Fine-Grained Source Code Vulnerability Detection via Graph Neural Networks

1st Jingjing Wang
National Key Laboratory of Science and Technology on Information System Security
Beijing, China
wang_1094412598@163.com

2nd Minhuan Huang†
National Key Laboratory of Science and Technology on Information System Security
Beijing, China
darbean@126.com

3rd Yuanpin Nie
National Key Laboratory of Science and Technology on Information System Security
Beijing, China
yuanpingnie@nudt.edu.cn

4th Xiaohui Kuang
National Key Laboratory of Science and Technology on Information System Security
Beijing, China
xiaohui_kuang@163.com

5th Xiang Li
National Key Laboratory of Science and Technology on Information System Security
Beijing, China
ideal_work@163.com

6th Wenjing Zhong
Xidian University
Xi'an, China
zhongwj@stu.xidian.edu.cn

***Abstract*—Although the number of exploitable vulnerabilities in software continues to increase, the speed of bug fixes and software updates have not increased accordingly. It is therefore crucial to analyze the source code and identify vulnerabilities in the early phase of software development. However, vulnerability location in most of the current machine learning-based methods tends to concentrate at the function level. It undoubtedly imposes a burden on further manual code audits when faced with large-scale source code projects. In this paper, a fine-grained source code vulnerability detection model based on Graph Neural Networks (GNNs) is proposed with the aim of locating vulnerabilities at the function level and line level. Our empirical evaluation on different C/C++ datasets demonstrated that our proposed model outperforms the state-of-the-art methods and achieves significant improvements even when faced with more complex, real-project source code.**

***Index Terms*—deep learning, program analysis, vulnerability detection**

I. Introduction

According to the report released by the National Institute of Standards and Technology (NIST) [1], the number of vulnerabilities found in 2022 is contributing to a sharp rise. The record number of vulnerabilities found over five consecutive years, along with the fact that bug fixes and software updates have not kept pace mean that we are now facing higher security risks than ever before. Consequently, in order to improve system security and code audit efficiency, as well as to further standardize programmers' coding behavior, it is crucial to identify potential vulnerabilities in the programs and fix these in a timely fashion through source code analysis in the early stage of software development.

†Corresponding author
DOI reference number: 10.18293/SEKE2023-115

The accuracy of conventional machine learning (ML) algorithms (i.e. Support Vector Machine, Decision Tree, Random Forest, etc.) for static source code analysis heavily depends on domain experts to perform feature engineering. However, this process becomes onerous and impractical as software source code scales up and functions become more complicated [2], [3]. Deep learning (DL) technology can overcome the drawbacks of conventional ML and automatically extract features from objects, provided that heuristic guidance strategies have been established. Nevertheless, source code is a structured language and deep neural networks often treat it as natural language in the feature extraction stage [4]–[7]. This results in the loss of program logic and structure information, limiting the DL model's potential for vulnerability feature learning. In recent years, GNNs [8] have offered new insights into the static vulnerability analysis of source code by using intermediate representations such as abstract syntax trees (AST), control flow graphs (CFG), and data flow graphs (DFG) [9]–[12].

The accuracy of conventional machine learning (ML) algorithms (i.e. Support Vector Machine, Decision Tree, Random Forest, etc.) for static source code analysis is highly dependent on domain experts to perform feature engineering. However, this process becomes increasingly impractical as software source code scales up and becomes more complex [2], [3]. Deep learning (DL) technology can automatically extract features from objects, but treating source code as natural language in the feature extraction stage [4]–[7] can result in the loss of program logic and structure information, thereby limiting the potential for vulnerability feature learning. GNNs using intermediate representations, such as AST, CFG, and DFG, have provided new insights into the static vulnerability analysis of source code in recent years

On the other hand, many ML-based vulnerability detection

models are trained on datasets that contain synthetic samples to some extent. Nonetheless, due to dataset labeling and model granularity limitations, vulnerability location is commonly concentrated at the function level. In contrast, some studies have focused on more precise locations at the slice [13], [14] or line level [15]. However, these refined locations require strict data labeling requirements, as well as a laborious data preprocessing process.

In order to address the aforementioned issue, we propose a new vulnerability detection model based on GNNs that can accurately locate vulnerabilities at both the function and line level. The contributions of our work can be summarized as follows:

- We propose a novel GNN-based approach, which learns source code information through the intermediate representation of multidimensional program features. It is developed to improve the performance of both function-level and line-level vulnerability location and achieve efficient code auditing without the need for heavy manual engineering.
- We propose a vulnerability dataset with function-level and line-level labels, which were collected from popular open-source C/C++ projects, to further evaluate the effectiveness of our method. Compared with the existing public vulnerability datasets, our dataset is relatively more complete and valuable for further research.

II. Related Works

As the earliest DL-based vulnerability detection systems, Vuldeepecker [14] and Sysevr [13] utilize Bi-directional Long Short-Term Memory (BiLSTM) to apply fine-grained program representation in order to locate vulnerabilities at the slice level. Follow up studies included μVuldeepecker [16] and VulDeeLocator [17]. In addition, many studies extract code semantics based on AST and adopt vulnerability detection models in combination with BiLSTM [5], [18]–[20], which attempt to achieve high classification precision at the function level. Furthemore, the CPG was first proposed by [21], providing a new insight into source code vulnerability feature extraction. Some studies have realized function-level source code vulnerability identification based on GNNs [9], [10], [12]. These studies prove that these methods can effectively capture the program structure and node information carried by the CPG and its variants [11], [12], [22]. This compensates for the loss of important code logic and structural information in other deep learning models due to their use of a serialized feature learning process.

III. Method

Objective The goal of our vulnerability detection model is to predict the label $y_i \in Y = \{0,1\}^m$ of the CPG $G_i \in \mathcal{G}$ corresponding to a given source code function $C_i \in C$ with a mapping function $f : \mathcal{G} \to Y$. Here, C represents the set of source code function, while m is the total number of function instances; moreover, a vulnerable function is labeled with 1, and otherwise 0. To this end, our model is designed to learn an entire CPG representation h_g through a set of node representations $\{H_v|v \in V\}$ obtained by a feature encoder that is used to decide a label $f(G) = \hat{y}$; here v refers to the node feature vector, and $\hat{y}$ is the prediction result. The mapping function f is then learned with a cross-entropy loss by minimizing the negative loglikelihood below:

$$\min \sum_{i=1} -y_i \log \hat{y}_i. \tag{1}$$

The architecture of our model, illustrated in Figure 1, comprises the following three modules: 1) *Embedding module*. The Code Property Graphs (CPGs) are adopted as the intermediate representation of the source code. A multidimensional program feature encoding scheme is then designed to convert the CPGs into vectors, which forms the input of the model. 2) *Location module*. A novel location module is designed to capture important nodes of CPGs according to their IS value and return the corresponding potential vulnerable lines of code. 3) *Classification module*. BiLSTM is introduced as a readout function to generate the global representation of CPGs for identifying vulnerable functions.

A. Embedding Module

1) Code Property Graph Generation: Compared with using a single property, CPGs have been shown to be able to model more common vulnerability types [24], enabling it to achieve efficient vulnerability mining. As for the implementation, Joern [21] is adopted to generate a joint data structure composed of code properties for source code.

2) Graph Embedding of Multidimensional Program Features: The node information of CPG consists of two parts: attributes and code. To encode the source code from various perspectives, such as function calls, logical operations, variable types, semantics, and syntax, a node compound feature embedding method has been devised in this stage. Figure 2 delineates the process of node feature embedding.

a) Node Attribute Embedding: The node attribute feature consists of vectors of five fields. According to the *tag*, all the nodes are divided into different categories, representing the different roles played by nodes in the CPG. For example, the V_{op} field contains the encoding for predefined program operations, such as assignment, judgment, comparison, and so on. Similarly, the V_{func} field reflects the call relationship between the program and specific functions. Moreover, V_{lite} describes the variables involved in the operation of the program, such as characters and numbers, while V_{type} corresponds to 16 fixed parameter types in the C/C++ language. All the vectors of $V_{attribute}$ are encoded via one-hot before being concatenated.

b) Node Code Embedding: The semantic information of each node in the CPG is encoded through vectorization of the corresponding code statements. As for implementation, after cleaning the comments and removing non-ASCII characters, the code is normalized to alleviate the burden of feature encoding caused by the presence of numerous user-defined functions and variables independent of vulnerabilities. Finally, the tokenized code sequences are mapped to feature vectors based on the pre-trained Word2Vec model to obtain the fixed-size V_{code} and concatenate it with $V_{attribute}$.

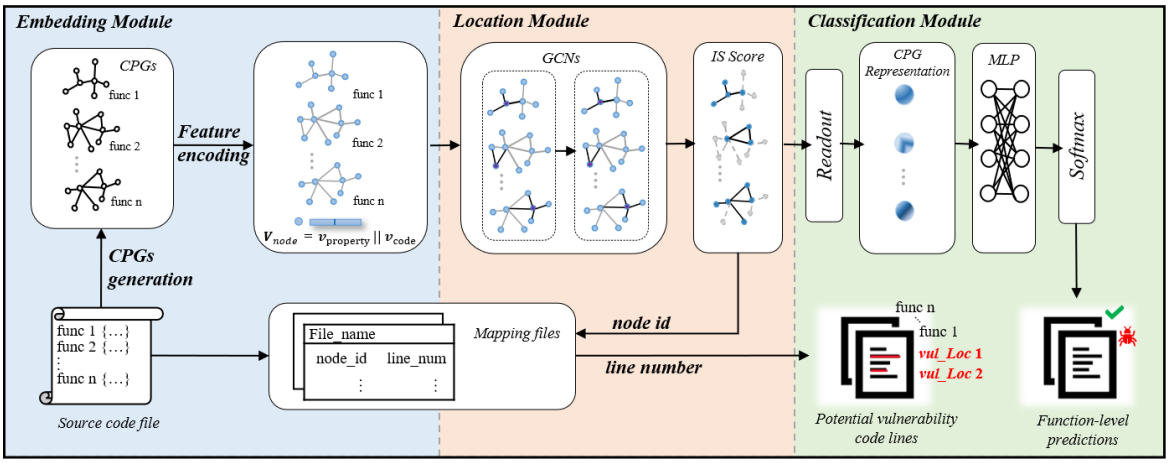

Fig. 1. The architecture of our model.

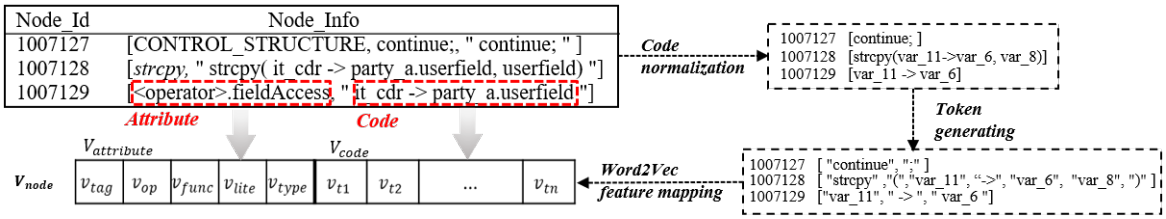

Fig. 2. The node feature embedding process.

B. Location Module

1) GCN Layers: To aggregate the neighborhood information, we use graph convolutional network (GCN), first proposed by [25].

For a CPG G_i with n nodes and d_v dimensional features, the definition of GCN is as follows:

$$H^{(l+1)} = \sigma(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}). \tag{2}$$

Here, $H^{(l)}$ is the node representation of the l-th layer and is initialized by the node features matrix $X \in \mathbb{R}^{n\times d_v}$, $\tilde{A} \in \mathbb{R}^{n\times n}$ is the adjacency matrix with self-connections, $\tilde{A} = A + I_N$, $\tilde{D} \in \mathbb{R}^{n\times n}$ is the degree matrix of $\tilde{A}$, $W \in \mathbb{R}^{d_{in}\times d_{out}}$ is the weight matrix with input feature dimension d_{in} and output feature dimension d_{out}, and $\sigma(\cdot)$ refers to the ReLU function [26], which is used as the activation function.

2) Line-level Location: To ensure that more attention is paid to the important nodes with high influence on vulnerabilities, the node score $Z \in \mathbb{R}^{n\times 1}$ is obtained by two-layer GCN learning, as follows:

$$Z(H, A) = \tanh(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}). \tag{3}$$

In the real world, differences between vulnerable and benign code may be subtle, but it is related to many nodes reflected in CPG, as shown in Figure 3. To make the node features after message-passing more distinguishable and attempt to capture more detailed vulnerability feature patterns, a learnable parameter matrix $\theta_l \in \mathbb{R}^{n\times 1}$ is introduced. Finally, the *Influence Score* : $IS \in \mathbb{R}^{n\times 1}$ of CPG nodes can be expressed, as follows:

$$IS(H, A) = LN(Z + \theta_l), \tag{4}$$

where LN is a layer normalization [29].

On this basis, the $\lceil kn \rceil$ CPG nodes with the highest IS would be retained; here, k is the keep ratio. Subsequently, according to the graph mapping files (generated by Joern), the node indexes are mapped to the relevant lines of code in the source code file to locate the vulnerability on the line level. The locating process can be described as follows:

$$idx = top\_rank(IS, \lceil kn \rceil), \tag{5}$$

$$\hat{H} = H_{idx}, \tag{6}$$

$$Loc = map(idx),\ map : idx \rightarrow line\_num \tag{7}$$

where top_rank returns the indices of the retained nodes, $\hat{H} \in \mathbb{R}^{kn\times 1}$ is the new feature matrix used as the input of the next layer, and Loc represents the set of line numbers of code mapped by idx.

C. Classification Module

It is worth noting that there is often a strong correlation between multiple lines of code that contribute to a specific vulnerability, which in turn correspond to the key nodes in the CPG. However, some commonly used approaches to graph pooling [31], [32] ignore the interaction between nodes, or cause the loss of node information [33], [34]. For this purpose, when CPG is summarized as $\lceil kn \rceil$ important nodes, BiLSTM is introduced as a readout function that further considers the dependencies and inter-node relationships among these nodes to learn a d_v-dimensional meaningful graph representation $r_i \in \mathbb{R}^{d_v}$, as follows:

$$r_i = BiLSTM(\hat{H}). \tag{8}$$

Finally, the function-level prediction $\hat{y}_i$ is achieved through the two fully connected layers with softmax outputs, as follows:

$$\hat{y}_i = \text{Softmax}(W_F^{(2)}(W_F^{(1)}r_i + b^{(1)}) + b^{(2)}), \tag{9}$$

where $W_F^{(\cdot)}$ and $b^{(\cdot)}$ are parameters of the layer.

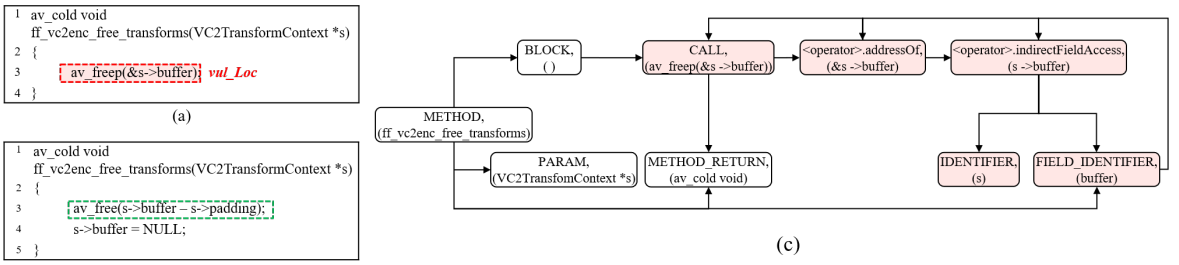

Fig. 3. (a): Example of a vulnerable function with an Out-of-bounds Read Error (CWE-125) from Ffmpeg (unpatched); (b): The fixed vulnerable function (patched); (c): The simplified CPG of the vulnerable function in (a). The red nodes in (c) corresponding to the red line of code labeled as *vul_Loc* in (a).

TABLE I
COMPARISON OF FUNCTION-LEVEL CLASSIFICATION OF KNOWN CWE TYPES ON HD AND RD DATASETS. P: PRECISION(%); F1: F1-SCORE(%)

Method		HD				RD							
		CWE-119		CWE-399		CWE-119		CWE-399		CWE-668		CWE-264	
		P	F1	P	F1	P	F1	P	F1	P	F1	P	F1
ML-based Models	XGBoost	88.7	86.6	86.9	86.8	29.1	29.5	30.6	28.1	31.2	31.0	35.1	33.0
	CNN	89.3	85.1	90.8	93.1	57.0	62.7	45.7	56.2	52.3	50.3	57.9	63.8
	Vuldeepecker [14]	91.7	86.6	94.6	95.0	54.1	61.4	46.6	63.6	49.6	66.3	49.8	66.5
	Devign [12]	88.6	87.9	91.3	89.9	80.0	72.7	75.0	66.7	50.0	60.0	55.6	66.7
Commercial Tools	Cppcheck [35]	52.8	52.6	70.9	19.2	49.7	28.1	50.0	17.8	50.2	22.8	50.1	14.6
	Flawfinder [36]	25.0	27.7	34.1	37.4	49.9	61.5	50.4	57.6	50.3	59.5	50.3	56.9
	RATS [37]	19.4	20.2	35.0	35.6	50.4	51.2	49.9	44.7	5.2	46.8	50.1	39.4
	Flint++ [38]	58.8	60.7	65.4	67.1	50.5	65.2	50.3	65.6	50.2	64.8	50.1	61.2
Ours		**98.1**	**97.4**	**98.8**	**99.0**	**81.3**	**85.0**	**87.5**	**87.4**	**88.7**	**88.6**	**78.9**	**80.6**

TABLE II
COMPARISON OF FUNCTION-LEVEL CLASSIFICATION OF REAL-WORLD PROJECTS WITH UNKNOWN CWE TYPES ON RD DATASETS. P: PRECISION(%); F1: F1-SCORE(%)

Method		RD											
		FFmpeg		Linux Kernel		Openssl		Qemu		Wireshark		Xen	
		P	F1	P	F1	P	F1	P	F1	P	F1	P	F1
ML-based Models	XGBoost	27.8	30.3	33.2	33.1	32.1	32.9	40.4	35.4	26.0	25.8	42.0	39.8
	CNN	50.7	65.1	50.8	32.5	52.9	61.7	57.1	32.0	50.0	57.1	72.7	8.1
	Vuldeepecker [14]	50.7	66.4	44.8	41.2	57.0	66.2	56.7	36.6	46.3	57.1	55.6	9.8
	Devign [12]	75.0	54.5	50.0	60.0	50.0	61.5	50.0	55.6	53.8	63.6	50.0	66.7
Commercial Tools	Cppcheck [35]	50.0	18.0	50.0	13.9	49.8	4.8	47.0	22.8	49.9	19.9	50.1	17.8
	Flawfinder [36]	49.8	60.3	50.3	59.2	50.0	59.8	51.7	64.9	50.3	53.1	49.9	55.6
	RATS [37]	50.0	34.1	50.0	32.8	49.6	62.4	49.0	50.8	49.5	36.8	50.0	31.4
	Flint++ [38]	49.8	61.4	50.3	64.7	50.0	66.3	53.9	67.8	50.0	66.8	49.9	65.1
Ours		**82.1**	**85.2**	**86.7**	**85.5**	**67.3**	**67.1**	**74.2**	**72.3**	**70.7**	**77.4**	**87.6**	**83.0**

IV. EXPERIMENTS

In this section, we conduct extensive experiments on two datasets to evaluate the effectiveness of the proposed model in performing fine-grained vulnerability location and compare it with that of state-of-the-art vulnerability detection methods.

A. Datasets

The experiments are carried out on two datasets: the Hybrid Dataset (HD) and the Real-project Dataset (RD).

1) Hybrid Dataset (HD): In order to verify the impact of different levels of dataset complexity on the performance of vulnerability detection methods, along with the gap between these methods and practice, the dataset proposed by VuleDeepecker [14] is utilized in the experiments, which includes two known vulnerabilities: Memory Buffer Errors (CWE-119) and Resource Management Errors(CWE-399).

2) Real-project Dataset (RD): The statistics presented in [10] indicate that samples from real projects constitute a minor proportion of the dataset HD. To ensure that our proposed model effectively detects vulnerabilities in real software applications and makes meaningful contributions to production-level code security audits, we collected a completely real project dataset, RD.

RD contains 17752 programs with function-level and partial line-level data labeling for 13 popular C/C++ libraries and is available at https://github.com/fgVDgnn/fgVDgnn. Functions and lines of code corresponding to the security commits from NVD are labeled according to the version changes before and after the patch. As shown in Figure 3, compared with patched function (3(b)), we label patch-related statements in vulnerability functions (3(a)) as a vulnerable line of code (vul_Loc).

B. Baselines

We select two categories of methods in the field of static source code vulnerability analysis for performance comparison, as follows. 1) *ML-based vulnerability detection models*: XGBoost, CNN, Vuldeepecker [14] and Devign [12]. We conduct experiments on the reproducible version of these methods to evaluate the performance of our model compared

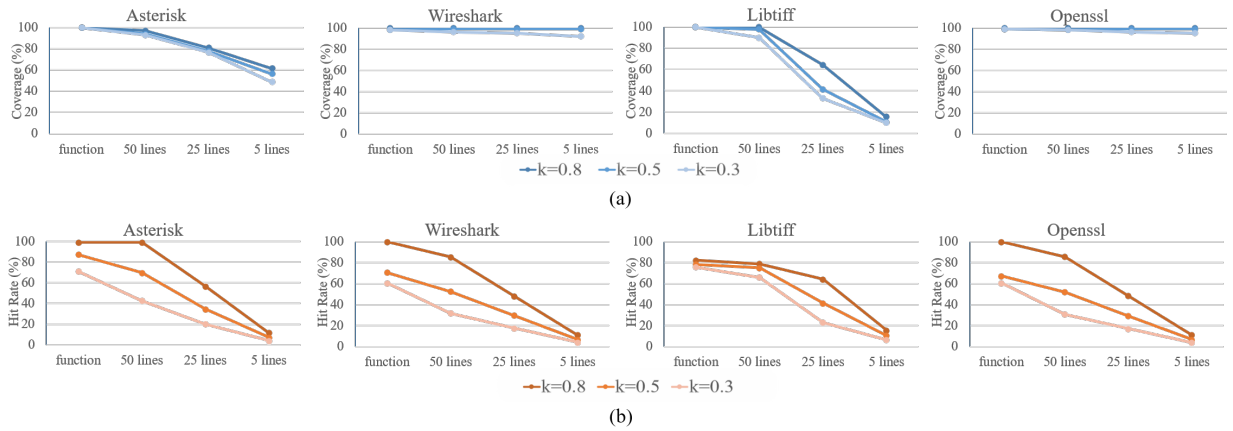

Fig. 4. Line-level location results on four projects. (a): The model's coverage of vulnerability lines under different k value; (b): The model's hit rate of vulnerability lines under different k value.

with typical ML-based methods. 2) *Commercial code analysis tools*: Cppcheck [35], Flawfinder [36], RATS [37] and Flint++ [38]. They are popular commercial tools for scanning code and reporting potential security vulnerabilities, which can be used as a simple guide to static source code analysis.

C. Results

1) Results on function-level classification: For the function-level classification task, we conduct experiments on HD and RD respectively to explore the effect of the vulnerability detection methods when facing source code with known types of vulnerability, along with the impact of different levels of data complexity on their performances. Furthermore, experiments are carried out on different projects of unknown vulnerability types on our proposed RD dataset to evaluate the effectiveness of different detection methods in practical applications.The experimental results are reported in Tables I and II.

In summary, our proposed model achieves state-of-the-art performance and significant improvements on both datasets. In particular, when faced with the sophisticated real-project samples from RD, the efficiency of almost all methods can be seen to significantly decrease; however, the relative precision (P) and F1 score gains achieved by our model is an average of 17.0% and 12.0%. It is further demonstrated that our proposed model can still maintain good vulnerability detection performance compared with other methods in practical applications.

2) Line-level Location Results: The line-level localization performance of the model is evaluated on four projects contained in our proposed RD; more detailed statistics are shown in Table III.

TABLE III
DETAILED LINE-LEVEL STATISTICS OF RD.

	ALP[1]	AFL[2]	PVFP[3]	PVP[4]
Asterisk	8546	160	0.16%	0.12%
Wireshark	5680	265	0.70%	0.02%
Libtiff	2871	96	1.18%	0.43%
Openssl	1592	184	0.68%	0.07%

[1] Average number of code lines per program.
[2] Average number of code lines per function.
[3] Percentage of vulnerable functions in the projects.
[4] Percentage of vulnerable code lines in the projects.

As is evident, vulnerability lines account for only a very small part of a program, and our goal is to more efficiently implement source code security audits during the software development phase. Therefore, we introduce two indicators of $HitRate$ and vulnerability line $Coverage$ of the model from the perspective of graph, which are expressed as follows:

$$HitRate = \sum_{i=0}^{m} n_{hits} / \sum_{i=0}^{m} \lceil kn \rceil, \tag{10}$$

$$Coverage = \sum_{i=0}^{m} n_{hits} / \sum_{i=0}^{m} n_{vul}, \tag{11}$$

where n_{hits} represents the number of nodes correctly predicted by our model, and n_{vul} is the number of nodes related to the vulnerability code in each CPG. We graph the experimental results in Figure 4.

In addition to function-level vulnerability location, we further divide three location ranges according to AFL to verify the effectiveness of the method under different granularities. It can be observed that, in most cases, the model's coverage of vulnerability lines can be maintained at a high level. Furthermore, a vulnerability location within 50 lines can reduce the amount of code required for function-level code auditing by at least 47.9% while still maintaining a ralatively promising hit rate.

V. CONCLUSION

In this paper, we propose a novel GNN-based source code vulnerability detection model designed to achieve fine-grained potential vulnerable code identification at a function level and line level through the intermediate representation of multidimensional program features. Extensive experiments reveal the superior performance of our model compared with other state-of-the-art methods. It is further demonstrated that our approach can be applied to support the source code vulnerability detection of real projects, which greatly reduces the workload associated with manual code audits.

REFERENCES

[1] NVD Analysis 2022: A Call to Action on Software Supply Chain Security. (2022, Dec 07). [Online]. Available: https://www.secure.software/reports/reversinglabs-nvd-analysis-2022-a-call-to-action-on-software-supply-chain-security

[2] R. Malhotra, "A systematic review of machine learning techniques for software fault prediction," *Applied Soft Computing*, vol. 27, pp. 504–518, 2015.

[3] S. K. Singh and A. Chaturvedi, "Applying deep learning for discovery and analysis of software vulnerabilities: A brief survey," *Soft Computing: Theories and Applications*, pp. 649–658, 2020.

[4] Y. Chen, "Convolutional neural network for sentence classification," Master's thesis, University of Waterloo, 2015.

[5] G. Lin, J. Zhang, W. Luo, L. Pan, and Y. Xiang, "Poster: Vulnerability discovery with function representation learning from unlabeled projects," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2539–2541.

[6] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, "Automated vulnerability detection in source code using deep representation learning," in *2018 17th IEEE international conference on machine learning and applications (ICMLA)*. IEEE, 2018, pp. 757–762.

[7] F. Wu, J. Wang, J. Liu, and W. Wang, "Vulnerability detection with deep learning," in *2017 3rd IEEE international conference on computer and communications (ICCC)*. IEEE, 2017, pp. 1298–1302.

[8] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun, "Spectral networks and locally connected networks on graphs," *arXiv preprint arXiv:1312.6203*, 2013.

[9] S. Cao, X. Sun, L. Bo, Y. Wei, and B. Li, "Bgnn4vd: constructing bidirectional graph neural-network for vulnerability detection," *Information and Software Technology*, vol. 136, p. 106576, 2021.

[10] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet," *IEEE Transactions on Software Engineering*, pp. 1–1, 2021.

[11] W. Zheng, Y. Jiang, and X. Su, "Vu1spg: Vulnerability detection based on slice property graph representation learning," in *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2021, pp. 457–467.

[12] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," *Advances in neural information processing systems*, vol. 32, 2019.

[13] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "Sysevr: A framework for using deep learning to detect software vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, 2021.

[14] 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," *arXiv preprint arXiv:1801.01681*, 2018.

[15] D. Hin, A. Kan, H. Chen, and M. A. Babar, "Linevd: Statement-level vulnerability detection using graph neural networks," *arXiv preprint arXiv:2203.05181*, 2022.

[16] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, "μvuldeepecker: A deep learning-based system for multiclass vulnerability detection," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 5, pp. 2224–2236, 2021.

[17] Z. Li, D. Zou, S. Xu, Z. Chen, Y. Zhu, and H. Jin, "Vuldeelocator: a deep learning-based fine-grained vulnerability detector," *IEEE Transactions on Dependable and Secure Computing*, 2021.

[18] G. Fan, X. Diao, H. Yu, K. Yang, and L. Chen, "Software defect prediction via attention-based recurrent neural network," *Scientific Programming*, vol. 2019, 2019.

[19] G. Lin, J. Zhang, W. Luo, L. Pan, Y. Xiang, O. De Vel, and P. Montague, "Cross-project transfer representation learning for vulnerable function discovery," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 7, pp. 3289–3297, 2018.

[20] S. Liu, G. Lin, Q.-L. Han, S. Wen, J. Zhang, and Y. Xiang, "Deepbalance: Deep-learning and fuzzy oversampling for vulnerability detection," *IEEE Transactions on Fuzzy Systems*, vol. 28, no. 7, pp. 1329–1343, 2019.

[21] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 590–604.

[22] 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 *IJCAI*, 2019, pp. 4665–4671.

[23] J. Lee, I. Lee, and J. Kang, "Self-attention graph pooling," in *International conference on machine learning*. PMLR, 2019, pp. 3734–3743.

[24] F. Yamaguchi, "Pattern-based vulnerability discovery," 2015.

[25] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[26] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Icml*, 2010.

[27] Q. Li, Z. Han, and X.-M. Wu, "Deeper insights into graph convolutional networks for semi-supervised learning," in *Thirty-Second AAAI conference on artificial intelligence*, 2018.

[28] K. Xu, C. Li, Y. Tian, T. Sonobe, K.-i. Kawarabayashi, and S. Jegelka, "Representation learning on graphs with jumping knowledge networks," in *International conference on machine learning*. PMLR, 2018, pp. 5453–5462.

[29] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," *arXiv preprint arXiv:1607.06450*, 2016.

[30] C. Cangea, P. Veličković, N. Jovanović, T. Kipf, and P. Liò, "Towards sparse hierarchical graph classifiers," *arXiv preprint arXiv:1811.01287*, 2018.

[31] J. Atwood and D. Towsley, "Diffusion-convolutional neural networks," *Advances in neural information processing systems*, vol. 29, 2016.

[32] M. Simonovsky and N. Komodakis, "Dynamic edge-conditioned filters in convolutional neural networks on graphs," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 3693–3702.

[33] H. Gao and S. Ji, "Graph u-nets," in *international conference on machine learning*. PMLR, 2019, pp. 2083–2092.

[34] M. Zhang, Z. Cui, M. Neumann, and Y. Chen, "An end-to-end deep learning architecture for graph classification," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 32, no. 1, 2018.

[35] Cppcheck: A tool for static C/C++ code analysis. [Online]. Available: http://cppcheck.net

[36] FlawFinder. [Online]. Available: https://dwheeler.com/flawfinder

[37] Rough Audit Tool for Security. [Online]. Available: https://github.com/andrew-d/rough-auditing-tool-for-security

[38] FlintPlusPlus. [Online]. Available: https://github.com/JossWhittle/FlintPlusPlus

[39] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *arXiv preprint arXiv:1710.10903*, 2017.