FINE-GRAINED SOURCE CODE VULNERABILITY DETECTION VIA GRAPH NEURAL NETWORKS

Anonymous authors
Paper under double-blind review

ABSTRACT

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. 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. First of all, detailed information about the source code is extracted through multi-dimensional program feature encoding to facilitate learning about patterns of vulnerability. Second, extensive experiments are conducted on both a public hybrid dataset and our proposed dataset, which is collected entirely from real software projects. It is 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. Finally, a novel location module is designed to identify potential key vulnerable lines of code. And the effectiveness of the model and its contributions to reducing human workload in practical production are evaluated.

1 INTRODUCTION

According to a report released by the National Institute of Standards and Technology (NIST) (Jonathan Greig , 2021), the number of vulnerabilities found in 2021 has reached 18378. Cyber security industry insiders have pointed out that the persistence of many exploitable vulnerabilities in software applications indicate that these vulnerabilities have not been alleviated in time, meaning that it is increasingly difficult to defend software against attacks. These bugs, which should have been optimized early in the development process, appear to have become an insurmountable security debt for security practitioners. 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.

Early source code-oriented static vulnerability analysis techniques can be divided into code similarity-based, rule-based, and symbolic execution-based methods. Although almost all of these methods can achieve high accuracy under specific deployment scenarios, they are also overly reliant on having complete prior knowledge of vulnerabilities, a high workload for human programmers, and limited scope of application; as a result, they cannot flexibly deal with new vulnerabilities and sophisticated vulnerability variants.

Moreover, the accuracy of conventional machine learning (ML) algorithms (i.e. Support Vector Machine, Decision Tree, Random Forest, etc.) for static source code analysis largely depends on the feature engineering completed by domain experts. When faced with sophisticated functions and increasingly large scales of software source code, this work undoubtedly becomes very onerous and impractical (Malhotra, 2015; Singh & Chaturvedi, 2020).

By contrast, deep learning (DL) technology solves the drawbacks of conventional ML and can automatically extract features from objects provided that heuristic guidance strategies have been obtained. However, source code is typically a structured language. In the feature extraction stage, deep

neural networks such as Convolution Neural Network (CNN), and Recurrent Neural Network (RNN) often regard such code as a natural language (Chen, 2015; Wu et al., 2017; Lin et al., 2017; Russell et al., 2018). As a result, the serialized flat intermediate representation only retains the semantics of the code, which leads to serious loss of program logic and structure information, and moreover limits the great potential of DL model to perform vulnerability feature learning to a certain extent.

In recent years, the emergence of GNNs (Bruna et al., 2013) has provided new insights into the static vulnerability analysis of source code. Some existing studies (Chakraborty et al., 2021; Zhou et al., 2019; Cao et al., 2021; Zheng et al., 2021) have extracted specific source code properties, such as abstract syntax trees (AST), control flow graphs (CFG), and data flow graphs (DFG), as intermediate representations, and combined them with GNNs to identify potential vulnerabilities.

At present, there is an urgent need for relatively complete and recognized datasets of vulnerabilities in this field of research. Notably, most of the datasets leveraged in current research are collected through time-consuming and laborious manual labeling or using automatic labeling tools, which may cause a certain rate of errors and a high sample repetition rate. Furthermore, many ML-based vulnerability detection models are trained on datasets that are partially or wholly composed of synthetic samples. The research shows that because the form of these samples is too simple compared with the vulnerabilities that exist in real projects, the model's vulnerability detection accuracy is significantly reduced when facing real source code data (Chakraborty et al., 2021).

Moreover, due to the limitation of the granularity of dataset labeling and the model itself, vulnerability location in most studies tends to concentrate at the function level. However, when faced with large-scale source code projects, there are often only a few lines of code that lead to vulnerabilities, which undoubtedly imposes a burden on further code audits. By contrast, although some fine-grained locations at the slice level (Li et al., 2018c; 2021b) or line level (Li et al., 2021a) can more directly reflect the location of vulnerabilities is often accompanied by strict data labeling requirements and laborious data preprocessing process.

In brief, it is necessary to achieve accurate vulnerability location and an efficient and automated source code vulnerability detection. To this end, we propose a new fine-grained vulnerability detection model based on GNNs, which can identify key vulnerability lines while also realizing function-level source code vulnerability detection.

First of all, we convert the source code file into an intermediate representation, namely Code Property Graphs (CPGs), at the function level. We then attempt to capture more abundant source code information through multidimensional program feature encoding in order to achieve the effective extraction of important vulnerability features. Second, the pooling layer of the GNNs is improved, meaning that the proposed model can still efficiently incorporate node feature information into the training process to identify functions and achieve higher accuracy compared with other baselines. We demonstrate the performance and availability of our method on a widely used hybrid dataset, along with our proposed dataset, which is derived entirely from real projects. Finally, another key innovation of our work is that we design a unique location module that employs a learnable parameterized weight matrix in combination with a self-attention mechanism to identify key graph nodes, which are mapped to the source file to locate lines of code with high vulnerability scores. It is worth noting that our proposed method can achieve efficient fine-grained source code vulnerability detection without the need for heavy manual engineering.

The contributions of our work can be summarized as follows:

- We propose a novel GNN-based source code vulnerability detection model, which learns source code information through the intermediate representation of multidimensional program features in order to capture potential vulnerability feature patterns and achieve efficient automated vulnerability detection.
- Our model is evaluated on a public hybrid dataset and our proposed dataset, which is collected from 13 real-world projects. It outperforms all other state-of-the-art methods by a large margin, which demonstrates its superior vulnerability identification performance.
- The results of locating potential vulnerability lines on four projects from the proposed dataset demonstrate the effectiveness of the location module in our model, which can contribute to reducing developers' workload during practical code audits.

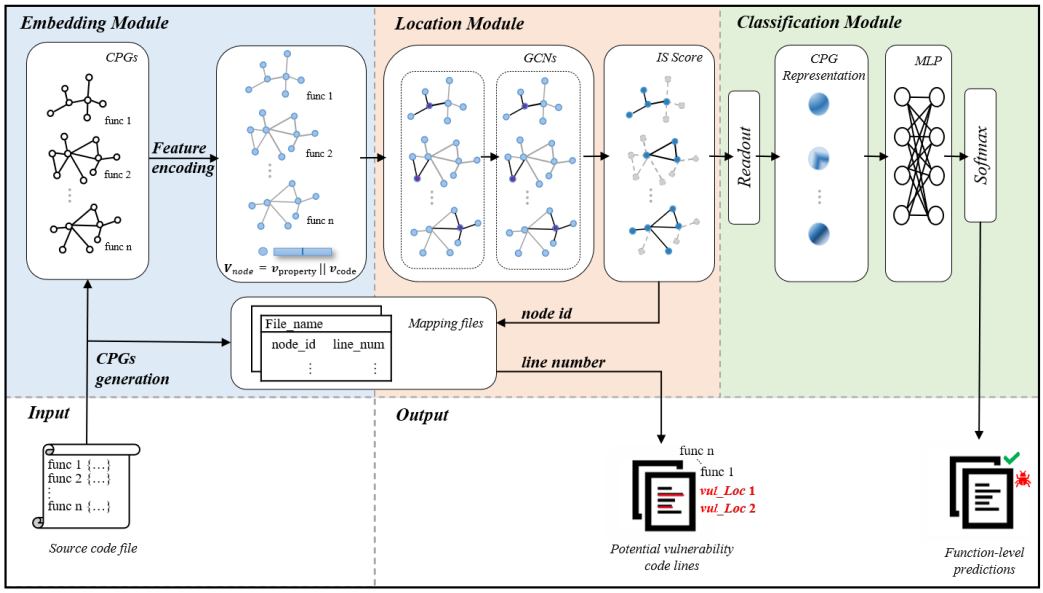

Figure 1: The architecture of our model.

2 Related Works

As the earliest DL-based vulnerability detection systems, Vuldeepecker (Li et al., 2018c) and Sysevr (Li et al., 2021b) 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 (Zou et al., 2021) and VulDeeLocator (Li et al., 2021a). In addition, many studies extract code semantics based on AST and adopt vulnerability detection models in combination with BiLSTM (Lin et al., 2017; 2018; Fan et al., 2019; Liu et al., 2019), which attempt to achieve high classification precision at the function level.

The CPG was first proposed by Yamaguchi et al. (2014), providing a new insight into source code vulnerability feature extraction. Some studies have realized function-level source code vulnerability identification based on GNNs (Chakraborty et al., 2021; Zhou et al., 2019; Cao et al., 2021). These studies prove that these methods can effectively capture the program structure and node information carried by the CPG and its variants (Zhou et al., 2019; Duan et al., 2019; Zheng et al., 2021). 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.

3 Method

Inspired by the graph classification model SAGPool (Lee et al., 2019), our model aims to better capture the inter-node dependencies in CPGs while learning the important feature information of the program. Based on the novel node influence score calculation method, the localization layer is proposed in order to output nodes with high scores and thereby locate potential vulnerable code lines. In addition, an improved global pooling method is adopted to more effectively aggregate the global information of CPGs and achieve more accurate function-level vulnerability identification.

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 $f : \mathcal{G} \rightarrow Y$. 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

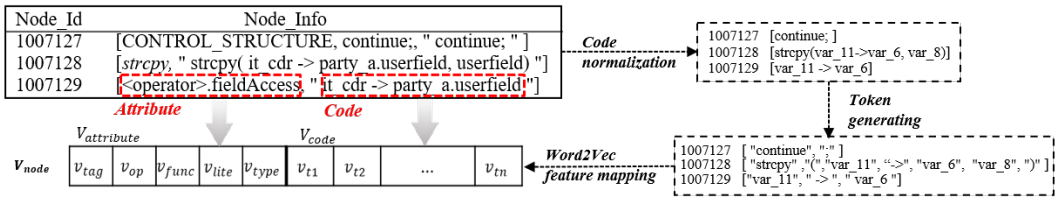

Figure 2: The node feature embedding process.

a cross-entropy loss by minimizing the negative loglikelihood below:

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

Given the source code files, without the need for domain experts to manually predefine vulnerability features, the vulnerability pattern can be automatically learned by the proposed model, which enables it to determine whether the program is vulnerable or not. The architecture of our model, illustrated in Figure 1, comprises the following three modules: 1) *Embedding module*. The CPGs, which combines five kinds of code properties, is 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 that further considers the dependencies and inter-node relationships among the key nodes and generates the global representation of CPG for identifying vulnerable functions.

3.1 Embedding Module

3.1.1 Code Property Graph Generation

The key insight behind using this function-level composite graph as an intermediate representation is that the important information of the source code can be more fully retained and that the vulnerability pattern can be explored in the feature extraction phase. Compared with using a single property, CPGs have been shown to be able to model more common vulnerability types (Yamaguchi, 2015), enabling it to achieve efficient vulnerability mining.

As for the implementation, Joern (Yamaguchi et al., 2014) is adopted to generate a joint data structure composed of five code properties for source code. In addition to the three data structures integrated in earlier versions of CPG, a control dependency graph (CDG) and a data dependency graph (DDG) also added.

3.1.2 Graph Embedding of Multidimensional Program Features

The spatial characteristic of graphic data is embodied through two aspects: node and structure. The node embedding method is designed to encode program features from a multidimensional point of view. Moreover the spatial feature of CPG is represented by its adjacency matrix.

The node information of CPG consists of two parts: attributes and code. In order to more comprehensively and effectively integrate the node features, and then further associate them with the vulnerability patterns, a node compound feature embedding method is designed to encode the source code from multiple dimensions, including function calls, logical operations, variable types, semantics, and syntax. The process of node feature embedding is summarized in Figure 2.

Node Attribute Embedding The node attribute feature consists of vectors of five fields. According to the *tag*, all the nodes are divided into 12 categories, representing the different roles played by nodes in the CPG. Moreover, 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, nodes with the tag CALL are associated with these two fields. In addition, 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.

Node Code Embedding Each node of CPG is associated with code in the source file. Therefore, the corresponding code statements of nodes are vectorized to encode semantic information. 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 $V_{attribute}$.

3.2 Location Module

3.2.1 GCN Layers

To aggregate the neighborhood information, we use graph convolutional network (GCN), first proposed by Kipf & Welling (2016). GCN can realize the end-to-end learning of graph data, and the supervision signal of the entire neural network guides both the GCN layer and the parameters of subsequent graph representation learning; as a result, the feature representations of nodes are more adaptive to the classification tasks of different source code vulnerabilities downstream.

Moreover, given that source code is a structured language, the learning of both CPG structure information and property information in the GCN layer will jointly affect the final node representation, making it highly important to explore the potential vulnerability patterns it contains. In addition, this function-level representation also limits the size of the graph structure to some extent. Based on the above considerations, we deem GCN to be more suitable for CPG node feature learning than other neural networks.

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 (Nair & Hinton, 2010), which is used as the activation function.

3.2.2 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. On the other hand, the stacked GCNs cause the over-smoothing of node features to some extent (Li et al., 2018a; Xu et al., 2018), and their impact should be taken into consideration even if our network is relatively shallow. Consequently, 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 to alleviate the above problems. Finally, the $\underline{I}nfluence\ \underline{S}core : IS \in \mathbb{R}^{n\times 1}$ of CPG nodes can be expressed, as follows:

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

where LN is a layer normalization (Ba et al., 2016).

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

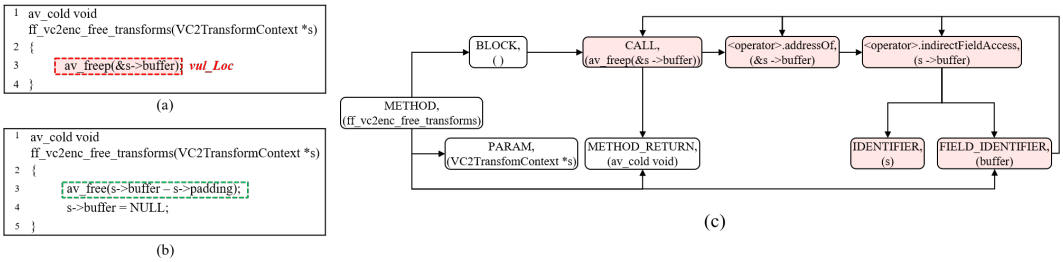

Figure 3: (a): Example of a vulnerable function with an Out-of-bounds Read Error (CWE-125) from Ffmpeg; (b): The fixed vulnerable function; (c): The simplified CPG of the vulnerable function in (a). The red nodes in the CPG corresponding to the red line of code labeled as *vul_ Loc* in (a).

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.

3.3 Classification Module

The purpose of the readout layer (Xu et al., 2018; Cangea et al., 2018) is to obtain a fixed-size global graph representation. 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 (Atwood & Towsley, 2016; Simonovsky & Komodakis, 2017) ignore the interaction between nodes, or cause the loss of node information (Zhang et al., 2018; Gao & Ji, 2019). 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.

4 Experiments

In this section, we conduct extensive experiments on two datasets to evaluate the performance of the proposed model and compare it with that of state-of-the-art vulnerability detection methods. Furthermore, we conduct experiments on different real software applications to demonstrate the effectiveness of the proposed model in performing fine-grained vulnerability location. Finally, we conduct ablation experiments to study the effectiveness of our proposed improvements.

4.1 Datasets

The experiments are carried out on two datasets: the Hybrid Dataset (HD) and the Real-project Dataset (RD). The details of the datasets are shown in Table 1.

Table 1: The details of Hybrid Dataset (HD) and Real-project Dataset (RD).

	Programs	CPGs	PVF*	CWE Type
HD	16137	154430	5.5%	CWE-119, CWE-399
RD	17752	1129800	8.7%	CWE-119, CWE-399, CWE-668, CWE-020, CWE-264 ...

* Percentage of vulnerable functions.

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 (Li et al., 2018c) is utilized in the experiments. It is the first dataset suitable for evaluating DL-based vulnerability detection systems and is composed of real-project data from National Vulnerability Database (NVD) and synthetic data from Software Assurance Reference Dataset (SARD), which includes two known vulnerabilities: Memory Buffer Errors (CWE-119) and Resource Management Errors(CWE-399).

In the data pre-processing phase, 16137 C/C++ programs were further processed and applied to our vulnerability detection model in the form of graph data, rather than in the slicing form adopted by Vuldeepecker (Li et al., 2018c) in the original dataset.

Real-project Dataset (RD) According to statistics, samples from real projects account for only a small proportion of HD. To further prove that our proposed model retains efficient automatic vulnerability detection capability when faced with real software application scenarios, and can accordingly make a real contribution to the code security audit in actual production, we collect the dataset RD from completely real projects.

RD contains 17752 programs with function-level and partial line-level data labeling for 13 popular C/C++ libraries, including Asterisk, FFmpeg, Libpng, OpenSSL, ImageMagick, Libtiff, Linux kernel, PHP, Qemu, VLC media player, Chrome, Wireshark, and Xen.

In the training phase, we set dropout to 0.5, the dimension of hidden states to 32, and ReLU as the activation function for both GCN layers and fully connected layers. The graph representation is fixed at 64. We employ the Adam optimizer with a learning rate of 5e-3 and weight decay of 5e-4. We run experiments on a machine with NVIDIA Quadro RTX 6000 GPU and Intel Xeon Gold 6240 CPU operating at 2.60GHz.

4.2 BASELINES

We select two kinds of methods in the field of static source code vulnerability analysis for performance comparison: the state-of-the-art ML-based vulnerability detection models, and commercial code analysis tools widely used in practical production.

ML-based Vulnerability Detection Models We utilize XGBoost and CNN to conduct the comparison with conventional ML models and classical CNN. Vuldeepecker (Li et al., 2018c) applies slice-level program representation to BiLSTM high-level representation learning to achieve fine-grained vulnerability detection. We use the same experimental settings to compare the performance of our proposed method with that of the typical DL model. Devign (Zhou et al., 2019) adopt GNNs as the backbone to predict source code vulnerability at the function level. We conduct experiments on the reproducible version of this method released by Chakraborty et al. (2021) to further evaluate the performance of our model compared with other GNN-based methods.

Commercial Code Analysis Tools Cppcheck is a static analysis tool for analyzing C/C++ code with the goal of having a very low rate of false positives. Flawfinder is a useful tool for scanning C/C++ code and reporting potential security vulnerabilities, which can be used as a simple guide to static source code analysis tools. RATS is a rough auditing tool developed by Secure Software Inc. that greatly aids manual code inspection. Flint++ is a cross-platform, zero-dependency port of Flint, developed and used on Facebook to flag errors and bad practices for review.

Table 2: Comparison of function-level classification of known CWE types on HD and RD datasets.

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	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	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	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	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	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++	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 3: Comparison of function-level classification of real-world projects with unknown CWE types on RD datasets.

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	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	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	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	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	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++	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**

4.3 RESULTS

4.3.1 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 2 and 3.

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.

4.3.2 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 4.

Table 4: 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.

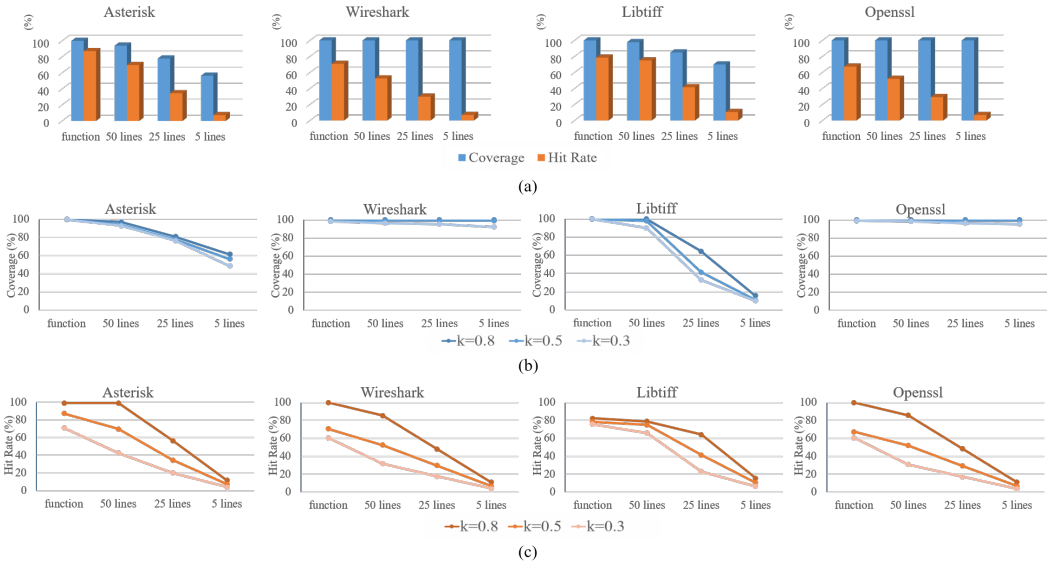

Figure 4: Line-level location results on four projects. (a): The model's coverage and hit rate of vulnerability lines with k = 0.5; (b): The model's coverage of vulnerability lines under different k value; (b): The model's hit rate of vulnerability lines under different k value.

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 hit rate and vulnerability line coverage of the model, which are expressed as follows:

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

$$Coverage = hits / total\_err, \tag{11}$$

where N_{hits} represents the number of correctly located CPG nodes related to the vulnerability code, and $hits$ is the number of correctly located vulnerability code lines, while $total\_err$ is the total number of vulnerability code lines contained in this project. 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.

Furthermore, we conduct ablation study on different feature encoding methods and pooling methods to demonstrate that our improved methods aid the model to further realize effective vulnerability identification. The detailed of ablation study is shown in Appendix A.1.

5 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

James Atwood and Don Towsley. Diffusion-convolutional neural networks. *Advances in neural information processing systems*, 29, 2016.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. Spectral networks and locally connected networks on graphs. *arXiv preprint arXiv:1312.6203*, 2013.

Cătălina Cangea, Petar Veličković, Nikola Jovanović, Thomas Kipf, and Pietro Liò. Towards sparse hierarchical graph classifiers. *arXiv preprint arXiv:1811.01287*, 2018.

Sicong Cao, Xiaobing Sun, Lili Bo, Ying Wei, and Bin Li. Bgnn4vd: constructing bidirectional graph neural-network for vulnerability detection. *Information and Software Technology*, 136: 106576, 2021.

Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering*, pp. 1–1, 2021. doi: 10.1109/TSE.2021.3087402.

Yahui Chen. Convolutional neural network for sentence classification. Master's thesis, University of Waterloo, 2015.

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 *IJCAI*, pp. 4665–4671, 2019.

Guisheng Fan, Xuyang Diao, Huiqun Yu, Kang Yang, and Liqiong Chen. Software defect prediction via attention-based recurrent neural network. *Scientific Programming*, 2019, 2019.

Hongyang Gao and Shuiwang Ji. Graph u-nets. In *international conference on machine learning*, pp. 2083–2092. PMLR, 2019.

Jonathan Greig . With 18,378 vulnerabilities reported in 2021, nist records fifth straight year of record numbers. `https://www.zdnet.com/article/with-18376-vulnerabilities-found-in-2021-nist-reports-fifth-straight-year-of-record-numbers`, 2021. Accessed: 2021-12-08.

Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

Junhyun Lee, Inyeop Lee, and Jaewoo Kang. Self-attention graph pooling. In *International conference on machine learning*, pp. 3734–3743. PMLR, 2019.

Qimai Li, Zhichao Han, and Xiao-Ming Wu. Deeper insights into graph convolutional networks for semi-supervised learning. In *Thirty-Second AAAI conference on artificial intelligence*, 2018a.

Ruoyu Li, Sheng Wang, Feiyun Zhu, and Junzhou Huang. Adaptive graph convolutional neural networks. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018b.

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*, 2018c.

Zhen Li, Deqing Zou, Shouhuai Xu, Zhaoxuan Chen, Yawei Zhu, and Hai Jin. Vuldeelocator: a deep learning-based fine-grained vulnerability detector. *IEEE Transactions on Dependable and Secure Computing*, 2021a.

Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 2021b.

Guanjun Lin, Jun Zhang, Wei Luo, Lei Pan, and Yang Xiang. Poster: Vulnerability discovery with function representation learning from unlabeled projects. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2539–2541, 2017.

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.

Shigang Liu, Guanjun Lin, Qing-Long Han, Sheng Wen, Jun Zhang, and Yang Xiang. Deepbalance: Deep-learning and fuzzy oversampling for vulnerability detection. *IEEE Transactions on Fuzzy Systems*, 28(7):1329–1343, 2019.

Ruchika Malhotra. A systematic review of machine learning techniques for software fault prediction. *Applied Soft Computing*, 27:504–518, 2015.

Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Icml*, 2010.

Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE international conference on machine learning and applications (ICMLA)*, pp. 757–762. IEEE, 2018.

Martin Simonovsky and Nikos Komodakis. Dynamic edge-conditioned filters in convolutional neural networks on graphs. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 3693–3702, 2017.

Shashank Kumar Singh and Amrita Chaturvedi. Applying deep learning for discovery and analysis of software vulnerabilities: A brief survey. *Soft Computing: Theories and Applications*, pp. 649–658, 2020.

Fang Wu, Jigang Wang, Jiqiang Liu, and Wei Wang. Vulnerability detection with deep learning. In *2017 3rd IEEE international conference on computer and communications (ICCC)*, pp. 1298–1302. IEEE, 2017.

Keyulu Xu, Chengtao Li, Yonglong Tian, Tomohiro Sonobe, Ken-ichi Kawarabayashi, and Stefanie Jegelka. Representation learning on graphs with jumping knowledge networks. In *International conference on machine learning*, pp. 5453–5462. PMLR, 2018.

Fabian Yamaguchi. Pattern-based vulnerability discovery. 2015.

Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pp. 590–604. IEEE, 2014.

Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen. An end-to-end deep learning architecture for graph classification. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

Weining Zheng, Yuan Jiang, and Xiaohong Su. Vu1spg: Vulnerability detection based on slice property graph representation learning. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, pp. 457–467. IEEE, 2021.

Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*, 32, 2019.

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*, 18(5):2224–2236, 2021. doi: 10.1109/TDSC.2019.2942930.

A APPENDIX

A.1 ABLATION STUDY

To study how our proposed multidimensional program feature encoding method influences the model's effective learning of source code vulnerability features, several feature variables are assessed. The details and results of the ablation study are shown in Table 5. We provide the following analysis of these results:

Table 5: Ablation study on feature encoding methods

Feature	CWE-119		CWE-399	
	P	F1	P	F1
$V_{attribute}$	90.8	91.8	58.2	46.0
V_{code}	92.1	93.1	65.3	47.2
Ours	98.2	97.4	98.9	99.0

Our proposed feature encoding method achieves better performance than other variables. It is demonstrated that this method aids the model to further realize effective vulnerability feature learning. And compared with $V_{attribute}$, V_{code} makes more significant contributions to vulnerability identification.

Furthermore, we conduct an ablation study to underline the importance of the BiLSTM-based readout layer to generating graph representations in order to achieve accurate vulnerable function identification. Three kinds of graph pooling methods —namely max pooling (Li et al., 2018b), average pooling (Atwood & Towsley, 2016), and SAGPool (Lee et al., 2019) —are adopted to aggregate the node information output from the upper layer. The results are shown in Table 6:

Table 6: Ablation study on pooling methods

Method	CWE-119		CWE-399	
	P	F1	P	F1
Max Pooling	85.9	90.6	73.8	48.2
Average Pooling	84.7	91.5	68.8	50.6
SAGPool	88.1	92.5	60.8	53.0
Ours	98.2	97.4	98.9	99.0

Our BiLSTM-based readout layer achieves the best performance, which proves its effectiveness in obtaining the dependencies and inter-node relationships among key nodes and generating meaningful graph representation for further classification.