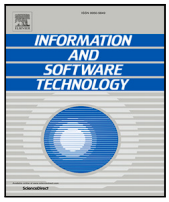
Information and Software Technology 158 (2023) 107168

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof



SedSVD: Statement-level software vulnerability detection based on Relational Graph Convolutional Network with subgraph embedding

Yukun Dong *, Yeer Tang, Xiaotong Cheng, Yufei Yang, Shuqi Wang

Qingdao Institute of Software,College of Computer Science and Technology, China University of Petroleum(East China), Shandong, China

ARTICLE INFO

Keywords:
Software vulnerability detection
Code property graph
Relational graph convolutional network
Subgraph embedding
Statement-level detection

ABSTRACT

Context: Current deep-learning based vulnerability detection methods have been proven more automatic and correct to a certain extent, nonetheless, they are limited to detect at function-level or file-level, which can hinder software developers from acquiring more detailed information and conducting more targeted repairs. Graph-based detection methods have shown dominant performance over others. Unfortunately, the information they reveal has not been fully utilized.

Objective: We design SedSVD (Subgraph embedding driven Statement-level Vulnerability Detection) with two objectives: (i) to better utilize the information the code-related graphs can reflect; (ii) to detect vulnerabilities at a finer-grained level.

Method: In our work, we propose a novel graph-based detection framework that embeds graphs at subgraph-level to realize statement-level detection. It first leverages Code Property Graph (CPG) to learn both semantic and syntactic information from source code, and then selects several center nodes (code elements) in CPG to build their subgraphs. After embedding each subgraph with its nodes and edges, we apply Relational Graph Convolutional Network (RGCN) to process different edges differently. A Multi-Layer Perceptron (MLP) layer is further added to ensure its prediction performance.

Results: We conduct our experiments on C/C++ projects from NVD and SARD. Experimental results show that SedSVD achieves 95.15% in F1-measure which proves our work to be more effective.

Conclusion: Our work detects at a finer-grained level and achieves higher F1-measure than existing state-of-art vulnerability detection techniques. Besides, we provide a more detailed detection report pointing the specific error code elements within statements.

1. Introduction

Software vulnerabilities have long been a headache for software developers since they are the major factor resulting in security flaws and cyber attack [1]. Therefore, how to identify and repair them is extremely crucial yet challenging. Traditional detecting methods like static analysis, dynamic analysis, symbolic execution and so on have achieved promising improvements. However, since these methods are not effective enough and require intense human labor, we call for more efficient and automated approaches.

In recent years, deep learning-based vulnerability detection has attracted a surge of interest. The insight is that through embedding the source code into vectors, which are then taken as the input of a certain deep learning network, we are allowed to judge whether this program is vulnerable or not according to the output (0 represents non-vulnerable and 1 represents vulnerable, for example). Hence, code embedding and deep learning networks undoubtedly become the two key parts in this area.

As for code embedding, there are methods based on sequence, text, Abstract Syntax Tree (AST), and graph [2]. Sequence-based methods [3] extract code features like identifiers, characters, tokens and so forth [4]. Text-based methods [5], however, treat source code as texts and quantify feature words extracted from them to represent text information. While AST-based techniques [6,7] adopt AST to extract hierarchical information of source code. Graph-based methods [8,9] employ code-related graphs to extract information, which allow for more effective abstraction of deep code features.

As for the network applied, Convolutional Neural Network (CNN) [10] and Recurrent Neural Network (RNN) [11] or RNN-variants are the most commonly used [12,13] in sequence-based and text-based methods. TextCNN [14], TextRNN [15] and TextRCNN [16] are also common in text-based approaches. Whereas, for AST-based ones, neural networks with tree-structure are more suitable [17,18]. Apparently, for graph-based ones, neural networks on graphs are the best

* Corresponding author.
E-mail address: dongyk@upc.edu.cn (Y. Dong).

https://doi.org/10.1016/j.infsof.2023.107168
Received 20 September 2022; Received in revised form 11 January 2023; Accepted 2 February 2023
Available online 8 February 2023

choice [19] like Gated Graph Neural Networks (GGNN) [20], Bilinear Graph Neural Network (BGNN) [21] and Gated Graph Recurrent Network (GGRN) [20].

However, there do exist some limitations in previous approaches. First of all, sequence-based and text-based methods treat source code as simple as natural language or plain text which will certainly discard the structural and logical information of source code [5,22]. Therefore, code-related graphs used in graph-based methods, which are considered as a comprehensive code representation, are more and more popular.

Graph-based methods, however, are inclined to ignore edge information, like edge direction and edge type. Devign [2] and ReVEAL [23] employ Gate Recurrent Unit (GRU) [24] to conduct neighborhood aggregation but neglected edge embedding. BGNN4D [25] takes edge direction into consideration and adopts BGNN to process both forward edges and backward edges. Zhuang, Y., et al. [26] embeds information and adopted Crystal Graph Convolutional Neural Network (CGCN) [27] to process different edge information. Both of them have made efforts but their performance is relatively low.

Next, most existing methods detect vulnerabilities at file-level or function-level [28,29], which is too coarse to provide more detailed information to assist in defect repair. VulDeepecker [30] and SySeVR [1] judge whether a set of source statements (code gadget or slice) is vulnerable or not. VulDeeLocator [31] indicates where the vulnerability is between two source statements. They have refined the detecting level, but we still pursue approaches detecting at a finer-grained level which has not yet received satisfactory effects [4,32,33].

To cope with these problems, in this paper, we propose a novel graph-based model detecting software vulnerabilities at statement-level. We first transform source code into CPG [34] to gain semantic and syntactic information. Through CPG, code elements are parsed into nodes, and relations between them are parsed into edges. To refine our detection level, we embed graphs at subgraph-level. Besides, to improve efficiency and put more attention on vulnerabilities themselves and their context, we will first select several center nodes which are more related to vulnerability and then build their subgraphs by collecting all its neighbor nodes and connected edges among them. For nodes, we embed their type with label encoding and code with Word2vec [35]. For edges, we embed both edge direction and edge type. After that, all subgraphs are learned and embedded as the input of graph-based neural network. In this method, we adopt RGCN [36], a heterogeneous graph neural network, to process graph data. After that, an MLP layer is added to conduct classification. Thus, for a certain subgraph, the model will judge whether each node in it is vulnerable or not, and then provide a vulnerability report for vulnerable nodes including corresponding error codes, error line numbers, error type and so on. We name this model as SedSVD (Subgraph embedding driven Statement-level Vulnerability Detection).

Our contributions are as follows:

- We embed graphs at a more proper granularity. Instead of graph-level or node-level which is mostly used in graph-related tasks, we embed graph at sub-graph-level. For vulnerability detection, graph-level is too coarse to provide specific information and node-level is too detailed and may isolate each node without locating them in the whole graph.
- We generate more accurate ground truth labels. Different from previous work which simply label a code element as vulnerable if it is in a deleted line, we make further refinement.
- We detect vulnerabilities at a finer-grained level. By splitting a single statement into several nodes and conducting node classification, we are able to realize statement-level detection and gain the specific vulnerable code elements within the statement. Furthermore, we provide a vulnerability report offering more specific and detailed information.
- We gain better performance. Experimental results have demonstrated that SedSVD outperforms existing state-of-art techniques with 95.15% on F1-measure.

The rest of the paper is organized as follows. Section 2 explains the technical details of our approach. Section 3 designs experiments and introduces the experimental setup information. Section 4 presents and analyzes the experimental results. Section 5 discusses the importance of our work. Section 6 lists the threats to validity. Section 7 reviews related work. Section 8 summarizes the paper and put forward future work.

2. Our approach

2.1. Basic idea and overview of the model

2.1.1. Basic idea

Since source code has both syntactic and semantic information, we cannot treat them as simple as natural language. There have been several structures representing code in different ways to facilitate code analysis including AST, Control Flow Graph (CFG) and Program Dependency Graph (PDG). Specifically speaking, AST, as the backbone of code analysis, encodes how statements are nested using a tree structure where inner nodes represent operators and leaf nodes represent operands [34]. CFG, however, uses directed edges connecting statements and predicates represented by nodes to describe execution order and the transfer of control [34]. PDG, on the other hand, explicitly describes dependencies between statements and predicates [34]. Particularly, it consists of Data Dependency Graph (DDG) and Control Dependency Graph (CDG) where the former represents the influence on a variable like accessing or modifying while the later corresponds to the influence of predicates on the value of a variable [34]. CPG, a combination of the above three, has been proven more effective to make a comprehensive analysis of source code and programs. Therefore, we apply CPG to analyze code and extract syntactic and semantic information. In this method, we use Joern [37] to extract CPG. Joern is an open source tool which can analyze C/ C++ source code and generate CPG for each function from source file or code snippets without compiling [38].

By extracting CPG of each source file, code elements are parsed into nodes, and relations between them are parsed into edges. For each node, it represents a certain code element including corresponding code, line number, column number, type (`Method`, `Return`, `ControlStructure`, etc.) and so on. As for edges, they reveal relations between each two nodes. They are defined below:

Definition 1 (*Program, Function [1]*)**.** A program P is a set of functions denoted as $P = \{f_1, f_2, \ldots, f_n\}$. A function f_i is a set of ordered statements.

Definition 2 (*AST [34]*)**.** Consider a program $P = \{f_1, f_2, \ldots, f_n\}$, the AST of f_i is denoted as $G^i_{ast} = \{V^i_{ast}, E^i_{ast}\}$ where V^i_{ast} is the set of all nodes in the tree of f_i and E^i_{ast} is the set of all edges connecting them.

Definition 3 (*CFG [34]*)**.** Consider a program $P = \{f_1, f_2, \ldots, f_n\}$, the CFG of f_i is denoted as $G^i_{cfg} = \{V^i_{cfg}, E^i_{cfg}\}$ where V^i_{cfg} is the set of nodes which can represent statements or predicates in V^i_{ast}. E^i_{cfg} is the set of edges revealing the control flow between a pair of nodes.

Definition 4 (*PDG [34]*)**.** Consider a program $P = \{f_1, f_2, \ldots, f_n\}$, the PDG of f_i is denoted as $G^i_{pdg} = \{V^i_{pdg}, E^i_{pdg}\}$ where $V^i_{pdg} = V^i_{cfg}$ and E^i_{pdg} is the set of edges representing data dependencies (accessing or modifying a variable) or control dependencies (the influence of predicates on the value of a variable) between a pair of nodes.

Definition 5 (*CPG [34]*)**.** Consider a program $P = \{f_1, f_2, \ldots, f_n\}$, the CPG of f_i is denoted as $G^i = \{V^i, E^i\}$ where $V^i = V^i_{ast}$ and $E^i = E^i_{ast} \cup E^i_{cfg} \cup E^i_{pdg}$.

```
1  int f(a, b)
2  {
3    int x=0;
4    if (a > x)
5    {
6      x = x + a;
7    }
8    x = x + b;
9    return x;
10 }
```

Fig. 1. Example source code.

Fig. 2 is an example of CPG of the source code shown in Fig. 1.

Previous graph-related tasks most commonly analyze at graph-level [39] or node-level [40]. Especially for graph-based software vulnerability detection approaches, graph-level is more common [3,23,25]. Whereas, in the case of vulnerability detection, the former is too coarse to provide specific information and the latter may isolate each code without taking them as a whole and take account of too many non-vulnerable nodes since most nodes are non-vulnerable. Efforts have been made to solve this problem. Hin, D., et al. [33] combines graph-level and node-level (or function-level and statement-level in the case of code) together to make prediction of each node and provides a report of vulnerable statements. Zhuang, Y., et al. [26] treats single multi-graph (AST, CFG, DFG) separately at first to fully utilize the vulnerable modes the specialized graph can reveal and combines them at last. With their contributions, however, their performance is relatively low. Inspired by previous work, we embed graph at subgraph-level which can not only make up for the shortcomings of graph-level embedding and node-level embedding, but locate vulnerable code elements in their contexts and gain a more comprehensive analysis. But if we build subgraph for every node, it is bound to include a single node into several graphs, leading to the decrease of efficiency and performance. To benefit from the effectiveness of subgraph-embedding while not suffering from the backwash brought by too many useless nodes, we only build subgraphs for center nodes which are more related to vulnerabilities. Through center nodes selection and subgraph-level embedding, we are capable of gaining a finer-grained result than graph-level and skipping useless nodes without isolating others.

As for graph embedding, although been recognized as a critical component in code-related graph, edges are rarely integrated into graph embedding. Given that in our work, multiple relation types contain different yet significant information that we cannot miss any of them, we retain all of them and process them differently by applying RGNN [36], which has made incremental improvements over Graph Convolutional Network (GCN) [41,42] to deal different edges in different ways.

Unlike existing techniques detecting at function-level, file-level, or slice-level, our model detects at statement-level and provide the specific error code elements within a statement. Through splitting a statement into several nodes and conducting node classification, if a node is reported as vulnerable, then its corresponding code, which is actually a code element belonging to a statement, is error. Figs. 3 and 4 give a vivid example.

As shown in Fig. 4, the statement in line 1147 is split into 6 nodes. Through node classification we can know that Node 3, Node 4 and Node 6 are vulnerable. Therefore, their corresponding code elements are vulnerable, that is, the addressing of the pointer `line_to_free` as a parameter in a function call within the *if* statement in line 1147 is error.

2.1.2. Overview of the model

The architecture of SedSVD is described in Fig. 5. Overall, there are two phases in this model: feature extraction and feature learning. In feature extraction, we transform source code into several graphs and embed them into vectors to be fed into neural network. In feature learning, we train a graph model to learn the features and make prediction of given nodes (vulnerable or not). For vulnerable nodes, a detailed vulnerability report will be attached.

2.2. Feature extraction

In this phase, we are supposed to convert code into graphs and represent them with vectors while retain the syntax and semantic information of the code to the greatest extent. We first select center nodes which are more related to vulnerabilities for the sake of efficiency and more attention on vulnerability. After that, we build subgraphs for center nodes and get their representations.

2.2.1. Center node selection

To select center nodes more related to vulnerabilities, we first analyze the characteristics of vulnerable nodes after generate ground truth label of nodes. Based on the observation, we finally build the selecting criterion of center nodes.

a. Generating ground truth label

We build our dataset from NVD and SARD. In NVD dataset [45], each program contains some vulnerability corresponding to a Common Vulnerabilities & Exposures Identifier (CVE ID) [46]. Each source code has a vulnerable version and its patched version accompanied with a *diff* file pointing their difference. As for SARD dataset [47], it is composed of test cases with three types: good for non-vulnerable, bad for vulnerable and mixed for vulnerable with patched version. For bad and mixed programs, which are identified by Common Weakness Enumeration Identifier (CWE ID) [48], a test case information file displaying flaw line numbers will be attached.

Here we mainly consider line deletion in *diff* files. But instead of simply labeling a code element as vulnerable if it is located in a deleted line, we make some refinement. Line deletion means the lines are deleted, modified, or moved. While deleted lines can simply be deemed as vulnerable, modified lines are partly vulnerable and moved lines themselves are actually non-vulnerable. Taking the example in Fig. 4, all nodes are located in the error line yet only Node 3, Node 4 and Node 6 are vulnerable since they contain error code. Inspired by this, we assume that

(i) If a node is located in a deleted line, then it is labeled as 1 (0 for non-vulnerable and 1 for vulnerable).
(ii) If a node is located in a modified line and its code has been changed by a patch, then it is labeled as 1.
(iii) If a node is located in a moved line and this file contains a certain vulnerability, then it is labeled as 1.

Algorithm 1 gives a high-level description of how to label each node in a code from NVD.

b. Observation of vulnerable characteristics

After generalizing ground truth labels and analyzing of vulnerable nodes, we find that their types are limited to certain kinds. Taking the example in Fig. 3, the error part causes the error of pointer addressing (`&line_to_free`), the function call (`get_function_body(...)`) and the *if* statement (`if (get_function_body(...))`), which can be exactly reflected by the types of the vulnerable nodes: Node 6 (`addressOf`), Node 3 (`Call`) and Node 4 (`ControlStructure`) in Fig. 4.

c. Selecting criterion building

Inspired by this, we assume that node types are crucial to reveal vulnerability characteristics. Therefore, we decide to select center nodes according to their types.

We first learn the information of all vulnerable nodes, including their types and code. After adequate analysis, we find that some node types embody the same vulnerability characteristics as Function Call, Array Usage, Pointer Usage and Arithmetic Expression summarized in SySeVR [1]. And these vulnerability characteristics are actually the most likely to cause common vulnerabilities, for instance, Pointer Usage may cause NULL Pointer Deference, Array Usage may cause Out of Bound. This is exactly consistent to our definition of center nodes: nodes which are more related to vulnerabilities. Therefore, we first summarize several vulnerability characteristics based on our extensive analysis on vulnerable nodes and then map them to specific node types as listed in Table 1, accompanied with example code.

Overall, nodes with these types are taken as center nodes. Node types listed in Table 1 constitute the set of center node type denoted as T.

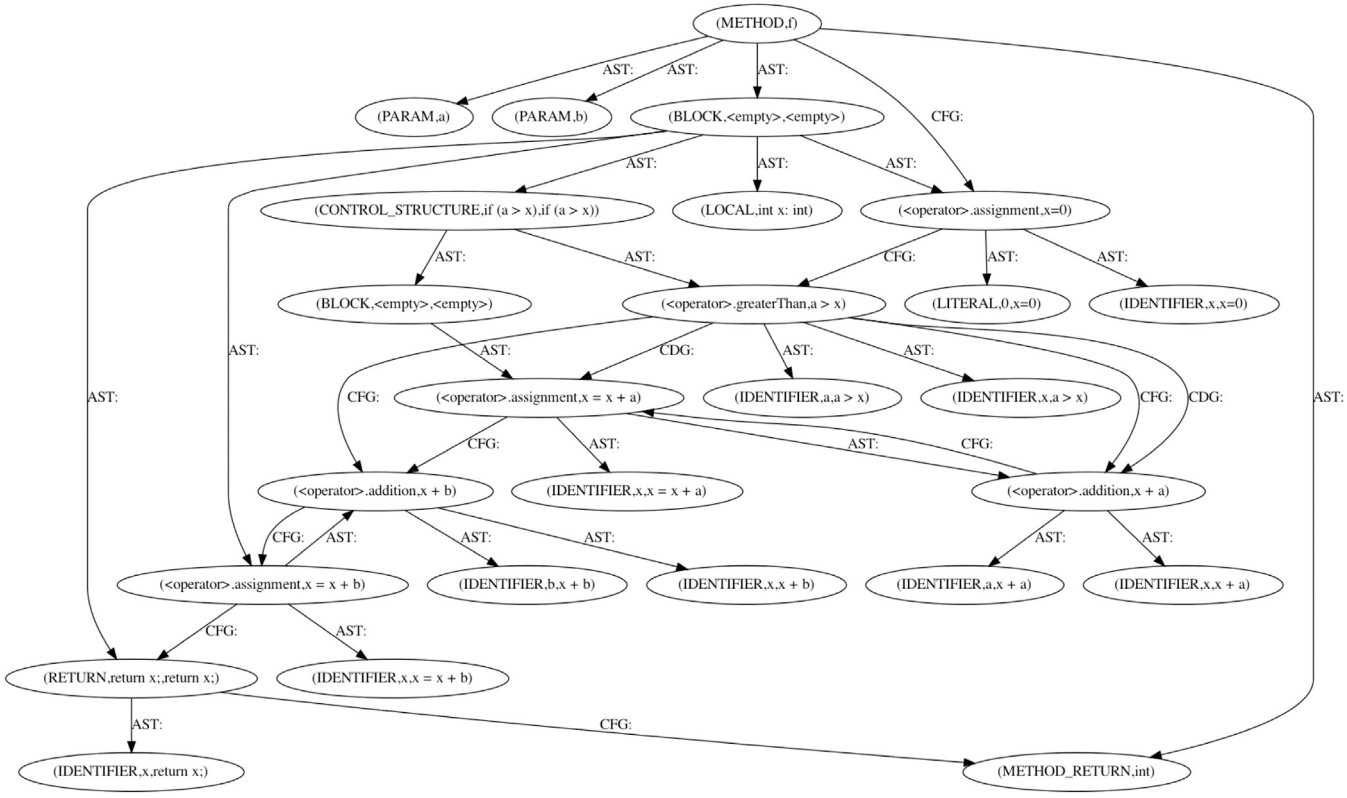


Fig. 2. The CPG of the example code in Fig. 1.

```
@@ -1144,12 +1150,9 @@ lambda_function_body(
1144 1150      }
1145 1151
1146 1152      ga_init2(&newlines, (int)sizeof(char_u *), 10);
1147      -    if (get_function_body(&eap, &newlines, NULL, &line_to_free) == FAIL)
1148      -    {
1149      -        if (cmdline != line_to_free)
1150      -            vim_free(cmdline);
     1153 +    if (get_function_body(&eap, &newlines, NULL,
     1154 +                                        &evalarg->eval_tofree_ga) == FAIL)
1151 1155          goto erret;
1152      -    }
1153 1156
1154 1157      // When inside a lambda must add the function lines to evalarg.eval_ga.
1155 1158      evalarg->eval_break_count += newlines.ga_len;
```

Fig. 3. A partial patch [43] for CVE 2022-0156 [44] with red rectangles pointing difference between line 1147 and line 1153-1154.

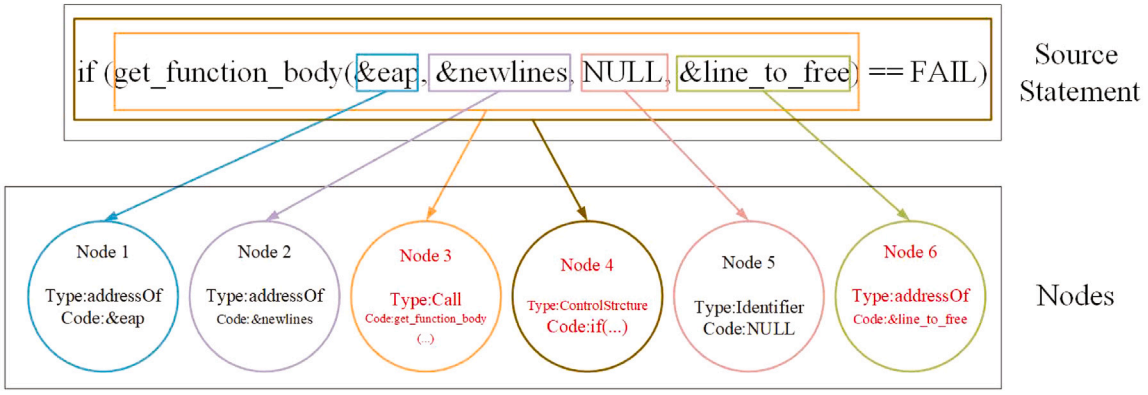


Fig. 4. The corresponding nodes of line 1147 in Fig. 3 where nodes with red text are error.

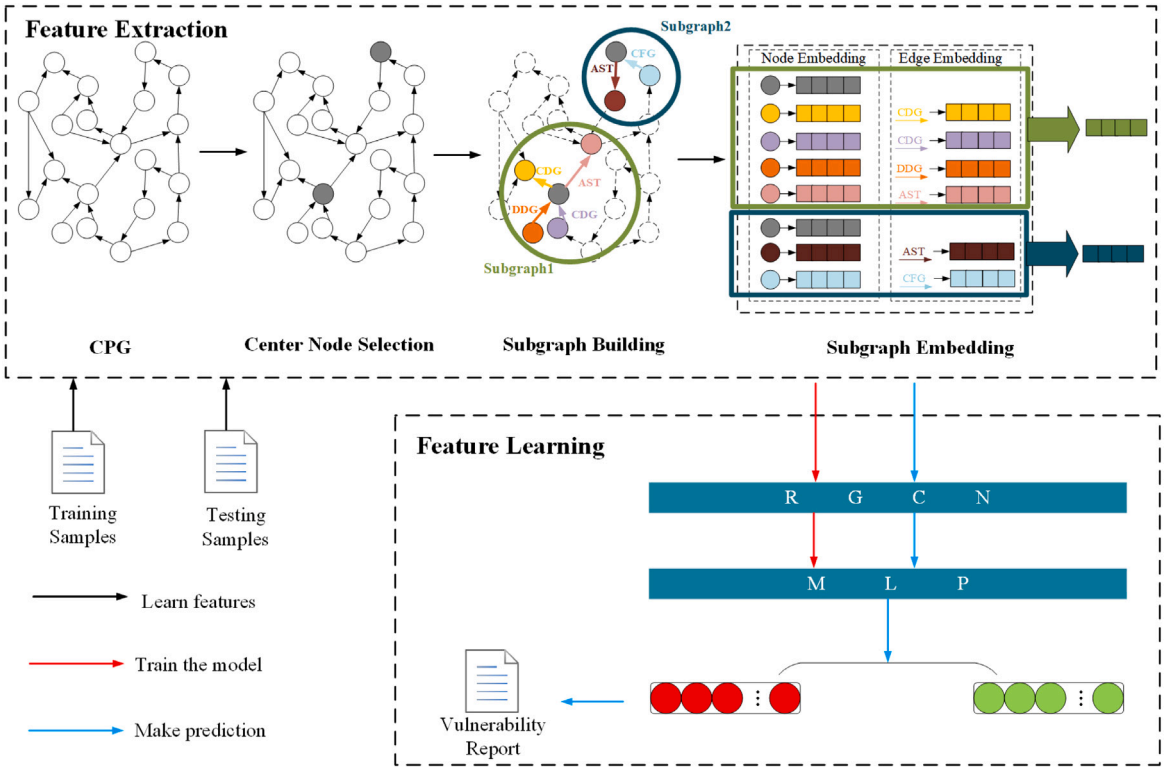


Fig. 5. The architecture of SedSVD.

2.2.2. Subgraph embedding

After selecting center nodes, we build their subgraphs and their vector representations so that they can be taken as the input of deep learning network.

Here we build subgraphs for center nodes selected above. For each center node, collect all its neighbor nodes and connected edges on the basis of CPG. We give the definition of subgraph in Definition 6.

Definition 6 (*Subgraph [49]*)**.** Consider a program $P = \{f_1, f_2, \ldots, f_n\}$, the CPG $G^i = \{V^i, E^i\}$ of each function f_i, the set of center node types T, and a node $v_j \in V^i$ where $v_j.type \in T$,

- The set of neighbor nodes of v_j is denoted in Eq. (1).

$$V^{i,j} = V_{in} \cup V_{out} \cup \{v_j\} \tag{1}$$

where

$$V_{in} = \{v_k \in V^i | e =< v_k, v_j >, e \in E^i\} \tag{2}$$

and

$$V_{out} = \{v_k \in V^i | e =< v_j, v_k >, e \in E^i\} \tag{3}$$

- The set of edges connecting nodes in $V^{i,j}$ is denoted as $E^{i,j}$ and defined in Eq. (4).

$$E^{i,j} = \{e =< v_k, v_p >, e \in E^i | v_k \in V^{i,j}, v_p \in V^{i,j}\} \tag{4}$$

Therefore, the subgraph of v_j can be denoted as

$$G^{i,j} = \{V^{i,j}, E^{i,j}\} \tag{5}$$

After that, we embed subgraph, including node embedding and edge embedding. As for node embedding, we embed each node with its type and code. For node types, we apply label encoding. For code, we will first tokenize them into tokens which means each element in a statement will be represented symbolically by mapping user-defined names to symbolic names like "FUN1" standing for a function and "VAR1" for a variable. After that, these tokens will be embedded with

Table 1
Vulnerability characteristics, their corresponding node types and example code.

Vulnerability characteristics	Node type	Example code
Function calls	`Call`	fput(c)
Use an array	`indirectIndexAccess`	data[i]
Use a pointer	`addressOf` `indirection`	&c *s
Conduct an arithmetic calculation	`addition/subtraction/multiplication/division` `postDecrement/postIncrement/preDecrement/preIncrement`	n+1 i++
Relate to control flow and code structure of the project	`ControlStructure` `JumpTarget`	if (err <0) case 1:
Assign values to variables	`assignment/assignmentPlus/assignmentMinus/` `assignmentMultiplication/ assignmentDivision`	k = 5
Open or discard a memory space	`new/delete`	delete info

Access a field of an object of aggregate type	`fieldAccess`
	`FieldIdentifier`
	`indirectFieldAccess`
		ii.iov
		iov
		ii.iov ->iov_len

Conduct a boolean operation	`logicalAnd/logicalOr/logicalNot`	*y && *y <h
Decide the type of the variable	`Identifier`	int i

Algorithm 1 Generating ground truth labels of nodes from each file in NVD

Input: the set of NVD source code files C, NVD *diff* files D
Output: A label set L of nodes in each file in C

for $file \in C$ **do**
  A set of code added by the patch of in this file $Code\_add \leftarrow \emptyset$
  A set of code deleted by the patch of in this file $Code\_del \leftarrow \emptyset$
  A set of line numbers of line deletion in this file $Line \leftarrow \emptyset$
  A set of line numbers of moved lines in this file $Line\_m \leftarrow \emptyset$
  $(V, E) \leftarrow extract\_CPG\_with\_Joern(file)$
  if $file$ is patched **then**
    for $n \in V$ **do**
      $n.label \leftarrow 0$
      $file\_label \leftarrow file\_label \cup n.label$
    $L \leftarrow file\_label$
    continue
  $diff = D[file.name]$
  for $line \in diff$ **do**
    if $line.prefix =$ "+" **then**
      $Code\_add \leftarrow Code\_add \cup line$
    if $line.prefix =$ "-" **then**
      $Code\_del \leftarrow Code\_del \cup line$
  $Code\_same \leftarrow Code\_add \cap Code\_del$
  for $line \in file$ **do**
    if $line \in Code\_del$ **then**
      $Line \leftarrow Line \cup line.number$
    if $line \in Code\_same$ **then**
      $Line\_m \leftarrow Line\_m \cup line.number$
  for $n \in V$ **do**
    if $n.get\_line\_number() \in Line\_m$ **then**
      $n.label \leftarrow 1$
    else if $n.get\_line\_number() \in Line$ and $n.get\_code() \notin Code\_add$ **then**
      $n.label \leftarrow 1$
    else
      $n.label \leftarrow 0$
    $file\_label \leftarrow file\_label \cup n.label$
  $L \leftarrow file\_label$
return L

Word2vec. For each statement, we will get its result representation by averaging the embedding vectors of all its tokens. Tokens are defined in Definition 7.

Definition 7 (*Token*)**.** Consider a program $P = \{f_1, f_2, \dots, f_n\}$, where f_i is a set of ordered statements denoted as $f_i = \{s_{i,1}, s_{i,2}, \dots, s_{i,m}\}$. Tokens of a statement $s_{i,j} \in f_i$ can be defined in Eq. (6).

$$t(s_{i,j}) = \{t_{i,j,1}, t_{i,j,2}, \dots, t_{i,j,p}\} \tag{6}$$

where $t_{i,j,k} \in t(s_{i,j})$ is a token of statement $s_{i,j}$ and p is the number of tokens in $s_{i,j}$.

Therefore, the representation vector of $s_{i,j}$ is the average vector of all embedding vectors of each $t_{i,j,k} \in t(s_{i,j})$. It can be denoted as Eq. (7).

$$vec_{s_{i,j}} = \frac{1}{p}\sum_{k=1}^{p} Word2vec(t_{i,j,k}) \tag{7}$$

Fig. 6 gives an example. Then, we can get the feature representation of nodes as described in Definition 8.

Definition 8 (*Node Feature Representation*)**.** Consider a program $P = \{f_1, f_2, \dots, f_n\}$, the CPG $G^i = \{V^i, E^i\}$ of each function f_i, a center node $v_j \in V^i$ and its subgraph $G^{i,j} = \{V^{i,j}, E^{i,j}\}$,

- The node type of v_j which is decoded by label encoding can be denoted as Eq. (8).

$$t_{v_j} = label\_encoding(v_j.type) \tag{8}$$

- Therefore, the feature vector of v_j can be denoted as Eq. (9).

$$\begin{aligned} &x_{v_j} = concatenate(t_{v_j}, vec_{v_j.get\_code()}),\\ &x_{v_j} \in \mathbb{R}^{1\times D} \end{aligned} \tag{9}$$

  where D is the embedding size.

Aggregating feature representations of all nodes in $V^{i,j}$, we can get the node feature representation of $G^{i,j}$ as shown in Eq. (10).

$$x_{V^{i,j}} = \sum_{v_k \in V^{i,j}} x_{v_k}, x_{V^{i,j}} \in \mathbb{R}^{N\times D} \tag{10}$$

where $N = |V^{i,j}|$.

As for edges, we embed their direction and type as described in Definition 9 and Definition 10 respectively.

Definition 9 (*Edge Direction Representation*)**.** Consider a program $P = \{f_1, f_2, \dots, f_n\}$, the CPG $G^i = \{V^i, E^i\}$ of each function f_i, a center node $v_j \in V^i$ and its subgraph $G^{i,j} = \{V^{i,j}, E^{i,j}\}$, the set of edge direction is shown in Eq. (11).

$$\begin{aligned} &I_{E^{i,j}} = \{< v_k.order, v_p.order > | e =< v_k, v_p >, e \in E^{i,j}\},\\ &I_{E^{i,j}} \in \mathbb{R}^{2\times|E^{i,j}|} \end{aligned} \tag{11}$$

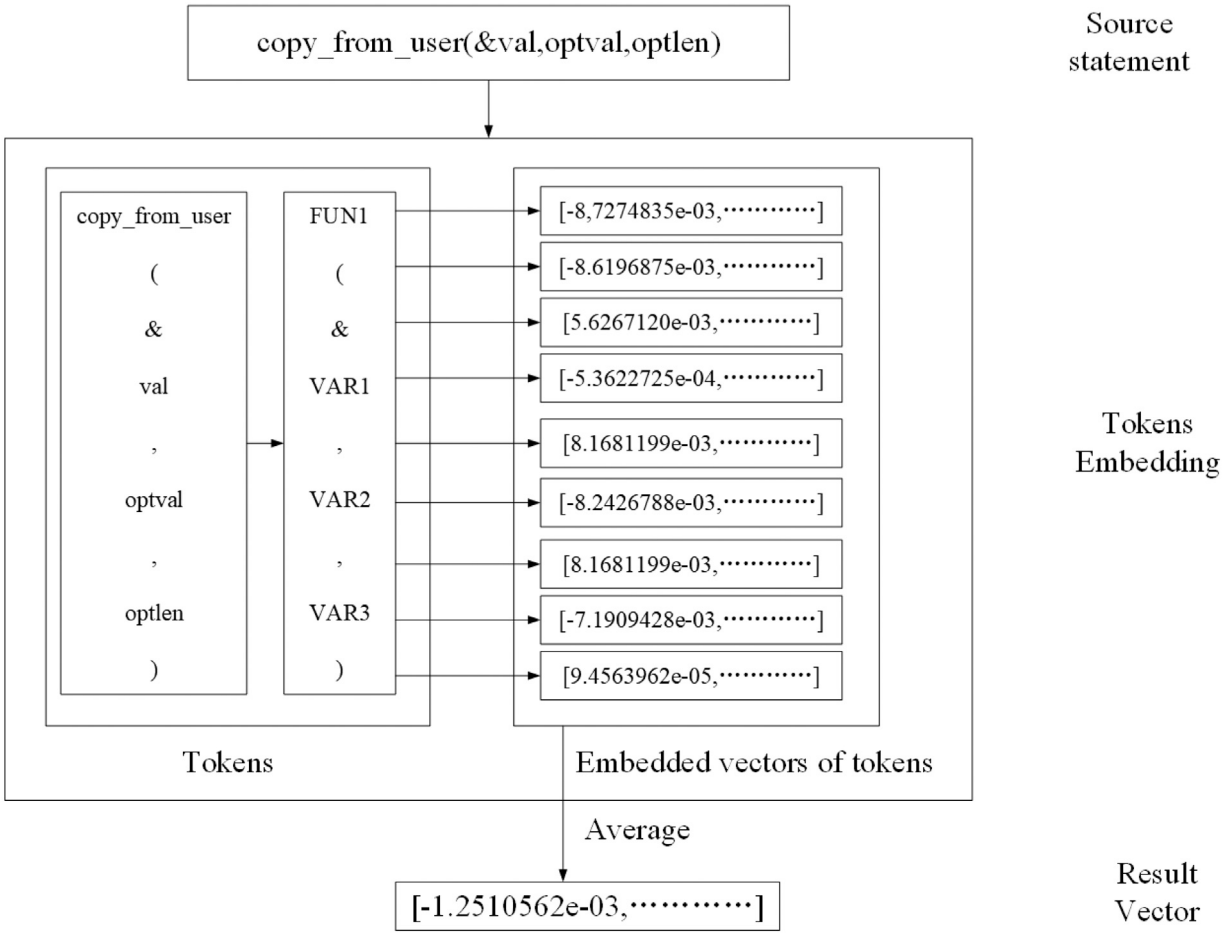


Fig. 6. An example of tokenizing statements.

Definition 10 (*Edge Type Representation*)**.** Consider a program $P = \{f_1, f_2, \ldots, f_n\}$, the CPG $G^i = \{V^i, E^i\}$ of each function f_i, a center node $v_j \in V^i$ and its subgraph $G^{i,j} = \{V^{i,j}, E^{i,j}\}$, the set of edge type is denoted in Eq. (12).

$$T_{E^{i,j}} = \{t_e | e \in E^{i,j}\}, T_{E^{i,j}} \in \mathbb{R}^{1\times|E^{i,j}|} \tag{12}$$

where

$$t_e = \begin{cases} 2, & if\ e.type = AST \\ 1, & else\ if\ e.type = CFG \\ 0, & else \end{cases} \tag{13}$$

Hence, we can get the feature representation of the subgraph as described in Definition 11.

Definition 11 (*Subgraph Feature Representation*)**.** Consider a program $P = \{f_1, f_2, \ldots, f_n\}$, the CPG $G^i = \{V^i, E^i\}$ of each function f_i, a center node $v_j \in V^i$ and its subgraph $G^{i,j} = \{V^{i,j}, E^{i,j}\}$. The node feature vector of $G^{i,j}$ is $x_{V^{i,j}}$, edge direction vector is $I_{E^{i,j}}$, and edge type vector is $T_{E^{i,j}}$, the feature vector of $G^{i,j}$ can be denoted as Eq. (14).

$$x_{G^{i,j}} = Graph\_Data(x_{V^{i,j}}, I_{E^{i,j}}, T_{E^{i,j}}) \tag{14}$$

where *Graph_Data* represents *Data* from *torch_geometric.data*[50], a data type representing graph structure using multiple attributes, including node, edge index and edge weight. Therefore, we apply it to integrate these three types of data.

Algorithm 2 shows the specific process of what has described above.

An example of subgraph embedding of a center node is illustrated in Fig. 7. In Fig. 7, we first build the subgraph of the center node represented by the red circle while yellow circles are neighbor nodes. In what follows, we embed their nodes and edges. The specific process of node embedding, as shown in "2.1 Node embedding" in Fig. 7, includes embedding node type with label encoding and node code with Word2vec and then concatenate them to form a vector. Perform the above operations on each node to get a vector with uniform length and concatenate all of them to get the overall node representation. Edge embedding, on the other hand, is comprised of edge index and edge type where edge index is a set of pairs of node order of the entry and exist node of an edge representing edge direction and edge type is a set of edge type embedding described in Definition 10 as shown in "2.2 Edge embedding" in Fig. 7. After that, these three part are integrated together to form the final representation of this subgraph as shown in "3 Subgraph embedding" in Fig. 7.

2.3. Feature learning

Having obtaining the final representation of subgraphs, we then build a RGCN module to learn the aggregated information since it is suitable to process heterogeneous graph. The results are then fed into MLP, which further trains the classifier and predicts the samples in terms of the probability.

2.3.1. Information exchange

This step takes graph representations from the previous step and conducts information exchange and gains an overall representation of graphs finally.

GCN is an extension of CNN on graphs. After applying a graph convolution operation on a given graph, nodes are updated by aggregating

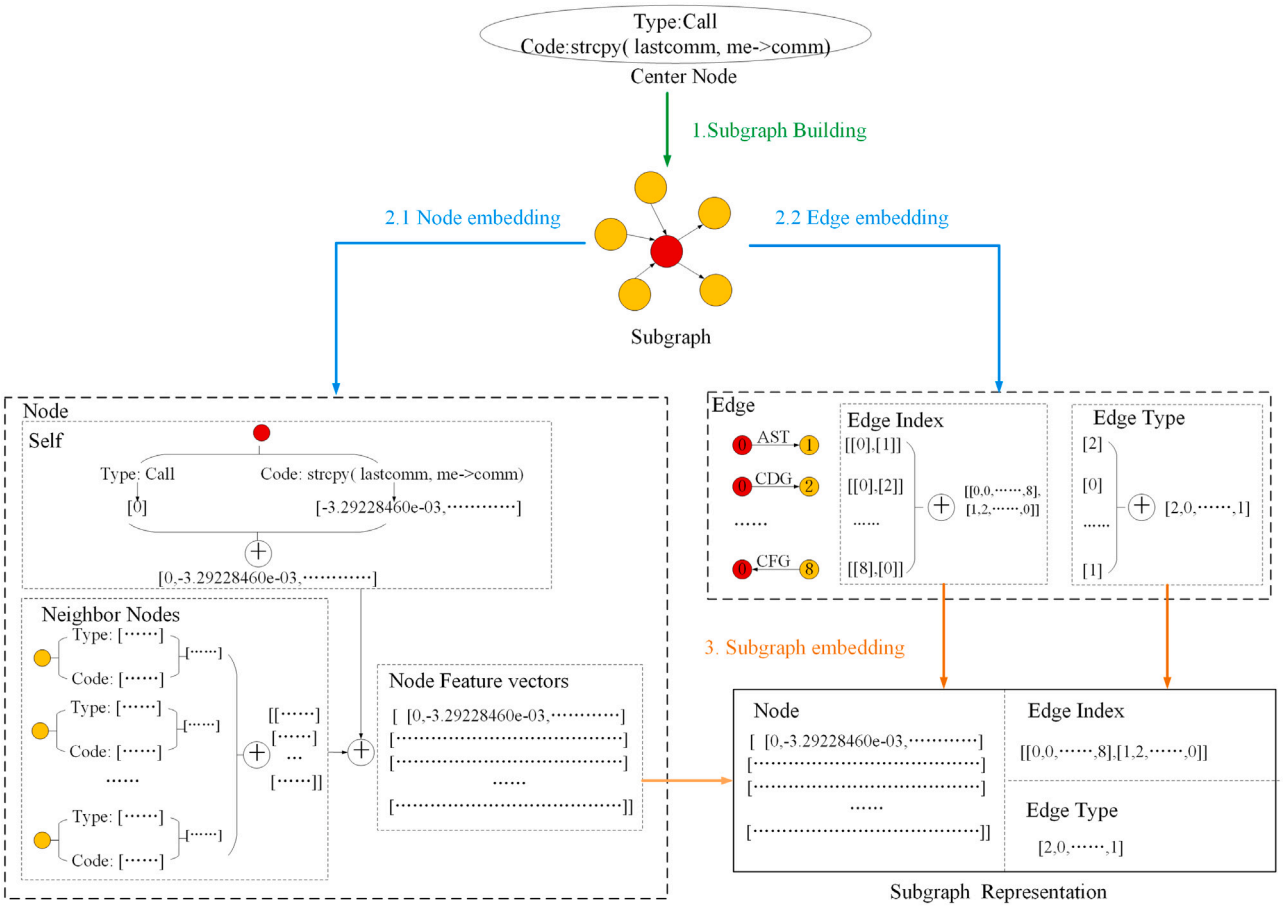


Fig. 7. An example of embedding the subgraph of a center node.

all the features of their neighbor nodes [51]. Compared with Recurrent Graph Neural Network, such as GGNN, which continuously calculates node states, GCN is more efficient and scalable since it updates node vectors once per-layer [19,26].

RGCN makes incremental improvements over GCN which treats different edges differently by updating nodes with neighbor nodes under each relation type. Thus, it is more suitable to process heterogeneous graph.

In each RGCN layer, updating a single node requires to accumulate information of all its neighbor nodes per type, which is later combined with the feature of this node itself and passed through an activation function, such as $ReLU$, to form the resulting representation. Fig. 8 visualizes this process.

Consider a program $P = \{f_1, f_2, \ldots, f_n\}$, the CPG $G^i = \{V^i, E^i\}$ of each function f_i, a center node $v_j \in V^i$, its subgraph $G^{i,j} = \{V^{i,j}, E^{i,j}\}$ and the representation of the subgraph $x_{G^{i,j}} = Graph\_Data(x_{V^{i,j}}, I_{E^{i,j}}, T_{E^{i,j}})$, the node representations are updated through Eq. (15) according to [52] at each layer.

$$x_{v_j}^{(l+1)} = x_{v_j}^{(l)} + \sum_{r \in R} \sum_{v_k \in V_r^{i,j}} \frac{1}{|V_r^{i,j}|} \sigma\left(W_r \left[x_{v_j}^{(l)}, x_{v_k}^{(l)}, I_{j,k}, T_{j,k}\right] + b_r\right), \quad x_{v_j}^{(l+1)} \in \mathbb{R}^{1 \times D} \tag{15}$$

where $\sigma(\cdot)$ represents a non-linear activation function and we use $ReLU(\cdot) = max(0, \cdot)$ for the rest of the paper. R is the set of edge types and $V_r^{i,j}$ is the set of neighbor nodes of v_j under relation $r \in R$. W_r and b_r are learnable parameters. $I_{j,k} \in I_{E^{i,j}}$ and $T_{j,k} \in T_{E^{i,j}}$ are the direction and type of the edge between v_j and v_k respectively.

2.3.2. Classifier

Due to the distinguished performance and dominant position in top classifiers of MLP [33], we further leverage the capability of MLP to train the classifier. The MLP we use consists of three layers: the input layer, the intermediate layer and the output layer [53].

Here the output from RGCN module is taken as the input layer. The node representation is transformed as Eq. (16).

$$h_{v_j} = \sigma\left(W_1 * x_{v^j}^L + b_1\right), h_{v_j} \in \mathbb{R}^{N \times D} \tag{16}$$

where W_1 and b_1 are learnable parameters and h_{v_j} is the representation of v_j. L is the number of RGCN layers.

Finally, the output layer will utilize graph features from the intermediate layer to make prediction. Consider a program $P = \{f_1, f_2, \ldots, f_n\}$, the CPG $G^i = \{V^i, E^i\}$ of each function f_i, a center node $v_j \in V^i$ and its subgraph $G^{i,j} = \{V^{i,j}, E^{i,j}\}$, we can get the predictive label $\widehat{y_{v_j}}$ of node v_j through Eq. (17).

$$\widehat{y_{v_j}} = Sigmoid(W_2 * h_{v_j} + b_2) \tag{17}$$

where W_2 and b_2 are learnable parameters.

3. Experimental setup

3.1. Research questions

The experiment results will be presented in the form of answering Research Questions (RQ) below:

RQ1: How effective SedSVD is compared with state-of art software vulnerability detection tools?

This question is designed to demonstrate the capability of SedSVD when compared with existing tools. To answer this question, we compare our work with the following state-of-art approaches:

- Group 1: VulDeePecker [30], SySeVR [1] and VulDeeLocator [31]

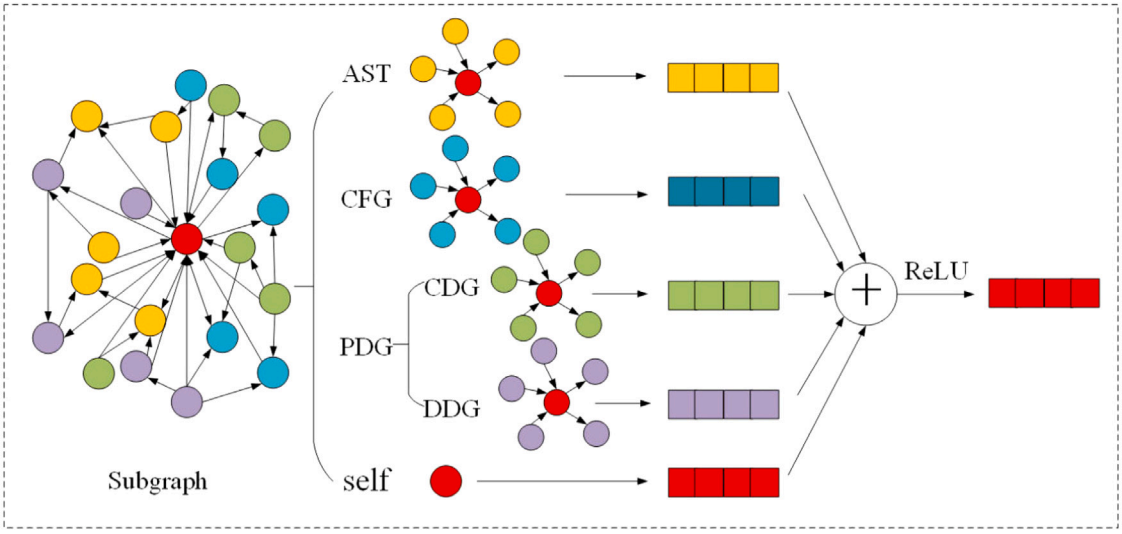


Fig. 8. The process of updating a node in a layer.

Algorithm 2 Subgraph Building and Embedding

Input: A program $P = \{f_1, f_2, \ldots, f_n\}$, The set of center node types T
Output: A set x_P of feature vectors for each function in P

$x_P \leftarrow \emptyset$
for $f \in P$ **do**
  $(V, E) \leftarrow extract\_CPG\_with\_Joern(f)$
  for $n \in V$ **do**
    if $n.type \notin T$ **then**
      continue
    $neighbor \leftarrow neighbor \cup n$
    for $e \in n.edges$ **do**
      if $e.node\_in \neq n$ and $e.node\_in \in V$ **then**
        $neighbor \leftarrow neighbor \cup e.node\_in$
      if $e.node\_out \neq n$ and $e.node\_out \in V$ **then**
        $neighbor \leftarrow neighbor \cup e.node\_out$
    $neighbors \leftarrow neighbors \cup neighbor$
  $x_f \leftarrow \emptyset$
  for $neighbor \in neighbors$ **do**
    for $n \in neighbor$ **do**
      $type_n = label\_encoding(n.type)$
      $token_n = tokenize(n.get\_code())$
      $code_n = mean(Word2vec(token_n))$
      $x_n = concatenate(type_n, code_n)$
      $x_{neighbor} \leftarrow x_{neighbor} \cup x_n$
      for $e \in n.edges$ **do**
        if $e.node\_in \in neighbor$ and $e.node\_in \neq n$ **then**
          $I_e \leftarrow [e.node\_in.order, n.order]$
          $I_{neighbor} \leftarrow I_{neighbor} \cup I_e$
          $t \leftarrow e.get\_type()$
          $T_{neighbor} \leftarrow T_{neighbor} \cup T_e$
        if $e.node\_out \in neighbor$ and $e.node\_out \neq n$ **then**
          $I_e \leftarrow [n.order, e.node\_out.order]$
          $I_{neighbor} \leftarrow I_{neighbor} \cup I_e$
          $t \leftarrow e.get\_type()$
          $T_{neighbor} \leftarrow T_{neighbor} \cup T_e$
      $x_g \leftarrow Graph\_Data(x_{neighbor}, I_{neighbor}, T_{neighbor})$
      $x_f \leftarrow x_f \cup x_g$
  $x_P \leftarrow x_P \cup x_f$
return x_P

- Group 2: Devign [54], ReVEAL [23], BGNN4D [25] and Zhuang, Y., et al. [26].

They are all deep learning-based vulnerability detection techniques where tools in Group 1 share the same dataset with us and tools in Group 2 are graph-based ones like us.

RQ2: Is RGCN more helpful to the detection of vulnerabilities?

In our approach, we employ RGCN to process different edges differently. To explore whether it can facilitate vulnerability detection and improve the performance of our model, we design this question and compare RGCN with GGNN [20] and GCN [41,42] respectively. GGNN is a kind of recurrent neural network using gating information to update nodes. GCN is the basis of RGCN without relation aware.

RQ3: Is building subgraphs more helpful to promote the performance of the model?

In our method, we promote subgraph embedding for center nodes selected before to avoid inviting too many useless nodes and narrow down analysis scope. To evaluate the rationality and superiority of building subgraphs for center nodes, we design this question.

Totally, we embed CPG of each source code with the following modes:

- G: Embed the whole graph.
- SubG-A: Embed subgraphs for all nodes.
- SubG-C: Embed subgraphs only for center nodes.

3.2. Deployment information

We implement GGNN, GCN and RGCN in Python using Pytorch [55]. We run the experiment on the Linux server with NVIDIA GeForce RTX 3090 GPU and Intel Core i7-10700 CPU operating at 2.90 GHz.

The trained hyper parameters are: the size of hidden layer is 50, batch size is 8, epoch is 10, optimizer is ADAM, dropout ratio is 0.2, learning rate is 1e-4, weight decay is 1e-2 and activation function is $ReLU$.

3.3. Dataset

We build a dataset from NVD and SARD. For NVD, we collect 1737 files among which 917 are vulnerable. As for SARD dataset, we collect 20048 files among which 12467 are vulnerable. Totally, there are 21785 programs in our dataset including 13384 vulnerable ones. See specific information in Section 2.2.1.

After building the dataset, we transform them into CPG. Then we extract nodes and edges. As for nodes, whose types are `Unknown` or `Comment` are removed since they are helpless. Then, we label them as described in Section 2.2.1. Since most nodes are non-vulnerable, there

Table 2
Vulnerability detection capability of the state-of-the-art methods and SedSVD (Group 1) (Metric unit:%).

Method	Adopted network	Detecting level	FPR	FNR	ACC	F1
VulDeePecker	BLSTM	Code gadget	22.90	16.90	80.10	80.80
SySeVR	BGRU	Slice	**1.70**	19.00	**96.00**	84.40
VulDeeLocator	BRNN-vdl	Statement	12.10	21.70	83.90	80.00
SedSVD	RGCN	Statement	2.97	**8.93**	91.69	**95.15**

exists an imbalance between vulnerable samples and non-vulnerable samples which may impact the performance of the model negatively. Hence, we undersample non-vulnerable samples to mitigate this problem by ignoring patched versions of projects since they only contain non-vulnerable samples. After that, the dataset reaches a balance, where the ratio of vulnerable samples to non-vulnerable samples is about 6:4. Subsequently, we shuffle and divide the training/test set into 8:2.

3.4. Evaluation metrics

In this model, we choose the following metrics which are widely used: False Positive Rate (FPR), False Negative Rate (FNR), Accuracy (ACC) and F1-measure (F1). Let TP be the number of non-vulnerable examples detected as non-vulnerable, TN be the number of vulnerable examples detected as vulnerable, FP be the number of vulnerable examples but detected as non-vulnerable and FN be the number of non-vulnerable examples but detected as vulnerable. The descriptions of these metrics are listed as follows:

- $FPR = \frac{FP}{FP+TN}$. It calculates the proportion of negative samples predicted to be positive among all negative samples.
- $FNR = \frac{FN}{TP+FN}$. It calculates the proportion of positive samples predicted to be negative among all positive samples.
- $ACC = \frac{TP+TN}{TP+TN+FP+FN}$. It calculates the proportion of correctly classified samples among all samples.
- $F1 = \frac{2\times P\times R}{P+R}$. It is an overall metric balancing P and R. $P = \frac{TP}{TP+FP}$ is precision, that is, the proportion of truly-classified positive samples among all claimed positive samples. $R = \frac{TP}{TP+FN}$ is recall, that is, the proportion of truly-classified positive samples among all true positive samples.

A model with high ACC, F1 and low FPR, FNR is deemed as effective.

4. Experimental results

4.1. Results for RQ1

Table 2, Table 3 and Fig. 9 tabulate the comparison between SedSVD and tools in Group 1, Group 2. Totally, SedSVD outperforms tools in Group1 and Group 2.

As shown in Table 2 and Fig. 9, SedSVD has the lowest FNR (8.93%) and the highest F1 (95.15%). Although our FPR is 1.27% higher and ACC is 4.31% lower than SySeVR, we achieve a tremendous decrease on FNR (10.07%) and a huge increase on F1 (10.75%). Given that F1 is a more comprehensive metric compared with ACC, we can conclude that SedSVD overperforms tools in Group 1. It is attributed to the graph representation grasping both semantic and syntactic information and subgraph embedding capturing vulnerability-related nodes and their contexts. RGCN also helps by considering edge information.

Actually, what mainly makes the accuracy of SySeVR outstanding is that they only focus on four vulnerability types whereas we take all types into consideration. Besides, all of the other three adopt RNN-variants (BLSTM (Bidirectional Long Short-Term Memory), BGRU (Bidirectional Gated Recurrent Unit)) or a network that extends RNN

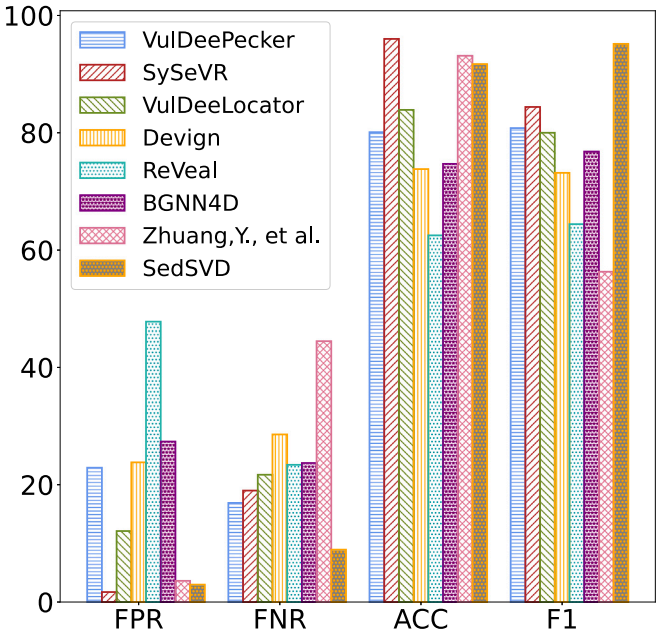


Fig. 9. Result for RQ1.

(BRNN-vdl (Bidirectional Recurrent Neural Network for vulnerability detection and locating)), which are less effective to process non-sequential information and may lose important code information. That is why their performance is undesirable. Moreover, their detection level is relatively coarse. VulDeepecker and SySeVR judge whether a set of source statements (code gadget or slice) is vulnerable or not. VulDeeLocator indicates where the vulnerability is between two source statements. Although they have refined detection granularity to a certain extent, SedSVD can locate vulnerability to specific line numbers and point out the vulnerable code elements as well.

As illustrated in Table 3 and Fig. 10, SedSVD has the lowest FPR (2.97%), FNR (8.93%) and the highest F1 (95.15%).

While Devign and ReVEAL employ Gated Recurrent Unit (GRU) to conduct neighborhood aggregation, both of them ignore edge embedding, thus losing significant code features. That is why their performance is the worst.

In contrast, BGNN4D not only consider edges, but adopt BGNN to process both forward edges and backward edges. Unfortunately, they do not distinguish different edges but actually treat them equally. Hence, they fail to take advantage of graph representation as well.

Similar to us, Zhuang, Y., et al. take in edge information by adopting CGCN, another GCN-variant. Surprisingly, Zhuang, Y., et al. has the highest ACC (93.13%) but the lowest F1 (56.31%). Given that accuracy is unsuitable in the case of imbalanced dataset and the dataset they use is highly imbalanced, F1 is more reliable [26] to measure their performance. So their high ACC and low F1 actually reveal the poor classification performance of their model. In contrast, although our ACC is 1.44% lower, we achieve 38.84% higher on F1, demonstrating the overall performance of our model.

Additionally, we note that all of them choose a combination of various graph structures instead of just one. In terms of the graph adopted, in addition to the common used graphs like CPG, CFG, and DFG, Devign adds Natural Code Sequence (NCS) which connects all the leaf nodes in AST and can preserve the programming logic of code sequence. BGNN4D proposes a new graph structure: Code Composite Graph (CCG) which combines all AST nodes and a set of mixed edges of AST, CFG and DFG. Since each graph structure reveals different vulnerable mode, combining them together can learn code information more comprehensively. In the future, we will look deeper into each

Table 3
Vulnerability detection capability of the state-of-the-art methods and SedSVD (Group 2) (Metric unit:%).

Method	Dataset	Adopted graph	Adopted network	FPR	FNR	ACC	F1
Devign	QEMU+FFmpeg	AST+DFG+CFG+NCS	GGRN	23.81	28.57	73.81	73.17
ReVEAL	QEMU+FFmpeg	CPG	GGNN	47.80	23.39	62.51	64.42
BGNN4D	NVD+github	CCG	BGNN	27.36	23.70	74.70	76.80
Zhuang, Y., et al.	Draper+QEMU+FFmpeg	AST+DFG+CFG	CGCN	3.61	44.47	**93.13**	56.31
SedSVD	NVD+SARD	CPG	RGCN	**2.97**	**8.93**	91.69	**95.15**

Table 4
Vulnerability detection capability of SedSVD using GGNN, GCN and RGCN respectively (Metric unit:%).

Adopted network	FPR	FNR	ACC	F1
GGNN	8.93	14.75	88.19	87.71
GCN	**1.28**	13.33	88.39	92.75
RGCN	2.97	**8.93**	**91.69**	**95.15**

Table 5
The performance of SedSVD with three graph embedding modes (Metric unit:%).

Graph embedding mode	FPR	FNR	ACC	F1
G	10.90	20.41	81.88	86.95
SubG-A	4.19	20.43	82.26	88.22
SubG-C	**2.97**	**8.93**	**91.69**	**95.15**

graph structure and utilize the specific vulnerability mode it reveals to optimize our model to detect certain vulnerability types.

Based on the above analysis, we can get the following finding.

Finding 1: SedSVD shows a superior performance compared with state-of-art techniques, whether using the same dataset with us or are graph-based like us.

4.2. Results for RQ2

Table 4 and Fig. 10 report the result. Totally, RGCN emerges more remarkable performance with much lower FNR and much higher ACC, F1, although FPR is 1.69% higher. The substantial gaps further justify the rationality of our approach. By integrating different relation types separately and then stacking them together, edge information can be reserved and utilized, which is beyond the capability of the other three. Moreover, to avoid the problem of over-fitting caused by the sharp increase of parameters due to multiple relations, RGCN adopts factorization and multi-relation parameter sharing.

On the basis recurrent graph neural network which recurrently update node state applying same set of parameters, GGNN employs GRU as a recurrent function to reduce the recurrence [19]. But it requires calculation for multiple times over all nodes due to the backpropagation through time (BPTT) they choose [19]. This will be problematic when facing large graphs. Therefore, it ends with the worst result among them. GCN is the classical convolutional graph network as mentioned in Section 2.3.1 and RGCN expands it to relational graph domain. In views of our multi-type-edges graphs extracted from CPG, RGCN is more suitable to differentiate edges in terms of their types. Therefore, RGCN achieves better performance than the other two.

Based on the above analysis, we can get the following finding.

Finding 2: SedSVD achieves better performance when applying RGCN.

4.3. Results for RQ3

The comparison is displayed in Table 5 and Fig. 11. Firstly, the comparison between "G" and "SubG-*" proves the priority of subgraph embedding. Just as what we have talked in Section 2.1.1, subgraph embedding can keep the graph-level embedding to take them as a whole while reach a finer-grained granularity than graph embedding and higher efficiency than node embedding.

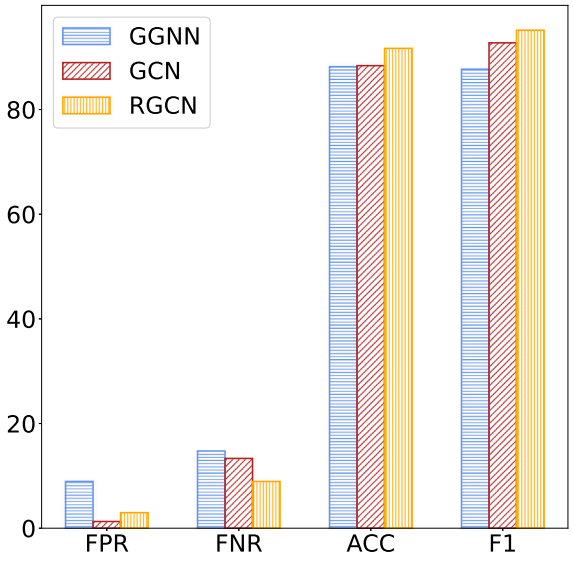


Fig. 10. Result for RQ2.

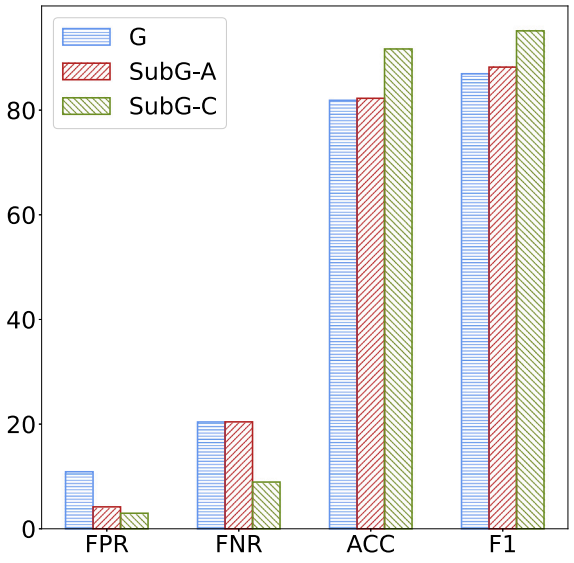


Fig. 11. Result for RQ3.

As for subgraph embedding, it is obvious that embedding subgraphs for center nodes selected is much more effective than for all nodes. Reasons are clear as explained in Section 2.1.1. Since it is inevitable to include a single node in several subgraphs and there are only a few vulnerable elements in source code, building subgraphs for all nodes will certainly invite too many useless and unrelated nodes while masking related ones. Thus, not only can we not benefit from subgraph building, but we have to bear a vast set of mixed information and its negative influence on the performance of our model. By selecting

center nodes which are more related to vulnerabilities, however, we can concentrate more on the vulnerabilities themselves and their contexts.

But the problem that one node may be processed for several times still exists. It can only be mitigated but not fully solved when building subgraphs for center nodes. In the future, we will focus on this problem and reduce the redundant operation on a single node as much as possible and improve the efficiency.

Based on the above analysis, we can get the following finding.

Finding 3: SedSVD proves to be more effective by applying subgraph embedding and embedding them for center nodes.

5. Discussion

To refine the vulnerability detection level, our study provides a statement-level vulnerability detection framework which can not only locate vulnerabilities but also point out the corresponding vulnerable code elements. Besides, we propose subgraph embedding, which has been rarely explored. Experimental results have demonstrated the advantage of our method. We infer that it may positively affect other detection systems.

We also put forward a new approach to generate more accurate ground truth label which focus specific code elements modified by patches. Although there still remains room for improvement, we believe this will mitigate the problem of low accuracy of ground truth label since it has been mentioned in numerous related papers. In the future, we will continue refining its accuracy.

6. Threats to validity

In this section, we discuss potential threats to the validity of our approach including external threats and internal threats.

External threats. First, since our model mainly aims at C/C++ programs, the detection results may be limited. But since the model we design does not rely on program language, it is expandable and scalable. Second, our dataset, which is constructed from NVD and SARD, cannot fully represent real-world projects. This may hinder our approach from identifying more complete and real vulnerabilities. Third, as for the accuracy of ground truth label we generated, although we have made refinement, we only consider line deletion in *diff* files. Vulnerabilities whose *diff* file only contain line addition are excluded actually.

Internal threats. First, due to subgraph embedding, the model may process one node for several times, resulting the redundancy of operation and the decrease of efficiency. Second, although we have gained some insights of the effectiveness of applying center nodes, more investigations are needed to explain it and explore different sets of center nodes selecting criterion and their impacts on model performance. Third, our model may be easily over-fit since there are too many hyper parameters to be adjusted. Dropout, factorization, multi-relation parameter sharing and other steps have been taken to mitigate this problem.

7. Related work

In this section, we review the two aspects most related to our work: code embedding and deep learning networks for vulnerability detection and highlight our novelty.

7.1. Code embedding

There exist several methods to embed and represent source code [2], including sequence-based, text-based, AST-based, graph-based and so forth. Sequence-based methods [3] mean extracting code features like identifiers, characters, tokens and so forth [4]. Text-based methods [5], however, treat source code as texts and quantify feature words extracted from them to represent text information. While these two methods mainly focus on local code elements and are inclined to neglect syntactic relations between code elements or statements, AST-based techniques [6,7], however, as the name implies, adopt AST, a tree representing the abstract syntactic structure of the source code [56], to extract hierarchical information of source code, thus gaining a better result.

Although in these methods, the syntactic and semantic features of the code are preserved to a certain extent, it has been proven that embedding code using graphs allows for more effective abstraction of deep code features. Common graphs consist of CFG, specifying the execution sequence of statements, and PDG, a composition of CDG and DDG, visualizing logical flow inside the program and constraint relationships between data. Benoit, T., J.Y. Marion, and S. Bardin. [8] focus on the shallow semantics in CFG to solve the problem of recovering the compiling chain generating a given stripped binary code. Ullah, F., et al. [9] extracts a set of weighted code features on the basis of PDG generated before.

Since AST, CFG, and PDG have their own priorities, Yamaguchi, F., et al. [34] creatively integrates them together and generates CPG which is regarded as one of the most effective way to express source code. There have been approaches using CPG [2,23]. What makes our work distinctive, however, is that we manage locating nodes at the whole context and embedding them at subgraph-level which has been rarely explored yet.

7.2. Deep learning networks used for vulnerability detection

As for sequence-based methods, CNN and RNN are the most popular ones. Since the strength of CNN is extracting local information while RNN does well in storing long-time information, Li, X.,et al. [12] puts forward a hybrid neural network model which combined them together so that the former can learn local vulnerability features and the latter is utilized to learn global features. Similar to them, Cao, D., et al. [13] chooses Long Short-Term Memory (LSTM) [57], however, a variant of RNN which is capable of solving the problem of long-term dependence in RNN.

While CNN and RNN are popular as well when embedding code using text, TextCNN [14], TextRNN [15] and TextRCNN [16] are more widely used since they are designed specially for text information. Dong Y., et al. [58] applies TextCNN to process text features extracted from source code.

Whereas, for AST-based ones, neural networks with tree-structure are more suitable. Zhang, J., et al. [17] advances AST-based Neural Network (ASTNN) [59] to first traverse each AST generated from code and then encodes them into vectors. Cai, Z., L. Lu, and S. Qiu [18] applies Tree-based Convolutional Neural Network (TBCNN) [60] to present a new method of Tree-Based-Embedding (TBE) to predict cross-project defects.

Apparently, for graph-based ones, neural networks on graphs are the best choice [19]. GGNN has been widely applied. For each node, it adopts GRU to update its embedding by assimilating all information of its neighbor nodes to avoid considering nodes in isolation [23]. With the aim to deal with the sequences forward and backward to obtain a deeper representation of code, Sc, A., et al. [25] applies BGNN by adding a backward edge. Zhou, Y., et al. [54], however, adopts GGRN to embed their graph and adds a convolution layer to select sets of nodes and features that are relevant to the current graph-level task.

Since our work distinguishes different types, RGCN is suitable to process the graph data. Although similar to Zhuang, Y., et al. [26], which applies CGCN to process heterogeneous graphs, our model outperforms it with 38.84% higher on F1 as reported in Section 4.1.

8. Summary and prospect

We propose SedSVD, a graph-based vulnerability detection framework at statement-level. In this work, we present a novel method to embed nodes and edges to learn more features from CPG. First, we utilize subgraph embedding, which has been slightly explored, to better serve statement-level vulnerability detection. Plus, we propose the idea of center nodes which can manifest vulnerability characteristics well and help our model get closer to vulnerabilities. Refinement is also made on the accuracy of ground truth label. Experimental results have proven our work to be more effective.

In future work, we will further improve SedSVD in four aspects: First, on the basis of subgraph-level embedding, reduce redundant operation on same data and improve efficiency. Second, expand SedSVD to suit more program languages. Third, explore different center nodes selecting criteria to promote the performance of our model. Fourth, research into each graph structure and utilize the specific vulnerability mode it reveals to optimize our model.

CRediT authorship contribution statement

Yukun Dong: Conceptualization, Methodology, Writing – review & editing, Funding acquisition, Supervision, Project administration. **Yeer Tang:** Conceptualization, Methodology, Software, Validation, Data curation, Writing – original draft, Visualization. **Xiaotong Cheng:** Software, Investigation, Validation. **Yufei Yang:** Software, Investigation, Validation. **Shuqi Wang:** Software, Investigation, Validation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The data that has been used is confidential.

Acknowledgments

This work was supported by the Shandong Provincial Natural Science Foundation (ZR2021MF058).

References

[1] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, Z. Chen, SySeVR: A framework for using deep learning to detect software vulnerabilities, IEEE Trans. Dependable Secure Comput. PP (2021) 1, http://dx.doi.org/10.1109/TDSC.2021.3051525.

[2] M. Gu, H. Sun, D. Han, S. Yang, W. Cao, Z. Guo, C. Cao, W. Wang, Y. Zhang, Software security vulnerability mining based on deep learning, Journal of Computer Research and Development 58 (10) (2021) 2140, http://dx.doi.org/10.7544/issn1000-1239.2021.20210620, https://crad.ict.ac.cn/CN/abstract/article_4507.shtml.

[3] T. Nguyen, T. Le, K. Nguyen, O.d. Vel, P. Montague, J. Grundy, D. Phung, Deep cost-sensitive kernel machine for binary software vulnerability detection, in: Pacific-Asia Conference on Knowledge Discovery and Data Mining, Springer, 2020, pp. 164–177.

[4] X. Li, L. Wang, Y. Xin, Y. Yang, Y. Chen, Automated vulnerability detection in source code using minimum intermediate representation learning, Appl. Sci. 10 (2020) 1692, http://dx.doi.org/10.3390/app10051692.

[5] L. Wang, X. Li, R. Wang, Y. Xin, G. Mingcheng, Y. Chen, PreNNsem: A heterogeneous ensemble learning framework for vulnerability detection in software, Appl. Sci. 10 (2020) 7954, http://dx.doi.org/10.3390/app10227954.

[6] Z. Zhuo, T. Cai, X. Zhang, F. Lv, Long short-term memory on abstract syntax tree for SQL injection detection, IET Softw. 15 (2) (2021) 188–197.

[7] H. Feng, X. Fu, H. Sun, H. Wang, Y. Zhang, Efficient vulnerability detection based on abstract syntax tree and deep learning, in: IEEE INFOCOM 2020-IEEE Conference on Computer Communications Workshops, INFOCOM WKSHPS, IEEE, 2020, pp. 722–727.

[8] T. Benoit, J.-Y. Marion, S. Bardin, Binary level toolchain provenance identification with graph neural networks, in: 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER, IEEE, 2021, pp. 131–141.

[9] F. Ullah, M.R. Naeem, A.S. Bajahzar, F. Al-Turjman, IoT-based cloud service for secured android markets using PDG-based deep learning classification, ACM Trans. Internet Technol. (TOIT) 22 (2) (2021) 1–17.

[10] G. Huang, Z. Liu, L. Van Der Maaten, K.Q. Weinberger, Densely connected convolutional networks, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2017, pp. 4700–4708.

[11] R. Pascanu, C. Gulcehre, K. Cho, Y. Bengio, How to construct deep recurrent neural networks, 2013, arXiv preprint arXiv:1312.6026.

[12] X. Li, L. Wang, Y. Xin, Y. Yang, Q. Tang, Y. Chen, Automated software vulnerability detection based on hybrid neural network, Appl. Sci. 11 (7) (2021) 3201.

[13] D. Cao, J. Huang, X. Zhang, X. Liu, Ftclnet: convolutional lstm with Fourier transform for vulnerability detection, in: 2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom, IEEE, 2020, pp. 539–546.

[14] Y. Chen, Convolutional neural network for sentence classification, University of Waterloo, 2015.

[15] P. Liu, X. Qiu, X. Huang, Recurrent neural network for text classification with MultiTask learning, 2016.

[16] S. Lai, L. Xu, K. Liu, J. Zhao, Recurrent convolutional neural networks for text classification, in: National Conference on Artificial Intelligence, 2015.

[17] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, X. Liu, A novel neural source code representation based on abstract syntax tree, in: 2019 IEEE/ACM 41st International Conference on Software Engineering, ICSE, IEEE, 2019, pp. 783–794.

[18] Z. Cai, L. Lu, S. Qiu, An abstract syntax tree encoding method for cross-project defect prediction, IEEE Access 7 (2019) 170844–170853.

[19] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, S.Y. Philip, A comprehensive survey on graph neural networks, IEEE Trans. Neural Netw. Learn. Syst. 32 (1) (2020) 4–24.

[20] Y. Li, D. Tarlow, M. Brockschmidt, R. Zemel, Gated graph sequence neural networks, 2015, arXiv preprint arXiv:1511.05493.

[21] H. Zhu, F. Feng, X. He, X. Wang, Y. Li, K. Zheng, Y. Zhang, Bilinear graph neural network with neighbor interactions, 2020, arXiv preprint arXiv:2002.03575.

[22] M. Alqarni, A. Azim, Software source code vulnerability detection using advanced deep convolutional neural network, in: Proceedings of the 31st Annual International Conference on Computer Science and Software Engineering, 2021, pp. 226–231.

[23] S. Chakraborty, R. Krishna, Y. Ding, B. Ray, Deep learning based vulnerability detection: Are we there yet, IEEE Trans. Softw. Eng. (2021).

[24] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, Y. Bengio, Learning phrase representations using RNN encoder-decoder for statistical machine translation, 2014, arXiv preprint arXiv:1406.1078.

[25] S. Cao, X. Sun, L. Bo, Y. Wei, B. Li, Bgnn4vd: Constructing bidirectional graph neural-network for vulnerability detection, Inf. Softw. Technol. 136 (2021) 106576.

[26] Y. Zhuang, S. Suneja, V. Thost, G. Domeniconi, A. Morari, J. Laredo, Software vulnerability detection via deep learning over disaggregated code graph representation, 2021, arXiv preprint arXiv:2109.03341.

[27] T. Xie, J.C. Grossman, Crystal graph convolutional neural networks for an accurate and interpretable prediction of material properties, Phys. Rev. Lett. 120 (14) (2018) 145301.

[28] S. Jeon, H.K. Kim, AutoVAS: An automated vulnerability analysis system with a deep learning approach, Comput. Secur. 106 (2021) 102308.

[29] S. Suneja, Y. Zheng, Y. Zhuang, J. Laredo, A. Morari, Learning to map source code to software vulnerability using code-as-a-graph, 2020, arXiv preprint arXiv:2006.08614.

[30] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, Y. Zhong, Vuldeepecker: A deep learning-based system for vulnerability detection, 2018, arXiv preprint arXiv:1801.01681.

[31] Z. Li, D. Zou, S. Xu, Z. Chen, Y. Zhu, H. Jin, Vuldeelocator: A deep learning-based fine-grained vulnerability detector, IEEE Trans. Dependable Secure Comput. (2021).

[32] J. Tian, W. Xing, Z. Li, BVDetector: A program slice-based binary code vulnerability intelligent detection system, Inf. Softw. Technol. 123 (2020) 106289.

[33] D. Hin, A. Kan, H. Chen, M.A. Babar, LineVD: Statement-level vulnerability detection using graph neural networks, 2022, arXiv preprint arXiv:2203.05181.

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

[35] T. Mikolov, K. Chen, G. Corrado, J. Dean, Efficient estimation of word representations in vector space, 2013, arXiv preprint arXiv:1301.3781.

[36] M. Schlichtkrull, T.N. Kipf, P. Bloem, R.v.d. Berg, I. Titov, M. Welling, Modeling relational data with graph convolutional networks, in: European Semantic Web Conference, Springer, 2018, pp. 593–607.
[37] Joern, https://joern.io/.
[38] T. Xiao, J. Guan, S. Jian, Y. Ren, J. Zhang, B. Li, Software vulnerability detection method based on code property graph and bi-gru, Journal of Computer Research and Development 58 (8) (2021) 1668, http://dx.doi.org/10.7544/issn1000-1239.2021.20210297, https://crad.ict.ac.cn/CN/abstract/article_4473.shtml.
[39] T. Nguyen, G.T. Nguyen, T. Nguyen, D.-H. Le, Graph convolutional networks for drug response prediction, IEEE/ACM Trans. Comput. Biol. Bioinform. 19 (1) (2021) 146–154.
[40] A. Shafee, A. Khoshghalb, Particle node-based smoothed point interpolation method with stress regularisation for large deformation problems in geomechanics, Comput. Geotech. 141 (2022) 104494.
[41] M. Gori, G. Monfardini, F. Scarselli, A new model for learning in graph domains, in: Proceedings. 2005 IEEE International Joint Conference on Neural Networks, Vol. 2, no. 2005, 2005, pp. 729–734.
[42] F. Scarselli, M. Gori, A.C. Tsoi, M. Hagenbuchner, G. Monfardini, The graph neural network model, IEEE Trans. Neural Netw. 20 (1) (2008) 61–80.
[43] Patches for CVE-2022-0156, https://github.com/vim/vim/commit/9f1a39a5d1cd7989ada2d1cb32f97d84360e050f.
[44] CVE-2022-0156, https://nvd.nist.gov/vuln/detail/CVE-2022-0156.
[45] National vulnerability database, https://nvd.nist.gov/.
[46] CVE, http://cve.mitre.org/.
[47] Nist software assurance reference dataset, https://samate.nist.gov/SARD.
[48] Common Weakness Enumeration, https://cwe.mitre.org/.
[49] J.M. Klusowski, Y. Wu, Estimating the number of connected components in a graph via subgraph sampling, Bernoulli 26 (3) (2020) 1635–1664.
[50] torch_geometric.data, https://pytorch-geometric.readthedocs.io/en/latest/modules/data.html.
[51] J. Zarzar, S. Giancola, B. Ghanem, PointRGCN: Graph convolution networks for 3D vehicles detection refinement, 2019, arXiv preprint arXiv:1911.12236.
[52] S. Feng, H. Wan, N. Wang, M. Luo, BotRGCN: Twitter bot detection with relational graph convolutional networks, in: Proceedings of the 2021 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining, 2021, pp. 236–239.
[53] R. Lin, Z. Zhou, S. You, R. Rao, C.-C.J. Kuo, From two-class linear discriminant analysis to interpretable multilayer perceptron design, 2020, arXiv preprint arXiv:2009.04442.
[54] Y. Zhou, S. Liu, J. Siow, X. Du, Y. Liu, Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks, Adv. Neural Inf. Process. Syst. 32 (2019).
[55] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al., Pytorch: An imperative style, high-performance deep learning library, Adv. Neural Inf. Process. Syst. 32 (2019).
[56] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, L. Bier, Clone detection using abstract syntax trees, in: Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272), IEEE, 1998, pp. 368–377.
[57] K. Greff, R.K. Srivastava, J. Koutník, B.R. Steunebrink, J. Schmidhuber, LSTM: A search space odyssey, IEEE Trans. Neural Netw. Learn. Syst. 28 (10) (2016) 2222–2232.
[58] D. Yukun, L. Haojie, W. Xinxin, T. Daolong, Software defect prediction based on features fusion of program structure and semantics, Comput. Eng. Appl. 58 (16) (2022) 84–93.
[59] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, X. Liu, A novel neural source code representation based on abstract syntax tree, in: 2019 IEEE/ACM 41st International Conference on Software Engineering, ICSE, IEEE, 2019, pp. 783–794.
[60] L. Mou, G. Li, Z. Jin, L. Zhang, T. Wang, TBCNN: A tree-based convolutional neural network for programming language processing, 2014, arXiv preprint arXiv:1409.5718.