Efficient Vulnerability Detection based on abstract syntax tree and Deep Learning

Hantao Feng, Xiaotong Fu, Hongyu Sun, He Wang, Yuqing Zhang*
*School of Cyber Engineering, Xidian University, Xi'an, China, zhangyq@ucas.ac.cn
National Computer Network Intrusion Protection Center,
University of Chinese Academy of Sciences, Beijing, China

***Abstract*—The automatic vulnerability detection on program source code is an important research topic. With the development of artificial intelligence, deep learning has been applied to vulnerability detection. Existing methods do not make full use of the syntax structure of the program source code that only treats the code as plain text, which brings much redundancy. Moreover, to avoid computation overhead caused by redundancy, existing methods often use the truncate method to process variable-length data, which also cause data loss. In this paper, we propose a data processing method based on the abstract syntax tree to extract all syntax features and reduce data redundancy. Besides, we apply the pack-padded method on the Bi-GRU network to train variable-length data without truncation and padding. Compared with the current methods, our framework does not rely on the experts or predefined rules so that it is suitable to process a large number of source code. To evaluate the ability of our framework, we collect the vulnerability dataset that includes more than 260,000 functions in 118 types of CWE, which is larger than the dataset of existing research. Experiments show that our framework has better performance than existing methods.**

***Index Terms*—Program Language Processing, Vulnerability Detection, Deep Learning,**

I. Introduction

With the increasing complexity of computer programs, the number of security vulnerability reports is also proliferating, which makes vulnerability detection a hot topic in the field of software security. There are 96 million open source projects now published on GitHub[1], and this number is rapidly increasing. The traditional research on security vulnerabilities is based on manual processes or predefined rules. This big data are not fully utilized. Hence, we need automated vulnerability detection methods to maintain the quick response of new vulnerabilities.

Traditional vulnerability detection methods require security researchers to manually analyze program binaries or codes using static or dynamic analysis tools. These methods rely on the experience and ability of the researchers to discover complex vulnerabilities but can do nothing about hundreds of thousands of programs. There are already some code analysis techniques, including some rule-based analysis[1][2], code-clone detection methods[3]. However, all of these existing solutions[3][1][2] rely on manual extraction of vulnerability features by security experts, and are limited to capture features from new data so that they can not take a quick reflection of new vulnerabilities. Recently, there have been some new approaches to automatic vulnerability detection, VUDDY[4] and VulPecker[5], based on code clone and code similarity to detect vulnerabilities from source code. These methods can extract known vulnerability features in advance, and detect code by comparing similarities, but cannot automatically extract vulnerability features to handle new vulnerabilities.

Deep Learning method has been successful in many fields, like computer vision (CV) and natural language processing (NLP). It is suitable to capture sophisticated features, especially for big data. Combining deep learning with vulnerability detection can significantly enhance the ability of automatic vulnerability detection. Two recent studies, VulDeePecker[6] and Cross-Project Learning[7] that applying deep learning into vulnerability detection fields, but both of them have several drawbacks. VulDeePecker treats source code as plain text and directly transform the words into vectors, which will lose the syntax structure of source code. Compared to VulDeePecker, we parsed source code into their abstract syntax tree (AST) to preserve more syntax information and reduce data redundancy. In our tests, the vectors generated from the AST is 70%-90% smaller than the vectors generated from the text (see Section II). VulDeePecker reduces the length of the vectors by truncating and padding the final vectors, which will also cause the information loss. Cross-Project Learning parsed source code into AST, but they truncate and pad the final vectors as VulDeePecker and cannot process variable-length data. On the other hand, they let the model output the probability value and then select the number of samples with the highest probability among all the samples as the vulnerability result. This choice depends on expert experience and cannot apply to a large amount of data.

In this paper, we present the framework of vulnerability detection to overcome the drawbacks we mentioned above.

First, we present a complete data processing method to convert program source code into vector representations, which can preserve all of the syntax features and reduce the data redundancy without any predefined rules or manual process. To achieve this goal, we parse the program source code into AST and slice the function node from it. Then we traverse the entire AST and map all user-defined names to fixed names. In order to learn this data by a deep neural network, we transform AST into node sequence by the preorder traversal. Using this method, we extract the function-level code, remove redundant

[1]https://github.com/

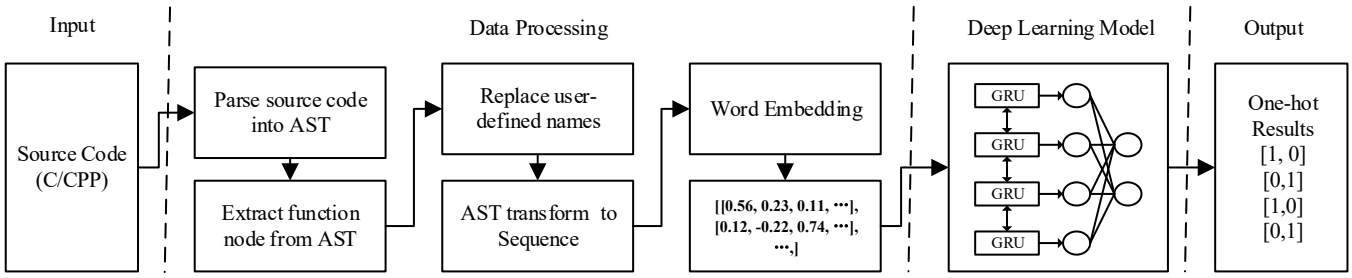

Fig. 1: An overview of the framework. The entire framework takes the program source code as input, then transforms the code into their vector representations, and finally input the vectors into the neural network model for training.

information, and retain complete syntax information. This algorithm is very efficient in handling a large number of programs. We apply the Word2Vec[8] model to map the node sequence into their vector representations that can be directly trained by neural networks.

Second, we construct a neural network with bidirectional gate recurrent unite (Bi-GRU) to achieve efficient vulnerability detection. Our model can learn vulnerability features from the vector representations extracted from the program source code and predict vulnerabilities. By applying the pack-padded method, our model does not need any truncation and padding for the final vectors, which allows our model to handle variable-length data.

Third, we collect the datasets based on Juliet Test Suite built by SARD with more than 260,000 functions to fully evaluate our model, especially the effect of the model in the face of big data. Our experiments show that our model can effectively handle big data. Compared with other vulnerability detection tools and frameworks, our model achieved high precision and low false positive rate, which is more effective than the existing methods.

In summary, our paper makes three key contributions.

1) We propose the data processing method that parses source code into AST and transforms it into vectors to preserve all of the syntax features and reduce the data redundancy without any predefined rules or manual process.
2) We implement a Bi-GRU network to capture features from vectors which generated from program source code to predict vulnerabilities. Besides, we apply the pack-padded method on network training to handle the variable-length data.
3) We collect the dataset based on SARD to fully evaluate our model. Experiments show that our model achieves better performance than the existing methods.

II. Design of Framework

Our goal is to create a comprehensive method that can learn features from known program source code by deep learning model and then predict the given source code, whether it is vulnerable or not. In order to achieve this goal, we focus on two parts: (i) Data processing, (ii) Model design. Figure 1 shows an overview of the entire framework.

A. Data Processing Method

We need a suitable method that can transform program source code into vectors. Source code is the text data with certain syntax rules, so we can not directly input it into the neural network. A straightforward approach treats source code as plain text and divides it into the sequence of tokens, which can be transformed into vectors by using embedding methods[6]. It is widely used in natural language processing, but we think it is not the best way to program language processing. Program source code is rich structured text, include a large number of symbols and control statements. If we only treat the source code as text, we will lose all of the syntax information and produce much redundant information. We think the better way to solve this problem is parsing source code into AST, which is a tree representation of the abstract syntax structure and can generate from code file by using compiler front such as clang[2]. This structure will preserve all of the semantic information and remove useless symbols and comments. The number of tokens extracted from the AST is much smaller than the number of tokens extracted from the text. In our tests, the data generated based on the AST is 70%-90% smaller than the data generated based on the plain code text. Figure 2 shows the basic flow of data processing.

1) Code Parsing: In this paper, we focus on the C/C++ program source code, so we use the Python binding of libclang as the code parser front. We can extract AST from source code file or a fragment of code by using libclang. Then we can walk through the whole tree to slice the useful node we needed. There are several levels we can use for the data processing, the file level, the function level, the statement level, and the token level. The file level represents the whole file, which is the elementary level when we extract AST from source code. This level cannot be directly used for training because it is too complicated to locate the point of vulnerability. The statement level and the token level represent a statement in code or even a word in code. We can directly locate the vulnerability point. However, the data is difficult to mark, and some vulnerabilities involve multiple statements and cannot be located. So the function level is the most suitable level to capture vulnerability patterns. Figure 2 the basic flow of code parsing. As shown in Figure 2 (b), the AST node has several properties, the most

[2]clang.llvm.org/

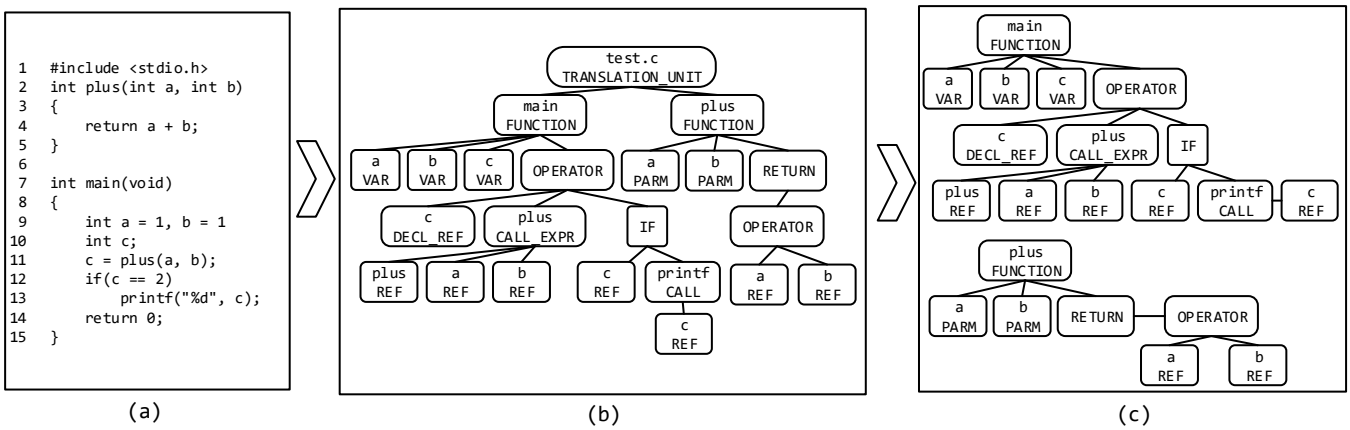

Fig. 2: Parsed source code into AST and extract function node from it. (a) The program source code, (b) The AST parsed from source code (omitted some node), (c) The AST of function node extract from file level AST.

important of which are 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 as 'a' and 'b' in Figure 2 (b). The kind of the node represents the current property of node, for example, the kind of variable 'a' in the declare statement is 'VAR_DECL' (Omitted as VAR in the Figure 2), the kind of variable 'a' in the reference statement is 'VAR_REF_EXPR' (Omitted as REF in the Figure 2). So it is easy for us to find the function node from AST by check their kind. Each function node and their child node represent a whole function, so we can traverse the whole AST and extract all function nodes from it.

2) AST standardization: As we extract the function node from file level AST, we need to map the user-defined variable names and the function names to fixed names, such as 'var0', 'var1', 'fun0', fun1'. In this way, functions of the same structure are mapped to the same patterns, regardless of their original names. There we can use the kind of each node to distinguish whether it is the user-defined names or not, then mapped all of the user-defined names to fixed names. We called this processing method to standardize because this method maps different functions with the same structure to the same function for the next process. This step is significant because we can use a limited element to represent all the functions. User-defined variable names and function names are varied, but the basic elements of the function are similar. We do not care what the specific name of a variable is; we only care about the role of the variable in the function.

3) AST serialization: After obtaining the standardized functions, we need to transform the AST into vectors. We cannot directly input the data of tree structure into the neural network. We must transform an AST into a sequence structure. Some studies are using the random walk to embedding graph into vectors[9][10], which leads us to realize that we can use a certain path to represent a tree into a sequence. There is some straightforward and natural approach called tree traversal algorithms, which can walk through the whole AST and return a certain sequence of nodes. We use the Preorder Traversal as a search algorithm because the AST is generated in the preorder. Hence, if we transform AST into a sequence by Preorder Traversal, these order of sequence is similar to the order of code. We also tested other traversal algorithms, and the experiment showed that Preorder Traversal is better than any other algorithms.

4) Word embedding: Now we need to transform the sequence of AST node into vectors that can input into the neural network. To achieve this goal, we need a method that maps each node to the vector representation. For each node, we focus on two information, the name of the node and the kind of the node. In our method, if a node has its name, we use that name to represent this node, if a node does not have its name, we use its kind to represent this node. In this way, the sequence of the AST node is mapped to a sequence of tokens. Then we can use some word embedding methods from NLP like Word2Vec[8] to map each word to their vector representation. Word2Vec model can learn the relationship between different words from the context and assign a vector to each word based on this relationship. If the relationship between the two words in the text is relatively close, their word vector distance will be relatively close too. We can collect all of the sequences of tokens as a corpus of words to training the Word2Vec model and produce vector space that map every token into suitable vectors. We trained the Word2Vec model in the CBOW algorithm because it takes less training time than the Skip-Gram algorithm. However, using different algorithms does not have much impact on the final result of the model. By applying the Word2Vec model, we can transform the sequence of tokens into the sequence of vectors.

B. Neural Network

Recurrent Neural Network (RNN) has achieved much success in the natural language processing (NLP) task that can learn the impact from the time series and is very suitable for learning features from the long sequence. We parse the code into AST and transform AST into the sequence of tokens, which is similar to the long word sequence. However, there are

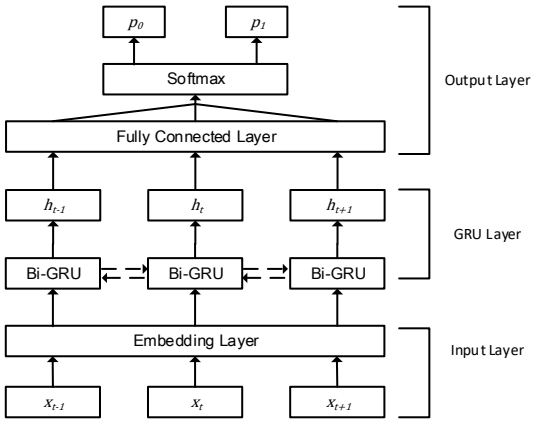

Fig. 3: An overview of our deep learning model

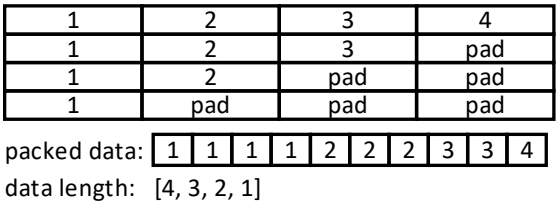

Fig. 4: The process of pack variable-length data into one sequence.

several differences between natural language and programming language that also require special attention. The sequence of the programming language is generally much longer than the natural language. A natural language sentence often includes 20-30 words, but for the programming language, a function often contains 20-2000 tokens. In our statistics, 95% of the functions have no more than 800 tokens, but some functions have more than 2000 tokens. Therefore, a deep neural network created for programming language requires a stronger ability to learn long sequences than a model for natural language. In this paper, we use a variant of RNN called Gated Recurrent Unit(GRU), which has a better effect than the basic RNN in long sequences. Compared to LSTM (Long Short Term Memory Network), the other prevalent RNN variant in long sequences, GRU works faster with the same effect.

Figure 3 shows the basic structure of our neural network. In this paper, we implement the deep neural network with two bidirectional GRU (Bi-GRU) layers and a dense layer with softmax activation. Compared to a single directional GRU, a bidirectional network can learn sequences from two directions, one from start to end and the other from end to start. Our neural network learns vectors extracted from source code and outputs the one-hot labels to predict. The length of the input sequence is the maximum length of the samples in each batch. Because we use pack-padded methods in Pytorch to handle variable-length input, we do not need to truncate the input data like the existing model[6][7]. These extra zero paddings are just for the convenience of processing input data in batches.

As shown in Figure 4, the batch of samples are packed into one sequence data, and the actual length of each sample is also recorded. Then we can input the packed data and the record of the actual length into the neural network model. The model will be trained according to the actual length of each sample. By using this method, we can avoid the impact of truncation and padding on model training, which is very important for correct training RNN models. The effect of data truncation on model training is easy to understand, which will directly lead to data loss. For zero padding, the RNN network records the impact of past sequences, and even a zero vector can have an impact on the final weight of the neural network. Therefore, only by using the pack-padded method can the neural network be trained appropriately.

III. Experiments

a) Environment: In this paper, we implement the Bi-GRU network in Python by using Pytorch and using Gensim to training embedding models. We run our experiments on a server with NVIDIA GeForce RTX 2070 GRU and Intel Xeon E5-2650 v4 CPU.

b) Metrics: Our model extract features from source code and builds a binary classification model to predict whether a given program source code has a vulnerability. Hence, we use some widely used metrics to evaluate our model, *Precision*, *Recall*, *False Positive Rate (FPR)*, *False Negative Rate (FNR)* and *F1 Score (F1)*. The precision (P) $Precision = \frac{TP}{TP+FP}$ represents the percentage of how many predicted vulnerabilities is a vulnerability. The recall (R) $Recall = \frac{TP}{TP+FN}$ calculates the percentage of vulnerabilities correctly predicted total vulnerabilities. The F1 score $F1 = \frac{2*P*R}{P+R}$ considered both precision and recall to take an overview metric. The false positive rate $FPR = \frac{FP}{FP+TN}$ calculate the fall out rate. The false negative rate $FNR = \frac{FN}{TP+FN}$ measures the miss rate.

A. Results

As far as we know, there is no well-designed vulnerability dataset for vulnerability detection. Therefore, we constructed a vulnerability dataset from SARD. The SARD dataset contains 118 types of CWE vulnerability and developed specifically for assessing the capabilities of static analysis tools. Each sample in the dataset are labeled by their function names with 'good' or 'bad' suffix string, so we can easily use this suffix information to generate the label of each sample.

We randomly divide the dataset into three parts, a training set, a validation set, and a test set, the ratio of the partition is 8:1:1this ratio is applied equally to every type of vulnerabilities. The training dataset is only used to training the neural network. The validation dataset is used to adjust the parameters of the model during the training process. After the parameter selection is completed, the neural network is finally tested using the test dataset and record the final results. In this way, we can avoid over-fitting problems caused by parameter selection.

In this paper, we extracted a variety of datasets from the Juliet Test Suite to fully validate the model's capabilities,

especially to prove that our model can fully utilize big data. The specific split of the dataset is shown below:

- BO: The code corresponding to Buffer Overflow, include CWE-121 Stack Based Buffer Overflow and CWE-122 Heap Based Buffer Overflow.
- 5K: The types of CWE that contain more than 5000 samples in the SARD dataset.
- ALL: The complete data of the SARD dataset.

TABLE I: DETAILS OF DATASETS

	All Function	Vulnerable	Non-vulnerable
BO	38,919	15,087	23,832
5K	196,580	64,788	131,792
ALL	269,985	89,071	180,913

Table I shown the details of the different datasets. We first selected two types of vulnerability, stack-based buffer overflow, and heap-based buffer overflow. These types of vulnerability are the largest number of samples in the dataset, and also the most common type of vulnerability. We hope to use this dataset to test the effect of our model on specific vulnerability types. Then we selected the types of CWE that contain more than five thousand samples in the dataset to test the capabilities of training on multiple types of vulnerabilities. Finally, we trained our model on the whole dataset to valid the effort of our model on big data.

We trained each dataset on the Bi-GRU network we described above and carefully adjusted the parameters through experiments. The vector size of the word embedding model is set to 100, so the input size of the network is 100 too. The hidden size of the network is set to 200; the number of layers is set to 4; the dropout is set to 0.5. We tested a variety of gradient descent algorithms, include SGD, Adadelta, Adam and Adamax, and finally used the mini-batch stochastic gradient descent with Adam[11] optimization, it is faster than other algorithms and has the same result. The batch size is set to 50, and the learning rate is set to 0.0001. For BO datasets, the learning rate is set to 0.0005. We used the Cross-Entropy as the loss function. In addition to the parameters we mentioned above, we use the default options for other parameters.

TABLE II: RESULTS OF DATASETS

	P(%)	R(%)	FNR(%)	FPR(%)	F1(%)
BO	93.81	73.52	26.48	2.99	82.43
5K	93.40	74.35	25.65	2.60	82.79
ALL	93.29	74.36	25.64	2.62	82.76

As shown in Table II, the model trained on a large number of samples is powerful than the model trained on a small number of samples. These results can be proved that the quality of the model is high related to the number of data. However, the model trained on all dataset is slightly weak than the model trained on 5K dataset, we infer that this is because there are some CWE types have only a few hundred or even dozens of samples in All dataset. The neural network cannot learn enough features on this small number of samples, so this additional data can not enhance the effect of the model but even caused some confusion to the model. We can conclude that the ability of the model depends not only on the number of datasets but also on data quality.

TABLE III: COMPARED WITH OTHER TOOLS

Dataset	System	P(%)	FNR(%)	FPR(%)	F1(%)
ALL	Rats(w1)	37.43	37.29	46.90	46.88
ALL	FlawFinder(m1)	41.05	56.24	26.91	42.36
ALL	Our model	93.29	25.64	2.62	82.76
BO	Rats(w1)	43.46	21.73	51.73	55.89
BO	FlawFinder(m1)	43.20	50.03	33.9	46.34
BO	VulDeePecker	85.44	28.04	4.41	78.12
BO	Our model	93.81	26.48	2.99	82.43

In order to fully demonstrate the capabilities of our model, we compared with two existing open-source vulnerability detection tools, Rats[1] and Flawfinder[2]. Of course, there is some state-of-the-art methods like VUDDY[4], VulDeePecker[6], and Cross-Project Learning[7]. However, for VUDDY, it can only be used to check for CVE vulnerabilities. For our dataset, it can do nothing about it. For VulDeePecker, it works well on a BO-like dataset, but its authors do not open source their methods and do not provide usable models. We use our reproduce model for comparative experiments. For Cross-Project Learning, the output of their model is probability value, which requires human experts to determine the boundary of the vulnerabilities and is not usable in practice.

We use the ALL and BO dataset for comparison with other methods, and the results are shown in Table III. Our model has advantages in all aspects. It can be considered that our model can more accurately locate the vulnerabilities, and it is easier to distinguish the regular code from the vulnerability code so that the low false positive rate and high precision can be achieved. For RATS and FlawFinder, they can only detect vulnerabilities by calculating the similarity of the code and comparing it with predefined rules, which makes them unable to deal with new data and leads to low F1 score and high FPR and FNR. However, our model does not have a significant advantage in the false negative rate. It can be explained that some vulnerabilities are difficult to identify only by the features extracted from static analysis. For example, buffer overflow is often caused by the wrong variable values. Our model cannot compare the values of different variables in the static analysis, so it is difficult for our model to find such vulnerabilities.

In the BO dataset, VulDeePecker and our model have similar results, both of which have better results than the existing vulnerability detection tools. Our model has better results in all aspects, but there is not much difference between us. It can be explained that the BO dataset is an elementary dataset. Our model and VulDeePecker have reached the limits of the dataset. Because VulDeePecker does not open source their methods and models, we cannot compare with them on a larger dataset, but the current experiments have fully

demonstrated the potential of deep learning models in the field of vulnerability detection.

IV. Related Work

Vulnerability detection has been a hot topic in security research. Early approaches employed software metrics, such as coupling and cohesion metrics[12]. Those approaches require experts to determine the metrics which are used to detect vulnerabilities.

Recent approaches extract text from source code to build vulnerability prediction models[13][14][15]. They used the term-frequencies and Bag-of-Words to build a model for program source code, which produced higher recall than software metric models. Some approaches leveraged a deep learning model to learn features for vulnerability detection automatically, such as clone detection model[16][4], which used RNN models detecting code clones.

Deep learning has been applied to vulnerability detection, VulDeePecker[6], Cross-Project Learning[7], VulSniper[17], μVulDeePecker[18]. These model generally contains a data processing method and a neural network model. The data processing process is often related to the data form. VulDeePecker and μVulDeePecker treat code as plain text, so they process code on text token level. VulSniper parses source code into code property graph and manually select some property encode as vector representation. Cross-Project Learning parses source code into the AST and encodes the syntax node into the vector representation. The neural network is often based on recurrent neural network and it variants, including Bi-LSTM network (Cross-Project Learning, VulDeePecker and μVulDeePecker) and attention network (VulSniper).

V. Conclusion

In this paper, we propose the framework of vulnerability detection to predict the vulnerability from source code. We aim to perform data processing and model learning without expert experience or predefined rules. Besides, we want to completely preserve the features without applying truncation and padding to the final vectors. We parse the source code into AST to preserve the syntax information and remove the redundant information. To overcome the truncation and padding problem, we applying the pack-padded method in the model training process to cope with variable-length data. In order to evaluate the ability of our model, we collect the dataset with more than 260,000 functions in 118 types of CWE vulnerabilities. Experiments show that our model can predict vulnerabilities with high precision and low false positive rate, which is more effective than the existing open-source vulnerability detection tools.

There still some limitations that need further study. First, If the vulnerabilities that involve multiple functions or multiple files, it is hard for our framework to extract features from it. Second, the function level is still rough, some functions have hundreds of lines, and some have only a few lines. How to locate the vulnerability in this different scale of functions is still worth studying. Third, we still lack datasets that can reflect the real code environment, which limits the further development of vulnerability detection.

VI. Acknowledgements

This research was supported by The National Natural Science Foundation of China (No.U1836210,No.61572460).

References

[1] CERN Computer Security Team. The rough auditing tool for security. https://security.web.cern.ch/.

[2] David A. Wheeler. Flawfinder. https://dwheeler.com/flawfinder/.

[3] Jiyong Jang, Abeer Agrawal, and David Brumley. Redebug: finding unpatched code clones in entire os distributions. In *2012 IEEE Symposium on Security and Privacy*, pages 48–62. IEEE, 2012.

[4] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. Vuddy: A scalable approach for vulnerable code clone discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 595–614. IEEE, 2017.

[5] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie 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. ACM, 2016.

[6] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018.

[7] Guanjun Lin, Jun Zhang, Wei Luo, Lei Pan, Yang Xiang, Olivier De Vel, and Paul Montague. Cross-project transfer representation learning for vulnerable function discovery. *IEEE Transactions on Industrial Informatics*, 14(7):3289–3297, 2018.

[8] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[9] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710. ACM, 2014.

[10] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864. ACM, 2016.

[11] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[12] Istehad Chowdhury and Mohammad Zulkernine. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal of Systems Architecture*, 57(3):294–313, 2011.

[13] Riccardo Scandariato, James Walden, Aram Hovsepyan, and Wouter Joosen. Predicting vulnerable software components via text mining. *IEEE Transactions on Software Engineering*, 40(10):993–1006, 2014.

[14] Aram Hovsepyan, Riccardo Scandariato, Wouter Joosen, and James Walden. Software vulnerability prediction using text analysis techniques. In *Proceedings of the 4th international workshop on Security measurements and metrics*, pages 7–10. ACM, 2012.

[15] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. Generalized vulnerability extrapolation using abstract syntax trees. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 359–368. ACM, 2012.

[16] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 87–98. ACM, 2016.

[17] Xu Duan, Jingzheng Wu, Shouling Ji, Zhiqing Rui, Tianyue Luo, Mutian Yang, and Yanjun 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.

[18] Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li, and Hai Jin. μvuldeepecker: A deep learning-based system for multiclass vulnerability detection. *IEEE Transactions on Dependable and Secure Computing*, 2019.