2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W) | 978-1-6654-0262-0/22/$31.00 ©2022 IEEE | DOI: 10.1109/DSN-W54100.2022.00039

VDBWGDL: Vulnerability Detection Based On Weight Graph And Deep Learning

Xin Zhang*§, Hongyu Sun†§, Zhipeng He*§, MianXue Gu‡§, Jingyu Feng*, Yuqing Zhang*†‡§¶
*School of Cyberspace Security, Xi'an University of Posts and Telecommunications, Xi'an, China
†School of Cyber Engineering, Xidian University, Xi'an, China
‡School of Cyberspace Security , Hainan University, HaiKou, China
§National Computer Network Intrusion Protection Center, University of Chinese Academy of Sciences, Beijing, China
¶Corresponding author, zhangyq@nipc.org.cn

***Abstract*—Vulnerability detection has always been an essential part of maintaining information security, and the existing work can significantly improve the performance of vulnerability detection. However, due to the differences in representation forms and deep learning models, various methods still have some limitations. In order to overcome this defect, We propose a vulnerability detection method VDBWGDL, based on weight graphs and deep learning. Firstly, it accurately locates vulnerability-sensitive keywords and generates variant codes that satisfy vulnerability trigger logic and programmer programming style through code variant methods. Then, the control flow graph is sliced for vulnerable code keywords and program critical statements. The code block is converted into a vector containing rich semantic information and input into the weight map through the deep learning model. According to specific rules, different weights are set for each node. Finally, the similarity is obtained through the similarity comparison algorithm, and the suspected vulnerability is output according to different thresholds. VDBWGDL improves the accuracy and F1 value by 3.98% and 4.85% compared with four state-of-the-art models. The experimental results prove the effectiveness of VDBWGDL.**

***Index Terms*—Vulnerability Detection, Code Variant, Weight Graph, Deep Learning**

I. Introduction

The rapid development of computer technology has extensively promoted the development of the economy and society. However, with the ever-increasing number of computer programs, software vulnerabilities are also increasing. According to the CVE[1](Common Vulnerabilities and Exposures) report, the number of software vulnerabilities has shown rapid growth in recent years. The popularity of open-source software makes it easier for software vulnerabilities to spread between computer systems. Once exploited by attackers, it may cause huge losses. Manual vulnerability detection and verification requires a lot of human resources, and researchers need to analyze and understand the target program deeply, and the empirical knowledge is difficult to reuse. Therefore, automatic detection and verification of vulnerabilities have always been a hot spot.

The first problem of vulnerability detection is uneven vulnerability quality. In addition, an excellent deep learning algorithm may need to train millions of samples to get a good model, but in reality, it is difficult for vulnerable samples of the same type to reach this scale. On the other hand, common graph-based structures easily lose source information (such as CFG, control flow graphs, which fail some declarations and statements). At the same time, others are too complicated (such as CPG, code property graph), which increases unnecessary comparison overhead and introduces a lot of useless information. Now deep learning has achieved success in many fields, and it is also suitable for static analysis of programs. Although there have been many examples of deep learning models applied to vulnerability detection, there are still some limitations, such as CNN and RNN can't deal with non-sequential information in source code and can only capture the superficial structure.

To this end, in order to restore the syntax and semantics of the source code as much as possible and improve the detection efficiency, we construct a weight graph model VDBWGDL. Firstly, VDBWGDL automatically collects vulnerability samples, accurately obtains the location of vulnerability function through some methods, and extracts vulnerability characteristic keywords from it. Then we divide vulnerabilities according to sensitive points and get a high-quality vulnerability data set through code variation rules. To deal with these data better, we use static analysis tools to transform them into graph representation structure and weight the graph according to the distance of sensitive words. Then, we transform the extracted code block syntax semantic information into vectors through a deep learning model. Through these methods, we can extract the deep structure information of the function, keep the complete syntax and semantic information, and remove redundant information. F1 and accuracy have reached the best compared with the four state-of-the-art models.

Our contributions are summarized as follows:

- A code variation method is designed according to the vulnerability triggering logic and the programmer's programming style. It generates a lot of relatively valuable vulnerability data.
- The weight graph model. It pays more attention to the feature points of the vulnerability function.
- This model's code block vectorization method can keep the rich syntax and semantic information of functions and simplify the complex graphic information.

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

2325-6664/22/$31.00 ©2022 IEEE
DOI 10.1109/DSN-W54100.2022.00039

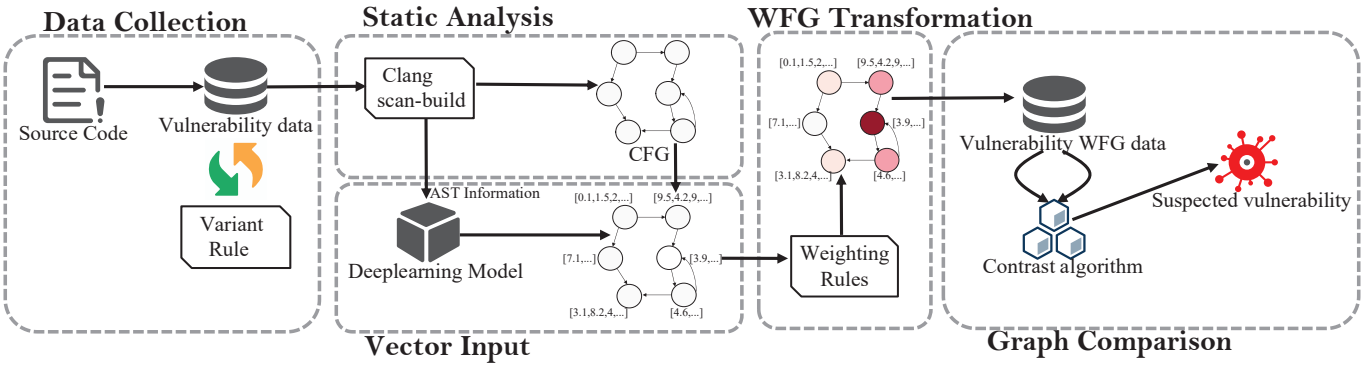

Fig. 1. The overall architecture of the model.$Step1$: Collect data, define variation rules for vulnerability codes, and generate many vulnerability codes and patch codes. $Step2$: Using static analysis tools to convert data into graph structure and extract code syntax and semantic information. $Step3$: Train the deep learning model, encode the source code blocks into vectors and input them into the graph structure. $Step4$: Weighting by weight graph. $Step5$: Comparison of graph similarity.

II. Design

Figure 1 describes the overview of the whole model, which includes five stages: data collection, static analysis, vector input, WFG transformation, and graph comparison.

A. Data Collection

VDBWGDL consists of two modules in the data collection stage, a data collector and a variant code generator.

Data Collector: Crawl CVE sites through the spider. For each CVE web page, we need to get the function name, file name, affected program version, and assigned time. VDBWGDL defines some regular expression rules to extract this information (e.g., extracting words before the "function" field and words ending in ".c"). At the same time, we default the first release version after the vulnerability assigned time to the patched version and extract the patch function from it. Finally, we collected 714 vulnerability data. Table 1 shows the information about the data set.

TABLE I
Vulnerability Data

Program Name	Number of CVEs	Our Vulnerability Data	Proportion
FFmpeg	417	258	31.89%
Wireshark	624	199	61.87%
Openssl	371	145	39.08%
Libtiff	201	112	55.72%

Variant code generator: Obtain the rank of sensitive keywords by using TextRank and TF-DF (term frequency/document frequency) algorithms for vulnerability code. Finally, both of them perform well as vulnerability-sensitive keywords. We tested the function coverage, and when selecting the top20 keyword, the function coverage reached 86.89%. In addition, we classify vulnerability codes according to vulnerability-sensitive keywords and divide vulnerability codes into vulnerability-related statements and vulnerability-irrelated statements. On the other hand, we set the traceable change position as the vulnerability-related statement by comparing the patch function. Finally, code variants are made for the vulnerability-irrelated.

Figure 2 shows the primary process of VDBWGDL in the code variant. Among them, VDBWGDL defines the following rules to meet the vulnerability trigger logic and the programming style of general programmers. Finally, we got 5,355 vulnerability functions.

- Rename
- Modify the parameter expression
- Modify one or more lines
- Reordering of statements
- Synonymous conversion of control statements
- Insert one or more lines
- Delete one or more lines

B. Static Analysis

Static analysis is a crucial module of VDBWGDL. It uses scan-build[2] to analyze the entire program, converts it into a function control flow graph. It works by constructing an imaginary compiler that executes Clang[3] to compile and then runs a static analyzer to analyze the code. Scan-build will generate a pair of entry and exit identifiers and program control flow graphs for each function, and store the AST information of the corresponding code block for each node.

C. Vector Input

When too much information is added to the graph, the information will be too miscellaneous, making the comparison difficult. VDBWGDL has tried TextRNN[12], GRU[13], Bert[14], and other deep learning models to convert the syntax and semantic information of local code blocks into vectors and input them into the weight graph. We use the static analysis tool to analyze the simplified vulnerability function and patch function and save the syntax and semantic information of the code block as much as possible through the hidden layer vector

[2]https://clang-analyzer.llvm.org/scan-build.html
[3]https://clang.llvm.org/

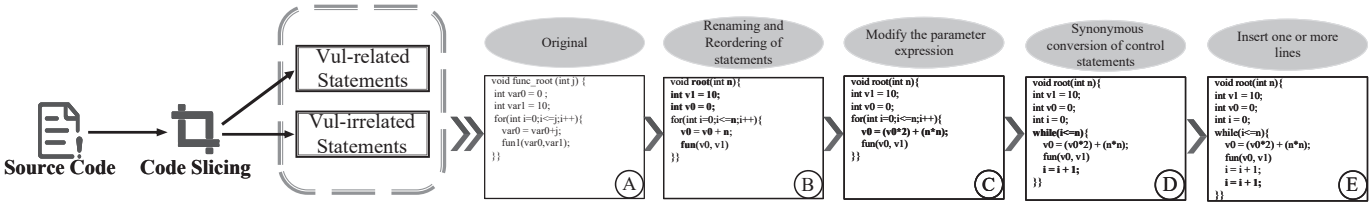

Fig. 2. Code Variant Process

of this model. Experiments show that this method can save the code syntax and semantic information as much as possible and simplify the amount of data. In the comparative experiment in the next section, we find that the GRU model performs best.

D. WFG Transformation

From the perspective of vulnerabilities, the contribution of code to vulnerabilities is different. For example, an inconsistent bits-per-sample value may trigger an assertion violation, leading to a denial of service. Before starting assertion conflict, it will go through multiple operations (CVE-2018-12459[4]). Therefore, this paper adopts the weight graph model to pay more attention to highly fragile code. Fig.3 describes the weight allocation process of VDBWGDL. By default, the code closer to sensitive keywords (control is more closely related to data dependence) contributes more to the vulnerability and is set as the root line. If the selected two roots are on different nodes, both root nodes need to generate corresponding subgraphs. If the two roots are on the same node, choose the one with the highest sensitivity ranking as the root row.

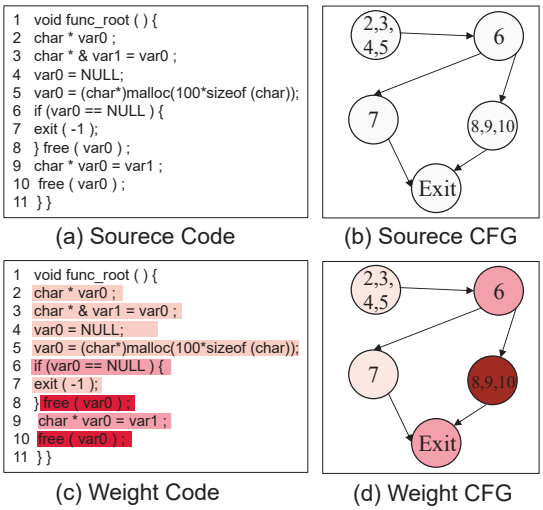

Fig. 3. Weight Graph Generation Process. (a) source code (b) Convert source code to CFG (c) Code for distributing line weights (d) CFG after node weight distribution

At the same time, VDBWGDL defines two weight distribution principles.

Row weight: Calculated according to the distance from the root row, starting from the root row, and assigning an attenuation value (A) to each layer. When the number of rows exceeds 25, it has little effect, and ignore them. The first layer is set to LW_0, then the row weight of the k layer is formed to

$$LW_k = LW_0 * A_{k-1} \quad (1)$$

Node weight: The node where the root row is located is defined as the root node and decays once after each layer. If the node contains multiple lines of code, the calculation with the largest row weight in the node is used. The first layer is set to NW_0, then the row weight of the k layer is formed to

$$NW_k = NW_0 * MAX(LW_j), j = \{1, 2, 3, ..., k-1\} \quad (2)$$

Finally, we set LW_0, NW_0, A to 1, 1, 0.9 by default.

E. Graph Comparison

Algorithm 1 shows the main flow of the graph similarity algorithm used in this paper. Graph similarity is an NP problem, and it isn't easy to find an optimal solution. Therefore we compute the similarity between two WFG graphs by bipartite graph matching. At the same time, because the patch code P is highly similar to the vulnerability code V, VDBWGDL will compare the similarity between the test code T with the patch code and the vulnerability code, respectively. Only when $Sim(T, V) > Sim(T, P)$ will the test code be considered as a suspected vulnerability code, which can reduce false positives.

III. Evaluation

A. Experimental Device

The hardware environment used in this article is Intel Xeon E5-2650 v4 CPU, NVIDIA GeForce RTX3090, 64GB DDR4 RAM.

The software environment used in this article: Ubuntu 18.04.5, Python 3.7.2, clang 12.0.1, LLVM 12.0.1.

The trained hyper-parameters in this article are: hidden size is 64 (Bert is 768, and the final result is reduced to 64 using PCA); batch size is 64; learning rate is 0.0002; dropout is 0.2.

B. Evaluation Indicators

We calculated the following performance metrics: Accuracy (A), Precision (P), Recall (R), and F1 metric.

Accuracy represents the ability of the model to classify correctly. Precision represents the proportion of true positive samples among all tagged positive samples. The recall represents the proportion of true positive samples among all vulnerable samples. F1-score is an indicator that considers both precision and recall.

[4]https://www.cve.org/CVERecord?id=CVE-2018-12459

Algorithm 1: Figure similarity comparison

Input: WFG : Graph converted to WFG
Output: Sim: Similarity score
max ← Maximum number of nodes in two graphs
min ← Minimum number of nodes in two graphs
if *max > 3*min* **then**
  return 0
end
if *max > 2*min* **then**
  $nodeDist = 10000$
else
  for *each node* $N_i \in wfg_1$ **do**
    for *each node* $N_j \in wfg_2$ **do**
      /*calculate the two nodes Euclidean
       Distance and Cosine Value*/
      cost_matrix ← Cost($Ni.vector, Nj.vector$)
    end
  end
end
nodeDist ← Hungarian(cost_matrix)
if *max > 2*min* **then**
  $edgeDist = 100$
else
  for *each node* $N_i \in wfg_1$ **do**
    for *each node* $N_j \in wfg_2$ **do**
      cost_matrix ← Cost($Ni.vector, Nj.vector$)
    end
  end
end
$edgeDist$ ← Hungarian(cost_matrix)
Sim ← Weight$_{sim}$($nodeDist, edgDist$)
return Sim

C. Evaluate and compare methods

To fully demonstrate the function of our model, we compare VDBWGL with four state-of-the-art models, including code similarity methods based on text, sequence, tree, and graph.

- AVDSCUDPL[15], a text-based method, the author uses three open-source static analysis tools (Clang, Flawfinder, Cppcheck) to generate labels for data. Each static analyzer detects from different angles based on CNN and RNN training.
- VulDeePecker[16] is a sequence-based method that uses function call sequence as essential feature information to build a multi-classification neural network model.
- EVDBAST[7] is a method based on an abstract syntax tree characterized by AST and adopts the GRU model.
- CBCD[17], a variant of the graph-based method, directly compares two PDGs.

Figure 4 shows the performance of each method. Compared with other models, our model has obvious advantages in all aspects. The AVDSCUDPL method directly inputs the source code as text, introducing a lot of useless information and ignoring a lot of grammatical and semantic information in model training. Hence, the accuracy is relatively low,

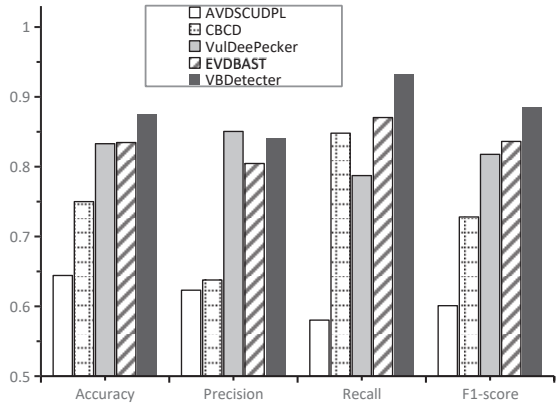

Fig. 4. Performance comparison of different methods

only 64.42%. Vuldeepacker's method describes syntax through sequence representation. Still, its model will truncate the final vector, which will lead to some information loss. It doesn't add an attention mechanism, so Vuldeepacker's feature extraction is rough, with low accuracy and a high false-positive rate. By formalizing the source code into a tree structure and training through bidirectional GRU, the performance of EVDBOAST will be better than VulDeePecker. However, it takes a long time to generate AST, and it is prone to the explosion of AST semantic information. Hence, it is difficult to apply to large-scale software systems with certain limitations. CBCD is superior to the AVDSCUDPL method in performance because it uses graph structure to represent relatively complete source code syntax and semantic information. However, the PDG information generated by it is too complex, and the comparison is the whole graph, so the overall performance is ordinary.

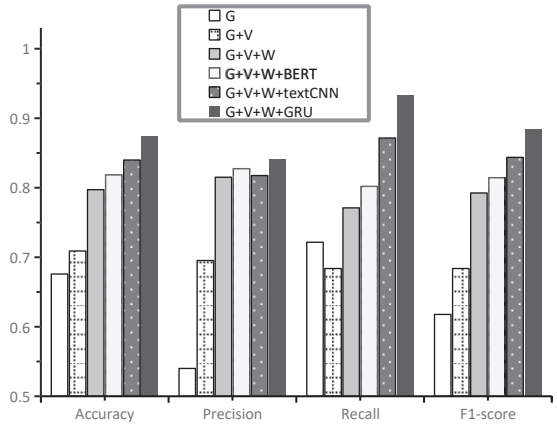

Fig. 5. Performance comparison under different efforts

Figure 5 shows the performance of VDBWGDL under different efforts, namely graph structure(G), code variation(V), graph weighting(W), TextCNN, GRU, and BERT. In addition, the combination is represented by +(e.g., G+W represents graph structure+graph weighting). After using graph weighting, the F1 value increased by 10.86%, after using code variants, the F1 value increased by 6.60%, and after using the

GRU model, the F1 value increased by 9.21%. Graph weighting made the model pay more attention to the fragile code. Code variants and the GRU model made the model training more robust and kept the syntax and semantic information as much as possible. In the choice of deep learning model, we tried to use TextCNN, GRU, and BERT. However, the BERT model doesn't perform well because we don't have enough data to pre-train a complete BERT model. As an alternative, CodeBERT[18] is used to train our data.

TABLE II
TIME EXPENDITURE

	Static Analysis	CFG Cnt	WFG Conversion	WFG Cnt
Openssl	35min	14497	4min13s	1236
FFmpeg	2h40min	127153	22min9s	10745

To demonstrate the efficiency of VDBWGDL in the field of vulnerability detection, we summarized the time overhead on two data sets, as shown in Table 2. VDBWGDL mainly spends time on three aspects: static analysis, WFG conversion, and WFG comparison. The cost primarily focuses on static analysis because this step needs to use scan-build to parse the whole program. The cost of WFG transformation is relatively small, and its main cost focuses on vector transformation and weight distribution. The first two steps only need to be operated once. The graph comparison speed is breakneck. It only took 72 seconds to complete the comparison of 1,528 WFGs with the vulnerable WFGs, which undoubtedly proves the potential of VDBWGDL in processing a large amount of data.

Based on the above experiments, the performance improvement of VDBWGDL is mainly due to three factors: Firstly, the weight graph model makes VDBWGDL pay more attention to the sensitive point and surrounding fragments. At the same time, the non-sensitive code segments have little influence. Secondly, the hidden layer vector of the deep learning model can relatively completely describe the function's rich syntax and semantic information. At the same time, it simplifies the complicated graph information and reduces the cost of the model. Thirdly, the code variation generates a large number of valuable vulnerability data to meet the training needs of the model and make the trained model more robust.

IV. CONCLUSION

This paper proposes a vulnerability detection model VDBWGDL, which is used to detect vulnerabilities by comparing similarities in a graph structure. In order to restore the syntax and semantics of the code as much as possible and improve the detection efficiency, VDBWGDL constructs a weight graph model to describe the code information with different sensitivities and inputs the rich syntax and semantics information of the code block into the weight graph through a deep learning model. Experimental results show that our model has better performance than the state-of-the-art methods.

ACKNOWLEDGEMENT

This work was supported by the Key Research and Development Science and Technology of Hainan Province (ZDYF202012, GHYF2022010), and the National Natural Science Foundation of China (U1836210).

REFERENCES

[1] Li, Z., Zou, D., Xu, S., Jin, H., Qi, H., Hu, J. (2016, December). Vulpecker: an automated vulnerability detection system based on code similarity analysis. In Proceedings of the 32nd Annual Conference on Computer Security Applications (pp. 201-213).

[2] Grieco, G., Grinblat, G. L., Uzal, L., Rawat, S., Feist, J., Mounier, L. (2016, March). Toward large-scale vulnerability discovery using machine learning. In Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy (pp. 85-96).

[3] Choi, M. J., Jeong, S., Oh, H., Choo, J. (2017). End-to-end prediction of buffer overruns from raw source code via neural memory networks. arXiv preprint arXiv:1703.02458.

[4] Roy, C. K., Cordy, J. R. (2008, June). NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In 2008 16th iEEE international conference on program comprehension (pp. 172-181). IEEE.

[5] Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., ... Zhong, Y. (2018). Vuldeepecker: A deep learning-based system for vulnerability detection. arXiv preprint arXiv:1801.01681.

[6] Sajnani, Hitesh, et al. "Sourcerercc: Scaling code clone detection to big-code." Proceedings of the 38th International Conference on Software Engineering. 2016.

[7] Feng, H., Fu, X., Sun, H., Wang, H., Zhang, Y. (2020, July). Efficient Vulnerability Detection based on abstract syntax tree and Deep Learning. In IEEE INFOCOM 2020-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS) (pp. 722-727). IEEE.

[8] Yamaguchi, F., Lottmann, M., Rieck, K. (2012, December). Generalized vulnerability extrapolation using abstract syntax trees. In Proceedings of the 28th Annual Computer Security Applications Conference (pp. 359-368).

[9] Zeng, Jingxiang, et al. "An Efficient Vulnerability Extrapolation Using Similarity of Graph Kernel of PDGs." 2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom). IEEE, 2020.

[10] Cui, L., Hao, Z., Jiao, Y., Fei, H., Yun, X. (2020). VulDetector: Detecting Vulnerabilities Using Weighted Feature Graph Comparison. IEEE Transactions on Information Forensics and Security, 16, 2004-2017.

[11] Li, X., Wang, L., Xin, Y., Yang, Y., Chen, Y. (2020). Automated vulnerability detection in source code using minimum intermediate representation learning. Applied Sciences, 10(5), 1692.

[12] Zhang, Y., Wallace, B. (2015). A sensitivity analysis of (and practitioners' guide to) convolutional neural networks for sentence classification. arXiv preprint arXiv:1510.03820.

[13] Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., Bengio, Y. (2014). Learning phrase representations using RNN encoder-decoder for statistical machine translation. arXiv preprint arXiv:1406.1078.

[14] Devlin, J., Chang, M. W., Lee, K., Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805.

[15] Russell, Rebecca, et al. "Automated vulnerability detection in source code using deep representation learning." 2018 17th IEEE international conference on machine learning and applications (ICMLA). IEEE, 2018.

[16] Zou, D., Wang, S., Xu, S., Li, Z., Jin, H. (2019). μ VulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection. IEEE Transactions on Dependable and Secure Computing, 18(5), 2224-2236.

[17] Li, J., Ernst, M. D. (2012, June). CBCD: Cloned buggy code detector. In 2012 34th International Conference on Software Engineering (ICSE) (pp. 310-320). IEEE.

[18] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., ... Zhou, M. (2020). Codebert: A pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155.