Code Vulnerability Detection Based on Graph Neural Network

Yege Yang
School of Computer Science and Engineering
Xi'an Technological University
Xi'an, 710021, China
E-mail: 664730726@qq.com

Guiping Li
School of Computer Science and Engineering
Xi'an Technological University
Xi'an, 710021, China
E-mail: 15693685@qq.com

***Abstract*—Deep learning has emerged as a vital approach for identifying and addressing vulnerabilities in software systems. A key challenge in this process lies in effectively representing code and leveraging AI techniques to capture and interpret its semantics and other intrinsic information. This paper employs bidirectional slicing techniques to extract code slices containing control and data dependencies from program dependency graphs, targeting key points of different vulnerabilities. To represent the node features within the slices, code tokens are mapped to integers and transformed into fixed-length vectors, leveraging Word2vec and BERT models to embed the code nodes and extract structural graph features. The embedded feature matrix is then fed into a Gated Graph Neural Network (GGNN), which aggregates information from nodes and their neighbors to enhance long-term memory of graph-structured data. By iterating through several time steps within GRU units, the final node features are generated. Additionally, edge relationships are used to propagate and aggregate information, further improving the accuracy of vulnerability detection. Experimental results demonstrate that the proposed model achieves an F1-score of 93.25% on the BigVul dataset, showcasing strong detection performance.**

Keywords-Software Security; Deep Learning; Program Analysis; Code Vulnerability Detection; Graph Neural Network

I. INTRODUCTION

A. Background Introduction

Software vulnerabilities are an important cause of network attacks and data leakage, posing a serious threat to software security. Despite efforts to pursue secure programming, vulnerabilities are still widespread due to the increasing complexity of software and the continuous expansion of the Internet. According to the vulnerability data released by China National Vulnerability Database of Information Security (CNNVD) in 2022, there were 24801 new vulnerabilities added in 2022, an increase of 19.28% compared to 2021. The growth rate has markedly increased, reaching record levels and sustaining its upward momentum. The percentage of extremely high-risk vulnerabilities is steadily rising [1]. Vulnerabilities in software, once exploited by malicious attackers, may cause serious consequences such as system paralysis and personal privacy data leakage, posing a ransomware risk to the company or posing a threat to public security. Therefore, ensuring the reliability of software has become a current focus of attention to protect internet users from cyber attackers. Code vulnerabilities often arise from minor errors and can rapidly proliferate due to the extensive use of open-source software and code reuse. Early detection of vulnerabilities to protect software security has become crucial. Detecting and fixing software vulnerabilities is a complex task because of the wide variety of vulnerabilities and their increasing frequency.

In the last dozen years, machine learning(ML) has made significant progress, particularly in areas such as deep learning (DL) algorithms [2], natural language processing techniques [3], and other data-driven approaches, which have proven highly effective in detecting software vulnerabilities. ML/DL models excel at identifying subtle patterns and correlations within large datasets, a critical capability for vulnerability detection, as vulnerabilities often stem from intricate code features and dependencies. These models handle diverse data types and formats, including source code [4-11], textual information [12], and numerical features like submission characteristics [13]. By processing and analyzing these diverse

data representations, ML/DL models enable effective vulnerability detection. This adaptability allows researchers to leverage multiple data modalities for a more comprehensive approach to detecting vulnerabilities.

II. Related Works

DL-based of vulnerability detection methods are currently the forefront of vulnerability detection, which can effectively narrow down the scope of code auditing, avoid expert defined features, and achieve the goal of automatic vulnerability detection [14]. The existing DL-based modeling techniques currently fall into two categories: sequence-based models and graph-based models [32].

A. Sequence Based Model

In the sequence-based modeling approach, the code is considered as a sequence of tokens, which are slices of the code, including methods based on token sequences of function call, source codes, intermediate code, and assembly code, respectively.

Wu et al. [15] employed Convolutional Neural Networks (CNN), Long Short Term Memory (LSTM), and CNN-LSTM architectures to model function call sequences from binary programs, generating numerical vector representations of the sequence data through Keras and enhancing vulnerability detection by analyzing function call patterns. Russell et al. [16] reduced token vocabulary size using a custom C/C++ lexer and applied CNN and Random Forest models to detect vulnerabilities at the function level, focusing on the sequential characteristics of source code. Yan et al. [17] introduced the HAN-BSVD model, which captures local sequential features using Bidirectional Gated Recurrent Unit (Bi-GRU) and Text-CNN, with word attention modules highlighting critical sequence regions for binary vulnerability detection. Li et al. [18] developed VulDeeLocator, which utilizes the Bidirectional Recurrent Neural Network (BRNN-vdl) to integrate sequence-based intermediate code analysis, combining data and control flow dependencies to improve vulnerability detection accuracy. Tian et al. [9] proposed BVDetector, utilizing Binary Gated Recurrent Unit (BGRU) and program slicing to analyze sequences of control and data flow, specifically detecting vulnerabilities related to library/API function call sequences in binary programs.

Based on the aforementioned methods, sequence-based vulnerability detection techniques are primarily employed for binary code analysis. These methods effectively capture the program's execution order and logical flow while minimizing the complexity of the analysis. Sequence models demonstrate strong adaptability in processing binary code, particularly in scenarios where source code is unavailable, making them a powerful tool for vulnerability detection.

B. Graph Based Model

The graph-based modeling approach treats the code as a graph and merges different syntactic and semantic dependencies, which can be used for vulnerability prediction using different types of syntactic and semantic graphs in two main ways, transforming the graph structure into a sequence or modeling the graph structure directly.

The Sequence Graph Hybrid Model utilizes both the graph structure and linear sequence in the program. The information extracted from the graph structure is usually transformed into linear sequences, which can be directly input into machine learning models or deep learning models (such as LSTM, GRU) for analysis and prediction. This transformation makes complex graph structure information easier to be processed by traditional sequence models. VulDeePecker [19] extracted code slices from a Data Dependency Graph (DDG) by capturing data flow dependencies between variables and using these slices as input sequences for an RNN and Bi-LSTM model. μVulDeePecker [20] enhanced the detection process by incorporating both data and control dependencies through System Dependency Graphs (SDG), using forward and backward slicing techniques to capture local and global semantic information. These code slices were processed with a Bi-LSTM model to improve vulnerability type identification. SySeVR [21] combined both Bi-LSTM and Bi-GRU models to analyze slices representing data and control flow dependencies, further improving vulnerability

detection by capturing the sequential relationships in these flows. Compared with modeling by converting graph structures into sequences, directly using graph neural networks to model graph structures can more fully learn the dependency relationships between nodes, program structure, and semantic information in graph based code representation. This method can provide more accurate feature vector representations for source code vulnerability detection models. In the method of directly modeling graph structures, one approach is to comprehensively represent the syntax and semantic information related to vulnerabilities at different levels of abstraction by integrating multiple intermediate representations of code, without using slicing techniques. Another approach is to use program slicing techniques to remove information unrelated to vulnerabilities from mixed code representations, in order to improve the accuracy and efficiency of vulnerability detection.

Zhou et al. [22] proposed Devign, which encoded source code into a Combined Program Dependency Graph (CPG), integrating the Abstract Syntax Tree (AST), Control Flow Graph (CFG), and Data Flow Graph (DFG). Natural Code Sequence (NCS) edges were added to preserve code order, and a Gated Graph Recurrent Network (GGRN) with convolutional layers was used for graph-level classification. Wang et al. [23] proposed FUNDED, which improved vulnerability prediction by automatically collecting high-quality samples through confidence prediction (CP). They combined AST and Program Control and Dependency Graphs (PCDG), extracting nine code relationships, and used GGNN with GRU-based models to capture higher-order neighborhood information for detection. Zheng et al. [24] proposed VulSPG, which merged control, data, and function call dependencies into Sliced Program Graphs (SPG) and utilized a Relational Graph Convolutional Network (R-GCN) for vulnerability detection, further enhanced by a triple attention mechanism. Cheng et al. [30] proposed DeepWukong, which generated program slices from Program Dependency Graphs (PDG) and employed GCN and k-dimensional GNN (k-GNN) models to process these slices. Cao et al. [25] introduced MVD, which incorporated function call information into PDGs and focused on detecting memory-related vulnerabilities. The method embedded the program slices using Doc2Vec and employed Flow Sensitive Graph Neural Networks (FS-GNN) to enhance vulnerability detection. Zou et al. [26] utilized PDG-based vulnerability slicing to capture vulnerabilities around pointers and sensitive APIs, employing GGNN models to learn and interpret vulnerability features.

The graph-based modeling approach treats code as graphics and combines different syntax and semantic dependencies. Graph-based representation methods can effectively preserve complex semantic information such as the logical structure and dependency relationships of the code.

III. Feature Extraction

A. Code Standardization

Semantic irrelevant information in source code, such as comments, complex variable names, and function names, can interfere with the training and prediction accuracy of deep learning models. To minimize the impact of this irrelevant information, this study first normalizes the source code. Comments, spaces, tabs, and line breaks are removed while ensuring the corresponding line numbers remain unchanged. Function names are replaced with a standardized identifier (e.g., FUNC1) under specific conditions: they are not keywords (e.g., boolean, const), preprocessor directives (e.g., #define, #include), library functions (e.g., snprintf, sleep), or the main function, and must be followed by an opening parenthesis indicating a function call. For variable names, candidates are identified by excluding function parameters, function names that are not keywords, preprocessor directives, library functions, or specific terms like 'argc' (representing the number of parameters passed to the main function) and 'argv' (representing the sequence or pointer of parameters passed to the main function). Additionally, the variable name is only standardized (e.g., VAR1) if it is not immediately followed by an opening parenthesis. This normalization process is illustrated in Figure1.

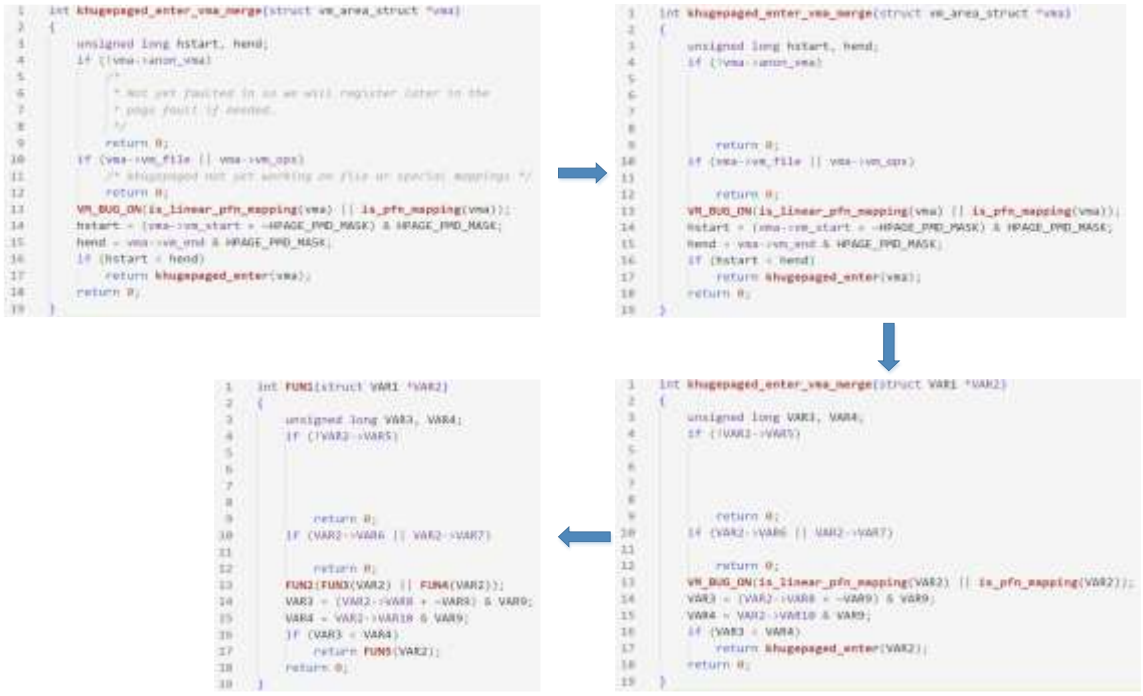

Figure 1. The Process of Code Standardization

B. Construction of Code Representation Graph

The code representation diagram is an effective method for visualizing code by clearly conveying its semantic, syntactic, and structural information. Among these, the PDG is a particularly expressive data structure that connects AST nodes through DDG and CDG. Using the Joern tool [27], binary (bin) files are generated, followed by the extraction of the CPG through the graph-for-funcs.sc script. This process generates both the AST and CFG, storing the resulting data in a JSON file.

The JSON format is chosen because the graph nodes are based on fine-grained AST elements rather than entire code statements. However, when analyzing source code, it is necessary to map these nodes to specific lines of code (for instance, diff files are line-based). By executing the command 'cpg.runScript("graph-for-funcs.sc").to String() |>.json', the graph is output as a JSON file, facilitating further analysis. The PDG, also generated by Joern, integrates AST and CFG structures, providing a more comprehensive representation of code features for vulnerability detection. The process of generating the PDG is illustrated in Figure2.

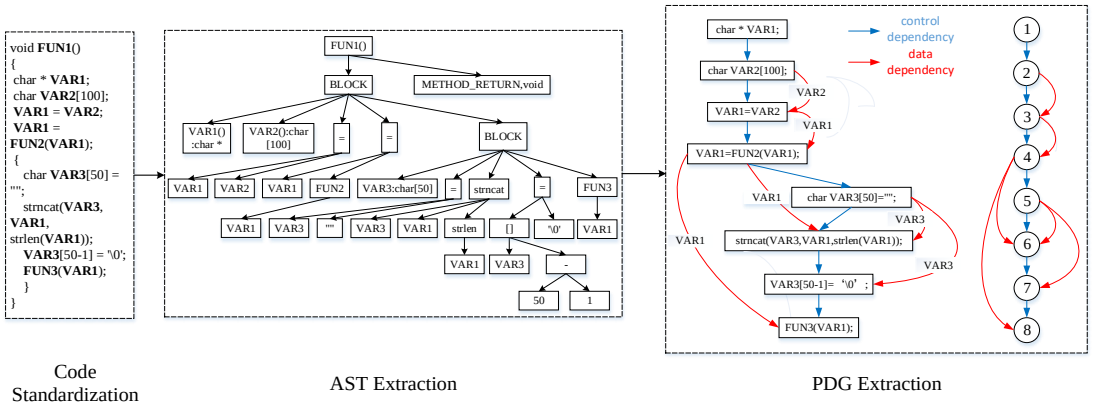

Figure 2. The Process of Generating the PDG

C. Slice of Code Representation Graph

A significant challenge in function-level vulnerability detection is the presence of a large number of vulnerability-independent noise statements, which can hinder the model's ability to effectively learn vulnerability features and ultimately degrade detection performance. Furthermore, the complexity of real-world software often results in program dependency graphs with numerous nodes and edges, leading to increased time and memory consumption during the training process. To address these challenges, this paper adopts a slice-level vulnerability detection approach, aiming to eliminate irrelevant information and enhance both detection accuracy and interpretability while minimizing resource overhead. By comparing vulnerable code with corresponding patched code, the study identifies four potential vulnerability-prone areas: pointers, arrays, expression operations, and sensitive APIs. These areas, referred to as "vulnerability focus points," represent key locations where vulnerabilities are most likely to occur.

1) Pointer Operations

Pointer operations can lead to dynamic memory errors, such as memory leaks, double-free errors, and null pointer dereferencing. To mitigate confusion associated with the use of the asterisk (*) in pointer declarations and avoid errors, this paper identifies specific node types in the code representation.

The node type 'Identifier' represents program entities such as variable names, functions, or classes. For instance, in the expression 'int* ptr', the term 'ptr' is classified as an identifier, and its data type is determined to be a pointer. Similarly, 'MethodParameterIn' nodes represent input parameters passed to methods or functions and are also identified as pointers when applicable. For example, in the function signature 'void foo(int* ptr)', 'ptr' is a method input parameter of pointer type. Additionally, 'FieldIdentifier' nodes are typically used to denote fields or member variables in a class or structure. Consider the following example:

```
struct MyStruct {
    int* ptr; // Pointer to an integer
};
```

In this case, the node representing 'ptr' is categorized as a pointer field. After identifying such node types, the system inspects the node and its type to check for the presence of the asterisk (*).

2) Array operation

Array operations often involve out-of-bounds reading or writing, with out-of-bounds writes potentially leading to memory overflow vulnerabilities. Specifically, selecting nodes classified as 'indirectIndexAccess' represents indirect index access, typically used to describe scenarios where arrays or collection elements are accessed indirectly through variables or expressions. For example, the following code illustrates a possible case of out-of-bounds access:

```
int arr [5] = {1, 2, 3, 4, 5};
int index = 2; int value = arr[index];
```

In this example, 'index' denotes an indirect access to the array. By analyzing such nodes in the code, the presence of array indexing symbols '[]' can be detected, allowing further checks for potential out-of-bounds access.

3) Expression Operations

Expression operations such as addition, subtraction, and multiplication can result in integer overflow. For instance, when performing arithmetic on 'int' types, if the result exceeds the representable range of the data type, overflow occurs. Division operations, on the other hand, may lead to division by zero errors.

The specific approach involves selecting nodes of type 'assignment', which represent assignment operations. If the node contains an equal sign ('='), the expression on the right side of the equal sign is extracted. Regular expressions are then used to match expressions that include arithmetic operations such as addition, subtraction, multiplication, and division (e.g., '((?:_|[A-Za-z])\w*(?:\s(?:\+|\-|*|\/)\s(?:_|[A-Za-z])\w*)+)' for strings like "a + b" or "x - y * z", where both operands start with a letter or underscore, followed by any number of letters, digits, or underscores). In cases where no equal sign is present, regular expressions are used to match division operations (e.g., '(?:\s\/\s(?:_|[A-Za-z])\w*\s)'), as division in

integer operations may trigger overflow or division-by-zero errors.

4) Sensitive API Function Operations

Improper use of functions that handle sensitive data can lead to various security vulnerabilities such as memory leaks, pointer errors, integer overflows, and buffer overflows. Examples include file handling functions (e.g., 'ifstream. open', 'ifstream.read*'), memory and pointer operations (e.g., 'xcalloc', 'IsBadReadPtr'), date and time functions (e.g., '_wctime_s', '_ctime64_s'), cryptographic functions (e.g., 'CC_SHA224_Update'), system calls and OS functions (e.g., 'chown', 'RegGetValue'), network communication functions (e.g., 'recvfrom', 'recv') and user input/output functions (e.g., 'getc', 'cin').

Starting from the four identified vulnerability focus points, the process involves traversing forward and backward along data dependency and control dependency edges, while preserving the original line numbers from the source code. These line numbers are then compared with those in the 'func_label.pkl' file. If the line numbers match, the slice is identified as containing a vulnerability and labeled as '1_'; otherwise, the slice is labeled as '0_', indicating no vulnerability in the slice.

As illustrated in Figure 3, slices 1 through 4 are derived using VAR1, VAR2, VAR3, and strncat as the slicing base points. Starting from these key points, forward and backward traversals along control and data dependency edges are conducted, recording the involved nodes and edges until no new nodes or edges emerge. The resulting subgraph of the program dependency graph, obtained through these steps, constitutes a program slice. Since the slice retains only nodes and edges that are dependent on the vulnerability focus points, it preserves the structural information of the original source code while eliminating irrelevant details.

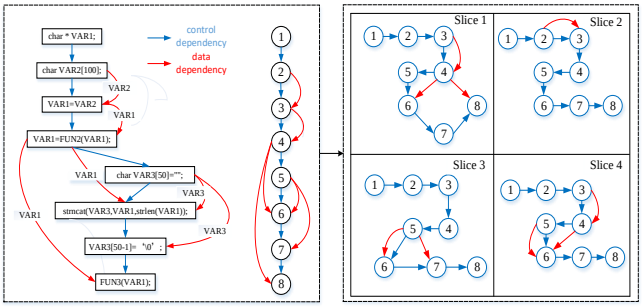

Figure 3. The Process of Slicing PDG

D. Extract Slice Features

Since the extracted code slices in this study are in an abstract graph format, they cannot be directly input into graph neural network-based vulnerability detection models. Therefore, key features from the graphs must be extracted to generate the corresponding feature vectors. The graph slices contain two types of features: code features within the nodes (referred to as node features) and the structural features of the graph.

Traditional Word2Vec models, which usually rely on local contextual information, are unable to capture long-range semantic relationships and global context. In contrast, BERT models excel in this area. BERT, through its bidirectional encoder pretraining, can deeply understand long-range dependencies within the context.

For node features, this study adopts an embedding representation approach, mapping tokens to integers and converting them into fixed-length vectors using a distributed representation technique. Specifically, the code within each node is treated as a sentence, tokenized into tokens, and embedded into a fixed-length vector. Node embeddings are achieved by combining the Word2Vec and BERT models.

Specifically, this study trains a pre-trained Word2Vec model using the token lists from all code slices. The pre-trained model is then applied to embed all nodes into vectors. The preprocessed slices are input into the Word2Vec model, which generates an $m \times n$ feature matrix Mf, where m represents the number of nodes in the slice, and n represents the dimension of the embedding vectors, which is set to 100 in this study. As shown in Figure 3, there are eight nodes in the graph, so the node feature matrix Mf has dimensions of 8×100.

For the BERT model, the pre-trained BERT model is utilized to embed the code within each node. The process involves using the 'BertTokenizer' to encode the tokens, followed by the 'BertModel' to generate context-sensitive dynamic word embeddings. Each node's code is first transformed into the input format accepted by BERT, which then captures the long-range dependencies and contextual information within the code snippets. The output of the BERT model for each node is a feature matrix with a shape of m×n, where n is 768 dimensions. As a result, the node feature matrix Mh in Figure 3 has a dimension of 8×768.

Finally, the word embeddings generated by Word2Vec and BERT are concatenated to form a combined vector of 8×868 dimensions, denoted as Mi.

For the graph structural features, this paper performs embedding representations of the edge relationships within the graph. Each edge can be represented as a triplet (source node, target node, edge type). Both the source and target nodes can be directly extracted from the program dependency graph, while the edge types are categorized into data dependency edges and control dependency edges. Taking Slice 1 as an example, it contains 9 edges, including 2 data dependency edges and 7 control dependency edges. Red edges represent data dependency, blue edges represent control dependency, and purple edges indicate both data and control dependencies. The output matrix AS represents the graph structure feature matrix, and its generation process is illustrated in Figure 4.

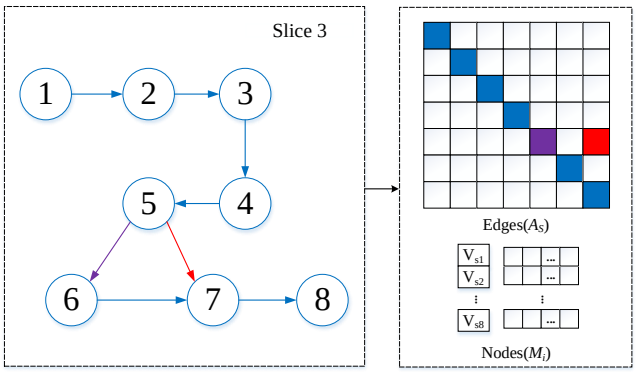

Figure 4. The Process of Extracting Features from the Slice Graph

E. Vulnerability Detection Model Based on GGNN

1) Figure Neural Network Module

This study transforms the source code into a graph structure that incorporates data dependencies and control dependencies. GNN help to further aggregate and propagate information updates, capturing both the structural and semantic information of the graph more effectively. Among GNN models, the GGNN is chosen for this work due to its enhanced ability to handle complex semantic and graph structure data by improving the network's long-term memory capacity. The principle behind GGNN involves aggregating information from a node and its neighboring nodes, then feeding the aggregated information and the current node into a GRU unit to obtain the updated state of the node. This process is repeated over several time steps, resulting in the final node representation for all nodes in the graph. As shown in the graph neural network module in Figure 5, after inputting the graph features gi(Mi,AS), GGNN embeds each node and its neighborhood into a new representation, transforming it into a slice feature matrix Mi' with dimensions m×n', where n' represents the final size of the slice feature. In this study, n' is set to 200, making the feature matrix Mi' of size 8×200.

For each node v_u in the graph, its initial feature vector $h_u^{(1)} = [m_u^T, 0]^T$ is constructed by concatenating the feature vector v_u with a zero padding. Setting T as the total number of time steps for neighborhood aggregation, each node communicates with its neighbors along the edges it depends on at each time step $t \le T$. The update formula is given by:

$$a_u^{(t)} = A_u^T (W_u [h_1^{(t)T}, ..., h_m^{(t)T}] + b) \quad (1)$$

where W_u represents trainable parameters, b is a bias term, and A_u^T denotes the adjacency matrix for the neighborhood of node v_u corresponding to edge type A_s. $a_u^{(t)}$ encapsulates the aggregated

information from node v_u 's neighbors through their interactions along the edges. This information is then combined with the node's current state through the aggregation function AGG, leading to the updated node state:

$$h_u^{(t+1)} = GRU(h_u^{(t)}, AGG(\{a_u^{(t)}\})) \quad (2)$$

This process continues iteratively, allowing the node's feature vector to evolve over time by incorporating information from neighboring nodes, until the final representation is obtained.

Vulnerability classification module

To perform graph-level vulnerability classification tasks, a feature set relevant to vulnerability characteristics is selected. Previous work [28] proposed using a classification pooling layer (SortPooling) after the graph convolutional layer to sort the output features, enabling the use of traditional neural networks for training and extracting useful features from the embedded slice vectors. In this paper, node features are first learned through GGNN layers, followed by one-dimensional convolutional and fully connected layers to capture features relevant to the graph classification task, enabling more effective classification. Specifically, skip connections are employed in the graph convolution and feature extraction phases, which help retain the details and semantic information from the original input data. This approach facilitates easier information propagation to deeper layers and prevents information loss. The process is expressed as follows.

$$\begin{aligned} H_3 = \mathrm{Relu}(&Conv3(\mathrm{Re\,lu}(Conv2(\mathrm{Re\,lu}(Conv1(X))) \\ &+ \mathrm{Re\,lu}(Conv1(X))))) \end{aligned} \quad (3)$$

The input graph structure data X is processed through the GNN layer to obtain node features Mi'. The first convolutional layer, Conv1, is used to extract initial node features, followed by Conv2, the second convolutional layer, which further captures higher-level node features. Conv3 represents the third convolutional layer and is responsible for extracting the final node features.

H3 denotes the node features after passing through all three convolutional layers. Finally, the obtained node features H3 are concatenated with the original input node features Mi, resulting in the feature matrix Mi'. This process is described as follows:

$$C_i = Contat(DeBatchify(H_3), DeBatchify(M_i)) \quad (4)$$

The function of DeBatchify is to restore node feature vectors into independent graph feature vectors when processing batch data, ensuring that each graph's data can be individually handled and analyzed.

In this paper, the classification pooling layer τ(M) is defined as follows:

$$\tau(M) = MaxPool(Relu(BN(Conv(M)))) \quad (5)$$

Here, Conv denotes the convolutional layer, BN represents the BatchNorm layer, ReLU indicates the activation function, MaxPool refers to the max-pooling layer, and M denotes a feature matrix. In this work, the node feature matrix Mi is concatenated with the corresponding slice feature matrix Mi1' to form a new matrix Mi1". τ-classification pooling operations are then applied separately to Mi1' and Mi1", resulting in outputs Y1 and Y2. These outputs are subsequently passed through fully connected layers with an output dimension of 2. The formulation for the weighted average and final output is presented as follows:

$$P = Sigmoid(Avg(MLPY(Y1) \cdot (MLPY(Y2))) \quad (6)$$

The fully connected layer performs a linear transformation on the feature matrix, followed by element-wise multiplication and averaging. The Sigmoid function is then applied to produce the binary classification probability output. P represents the binary classification result, consisting of two dimensions: the first dimension indicates the probability of no vulnerability, while the second dimension represents the probability of a vulnerability. The model outputs the final classification result by selecting the higher probability between the two. The model is trained using the CrossEntropyLoss function to correct misclassifications, along with the Adam optimization algorithm [29] with a learning rate of

0.0001 and a weight decay of 0.001 to update the parameters and b in the graph neural network module, as specified in Equation (1). After training, the model is used to determine whether new code slices contain vulnerabilities. The architecture of the vulnerability detection model is illustrated in Figure 5.

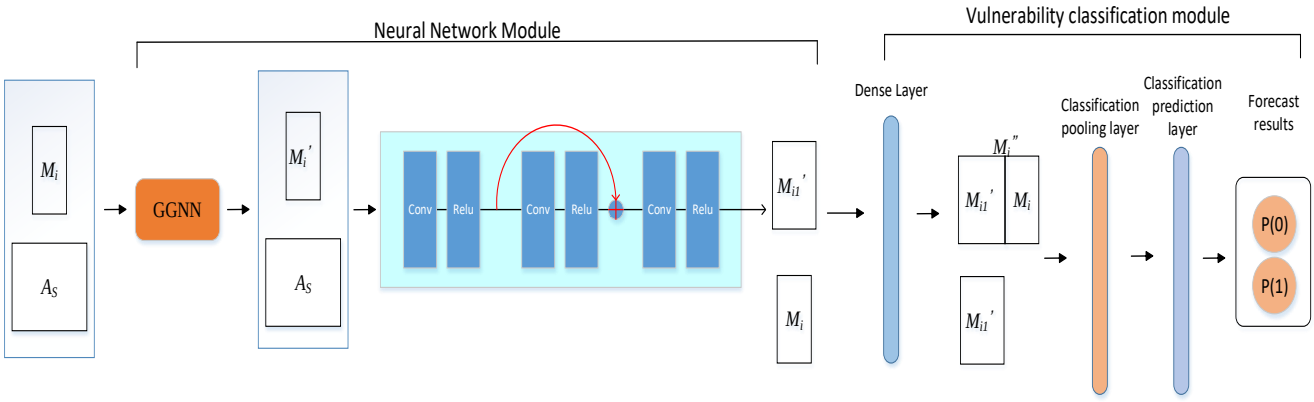

Figure 5. Vulnerability detection model architecture

IV. Experiment

A. Simulation parameter settings

This experiment is based on Python 3.8 to simulate and analyze the proposed algorithm, using the Pytorch 1.13 deep neural network framework and CUDA 11.6. The extraction of graphs in Joern is done using JDK17.0.11. Table I details the specific parameters used in the model during training.

TABLE I. TABLE TYPE STYLES

Parameter	Value
Loss Function	CrossEntropyLoss
Optimization Algorithm	Adam
Learning Rate	0.0001
Weight Decay	0.001
Batch Size	16
Training Epochs	500
Max Steps	10000

B. Training Results of the Proposed Network

This study utilizes the publicly available BigVul dataset, which includes 348 CVE (Common Vulnerabilities and Exposures) entries, consisting of 11,834 vulnerable functions and 253,096 non-vulnerable functions. From this dataset, 9,653 vulnerable functions and their corresponding 9,653 patched functions were selected for analysis. A total of 19,621 vulnerable code slices and 324,690 non-vulnerable slices were extracted. The difflib library was employed to generate the differential content between each vulnerable file and its respective patch file. Additionally, the specific lines of code containing vulnerabilities in each vulnerable file were recorded in the test_label.pkl file.

The Proposed Network achieved the highest test accuracy of 93.06%, precision of 92.22%, recall of 94.3%, and F1 score of 93.25%, all while maintaining stable training and test losses of 20% and 12.5%, respectively. These results indicate that the model is highly effective at accurately identifying vulnerabilities while maintaining a good balance between precision and recall. This suggests robust generalization and reliability in detecting both vulnerable and non-vulnerable code slices. Figure 6 shows the results of the model training in this article.

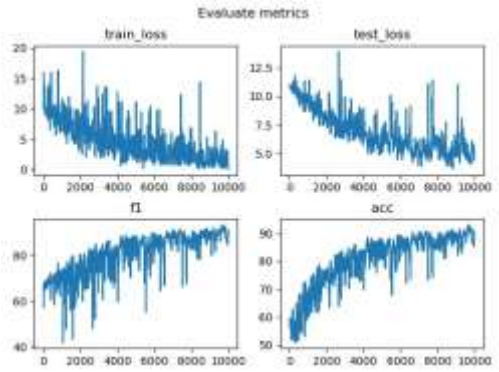

Figure 6. Results of the Model Training in the Proposed Network

C. Performance Comparison with Various Networks

Figure 7 presents a comparison of detection results using the proposed network under two approaches: one utilizing only Word2Vec and the other combining Word2Vec and BERT. The results for the model using only Word2Vec are as follows: Test Accuracy: 88.26, Precision: 87.28, Recall: 88.95, and F1 Score: 88.11. In contrast, the model combining Word2Vec and BERT achieved the following results: Test Accuracy: 93.06, Precision: 92.22, Recall: 94.30, and F1 Score: 93.25. These results demonstrate that integrating BERT significantly improves all evaluation metrics, highlighting its superior ability to capture contextual information and effectively extract deep semantic features.

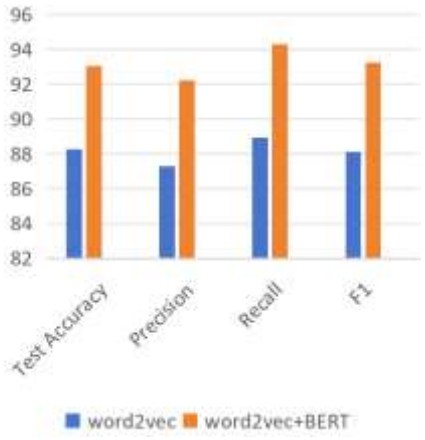

Figure 7. Ablation experiment

Through ablation experiments, we found that V1 (removing residual connections) achieved 91.13% accuracy, 88.65% precision, 94.65% recall, and 91.55% F1 score, but showed high training loss (100%). V2 (removing batch normalization) achieved 90.95% accuracy, 88.06% precision, 95.07% recall, and 91.43% F1 score, with training and test losses of 100% and 40%. V3 (replacing GGNN) performed the worst, with 71.99% accuracy, 72.7% precision, 71.83% recall, and 72.26% F1 score, alongside training and test losses of 20% and 50%. V_GIN, based on GIN layers, performed better than V1, V2, and V3, achieving 92.09% accuracy, 89.41% precision, 95.77% recall, and 92.49% F1 score, and had training and test losses of 15% and 40%. The performance comparison is illustrated in Figure 8, while the loss comparison is presented in Figure 9.

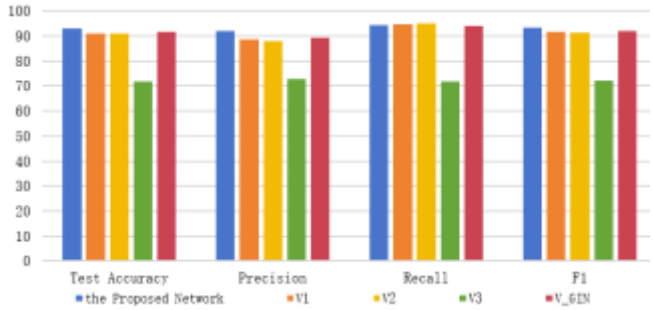

Figure 8. Ablation experiment

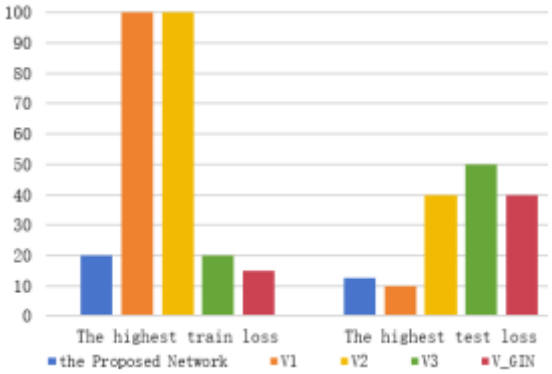

Figure 9. loss comparison

This study compares the proposed model with four deep learning-based vulnerability detection methods, as shown in Figure 10. TokenCNN [16], a token-based approach, treats source code as plain text and ignores semantic and structural information, leading to significant information loss and poor detection performance. StatementLSTM [31] improves on this by treating each line of code as a natural language sentence and embedding it into fixed-length vectors, reducing semantic loss. However, it also processes code as plain text, failing to preserve crucial syntactic and semantic details. Devign [22], a function-level method, uses code property graphs (CPGs) to capture comprehensive semantic and syntactic information. However, its inclusion of irrelevant nodes and edges, along with the absence of slicing techniques, hampers its ability to detect vulnerabilities effectively. Vuldetexp [28] simplifies code representation using slicing but relies solely on Word2Vec for embeddings, which limits its ability to extract rich code information. In contrast, the proposed model fully leverages

code semantics and structure while incorporating slicing techniques, achieving superior detection performance, robustness, and generalization compared to these methods.

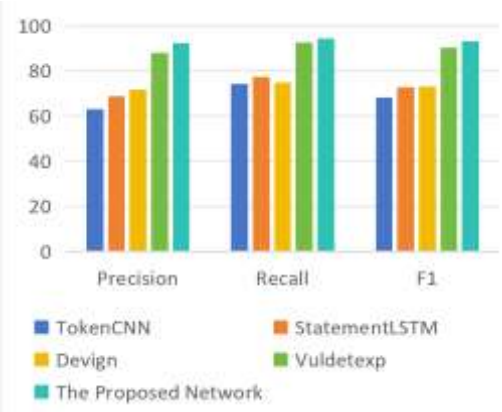

Figure 10. Performance comparison of different models under evaluation

V. Conclusions

This article provides a comprehensive overview of recent advancements in deep learning-based code vulnerability detection, categorizing the methods into sequence-based and graph-based approaches. It details the preprocessing steps involved, including code standardization, PDG (Program Dependency Graph) generation, PDG slicing, and the use of Word2Vec and BERT to extract comprehensive information from sliced graphs. Additionally, the study introduces a novel vulnerability detection method based on Graph Neural Networks (GNN), which extends traditional GGNNs by integrating skip connections, batch normalization, and advanced feature fusion mechanisms. Through ablation studies and comparisons with other deep learning-based methods, the proposed model demonstrates better performance in terms of accuracy, precision, recall, F1 score, and loss minimization. These findings highlight the effectiveness of skip connections in preserving features, batch normalization in enhancing training stability, self-attention mechanisms in capturing global dependencies, and BERT's ability to better extract features by leveraging contextual relationships in graph data, collectively enabling superior performance in vulnerability detection tasks.

Although the proposed model demonstrates significant improvements across several metrics, there are still areas that require further refinement. First, the model primarily analyzes code slices within single functions, making it challenging to handle the complex dependencies present in real-world vulnerabilities that span multiple functions. Future work should incorporate interprocedural analysis to enhance the detection of vulnerabilities involving multiple functions. Second, there is a severe imbalance between the number of vulnerable and non-vulnerable slices in the dataset, which can affect the model's generalization capability. Addressing this imbalance through techniques such as oversampling, undersampling, or the use of Generative Adversarial Networks (GANs) could help mitigate this issue. Additionally, the current model lacks interpretability, as it does not provide a clear indication of the specific code lines where vulnerabilities are detected. Future efforts should focus on integrating and improving tools like GNNExplainer to offer fine-grained explanations of the detection results, thereby enhancing the model's interpretability and practical utility.

Acknowledgment

The authors would like to thank the editor and reviewers for their constructive comments. This paper is supported in part by the Science and Technology Plan Project of Xi'an Beilin District(GX2214) under Grant, and in part by the Plan Project of the Xi'an Science and Technology(22GXFW0047) under Grant.

References

[1] Cnnvd. [EB/OL]. https: //www.cnnvd.org.cn.2023-7-19

[2] Hinton G E, Osindero S, Teh Y W. A Fast Learning Algorithm for Deep Belief Nets[J]. Neural Computation, 2006, 18(7): 1527-1554.

[3] Jacob D, Ming-Wei C, Kenton L, Kristina T, et al. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. [C], North American Chapter of the Association for Computational Linguistics, 2019, abs/1810.04805()

[4] Hoa K D, Truyen T, Trang P, Shien W N, John G, Aditya G, et al. Automatic feature learning for vulnerability prediction. [J], arXiv: Software Engineering, 2017, abs/1708.02368()

[5] Hoa K D, Truyen T, Trang P, Shien W N, John G, Aditya G, et al. Automatic Feature Learning for Predicting Vulnerable Software Components[J], IEEE Transactions on Software Engineering, 2021, 47(1): 67-85.

[6] Sanghoon Jeon, Huy Kang Kim. Autovas: An Automated Vulnerability Analysis System with A Deep Learning Approach[J], Computers & security, 2021, 106: 102308.

[7] Shigang L, Guanjun L, Lizhen Q, Jun Z, Olivier Y D V, Paul M, Yang X, et al. CD-VulD: Cross-Domain Vulnerability Discovery Based on Deep Domain Adaptation[J], IEEE Transactions on Dependable and Secure Computing, 2022, 19(1): 438-451.

[8] Thomas Shippey, David Bowes, Tracy Hall. Automatically identifying code features for software defect prediction: Using AST N-grams. [J], Information & Software Technology, 2019, 106(): 142-160.

[9] Junfeng Tian, Wenjing Xing, Zhen Li. BVDetector: A Program Slice-based Binary Code Vulnerability Intelligent Detection System[J], Information & Software Technology, 2020, 123(): 106289.

[10] Song Wang, Taiyue Liu, Lin Tan. Automatically learning semantic features for defect prediction. [J], Proceedings - International Conference on Software Engineering. International Conference on Software Engineering, 2016: 297-308.

[11] Fabian Y, Christian W, Hugo G, Konrad R, et al. Chucky: exposing missing checks in source code for vulnerability discovery[J], Computer Science, 2013: 499-510.

[12] Thong H, Hoa K D, Yasutaka K, David L, Naoyasu U, et al. DeepJIT: an end-to-end deep learning framework for just-in-time defect prediction[C], IEEE Working Conference on Mining Software Repositories, 2019: 34-45.

[13] Luca Pascarella, Fabio Palomba, Alberto Bacchelli. Fine-grained just-in-time defect prediction. [J], Journal of Systems and Software, 2019, 150(): 22-36.

[14] Yan, X. D. Research on Software Vulnerability Detection Technology Based on Static Taint Analysis and Deep Learning [Master's thesis, Harbin Institute of Technology]. DOI: 10.27061/d.cnki.ghgdu.2021.003610.

[15] Wu F, Wang J, Liu J, Wang W. Vulnerability detection with deep learning//Proceedings of the International C onference on Computer and Communications. Chengd u, China, 2017: 1298-1302.

[16] Rebecca L R, Louis K, Lei H H, Tomo L, Jacob A H, Onur O, Paul M E, Marc W M, et al. Automated Vulne rability Detection in Source Code Using Deep Represe ntation Learning[J], 2018 17th IEEE International Conf erence on Machine Learning and Applications (ICML A), 2018, abs/1807.04320: 757-762.

[17] Han Y, Senlin L, Limin P, Yifei Z, et al. Han-Bsvd: A Hierarchical Attention Network for Binary Software V ulnerability Detection[J], Computers & security, 2021, 108: 102286.

[18] Li Z, Zou D, Xu S, et al. VulDeeLocator: A Deep Learning-based Fine-grained Vulnerability Detector[J]. IEEE Transactions on Dependable and Secure Computing, 2022, 19(4): 2821-2837.

[19] Li Z, Zou D, Xu S, et al. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection[C]//Proceedings 2018 Network and Distributed System Security Symposium. 2018.

[20] Zou D, Wang S, Xu S, et al. μDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection[J]. IEEE Transactions on Dependable and Secure Computing, 2021, 18(5): 2224-2236.

[21] Li Z, Zou D, Xu S, et al. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities[J]. IEEE Transactions on Dependable and Secure Computing, 2022, 19(4): 2244-2258.

[22] Zhou, Y, Liu, S, Siow, J, Du, X, Liu, Y, et al. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks[J], Advances in neural information processing systems, 2019, 32(): 10197-10207.

[23] Wang H, Ye G, Tang Z, et al. Combining Graph-Based Learning with Automated Data Collection for Code Vulnerability Detection[J]. IEEE Transactions on Information Forensics and Security, 2021, 16: 1943-1958.

[24] Zheng W, Jiang Y, Su X. Vu1SPG: Vulnerability Detection Based on Slice Property Graph Representation Learning[J], IEEE International Symposium on Software Reliability Engineering, 2021.

[25] Cao S, Sun X, Bo L, et al. MVD: Memory-Related Vulnerability Detection Based on Flow-Sensitive Graph Neural Networks[C]//Proceedings of the 44th International Conference on Software Engineering. 2022: 1456-1468.

[26] Zou D, Hu Y, Li W, Wu Y, Zhao H, Jin H. mVulPreter: A Multi-Granularity Vulnerability Detection System with Interpretations[J], IEEE Transactions on Dependable and Secure Computing, 2022, PP (99): 1-12.

[27] Fabian Y, Nico G, Daniel A, Konrad R, et al. Modeling and Discovering Vulnerabilities with Code Property Graphs[C], IEEE Symposium on Security and Privacy, 2014: 590-604.

[28] Hu Yutao, Wang Suyuan, Wu Yueming, et al. A Graph Neural Network-Based Method for Slice-Level Vulner a-bility Detection and Explanation[J]. Journal of Softw are,2023,34(06): 2543-2561. DOI:10.13328/j.cnki.jos.0 06849.

[29] Kingma, Diederik P., and Jimmy Lei Ba. Adam: A Method for Stochastic Optimization[J]. International Conference on Learning Representations (ICLR), 2014, abs/1412.6980.

[30] Cheng X, Wang H, Hua J, et al. DeepWukong: Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network[J]. ACM Transactions on Software Engineering and Methodology, 2021, 30(3): 1-33.

[31] Lin G, Xiao W, Zhang J, et al. Deep learning-based vulnerable function detection: A benchmark. In: Proc. of the 21st Int ′ l Conf. on Information and Communications Security (ICICS 2019). 2019. 219-232.

[32] Yang Y, Li G. On the Code Vulnerability Detection Based on Deep Learning: A Comparative Study[J]. IEEE Access, 2024.