysis methods is currently the most common software analysis method [7] [8], but these methods also need the predefined program rules for vulnerability detection. In general, all of these methods are limited to capture features from new programs so that they can not take a quick response to new vulnerabilities.

Now deep learning has achieved many successes in the natural language process (NLP). The recurrent neural network (RNN) has a strong ability to process long sequences [9]. To better understand text sequences, attention mechanisms have been widely used in RNN [10] to improve the accuracy of information extraction. By applying the attention mechanism at the sentence level and the word level [11], it is possible to distinguish the important information in different sentences and different words.

To this end, in order to achieve interpretable and fine-grained vulnerability detection, we construct a hierarchical attention network [11] to learn code features from their vector representations. The code structure is similar to document structure, the tokens from lines and the lines from code blocks. We implement the hierarchical attention network that consists of two BGRU layers with attention mechanisms. By applying attention mechanisms at the line-level and the token-level, our model is well interpretable. These structures let the model not only pay attention to different lines but also to different tokens to locate the vulnerable points more accurately.

*Both authors contributed equally to this work, and Yuqing Zhang is the corresponding author.

[1]https://cve.mitre.org/

[2]https://github.com/

[3]https://dwheeler.com/flawfinder/

978-1-6654-0926-1/22/$31.00 ©2022 IEEE

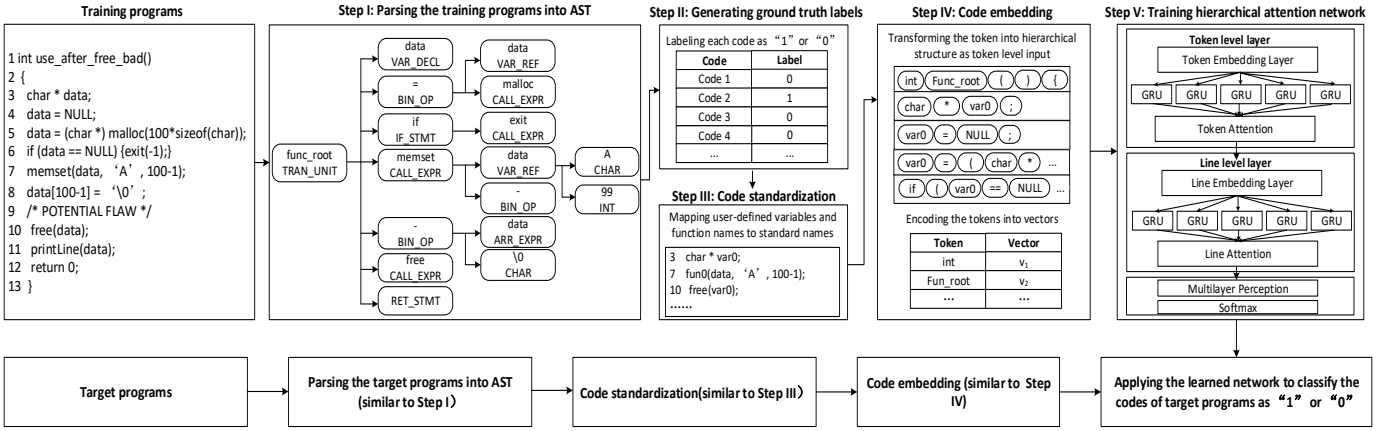

Fig. 1. The overview of our model. (a) The training phase generates vulnerability patterns and is centered on obtaining program parsing, code standardization, code embedding and model learning. (b) The detection phase takes advantage of these vulnerability patterns to classify whether a given target program is vulnerable or not.

Intuitively, since the program source code is a structured text, it cannot be directly inputted into the neural network for training. We present a complete processing method for embedding program source code into vector representations without predefined rules and preserve all of the syntax features. To overcome this challenge, we parse the program source code into AST (Abstract Syntax Tree) and slice the function node from it. Then we traverse the entire AST and map all user-defined names to fixed names. To learn this data by a hierarchical attention network, we extract all tokens from the function-level AST and arrange the token sequence in the order of the lines in the original code. By using this method, we extract the function-level code, remove redundant information, and retain complete syntax information. Then we apply the Word2Vec model [12] to map the tokens into vectors that can be directly trained by neural networks.

We evaluate our model based on two SARD[4] datasets, CWE-119 (Buffer Error) and CWE-399 (Resource Management Error). These datasets built by SARD are widely used as the benchmark dataset to evaluate vulnerability detection models [13] [14]. Compared with other vulnerability detection tools and models, experimental results show that our model is more effective than the state-of-the-art methods. In particular, our model can directly extract the critical features from different lines and different tokens by applying attention mechanism and provide useful information for further vulnerability exploitation and repair.

Our main contributions are as follows.

- A technical innovation is that we parse the program source code into AST to reduce the data redundancy and preserve more syntax features. In addition, we apply the pack-padded method to handle variable-length data without any truncation or padding on model training.
- Another innovation underlying our proposed model is a novel hierarchical attention network structure. The improved model is mainly constructed from the building-block BGRU network and aims to pinpoint the important codes for discovering fine-grained vulnerabilities both from the line-level and the token-level, which embodies the advantages of hierarchical attention network.
- In order to demonstrate the effectiveness of our proposed model, we evaluate our model on two benchmark datasets built by SARD, and the experiments show that our model can achieve much higher performance than the state-of-the-art methods and is highly interpretable.

II. Design of Our Model

In order to realize an interpretable and fine-grained vulnerability detection model, we take advantage of both program analysis and deep learning techniques to predict whether the given source code is vulnerable or not. Figure 1 depicts an overview of the whole model. The proposed model mainly focuses on two components, namely training phase and detection phase.

A. Training Phase

As highlighted in Figure 1 (a), the training phase has five steps.

Step I: Parsing the source code into AST. Due to the widespread application of C/C++ language and the convenience of data collection, our research is based on C/C++ source code. For robust code parsing, we use Clang[5] as the code compiler front to parse AST from source code. AST is an equivalent representation of program source code, which contains all the syntax information of the code. The AST node contains two major properties, the name of the node and the kind of the node. The name of the node is the display name of the node, for the variable node, the name of the node is the variable name like "data". The kind of the node represents the current type of the node, for example, the kind of variable "data" in the declare statement is "VAR_DECL", the kind of variable "data" in the reference statement is

[4]https://samate.nist.gov/SARD/

[5]https://clang.llvm.org/

"VAR_REF_EXPR". So the function node from AST can be extracted by checking their kind attribute.

Step II: Generating ground truth labels of training programs. This step labels each code as "1" (i.e., vulnerable) or "0" (i.e., not vulnerable). These labels are served as the ground truth for training an interpretable and fine-grained vulnerability detection model. All these functions of our used vulnerability dataset are labeled by their function names with Non-vulnerable or vulnerable suffix string, so we can easily generate ground truth labels of training programs.

Step III: Code standardization. In a real environment, due to the different coding styles of different programmers, functions with similar abilities may have significant differences in code. In order to extract the same information from different styles of code, we need to standardize similar function codes. For functions extracted from AST, we first map the user-defined variable names and the function names to fixed names, such as "var0" and "fun0". If a new token is not included in the word embedding model, we directly map it into zero vector. This processing method allows us to represent unlimited source code with limited tokens, and it is easy to identify functions with the same structure.

Step IV: Code Embedding. After standardizing the function code, we need to convert the code into a vector representation. For the hierarchical attention network, we also need to arrange the token sequence into a hierarchical structure in the order of the original lines. All of the syntax tokens can be obtained by traversing the entire AST in preorder. Then we use the semicolon and the left brace as separators to re-divide the token sequence into hierarchical data and maintain the order of lines and tokens in the original code. Then we collect all of these tokens to training the Word2Vec [12] model that maps every token into a suitable vector representation. We trained the Word2Vec model in default settings, the word vector size is 100, and the training algorithm is CBOW because it faster than Skip-Gram and has the same results in our experiments.

Step V: Training a hierarchical attention network. After encoding the code into vectors and generating their ground truth labels, the process for training a hierarchical attention network is standard. The designed system is used for interpretable and fine-grained vulnerability detection.

The overview of our neural network model is shown in Figure 2. In this paper, we implement a hierarchical attention network with two-level layers, a token-level layer and a line-level layer. Both of these layers have similar parts, a BGRU encoder with attention mechanism. In addition, we apply the pack padded method to all GRU encoders to allow the model can handle variable-length data. In order to fully retain the final vectors of source code, we apply the pack padded method in both the line-level and the token-level GRU encoders to allow the RNN layer to handle variable-length input data. The pack padded method is shown in Figure 3. For each batch, we first sort them according to their vector length and record the original order. For the sorted batch, we pack them in one sequence according to time series to remove the padding vectors. In this way, RNN can process variable-length data without being affected by truncation and padding operations. In the actual calculation, the padding of each batch of samples is only used to conveniently process the vectors and does not participate in the actual calculation.

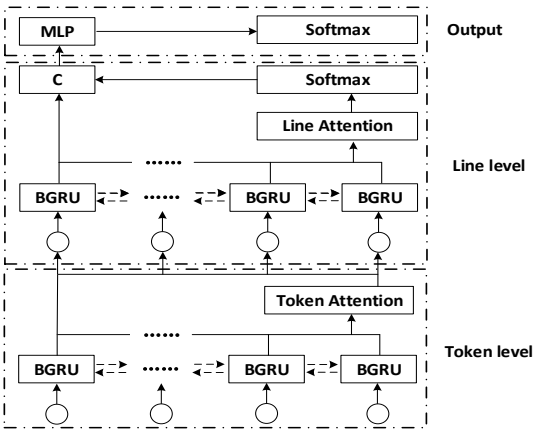

Fig. 2. The structure of our two-level attention network.

B. Detection Phase

A learned hierarchical attention network can detect whether given vectors corresponding to the codes are vulnerable or not. When a vector is classified as "1", it means that the corresponding code is vulnerable. Otherwise, the corresponding code is classified as "0" (i.e., not vulnerable). As highlighted in Figure 1(b), this phase has the following steps.

Step I: Parsing the target programs into AST. This step extracts the function nodes of parsed AST from target programs (and is similar to Step I in training phase).

Step II: Code standardization. This step maps user-defined variables and function names to same values (and is similar to Step III in training phase).

Step III: Code Embedding. This step takes advantage of word embedding technique to convert normalized codes to fixed-length vectors (and is similar to Step IV in training phase).

Step IV: Using the learned deep learning network to detect given target programs. This step uses the learned deep learning network to classify given vectors corresponding to the codes that are extracted from the target programs as "1" (i.e., vulnerable) or "0" (i.e., not vulnerable).

aa	bb	cc	dd
aa	bb	cc	<pad>
aa	bb	<pad>	<pad>
aa	<pad>	<pad>	<pad>

Pack:	aa	aa	aa	aa	bb	bb	bb	cc	cc	dd

Length: [4, 3, 2, 1]

Fig. 3. Pack input vectors by time series

TABLE I
STATISTICS OF THE TWO BENCHMARK DATASETS

Dataset	Functions	Vulnerable	Non-vulnerable
SARD_CWE119	30750	12492	18258
SARD_CWE399	29289	9794	19495

III. EXPERIMENTS

In this section, we evaluate the performance of our model on some widely used benchmark datasets and compare it with some state-of-the-art methods.

a) Experimental Environment: We implement the hierarchical attention network in Python by using Pytorch and using Gensim to training pre-trained word embedding models. We run our experiments on a server with NVIDIA GeForce RTX 2070 GPU and Intel Xeon E5-2650 v4 CPU.

b) Data Preparation: We select the widely used vulnerability dataset - SARD to fully evaluate the performance of our model. All these functions are labeled by their function names with Non-vulnerable or vulnerable suffix string, so we can easily use this suffix information to generate the label of each function. In this paper, we focus on two types of vulnerabilities, CWE-119 (Buffer Error) and CWE-399 (Resource Management Error), which are very common and are widely used as benchmark datasets. All samples were generated from the raw dataset using our data processing method.

c) Evaluation Metrics: We use some widely used metrics to evaluate the ability of our model on vulnerability detection, including Precision (P), Recall (R), False Positive Rate (FPR), False Negative Rate (FNR), Precision-Recall Area Under Curve (PR AUC) and F1 Score (F1). The high precision and low false positive rate ensure that every vulnerability we found is valuable. If the model with a high false positive rate, we need to spend more extra time to identify the valuable results of all predictions, we can say that this model is not usable. In this paper, we try to keep the false positive rate as low as possible while keeping the low false negative rate, high precision and PR AUC .

d) Experiment Settings: We randomly divide the dataset into three parts, a training set, a validation set, and a test set, the ratio of the partition is $6:2:2$, this ratio is applied equally to every type of vulnerabilities.

We trained each dataset on the hierarchical attention network we described above. The word vector size of Word2Vec is 100, so the input size of the token-level GRU is 100 too. The hidden size of token-level attention and line-level attention is set to 100. The number of layers in the token-level GRU and the line-level GRU are set to 2. The dropout is set to 0.5. We tested a variety of gradient descent algorithms, include SGD, Adadelta [16], and Adam [17], and finally used the mini-batch stochastic gradient descent with Adam optimization, it is faster than other algorithms and has the same result. The batch size is set to 64 and the learning rate is set to 0.0002. We use the Cross-Entropy as the loss function. In addition to the parameters we mentioned above, we use the default options for other parameters. The pack padded sequence method is applied at both the line-level and the token-level to handle variable-length data.

TABLE II
TIME OVERHEAD OF VULNERABILITY DETECTION MODELS

Datasets	Models	Training time	Validation time
CWE-199	Rats	*	6m32s
	Flawfinder	*	8m19s
	VulDeePecker	10h6m12s	2m36s
	Ours	**51m35s**	**27s**
CWE-399	Rats	*	6m02s
	Flawfinder	*	8m11s
	VulDeePecker	7h42s	1m16s
	Ours	**59m59s**	**14s**

A. Experimental Results

In order to fully demonstrate the capabilities of our model, we compared it with two existing open-source vulnerability detection tools, Rats[6] and Flawfinder[7], and two state-of-the-art deep learning models, VulDeePecker [13] and VulSniper [14]. RATS and Flawfinder a re s tatic a nalysis t ools t hat discover vulnerabilities based on predefined r ules. V ulDeePecker and VulSniper are two state-of-the-art vulnerability detection models based on deep learning. The reason we chose these two models is that, to the best of our knowledge, VulDeePecker is the first m odel t o a pply d eep l earning i n vulnerability detection, and VulSniper is the first m odel t o u se attention mechanism. These two models are excellent benchmarks for evaluating the capabilities of our model. Since VulDeePecker does not provide its source code, we implemented the model ourselves for comparison.

To demonstrate the potential of deep learning in the field of vulnerability detection, we summarize the time overhead of different vulnerability detection models on two data sets in the Table II. For RATS and Flawfinder, t hese t ools have a significant a dvantage o n t otal t ime o verhead. B ut a fter the training is completed, the deep learning model has higher efficiency than these tools. Our model can detect 6150 samples in 27 seconds, which can definitely prove the potential of deep learning-based vulnerability detection models when dealing with large amounts of data.

The comparison results of vulnerability detection are shown in Table III. We can see that the deep learning-based model has a huge advantage in the effect of vulnerability detection compared to two static analyzers based on predefined rules. It can be explained that RATS and FlawFinder only detect vulnerabilities by calculating the similarity of the code and comparing it with predefined rules, which leads to low F1 score and high FPR and FNR. There is no doubt that deep learning has huge advantages and potential in the field of vulnerability detection.

[6]https://security.web.cern.ch/
[7]https://dwheeler.com/flawfinder/

TABLE III
PERFORMANCE COMPARISONS OF OUR MODEL WITH DIFFERENT TOOLS ON THE TWO DATASETS

Dataset	Models	P (%) (↑)	R (%) (↑)	FNR (%) (↓)	FPR (%) (↓)	PR AUC (%) (↑)	F1 (%) (↑)
CWE119	RATS	43.5	78.2	21.7	51.7	34.0	55.9
	FlawFinder	43.2	49.9	50.0	33.9	21.6	46.3
	VDP	85.4	71.9	28.0	4.41	61.4	78.1
	VulSniper	88.7	73.9	26.2	6.42	65.5	80.6
	Ours	**94.3**	**79.2**	**20.7**	**3.24**	**74.7**	**86.1**
CWE399	RATS	37.4	62.9	37.3	46.9	23.5	46.9
	FlawFinder	41.0	43.9	56.2	26.9	18.0	42.4
	VDP	77.0	81.7	18.3	12.2	62.9	79.3
	VulSniper	80.4	67.4	32.7	8.49	54.2	73.3
	Ours	**96.1**	**84.6**	**15.3**	**1.75**	**81.3**	**90.0**

Compared with other state-of-the-art deep learning-based models and two static analyzers, it is obvious that our model has advantages in terms of precision, false positive rate and PR AUC. For VulDeePecker, in order to reduce data redundancy and keep the fixed input length, VulDeePecker truncates the final vector, which causes a lot of information loss. Therefore, the granularity of feature extraction of VulDeePecker is relatively rough, which results in low precision and high false negative rate. VulSniper parses program source code into CPG, which destroys the original structure of the code, resulting in low precision and challenging to interpret the prediction results. To overcome these problems, our model processes program source code based on the AST to reduce data redundancy and retain all syntax information. Then we construct a hierarchical attention network extract features at both the line-level and the token-level, which allows us to notice which lines or tokens are important for vulnerability discovery. Therefore, our model can penetrate the details of the program source code for fine-grained vulnerability detection.

Through the attention mechanism, we can analyze the effect of our model on different vulnerabilities in more detail. It can be seen from the experimental results that our model on CWE-399 is significantly better than those on CWE-119 , which can be explained by the attention mechanism. As shown in Figure 4, we can see that for CWE-119 and CWE-399, attention mechanism can focus on important lines and tokens. The buffer error vulnerabilities have a great relationship with the size of the variable values. Based on the static analysis method, our model does not know the size relationship between different values, which makes it difficult to achieve good results on many buffer error vulnerabilities. As shown in Figure 4 (b), the pointer is freed twice incorrectly, which will cause a heap overflow attack. Our model can accurately locate the allocation and free of different variables, so it can correctly predict whether the code is vulnerable.

IV. RELATED WORK

Early automated vulnerability detection methods highly rely on security researchers manually extract vulnerability features for vulnerability detection. There existing many tools based on predefined rules to discover vulnerabilities, open-source tools such as RATS and Flawfinder, commercial tools such as Checkmarx[8] and Coverity[9]. These tools extract program features by lexical analysis and compare it with predefined rules to detect vulnerabilities.

Existing studies concerning vulnerabilities mainly focus on machine learning [6] [18] or deep learning models [13] [14] [15] [19] [20] [21]. And these methods have similar processes, including a data processing method to transform the code into a vector, a neural network model to train, and a fully connected layer with softmax to output the binary classification results. Li et al. proposed VulDeePecker that slices the source code into code blocks and directly embed the source code into vectors. Then they construct a Bi-LSTM network for training [11]. Duan et al. proposed the VulSniper that parsed code into Code Property Graph and construct the attention network for training, the F1 score of VulSniper achieves 80.6% (CWE-119) and 73.3% (CWE-399) on two benchmark dataset [14]. Russel et al. extract code features both in the CFG (Control Flow Graph) and vector representations extract from code text. Then they build three models, the Bag-of-Words vectors with Extra Trees classifiers, the Word2Vec vectors with TextCNN, and the Word2Vec vectors with TextCNN and Extra Tree classifiers. Experiments show that the TextCNN with Extra Tree classifiers has the best results [21]. Although these methods mentioned above have successfully improved vulnerability detection capabilities and demonstrated the potential of deep learning, it is still challenging to achieve good results in the real environment due to the complexity of the vulnerability features.

V. LIMITATIONS

Vulnerability detection based on program source code is still a challenging research direction. There exist some limitations in our vulnerability detection model, which suggest some open problems for future research.

First, our model is limited to discovering vulnerabilities involving multiple functions and files. Current data processing methods can only process a single function-level program source code, which limits our model to find complex vulnerabilities. It is an important work to improve the granularity

[8]https://www.checkmarx.com/
[9]https://scan.coverity.com/

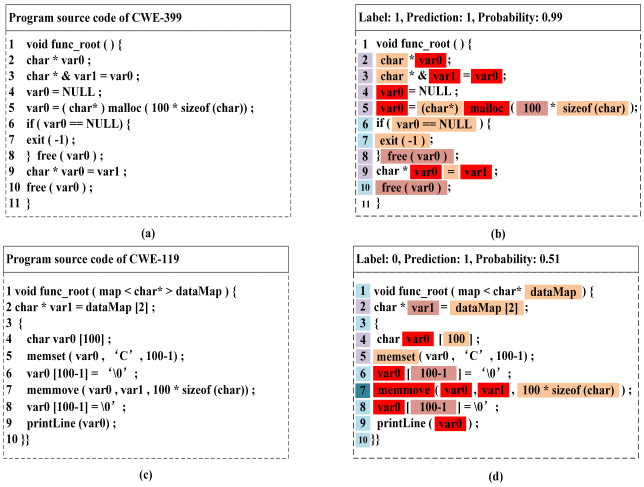

Fig. 4. (a) Program source Code of CWE-119 without Masks; (b) Program source Code of CWE-119 with Varying Degrees of Masks, both from different tokens and lines. (c) Program source Code of CWE-399 without Masks; (d) Program source Code of CWE-399 with Varying Degrees of Masks, both from different tokens and lines.

of the data processing method to enhance the ability of vulnerability detection.

Second, the data processing method in this paper is based on static analysis technique, which makes our model ineffective in determining the boundary of the overflow problem. Therefore, discovering which types of vulnerabilities are more suitable for detection by using deep learning-based techniques is also a good topic of future research.

Third, we still lack datasets that can reflect the real code environment, which limits the further development of vulnerability detection. We also need to test whether our framework can detect vulnerabilities in a real environment.

VI. Conclusion

In this paper, we build a deep learning-based vulnerability detection model to predict vulnerabilities from program source code. To achieve interpretable and fine-grained vulnerability detection, we first apply the hierarchical attention network into vulnerability detection to accurately locate the vulnerable points on program source code. In addition, we embed the program source code into their vector representations at the function-level based on the abstract syntax tree, which can retain all the syntax information and reduce redundancy. Experimental results show that our model has better performance than the state-of-the-art methods. By applying the two-level attention mechanism, our model can directly locate the important code at the line and the token-level, which can improve the efficiency of vulnerability exploitation and repair.

Acknowledgment

This work was supported by the Key Research and Development Science and Technology of Hainan Province (ZDYF202012), and the National Natural Science Foundation of China (U1836210).

References

[1] M. Sutton, A. Greene, and P. Amini. Fuzzing: brute force vulnerability discovery, *Addison-Wesley Professional*, 2007.

[2] T. Wang, T. Wei, Z. Lin, and W. Zou. IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution. In *Proceedings of the Network and Distributed System Security Symposium*. Internet Society, 2009.

[3] J. Newsome, and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the Network and Distributed System Security Symposium*. Internet Society, 2005.

[4] J. Jang, A. Agrawal and, D. Brumley. ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions. *IEEE Symposium on Security and Privacy*, pages 48-62. IEEE, 2012.

[5] S. Kim, W.Seunghoon, L. Heejo, and O. Hakjoo. Vuddy: A scalable approach for vulnerable code clone discovery. *IEEE Symposium on Security and Privacy*, pages 595–614. IEEE, 2017.

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

[7] G. Grieco, G. Grinblat, U. Lucas, R. Sanjay, F. Josselin, and M. Laurent. Toward Large-Scale Vulnerability Discovery using Machine Learning. In *Proceedings of the Sixth ACM on Conference on Data and Application Security and Privacy*, pages 85-96. ACM, 2016.

[8] H. Kihong, o. Hakjoo, and Y. Kwangkeun. Machine-learning-guided selectively unsound static analysis. In *Proceedings of the 39th International Conference on Software Engineering*, pages 519-529, 2017.

[9] S. Hochreiter, and J. Schmidhuber. Long Short-Term Memory. *Neural Computation*. 9(8):1735–1780, 1997.

[10] D. Bahdanau, K. Cho, and Y. Bengio. Neural Machine Translation by Jointly Learning to Align and Translate. In *Proceedings of the 3rd International Conference on Learning Representations*, 2015.

[11] H. Feng, X. Fu, H. Sun, H. Wang, and Y. Zhang. Efficient Vulnerability Detection based on abstract syntax tree and Deep Learning. In *Proceedings of the IEEE Conference on Computer Communications Workshops*, pages 722-727. IEEE, 2020.

[12] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and D. Jeffrey. Distributed Representations of Words and Phrases and their Compositionality. In *Proceedings of the 27th Annual Conference on Neural Information Processing Systems*, pages 3111–3119. ACM, 2013.

[13] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium*. Internet Society, 2018.

[14] X. Duan, J. Wu, S. Ji, Z. Rui, T. Luo, M. Yang, and Y. Wu. VulSniper: Focus Your Attention to Shoot Fine-Grained Vulnerabilities. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, pages 4665–4671. AAAI Press, 2019.

[15] G. Lin, J. Zhang, W. Luo, L. Pan, Y. Xiang, Y. Olivier, and M. Paul. Cross-Project Transfer Representation Learning for Vulnerable Function Discovery. *IEEE Transactions on Industrial Informatics*. 14(7):3289–3297, 2018.

[16] M. D. Zeiler. ADADELTA: An Adaptive Learning Rate Method. *arXiv preprint arXiv:1212.5701v1*, 2012.

[17] D. Kingma, and B. Jimmy. Adam: A Method for Stochastic Optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[18] F. Yamaguchi, N. Golde, A. Daniel and R. Konrad. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 590–604. IEEE, 2014.

[19] H. Wang, G. Ye, Z. Tang, S. H. Tan, and S. Huang. Combining Graph-Based Learning With Automated Data Collection for Code Vulnerability Detection. *IEEE Transactions on Information Forensics and Security*. 16:1943-1958, 2021.

[20] L. Cui, Z. Hao, Y. Jiao, H. Feng, and X. Yun. VulDetector: Detecting Vulnerabilities Using Weighted Feature Graph Comparison. *IEEE Transactions on Information Forensics and Security*. 16:2004-2017, 2021.

[21] R. L. Russell, L. Y. Kim, L. H. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. M. Ellingwood and M. W. McConley. Automated Vulnerability Detection in Source Code Using Deep Representation Learning. In *Proceedings of the 17th International Conference on Machine Learning and Applications*, pages 757–762, IEEE, 2018.