2022 IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom) | 978-1-6654-9425-0/22/$31.00 ©2022 IEEE | DOI: 10.1109/TrustCom56396.2022.00070

Source Code Vulnerability Detection Using Vulnerability Dependency Representation Graph

Hongyu Yang
School of Safety Science and Engineering
School of Computer Science and Technology
Civil Aviation University of China
Tianjin, China
yhyxlx@hotmail.com

Haiyun Yang
School of Computer Science and Technology
Civil Aviation University of China
Tianjin, China
secyhy@163.com

Liang Zhang
School of Information
The University of Arizona
Tucson, USA
liangzh@arizona.edu

Xiang Cheng
School of Information Engineering
Yangzhou University
Yangzhou, China
huozhai9527@126.com

***Abstract*—Aiming at the fact that the existing source code vulnerability detection methods did not explicitly maintain the semantic information related to the vulnerability in the source code, which made it difficult for the vulnerability detection model to extract the vulnerability sentence features and had a high detection false positive rate, a source code vulnerability detection method based on the vulnerability dependency graph is proposed. Firstly, the candidate vulnerability sentences of the function were matched, and the vulnerability dependency representation graph corresponding to the function was generated by analyzing the multi-layer control dependencies and data dependencies of the candidate vulnerability sentences. Secondly, abstracted the function name and variable name of the code sentences node and generated the initial representation vector of the code sentence nodes in the vulnerability dependency representation graph. Finally, the source code vulnerability detection model based on the heterogeneous graph transformer was used to learn the context information of the code sentence nodes in the vulnerability dependency representation graph. In this paper, the proposed method was verified on three datasets. The experimental results show that the proposed method have better performance in source code vulnerability detection, and the recall rate is increased by 1.50%~22.27%, and the F1 score is increased by 1.86%~16.69%, which is better than the existing methods.**

Keywords—source code representation, vulnerability dependency representation graph, source code vulnerability detection, heterogeneous graph transformer

I. Introduction

Software vulnerabilities are the cause of cyber attacks, it is necessary to detect and fix these vulnerabilities at the source code level to avoid possible vulnerabilities[1,2]. Most vulnerability detection methods employ vulnerability features or template rules that are manually defined by security experts. These methods not only have a high false positive rate and low vulnerability coverage but also template rules need to be dynamically updated with the emergence of new vulnerability patterns[3].

With the development of deep learning, researchers try to use deep learning methods to detect source code vulnerabilities[4][5]. The architecture of the vulnerability detection model based on deep learning depends on the representation structure of the source code[6][7]. Existing source code representation methods can be divided into token-based methods and graph-based methods[8].

Token-based methods treat source code as token sequences. Russell[9] extracts the source code feature vector through CNN and uses the random forest algorithm to detect and classify source code vulnerabilities, but it is difficult to extract the features of long token sequences and don't consider the rich semantic information in the source code. The VulDeePecker method proposed by Li[10] uses vulnerability-related sensitive APIs to slice the source code and selects BLSTM to model the source code, but VulDeePecker only used semantic information arising from data dependencies. The SySeVR method proposed by Li[3] uses four vulnerability syntax characteristics to obtain candidate vulnerability sentences and slice the source code by analyzing the data dependency and control dependency information of the candidate vulnerability sentences. In addition, SySeVR uses BGRU for source code vulnerability detection. However, this method don't explicitly maintain dependencies in source code slices, which make it difficult for the model to learn and reason about source code semantic information.

Graph-based approaches treat source code as a graph structure. Graph-based models use different types of code to represent graphs, such as abstract syntax tree (AST), data flow graph (DFG), and control flow graph (CFG), etc[11]. The Devign method proposed by Zhou[12] uses a joint graph that combined AST, CFG, DFG, and code execution order to represent source code, and explicitly point out code execution flow and data propagation flow by connecting edges. Devign uses the gated graph neural network (GGNN)[13] to learn the code representation of the joint graph for further source code vulnerability detection. The Reveal method proposed by Chakraborty[7] uses a code property graph (CPG) to represent the source code and encodes the node type of the code property graph in the eigenvector of the vertex. Reveal uses GGNN to extract the features of CPG, and finally performs source code vulnerability detection. The above two graph-based methods simply use one or more representation graphs of the source code to represent the source code and don't further analyze the nodes and edges related to the vulnerability sentences in the representation graph, so that the model can't fully learn and reason about the vulnerability sentences context.

In order to alleviate the shortcomings of the above methods, this paper proposes a vulnerability dependency representation graph-based source code vulnerability detection method (SDBV). The SDBV contributions are as follows:

- A source code representation structure for the function called vulnerability dependency representation graph (VDRG) is designed and proposed. In order to enable the source code vulnerability detection model to fully learn the context information of candidate vulnerability sentences, candidate vulnerability sentences is further analyzed on the control dependency graph and data dependency graph. The nodes of the graph and the data dependency graph are divided into 2 types, and the edges are divided into 4

This work was supported by the National Natural Science Foundation of China (No. U1833107).

types. The VDRG is obtained by merging the processed control dependency graph and data dependency graph.

- This paper proposes a source code vulnerability detection model based on the heterogeneous graph transformer (HGT)[14]. The vulnerability detection model can effectively model the heterogeneity of VDRG, it accurately learns the context information of candidate vulnerability sentences through the heterogeneous attention mechanism that depends on edge types and node types. This paper verifies the effectiveness of the proposed source code vulnerability detection model on three datasets.

II. Method Overview

The SDBV method proposed in this paper is divided into 3 stages: VDRG generation, node embedding, and vulnerability detection. The framework of SDBV is shown in Fig. 1, and the contents of each stage are as follows:

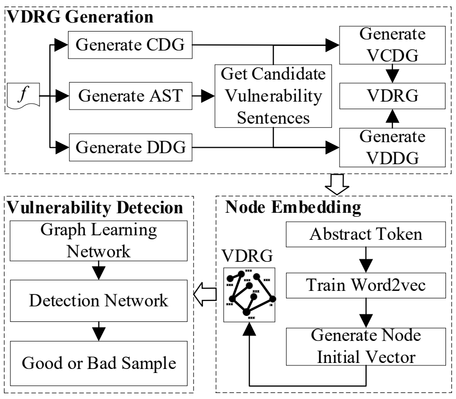

Fig. 1. The SDBV architecture.

1) VDRG generation: Firstly, use the code analysis tool Joern to generate the AST, control dependency graph (CDG), and data dependency graph (DDG) of the function *f*. Secondly, candidate vulnerability sentences are obtained through matching vulnerability syntax characteristics on the AST. Then, analyze the control dependencies and data dependencies of the candidate vulnerability sentences, modify the nodes and edges of the control dependency graph and data dependency graph, and obtain the vulnerability control dependency graph (VCDG) and vulnerability data dependency graph (VDDG) respectively. Finally, combine VCDG and VDDG to obtain the VDRG corresponding to the function.

2) Node embedding: Firstly, the code token is abstracted by renaming variable and function names in function slices. Secondly, the word vector model word2vec is trained using the code tokens of all functions in the dataset. Finally, use the pre-trained word2vec model generation node to generate the initial vector.

3) Vulnerability detection: The source code vulnerability detection model is constructed based on the heterogeneous graph transformer. The model is divided into two parts: the graph learning network and the detection network. The part of the graph learning network takes VDRG as the input, and each node actively exchanges information with its neighbor nodes to update its own vector representation. The detection network part obtains the global vector representation of the VDRG, then further learns the low-dimensional features of the VDRG global representation vector, and finally obtains the source code vulnerability detection result through the fully connected network.

III. VDRG Generation

The representation of the function-level source code proposed in this paper can be formally denoted as $f_{vdrg}=(V, E, X, D)$, where V represents a set of nodes, and each node is a code statement. E represents a set of directed edges, where a certain type of relationship exists between the two nodes connected by each directed edge. X represents the node attribute set, which saves the type attribute of the node. There are 2 types of node types, the node corresponding to the potential vulnerability statement and the node corresponding to the benign statement. D represents the attribute set of the edge, which saves the type attribute of the edge. There are 4 kinds of edge types in VDRG, namely control dependency edge (CDE), vulnerability control dependency edge(VCDE), and vulnerability data dependency edge (VDDE) and data dependency edge (DDE). CDE is the edge in the control dependency graph, and DDE is the edge in the data dependency graph. The vulnerability-related dependency edges in the control dependency graph and data dependency graph are denoted as VCDE and VDDE respectively.

In the VDRG generation step, the function code slice *f* is used as input, the output of algorithm1 are several VDRGs corresponding to *f*. Algorithm1 shows the generation process of VDRG. Firstly, generate the abstract syntax tree corresponding to the function code fragment *f*, match the vulnerability syntax feature on the abstract syntax tree, and obtain several code statements with potential vulnerability. Then, the control dependencies and data dependencies related to each statement are calculated to generate the corresponding VDRG, so the single function slice corresponds to several VDRGs, and the final output of algorithm1 is the VDRG set.

```
void example_uaf(char * content)
{
    int flag1 = -1;
    int flag2 = 1;
    char * data;
    int len = strlen(content)
    data = (char *)malloc(len);
    strncpy(data, content, len);
    if (flag1<0)
    {
        if(flag2<0)
        {
            free(data);
            printf("%d\n",len);
        }
    }
    printf("%s\n",data);
}
```

Fig. 2. A function sample

Lines 1 to 5 of the algorithm1 generates the abstract syntax tree f_{ast} corresponding to the function slice *f*, the data dependency graph f_{ddg}, and the control dependency graph f_{cdg}. When performing vulnerability syntax feature matching on AST, this paper uses the library function call (FC), array usage (AU), pointer usage (PU), arithmetic expressions (AE) 4 characteristics proposed by SySeVR, qualified statements are called syntax-based vulnerability candidates (SyVC)[3]. If during the traversal of the AST, the node conforms to one of the above 4 vulnerability syntax characteristics, the code statement corresponding to the current node is added to S_{SyVC}. Fig. 3 shows the partial AST corresponding to the code fragment of Fig. 2, where "*free*(*data*)" successfully matches the library function call vulnerability characteristic.

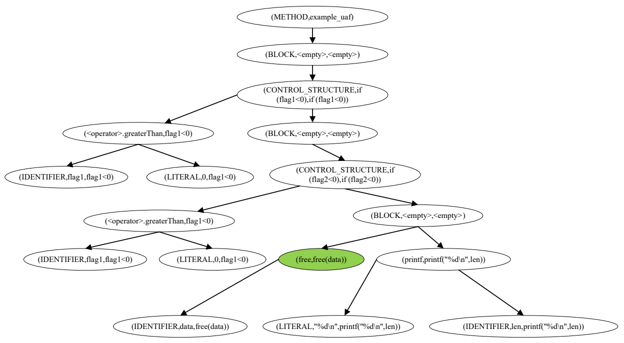
(METHOD,example_uaf)
(BLOCK,<empty>,<empty>)
(CONTROL_STRUCTURE,if (flag1<0),if (flag1<0))
(<operator>.greaterThan,flag1<0)
(BLOCK,<empty>,<empty>)
(IDENTIFIER,flag1,flag1<0)
(LITERAL,0,flag1<0)
(CONTROL_STRUCTURE,if (flag2<0),if (flag2<0))
(<operator>.greaterThan,flag1<0)
(BLOCK,<empty>,<empty>)
(IDENTIFIER,flag1,flag1<0)
(LITERAL,0,flag1<0)
(free,free(data))
(printf,printf("%d\n",len))
(IDENTIFIER,data,free(data))
(LITERAL,"%d\n",printf("%d\n",len))
(IDENTIFIER,len,printf("%d\n",len))

Fig. 3. The partial AST corresponding to the function sample.

In this method, each SyVC corresponds to a VDRG, and the VDRG is combined from the further processed control dependency graph and data dependency graph. Lines 11-24 of the algorithm1 process the control dependency graph corresponding to a certain SyVC. The further processed control dependency graph is recorded as f_{vcdg} in the algorithm1. Each f_{vcdg} is initialized to the control dependency graph corresponding to the current function slice, and then according to the SyVC in the context, processing the original control dependency graph to obtain f_{vcdg}. For a vulnerable statement, its direct control dependency determines execution flow direction, and the context of its direct control dependency also affects its execution. Therefore, the control dependency of SyVC is recursively calculated in the algorithm1, i.e., the current SyVC corresponds to the control dependency. If the control dependent node set S_c is not empty, a statement element is removed from S_c cyclically, and then each edge in f_{vcdg} is traversed until the destination node of an edge is the currently processed statement, and the attribute of the destination node is set to "bad", indicating that this node is a potential vulnerability node, and setting the attribute of the edge to the vulnerability control dependent edge, indicating that the execution flow of the potential vulnerability node is related to this edge. At this time, the source node of the edge is the control dependent node of the current statement, and the source node corresponding to the edge is added to S_c , then calculate the control dependence of source node. If the above conditions are satisfied, the label of the edge in processing is set to the original control-dependent edge, and the attributes of other nodes are set to "good", which means these nodes no potential vulnerability. When S_c is empty, the control dependency graph is processed, and the f_{vcdg} of the current SyVC is obtained. Fig. 4 shows the VCDG corresponding to the SyVC "*free*(*data*)" library function call.

Lines 25-33 of the algorithm1 process the data dependency graph corresponding to function *f*. The further processed data dependency graph is recorded as f_{vddg} in the algorithm1. Each f_{vddg} is initialized as the data dependency graph corresponding to the current function slice, and then the data dependency of SyVC is analyzed, and the original control dependency graph is processed to obtain f_{vddg}. The variable used by a vulnerable statement also affects its execution effect, so the VDRG generation algorithm calculates its data dependency for each variable, i.e., analyzes and traces the source of the variable. For the SyVC currently being processed, traverse each edge in f_{vddg}, if the variable name is consistent with the label of an edge, i.e., the edge label of the data dependency graph is the variable name, then set the label of the edge as the vulnerability data dependent edge, i.e. VDDE. In other cases, the label of the edge is set as the original data dependency edge, i.e., DDE. After processing each variable contained in the current SyVC, the final vulnerability data dependency graph f_{vddg} is obtained.

Algorithm 1: Generate VDRG
Input: Function $f=\{s_1,\dots, s_i\}$, s_i represents a statement within a function;
Output: A set *S* with VDRG as elements;

```
1  S←∅;
2  Generate the abstract syntax tree $f_{ast}$ corresponding to the function f;
3  Perform vulnerability syntax feature matching on $f_{ast}$ to generate the SyVC set $Ss_{yVC}$ corresponding to $f_{ast}$;
4  Generate the control dependency graph $f_{cdg}$ corresponding to the function f;
5  Generate the data dependency graph $f_{ddg}$ corresponding to the function f;
6  for each SyVC ∈ $Ss_{yVC}$ do
7    $f_{vcdg}$←$f_{cdg}$;
8    $f_{vddg}$←$f_{ddg}$;
9    Let the control of SyVC depend on the node set Sc←∅;
10   Sc←Sc ∪ SyVC;
11   while Sc do
12     remove element SyVC from Sc;
13     for each $edge_{vcdg}$ ∈ $f_{vcdg}$ do
14       if $edge_{vcdg}.dst$ == SyVC and $edge_{vcdg}.label$ ≠ 'vcde'
15         Sc←Sc ∪ $edge_{vcdg}.src$;
16         $edge_{vcdg}.dst.type$←'bad'
17         $edge_{vcdg}.label$← 'vcde';
18       else if $edge_{vcdg}.dst \neq s_{pvs}$ and $edge_{vcdg}.label$ ≠ 'vcde'
19         $edge_{vcdg}.dst.type$←'good'
20         $edge_{vcdg}.label$← 'cde';
21       end if
22       $edge_{vcdg}.src.type$←'good'
23     end for
24   end while
25   for each var ∈SyVC do
26     for each $edge_{vddg}$ ∈ $f_{vddg}$ do
27       if var.name == $edge_{vddg}.label$ and $edge_{vddg}.label$ ≠ 'vdde'
28         $edge_{vddg}.label$← 'vdde';
29       else if $edge_{vddg}.label$ ≠ 'vdde'
30         $edge_{vddg}.label$← 'dde';
31       end if
32     end for
33   end for
34   $f_{vdrg}$←$f_{vcdg}$ ∪ $f_{vddg}$;
35   S←S ∪ $f_{vdrg}$;
36 end for
```

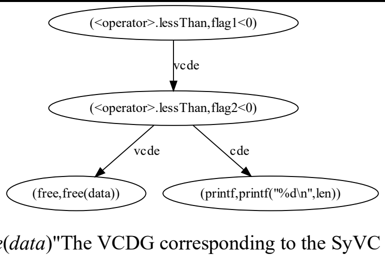
(<operator>.lessThan,flag1<0)
vcde
(<operator>.lessThan,flag2<0)
vcde
cde
(free,free(data))
(printf,printf("%d\n",len))

Fig. 4. "*free*(*data*)"The VCDG corresponding to the SyVC "*free*(*data*)".

After the control dependency graph and data dependency graph corresponding to SyVC are processed, the generated vulnerability control dependency graph f_{vcdg} and vulnerability data dependency graph f_{vddg} are merged into the vulnerability dependency graph f_{vdrg}, and the generated f_{vdrg} is added to the set *S*. When all SyVCs are processed, all VDRGs corresponding to function *f* is obtained. The VDRG corresponding to the SyVC "*free*(*data*)" library function call is shown in Fig. 5.

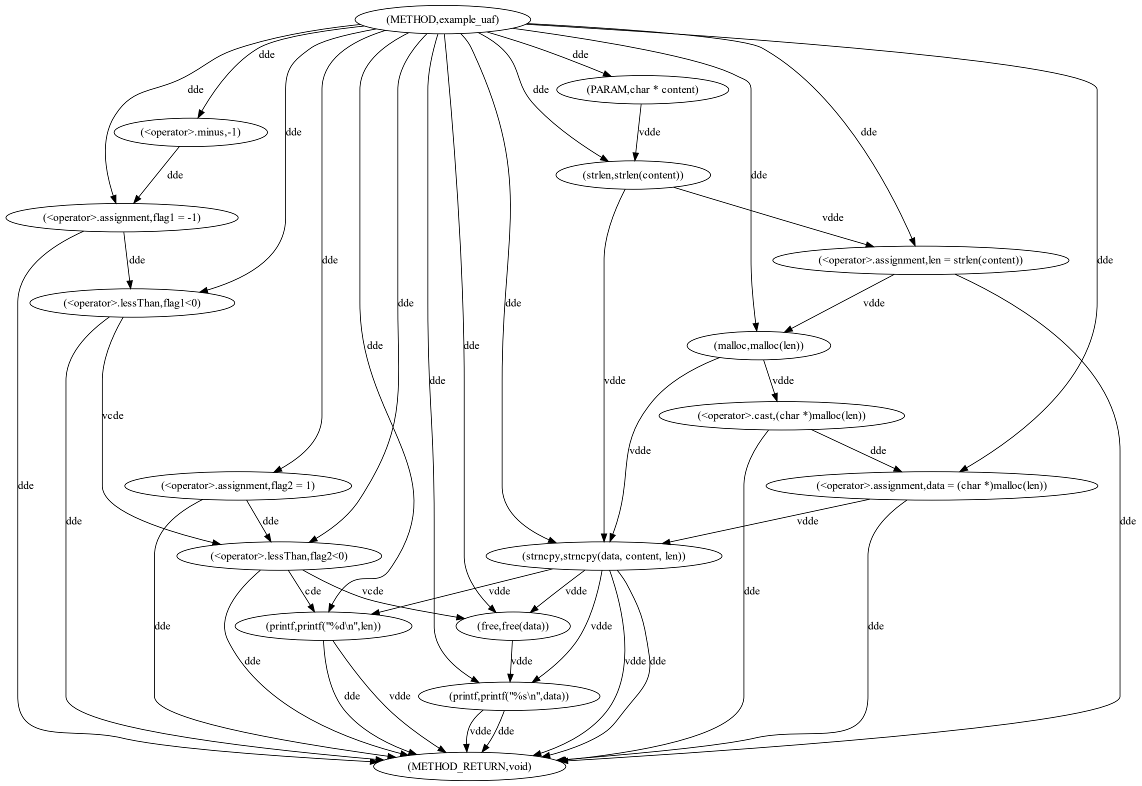
(METHOD,example_uaf)
(PARAM,char * content)
(<operator>.minus,-1)
(strlen,strlen(content))
(<operator>.assignment,flag1 = -1)
(<operator>.assignment,len = strlen(content))
(<operator>.lessThan,flag1<0)
(malloc,malloc(len))
(<operator>.cast,(char *)malloc(len))
(<operator>.assignment,flag2 = 1)
(<operator>.assignment,data = (char *)malloc(len))
(<operator>.lessThan,flag2<0)
(strncpy,strncpy(data, content, len))
(printf,printf("%d\n",len))
(free,free(data))
(printf,printf("%s\n",data))
(METHOD_RETURN,void)
dde
vdde
vcde
cde

Fig. 5. The VDRG corresponding to the SyVC "*free*(*data*)".

IV. Nodes Embedding

The essence of the graph neural network is to aggregate neighbor information through the exchange of information between nodes to obtain a new node representation[15]. The goal of the node embedding stage is to generate an initial representation vector for each node of the VDRG. The node embedding stage takes VDRG as input and goes through the following steps:

A. Token Abstraction

To prevent variable names and function names from affecting the learning of semantic relationships in the program. Firstly, the function *f* is split into a combination of single code token. Then the variable names are uniformly named *var1*, *var2*..., function names are uniformly named *fun1*, *fun2*…. Finally, traverse the nodes of the VDRG and modify the updated variable and function names.

B. Word2vec Training

Word2vec is a word embedding model, and its main idea is to construct a vector space so that words with similar contexts are close to each other in the vector space, and words with different contexts have a larger distance in the vector space. In this paper, the word2vec model is used to generate the representation vector of a single code token. First, the function that has been abstracted is divided into a combination of code tokens, then all tokens are organized into a vocabulary file. Finally, use the vocabulary file as the input of the word2vec model to train the word2vec model to obtain the best model parameters.

C. Node Initial Vector Generation

Traverse each node of the VDRG, and split the code sentence corresponding to the node into a combination of code tokens. For each code token, use the pre-trained word2vec to generate its representation vector, and add the representation vectors of all code tokens to get the initial node vector. In addition, to make the initial representation vector contain the type information of the node, a dimension is added at the beginning of the initial vector of the node. The value of this dimension is 1 for the node corresponding to the candidate vulnerability sentence, and the value of this dimension is 0 for other nodes.

V. Vulnerability Detection Model

A. Vulnerability Detection Model Architecture

After the previous processing, the source code function slices are represented as several graph structures, and each node of the graph has an initial representation vector. In this section, this paper proposes a vulnerability detection model for VDRG based on the heterogeneous graph transformer[14].

The model is divided into a graph learning network module and a vulnerability detection network module. The vulnerability detection model is shown in Fig. 6.

The model minimizes the loss over the training set to obtain optimal model parameters. When detecting a new function, after VDRG generation and node embedding, the generated VDRG is input into the pre-trained vulnerability detection model to obtain the detection result.

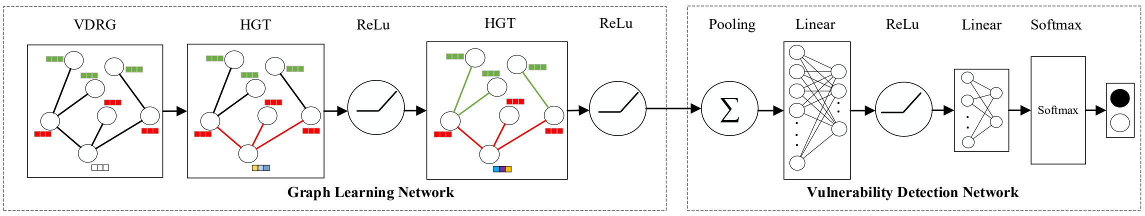

Fig. 6. Source code vulnerability detection model based on heterogeneous graph transformer.

B. Graph Learning Network

The graph learning network module is mainly responsible for reasoning and learning the syntactic and semantic information contained in the VDRG. The heterogeneous graph transformer calculates the attention of each edge by using the parameters depending on the node type and edge type maintains specific representations for different node types and edge types, and is suitable for VDRG modeling and learning.

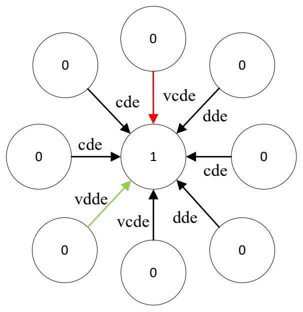

Fig. 7. Heterogeneous attention computation on VDRG

1) Heterogeneous Attention Computation

For the target node t, this step calculates the attention scores of all neighbor nodes s of t relative to t and denotes all the neighbors of s as $N(t)$, the number of attention heads as h, and the connection edge between t and s as e, l represents the number of layers, d is the dimension of the node representation vector, the function τ is used to obtain the node type, and the function ϕ is used to obtain the edge type.

$$K^i(s) = K\_Linear^i_{\tau(s)}(H^{(l-1)}[s]) \tag{1}$$

$$Q^i(t) = Q\_Linear^i_{\tau(t)}(H^{(l-1)}[t]) \tag{2}$$

$$ATT-head(s,e,t)^i = (K^i(s) W^{ATT}_{\phi(e)} Q^i(t)^T) \cdot \frac{\mu_{<\tau(s),\phi(e),\tau(t)>}}{\sqrt{d}} \tag{3}$$

$$Attention_{HGT}(s,e,t) = Softmax(\underset{\forall s \in N(t)}{} \underset{i\in[1,h]}{\|} ATT-head^i(s,e,t)) \tag{4}$$

For the calculation of the i-th attention head score, first use (1) to obtain the key vector K of the source node s, and the linear mapping used to calculate the key vector depends on the type of node, i.e., different types of nodes use different linear mappings, which enables nodes containing potential vulnerabilities and benign nodes to have different feature spaces, and these linear mappings are learnable to further optimize node representation. (2) is used to calculate the query vector Q, which is the same as (1).

The heterogeneous graph transformer introduces a transformation matrix W in (3), which depends on the edge type. HGT enables the 4 edge types of VDRG to be effectively distinguished, and this matrix W can be learned to further quantify the different effects of edge types.

For each attention head, after calculating the attention scores of target node t and neighbor s using (3), each attention head is connected by (4), and processed by the softmax function, so that the score on each attention head conforms to the probability distribution.

Fig.7 shows the process of heterogeneous attention computation on VDRG. The intermediate nodes are candidate vulnerability statement nodes, and the other nodes have no potential vulnerabilities. After the heterogeneous attention calculation, the nodes connected by VCDE and VDDE will get higher attention relative to the intermediate candidate vulnerability statement nodes, and other nodes will get lower attention.

2) Neighbor Information Passing

In this step, the neighbor node information s is transmitted to the target node t, and the edge relationship is also included in the calculation process. The message headers are calculated on each attention head using (5), and then the h message headers are concatenated using (6).

$$MSG-head(s,e,t)^i = M\_Linear^i_{\tau(s)}(H^{(l-1)}[s]) W^{MSG}_{\phi(e)} \tag{5}$$

$$Message_{HGT}(s,e,t) = \underset{i\in[1,h]}{\|} MSG-head^i(s,e,t) \tag{6}$$

3) Target Node Aggregation

The aggregated message vector is obtained by the weighted summation of the neighbors of the target node t and their corresponding attention scores. The specific calculation is shown in (7). (8) first processes the aggregated message vector through the activation function, then maps the aggregated message vector to the feature space of the target node type through the node type-specific linear layer, and finally adds the representation vector of the target node in the previous layer to obtain the updated the target node vector.

$$\tilde{H}^{(l)}[t] = \underset{\forall s\in N(t)}{\oplus} (Attention_{HGT}(s,e,t) \cdot Message_{HGT}(s,e,t)) \tag{7}$$

$$H^{(l)}[t] = A-Linear_{\tau(t)}(\sigma(\tilde{H}^{(l)}[t])) + H^{(l-1)}[t] \tag{8}$$

C. Detection Network

The detection network further extracts the features of the VDRG updated by the nodes to obtain the detection results. Firstly, through the global mean pooling layer, the global representation vector H of the VDRG is obtained, and the calculation method of the global mean pooling is shown in (9). The vector not only reflects the state of all nodes at this time but also contains data dependencies and control dependencies, reflecting the syntax and semantic information of VDRG as a whole.

$$H_{vdrg} = \frac{1}{N}\sum_{n=1}^{N} x_n \tag{9}$$

To further extract the features of the global representation vector, redundant features are removed, and only the features related to vulnerability information are retained. After the global mean pooling layer, a linear layer is added for global feature extraction. In this paper, the input dimension of this linear layer is the node dimension, and the output dimension is 1/2 of the original node dimension. After the global feature extraction, a fully connected linear layer is set, and the output dimension is 2. To make the detection result conform to the probability distribution, it is processed by the softmax function to obtain the final detection result.

VI. Experiments

A. Experiments Settings

1) Datasets

This paper uses 3 datasets to verify the effectiveness of the SDBV method. The first dataset is the Software Assurance Reference Dataset (SARD), whose data is partly derived from the semi-synthetic code of source programs in production environments and partly artificially synthesized code for academic research. The second dataset is the Devign dataset proposed by Zhou[13]. The Devign dataset extracts code submissions related to security issues from 4 projects. The third dataset is the Reveal dataset proposed by Chakraborty[7]. The Reveal dataset is generated by tracking historical vulnerabilities of two open source projects, Linux Debian Kernel and Chromium.

2) Experiments Environment

The CPU used in the experiment is Intel(R) Xeon(R) Gold 5215, the memory size is 48G, and the graphics card model is Tesla-V100. Furthermore, the number of attention heads of the heterogeneous graph transformer of the graph learning network module is set to 2. The ratio of the training set to the test set is 7:3, and the number of training rounds is 100.

3) Evaluation Standard

In this experiment, 4 indicators are used to evaluate the vulnerability detection performance, and the accuracy rate (A) refers to the proportion of correctly classified samples to the total samples. Precision (P) refers to the proportion of correct samples among all samples judged to be vulnerable. The recall rate (R) refers to the proportion of successfully detected vulnerability samples to all vulnerability samples. The F1-score (F1) is the harmonic mean of precision and recall, reflecting the overall performance of the model.

B. Data Preprocessing

Before conducting formal experiments, the source programs in the dataset are first processed into the VDRG representation proposed in this paper. Data Preprocessing is divided into the following 4 steps.

1) API preprocessing

When performing vulnerability syntax feature matching of library function call types, this paper uses 818 function calls collected by SySeVR, which cover 106 CWE vulnerability types. But in actual programs, it is possible to use different forms of the API through macro definitions. Therefore it is necessary to consider the different form transformations of each API.

2) Function Slicing and Label Setting

For the SARD dataset, this paper uses Ecplise CDT to obtain all the declarations in the source code file, and then traverses each declaration to determine whether it is a function declaration. If it is, get its function definition, and write it to the function slice file. At the same time, it is judged whether the function declaration contains the "good" or "bad" keyword. If so, the corresponding label values 0 and 1 are written into the slice file name. 1 means that the function slice is vulnerable, and 0 means no vulnerability. The other two datasets already contain labels and require no further processing.

After the preprocessing step, the information of SARD dataset is shown in Table 1, and the information of Devign dataset and Reveal dataset is shown in Table 2.

TABLE I. Preprocessed SARD dataset information

CWE Types	Description	Number of Function Slices	Number of VDRGs	The Number of VDRGs with Vulnerabilities	Number of Benign VDRGs
CWE20	Improper input validation	3452	4015	846	3169
CWE78	Command injection vulnerability	17000	18420	4200	14220
CWE129	Improper array index validation	11208	10019	2977	7042
CWE190	Integer overflow	25913	28943	6925	22018
CWE400	resource exhaustion	9990	10023	2744	7279
CWE787	out-of-bounds write	14797	14980	4210	10770
CWE789	Out of control memory allocation	7300	8167	1241	6926

TABLE II. Preprocessed Devign and Reveal dataset information

Dataset Name	Number of Function Slices	Number of VDRGs	The Number of VDRGs with Vulnerabilities	Number of Benign VDRGs
Devign	27313	28760	13754	15006
Reveal	22725	20109	2753	17356

C. Experimental Results and Analysis

To verify the effectiveness of SDBV, 5 advanced methods of source code vulnerability detection are selected for comparison with SDBV in this experiment. The methods proposed by Russell et al.[9], SySeVR[3], and VulDeepecker[10] belong to sequence models. These methods treat the source code as a sequence of the token. Reveal[7], Devign[12] represent the source code as graph structure is then processed further. The comparison results on the SRAD dataset are shown in Tables 3 to 6, and the comparison results on the Devign dataset and Reveal dataset are shown in Fig. 8 and Fig. 9 respectively. The experimental results show that SDBV performs well in all datasets, and the key indicators are better than other methods in most datasets. The accuracy rate is 2.23%~7.39% higher than the best indicator, and the accuracy rate is increased by 2.22%~9.56%, the recall rate is increased by 1.50%~22.27%, and the F1 score is increased by 1.86%~16.69%.

The experiments on the SARD dataset include 7 vulnerability types, and SDBV outperforms all other methods in 4 indicators of the 5 vulnerability types. The occurrence of these 5 vulnerabilities is closely related to the context. The data dependencies and multi-layer control dependencies contained in the VDRG representation can accurately reflect the context in which the vulnerability occurs, so it has high detection performance.

TABLE III. COMPARISON RESULTS OF THE ACCURACY A(%) OF EACH METHOD ON THE SARD DATASET

Method Name	CWE20	CWE78	CWE129	CWE190	CWE400	CWE787	CWE789
Russell et al.	72.45	92.71	85.97	87.49	88.58	74.42	89.24
VuldeePecker	75.19	95.71	83.31	86.97	91.26	75.62	89.85
SySeVR	78.58	97.01	90.25	93.00	94.59	85.69	91.90
Devign	77.93	96.66	90.19	95.53	96.47	85.09	91.97
Reveal	78.93	97.28	93.49	97.32	**97.99**	85.58	92.46
SDBV	**81.42**	**98.37**	**96.24**	**98.17**	97.93	**87.59**	**94.72**

TABLE IV. COMPARISON RESULTS OF THE ACCURACY P(%) OF EACH METHOD ON THE SARD DATASET

Method Name	CWE20	CWE78	CWE129	CWE190	CWE400	CWE787	CWE789
Russell et al.	38.18	87.01	79.88	85.12	77.78	53.44	67.43
VuldeePecker	42.86	**95.96**	75.61	82.53	83.72	55.56	68.93
SySeVR	49.36	92.94	83.33	90.16	88.64	66.75	75.37
Devign	48.18	91.83	83.10	94.19	92.59	65.82	75.55
Reveal	50.00	93.02	84.77	94.89	**93.35**	66.62	76.60
SDBV	**54.55**	94.31	**90.63**	**97.00**	93.33	**69.23**	**83.92**

TABLE V. COMPARISON RESULTS OF THE ACCURACY R(%) OF EACH METHOD ON THE SARD DATASET

Method Name	CWE20	CWE78	CWE129	CWE190	CWE400	CWE787	CWE789
Russell et al.	49.65	79.76	70.54	57.83	83.14	51.02	56.41
VuldeePecker	53.19	84.76	67.18	57.76	85.51	54.66	60.44
SySeVR	63.83	94.04	83.98	79.42	92.64	95.12	69.30
Devign	62.65	93.67	84.08	86.64	95.04	94.75	69.70
Reveal	65.01	95.23	95.20	93.86	**99.98**	94.90	72.52
SDBV	**70.92**	**98.80**	**97.41**	**95.27**	99.76	**98.40**	**80.74**

TABLE VI. COMPARISON RESULTS OF THE ACCURACY F1(%) OF EACH METHOD ON THE SARD DATASET

Method Name	CWE20	CWE78	CWE129	CWE190	CWE400	CWE787	CWE789
Russell et al.	43.17	83.23	74.92	68.87	80.37	52.20	61.43
VuldeePecker	47.47	90.01	71.15	67.96	84.61	55.11	64.40
SySeVR	55.67	93.49	83.66	84.45	90.59	78.45	72.21
Devign	54.47	92.74	83.59	90.26	93.79	77.68	72.51
Reveal	56.53	94.12	89.68	94.37	**96.55**	78.28	74.50
SDBV	**61.67**	**96.51**	**93.89**	**96.13**	96.44	**81.27**	**82.30**

For example, the CWE787 out-of-bounds write vulnerability, assuming that there is an array buffer, to determine the out-of-bounds write needs to obtain the size of the buffer, the size of the data to be written, and the control information of the written statement to determine whether the program execution flow will be a written statement is reached. In the VDRG generation process, the memory write functions are sensitive function calls, the multi-layer control dependencies of the memory write function will be extracted into the VDRG representation, and the data dependencies of the variables involved will also be included in the VDRG representation, these variables may be buffer pointers or the size of written data. The VDRG generated by this key information is input into the vulnerability detection model for inference and learning, which can accurately determine whether there is an out-of-bounds write vulnerability.

The performance of SDBV in CWE400 vulnerability types is slightly lower than that of Reveal. CWE400 refers to resource exhaustion-type vulnerabilities. The inspection and judgment of such vulnerabilities need to obtain various information such as resource quantity, allocation strategy, recovery strategy, etc. Reveal uses CPG to represent source code. CPG integrates AST, CFG, DFG, DDG, and other representation graph of the source code. Although there is some redundant information, it completely contains all the information. To distinguish the types of data dependencies, SDBV modified some data flow information, which makes Reveal slightly better than SDBV on CWE400. In addition, Vuldeepecker only extracts sensitive API features, then obtains its related semantic slices according to the sensitive API, and obtains the highest accuracy rate on CWE78, because command injection vulnerabilities are closely related to sensitive APIs. SDBV acquires multiple features including sensitive APIs. The combination of multiple features may cause misjudgments in the model, resulting in lower accuracy than Vuldeepecker.

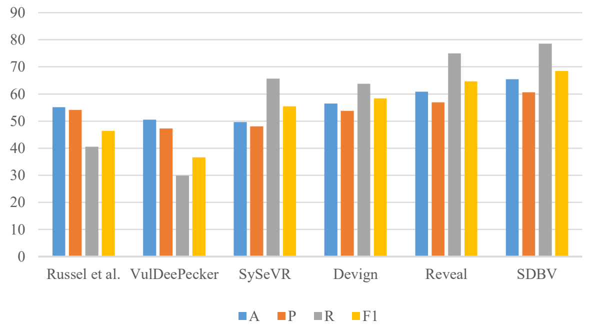

Fig. 8. Comparison results of various methods on the Devign dataset

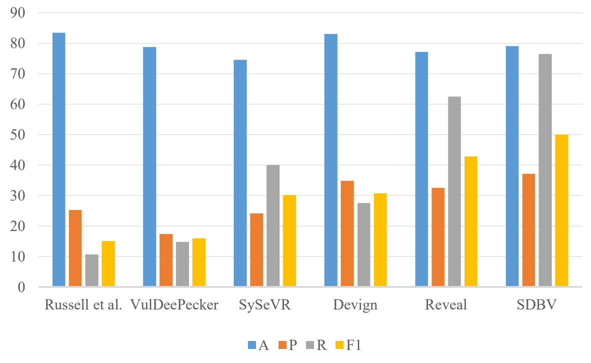

Fig. 9. Comparison results of various methods on the Reveal dataset

On the two real datasets, SDBV still performs well, and the key indicators are better than other methods. The recall rate of vulnerable samples is up to 22.27%, which shows that SDBV can find more vulnerable samples, and the F1 score is up to 16.69%. It is proved that the comprehensive performance of the SDBV model is more stable.

SDBV extracts the potential vulnerability features of the source code function slice and generates a corresponding VDRG for each potential vulnerability feature. One of the VDRGs is confirmed to have vulnerabilities by the vulnerability detection model, and this function is vulnerable. SDBV extracts 4 vulnerability characteristics, covering most of the CWEs, so the key indicators are better than other methods.

Although Reveal uses CPG to represent the source code, it does not further analyze the data flow and control flow closely related to the vulnerability, and there is a lot of redundant information, so the performance is lower than SDBV.

Devign uses AST, CFG, DFG, and natural code sequences to represent source code, which can reflect source code characteristics to a certain extent, but also does not consider data and control dependencies directly related to vulnerability information. The remaining 3 methods process the source code as a sequence of tokens.

SySeVR uses 4 vulnerability syntax features to slice the source code, and obtains the original information of the source code to the greatest extent, so it performs the best among the 3 methods based on the sequence of tokens, but is limited by the problem that the token sequence itself cannot model control and data-dependent information over a long distance, the performance is lower than that of graph-based methods.

Compared with SySeVR, Vuldeepecker only uses sensitive API for function slicing, ignoring a large number of arrays, pointers, and expressions related to vulnerability information. Compared with Vuldeepecker's use of BLSTM to deal with the long-distance context dependency problem of sentences, the method proposed by Russell uses RNN to build a vulnerability detection model. When using RNN to process long sequences, there may be a problem of gradient disappearance, which affects the vulnerability detection performance.

CONCLUSION

In view of the fact that the existing source code vulnerability detection methods do not explicitly maintain the semantic information related to vulnerabilities in the source code, which makes it difficult for the vulnerability detection model to extract the characteristics of vulnerability statements and has a high detection false positive rate, this paper proposes a source code vulnerability detection method based on vulnerability dependency graph, which uses a novel source code representation structure VDG, and analyzes the control dependency chain and data dependency chain of candidate vulnerability statements, then explicitly maintain the control dependency and data dependency of candidate vulnerability statements in VDG, so that the model can more easily capture the semantic information of candidate vulnerability statements. The VDNN for VDG can effectively infer and perceive the semantic information contained in VDG, and further extract features according to the previous learning information to complete the final vulnerability detection. The experimental results on 3 datasets show that the source code vulnerability detection performance of this method is better than other existing methods.

In future research, we will try to design a more efficient code representation structure and vulnerability detection neural network to further improve the accuracy and reduce the false positive rate, and improve the performance of the source code vulnerability detection model.

ACKNOWLEDGMENT

The work was supported by the National Natural Science Foundation of China (No.U1833107). Hongyu Yang and Xiang Cheng are corresponding authors.

REFERENCES

[1] G. Lin, S. Wen, Q. Han, J. Zhang, and Y. Xiang, "Software vulnerabil ity detection using deep neural networks: a survey," Proc. IEEE, vol. 108, no. 10, pp. 1825-1848, May 2020.

[2] Y. Miao, C. Chen, L. Pan, Q. Han, J. Zhang, and Y. Xiang, "Machine learning-based cyber attacks targeting on controlled information: a sur vey," ACM Comput. Surv., vol. 54, no. 7, pp. 1-36, November 2021.

[3] Z. Li, D. Zou, S. Xu, H. Jin, and Y. Zhu, "SySeVR: a framework for u sing deep learning to detect software vulnerabilities," IEEE Trans. De pendable Secur. Comput., vol. 19, no. 4, pp. 2244-2258, January 2021.

[4] J. Zhang, L. Pan, Q. Han, and C. Chen, "Deep learning based attack d etection for cyber-physical system cybersecurity: a survey," IEEE-CA A J. Automatica Sin., vol. 9, no. 3, pp. 377-391, September 2021.

[5] J. Qiu, J. Zhang, W. Luo, L. Pan, S. Nepal, and Y. Xiang, "A survey o f android malware detection with deep neural models," ACM Comput. Surv., vol. 53, no. 6, pp. 1-36, December 2020.

[6] H. Wang, G. Ye, Z. Tang, S. Tan, S. Huang, D. Fang, Y. Feng, L. Bia n, and Z. Wang, "Combining graph-based learning with automated dat a collection for code vulnerability detection," IEEE Trans. Inf. Forens ic Secur., vol. 16, no. 1, pp. 1943 – 1958, December 2020.

[7] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning base d vulnerability detection: are we there yet," IEEE Trans. Softw. Eng., vol. 9, no. 1, pp. 1-17, June 2021.

[8] F. Yamaguchi, N. Golde, and K. Rieck, "Modeling and Discovering V ulnerabilities with Code Property Graphs," in 2014 IEEE Symposium on Security and Privacy. Piscataway: IEEE Press, 2014, pp. 590–604.

[9] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, "Automated Vulnerability Detection in Source Code Using Deep Representation Learning," in 17th IEEE I nternational Conference on Machine Learning and Applications (ICM LA). Piscataway: IEEE Press, 2018, pp. 757–762.

[10] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "VulDeePecker: A Deep Learning-Based System for Vulnerability De tection," in the 2018 Network and Distributed System Security Sympo sium. Reston, VA: Internet Society, 2018, pp. 1–15.

[11] M Allamanis, M Brockschmidt, and M Khademi, "Learning to Repres ent Programs with Graphs," in 2018 International Conference on Lear ning Representations. Ithaca, NY: arXiv.org, 2018, pp. 1–17.

[12] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective Vulne rability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks," in the 33rd Conference on Neural Infor mation Processing Systems. Vancouver: NeurIPS, 2019, pp. 1–11.

[13] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated Graph Seq uence Neural Networks," in 2016 International Conference on Learnin g Representations. Ithaca, NY: arXiv.org, 2016, pp. 1–20.

[14] Z. Hu, Y. Dong, K. Wang, and Y. Sun, "Heterogeneous Graph Transf ormer," in the Web Conference 2020. New York: ACM, 2020, pp. 27 04–2710.

[15] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Yu, "A comprehen sive survey on graph neural networks," IEEE Trans. Neural Netw. Lea rn. Syst., vol. 32, no. 1, pp. 4-24, March 2020