Received: 10 January 2023 | Revised: 8 March 2023 | Accepted: 17 March 2023

DOI: 10.1002/spe.3205

RESEARCH ARTICLE

WILEY

VulGraB: Graph-embedding-based code vulnerability detection with bi-directional gated graph neural network

Sixuan Wang | **Chen Huang** | **Dongjin Yu** | **Xin Chen**

School of Computer Science and Technology, Hangzhou Dianzi University, Hangzhou, 310018, China

Correspondence
Dongjin Yu, School of Computer Science and Technology, Hangzhou Dianzi University, Hangzhou, 310018, China.
Email: yudj@hdu.edu.cn

Funding information
Natural Science Foundation of Zhejiang Province, Grant/Award Numbers: LY23F020016, LY21F020020; National Natural Science Foundation of China, Grant/Award Number: 61902096; Key Research and Development Program of Zhejiang Province, Grant/Award Number: 2023C03200

Abstract
Code vulnerabilities can have serious consequences such as system attacks and data leakage, making it crucial to perform code vulnerability detection during the software development phase. Deep learning is an emerging approach for vulnerability detection tasks. Existing deep learning-based code vulnerability detection methods are usually based on word2vec embedding of linear sequences of source code, followed by code vulnerability detection through RNNs network. However, such methods can only capture the superficial structural or syntactic information of the source code text, which is not suitable for modeling the complex control flow and data flow and miss edge information in the graph structure constructed by the source code, with limited effect of neural network model. To solve the above problems, this article proposes a code vulnerability detection method, named VulGraB, which is based on graph embedding and bidirectional gated graph neural networks. VulGraB uses node2vec to convert the program-dependent graphs into graph embeddings of the code, which contain rich structure information of the source code, improving the ability of features to express nonlinear information to a certain extent. Then the BiGGNN is used for training, and finally the accuracy of the detection results is evaluated using target program. The bi-directional gated neural network utilizes a bi-directional recurrent structure, which is beneficial to global information aggregation. The experimental results show that the accuracy of VulGraB is significantly improved over the baseline models on two datasets, with F1 scores of 85.89% and 97.24% being the highest, demonstrating that VulGraB consistently outperforms other effective vulnerability detection models.

KEYWORDS
bi-directional gated neural network, code vulnerability detection, deep learning, graph embedding, node2vec

Abbreviations: AST, abstract syntax tree; CFG, control flow graph; DFD, data flow diagram; NVD, National Vulnerability Database; PDG, program dependence graph; DDG, data dependence graph; GNN, graph neural network; GGNN, gated graph sequence neural network; LSTM, long short term memory; BiLSTM, bi-directional long short term memory; RNN, recurrent neural network; BRNN, bi-directional recurrent neural network; BiGGNN, bi-directional gated graph sequence neural network; GRU, gated recurrent units; DFS, depth first search; BFS, breadth first search; NLP, natural language processing.

1 INTRODUCTION

Over the past decades, the number of software programs has increased dramatically, such as the open-source software platform Github,[1] which hosts over 100 million open-source software programs. At the same time, the software itself is constantly iterating and evolving, which further expand the number of software programs; for example, OpenSSL (Open Secure Sockets Layer)[2] has more than 300 versions. However, unintentional operations (e.g., low quality coding) and intentional operations (e.g., code cloning) during the development process, as well as the insufficient degree of security testing, have resulted in a large number of code vulnerabilities in software programs.

Code vulnerabilities, which can be briefly defined as "software vulnerabilities with security risks", expose software to attack threats, including information leakage, remote control, and denial of service.[3–5] There are a large number of vulnerabilities hidden in the vast amount of software programs available today. Once exploited by malicious attackers, code vulnerabilities have the potential to trigger global cyberattacks with incalculable consequences, such as the WannaCry virus[6] that has caused $4 billion in economic losses worldwide. In addition, the National Vulnerability Database (NVD)[7] released over 17,000 new vulnerabilities in 2019. In the last three years, the NVD has added more than 50,000 new vulnerabilities.

Therefore, timely code vulnerability detection during the software lifecycle is critical, making code vulnerability detection a hot topic in the research field in recent years. However, current code vulnerability detection efforts suffer from a number of shortcomings mainly in the following aspects.

First of all, previous vulnerability detection efforts usually treat code as a linear sequence of text, ignoring the rich semantic information in the code. Code graph structure can be used to represent source code syntax and semantic information, which mainly includes abstract syntax tree (AST), control flow graph (CFG), data flow diagram (DFD), and program dependence graph (PDG). Although some researchers extract program fragments based on the PDG of the code,[8,9] which can capture the data dependency information and control dependency information in the code to some extent, they still consider the code as linear sequence information, and only use word2vec to for feature extraction and training. However, the code also implicitly contains a lot of nonlinear information, which is not enough to be expressed by the embedding vector generated by the word2vec algorithm.

Furthermore, the feature vectors generated by the current code representation methods only include the textual feature vectors of the codes, but not the directed edge feature vectors contained in the code graph structure, resulting in a single feature information, which is insufficient to express the complex connectivity features among the nodes in the code graph structure.

In addition, recent work on code vulnerability detection is often based on deep learning approaches. Among these approaches, neural networks are an important component, which can be used as classifiers for code vulnerability detection. Various neural network models have been used recently, graph neural network (GNN), graph convolutional neural network, and gated graph sequence neural network (GGNN) have shown superior performance in the field of graph-based code vulnerability detection.[10,11] It has been demonstrated that bi-directional long short-term memory (BiLSTM) and bi-directional recurrent neural network (BRNN) outperform the traditional unidirectional long short-term memory neural network (LSTM) and recurrent neural network (RNN) in network models that consider the source code as a linear structure.[12] However, the effects of these various neural network models vary a lot.

The research goal of this article is to propose a novel code vulnerability detection method in the field of code vulnerability detection, named VulGraB, to solve the problems of insufficient nonlinear semantic information representation, missing edge information in the code graph structure, and limited effect of neural network model. To achieve this research goal, we make an innovative attempt in the following aspects.

On the one hand, we use the knowledge of graph theory to analyze the code graph structure in order to learn the nonlinear information in the code semantics. The depth-first traversal information and breadth-first traversal information of nodes can reflect the important connectivity features between graph nodes and express the nonlinear information in the graph. Node2vec[13] embedding model can reflect the characteristics of both depth-first and breadth-first sampling strategies by random wandering, and can learn the nonlinear information in the graph. To our best knowledge, no researcher has used node2vec in the field of code vulnerability detection to learn nonlinear structural information to improve the accuracy of vulnerability detection. Furthermore, we add both text feature vectors and directed edge feature vectors to the feature vectors, thus incorporating the information from different code perspectives and making the feature information more diverse and accurate.

On the other hand, inspired by the above work related to deep learning, this article proposes the use of bi-directional gated graph sequence neural network (BiGGNN)[14] to perform the task of code vulnerability detection. The main idea of BiGGNN is that based on the gated GNN model, a bi-directional recurrent mechanism is added to the network model training and the node embedding is updated with gated recurrent units (GRU) to improve the model learning. This can solve the problem that GNN and GGNN can only be limited to local information and cannot learn global information. Based on our existing knowledge, no researcher has used BiGGNN as a neural network to perform code vulnerability detection tasks.

Based on the ideas mentioned above, this article proposes VulGraB, a novel code VULnerability detection method based on GRAph embedding and Bidirectional gated GNN. VulGraB integrates the textual information and directed edge information in graph structure of the source code, performs embedding learning, and finally trains the BiGGNN to achieve the purpose of detecting code vulnerabilities in the target program with high accuracy. VulGraB is able to enhance the data preprocessing capability, fully learn the graph structure information of the code, and train the BiGGNN by optimizing the parameter settings for a more accurate code vulnerability detection effect.

We demonstrate the advantages of VulGraB by detecting function-level vulnerabilities in program source code in two datasets. We compare VulGraB with three effective detection methods for code vulnerabilities, and compare them separately for the neural network module and the code feature learning module. The experimental results show that VulGraB is able to detect more code vulnerabilities with lower false alarm rate and higher F1 score, which are 2.81% and 97.24% respectively, and consistently outperforms competing methods in the evaluation metrics.

The main contributions of this article are in the following three aspects.

1. In the field area of source code representation for code vulnerability detection, this article proposes for the first time to use node2vec to learn the text information of nodes in program dependency graphs in order to represent the information of nonlinear structure in the code graph structure in the generated source code text feature vectors.
2. In the field of diversity of feature vectors for code vulnerability detection, this article adds feature vectors with directed edges to allow subsequent neural network models to learn both text features of nodes and edge features between nodes.
3. In the field of deep learning for code vulnerability detection, this article for the first time applies BiGGNN, and it adds a bidirectional loop mechanism to the gated GNN model to preserve global information and effectively learn more complex code features.

The rest of this article is structured as follows. Section 2 introduces the scope of the problem that this article focuses on and the need for the proposed approach in this article. Section 3 shows the proposed vulnerability detection method VulGraB. Section 4 presents the experimental setup and analyzes experimental results. Section 5 discusses the limitations of this article. Section 6 presents the work in the area related to this article. Section 7 concludes this article.

2 | BACKGROUND

2.1 | Problem scope

VulGraB focuses on the area of code vulnerability detection. Code vulnerability detection methods can be divided into rule-based methods, code-similarity-based methods, and pattern-based methods.[15] Pattern-based vulnerability detection methods are further divided into machine learning-based methods[16] and deep learning-based methods.[11] In this article, we focus on the deep learning-based methods.

In this article, we apply VulGraB to identify vulnerabilities in given functions from its source code. Note that our goal is not to discover new types of vulnerabilities. Instead, we want to learn the vulnerability code pattern from the source code and detect whether a new piece of code contains vulnerable code patterns similar to those seen in the source code. Thus, VulGraB is useful for detecting common and recurring software vulnerabilities. For this reason, our work is focused on detecting static code vulnerabilities, rather than focusing on problems during dynamic code execution.

We mainly consider four RQs to investigate: (1) the overall improvement of VulGraB compared to other effective vulnerability detection models; (2) the selection of the most effective neural network for VulGraB; (3) the effectiveness of node2vec compared to word2vec in the code representation learning module; (4) the performance of VulGraB in multiple vulnerability types.

2.2 | The need for graph-embedding methods

In the field of code vulnerability detection, previous works based on deep learning have typically treated code as a linear sequence of text, ignoring the rich semantic information in the code. However, PDG can express rich semantic information, which is a representation of the graph structure of the source code. It can clearly express the data and control dependence of each operation in the program. Control dependencies represent all paths of the program during execution, while data dependence represents the logical flow and logical transformation of data within the system. The program dependency graph can well represent the logical function of the program in the form of a graph structure.

As described above, the PDG of the code contains not only linear sequential information but also implicitly contains a lot of nonlinear information, which is not sufficient to be expressed by the embedding vector generated by the word2vec algorithm.[17] Therefore, a more efficient embedding technique is needed to provide richer semantic information to the code learning system.

With embedding techniques, code can be represented as feature vectors. In the previous feature vectors generated from code representations, they often contain only the textual feature vectors of the code, while missing the feature vectors of directed edges contained in the code graph structure, and ignoring the relationships between code lines, resulting in the obtained feature information being more homogeneous and insufficient to express the complex connectivity features between nodes in the code graph structure.[18,19]

VulGraB aims to provide graph embedding functionality by extending the node2vec and BiGGNN architectures in the field of code vulnerability detection, to solve the problem of insufficient representation of nonlinear semantic information and missing edge information for code graph structures, and to improve the neural network model effect.

2.3 | Basic definitions

To detect vulnerabilities, we transform the source code into its PDG form, then generate the embedding with node2vec, and finally train it with BiGGNN. This requires us to understand the relevant basic definitions to better understand VulGraB.

Definition 1. (CFG). For a program $P = \{f_1, \dots, f_\eta\}$, the CFG of function f_i is a graph $G_i = (V_i, E_i)$, where $V_i = \{n_{i,1}, \dots, n_{i,c_i}\}$ is a set of nodes with each node representing a statement or control predicate, and $E_i = \{\epsilon_{i,1}, \dots, \epsilon_{i,d_i}\}$ is a set of direct edges with each edge representing the possible flow of control between a pair of nodes.

Definition 2. (Data dependency). Consider a program $P = \{f_1, \dots, f_\eta\}$, the CFG $G_i = (V_i, E_i)$ of function f_i, and two nodes $n_{i,j}$ and $n_{i,l}$ in G_i where $1 \le j, l \le c_i$ and $j \neq l$. If there is a path from $n_{i,l}$ to $n_{i,j}$ in G_i and a value computed at node $n_{i,l}$ is used at node $n_{i,j}$, then $n_{i,j}$ is data-dependent on $n_{i,l}$.

Definition 3. (Control dependency). Consider a program $P = \{f_1, \dots, f_\eta\}$, the CFG $G_i = (V_i, E_i)$ of function *fi*, and two nodes $n_{i,j}$ and $n_{i,l}$ in Gi where $1 \le j, l \le c_i$ and $j \neq l$. It is said that $n_{i,j}$ post-dominates $n_{i,l}$ if all paths from $n_{i,l}$ to the end of the program traverse through $n_{i,j}$. If there exists a path starting at $n_{i,l}$ and ending at $n_{i,j}$ such that (i) $n_{i,j}$ post-dominates every node on the path excluding $n_{i,l}$ and $n_{i,j}$, and (ii) $n_{i,j}$ does not post dominate $n_{i,l}$, then $n_{i,j}$ is control-dependent on $n_{i,l}$.

Based on data dependency and control dependency, PDG can be defined as follows.

Definition 4. (PDG). For a program $P = \{f_1, \dots, f_\eta\}$, the PDG of function f_i is denoted by $G'_i = (V_i, E'_i)$, where *Vi* is the same as in CFG G_i, and $E'_i = \{\epsilon'_{i,1}, \dots, \epsilon'_{i,d'_i}\}$ is a set of direct edges with each edge representing a data or control dependency between a pair of nodes.

Definition 5. (node2vec).[13] Node2vec is a graph representation-based learning framework that generates continuous vector representations for nodes based on the network structure in the graph. It employs several parameters to control the random walk so that the resulting sequence is a combination of depth first search (DFS) and breadth first search (BFS).

The specific parameters include *walk_length*, *num_walks*, *p*, *q* and *window_size*, which respectively represent the walk length, the number of walks, the probability of visiting the previous node, the depth-first or breadth-first direction of the walk (depth-first for $q < 1$, breadth-first for $q > 1$) and the window size. For a particular node $u \in V$, it has several neighboring nodes, $N_S(u) \subset V$ is the set of nearest neighboring nodes of node u generated by the sampling strategy S, n_i is a particular node in $N_S(u)$, and $\boldsymbol{f}(u)$ is the mapping function that maps node u to a feature vector. The optimization objective of the node2vec model is to maximize the probability of occurrence of given node's nearest neighbors, and its objective function is:

$$\max_f \sum_{u \in V} \left[-\log \boldsymbol{Z_u} + \sum_{n_i \epsilon N_S(u)} \boldsymbol{f}(n_i) \cdot \boldsymbol{f}(u) \right] \tag{1}$$

In order to normalize the above equation, a normalization factor $\boldsymbol{Z_u} = \sum_{n_i \epsilon N_S(u)} \exp(\boldsymbol{f}(n_i) \cdot \boldsymbol{f}(u))$ is added here, which is computationally expensive, so the negative sampling technique $-\log$ is used to optimize $\boldsymbol{Z_u}$.

Definition 6. (BiGGNN).[14] Given a program graph $G = (V, E)$, BiGGNN learns the node embeddings from both incoming and outgoing directions for the graph. In particular, each node $v \in V$ is initialized by a learnable embedding matrix $\boldsymbol{E}$ and gets its initial representation $h_v^0 \in R^d$ where d is the dimensional length. We apply the message passing function for a fixed number of hops, that is, K. At each hop $k \leq K$, for the node v, we apply a summation aggregation function to take as input a set of incoming (or outgoing) neighboring node vectors and outputs a backward (or forward) aggregation vector. The message passing is calculated as follows, where $N(v)$ denotes the neighbors of node v and $\leftarrow / \rightarrow$ is the backward and forward direction.

$$\boldsymbol{h}_{N_{\rightarrow(v)}}^k = \mathrm{SUM}\left(\left\{\boldsymbol{h}_u^{k-1}, \forall u \in N_{\rightarrow(v)}\right\}\right) \tag{2}$$

$$\boldsymbol{h}_{N_{\leftarrow(v)}}^k = \mathrm{SUM}\left(\left\{\boldsymbol{h}_u^{k-1}, \forall u \in N_{\leftarrow(v)}\right\}\right). \tag{3}$$

Then, we fuse the node embedding for both directions:

$$\boldsymbol{h}_{N_{(v)}}^k = \mathrm{Fuse}\left(\boldsymbol{h}_{N_{\rightarrow(v)}}^k, \boldsymbol{h}_{N_{\leftarrow(v)}}^k\right), \tag{4}$$

where $\boldsymbol{h}_{N_{\rightarrow(v)}}^k$ is the forward output and $\boldsymbol{h}_{N_{\leftarrow(v)}}^k$ is the reverse output, and the fusion formula $\mathrm{Fuse}(\boldsymbol{a}, \boldsymbol{b})$ is expressed by the formula:

$$\mathrm{Fuse}(\boldsymbol{a}, \boldsymbol{b}) = \boldsymbol{z} \odot \boldsymbol{a} + (1 - \boldsymbol{z}) \odot \boldsymbol{b}, \tag{5}$$

$$\boldsymbol{z} = \sigma\left(\mathbf{W}_z[\boldsymbol{a}; \boldsymbol{b}; \boldsymbol{a} \odot \boldsymbol{b}; \boldsymbol{a} - \boldsymbol{b}] + \boldsymbol{b}_z\right), \tag{6}$$

where $\odot$ is the component-wise multiplication, σ is a sigmoid function and z is the gating vector. Finally, we use GRU to update node representations.

$$\boldsymbol{h}_v^k = \mathrm{GRU}\left(\boldsymbol{h}_v^{k-1}, \boldsymbol{h}_{N_{(v)}}^k\right). \tag{7}$$

3 | METHOD

We propose VulGraB, a novel vulnerability detection method based on graph embedding and BiGGNN. In this section, the general framework of VulGraB is described, as shown in Figure 1. VulGraB includes four modules: (1) **Code semantic representation module**, which encodes the source code of a function into a graph structure with compound program semantics; (2) **node2vec graph embedding module**, which learns the feature vectors from node texts and edges, by deep random wandering and breadth random wandering of neighboring nodes in PDG; (3) **BiGGNN module**, which applies embedding vectors to train the deep learning neural network; (4) **Application module**, which uses the trained feature vector dictionary and the neural network model to detect the presence of vulnerabilities in the functions of the target program.

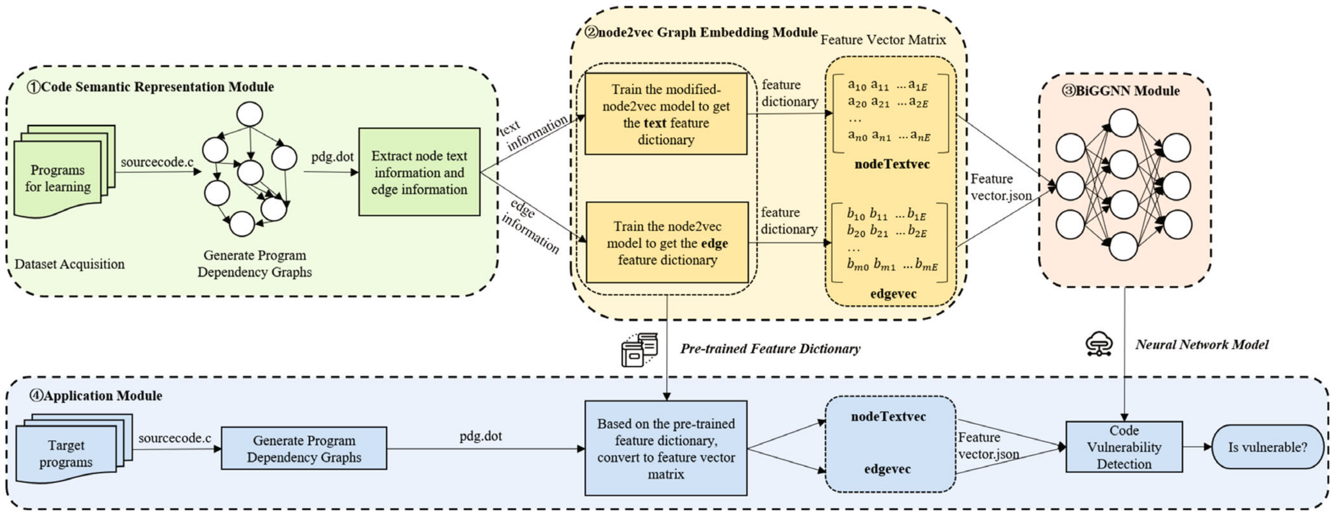

①Code Semantic Representation Module
Programs for learning
sourcecode.c
Dataset Acquisition
Generate Program Dependency Graphs
pdg.dot
Extract node text information and edge information
text information
edge information
②node2vec Graph Embedding Module
Train the modified-node2vec model to get the text feature dictionary
Train the node2vec model to get the edge feature dictionary
feature dictionary
Feature Vector Matrix
nodeTextvec
edgevec
Feature vector.json
③BiGGNN Module
Pre-trained Feature Dictionary
Neural Network Model
④Application Module
Target programs
Based on the pre-trained feature dictionary, convert to feature vector matrix
Code Vulnerability Detection
Is vulnerable?

FIGURE 1 Overall framework of the VulGraB method.

3.1 | Code semantic representation module

This module converts the difficult-to-handle source code format into an easy-to-understand code semantic format.

3.1.1 | Construct program dependency graphs

Since PDG helps to obtain a better code graph structure, instead of extracting textual information directly from the source code, we extract textual and edge information from the PDG of chosen source code at the function-level. With the explicit representation structure of PDG, we can better distinguish between code graph structures at the semantic level. In this article, we use Joern to construct PDGs for a given C/C++ dataset on Linux system and save them as graph description files of DOT type. This process is illustrated in Figure 2 with an example of stack overflow vulnerability. The text description of a node in graphs is separated by a comma, the description before the comma is the node type and the description after the comma is the source code. For example, in the top node of the PDG generated via Joern in Figure 2, the "METHOD" before the comma indicates that the type of this node is method declaration, and the "main" after the comma indicates that the name of this function is "main".

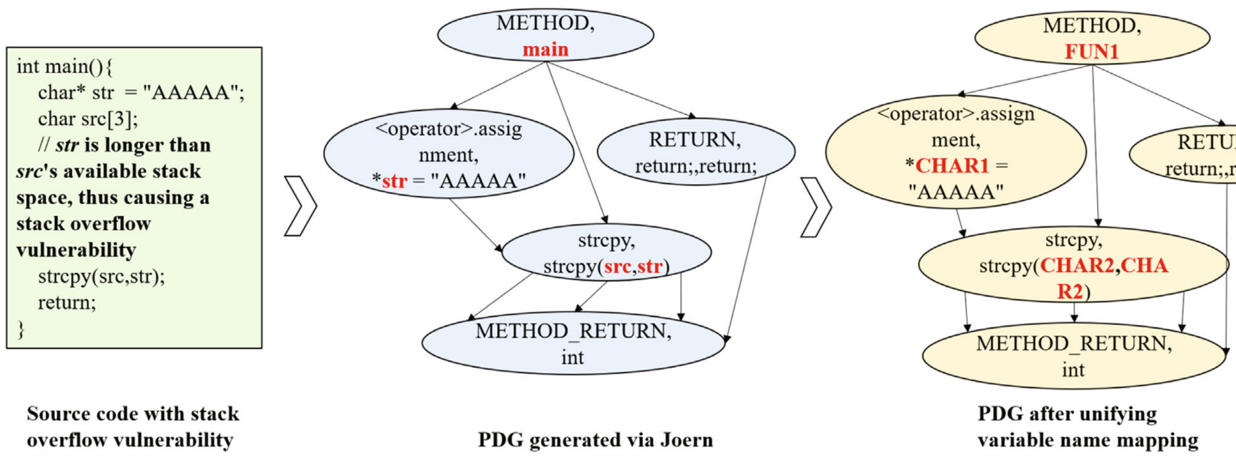

```
int main(){
   char* str = "AAAAA";
   char src[3];
   // str is longer than src's available stack space, thus causing a stack overflow vulnerability
   strcpy(src,str);
   return;
}
```
Source code with stack overflow vulnerability
METHOD, main
<operator>.assignment, *str = "AAAAA"
RETURN, return;,return;
strcpy, strcpy(src,str)
METHOD_RETURN, int
PDG generated via Joern
METHOD, FUN1
<operator>.assignment, *CHAR1 = "AAAAA"
RETURN, return;,return;
strcpy, strcpy(CHAR2,CHAR2)
METHOD_RETURN, int
PDG after unifying variable name mapping

FIGURE 2 Preprocessing the source code to PDG.

In order to eliminate the negative impact of irrelevant information such as user-defined variable names and function names on the subsequent feature vector training, we use a unified variable name mapping approach to map user-defined variables to symbolic names in the PDG graph description file (e.g., unify integer variables as “INT1” and “INT2,” character variables as “CHAR1” and “CHAR2,” and floating-point variables as “ DOUBLE1,” “DOUBLE2”). Note that if several variables appear in different functions, they can be mapped to the same symbolic name. We also map user-defined functions to symbolic names on a one-to-one manner (e.g., “FUN1,” “FUN2”). Similarly, if several function names appear in different functions, they can be mapped to the same symbolic name. This process does not change the default library function in C/C++, such as “strcpy” in Figure 2.

3.1.2 | Extract key information

The core information in the program dependency graph $\boldsymbol{G}$ is the information of nodes and of edges:

$$\boldsymbol{G} = \{\boldsymbol{V}, \boldsymbol{E}\} \tag{8}$$

In the above equation, $\boldsymbol{V}$ represents the set of nodes, $\boldsymbol{E}$ represents the set of edges. In order to extract the key information into a formal format for feature learning, the directed edge relationship between nodes in the DOT file is extracted using the regular text matching method. Specifically, a directed edge relationship can be expressed as:

$$\boldsymbol{E}_{ij} = \boldsymbol{V}_i \rightarrow \boldsymbol{V}_j \tag{9}$$

In the above equation, $\boldsymbol{V}_i, \boldsymbol{V}_j \in \boldsymbol{V}, \boldsymbol{E}_{ij} \in \boldsymbol{E}, 1 \leq i, j \leq k$, and k is the number of nodes. In this way, the set of all directed edges is obtained and saved as a text file. Similarly, using the regular matching method for code to extract the code text corresponding to the node IDs in the DOT file, where a single node can be represented as an array of characters consisting of the smallest text unit *Token*:

$$\boldsymbol{V}_i = \left[Token_0, Token_1, \ldots, Token_n\right] \tag{10}$$

In the above equation, $\boldsymbol{V}_i \in \boldsymbol{V}, 1 \leq i \leq k, k$ is the number of nodes, and n is the number of *Token* in the text message of a single node. Attention is paid to eliminate line breaks and extra spaces in the text. In this way, the set of all nodes is obtained and saved as a dictionary file to preserve the mapping relationship between ID and text.

3.2 | Node2vec graph embedding module

Next, after obtaining all the text and the edge information in the graph, this information is then trained with the node2vec model separately to obtain their feature vector representation.

3.2.1 | Feature training

First, we use the node2vec model for text feature training, and the process is shown in Figure 3. Since node2vec itself treats a node as the smallest unit, here we modify node2vec to treat the token inside the node as the smallest unit. It takes a text file as input that holds directed edges after preprocessing in Section 3.1. When the nodes are visited, the node IDs are combined and represented as “sentences” composed of node2vec, and the “sentences” are sampled and trained to generate a new dictionary file, which holds the text embedding feature vector corresponding to each minimum text unit *Token*. The output vectors of this step are the vectors corresponding to the smallest text unit *Token*, representing the dependency vectors between texts. All the output text feature vectors are stored in a dictionary.

Then, we use the node2vec model for edge feature training. The nodes are identified as unique node IDs instead of the textual attributes mentioned above, and the nodes are modeled with the dependencies between nodes. The training process and target of edge feature training are similar to those of text feature training, as shown in Figure 4. We combine the

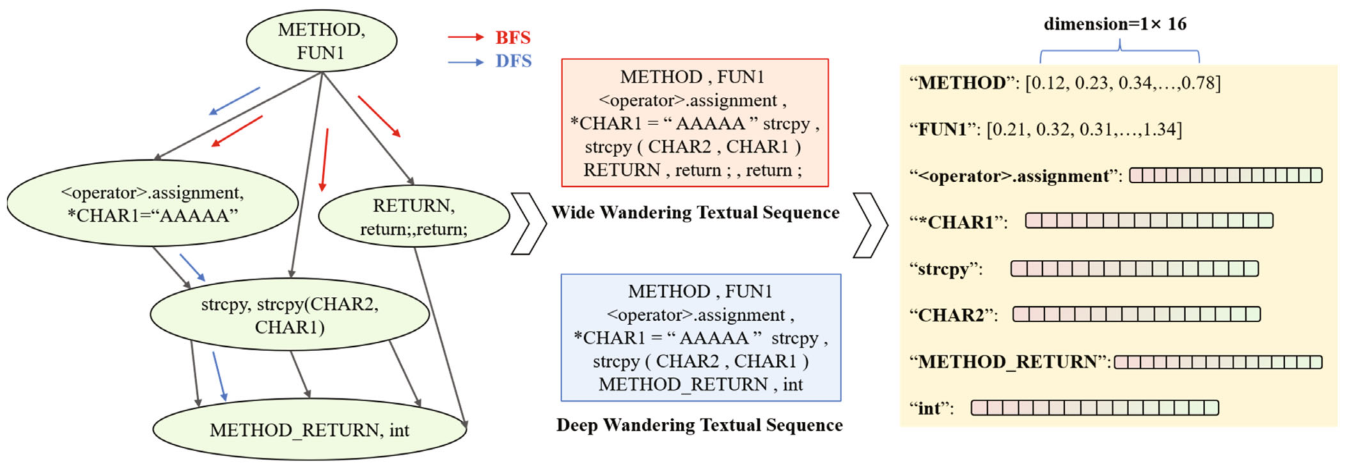

METHOD, FUN1
<operator>.assignment, *CHAR1="AAAAA"
RETURN, return;,return;
strcpy, strcpy(CHAR2, CHAR1)
METHOD_RETURN, int
BFS
DFS
METHOD , FUN1 <operator>.assignment , *CHAR1 = " AAAAA " strcpy , strcpy (CHAR2 , CHAR1) RETURN , return ; , return ;
Wide Wandering Textual Sequence
METHOD , FUN1 <operator>.assignment , *CHAR1 = " AAAAA " strcpy , strcpy (CHAR2 , CHAR1) METHOD_RETURN , int
Deep Wandering Textual Sequence
dimension=1× 16
"METHOD": [0.12, 0.23, 0.34,…,0.78]
"FUN1": [0.21, 0.32, 0.31,…,1.34]
"<operator>.assignment":
"*CHAR1":
"strcpy":
"CHAR2":
"METHOD_RETURN":
"int":

FIGURE 3 The process of training the textual features of the stack overflow vulnerability code with modified node2vec.

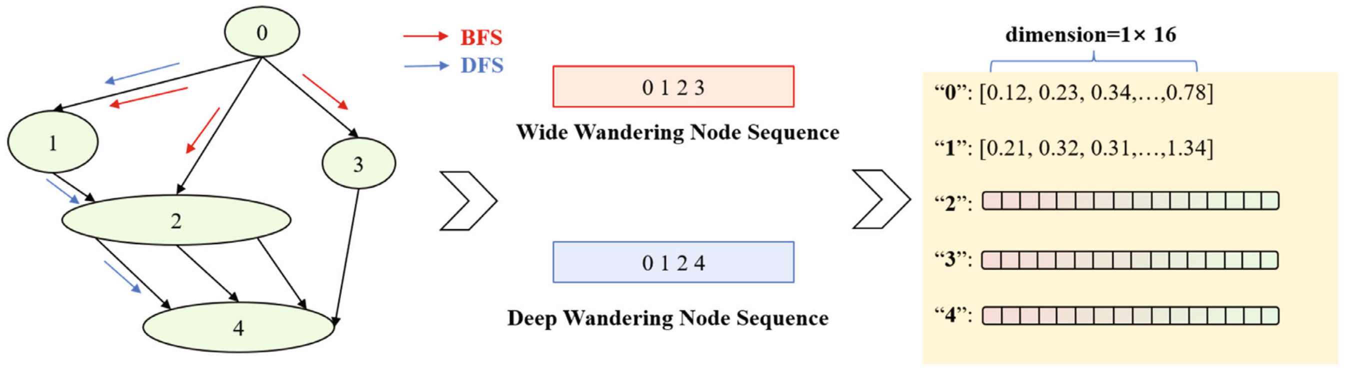


FIGURE 4 The process of training the node features of the stack overflow vulnerability code with node2vec.

wide wandering node sequence and the deep wandering node sequence. The output vector of this step is the dependency vector between graph nodes, and the vector is saved with a dictionary.

3.2.2 | Convert PDG to feature matrix

In order to make each function an individual to allow the neural network classifier to learn, this section converts the PDG into a JSON file format, which represented by feature vectors, to be used as input for the subsequent neural network. Each JSON file has two keys, "jsonNodesVec" corresponding to the text feature matrices of nodes and "jsonEdgesVec" corresponding to the edge feature matrices of the graph.

On the one hand, for "jsonNodesVec," first, we split the text description of each node into *Tokens*, and then convert each *Token* into the corresponding embedding vector through the text vector dictionary obtained in Section 3.2.1. The node ID is used as the key of the node, and the key can be used to quickly find the dictionary encapsulating the attributes of the node, and then find the value of the corresponding attribute.

On the other hand, the conversion of edge information is similar to the conversion of text information, and this step also uses the ID of a node as a key. Unlike the textual information of a node, a node is considered as a whole here without focusing on its internal textual composition. A directed edge represents the existence of a head node and a tail node, and by querying the node ID dictionary obtained from Section 3.2.1 using the IDs of these two nodes as keys, we obtain the head node vector v_s and the tail node vector v_e, both of which have a dimension of 1×16. The two node vectors are subtracted to obtain the embedding vector $v_{s \rightarrow e}$ corresponding to a directed edge. The above process is performed for each directed edge in the list of each program dependency graph to obtain all edge vectors for all functions.

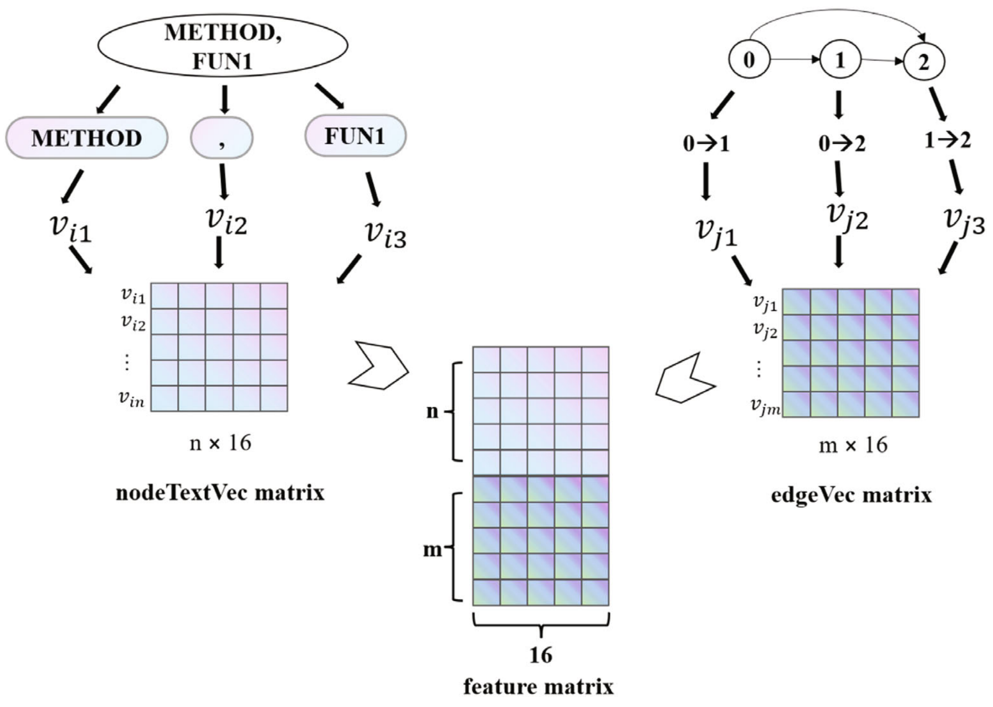


FIGURE 5 The process of converting PDG to feature matrix.

The above process is shown in detail in Algorithm 1 and Figure 5. In Figure 5, we obtain the feature vectors of the three tokens in node "Method, FUN1" through the pre-trained text feature dictionary, combine them into nodeTextVec matrix, and obtain the edgeVec matrix through the pre-trained edge feature "0," "1," "2" in the graph. Then, the node text vectors are encapsulated together with the edge vectors into a JSON file corresponding to the program dependency graph. Thus, this JSON file can be considered as a combination of an $N \times 16$ two-dimensional vector matrix and an $M \times 16$ two-dimensional vector matrix, where N is the number of nodes in a program dependency graph and M is the number of edges.

Algorithm 1. Algorithm for converting PDG to feature vectors.

```
Input: Set of text vectors $F_w$, Set of node vectors $F_e$, Source code file list $L$, Node text dictionary W
Output: Set of feature vector matrices of the source code matrixset
1 matrixset ← ∅; // initialize output
2 FOR file in L DO
3    nodeTextVec ← ∅; // initializing text features
4    edgeVec ← ∅; // initializing edge features
5    $C$ ← read(file); // read file contents
6    allnodes ← getNodeIdList($C$); // get a list of all node IDs in the graph
7    alledges ← getEdgeList($C$); // list of regular match node text attributes
8    FOR node in allnodes DO
9        sentenceVec ← ∅; // initializing sentence features
10       textlist ← getNodeText(W, node); // get the token list of the sentence
11       FOR text in textlist DO
12           sentenceVec ← sentenceVec ∪ getTextEmbedding($\mathbf{F_w}$, text);
13       END FOR
14       nodeTextVec ← nodeTextVec ∪ sentenceVec; // add to text feature set
15    END FOR
16    FOR edge in alledges DO
17       $\mathbf{v_{start}}$ ← getStartNodeEmbedding($\mathbf{F_e}$, edge); // get the directed edge head node feature vector
```

```
18          $\mathbf{v_{end}}$ ← getEndNodeEmbedding($\mathbf{F_e}$, edge); // get the directed edge tail node feature vector
19          edgeVec ← edgeVec ∪ ($\mathbf{v_{end}} - \mathbf{v_{start}}$); // add to edge feature vector set
20      END FOR
21      matrix ←contact(nodeTextVec, edgeVec);
22      matrixset ←addToSet(matrixset, matrix); // add matrix to the set of feature vector matrices
23 END FOR
24 RETURN matrixset
```

3.3 | Bidirectional gated graph neural network module definition

A vulnerability is often caused by multiple semantically related lines of code, which are scattered throughout the code block. In order to capture such scattered semantic associations in PDG, we use the BiGGNN network, which is better at capturing global semantic information to establish contextual semantic associations. Based on our best knowledge, no researcher has used BiGGNN as a neural network to perform code vulnerability detection tasks.

In this section, after obtaining node text feature vectors and edge feature vectors, the feature vectors are applied to train BiGGNN. Figure 6 shows the schematic structure of our neural network. First, we take as input the features generated in the previous module as well as the edge connectivity information. Then, we use the BiGGNN neural network, which learns the input. BiGGNN is an extension of GGNN, which consists of two GGNNs: a forward GGNN model that accepts forward inputs and a backward GGNN model that learns the backward inputs. The BiGGNN considers both outward and inward neighboring nodes of a nodal entity. For each entity, the outward and inward entity representations still retain the outward and inward features of their neighbors. Compared with GGNN, BiGGNN is more advantageous for extracting deep-level features of the code. Then, we use a dense layer for learning the classification and finally pass the data through the softmax layer to get the predicted labels.

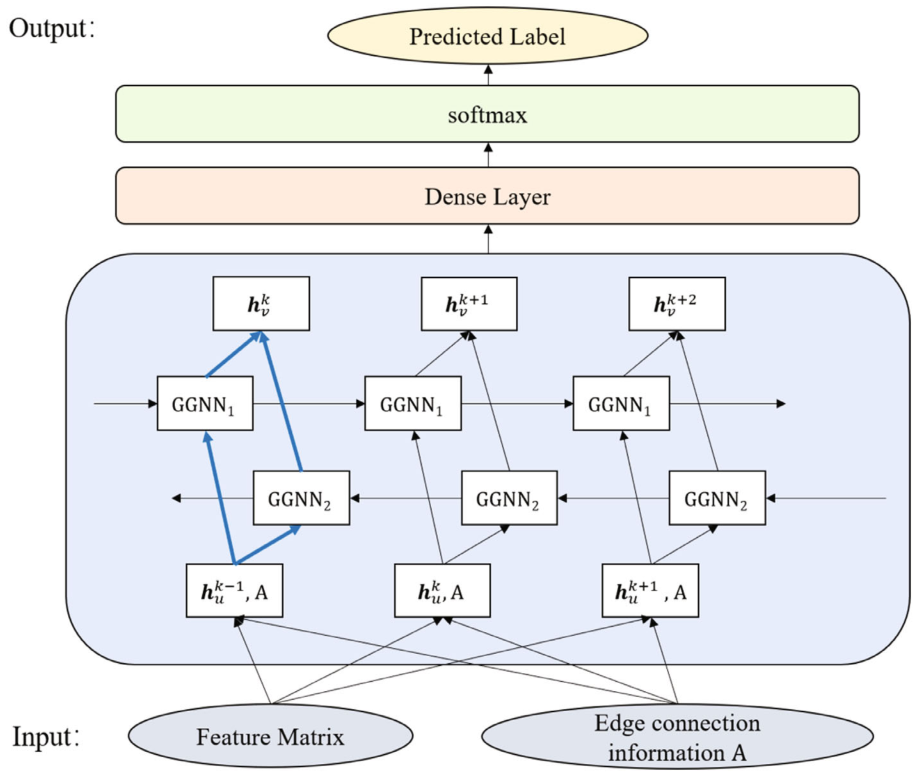


FIGURE 6 The structure of BiGGNN model.

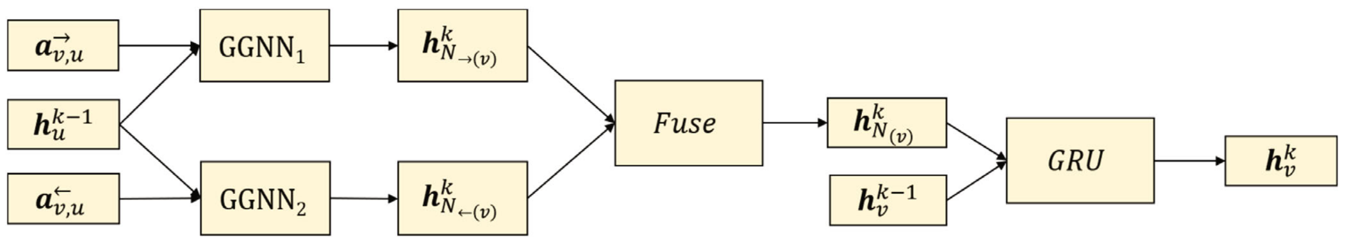


FIGURE 7 Algorithm details of BiGGNN.

Since VulGraB aims to learn the semantics in the code, a weighted average is used as the aggregation function,[14] where the weights are derived from the normalized forward adjacency matrix $\boldsymbol{A}^{\rightarrow}$ and the backward adjacency matrix $\boldsymbol{A}^{\leftarrow}$. Figure 7 shows the specific algorithm process diagram for the bolded part in Figure 6.

When training the BiGGNN convolutional neural network, 70% of the data samples in the JSON file dataset generated in Section 3.4 are first selected as the training dataset, and the training data are randomly disrupted to ensure that the model gets different data samples in different training batches. This treatment will not only improve the convergence rate of the model, but also enhance the prediction results of the model on the test dataset.

3.4 | Application modules

After obtaining the pre-trained feature vector dictionary and neural network, we can detect code vulnerabilities in target program. First, the target program source code is preprocessed as in Section 3.1 to obtain the preprocessed PDG. Next, based on the pre-trained text feature dictionary and edge feature dictionary in Section 3.2.1, perform the steps of converting the PDG to feature vector matrix as in Section 3.2.2 and save it as the JSON file of the target program. Then, reusing the neural network model of the BiGGNN module, the function-level feature vector matrix generated by the target program is used as input for code vulnerability detection, and the output of the neural network model will be normalized by the softmax layer, with the final output encoding.

By using pre-trained text feature dictionaries and edge feature dictionaries, VulGraB can reduce the steps and time of the preprocessing process of the target program, thus achieving fast code vulnerability detection for the target program. Moreover, our dictionary is reusable. Researchers can add new supplementary information to the dictionary as needed to improve it, so that we can feature vectorize more textual information. Instead of retraining the neural network for the target program, we reuse the neural network model that has been trained to work best, and can detect vulnerabilities with patterns similar to those in the source code directly from the target program.

4 | EXPERIMENT

In order to investigate the specific advantages of VulGraB, we designed RQs and set up experiments with reasonably selected datasets. We further analyze the experimental results.

4.1 | Experimental environment and evaluation

4.1.1 | Experimental environment

The experiments in this article are conducted on a device with an Intel(R) Xeon(R) Silver 4108 CPU @ 1.80 GHz, a Matrox Electronics Systems Ltd. Integrated Matrox G200eW3 Graphics Controller, and 64 GB of RAM.

4.1.2 | Evaluation

False positive rate (FPR), false negative rate (FNR), recall rate (Recall), precision (P) and weighted average of precision and recall (F1) are five commonly used metrics to evaluate the effectiveness of models. In this article, assuming that FP is the number of samples without vulnerability but judged as having vulnerability, FNR is the number of samples with vulnerability but judged as not having vulnerability, TP is the number of samples with vulnerability and judged as having vulnerability, TN is the number of samples without vulnerability and judged as not having vulnerability.

$$FPR = \frac{\mathrm{FP}}{\mathrm{FP} + \mathrm{TN}} \tag{11}$$

$$\mathrm{FNR} = \frac{\mathrm{FN}}{\mathrm{TP} + \mathrm{FN}} \tag{12}$$

$$\mathrm{Recall} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}} \tag{13}$$

$$\mathrm{P} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}} \tag{14}$$

$$\mathrm{F1} = \frac{2 \times \mathrm{P} \times \mathrm{Recall}}{\mathrm{P} + \mathrm{Recall}} \tag{15}$$

From Equation (11), FPR indicates the proportion of samples that are incorrectly identified as vulnerable among all samples that are not vulnerable. From Equation (12), FNR indicates the proportion of samples with vulnerabilities that are incorrectly identified as having no vulnerabilities. The lower the FPR and FNR, the better the model detection. From Equation (13), Recall indicates the proportion of all samples with vulnerabilities that are correctly identified as having vulnerabilities. From Equation (14), P indicates the proportion of samples correctly identified as vulnerable code among all samples identified as vulnerable. From Equation (15), F1 indicates the weighted average of Recall and P. The higher the Recall, P and F1, the better the model detection.

4.2 | Datasets

For code in a high-quality project, a function is a representation of a piece of code with a specific function that is harder to segment, so we analyze programs at the granularity of the function to experiment with them. In the field of code vulnerability, researchers usually study C/C++ code due to the lack of other language datasets.[20,21] Therefore, in order to verify the effectiveness of VulGraB on the datasets of single type vulnerability and composite type vulnerability respectively, two C/C++ datasets are also set up in this article. The safe and dangerous functions in the source code files of the two datasets are extracted, and the safe function is marked as negative sample "0," which means the function does not have vulnerability; the dangerous function is marked as positive sample "1," which means the function has a vulnerability. We select the "CWE-78 dataset"[22] as the single type vulnerability dataset and the "FLL dataset"[23] as the composite type vulnerability dataset, and construct them in function units. The "FLL complex type dataset" contains 330 positive samples and 6873 negative samples, and the "CWE-78 single type dataset" contains 6101 positive samples and 6101 negative samples, totaling 6431 positive samples and 12,974 negative samples. The two datasets are described as follows.

4.2.1 | Single type dataset

The NVD[7] is a vulnerability dataset maintained by the National Institute of Standards and Technology that can be used as a benchmark dataset for software vulnerability classification research. In this article, we collect CWE-78 type vulnerabilities in NVD in the form of source code for C/C++ language, which is a non-neutralizing operating system command injection vulnerability for special elements used in operating system commands, and is listed by CWE[22] as one of the 25 most dangerous vulnerabilities in 2022, with a hazard score ranking of 6th, and there is enough data in NVD for this type of vulnerability to support model training. This type of vulnerability is usually easy to find and exploit, and can allow an attacker to completely take over a system, steal data, or prevent an application from running, so it is valuable to start researching from it. This dataset includes a total of 6101 positive and 6101 negative samples.

4.2.2 | Composite type dataset

The composite type dataset uses the real and widely used FFmpeg, LibPNG, and LibTIFF projects. Each project contains a collection of nine vulnerability types, which include several widely available typical code vulnerability types such as buffer errors, resource management errors, format string vulnerabilities, numeric errors, division by zero, input validation, and null pointer dereferencing.[23] The distribution of positive and negative samples in the dataset is shown in Table 1.

It can be observed that the above dataset includes a total of 330 positive samples and 6873 negative samples, the number of negative samples is much larger than the number of positive samples, and there is a class imbalance problem, which may lead to more serious problems when training the model. Therefore, in this article, under-sampling and over-sampling are used to deal with this problem in order to balance the number of positive and negative samples as much as possible.

4.3 | Analysis of results

RQ1: How is the overall performance of VulGraB compared with other effective vulnerability detection models?

To answer this question, methods from the dominant deep learning models in the field of code vulnerability detection are selected as the baseline methods in this article, and they are the same as the domain covered in this article. For each method, we conduct experiments based on two datasets separately and explain the experimental results.

A. VulDeePecker

Li et al.[24] proposed VulDeePecker, which relies on data flow information to represent source code. It locates vulnerability-related APIs in the source code, then extracts associated program slices, and thus generates code gadgets by combining the API-related program slices for accurate vulnerability detection. It uses code gadgets as the granularity for pinpointing vulnerabilities. Word2vec is used as the word embedding method, and BiLSTM is selected as the neural network in the article.

B. FUNDED

Wang et al.[25] proposed FUNDED, a new learning framework for building vulnerability detection models. This framework uses GNNs to develop a new graph-based learning approach to capture information. Unlike previous work that treats

TABLE 1 Analysis of the number of positive and negative samples in the data set.

				Total of CWE-78	
Number of positive sample				6101	
Number of negative sample				6101	
Total				12,202	
	FFmpeg	**LibPNG**	**LibTIFF**	**Total of FLL**	**Total after class balance**
Number of positive sample	192	44	94	330	1239
Number of negative sample	5565	577	731	6873	1240
Total	5757	621	825	7203	2479

programs as continuous sequences or untyped graphs, FUNDED learns and operates graph representations of program source code, where individual statements are connected to other statements by relational edges.

C. DeepWukong

Cheng et al.[19] proposed DeepWukong, a new deep-learning based embedding method for static detection of software vulnerabilities in C/C++ programs. They make a new attempt to first extract XFG (a subgraph of the PDG) unstructured and structure information from PDG based on program points of interest, then embed them using doc2vec, and finally leverage a graph neural architecture comprising a graph convolutional layer, a graph pooling layer, and a graph readout layer to perform the graph classification task for XFGs.

The performance of VulGraB compared with the baseline model methods on the two datasets is shown in Table 2. In CWE-78 dataset, VulDeePecker obtains only 78% of the F1, with the highest FPR and FNR of 37.45% and 12.12% respectively, and FUNDED obtains 91.95% of the F1 while DeepWukong obtained 94%.VulGraB shows significant improvement in all five evaluation metrics relative to the baseline model, with F1 values exceeding 97%, including a 33.3% reduction in FPR and a 17.9% increase in F1 scores relative to VulDeePecker, and a 3.1% reduction in FPR and a 3.95% increase in F1 scores relative to FUNDED. In the FLL dataset, VulDeePecker only obtains 74.76% of F1 and FUNDED obtains 82.35%, and DeepWukong obtains 85.13%. The F1 value of VulGraB reaches 85.89%, which improve 9.13%, 3.54%, and 0.76% respectively over the three baseline methods, and obtains the best evaluation index value. In addition, VulGraB has relatively similar Recall, P and F1, and all three are maintained at a high level, which indicates that the detection effect of VulGraB is good for both positive and negative samples.

In the experimental results, it can be observed that VulGraB exhibits the lowest FPR and FNR in the dataset, while VulDeePecker's FPR and FNR metrics are the highest. Therefore, we would like to analyze the reasons "Why VulGraB can achieve the lowest FNR and FPR?". First, although VulDeePecker claims to perform well on low-level vulnerabilities such as buffer overflows, it does not handle high-level vulnerabilities very well. This may be because most of the high-level vulnerabilities in the CWE-78 and FLL datasets have complex control flow and data flow, while VulDeePecker only considers the data flow and not the control flow such as branch conditions. To this end, we choose a double-free vulnerability in FLL as an example, to illustrate the importance of control flow in vulnerability detection. This example refers to "if" control flow.

The top figure in Figure 8 shows an example of a double-free vulnerability, while the bottom figure shows an example without vulnerability. Both of them have the same data dependency and different control dependencies, and if only the data dependency is considered, it may be considered vulnerable, resulting in a high FPR value. In addition to the control flow related to "if", there are also various control flows such as "while/for," "switch," "goto" and so forth. VulGraB integrates control flow and data flow is more effective in detecting such vulnerabilities.

Now, we analyze the vulnerability detection from the library/API perspective. As described in VulDeePecker, due to the difficulty of finding the critical point of vulnerability (i.e., API), it can only extract a limited number of code gadgets from the dataset in some cases, and cannot find vulnerabilities that are not related to the API. Table 3 summarizes

TABLE 2 Validity of VulGraB compared with the baseline method in the dataset (unit: %).

Dataset	Method	FPR	FNR	Recall	P	F1
CWE-78	VulDeePecker	37.45	12.12	87.88	70.12	78.00
	FUNDED	6.23	9.60	90.40	93.55	91.95
	DeepWukong	4.83	5.85	94.15	95.13	94.63
	VulGraB	**2.81**	**2.72**	**97.28**	**97.19**	**97.24**
FLL	VulDeePecker	41.24	15.69	84.30	67.15	74.76
	FUNDED	15.09	18.75	81.25	83.48	82.35
	DeepWukong	**12.77**	16.42	83.58	**86.47**	85.13
	VulGraB	13.37	**14.00**	**86.00**	85.79	**85.89**

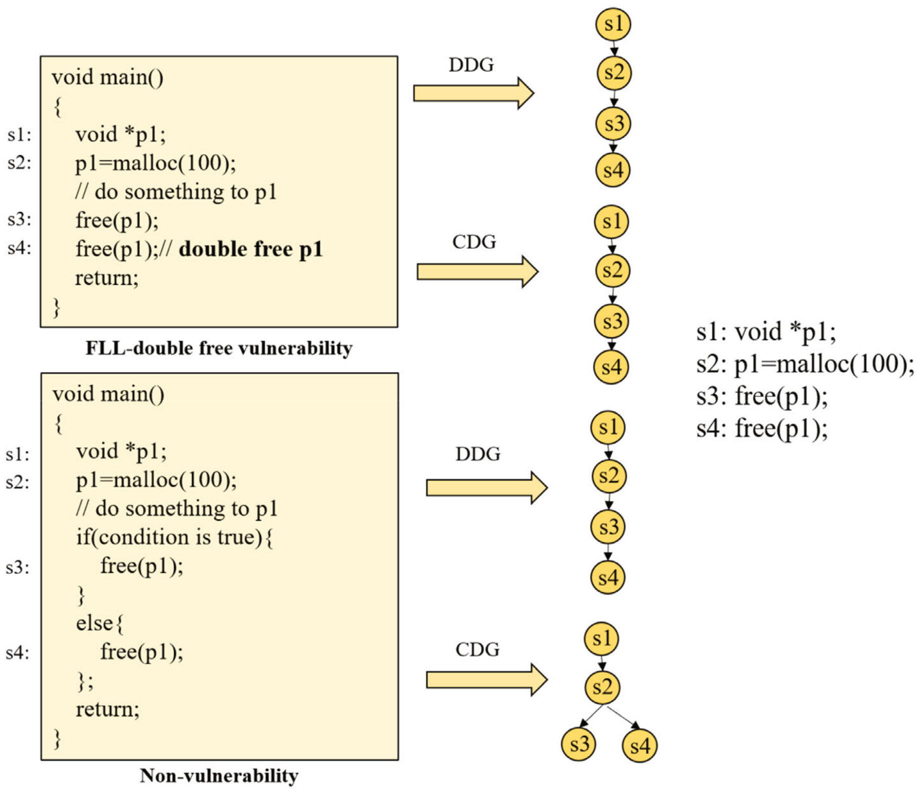


FIGURE 8 The importance of *if* control flow in code vulnerability.

TABLE 3 Analysis of the number of vulnerabilities related and unrelated to library/API function calls.

CWE ID	Part of C/C++ library/API function calls related to vulnerabilities	Number of API-related vulnerabilities	Number of vulnerabilities not related to API
CWE-78	cin, exit, execvp, stdin, recv, spawnv, wcsncat, execlp, fclose, close, execv, strlen, read, system	5722/6101	379/6101
FLL	_snprintf*, perror, strlen, strrchr, realloc, assert, memset, _alloca*, memmove, memcmp	273/330	57/330

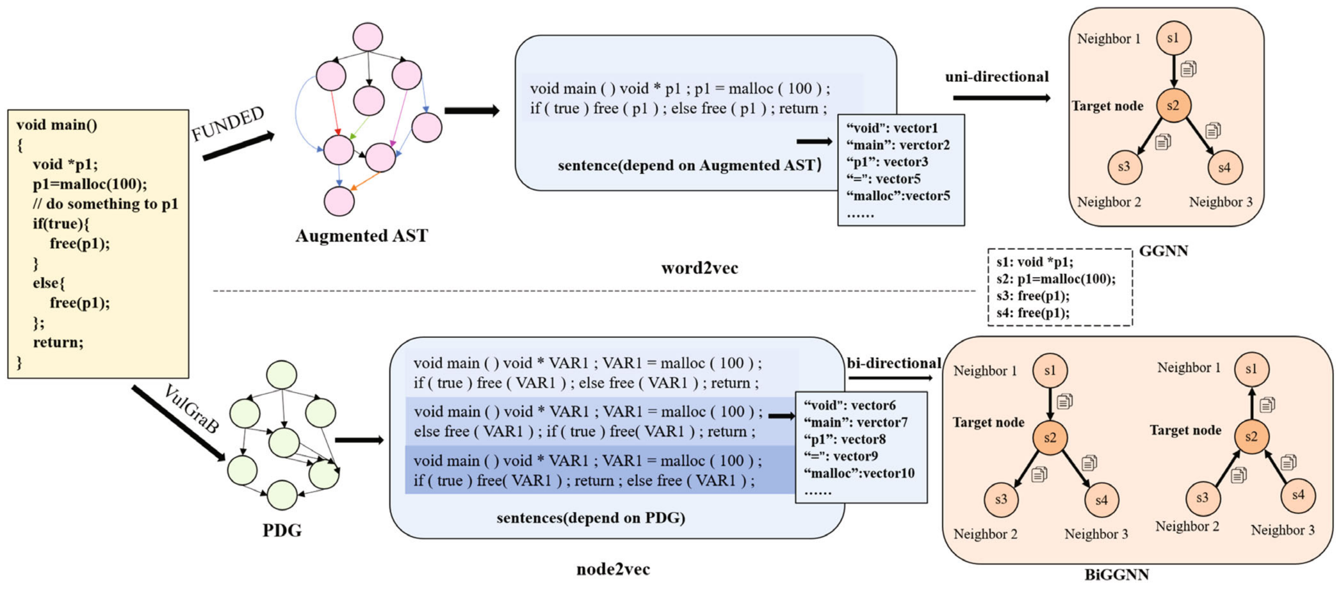
the number of vulnerabilities related and unrelated to C/C++ library/API function calls in the two datasets, and lists some of the C/C++ library/API names, where "*" indicates a wildcard. The API-related vulnerabilities fail to cover all functions in the dataset, missing 5% to 10% of the vulnerabilities, which causes false negatives in the VulDeePecker detection results. VulGraB, in contrast, can extract all functions in the source code and theoretically detect all function-level vulnerabilities.

For further study, we experiment with VulDeePecker on samples in the dataset related to API only, and the results are shown in Table 4, where its effectiveness improves relative to that of VulDeePecker on all samples, and Recall even exceeds VulGraB's. This is because it ignores all API-unrelated vulnerabilities and does not take them into account for detection, resulting in a reduction in the total number of samples. For a given number of correct detections, the smaller the sample size, the better the evaluation metrics and therefore the higher the recall. Nevertheless, the F1 value of VulGraB is still higher than that of VulDeePecker.

Then we analyze the advantages of VulGraB over the FUNDED baseline method, which takes into account the use of GGNN to learn the code structure graph, and has a certain degree of improvement over the traditional linear structure neural network. However, it uses word2vec to learn an enhanced abstract syntax tree, which focuses on the

TABLE 4 Vuldeepecker's experimental results in samples related to API (unit: %).

Dataset	Method	FPR	FNR	Recall	P	F1
FLL-API only	VulDeePecker	38.83	**11.31**	**88.69**	75.47	81.85
FLL	VulDeePecker	41.24	15.69	84.30	67.15	74.76
FLL	VulGraB	**13.37**	14.00	86.00	**85.79**	**85.89**
CWE-78-API only	VulDeePecker	25.24	8.44	91.56	83.16	87.16
CWE-78	VulDeePecker	37.45	12.12	87.88	70.12	78.00
CWE-78	VulGraB	**2.81**	**2.72**	**97.28**	**97.19**	**97.24**



```
void main()
{
    void *p1;
    p1=malloc(100);
    // do something to p1
    if(true){
        free(p1);
    }
    else{
        free(p1);
    };
    return;
}
```

void main ( ) void * p1 ; p1 = malloc (100) ; if (true) free (p1) ; else free (p1) ; return ;

"void": vector1
"main": verctor2
"p1": vector3
"=": vector5
"malloc":vector5
......

s1: void *p1;
s2: p1=malloc(100);
s3: free(p1);
s4: free(p1);

void main ( ) void * VAR1 ; VAR1 = malloc (100) ; if (true) free (VAR1) ; else free (VAR1) ; return ;

void main ( ) void * VAR1 ; VAR1 = malloc (100) ; else free (VAR1) ; if (true) free(VAR1) ; return ;

void main ( ) void * VAR1 ; VAR1 = malloc (100) ; if (true) free(VAR1) ; return ; else free (VAR1) ;

"void": vector6
"main": verctor7
"p1": vector8
"=": vector9
"malloc":vector10
......

FIGURE 9 Double-free vulnerability and its detection process in FUNDED and in VulGraB.

syntactic learning for words and lacks the contextual semantics between tokens and sentences. In contrast, we use node2vec, which can learn grammar based on node relations first, and also semantics based on word relations. Also, BiGGNN can fully learn the forward and backward neighborhood relationship of nodes compared to GGNN, achieving the best results.

Figure 9 shows the difference in the detection of code vulnerabilities using the two methods with a specific example of a double-free vulnerability, which is related to the "if" control flow. For the training of embedding vectors, word2vec can only generate a sentence based on the augmented AST, which It is limited to local syntactic information, while node2vec is based on the random wandering sampling strategy of the code graph, generating multiple sentences, and from multiple different sentences, the embedding vector of the token is trained. Node2vec is based on the graph, taking into account the multiple possibilities of the sequence. In addition, as a deep learning classifier, the GGNN training process can only update node information based on one-way edge paths, and for the target node, it mainly obtains the information of predecessor nodes based on directed edges, and can only obtain a very small amount of information of successor nodes. However, in BiGGNN, the target node can not only obtain the information of predecessor nodes based on forward input, but also obtain a lot of information of successor nodes based on backward input to fully learn the neighborhood information and maximize the information utilization and aggregation.

DeepWukong's experimental results are the closest to VulGraB's. On the FLL dataset, its FPR is 0.6% lower than ours and its P is 0.68% higher than ours, while our Recall is 2.4% higher than theirs and overall F1 is 0.76% higher than theirs, which means VulGraB predicts more positive samples than DeepWukong. This may be due to the fact that we added the embedding information of directed edges to the code embedding and learned the embedding using a network with bidirectional structure, and thus were more sensitive to the information that may contain vulnerabilities during the prediction process. This gives us fewer false positives and reduces some accuracy, but greatly reduces false negatives as well as improves recall and improves overall detection. By the way, since vulnerabilities can lead to serious security risks for software, it is more important for us to find a vulnerability than to miss it when possible. Therefore, it is worthwhile to increase a small number of FP to decrease a larger number of FN in the vulnerability detection task.

Insight: Regardless of whether it is API-related or not, VulGraB can accurately detect vulnerabilities through syntactic and semantic information in the code graph. VulGraB can achieve 2.81% in FPR and 97.24% in F1 in the CWE-78 dataset and 13.37% in FPR and 85.89% in F1 in the FLL dataset.

RQ2: For the neural network classifier, how does the BiGGNN used in this article compare with other neural network models?

Deep learning neural network is an important part of VulGraB. We investigate both sequence-based neural networks and graph-based neural networks, which are RNN, BiLSTM, BGRU, GNN, GGNN, and BiGGNN. The first three are sequential sequence-based neural networks, and the last three are networks based on graph structure processing. Except for BiGGNN, the other five neural networks are the current mainstream networks widely used in the field of code vulnerability detection, while no researcher has used BiGGNN. In order to investigate the effect of the BiGGNN neural network module in VulGraB, this experiment controls the relevant variables, while keeping the parameters in the previous node2vec graph embedding module consistent (p takes 0.5 and q takes 1).

The results of the 6 neural network models are shown in Table 5. In the CWE-78 dataset, the F1 of the networks based on graph structure processing are all over 95%, and the F1 of BiGGNN reaches over 96%, while the F1 value of BGRU, which is the most effective neural network based on sequential sequences, is only 95%. And in the FLL dataset, BiGGNN also reaches the best metrics. This may be because the program contains complex contextual relationships, while the sequential sequence-based neural network treats the program as a one-dimensional sequence, and it is difficult to learn the structure and contextual information in the program, and the network based on graph structure processing can overcome this drawback to some extent.

Moreover, in the CWE-78 dataset, bidirectional RNNs (e.g., BiLSTM and BGRU) can improve Recall and F1 by 2.97% and 3.32% on average compared to unidirectional RNNs, which is due to the fact that the bidirectional structure can accommodate more contextual information. In addition, GGNN outperforms GNN, probably because GGNN is based on GNN with the addition of GRU, which can be used to solve the long dependency problem in RNN networks. From the perspective of code vulnerability detection, GRU has the advantage that for longer vulnerability code input, it can easily retain the specific features that are already present to reserve the global code information, instead of limiting itself to local vulnerability features.

To better observe the effectiveness of the neural network as classifiers, we input the preprocessed 4269 test samples of the CWE-78 dataset into 6 pre-trained neural networks, project the 64-dimensional output of the neural network into

TABLE 5 The effect on 5 traditional neural networks versus BiGGNN.

Dataset	Model	FPR	FNR	Recall	P	F1
CWE-78	RNN	13.93	8.07	91.93	91.47	91.28
	BiLSTM	5.62	5.28	94.72	94.19	94.19
	BGRU	5.12	4.93	95.07	94.94	95.00
	GNN	5.38	4.51	95.49	95.26	95.36
	GGNN	4.34	4.10	95.90	95.57	95.70
	BiGGNN	**4.15**	**3.36**	**96.64**	**96.23**	**96.36**
FLL	RNN	20.41	20.33	79.67	78.55	79.11
	BiLSTM	18.06	17.92	82.08	81.00	81.54
	BGRU	17.28	16.83	83.17	81.87	82.51
	GNN	16.89	16.58	83.42	82.25	82.83
	GGNN	15.79	16.17	83.83	83.27	83.56
	BiGGNN	**15.01**	**15.16**	**84.83**	**84.13**	**84.48**

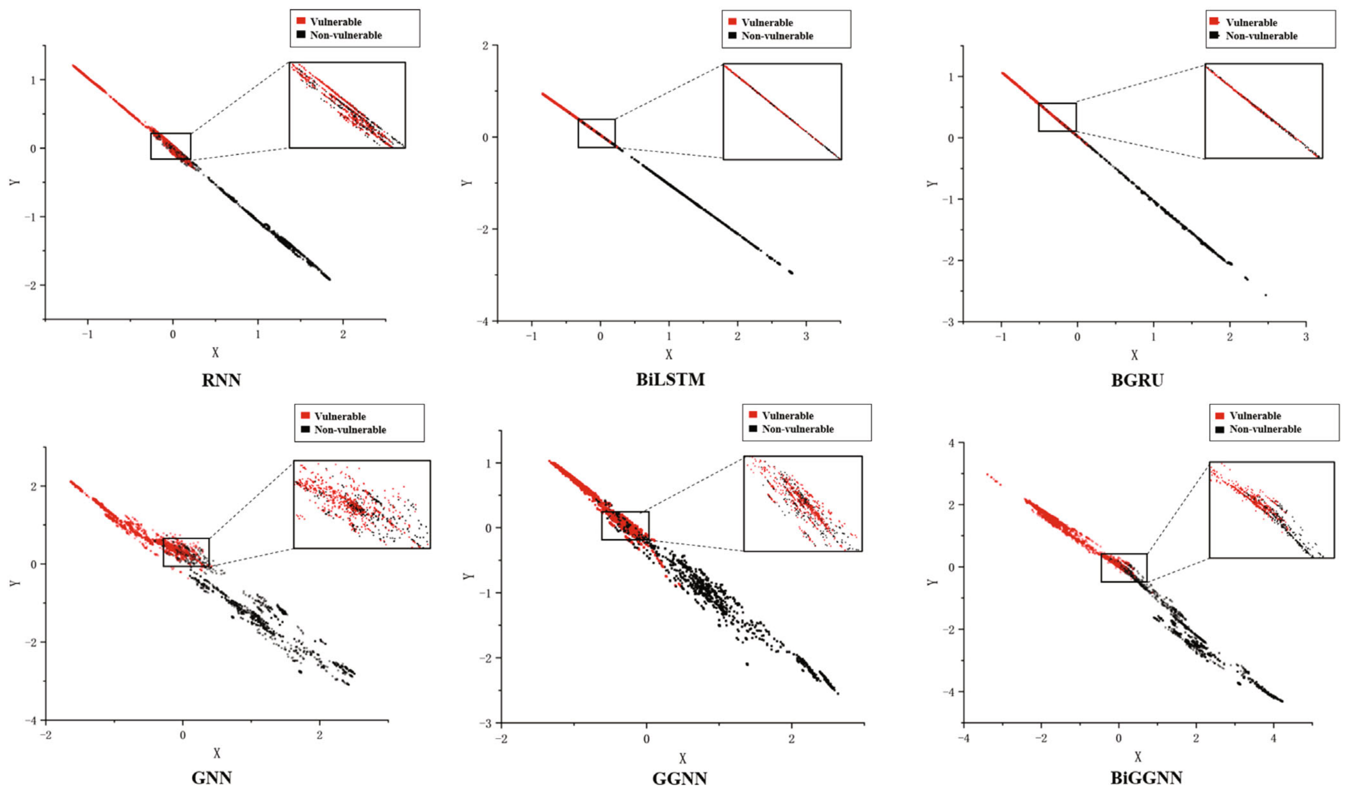


FIGURE 10 Dimensionality reduction distribution of the feature space of 6 neural networks on the test samples.

the 2-dimensional space by the fully connected layer downscaling method, and observe the distribution of the data in the feature space by visualization, as shown in Figure 10. Among them, red dots are vulnerability samples with labeling and black dots are non-vulnerability samples with labeling. The clearer the demarcation between the two types of samples, the better the classification effect of the neural network. We can observe that the RNN-based neural networks (RNN, BiLSTM, BGRU) tend to aggregate the samples into a line, while the GNN-based neural networks project the samples more scattered. Besides, we zoom in on the transition area between red and black to see the more refined classification effect. The first five neural networks show a mixed region at the junction of the two samples that is difficult to separate, and the RNN-based neural network projects all samples onto a single line, making all samples less distinguishable from each other. In contrast, the boundary between the vulnerable and non-vulnerable samples in BiGGNN is more obvious, and the distribution of vulnerable and non-vulnerable samples is very uniform, and the dividing line is in the middle of the distribution map, which reflects that the BiGGNN we use has a more excellent classification prediction effect.

Insight: VulGraB-enabled GNNs (especially BiGGNN) are more effective than VulGraB-enabled RNNs. By clearly delineating the boundary between vulnerable and non-vulnerable codes in the feature space, BiGGNN achieves the best code vulnerability detection in our method.

RQ3: For the code representation learning, can node2vec achieve better code representation learning results than word2vec?

To answer this question and to study the improvement effect of node2vec over word2vec, this RQ replaces the node2vec graph embedding module with word2vec for experiments, and the subsequent neural network models are BiGGNN. For each method, we conduct experiments based on two datasets separately and explain the experimental results. During the experiments, we also conduct several experiments with different parameters of node2vec in order to select the optimal parameters.

The experimental results are shown in Figure 11. Figure 11A shows the effect of the two methods on the CWE78 dataset, and Figure 11B shows the effect of the two methods on the FLL dataset, and node2vec is optimal on both datasets. In the CWE-78 dataset, the F1 of the word2vec method is 92.98% and the F1 of the node2vec method is 97.24%, an improvement of 4.24%; in the FLL dataset, the F1 of the word2vec method is 83.22% and the F1 of the node2vec method is 85.89%, an improvement of 2.67%. In addition, the precision of VulGraB reaches 97.09% and 85.79%, respectively, while the FPR is only 3.13% and 13.37%.

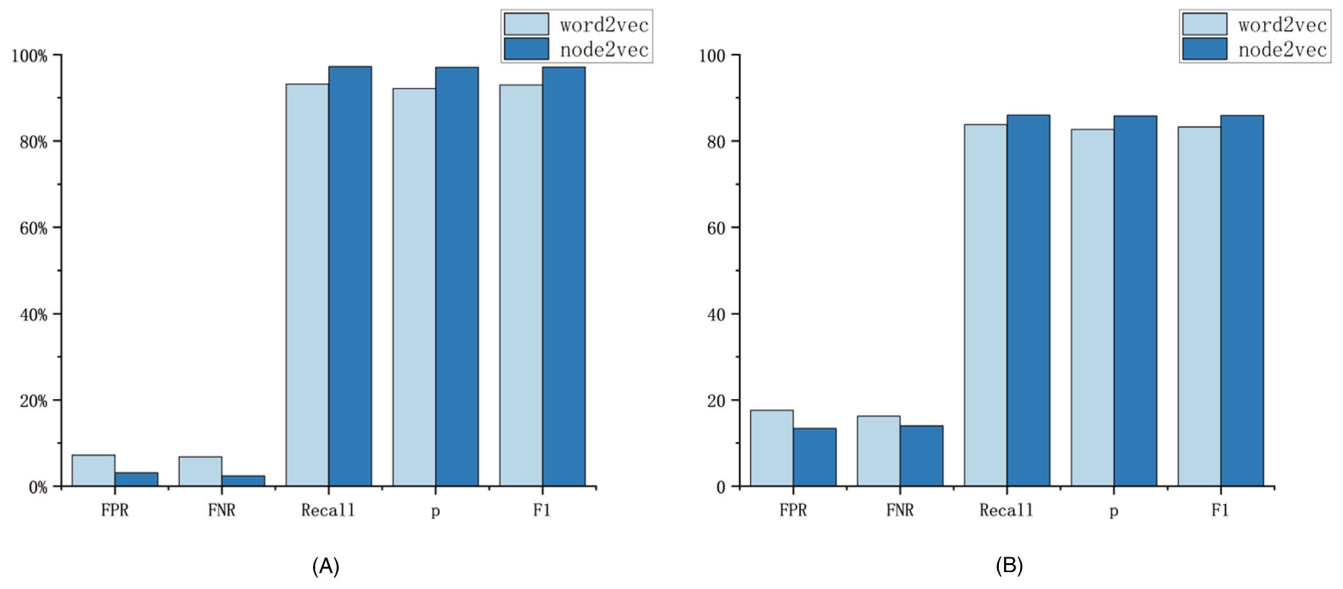
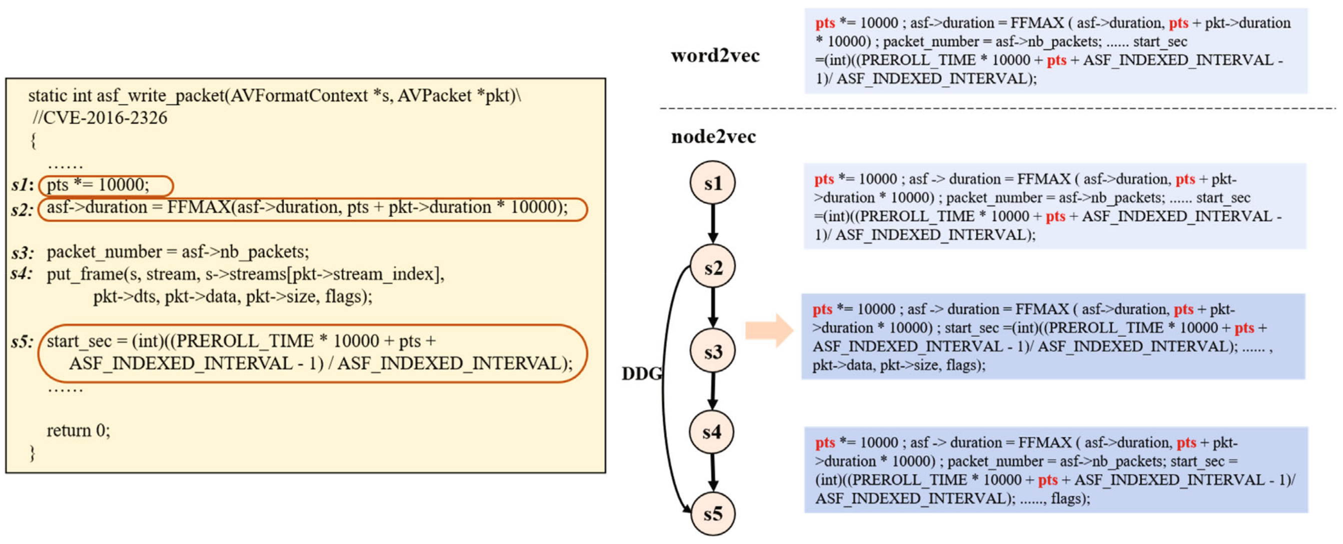


FIGURE 11 Enhancement of node2vec over word2vec(unit: %). (A) Comparison of evaluation on CWE-78 dataset. (B) Comparison of evaluation on FLL dataset.



```
static int asf_write_packet(AVFormatContext *s, AVPacket *pkt)\
 //CVE-2016-2326
 {
    ......
s1: pts *= 10000;
s2: asf->duration = FFMAX(asf->duration, pts + pkt->duration * 10000);

s3: packet_number = asf->nb_packets;
s4: put_frame(s, stream, s->streams[pkt->stream_index],
          pkt->dts, pkt->data, pkt->size, flags);

s5: start_sec = (int)((PREROLL_TIME * 10000 + pts +
       ASF_INDEXED_INTERVAL - 1) / ASF_INDEXED_INTERVAL);
    ......

    return 0;
 }
```

word2vec

pts *= 10000 ; asf->duration = FFMAX (asf->duration, pts + pkt->duration * 10000) ; packet_number = asf->nb_packets; ...... start_sec =(int)((PREROLL_TIME * 10000 + pts + ASF_INDEXED_INTERVAL - 1)/ ASF_INDEXED_INTERVAL);

node2vec

pts *= 10000 ; asf -> duration = FFMAX (asf->duration, pts + pkt->duration * 10000) ; packet_number = asf->nb_packets; ...... start_sec =(int)((PREROLL_TIME * 10000 + pts + ASF_INDEXED_INTERVAL - 1)/ ASF_INDEXED_INTERVAL);

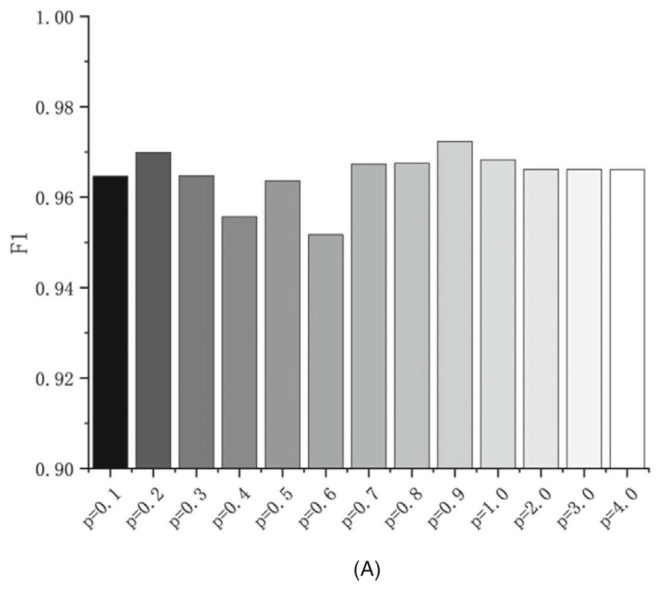
pts *= 10000 ; asf -> duration = FFMAX (asf->duration, pts + pkt->duration * 10000) ; start_sec =(int)((PREROLL_TIME * 10000 + pts + ASF_INDEXED_INTERVAL - 1)/ ASF_INDEXED_INTERVAL); ...... , pkt->data, pkt->size, flags);

pts *= 10000 ; asf -> duration = FFMAX (asf->duration, pts + pkt->duration * 10000) ; packet_number = asf->nb_packets; start_sec = (int)((PREROLL_TIME * 10000 + pts + ASF_INDEXED_INTERVAL - 1)/ ASF_INDEXED_INTERVAL); ......, flags);

FIGURE 12 Advantages of node2vec in code vulnerability detection.

This can be explained using the example in Figure 12, which is related to data flow. Figure 12 shows an integer overflow vulnerability in FLL, where "pts" is a time-dependent value. Here we need to pay attention to the sliding window, for both word2vec and node2vec learn code based on a sliding window. In word2vec, there are two "pts"-independent lines of code, s3 and s4, between lines of code s2 and s5. Since word2vec is based on a continuous sliding window for learning, word2vec may contain irrelevant information such as in s3 and s4. In contrast, in node2vec, we can randomly wander based on the control flow and data flow of the code. In this example, there is a data flow between s2 and s5, so we can get the statement of s5 immediately after s2, thus focusing the semantic information of "pts" in a smaller window scope for the code vulnerability detection model to focus on the contexts related to "pts." In this way, we can focus more on the semantic information related to the vulnerability itself without the interference of irrelevant information as much as possible.

In order to select the optimal parameters, we experimented with several parameters of node2vec. Node2vec contains some sampling policy parameters, specifically *walk_length*, *num_walks*, *p*, *q* and *window_size*, which separately



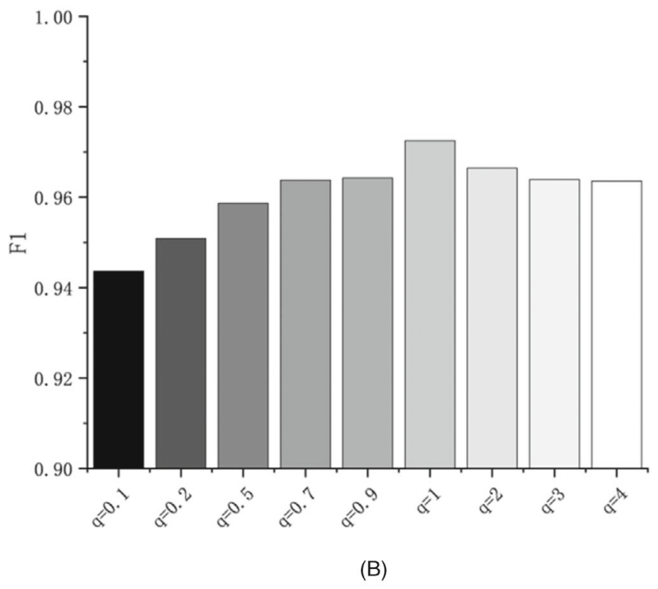


FIGURE 13 The effect of parameters p and q on the effect of VulGraB. (A) The effect of parameter p in node2vec. (B) The effect of parameter q in node2vec.

represent the walk length, number of walks, probability of visiting the previous node, the direction of the walk biased depth-first or (depth-first for $q < 1$, breadth-first for $q > 1$) and window size. The CWE-78 dataset with balanced positive and negative samples is chosen for this experiment so that the experimental results are not affected by the sample distribution.

To control the variables, *walk_length*=5, *num_walks*=10, q=1, *window_size*=5 are chosen here to study the effect of parameter p on the effect of the model. The experimental results are shown in Figure 13A, where the vertical coordinate is F1. It can be observed that the minimum F1 of 95.17% is obtained for $p = 0.6$, and the maximum F1 of 97.24% is obtained for $p = 0.9$. After repeated experiments, $p = 0.9$ is the only parameter choice with F1 exceeding 97%. In addition, the F1 fraction is about 96.6% for $p = 2$, $p = 3$, and $p = 4$ in this experiment.

The parameter $p = 0.9$, which has the best effect in the above experiments, is selected, and the effect of parameter q on the model effect is studied by keeping other parameters constant. The experimental results are shown in Figure 13B. The minimum F1 of 94.36% is obtained for $q = 0.1$, and the maximum F1 of 97.24% is obtained for $q = 1$. After repeated experiments, $q = 1$ is the only parameter choice for which F1 exceeds 97%. This reflects that depth-first is as important as breadth-first when node2vec extracts graph structure features.

The parameters $p = 0.9$ and $q = 1$, which have the best effect in above experiments, are set, and the effect of the parameter *walk_length* on the model effect is studied while keeping other parameters constant. The experimental results are shown in Figure 14A. The maximum F1 is 97.24% for *walk_length*=5. After repeated experiments, $q = 1$ is the only choice for which F1 exceeds 97%. It indicates that a walk length of 5 is optimal when node2vec performs random walks.

Taking the parameters $p = 0.9$, $q = 1$, *walk_length*=5, and keeping *window_size*=5 constant in the above experiments, the effect of the parameter *num_walks* on the effect of the model is studied. The experimental results are shown in Figure 14B, where the maximum F1 is 97.24% for *num_walks*=10. The F1 also exceeds 97% for *num_walks*=15 and 20. This indicates that the best number of walks is 10 when node2vec performs random walks.

The parameters $\mathrm{p} = 0.9, \mathrm{q} = 1$, *walk_length*=5, and *num_walks*=10, which have the best effect in the above experiments, are selected to study the effect of parameter *window_size* on the model effect. The experimental results are shown in Figure 14C, where it can be observed that the maximum F1 is 97.24% for *window_size*=5. F1 also exceeds 97% for *window_size*=10. This indicates that a window of 5 is optimal when node2vec is used for feature training.

In summary, $p = 0.9$ means that for a node in a code PDG graph, its predecessor node is important for the semantic information of that node, and $q = 1$ means that deep random wandering and breadth random wandering are of equal importance. Besides, the best result of VulGraB is achieved when the wander length is 5, the number of wanderings is 10, and the window size is 5.

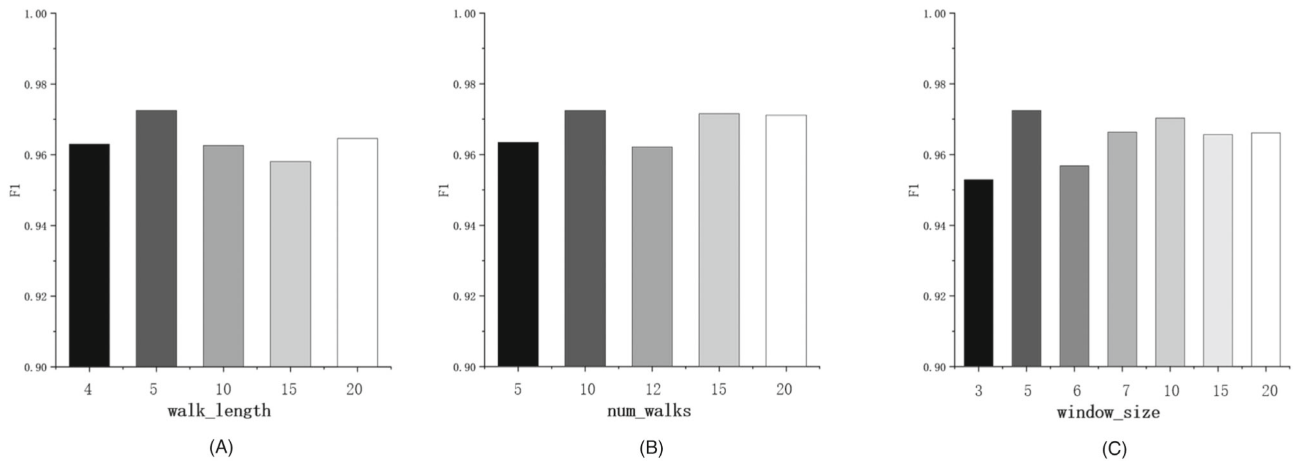


FIGURE 14 The effect of parameters *walk_length*, *num_walks* and *window_size* on the effect of VulGraB. (A) The effect of parameter *walk_length* in node2vec. (B) The effect of parameter *num_walks* in node2vec. (C) The effect of parameter *window_size* in node2vec.

Insight: In the field of embedding technology, node2vec is more effective than word2vec in semantic learning of vulnerable code. Specifically, VulGraB works best when $p = 0.9$, $q = 1$, *walk_length*=5, *num_walks* =10, and *window_size*=5 in node2vec.

RQ4: Among the 9 vulnerability types in FLL, in which ones does VulGraB work best?

In the experimental results of RQ1, although VulGraB has the best performance in the FLL dataset, the effect is still significantly lower than that of the CWE-78 dataset. We would like to investigate this data for discussion. This is because the FLL dataset contains nine types of vulnerabilities with different characteristics. For example, buffer vulnerabilities need to be judged by combining buffer size with stored data size, and null pointer vulnerabilities need to be judged by combining null pointer location and specific trigger conditions, while the CWE-78 dataset contains only one type of vulnerability, and this type does not reasonably check the command line entered by the user and has a relatively single pattern. Also, the CWE-78 dataset contains more samples than the FLL dataset, and its features are easier to be fully captured and learned by the neural network, thus achieving a better detection effect.

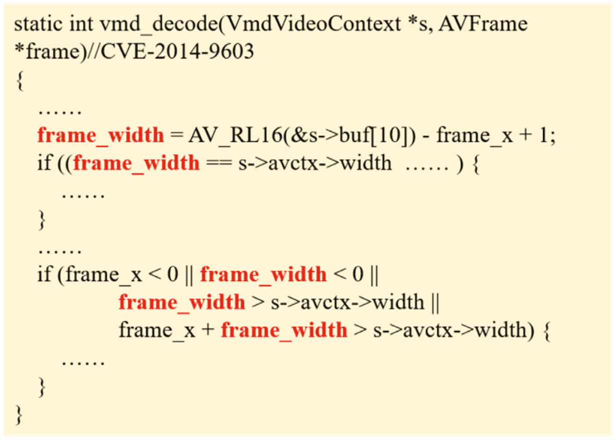
To study which type of vulnerability is better detected by VulGraB among the 9 types of vulnerabilities on the FLL dataset, we separate the 9 types of vulnerabilities in FLL, and the number of separations is shown in Table 6. The number of buffer errors is the largest, accounting for 39% of the entire dataset, while format string vulnerability accounts for only 3%. Considering that the experimental samples of some vulnerability types are too small and unfavorable for the experiments, we conduct experiments on the types with more than 30 positive samples, and

TABLE 6 Number of 9 vulnerability types in FFmpeg, LibPNG, LibTIFF datasets.

Vulnerability types	FFmpeg	LibPNG	LibTIFF	Total
Buffer errors	79	12	37	128
Resource management errors	36	8	16	60
Format string vulnerability	3	2	4	9
Numeric errors	20	5	8	33
Division by zero	4	2	6	12
Input validation errors	26	6	5	37
Null-pointer dereference	7	3	2	12
Information leak	5	2	8	15
Integer overflow	12	4	8	24
Total	192	44	94	330

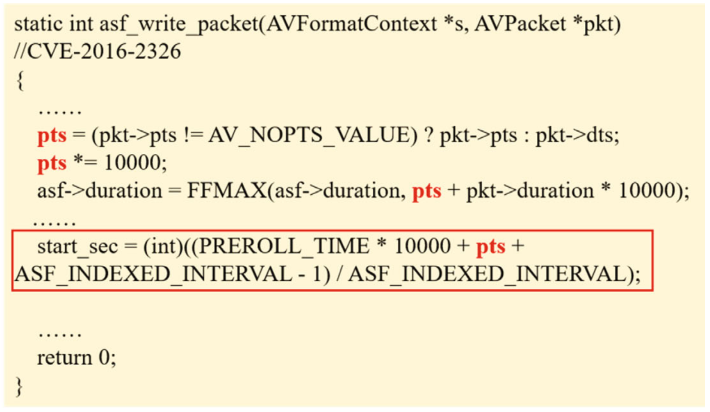
TABLE 7 Results of VulGraB on 5 single types of vulnerabilities.

Vulnerability types	Number of positive samples	Number of negative samples	FPR	FNR	Recall	P	F1
Buffer errors	128	130	2.31	1.56	98.44	97.67	98.05
Resource management errors	60	60	15.00	21.67	78.33	83.93	81.03
Numeric errors	33	35	94.29	48.48	51.52	34.00	40.96
Input validation errors	37	40	2.50	5.41	94.59	97.22	95.89
CWE-78	6101	6101	2.81	2.72	97.28	97.19	97.24



```
static int vmd_decode(VmdVideoContext *s, AVFrame
*frame)//CVE-2014-9603
{
  ……
  frame_width = AV_RL16(&s->buf[10]) - frame_x + 1;
  if ((frame_width == s->avctx->width  …… ) {
      ……
  }
  ……
  if (frame_x < 0 || frame_width < 0 ||
          frame_width > s->avctx->width ||
          frame_x + frame_width > s->avctx->width) {
      ……
  }
}
```

(A)



```
static int asf_write_packet(AVFormatContext *s, AVPacket *pkt)
//CVE-2016-2326
{
  ……
  pts = (pkt->pts != AV_NOPTS_VALUE) ? pkt->pts : pkt->dts;
  pts *= 10000;
  asf->duration = FFMAX(asf->duration, pts + pkt->duration * 10000);
  ……
  start_sec = (int)((PREROLL_TIME * 10000 + pts +
ASF_INDEXED_INTERVAL - 1) / ASF_INDEXED_INTERVAL);

  ……
  return 0;
}
```

(B)

FIGURE 15 Input validation error and numeric error examples. (A) Input validation error example in the FLL dataset. (B) Numeric error example in the FLL dataset.

the experimental results are shown in Table 7. VulGraB achieves the best results in buffer errors type with an F1 value of 98.05%, followed by input validation errors with an F1 value of 95.89%, while the worst result is numeric errors with an F1 value of only 41%. This is because the buffer errors are related to the buffer data, whose semantic information is relatively independent and easy to be captured by the deep learning model, and there are relatively enough samples for VulGraB to learn. Input validation errors are flawed because the validity of the function input is not verified, and the learning model only needs to track the semantic information of the function inputs and user input data.

Figure 15A shows an input validation error in the FLL dataset, which does not validate the relationship between a certain length value and the *frame_width*, allowing remote attackers to cause a denial of service. Numeric errors are related to incorrect numeric computations or conversions, which involve many subtle types of vulnerabilities, such as integer overflow, numeric truncation errors, operator errors and so on. Figure 15B shows an integer overflow vulnerability[26] in the FLL dataset, which is a numeric error. Numeric errors in a line of code for multiple variables may involve the control flow and data flow information of multiple variables, the information related to the variables is complex. For example, the mathematical code in the red box involves four variables, and if we were to trace the information related to "pts," it would involve the calculation of three more formulas. Also, the experimental sample of this type is small, only 33, so it is not being effectively learned, resulting in relatively poor experimental results. Few researchers have studied for numeric errors, and in future work, we will further investigate such vulnerabilities. In addition, the data set CWE-78 in this article is a kind of OS command injection type vulnerability, and VulGraB is also good at detecting such vulnerabilities.

Insight: VulGraB does well in buffer errors, input validation errors, OS command injection type vulnerabilities. Vulnerability detection in the field of numeric errors is a field worth exploring.

5 | THREATS TO VALIDITY

VulGraB is a novel method for code vulnerability detection, however, it still has some limitations for future work and further improvement.

1. We have only experimented with VulGraB on two datasets involving 10 vulnerability types, where the relatively small number of code vulnerabilities of the nine types in the FLL dataset makes our method insufficient to fully learn the characteristics of these vulnerabilities. The lack of labeled data leads to lower performance and reliability of the detection model, which is a widespread problem in the field of code vulnerabilities. We will try to build datasets based on more real and high-quality project codes and study more types and larger number of vulnerability datasets in the future.
2. We focus on detecting vulnerabilities in C/C++ program source code, meaning that the framework may need to be adapted to cope with other programming languages, such as Java, PHP. In the longer term, we will explore the similarities and differences of code vulnerabilities in different languages in order to adapt the detection model to better fit different languages.
3. We detect vulnerabilities at the function level, which may be coarse-grained. When a function block contains too many lines of code, it can be difficult for security personnel to locate where the vulnerabilities are located. This can be improved, and in future work we will detect code vulnerabilities at a fine-grained level to more precisely identify the lines of code that contain vulnerabilities.
4. In the work of this article, for each dataset, we need to train it once to detect vulnerabilities in the target program that are similar to the pattern of the dataset. This can be expensive and time-consuming when the number of datasets is too large. Therefore, we consider future work to investigate transfer learning methods in the field of code vulnerabilities, where researchers are able to reuse vulnerability detection models in different software projects. In the long run, this increases the possibility of a common vulnerability detection model applied to several different software projects.
5. From our experimental results, VulGraB can detect several single-type vulnerabilities very well, such as OS command injection and buffer errors, but it does not perform as well as single-type on multi-type vulnerability datasets. However, real-world projects often contain multiple types of vulnerabilities. In the future, we will investigate how to automatically detect multiple types of code vulnerabilities in projects.
6. We have validated the effectiveness of VulGraB using five common evaluation criteria, however, new evaluation criteria have recently been proposed, such as BTP (an evaluation metrics using bug-triggering paths),[27] which measures the trigger path of bugs at the method level, slice level and statement level of granularity. We will apply this criterion in the future to assess the effectiveness of our work.

6 | RELATED WORK

Emerging deep learning techniques offer new potential in the field of code vulnerability detection.[28] On the one hand, the hierarchical structure of neural network-based models helps facilitate the learning of abstract and highly nonlinear patterns, which can capture the intrinsic structure of complex data. On the other hand, neural networks can automatically extract features and have multiple levels of abstraction and higher generalization capabilities, thus freeing experts from the labor-intensive and potentially error-prone task of feature definition. In addition, deep learning methods are able to discover potential features that human experts may never consider, greatly expanding the feature search space.[29] Deep-learning based code vulnerability detection techniques can be roughly divided into three parts, which are code representation, feature vector generation, and training of deep learning neural models. The state of art on each of these three parts is described below.

6.1 | Code representation

Deep learning-based vulnerability detection methods can be divided into learning of four main types of code representations: sequence-based representations, text-based representations (source code text), graph-based representations (AST, CFG, PDG, etc.), and hybrid representations based on all three.[15,30]

In the work on sequence-based representations, Grieco et al.[31] propose an approach that uses lightweight features extracted from static and dynamic analysis to predict memory corruption vulnerabilities in operating system-level programs represented in binary. The research work based on textual representation does not use any code analysis method. They directly use code text as input for learning feature representations and expect expressive neural models to learn potentially vulnerable code semantics hidden in the surface text of the code. In the work on code-based hybrid representation, the framework proposed by Lin et al.[32] use two Bi-LSTM networks to obtain feature representations from two different types of hybrid data sources, AST and source code, to capture more vulnerability features.

However, the sequence-based and text-based representations are completely one-dimensional compared to the graph-based representation, and they only allow the neural model to capture relatively single stream patterns and features. Although hybrid feature representations may enable neural networks to capture more vulnerability features,[33] their resources and time for data processing also increase exponentially, limiting the advancement of research to some extent. Therefore, graph-based representation is the current mainstream representation, which can be used to capture complex program structures yet consumes relatively less resources and time.[28] A large number of studies have applied neural networks to learn feature representations from different types of graph-based program representations, including abstract syntax trees, control flow graphs, program dependency graphs, and combinations of them. Buch et al.[34] proposed a recursive neural network based on AST for code clone detection, where AST is then used to represent static source code. In the SySeVR system,[8] the authors first extract the syntactic information of the source code using the AST, and then extract the semantic information of the source code using the CFG and DFG of the source code.

Some work has been done on a flow-based basis by Xiao Cheng et al.[35] They study for high-level control flow related (CFR) vulnerabilities and proposed VGDETECTOR to automatically and statically detect CFR vulnerabilities without knowing the explicitly defined patterns of such vulnerabilities. They also proposed Contra Flow,[36] a selective but accurate contrastive value flow embedding method for static detection of software vulnerabilities. Its novelty lies in selecting and preserving feasible value flow paths by using a pre-trained path embedding model with self-supervised contrast learning, thus reducing the amount of labeled data required to train expensive downstream path-based vulnerability detection models.

The AST is too detailed in its representation of the code, and it is mainly used to represent the direct syntactic information of the code token, while losing some coarse-grained semantic information. And if the information of CFG alone or DDG (data dependence graph) alone is used, the semantic information is not comprehensive enough and will cause false positives and false negatives.[17] In summary, the PDG that combines CFG and DDG is the best choice for semantic representation.

6.2 | Generating feature vectors

When we want to better represent, and characterize vulnerabilities, we need to apply effective embedding techniques. When transforming code into feature vector representations, the goal is to preserve as much of the semantics of the source data as possible so that deep learning models can learn richer, more expressive semantics. A large body of existing work relies on frequency-based (e.g., n-gram models) and prediction-based (e.g., word2vec models) word embedding solutions.[37,38] Frequency-based embedding techniques generate word embeddings based on the frequency of words occurring in a given context, thus capturing a limited number of word semantics. Prediction-based techniques, such as word2vec, can learn word semantics from a relatively small contextual window, but cannot capture the interdependent meaning of the target word and its entire context. Therefore, a more efficient embedding technique that provides richer semantic information to the learning system is needed.

Word2vec is a feature vector embedding technique, which is designed for natural language processing (NLP).[39] Given the linear nature of text in NLP, the notion of neighborhood can be defined naturally using sliding windows on consecutive words. However, the structure of the code graph is not linear and thus requires a richer notion of neighborhood. Unlike word2vec, node2vec can capture nonlinear neighborhood information. Node2vec first samples a set of paths from the input graph using a predefined random wandering strategy. The skip-gram algorithm is then applied on the paths to maximize the probability of observing the node neighborhoods when computing the node embeddings. In summary, the key contribution of the node2vec embedding technique is that the concept of node neighborhood is flexibly defined by providing a trade-off between breadth-first sampling (BFS) and depth-first sampling (DFS) graph search strategies.[13]

Yulei Sui et al.[40] have considered and experimented with more fine-grained embedding techniques for value-flow. For the problem of insensitivity to most existing pointer analyses, they propose SUPA (Strong UPdate Analysis), which aim to find the point-to relationships between dereferenced function pointers and address-taken functions. In addition, they propose Flow2Vec,[41] a new code embedding approach that precisely preserves inter-procedural program dependence.

6.3 | Deep learning models

Network models for vulnerability detection are becoming more sophisticated and expressive in order to better learn the code semantics of vulnerable code fragments, and to reduce the amount of effort required for code analysis efforts. Research has evolved from early applications of multi-layer perceptron (MLP) to more recent studies using CNNs or LSTMs,[42,43] and until recently using GNNs-based approaches.[44] The evolving network structure suggests that researchers have invested a significant amount of research effort to explore the potential of neural networks for semantic reasoning about code, as well as for facilitating rich patterns of vulnerability discovery. Researchers have also attempted to adopt state-of-the-art code analysis tools and methods for vulnerability detection, inspired by the fields of machine learning and NLP.[45,46]

With the goal of automatic vulnerability detection, Duan et al.[47] proposed a software vulnerability detection method based on attentional bidirectional LSTM neural networks. Zhou et al.[48] proposed Devign, a GNN vulnerability detection model based on composite code representation, which learns node features by aggregating information from neighboring nodes from GNN and finally selects, from the Conv module, the nodes and feature sets relevant to the current task, and perform graph-level classification detection by one-dimensional convolution and dense layers. Shar et al.[49] applied MLP, which manually extracts features from CFG and DDG for learning input validation and processing code patterns. Later studies used CNNs, which can learn features directly from complex source code with less code analysis effort required, while memory network-based studies[50] used source code as input without any code analysis. These neural networks usually consider only a unidirectional flow and are limited to local semantic information, which is not sufficient to fully learn the global code semantics.

GNNs-based vulnerability detection models are able to learn information about the matching relationship between source code relational paths and node semantics, and it has shown its superiority in code vulnerability detection tasks.[10,30] Among them, GGNN has the best results.[11] As an extension of GGNN, researchers have recently proposed BiGGNN for the field of natural problem generation. Some researchers have demonstrated that BiGGNN has also shown its effectiveness in the task of code semantic learning.[51] LeClair A used it for the task of automatic source code documentation generation[52] and Wu et al. used it for capturing code commit structure to enhance security patch identification.[53]

In general, the related work cannot extract feature from the nonlinear semantic information in the code graph structure and do not use a suitable enough neural network to learn the global features of PDG. In the field of code vulnerability detection, no one has done research with node2vec and BiGGNN based on deep learning methods.

6.4 | Other related work

Unlike the generic detection methods mentioned above, Yin et al.[54] propose a static memory safety analysis method called SafeOSL. The C program to be checked is first transformed into an OSL program and then detected by OSL semantics. SafeOSL is demonstrated to be effective in detecting memory errors in the C language.

In addition, some researchers have made some new attempts inspired by the field of code cloning to detect recurring code vulnerabilities by identifying whether the target code is a clone of vulnerable code or patch code. Yang Xiao et al.[55] proposed MVP to extract vulnerability and patch signatures at the syntactic and semantic levels from vulnerable functions and their patching functions. Then, if the target function matches the vulnerability signature but not the patch signature, it is identified as a potentially vulnerable function. Seunghoon Woo et al.[56] identify vulnerable code clones by considering the oldest vulnerable functions and extracting only the core vulnerable and patch lines from the security patches. Akram et al.[57] proposed a vulnerability detection method based on code clone detection techniques and detected hundreds of vulnerabilities in thousands of GitHub open-source projects that have not been previously noticed as vulnerable.

In general, focusing the scope of vulnerability detection on advanced vulnerability types is promising as some complex vulnerabilities are difficult to learn and detect. In addition, we can take inspiration from other areas of code analysis to further innovate our work on code vulnerability detection.

7 | CONCLUSION

To address the problem of insufficient expression of nonlinear information in the code graph structure in the field of code vulnerability detection, this article proposes VulGraB, a vulnerability detection method based on graph embedding and bidirectional gated GNN. VulGraB first uses existing tools to construct a program dependency graph, then extracts key information of nodes and directed edges from the program dependency graph, learns textual features and edge features of nodes using improved node2vec, and finally uses a bidirectional gated GNN for feature learning, in which the bidirectional cyclic structure learns the outward and inward features between nodes and neighboring nodes. The evaluation of VulGraB proposed in two vulnerability datasets shows that it can be successfully used for code vulnerability detection tasks and can accurately detect vulnerabilities in the target program.

ACKNOWLEDGMENTS

The work was supported in part by the Natural Science Foundation of Zhejiang Province under Grant LY23F020016 and Grant LY21F020020, in part by the National Natural Science Foundation of China under Grant 61902096, in part by the Key Research and Development Program of Zhejiang Province under Grant 2023C03200.

AUTHOR CONTRIBUTIONS

Sixuan Wang: Conceptualization, Methodology, Investigation, Data curation, Writing – Original Draft, Writing – review & editing. **Chen Huang**: Methodology, Investigation, Data curation, Writing – Original Draft, Writing-review & editing. **Dongjin Yu**: Conceptualization, Supervision, Writing – review & editing. **Xin Chen**: Conceptualization, Validation, Writing – review & editing.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available in CWE-78 at https://github.com/HuantWang/FUNDED_NISL/tree/main/FUNDED/data/data/CWE-77.zip, reference number 25.

ORCID

Sixuan Wang https://orcid.org/0000-0002-1389-2488

REFERENCES

1. GitHub. https://octoverse.github.com/.
2. OpenSSL. https://www.openssl.org/.
3. Heartbleed. https://github.com/.
4. Kim S, Woo S, Lee H, Oh H. VUDDY: a scalable approach for vulnerable code clone discovery. Proceedings of the 2017 IEEE Symposium on Security and Privacy; 2017:595-614.
5. Zhang M, de Carnavalet XDC, Wang L, Ragab A. Large-scale empirical study of important features indicative of discovered vulnerabilities to assess application security. *IEEE Trans Inf Forensic Secur*. 2019;14(9):2315-2330.
6. Wannacry. https://en.wikipedia.org/wiki/WannaCry_ransomware_attack.
7. Standards N. National vulnerability database; 2011.
8. Li Z, Zou D, Xu S, Jin H, Zhu Y, Chen Z. SySeVR: a framework for using deep learning to detect software vulnerabilities. *IEEE Trans Dependable Secure Comput*. 2021;19:2244-2258.
9. Bowman B, Huang HH. VGRAPH: a robust vulnerable code clone detection system using code property triplets. Proceedings of the 2020 IEEE European Symposium on Security and Privacy; 2020:53-69.
10. Ghaffarian SM, Shahriari HR. Neural software vulnerability analysis using rich intermediate graph representations of programs. *Inf Sci*. 2021;553:189-207.
11. Chakraborty S, Krishna R, Ding Y, Ray B. Deep learning based vulnerability detection: are we there yet? *IEEE Trans Softw Eng*. 2021;48:3280-3296.
12. Tian J, Xing W, Li Z. BVDetector: a program slice-based binary code vulnerability intelligent detection system. *Inf Softw Technol*. 2020;123:106289.

13. Grover A, Leskovec J. node2vec: scalable feature learning for networks. Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining; 2016:855-864.
14. Chen Y, Wu L, Zaki MJ. Reinforcement learning based graph-to-sequence model for natural question generation. Proceedings of the 8th International Conference on Learning Representations; 2020.
15. Shen Z, Chen S. A survey of automatic software vulnerability detection, program repair, and defect prediction techniques. *Secur Commun Netw*. 2020;2020:1-16.
16. Letychevskyi O, Hryniuk Y. Machine learning methods for improving vulnerability detection in low-level code. Proceedings of the 2020 IEEE International Conference on Big Data; 2020:5750-5752.
17. Li Y, Wang S, Nguyen TN. Vulnerability detection with fine-grained interpretations. Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering; 2021:292-303.
18. Wang Z, Zheng Q, Sun Y. GVD-net: graph embedding-based machine learning model for smart contract vulnerability detection. Proceedings of the 2022 International Conference on Algorithms, Data Mining, and Information Technology; 2022:99-103.
19. Cheng X, Wang H, Hua J, Xu G, Sui Y. Deepwukong: statically detecting software vulnerabilities using deep graph neural network. *ACM Trans Softw Eng Methodol*. 2021;30(3):1-33.
20. Russell R, Kim L, Hamilton L, et al. Automated vulnerability detection in source code using deep representation learning. Proceedings of the 2018 17th IEEE International Conference on Machine Learning and Applications; 2018:757-762.
21. Maiorca D, Biggio B. Digital investigation of pdf files: unveiling traces of embedded malware. *IEEE Secur Priv*. 2019;17(1):63-71.
22. Common Weakness Enumeration, 2020, https://cwe.mitre.org/data/index.html.
23. Liu S, Lin G, Qu L, et al. CD-VulD: cross-domain vulnerability discovery based on deep domain adaptation. *IEEE Trans Dependable Secure Comput*. 2020;19:438-451.
24. Li Z, Zou D, Xu S, et al. Vuldeepecker: a deep learning-based system for vulnerability detection. Proceedings of the Network and Distributed Systems Security (NDSS) Symposium 2018; 2018.
25. Wang H, Ye G, Tang Z, et al. Combining graph-based learning with automated data collection for code vulnerability detection. *IEEE Trans Inf Forensic Secur*. 2020;16:1943-1958.
26. CVE, https://cve.mitre.org/.
27. Cheng X, Nie X, Li N, Wang H, Zheng Z, Sui Y. How about bug-triggering paths? Understanding and characterizing learning-based vulnerability detectors. *IEEE Trans Dependable Secure Comput*. 2022;14(8):1-18.
28. Gao H, Wu L, Hu P, Wei Z, Xu F, Long B. Graph-augmented learning to rank for querying large-scale knowledge graph. Proceedings of the 2nd Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics and the 12th International Joint Conference on Natural Language Processing; 2022:82-92.
29. Wu F, Wang J, Liu J, Wang W. Vulnerability detection with deep learning. Proceedings of the 3rd IEEE International Conference on Computer and Communications; 2017:1298-1302.
30. Zhuang Y, Suneja S, Thost V, Domeniconi G, Morari A, Laredo J. Software vulnerability detection via deep learning over disaggregated code graph representation. *CoRR* abs/2109.03341; 2021.
31. Grieco G, Grinblat GL, Uzal L, Rawat S, Feist J, Mounier L. Toward large-scale vulnerability discovery using machine learning. Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy; 2016:85-96.
32. Lin G, Zhang J, Luo W, et al. Software vulnerability discovery via learning multi-domain knowledge bases. *IEEE Trans Dependable Secure Comput*. 2019;18(5):2469-2485.
33. Gong X, Xing Z, Li X, Feng Z, Han Z. Joint prediction of multiple vulnerability characteristics through multi-task learning. Proceedings of the 2019 24th International Conference on Engineering of Complex Computer Systems; 2019:31-40.
34. Buch L, Andrzejak A. Learning-based recursive aggregation of abstract syntax trees for code clone detection. Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering; 2019:95-104.
35. Cheng X, Wang H, Hua J, et al. Static detection of control-flow-related vulnerabilities using graph embedding. Proceedings of the 2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS); November 2019:41–50.
36. Cheng X, Zhang G, Wang H, Sui Y. Path-sensitive code embedding via contrastive learning for software vulnerability detection. Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis; 2023.
37. Mikolov T, Chen K, Corrado G, Dean J. Efficient estimation of word representations in vector space. ICLR Workshop Track Proceedings; 2013.
38. Mikolov T, Sutskever I, Chen K, Corrado GS, Dean J. Distributed representations of words and phrases and their compositionality. Proceedings of the 26th International Conference on Neural Information Processing Systems, vol. 26; 2013.
39. Yang W, Li L, Zhang Z, Ren X, Sun X, He B. Be careful about poisoned word embeddings: exploring the vulnerability of the embedding layers in NLP models. *CoRR* abs/2103.15543; 2021.
40. Sui Y, Xue J. Value-flow-based demand-driven pointer analysis for C and C++. *IEEE Trans Softw Eng*. 2020;46(8):812-835.
41. Sui Y, Cheng X, Zhang G, Wang H. Flow2Vec: value-flow-based precise code embedding. *Proc ACM Program Lang*. 2020;4:1-27.
42. Guo J, Wang Z, Li H, Xue Y. Detecting vulnerability in source code using CNN and LSTM network. *Soft Comput*. 2021;27:1131-1141.
43. Lee YJ, Choi SH, Kim C, Lim S, Park K. Learning binary code with deep learning to detect software weakness. Proceedings of the 9th International Conference on Internet (ICONI) 2017 Symposium; 2017.
44. Feng Q, Feng C, Hong W. Graph neural network-based vulnerability predication. Proceedings of the 2020 IEEE International Conference on Software Maintenance and Evolution; 2020:800-801.
45. Wu J. Literature review on vulnerability detection using NLP technology. *CoRR* abs/2104.11230, 2021.

46. Bilgin Z, Ersoy MA, Soykan EU, Tomur E, Çomak P, Karaçay L. Vulnerability prediction from source code using machine learning. *IEEE Access*. 2020;8:150672-150684.
47. Duan X, Wu J, Ji S, et al. VulSniper: focus your attention to shoot fine-grained vulnerabilities. Proceedings of the 28th International Joint Conference on Artificial Intelligence; 2019:4665-4671.
48. Zhou Y, Liu S, Siow J, Du X, Liu Y. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. Proceedings of the 33rd International Conference on Neural Information Processing Systems, vol. 32; 2019.
49. Shar LK, Tan HBK. Predicting common web application vulnerabilities from input validation and sanitization code patterns. Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering; 2012:310-313.
50. Choi M, Jeong S, Oh H, Choo J. End-to-end prediction of buffer overruns from raw source code via neural memory networks. Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence (IJCAI-17); 2017:1546-1553.
51. Liu S, Xie X, Ma L, Siow J, Liu Y. GraphSearchNet: Enhancing GNNs via capturing global dependency for semantic code search. *CoRR* abs/2111.02671, 2021.
52. Alexander L. *Neural Models of Automated Documentation Generation for Source Code*. Dissertation. University of Notre Dame; 2022.
53. Wu B, Liu S, Feng R, Xie X, Siow J, Lin SW. Enhancing security patch identification by capturing structures in commits. *IEEE Trans Dependable Secure Comput*. 2022;14(8):1-15.
54. Yin X, Huang Z, Kan S, et al. SafeOSL: ensuring memory safety of C via ownership-based intermediate language. *Softw Pract Exp*. 2022;52(5):1114-1142.
55. Xiao Y, Chen B, Yu C, et al. MVP: detecting vulnerabilities using patch-enhanced vulnerability signatures. Proceedings of the 29th USENIX Security Symposium; August 12–14, 2020.
56. Woo S, Hong H, Choi E, Lee H. MOVERY: a precise approach for modified vulnerable code clone discovery from modified open-source software components. Proceedings of the 31st USENIX Security Symposium; August 10–12, 2022.
57. Akram J, Luo P. SQVDT: a scalable quantitative vulnerability detection technique for source code security assessment. *Softw Pract Exp*. 2021;51:294-318.

How to cite this article: Wang S, Huang C, Yu D, Chen X. VulGraB: Graph-embedding-based code vulnerability detection with bi-directional gated graph neural network. *Softw: Pract Exper*. 2023;53(8):1631-1658. doi: 10.1002/spe.3205