EAI Endorsed Transactions
on Scalable Information Systems
Research Article EAI.EU

E-GVD: Efficient Software Vulnerability Detection Techniques Based on Graph Neural Network

Haiye Wang[2], Zhiguo Qu[1,2,*] and Le Sun[1,2]

[1]Engineering Research Center of Digital Forensics, Ministry of Education, Nanjing University of Information Science and Technology, Nanjing, 210044, Jiangsu, China

[2]School of Computer Science, Nanjing University of Information Science and Technology, Nanjing, 210044, Jiangsu, China

Abstract

INTRODUCTION: Vulnerability detection is crucial for preventing severe security incidents like hacker attacks, data breaches, and network paralysis. Traditional methods, however, face challenges such as low efficiency and insufficient detail in identifying code vulnerabilities.
OBJECTIVES: This paper introduces E-GVD, an advanced method for source code vulnerability detection, aiming to address the limitations of existing methods. The objective is to enhance the accuracy of function-level vulnerability detection and provide detailed, understandable insights into the vulnerabilities.
METHODS: E-GVD combines Graph Neural Networks (GNNs), which are adept at handling graph-structured data, with residual connections and advanced Programming Language (PL) pre-trained models.
RESULTS: Experiments conducted on the real-world vulnerability dataset CodeXGLUE show that E-GVD significantly outperforms existing baseline methods in detecting vulnerabilities. It achieves a maximum accuracy gain of 4.98%, indicating its effectiveness over traditional methods.
CONCLUSION: E-GVD not only improves the accuracy of vulnerability detection but also contributes by providing fine-grained explanations. These explanations are made possible through an interpretable Machine Learning (ML) model, which aids developers in quickly and efficiently repairing vulnerabilities, thereby enhancing overall software security.

Received on 07 February 2024; accepted on 20 March 2024; published on 21 March 2024

Keywords: vulnerability detection, graph neural network, pre-trained model, interpretable machine learning

doi:10.4108/eetsis.5056

1. Introduction

Cyberspace security is increasingly becoming a focal point of global concern, with a burgeoning body of research emerging to tackle cyber threats [1–4]. In recent years, the surge in software vulnerabilities has significantly impacted software development and maintenance. To combat this, machine learning (ML) [5] and deep learning (DL) [6] have been applied to vulnerability detection, treating it as a binary classification task. Traditional methods depended on manual feature extraction, which is inefficient for the constantly evolving code libraries. In contrast, DL, especially graph-based DL techniques using graph neural networks (GNNs), has shown promise by learning from code structures to detect vulnerabilities. However, these techniques still face challenges in the preprocessing of source code and in providing detailed, actionable insights. The high heterogeneity and dynamism of code require complex and dynamic preprocessing processes. This not only increases the workload before model training but may also affect the model generalizability and accuracy. Moreover, although existing technologies can identify potential vulnerabilities, they rarely provide sufficient information to explain the specific reasons behind the vulnerabilities or offer targeted repair suggestions, limiting their practical utility in the actual development process.

To overcome these limitations, we introduce E-GVD, an advanced GNN-based methodology that interprets

*Corresponding author. Email: 002359@nuist.edu.cn

programming languages by converting source code into graph structures, with tokens as nodes and their relations as edges. Utilizing the GraphCodeBERT model for code representation and graph convolutional network (GCN) layers for deep learning, E-GVD enhances vulnerability detection. It employs various pooling strategies for graph embedding and leverages GNNExplainer for interpretable, fine-grained analysis, pinpointing specific vulnerability locations, thus offering a practical and transparent solution for software security.

The main contributions and work of this paper are summarized as follows.

First, we introduce E-GVD, a more sophisticated GNN-based vulnerability detection method integrated with the interpretability technology of GNNExplainer. The model conducts vulnerability detection at a coarse-grained, function level. Additionally, it offers detailed explanations for the predictions. These explanations are provided through an optimized model framework combined with the interpretability of machine learning.

Second, in coarse-grained vulnerability detection, we incorporate the consideration of local programming logic. This is achieved by efficiently transforming the source code into a graph representation, thereby reducing hardware resource consumption. Furthermore, we employ advanced programming language (PL) pre-trained methods and residual connections, whose synergy significantly enhances the capability of the model in vulnerability detection.

Third, through simulation experiments, we have demonstrated the effectiveness of E-GVD in both vulnerability detection and interpretability. Compared to existing technologies, E-GVD exhibits higher efficiency and accuracy when processing real-world vulnerability data samples. It provides in-depth explanations for vulnerability detection, aiding in the precise localization of vulnerabilities.

2. Related Work

2.1. Vulnerability Detection Based on Deep Learning

Deep learning (DL) is being extensively used in various fields [7–11]. Since 2020, significant advancements have been made in DL-based vulnerability detection. In 2020, Wang *et al.* [12] proposed FUNDED, leveraging a multi-relation gated graph neural network (GGNN) for enhanced detection performance. In 2021, Li *et al.* [13] introduced the concepts of SyVCs and SeVCs and proposed a DL-based system framework called SySeVR for detecting C/C++ source code vulnerabilities. In 2022, Rathore *et al.* [14] explored the vulnerability of Android malware detection models to adversarial samples and introduced the ACE strategy. In 2023, Jiang *et al.* [15] combined long short-term memory network (LSTM) with convolutional neural network (CNN) in VDDL for detecting various types of vulnerabilities in smart contracts.

However, these algorithms can mostly only predict whether the source code at the block or function level is vulnerable, and cannot provide more comprehensive location information related to the vulnerability. In view of this issue, we attempt to integrate interpretable techniques, which have been widely applied in various domains [16, 17], on top of function-level vulnerability detection methods, thereby providing information related to vulnerability code lines while ensuring the performance of vulnerability detection.

2.2. Graph Neural Networks

Graph Neural Networks (GNNs) have advanced significantly with deep learning, showing remarkable performance in diverse fields [18–20]. GNN-based models for vulnerability detection fall into graph classification and regression categories. In 2020, Cui *et al.* [21] used code similarity to detect vulnerabilities based on Weighted Feature Graphs (WFGs). In 2021, Cao *et al.* [22] proposed a model using Bidirectional Graph Neural Network (BGNN) to detect vulnerabilities that leverages "back edges" to differentiate vulnerable from normal code.In 2022, Yin *et al.* [23] introduced MAGCN, which formulates vulnerability coexploitation behavior discovery as a link prediction problem between vulnerability entities. In 2023, Wang *et al.* [24] proposed a vulnerability detection method named VulGraB, based on graph embedding and bidirectional gated graph neural network (BiGGNN).

Although graph representations effectively convey the logic and structure of source code, the complex preprocessing steps in existing methods are both time-consuming and challenging to implement when dealing with large volumes of open-source code. Our work leverages advanced pre-trained models to streamline this process, effectively initializing node feature vectors to obtain high-quality code embeddings.

3. The Proposed Algorithm E-GVD

3.1. Problem Definition

For coarse-grained vulnerability detection methods, we create a graph $g_i(\mathcal{V}, \boldsymbol{X}, \boldsymbol{A}) \in \mathcal{G}$ for each provided source code function c_i, consisting of $\mathcal{V}$ as the set of m nodes and $\boldsymbol{X} \in \mathbb{R}^{m \times d}$ as the node feature matrix. Each node v_j in $\mathcal{V}$ is represented by a d-dimensional real-valued vector $\boldsymbol{x}_j \in \mathbb{R}^d$, while $\boldsymbol{A} \in \{0,1\}^{m \times m}$ denotes the adjacency matrix, where $\boldsymbol{A}_{v,u}$ is 1 if nodes v and u share an edge, otherwise it is 0. The vulnerability detection model aims to learn a mapping function $f : \mathcal{G} \rightarrow \mathbb{Y}$ to ascertain whether a given source code is vulnerable to attacks. For the interpretable model based on results, we attempt to derive location information for

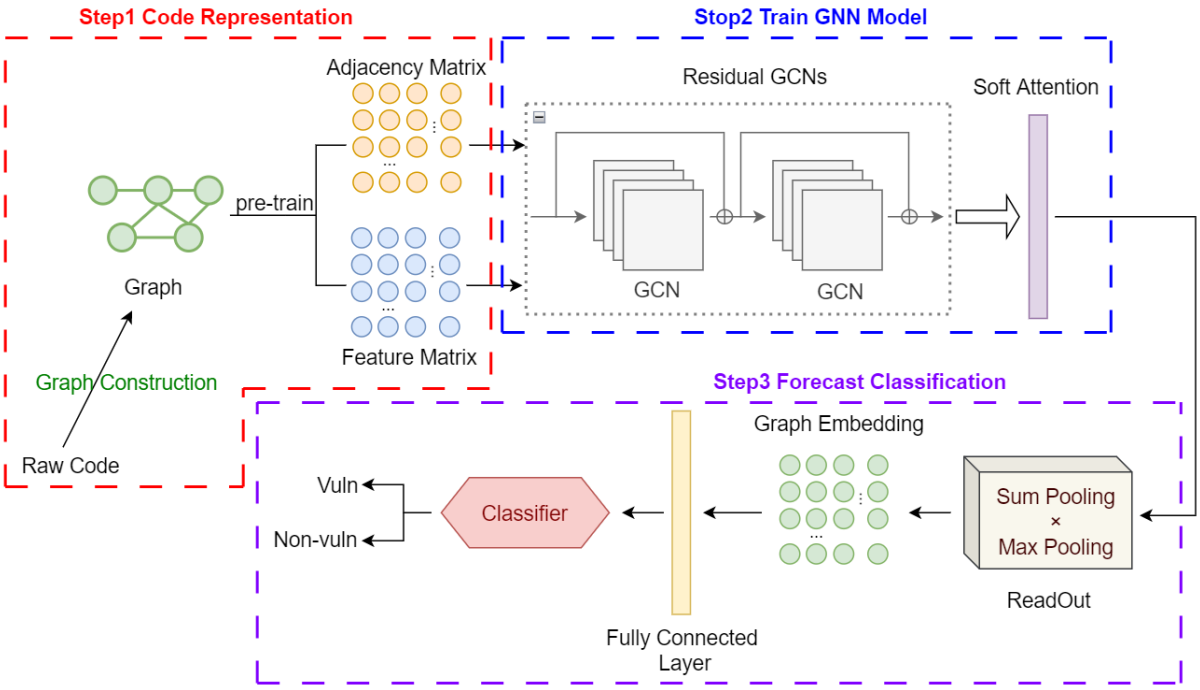

Figure 1. E-GVD Vulnerability Detection Model, the model has three steps: *Step 1 Code Representation*, *Step 2 Train GNN Modelv* and *Step 3 Forecast Classification*.

vulnerable lines of code through induced subgraphs in the output and interpretation results, so that developers can fix vulnerabilities and improve the native security of the code.

3.2. Vulnerability Detection Model Based on Graph Neural Networks

In this section, the input is function source code, aimed at assessing the susceptibility of a given function to attacks. The model initially utilizes PL pre-trained models to enhance understanding of function source code. It then transforms the code into a graph by constructing nodes and edges based on the linear sequence of tokens. Subsequently, the model employs a two-layer GCN with residual connections for deep learning, updating node vector representations by iteratively aggregating vectors from neighboring nodes. Finally, vulnerability detection for function source code is achieved through learning the overall graph embedding vector using a graph-level readout layer. This process encompasses three stages: code representation learning of source code, implementation of the GNN model, and function-level vulnerability detection using global graph embeddings. The diagram of the E-GVD vulnerability detection model is shown as Figure 1.

Step 1: Learning Source Code Representation

E-GVD treats each individual source code function as a token sequence to preserve its local logical structure, which is then transformed into a graph structure. In this graph, each unique token is represented as a node, and the edges between nodes are defined by token co-occurrences within a fixed-size sliding window. The resultant graph can be represented by an adjacency matrix $\boldsymbol{A}$, where the matrix elements are valued based on whether nodes co-occur within the sliding window if the node v and node u co-occur in the sliding window and $v \neq u$, then $\boldsymbol{A}_{v,u}$=1, otherwise $\boldsymbol{A}_{v,u}$=0). An example of the graph construction is illustrated in Figure 2. As the size of such graphs is significantly smaller than the actual length of the source code, it effectively reduces the GPU memory requirements. To provide a deeper understanding of the code, the embedding layer of the pre-trained GraphCodeBERT model is used to initialize the feature vectors of the nodes.

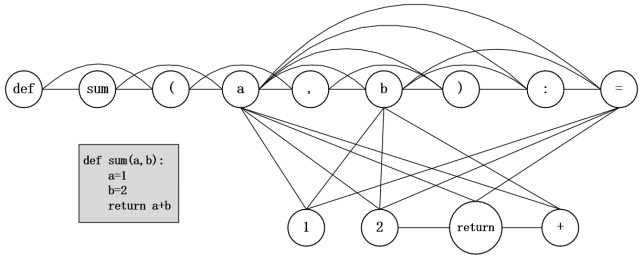

Figure 2. The graph construction method of E-GVD, exemplified with a sliding window size of 3.

Step 2: Implementation of the GNN Model

E-GVD employs a dual-layer GCN as its core architecture. The principal operation of GCN is the aggregation of neighborhood features. Formally, GCN is represented as:

$$\boldsymbol{h}_v^{(k+1)} = \phi\left(\sum_{u\in N_v} a_{v,u}\boldsymbol{W}^{(k)}\boldsymbol{h}_u^{(k)}\right), \forall v \in \mathcal{V}. \quad (1)$$

Here, $\boldsymbol{h}_v^{(k+1)}$ represents the vector representation of node v at the k-th iteration/layer, $\boldsymbol{h}_v^{(0)}$ denotes the feature vector of node v, N_v refers to the neighbor set of node v, $a_{v,u}$ is the edge constant between nodes v and u in the Laplacian re-normalized adjacency matrix $\boldsymbol{D}^{-\frac{1}{2}}\boldsymbol{A}\boldsymbol{D}^{\frac{1}{2}}$, wherein $\boldsymbol{D}$ is the diagonal node degree matrix of $\boldsymbol{A}$, $\boldsymbol{W}^{(k)}$ is the weight matrix, and ϕ is the non-linear activation function, such as $ReLU$.

To augment the expressive and learning capabilities of the model, dual-layer graph convolution operations are utilized to progressively extract and integrate feature information of nodes within the graph. Through the aggregation of information between layers, GCN effectively captures complex relationships in the graph structure. Furthermore, residual connections are incorporated to integrate information learned in lower layers into higher layers. By combining multi-layer structures with residual connections, E-GVD enhances its capability for learning graph data features. The hidden size for different GCN layers is standardized, allowing the residual connections to adapt to the GCN layers. Thus, let redefine Equation (1) as

$$\boldsymbol{h}_v^{(k+1)} = \boldsymbol{h}_v^{(k)} + \phi\left(\sum_{u\in N_v} a_{v,u}\boldsymbol{W}^{(k)}\boldsymbol{h}_u^{(k)}\right). \quad (2)$$

Following the dual-layer GCN with residual connections, a soft attention mechanism layer is embedded to weight the features extracted by the GCN layers. This mechanism dynamically weights the input features, enabling the model to focus on significant node features.

Step 3: Global Graph Embedding

After the iterative learning and updating process in the dual-layer GCN architecture, the information from nodes and edges is locally aggregated and transmitted to adjacent nodes and edges. To enable classification or prediction across the entire graph structure, we develop a graph-level readout layer. Essentially, the primary task of the readout layer is to integrate information from all nodes and edges, forming a comprehensive embedding representation of the entire graph. E-GVD achieves this by combining sum pooling and max pooling in the construction of the graph-level readout layer, thereby converting node-level features into graph-level embedding representations. The generated graph embeddings are defined as follows:

$$\boldsymbol{e}_v = \sigma\left(\boldsymbol{w}^\top\boldsymbol{h}_v^{(K)} + b\right) \odot \phi\left(\boldsymbol{W}\boldsymbol{h}_v^{(K)} + \boldsymbol{b}\right), \forall v \in \mathcal{V}, \quad (3)$$

$$\boldsymbol{e}_g = \sum_{v\in\mathcal{V}} \boldsymbol{e}_v \odot MAXPOOL\{\boldsymbol{e}_v\}_{v\in\mathcal{V}}. \quad (4)$$

Here, $\boldsymbol{e}_v$ represents the final vector representation of node v, $\sigma\left(\boldsymbol{w}^\top\boldsymbol{h}_v^{(K)} + b\right)$ acts as the soft attention mechanism on the nodes, $\boldsymbol{h}_v^{(K)}$ denotes the vector representation of the node v at the last K layer, and $\boldsymbol{e}_g$ signifies the graph embedding generated by the inner product of sum pooling and max pooling.

Subsequently, the generated graph embeddings are passed to a fully connected layer, which integrates and refines key features. Finally, classification is performed through a softmax layer to predict the vulnerability of a given source code function to potential attacks, defined as follows:

$$\hat{\boldsymbol{y}}_g = softmax\left(\boldsymbol{W}_1\boldsymbol{e}_g + \boldsymbol{b}_1\right). \quad (5)$$

For ease of understanding and implementation, let summarize the training process of the E-GVD vulnerability detection model using Algorithm 1 (pseudocode).

Algorithm 1 The Training Process Algorithm of Vulnerability Detection Model

Input: D_{train}: the train dataset; Y: the labels of D_{train};
Output: the trained model M;

for dataloader D_{dl} in D_{train} **do**
  $g(\mathcal{V}, \boldsymbol{X}, \boldsymbol{A}) \leftarrow$ BUILD_GRAPH(D_{dl})
  $\boldsymbol{H}^{(0)} \leftarrow \boldsymbol{X}$
  for k=0,1,...,K-1 **do**
    $\boldsymbol{H}^{(k+1)} \leftarrow \boldsymbol{H}^{(k)} + \mathrm{GCN}\left(\boldsymbol{A}, \boldsymbol{H}^{(k)}\right)$
  end for
  $\boldsymbol{e}_v \leftarrow \sigma\left(\boldsymbol{w}^\top\boldsymbol{h}_v^{(K)} + b\right) \odot \phi\left(\boldsymbol{W}\boldsymbol{h}_v^{(K)} + \boldsymbol{b}\right)$
  $\boldsymbol{e}_g \leftarrow \sum_{v\in\mathcal{V}} \boldsymbol{e}_v \odot MAXPOOL\{\boldsymbol{e}_v\}_{v\in\mathcal{V}}$
  $\boldsymbol{y}_{dl} \leftarrow softmax\left(\boldsymbol{W}_1\boldsymbol{e}_g + \boldsymbol{b}_1\right)$
end for

3.3. Result-based Interpretable Model

To provide a deeper explanation of the predictions made by the GNN-based vulnerability detection model on function source code, this section integrates GNNExplainer [25], a ML-based interpretability model. The architecture of GNNExplainer is closely aligned with the GNN model; it employs a perturbation-based approach to emphasize the nodes and edges crucial for model predictions. The aim is to minimize the difference between the main GNN prediction and the

target output, and maximize the similarity between the generated explanation and the factual basis, thereby providing an interpretation of the predictions. Formally, the output explanation of GNNExplainer is (G_S, X_S), where G_S is the output explanatory subgraph, and X_S is the relevant features of G_S. GNNExplainer takes the trained GNN model, the graph structure $\mathcal{G}$ of the function source code, and the prediction result $\hat{y}$ for that function as the input, with the goal of finding a subgraph G_S in the entire graph $\mathcal{G}$ such that the discrepancy in prediction scores between the entire graph $\mathcal{G}$ and the minimal graph G_S is minimized when input into the model. Figure 3 is used to illustrate how GNNExplainer works.

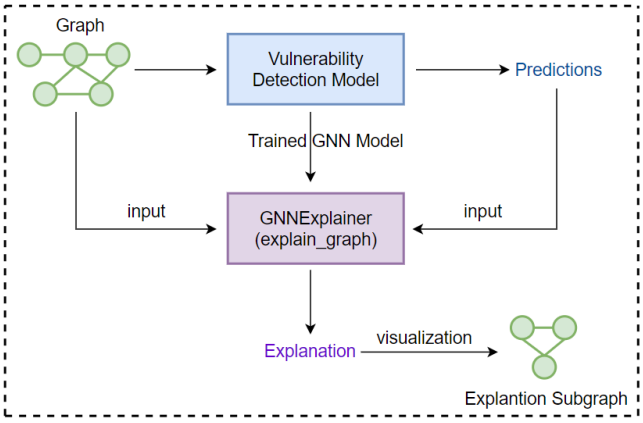

Figure 3. E-GVD Result-based Interpretable Model, the model is centered around the GNNExplainer.

According to [25], when performing graph classification, GNNExplainer optimizes conditional entropy by calculating the mask of the union of adjacency matrices of all nodes in the graph. The resulting subgraph G_S can be directly used as the explanation for prediction $\hat{y}$. For vulnerable source code functions, G_S represents the subgraph containing the statement that causes the vulnerability, while for non-vulnerable source code functions, G_S represents the subgraph containing the important statements that make the function stable.

To facilitate understanding and illustration, let use Algorithm 2 (pseudocode) to outline the implementation process of the E-GVD explanation generation algorithm.

4. The Experiment and Analysis

This section will evaluate the performance of E-GVD, so we designed the following questions:

RQ1: How does E-GVD perform compared to other function-level vulnerability detection algorithms?

RQ2: Do the various components used by E-GVD have advantages for vulnerability detection performance?

Algorithm 2 The Generate Explanation Algorithm of E-GVD

Input: D_{test}: the test dataset; M_{GNN}: the trained best GNN model; Y: the predictions of D_{test};
Output: the explanation subgraph G_s;

```
for each sample D_sample in D_test do
    g(V, X, A) ← BUILD_GRAPH(D_sample)
    if y_sample =1 then
        explainer ← GNNExplainer(M_GNN)
        explanation ← explainer(X, A)
        G_s ← visualize_subgraph(explanation)
    else
        pass
    end if
end for
```

Table 1. Sample count of training/validation/testing sets.

SubDataset	Examples
Train Dataset	21854
Valid Dataset	2732
Test Dataset	2732

RQ3: Can the interpretable model of E-GVD provide reasonable explanations for the vulnerability detection results?

4.1. The Experiment Setup

Environment Configuration: This experiment was conducted on a Win11 system with an i7-1260P central processor and NVIDIA GeForce RTX 3090 GPU, based on Transformer 4.4, PyTorch 1.9, and Python 3.7.

Dataset: The GNN-based vulnerability detection model uses the CodeXGLUE [26] real benchmark dataset to detect function-level vulnerabilities. The dataset includes 27,318 hand-labeled vulnerable and non-vulnerable functions, derived from security-related commits in two widely used C programming language open-source projects (*i.e.*, QEMU and FFmpeg), with diverse functions. The dataset is divided into training/validation/testing sets at a ratio of 80%/10%/10%, and Table 1 shows the sample count for the training/validation/testing sets.

4.2. The Experimental Results and Performance Analysis

RQ1: How does E-GVD perform compared to other function-level vulnerability detection algorithms?

We trained and tested E-GVD along with BiLSTM [27], TextCNN [28], RoBERTa [29], CodeBERT [30], GraphCodeBERT [31], and Devign [32] baseline algorithms on the same dataset to evaluate the performance of E-GVD to detect vulnerabilities. We chose the best-performing parameter combination for

Table 2. Accuracy of E-GVD and other baseline algorithms (Accuracy represents the ability of the model to correctly predict samples).

MODEL	Accuracy(%)
BiLSTM	59.37
TextCNN	60.69
RoBERTa	61.05
CodeBERT	62.08
GraphCodeBERT	62.30
Devign	59.77
E-GVD	**64.35**

training and testing. At the same time, to fairly compare with the baseline algorithms, the same graph construction method and the same training protocol are used to implement Devign in the experiment.

Table 2 presents the accuracy results of E-GVD and other baseline algorithms on the CodeXGLUE dataset. The experimental results show that the accuracy of E-GVD is superior to that of all the baseline algorithms, indicating that the E-GVD model is advanced in vulnerability detection and can more accurately predict the categories or labels of vulnerabilities. Specifically, the accuracy of E-GVD can be improved by 2.05% compared to the second-best-performing GraphCodeBERT, and the maximum gain of the vulnerability detection performance over the baseline algorithms is 4.98%.

We further observe from the Table 2 that RoBERTa performs better in vulnerability detection compared to models based on BiLSTM and TextCNN. This is because RoBERTa, based on the transformer architecture, can process information in sequences in parallel, which gives RoBERTa a superior performance in downstream tasks of vulnerability detection. CodeBERT and GraphCodeBERT are improved versions of RoBERTa, the unique encoder structure also allows CodeBERT and GraphCodeBERT models to better capture the structural information of source code. From the Table 2, we can see that the accuracy of CodeBERT and GraphCodeBERT models compared to RoBERTa has increased by 1.03% and 1.25%, respectively.

Our algorithm, E-GVD, outperforms the existing baseline algorithms, with accuracy improvements of 2.27% and 2.05% compared to CodeBERT and GraphCodeBERT models, respectively. Notably, the performance of Devign on real-world datasets in our experiments can only achieve 59.77% accuracy, and under the same training protocol, the accuracy of E-GVD is improved by 4.58%. In summary, our model can effectively distinguish and classify vulnerabilities, demonstrating advanced and robust performance in vulnerability detection tasks.

RQ2: Do the various components used by E-GVD have advantages for vulnerability detection performance?

To answer **RQ2**, we employed pre-trained PL models CodeBERT and GraphCodeBERT to learn the initial graph vector representations. To investigate whether the graph-level readout layer of E-GVD has more advantages, we compared the graph-level readout layer of E-GVD to the Conv pooling of Devign under the same training protocol. We assessed the impact of residual connections on vulnerability detection by using them in experiments. Results are shown in Figures 4 and 5.

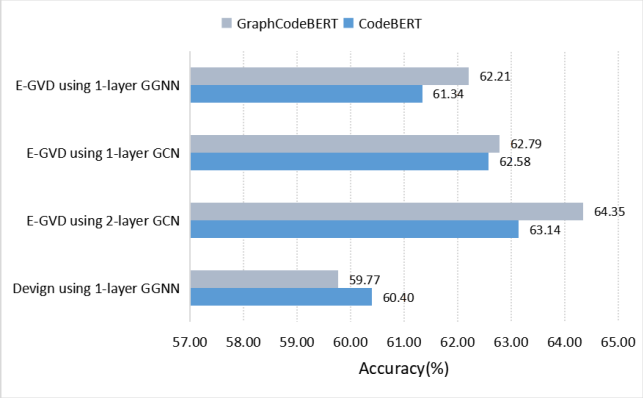

Figure 4. Accuracy of different pre-trained models and different GNN layers.

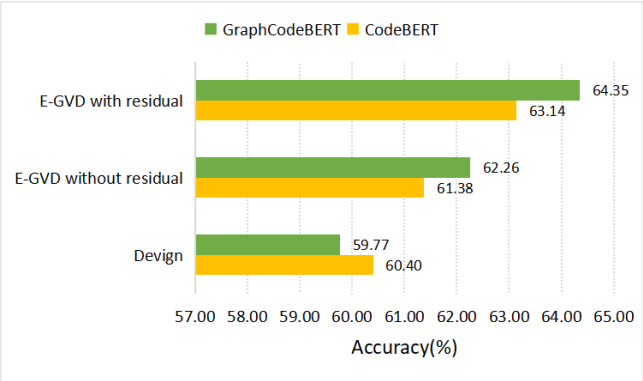

Figure 5. Accuracy of different pre-trained models and with or without residual connections.

Figure 4 shows the accuracy of E-GVD using different GNN layers under different pre-trained models. Figure 5 shows the accuracy of E-GVD with or without residual connections under different pre-trained models. To demonstrate the advantages of the graph-level readout layer in the algorithm, we also provided the accuracy of Devign under different pre-trained models.

The experimental results show that in terms of pre-trained models, our algorithm E-GVD generally performs better when using GraphCodeBERT than when using CodeBERT. This indicates that initializing node feature vectors with GraphCodeBERT in downstream vulnerability detection tasks has advantages. For the 2-layer GCN used in the algorithm, Figure 4 shows that E-GVD performs best under the 2-layer GCN, better than when using 1-layer GNN, with an accuracy gain of 1.56%. In addition, our algorithm achieves 62.21% accuracy under 1-layer GGNN, improving the accuracy of Devign by 2.44%. It shows that the graph-level readout layer of E-GVD, based on the inner product of sum pooling and max pooling, has certain advantages over the simple Conv layer. In terms of using residual connections, Figure 5 shows that residual connections can increase the accuracy of E-GVD, with a maximum accuracy gain of 2.09%. This suggests that using residual connections in our algorithm can integrate the information learned by GCN layers and more effectively classify the function source code for vulnerability.

RQ3: Can the interpretable model of E-GVD provide reasonable explanations for the vulnerability detection results?

To assess the interpretable model performance, we evaluated if important statements leading to vulnerable code snippets can be inferred through explanatory subgraphs. For each input instance, an induced subgraph is returned, which helps with explanation results determine important nodes for predicted labels, inferring crucial statements for vulnerable code snippets. We analyzed the following instance with predicted labels of 1 to evaluate the performance of the result-based interpretable model.

```
1  int VerifyAdmin(char *password) {
2      if (strcmp(password, "Mew!")) {
3          printf("Incorrect Password!\n");
4          return(0)
5      }
6      printf("Entering Diagnostic Mode...\n");
7      return(1);
8  }
```

Figure 6. Vulnerable example with the predicted label 1, the vulnerable code lines identified by E-GVD are lines 1 and 2.

Figure 6 displays code example labeled as vulnerable in the test set. This example illustrates a hard-coded credential use vulnerability from CWE-798. In the example shown in Figure 6, the defect line is at *strcmp (password, "Mew!")* in line 2, where an attacker can use the password to access the backend diagnostic mode and extract the fixed password, posing an even greater threat if distributed in binary form.

E-GVD infers the important statements leading to vulnerable code by generating induced subgraphs and interpreting the subgraph probabilities in the results. If the probability of a subgraph surpasses a threshold, it is deemed influential on the prediction. For ease of explanation, we provided an example of an induced subgraph. Figure 7 shows the induced subgraph for the example. According to our manually derived results, the code lines potentially containing important statements are marked in red in the examples in Figure 6. The vulnerable code lines identified by E-GVD are lines 3, 6, 10, and 19. Based on the instance, it is not difficult to find that E-GVD can provide fine-grained interpretation results for function-level vulnerability detection, demonstrating the feasibility of result-based interpretable model and the applicability of graph interpretability.

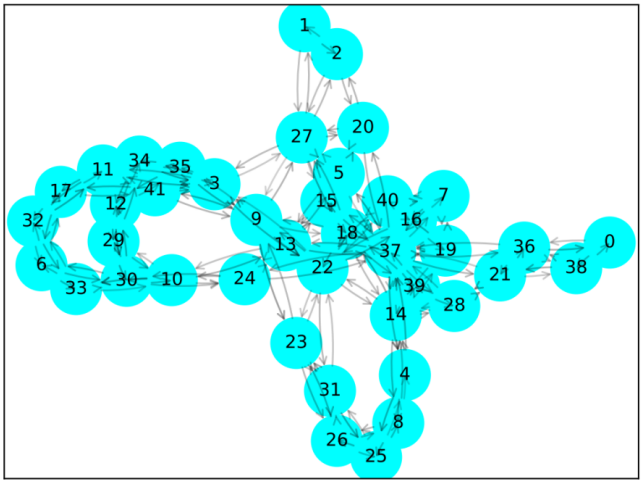

Figure 7. Induced subgraph of Code Example.

5. Conclusion and Future Prospect

This paper introduces E-GVD, a more sophisticated and interpretable source code vulnerability detection algorithm based on GNN. E-GVD initially transforms source code into a graph structure effectively and uses PL pre-trained models to initialize node features, preparing the input for the DL model. It then updates node features using a dual-layer GCN with residual connections and generates global graph embeddings by combining sum pooling and max pooling techniques, enabling the detection of vulnerabilities in source code. Additionally, E-GVD employs GNNExplainer to interpret the predictions of graph classification, revealing the locations of vulnerabilities. We achieved optimal vulnerability detection performance and verified the superior performance and interpretability of E-GVD compared to other baseline methods. Future work will concentrate on enhancing the performance of DL-based vulnerability detection models. We will also explore the dependencies of vulnerable line information in interpretative results, aiming to improve model interpretability comprehensively.

Acknowledgement. This work was supported by the National Natural Science Foundation of China (No. 61373131, 62071240), PAPD and CICAEET funds.

References

[1] Shu, J., Jia, X., Yang, K. and Wang, H. (2018) Privacy-preserving task recommendation services for crowdsourcing. *IEEE Transactions on Services Computing* **14**(1): 235–247.

[2] Patil, D.R. and Pattewar, T.M. (2022) Majority voting and feature selection based network intrusion detection system. *EAI Endorsed Transactions on Scalable Information Systems* **9**(6): e6–e6.

[3] Ge, Y.F., Wang, H., Bertino, E., Zhan, Z.H., Cao, J., Zhang, Y. and Zhang, J. (2023) Evolutionary dynamic database partitioning optimization for privacy and utility. *IEEE Transactions on Dependable and Secure Computing* .

[4] Venkateswaran, N. and Prabaharan, S.P. (2022) An efficient neuro deep learning intrusion detection system for mobile adhoc networks. *EAI Endorsed Transactions on Scalable Information Systems* **9**(6): e7–e7.

[5] Jordan, M.I. and Mitchell, T.M. (2015) Machine learning: Trends, perspectives, and prospects. *Science* **349**(6245): 255–260.

[6] LeCun, Y., Bengio, Y. and Hinton, G. (2015) Deep learning. *nature* **521**(7553): 436–444.

[7] Qu, Z., Tang, Y., Muhammad, G. and Tiwari, P. (2023) Privacy protection in intelligent vehicle networking: A novel federated learning algorithm based on information fusion. *Information Fusion* **98**: 101824.

[8] Tawhid, M.N.A., Siuly, S., Wang, K. and Wang, H. (2023) Automatic and efficient framework for identifying multiple neurological disorders from eeg signals. *IEEE Transactions on Technology and Society* **4**(1): 76–86.

[9] Singh, R., Subramani, S., Du, J., Zhang, Y., Wang, H., Miao, Y. and Ahmed, K. (2023) Antisocial behavior identification from twitter feeds using traditional machine learning algorithms and deep learning. *EAI Endorsed Transactions on Scalable Information Systems* **10**(4): e17–e17.

[10] Qu, Z., Liu, X. and Sun, L. (2022) Learnable antinoise-receiver algorithm based on a quantum feedforward neural network in optical quantum communication. *Physical Review A* **105**(5): 052427.

[11] Liu, F., Zhou, X., Cao, J., Wang, Z., Wang, T., Wang, H. and Zhang, Y. (2020) Anomaly detection in quasi-periodic time series based on automatic data segmentation and attentional lstm-cnn. *IEEE Transactions on Knowledge and Data Engineering* **34**(6): 2626–2640.

[12] Wang, H., Ye, G., Tang, Z., Tan, S.H., Huang, S., Fang, D., Feng, Y. *et al.* (2020) Combining graph-based learning with automated data collection for code vulnerability detection. *IEEE Transactions on Information Forensics and Security* **16**: 1943–1958.

[13] Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y. and Chen, Z. (2021) Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* **19**(4): 2244–2258.

[14] Rathore, H., Sharma, S.C., Sahay, S.K. and Sewak, M. (2022) Are malware detection classifiers adversarially vulnerable to actor-critic based evasion attacks? *EAI Endorsed Transactions on Scalable Information Systems* **10**(1).

[15] Jiang, F., Cao, Y., Xiao, J., Yi, H., Lei, G., Liu, M., Deng, S. *et al.* (2022) Vddl: A deep learning-based vulnerability detection model for smart contracts. In *International Conference on Machine Learning for Cyber Security* (Springer): 72–86.

[16] Zhu, Z. and Wang, S. (2023) Odet: Optimized deep elm-based transfer learning for breast cancer explainable detection. *EAI Endorsed Transactions on Scalable Information Systems* **10**(2): e4–e4.

[17] Kumar, S.B. and Pande, S.D. (2024) Explainable neural network analysis on movie success prediction. *EAI Endorsed Transactions on Scalable Information Systems* .

[18] Geng, Y. (2021) Self-organizing incremental and graph convolution neural network for english implicit discourse relation recognition. *EAI Endorsed Transactions on Scalable Information Systems* **9**(36).

[19] Qu, Z., Liu, X. and Zheng, M. (2022) Temporal-spatial quantum graph convolutional neural network based on schrödinger approach for traffic congestion prediction. *IEEE Transactions on Intelligent Transportation Systems* .

[20] Ni, M., Song, Y., Wang, G., Feng, L., Li, Y., Yan, L., Li, D. *et al.* (2023) Mied: An improved graph neural network for node embedding in heterogeneous graphs. *EAI Endorsed Transactions on Scalable Information Systems* **10**(6).

[21] Cui, L., Hao, Z., Jiao, Y., Fei, H. and Yun, X. (2020) Vuldetector: Detecting vulnerabilities using weighted feature graph comparison. *IEEE Transactions on Information Forensics and Security* **16**: 2004–2017.

[22] Cao, S., Sun, X., Bo, L., Wei, Y. and Li, B. (2021) Bgnn4vd: Constructing bidirectional graph neural-network for vulnerability detection. *Information and Software Technology* **136**: 106576.

[23] Yin, J., Tang, M., Cao, J., You, M., Wang, H. and Alazab, M. (2022) Knowledge-driven cybersecurity intelligence: Software vulnerability coexploitation behavior discovery. *IEEE transactions on industrial informatics* **19**(4): 5593–5601.

[24] Wang, S., Huang, C., Yu, D. and Chen, X. (2023) Vulgrab: Graph-embedding-based code vulnerability detection with bi-directional gated graph neural network. *Software: Practice and Experience* .

[25] Ying, Z., Bourgeois, D., You, J., Zitnik, M. and Leskovec, J. (2019) Gnnexplainer: Generating explanations for graph neural networks. *Advances in neural information processing systems* **32**.

[26] Microsoft, Codexglue: Defect-detection, Available online: https://github.com/microsoft/CodeXGLUE/tree/main/Code-Code/Defect-detection (accessed on 16 January 2024).

[27] Huang, Z., Xu, W. and Yu, K. (2015) Bidirectional lstm-crf models for sequence tagging. *arXiv preprint arXiv:1508.01991* .

[28] Kim, Y. (2014) Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882* .

[29] Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O. *et al.* (2019) Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* .

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

[31] Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L. *et al.* (2020) Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366* .

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