Contents lists available at ScienceDirect

Computers & Security

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

HyVulDect: A hybrid semantic vulnerability mining system based on graph neural network

Wenbo Guo[1], Yong Fang, Cheng Huang[2,*], Haoran Ou, Chun Lin, Yongyan Guo[3]

School of Cyber Science and Engineering, Sichuan University, Chengdu 610065, China

ARTICLE INFO

Article history:
Received 23 December 2021
Revised 8 April 2022
Accepted 29 June 2022
Available online 1 July 2022

Keywords:
Program analysis
Vulnerability mining
Graph neural network
Taint analysis
Code representation,

ABSTRACT

In recent years, software programs tend to be large and complex, software has become the infrastructure of modern society, but software security issues can not be ignored. software vulnerabilities have become one of the main threats to computer security. There are countless cases of exploiting source code vulnerabilities to launch attacks. At the same time, the development of open source software has made source code vulnerability detection more and more critical. Traditional vulnerability mining methods have been unable to meet the security analysis needs of complex software because of the high false-positive rate and false-negative rate. To resolve the existing problems, we propose a graph neural network vulnerability mining system named HyVulDect based on hybrid semantics, which constructs a composite semantic code property graph for code representation based on the causes of vulnerabilities. A gated graph neural network is used to extract deep semantic information. Since most of the vulnerabilities are data flow associated, we use taint analysis to extract the taint propagation chain, use the BiLSTM model to extract the token-level features of the context, and finally use the classifier to classify the fusion features. We introduce a dual-attention mechanism that allows the model to focus on vulnerability-related code, making it more suitable for vulnerability mining tasks. The experimental results show that HyVulDect outperforms existing state-of-the-art methods and can achieve an accuracy rate of 92% on the benchmark dataset. Compared with the rule-based static mining tools Flawfinder, RATS, and Cppcheck, it has better performance and can effectively detect the actual CVE source code vulnerabilities.

1. Introduction

With the rapid development of computer technology, software applications have become indispensable and essential in daily life. When the software brings convenience to people, they also bring many safety problems. Criminals exploiting software vulnerabilities can cause significant threats, steal personal private information, and even launch attacks that threaten national security.

Software vulnerabilities often originate from potentially insecure code. The analysis and mining of source code features have always been an essential method of vulnerability detection. However, many vulnerabilities are inadvertently generated, and it is not easy to prevent them effectively. At the same time, with the development of open source software. Open source code is penetrating all aspects of the software field, and it has become increasingly difficult for users to avoid the introduction of open source (Guo et al., 2021; Ponta et al., 2018). The security vulnerabilities in the open source code are easier to obtain and exploit by hackers, posing a huge threat (Zhang et al., 2021). Therefore, code safety is of critical.

The existing software tends to be large-scale and complex, and the amount of code of the project increases sharply. The cost of simply using manual audit code is very high, and it is difficult to find vulnerabilities with complex trigger conditions. The development of machine learning and deep learning has also been widely used in software vulnerability mining. Methods based on traditional machine learning need to manually extract the features of vulnerabilities and rely on a large amount of expert knowledge. Based on the deep learning methods (Lin et al., 2020), the source code is treated as a natural language sequence, and the existing natural language processing methods are used for features representation to summarize the features of the vulnerability for detection and classification. These methods can effectively capture the contextual information triggered by vulnerabilities in the source

* Corresponding author.

E-mail addresses: honywenair@163.com (W. Guo), yongfangscu@gmail.com (Y. Fang), opcodesec@gmail.com (C. Huang), ouhr1111@163.com (H. Ou), linc7799@gmail.com (C. Lin), guoyongyan1998@gmail.com (Y. Guo).

[1] [orcid=0000-0001-6655-8179]

[2] [orcid=0000-0002-5871-946X]

[3] [orcid=0000-0001-6623-7201]

https://doi.org/10.1016/j.cose.2022.102823
0167-4048/© 2022 Elsevier Ltd. All rights reserved.

code. However, the structural characteristics of the source code are not considered, and in addition, much semantic information is lost in the code representation. Although graph neural network can handle non-Euclidean data such as code graph representation, the existing methods represent the source code as AST and CFG, which lack of the data dependency information of the source code and are not conducive to the detection of vulnerabilities. Meanwhile, using the program source code directly as the input of the graph neural network introduces a lot of redundant code, which is not conducive to the learning of the model.

We propose a graph neural network vulnerability mining system named HyVulDect based on hybrid semantics to solve the above problems. The system combines source code graph-level features and taint propagation chain tokens-level features for vulnerability detection. First, the source code is preprocessed and represented as a composite semantic code property graph, and then the gated graph neural network with multi-head attention mechanism is used to extract the deep semantic and structural information. Simultaneously, the source->sink taint propagation chain in the source code is extracted by the taint analysis method and represented as tokens, and the syntax and context information are extracted by BILSTM with self-attention mechanism. Finally, the graph-level and tokens-level features are fused, and the XGboost classifier is used for classification. This method can not only more comprehensively extract the rich semantic information in the source code, but also pay attention to the code context information and has better detection capability.

The specific contributions of this work are the following:

- We propose a graph neural network vulnerability mining system based on hybrid semantics, which utilizes a gated graph neural network and BiLSTM network with a dual-attention mechanism to extract source code graph-level and tokens-level features. Deep features that fuse two dimensions can be used effectively to detect vulnerabilities.
- We have improved the program slicing algorithm based on API calls (Li et al., 2021) by supplementing structure of the program slice, which reserves the structural information of the code while extracting the vulnerability context information.
- Experiments based on the design scheme show that HyVulDect detection performance is better than traditional static scanning tools. Compared with state-of-the-art detectors Devign, VUDDY, and BGNN4VD achieves 27.6%, 14.2%, and 4.9% improvement in precision respectively. Meanwhile it can effectively detect existing CVE vulnerabilities.

The rest of the paper is organized as follows: Section 2 presents the related work. Section 3 introduces the preliminaries. Section 4 details the implementation process of the HyVulDect system. Section 5 presents the experiments and analyses. Section 6 summarizes the conclusion and proposes future works.

2. Related work

2.1. Source code vulnerability mining methods

Vulnerability mining based on code similarity

The same vulnerability has the same trigger pattern, so similar code is likely to contain the same vulnerability. VulPecker (Li et al., 2016) designed a set of definitions to describe the characteristics of software patches and proposed a code similarity calculation algorithm that is effective for all types of vulnerabilities. In the learning phase, by inputting candidate code similarity algorithm, vulnerability diff blocks features vector, accuracy threshold, and VCID database into the algorithm engine, the engine will automatically output the CVE algorithm mapping table. In the detection phase, the vulnerability detection engine uses the appropriate code similarity algorithm to search for the vulnerability signature from the target program signature. If found, it indicates that the target program has a vulnerability. VDSimilar (Sun et al., 2021) performs vulnerability detection based on the code similarity of vulnerabilities and patches. Use the BiLSTM-based siamese network model to learn the difference between vulnerability-vulnerability and vulnerability-patch and introduce an attention mechanism to improve the detection accuracy of the model. SQVDT (Zhang et al., 2021) and Buggraph (Ji et al., 2021) are also detection models based on code similarity. Although this type of model is efficient, its detection capability is limited by the size of the vulnerability database and can only detect vulnerabilities triggered by replication.

Vulnerability mining based on deep learning

Zhen Li et al. proposed a vulnerability mining system named Vuldeepecker based on deep learning Li et al. (2018). A collection of code statements constructed based on heuristic methods: code gadgets, extracting the code associated with the vulnerability in large sections of code and then using Word2vec for vectorization to train BiLSTM models for vulnerability detection. However, the method only considers that the data dependency retains semantically related lines of code as code gadgets to represent the program, ignoring other semantic information, and the structural information of the code context is incomplete due to program slicing. In addition, the solution only supports the detection of vulnerabilities related to buffer overflow and resource management. Xu Duan et al. proposed a fine-grained vulnerability detection method based on an attention mechanism: VulSnipper (Duan et al., 2019). It consists of three modules: embedding module, attention module and classification module. The embedding module converts the features tensor into a matrix composed of the features vector of the node by flattening and mapping. The attention module assigns attention weights to different nodes, which are implemented by bottom-up and top-down structures. The classification module integrates features and finally realizes two classifications. There are many vulnerable code in the open source community. PyVul (Guo et al., 2021) uses deep neural networks to detect Python code snippets, which can efficiently assist developers with secure programming.

Vulnerability mining based on graph neural network

Devign (Zhou et al., 2019), a model based on a general graph neural network, performs graph-level classification and representation by learning a rich set of code semantics. Devign uses the graph embedding layer to encode the source code into a composite semantic joint graph, gathers and transfers the information of the adjacent nodes in the graph through the gated graph recurrent layer to learn the features of the nodes, and finally uses the Conv module to extract important nodes for prediction. Devign can effectively extract semantic information in the source code. Huanting Wang et al. proposed a multi-relational gated graph neural network detection model named Funded (Wang et al., 2020), which integrates the data and control path extracted from PCDG into the AST to build an extended AST and uses GGNN to build a vulnerability detection model to capture program syntax, semantics information, and call relationship. At the same time, Funded combines probabilistic learning and statistical evaluation to develop an ”expert hybrid” method to solve the shortage of vulnerable training code samples and uses transfer learning to transplant vulnerability detection models across programming languages. BGNN4VD (Cao et al., 2021) combines multiple algorithms for vulnerability detection, extracts grammatical and semantic information in the source code through AST, CFG, and DFG, and then vectorizes it as the input of BGNN, introducing edges based on traditional graph neural network (GNN) to learn different features between vulnerable and non-vulnerable code. Finally, using a convolutional neural network (CNN) to further extract features and detect vulnerabilities using a classifier. DeepTective (Rabheru et al., 2020) extracts syntactic and semantic information through gated recurrent units and graph convolutional networks. However, when the code is represented as CFG, only the control dependency is considered, and other dependency information is ignored.

2.2. Source code representation

The source code cannot be directly input into the neural network. For this reason, it is necessary to convert the source code into a vector that can express the features of the vulnerability (Ben-Nun et al., 2018). Due to the need to retain the semantic information of the data (such as data dependency and control dependency), intermediate representation is usually introduced as a transition between source code and vector representation. The intermediate representation of the code is the code representation. A proper representation form can extract features from the code as much as possible, and has a richer ability to express vulnerability features (Li et al., 2019). We take the code in Sample.c as an example to introduce the following common representation methods.

Sample.c

```
int func(x,y)
{
    int MAX = 10;
    if(x < MAX)
    {
        int y = 2 * x;
        foo(y);
    }
    return y;
}
```

1. **Token sequence representation**The program source code is treated as plain text. First, lexical analysis is used to scan the source file, and the character stream is divided into tokens (Fang et al., 2019). In general, the tokens in the programming language are: constants (integers, decimals, characters, strings, etc.), operators (arithmetic operators, comparison operators, logical operators), separators (commas, semicolons, brackets, etc.), reserved words, identifiers (variable names, function names, class names, etc.), etc.
2. **Abstract syntax tree**An AST (Neamtiu et al., 2005) is an ordered tree representation of the source code. Typically, it is used by code parsers to understand the basic structure of a program and check for syntax errors in the first step of the representation. Starting from the root node, the code is divided into blocks, statements, declarations, expressions, etc., and finally forms the token of the leaf node.
3. **Control flow graph**Describes all paths that may be traversed during execution. In CFG, nodes represent statements and conditions. Directed edges connect them to represent the transfer of control.
4. **Data flow graph**Tracking the use of variables throughout the CFG, the data flow is variable-oriented. Any data flow involves accessing or modifying some variables.
5. **Program dependency graph**The program dependency graph is a graphical representation of a program, which is a directed multigraph with markers (Nasirloo and Azimzadeh, 2018). The program dependency graph can represent the control dependency and data dependency of the program. So it is composed of (data dependence graph) DDG and (control dependence graph) CDG.
6. **Code property graph**Code property graph (CPG) (Xiaomeng et al., 2018) combines the characteristics of the abstract syntax tree, control flow graph, and program dependency graph in a joint data structure. CPG can combine the advantages of different representations. AST represents all Tokens and predicates in the source code as a node, which is further subdivided than CFG and PDG. Therefore, with AST as the main body, control and data dependence are integrated to form a code property graph. In the paper, we have modified the CPG to merge and represent it as a new edge when both control and data flows exist between two nodes, as shown in *CAD* in Fig. 5.

3. Preliminaries

3.1. Graph neural network

With the development of machine learning and deep learning, speech, image, and natural language processing have gradually made great breakthroughs (Lopez and Kalita, 2017; Minaee et al., 2021; Wang and Chen, 2018). However, speech, image, and text are all simple sequence or grid data, which are very structured. Deep learning is very good at dealing with that type of data. Although traditional deep learning methods have been successfully applied to extract features from Euclidean spatial data, many practical application scenarios generate data from non-Euclidean spaces. The real world is often abstracted into graph structures, such as social networks, the World Wide Web, etc. In contrast, the performance of traditional deep learning methods in dealing with non-Euclidean spatial data such as graph structure is still unsatisfactory. Graph Neural Network (GNN) (Scarselli et al., 2008), a powerful learning method for deep representation of graph data, has shown excellent performance in graph data analysis. It can use deep neural networks to learn node representations based on node features and graph structures.

GNN is a deep neural network that appears to process graph data. The graph structure $G = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ denotes the set of nodes, $\mathcal{E}$ denotes the set consisting of all edges, the features vector of nodes is denoted as $h_v \in \mathbb{R}^D$, In node classification, each node v can be represented by its node features vector x_v, and is associated with the label t_v. Given a labeled graph G, the goal is to use these labeled nodes to predict the unknown node label, which is obtained through learning the d dimensional vector of each node is represented as h_v, which contains the information of its neighboring nodes.

GNN maps graph data to lower dimensions for output in two steps. The first step is the propagation process, which is the process of updating node representations over time, and the second step is the output process, which is the process of obtaining the target output (such as the category of each node) based on the final node representations.

The GNN propagation process refers to the process of acquiring node representations by iteration, first initializing the node representation $h_v^{(1)}$, and then the representation of each node is updated using a recurrence, where t denotes the timestep.

$$h_v^t = f(X_v, X_{CO_v}, X_{NBR_{(v)}}, h_{NBR(v)}^{(t-1)}) \tag{1}$$

X_{CO_v} denotes the features of the edge connecting vertex v, $h_{NBR(v)}$ denotes the embedding representation of the neighboring nodes of vertex v, and $X_{NBR_{(v)}}$ denotes the features of the neighboring nodes of vertex v. f is the transfer function that projects the input into the d dimensional space, and since a unique solution for h_v is required, we apply Banach's immobility theory to rewrite the above equations for iterative updating.

$$H^{(t+1)} = F(H^t, X) \tag{2}$$

H and X denote all connections of h and x, and the output of GNN is calculated by passing the state h_v and the features x_v to the output function g.

$$o_v = g(h_v, x_v) \tag{3}$$

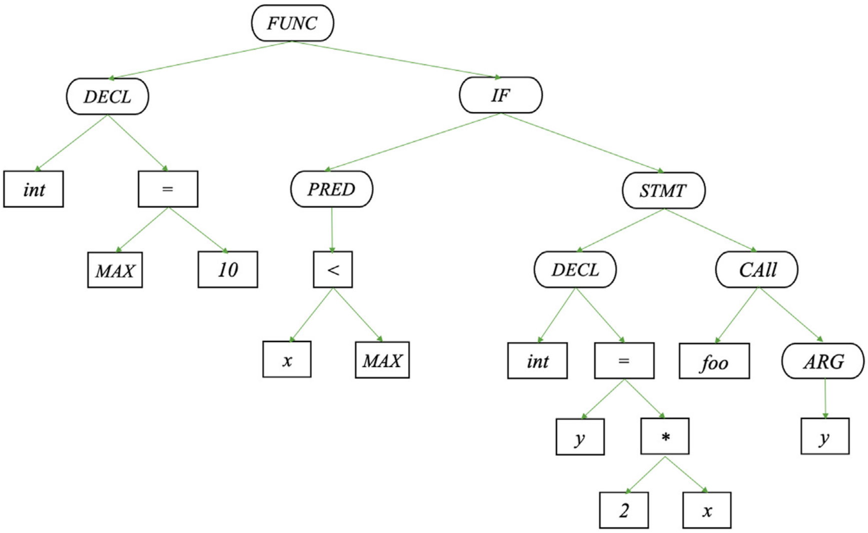


Fig. 1. Abstract syntax tree(AST).

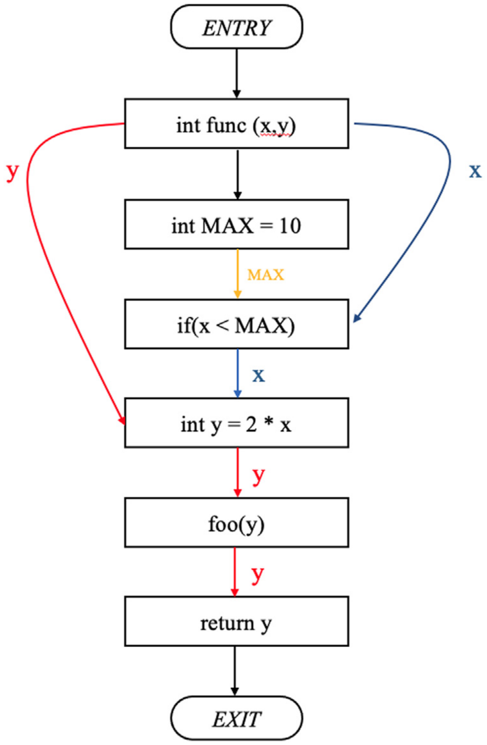


Fig. 2. Data flow graph(DFG).

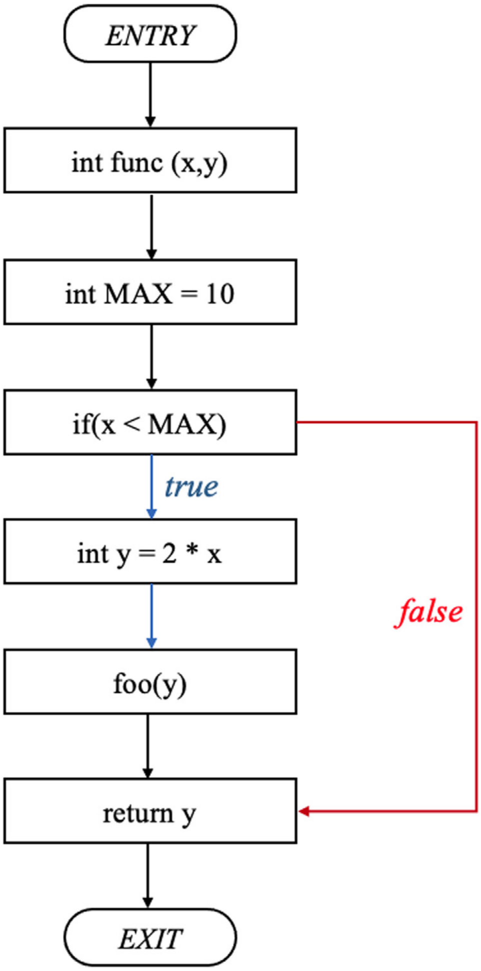


Fig. 3. Control flow graph(CFG).

In GNN, the final representation of a node has no relationship with the initial state of the node because the compressed mapping will ensure that the node representation reaches the immobility point at the final moment, while gated graph neural networks are not based on Fixed Point theory. This means: f no longer needs to be a compressed map; the iteration does not need to converge to output, and it can be iterated with a fixed step; the optimization algorithm has also changed from AP algorithm to BPTT.

Gated graph neural networks (Li et al., 2015), compared to graph neural networks, introduce ”gates”, such as GRU and LSTM, which aim to improve the long-term dependence of graph structure information while incorporating edge information. Yujia Li & Richard Zeme proposed a gated graph neural network, which uses a gated recurrent unit (GRU) (Dey and Salem, 2017) in the propagation step, unrolls the loop in a fixed number of steps T, and uses backpropagation over time to calculate the gradient. The gate structure contains the sigmoid function, which takes a value be-

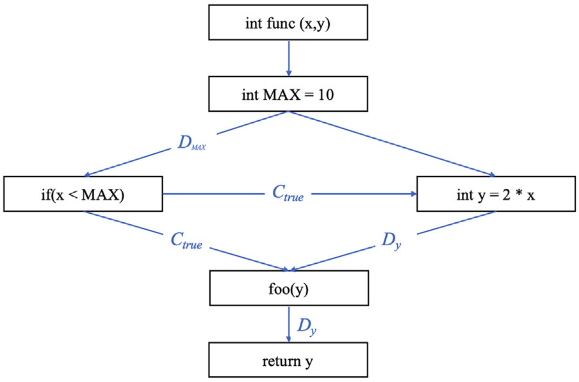


Fig. 4. Program dependency graph(PDG).

tween 0 and 1, as shown in formula. 4.

$$\sigma = \frac{1}{1+e^t} \tag{4}$$

$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right) \tag{5}$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right) \tag{6}$$

$$\widetilde{h}_t = tanh(W \cdot [r_t * h_{t-1}, x_t]) \tag{7}$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \widetilde{h}_t \tag{8}$$

Consistent with the paradigm defined by the graph neural network, GGNN also has two processes: state update and output. Compared with GNN, its main difference comes from the status update stage. Specifically, GGNN refers to the design of GRU and regards the information of neighbor nodes as input, and the state of the node itself as hidden state, and its state update function is as follows:

$$H_v^{(t+1)} = GRU\left(h_v^t, \sum_{u \in NBR(v)} W\_edge\ h_u^t\right) \tag{9}$$

On the basis of GRU, GGNN introduces learnable parameters W for different types of edges, and each type of edge corresponds to a W_edge, so that GGNN can handle heterogeneous graphs.

3.2. Taint analysis

Taint analysis is a technique that tracks and analyzes the flow of tainted information through a program (Zheng et al., 2019). A taint is a contaminated message. In program analysis, information from outside and entering the program is treated as tainted information. Depending on the need for analysis, the data used within the program can also be used as taint information, and the flow of information corresponding to it can be analyzed. According to whether the program is running during taint analysis, it can be divided into static and dynamic taint analysis.

Taint analysis mainly includes the source of the taint, taint convergence point and the sanitizer. The taint source means that the tainted data is introduced into the system; the taint aggregation point means that the system outputs the tainted data to the sensitive data area or the outside world, causing the sensitive data area to be illegally rewritten or the privacy data is leaked; sanitizer means that the data transmission is no longer harmful to the integrity and confidentiality of the system through operations such as data encryption or reassignment.

The process of taint analysis is to identify the generation point of taint information in the program and mark the taint information; to trace and analyze the propagation process of taint information in the program using specific rules; and to check whether the critical operation will be affected by the taint information at some critical program points. The generation point of taint information is called the source point, and the checkpoint of taint information is called the sink point.

In vulnerability analysis, taint analysis technology is used to mark the data of interest (usually from the external input of the program, assuming that all inputs are dangerous) as tainted data. Then by tracking the flow of information related to the tainted data, it is possible to know if they will affect certain critical program operations and explore program vulnerabilities. The question of whether there is a vulnerability in the program is transformed into whether the operation will use the tainted information on the sink point.

4. Methodology

In order to better mine various vulnerabilities in the source code, we propose a graph neural network vulnerability detection system based on hybrid semantics: HyVulDect. Fig. 6 shows the detailed design of the system, which is mainly divided into three parts. Source code graph-level features extraction: represent the source code as a joint graph CPG containing multiple semantic in-

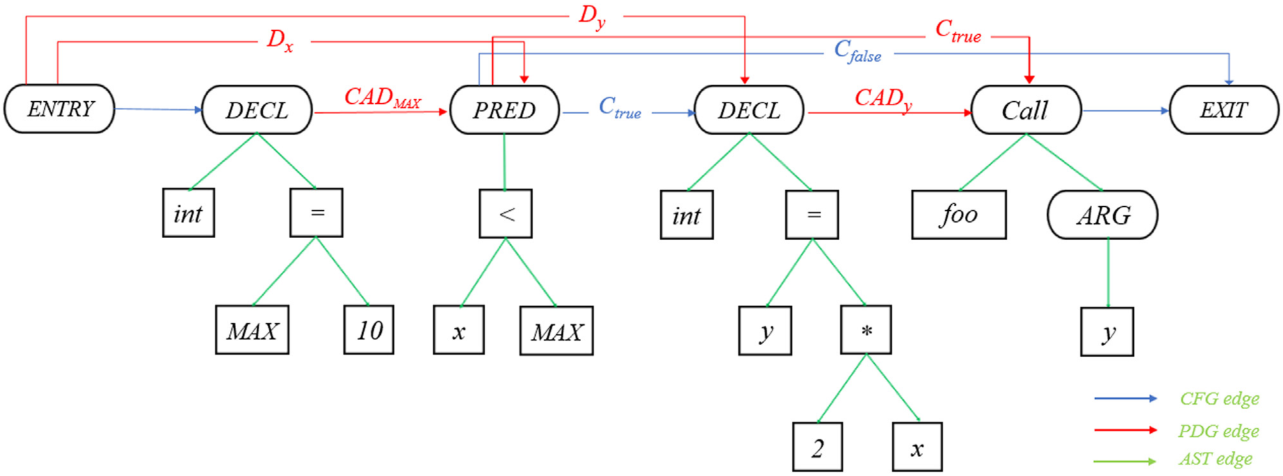


Fig. 5. Code property graph(*CAD* means that it contains both control dependency and data dependency).

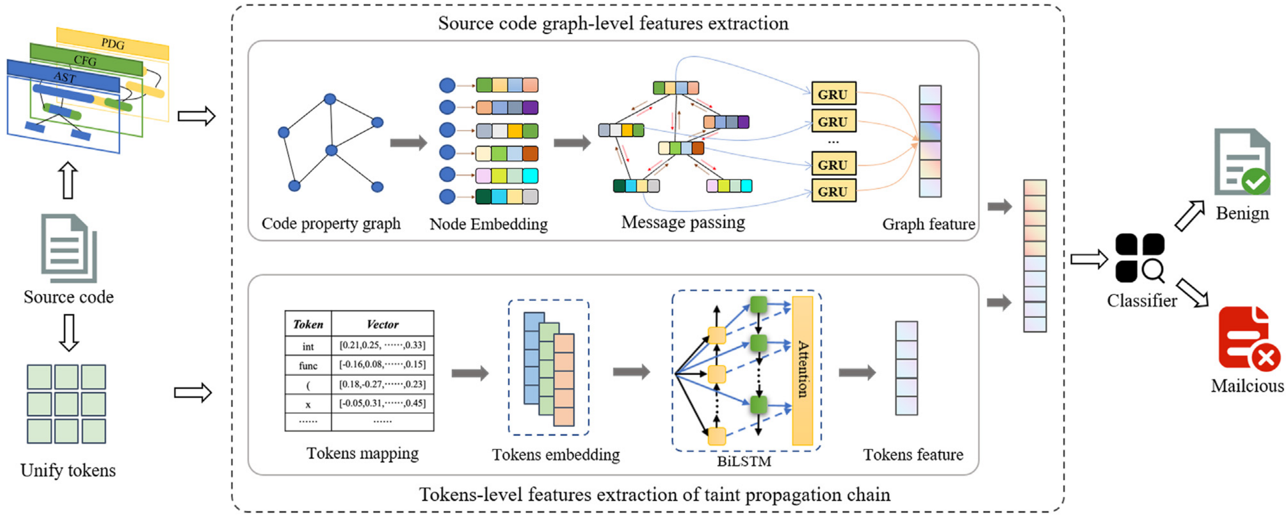


Fig. 6. The overall architecture of proposed HyVulDect.

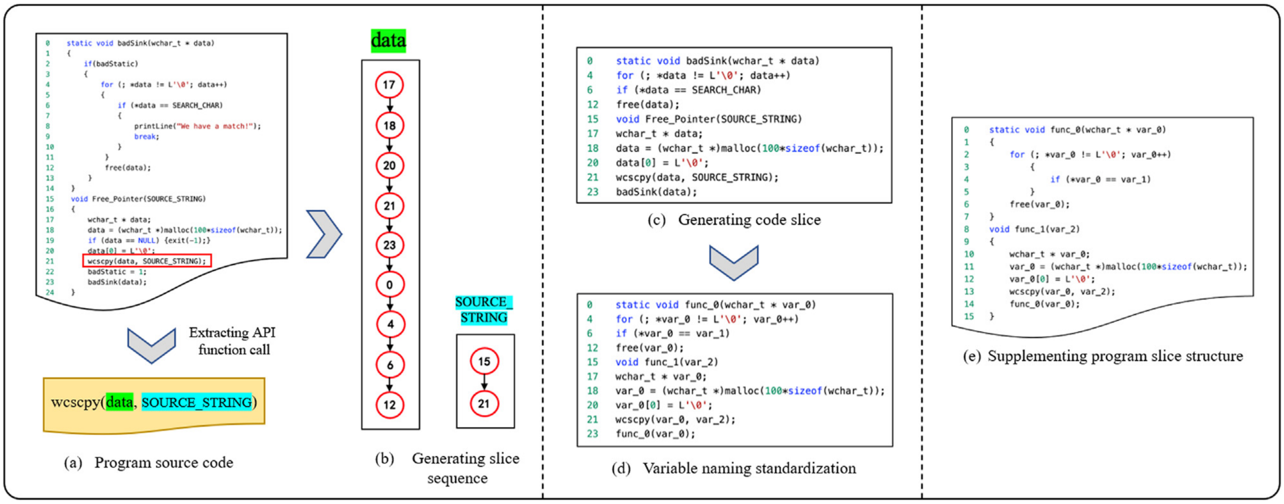
formation, and then use a gated graph neural network to extract source code graph-level features. Tokens-level features extraction of taint propagation chain: extract the taint propagation chain in the source code through taint analysis, and then use the BiLSTM algorithm with self-attention mechanism to extract tokens-level features. Classification module: fuse the source code features extracted from the previous two parts, and use the pre-trained XG-boost classification model for vulnerability classification.

4.1. Source code graph-level features extraction

The source code contains much semantic information, which cannot be fully extracted by simply using a graph structure. AST can represent syntax information, CFG, DDG, CDG can represent the semantic information of the source code, including control flow information, data dependency information, control dependency information. CPG is a graph structure that integrates various information of AST, CFG, DDG, and CDG. Large projects contain thousands of lines of code, but in most cases, the lines of code that cause vulnerabilities are always concentrated in a few lines. So we first preprocessed the code to reduce the burden during training.

For most of the vulnerabilities, including but not limited to buffer overflow, injection, etc. The root cause of the vulnerability is that the external input has not been safely handled and entered the dangerous function. To this end, we first determine the dangerous function, backtrack all the associated lines of code according to the parameter passing of the dangerous function, and then reorganize these key lines of code in the original order. At the same time, because different developers have different writing habits, there are huge differences in the naming of variables and functions. In order to eliminate the error caused by different naming and reduce the dimensionality of vectorization, we have made a uniform replacement. User-defined function name, use $func\_N(N = 0, 1, \ldots, n)$ to indicate, user-defined variable name, use $var\_N(N = 0, 1, \ldots, n)$ to indicate. Although the code extracted in this way is semantically related, the original structure is destroyed and the context dependency is disrupted. To this end, we have supplemented the structure of the program slice.

We refer to the method proposed in the VulDeepecker Li et al. (2018) for the program slicing, which is efficient and valuable. However, this method will cause the code structure to be destroyed, and the code property graph cannot be generated effectively. For this reason, we add the program slice structure supplementation to this method. Fig. 7 shows an example of program slicing. Fig. 7(a) is the source code of the program, which calls the *wcscpy* function to copy a wide string from the source to the target. First, we locate the API function *wcscpy* function, which has two parameters *data* and *SOURCE_STRING*. For the parameter *data*, which belongs to the user-defined function *Free_Pointer*(), the internal slice of the function consists of five statements, namely lines 17, 18, 20, 21, 23 of the program. The *badSink*() function on line 23 is a user-defined function that receives the external parameter *data*, the internal slice of the function consists of four lines of statements, namely lines 0, 4, 6, and 12 of the program. For the parameter *SOURCE_STRING*, the associated statements are lines 15, 21. The final slice sequence obtained by the parameter *data* is: 17->18->20->21->23->0->4->6->12. The final slice sequence obtained by the parameter *SOURCE_STRING* is: 15->21. Combine the two slice sequences according to the original code order, and remove the repeated lines of code, and the final slicing code sequence based on the API function *wcscpy*() is: 0->4->6->12->15->17->18->20->21->23. Fig. 7(c) shows the code slices generated based on the API function, which can be seen that effectively reduces the number of lines of code. To reduce the scale of the corpus, we replace user-defined functions and variables one-to-one (e.g. *badSink*() -> *func*_0(), *data* -> *var*_0). Fig. 7(d) shows the program slice after variable naming standardization, which effectively reduces the number of tokens. Although the previous operation effectively extracted the code lines related to the vulnerability and removed the irrelevant code, the structure of the source code is no longer complete. In order to supplement the structure information of the source code, we have supplemented the structure of the program slice. We supplement the structure of the code slice according to the structure of the program source code. Algorithm 1 describes the supplementary process of the program slice structure in detail. In parsing the source code, extract user-defined functions and control blocks in the code to supplement the sliced code. After completing the slicing, first, scan the code slice from top to bottom. If the line of code is a user-defined function, add the structure code "{" under the line, and continue scanning down until you encounter a line of code that does not belong to the user-defined function, and add "}" before it, which completes a supplementation of function structure. For control structure, there are four main structures: if conditional statements, switch conditional statements, for loop statements, and while loop statements. The supplementation logic is the same as function. In the sliced structure supplementing process, we follow this principle: first, complete the structure of the function; then, the control struc-



```
0    static void badSink(wchar_t * data)
1    {
2        if(badStatic)
3        {
4            for (; *data != L'\0'; data++)
5            {
6                if (*data == SEARCH_CHAR)
7                {
8                    printLine("We have a match!");
9                    break;
10               }
11           }
12           free(data);
13       }
14   }
15   void Free_Pointer(SOURCE_STRING)
16   {
17       wchar_t * data;
18       data = (wchar_t *)malloc(100*sizeof(wchar_t));
19       if (data == NULL) {exit(-1);}
20       data[0] = L'\0';
21       wcscpy(data, SOURCE_STRING);
22       badStatic = 1;
23       badSink(data);
24   }
```

```
wcscpy(data, SOURCE_STRING)
```

(a) Program source code

(b) Generating slice sequence

```
0    static void badSink(wchar_t * data)
4    for (; *data != L'\0'; data++)
6    if (*data == SEARCH_CHAR)
12   free(data);
15   void Free_Pointer(SOURCE_STRING)
17   wchar_t * data;
18   data = (wchar_t *)malloc(100*sizeof(wchar_t));
20   data[0] = L'\0';
21   wcscpy(data, SOURCE_STRING);
23   badSink(data);
```

(c) Generating code slice

```
0    static void func_0(wchar_t * var_0)
4    for (; *var_0 != L'\0'; var_0++)
6    if (*var_0 == var_1)
12   free(var_0);
15   void func_1(var_2)
17   wchar_t * var_0;
18   var_0 = (wchar_t *)malloc(100*sizeof(wchar_t));
20   var_0[0] = L'\0';
21   wcscpy(var_0, var_2);
23   func_0(var_0);
```

(d) Variable naming standardization

```
0    static void func_0(wchar_t * var_0)
1    {
2        for (; *var_0 != L'\0'; var_0++)
3            {
4                if (*var_0 == var_1)
5            }
6        free(var_0);
7    }
8    void func_1(var_2)
9    {
10       wchar_t * var_0;
11       var_0 = (wchar_t *)malloc(100*sizeof(wchar_t));
12       var_0[0] = L'\0';
13       wcscpy(var_0, var_2);
14       func_0(var_0);
15   }
```

(e) Supplementing program slice structure

Fig. 7. Example of program slicing based on API function calls. (a)(b) extract API functions from the program source code and generate code slice sequences based on the function parameters, (c) shows the generated code slice, (d) normalizes the naming of the generated slice code, and (e) supplements the structure of the slice code based on the source program structure.

```
Algorithm 1: Supplement structure of program slice.
Input: User-defined function set $F = \{f_1, f_2, \ldots, f_n\}$,
    $f_i = \{f_i\_sl, f_i\_el\}$, $f_i\_sl$ represents the start code line of
    the function block, $f_i\_el$ represents the end code line
    of the function block. Control block set
    $C = \{c_1, c_2, \ldots, c_n\}$, $c_i = \{c_i\_sl, c_i\_el\}$, $c_i\_sl$ represents
    the start code line of the control block, $c_i\_el$
    represents the end code line of the control block.
    $L = \{l_1, l_2, \ldots, l_n\}$ represents sliced lines of code.
Output: Program slice with complete structure
for $l_i$ in $L$ do
    if $l_i$ in $F.F_sl$ then
        Add "{" before the line
    else if $l_i$ in $C.C_sl$ then
        Add "{" before the line
    else if $l_i$ in $C.C_el$ then
        Add "}" at the end of the line
    else if $l_i$ in $F.F_el$ then
        Add "}" at the end of the line
    else
        Continue to the next line
```

```
Algorithm 2: Graph data generation.
Input: .dot file
Output: G: DGL graph data
for edge in dot.edges do
    src,dst = edge.src,edge.dst;
    src_index = dot.nodes.index(src);
    dst_index = dot.nodes.index(dst);
    edge_type = preprocess(edge.attribute);
    G.edata['w'] = edge_type;
for node in dot.nodes do
    node_index = dot.nodes.index(node);
    node_attr = node.attribute;
    node_feature = Word2vec(node_attr);
    G.nodes[node_index].data['x'] = node_feature;
return G;
```

ture, because the control structure is usually contained in the function body. as shown in Fig. 7(e) is the final generated program slice.

We use Joern (2019) to parse the source code into graph data. Joern is a C/C++ code analysis tool. The primary function is to generate abstract syntax trees from source code, control flow graphs and program dependencies, code property graphs, and store them in a graph database, neo4j. During preprocessing, we found two nodes in the CPG generated by Joern contain both control flow edge and a data-dependent edge between them, which we merge into a new edge as CAD. After generating the CPG, we use DGL (Wang et al., 2019) to construct the graph data. The construction algorithm is Algorithm 1. For the node and edge features in the graph, we use the pre-trained Word2vec model for vectorization. The Word2Vec (Church, 2017) model learns the relationship between different words from the context and assigns a vector to each word based on this relationship. If the relationship between two words in the text is closer, their word vector distances will also be closer. In actual experiments, it is found that 97.7% of the features lengths of the nodes are less than 20. Therefore, we set the maximum length of the features to 20, use 0 to make up for any length less than 20, and truncate the long ones. There are four types of node edges, CDG: 0; DDG: 1, AST: 2, CFG: 3, CAD: 4. Different types of edges map different semantic information.

In order to better extract complex semantic information from the source code, we construct a detection model based on gated graph neural networks. The gated graph neural network can learn the relevant semantic information of the vulnerable code well. In the node classification task, the classification of unlabeled nodes is realized by learning the labeled nodes. In the source code graph-level features extraction process, the features are updated iteratively by aggregating the attributes of neighboring nodes to finally output a new node feature representation. At this time, the features have been integrated with the upstream and downstream related information of the node. Graph-level features are calculated as follows:

$$G_f = tanh(\sum_{v \in V} \sigma(i(h_v^{(T)}, x_v))) \tag{10}$$

$\sigma(\cdot)$ is short for $\sigma(i(h_v^{(T)}, x_v))$ and can be generalized by the following equation:

$$\sigma(\cdot) = Sigmoid(Norm(Gconv(G, N_f, E_f))) \tag{11}$$

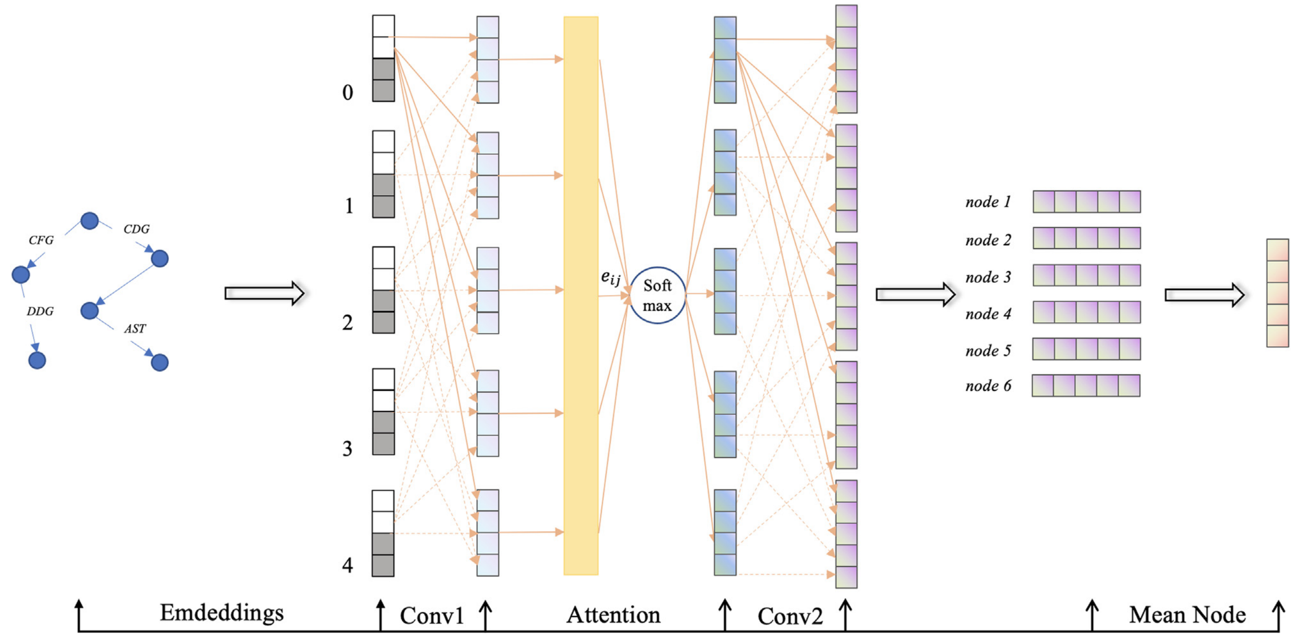


Fig. 8. Graph-level features extraction module.

where $Gconv(\cdot)$ denotes the graph convolution operation, G denotes the generated code attribute graph, N_f is the initial representation of nodes in CPG, and E_f is the type of edges in the graph.

We introduce the attention mechanism Veličković et al. (2017) to make the model more focused on the nodes related to vulnerabilities. The attention mechanism is used to perform a weighted summation of neighboring nodes and calculate the hidden state of each node, while the multi-head mechanism is used to divide the model into multiple heads to form multiple subspaces so that different heads can focus on different aspects of information, which helps the model capture richer feature information.

$$\vec{h}_i' = \Big\Vert_{k=1}^{K} \sigma\Big(\sum_{j\in N_i} \alpha_{ij}^k W^k \vec{h}_j\Big) \tag{12}$$

Where W is the weight matrix, α is the attention cross-correlation coefficient, σ is the nonlinear activation function, and K is the number of heads.

Fig. 8 shows the extraction process of graph-level features in detail. First, use the gated graph convolutional layer to extract some low-level features, and then introduce the Normalization layer, which can make the larger learning rate more stable for gradient propagation and increase the generalization ability of the network. The sigmod function is used as the activation function, and then use the graph attention layer to extract the higher-level features, use the ReLu function to activate the neurons, and the Dropout layer with parameters of 0.3. Disconnect 30% of the neurons, which can reduce the size of the model, while improving the convergence speed of the model and preventing overfitting. Finally, another layer of graph convolution layers is overlaid to extract the features of the highest layer. Multiple convolutional layers enable the iterative extraction of more complex features from lower-level features.

4.2. Tokens-level features extraction of taint propagation chain

The triggering of vulnerabilities has a lot to do with the context. Although the composite graph structure can synthesize various semantic information, it cannot detect vulnerabilities with more complicated trigger conditions. Taint analysis can track and analyze the flow of taint information in the program. We use the taint analysis method to extract the source->sink propagation chain in the source code. In order to simplify the tracking process and streamline the code size, the flow of tainted information is analyzed by tracking the variables, and the queue is used to record the polluted variables. Then extract the taint propagation chain of the source code in this way, use the same method for preprocessing, use the uniform naming scheme to rename, and use the Word2vec pre-trained model to vectorize. We found that 91.4% of the source code taint propagation chain tokens number is not more than 800. Therefore, we choose 800 as the threshold for all vectors. Considering that some special token symbols may contain rich semantic information in the source code analysis, we use lexical analysis techniques to segment the token to maximize the separation of different identifiers and other symbols.

To better capture the semantic information of the taint propagation chain context, we use the BiLSTM model for Tokens-level features extraction. BiLSTM(Bi-directional Long Short-Term Memory) (Chen et al., 2017) is a combination of forward LSTM and backward LSTM (Sundermeyer et al., 2012). BiLSTM can learn the features of serialization and long-term dependency and capture the implicit dependency between sequences. Our model represents the Tokens sequences as matrices in the Embedding layer. The BiLSTM layer receives the matrices as input and learns the deep connections between Tokens and outputs intermediate representations of Tokens. Adding an Attention layer can make the model pay more attention to the vulnerability-related Tokens. The Attention layer receives the hidden representation of the code as input and then initializes the three matrices W_q, W_k, and W_v to transform the output of the upper layer into Q, K, V to get the deep representation of the code. Then adding the Dropout layer to randomly disconnect some neurons to reduce model complexity, enhance the generalization ability of the model and prevent overfitting, and a Dense layer to match the previous layer of neural network. A Normalization layer is added to solve the gradient disappearance and explosion problem. Next, use the ReLu function to activate the neuron and pass through the Dropout and Dense layers again. Finally, the Token-level deep features are obtained.

$$Output = Softmax(QK^T/\sqrt{d_k})V) \tag{13}$$

Table 1
CWE-ID and related description.

CWE ID	Description	Number of samples
CWE-404	Improper Resource Shutdown or Release	572
CWE-476	NULL Pointer Dereference	370
CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	5149
CWE-706	Use of Incorrectly-Resolved Name or Reference	1639
CWE-400	Uncontrolled Resource Consumption	298
CWE-74	Improper Neutralization of Special Elements in Output Used by a Downstream Component ('Injection')	1138
CWE-704	Incorrect Type Conversion or Cast	1632

4.3. Vulnerability classification

After extracting the source code graph-level features and the tokens-level features of taint propagation chain, the final source code features are obtained by concatenating them. CPG-based graph-level features contain source code data dependence, control dependence, and other semantic information. Tokens-level features based on taint analysis contain source code vulnerability contextual semantic information and code sequence syntactic information. The final obtained features incorporate structural, syntactic, and semantic information of the code and can cover multiple types of vulnerabilities. To further complete the classification of vulnerabilities, we use the XGBoost algorithm. The overall objective function can be written as:

$$\mathcal{L}(\phi) = \sum_i l(y_i, \hat{y}_i) + \sum_k \Omega(f_k) \tag{14}$$

where $\mathcal{L}(\phi)$ is the expression on the linear space, i is the i-th sample, k is the k-th tree, and y_i is the predicted value of the i-th sample x_i.

The basic idea of the XGBoost algorithm is similar to that of the GBDT Ke et al. (2019). Where a tree is continuously grown by features splitting, each round of learning a tree fits the residual between the predicted value and the actual value of the previous model round. When the training is completed to get k trees, to predict the score of a sample, according to the features of this sample, in each tree will fall to a corresponding leaf node, each leaf node corresponds to a score, and finally add up the scores corresponding to each tree is the predicted value of this sample.

5. Evaluation

5.1. Dataset

The dataset for this paper consists of two parts. One part comes from the National Institute of Standards and Technology (NIST) NIST (1901) Software Assurance Reference Data Set (SARD) SARD (2017), which contains the source code of various vulnerabilities according to the CWE number. In order to further expand the dataset and improve the detection capability of the model in the real world, we collected the CVE vulnerability and patch source code of FFmpeg, Qemu, PHP, OpenSSL, Libav, Linux open source projects. We built an automated crawler tool. First, we collected all known CVE-related information from the National Vulnerability Database (NVD), including the public time, vulnerability description, CWE ID, and hyperlinks of the patch. Then the corresponding Github patch commit is extracted from the link. Finally, the CVE vulnerability file, patch file, and diff file are collected by the CommitId. A CVE can contain multiple diff files.

After collecting the original vulnerability dataset, we first preprocess and delete all annotations, which have no significance for constructing graph data. According to the diff file mark, the related functions in the patch file and the vulnerability file are extracted and marked. We mark the patch function as "0" and the vulnerability function as "1", and classify them according to the CWE number. As shown in Table 1, it is the CWE type dataset and the quantity we collected. The dataset contains 28,561 C/C++ code fragments, of which 18,276 are positive samples, and 10,285 are negative samples.

Table 2
Confusion matrix definition.

Actual Labels/Predicted Labels	vulnerabilities	non-vulnerabilities
vulnerabilities	TP	FN
non-vulnerabilities	FP	TN

5.2. Evaluation methods

Evaluation indicators are of great significance to machine learning evaluation. During the evaluation process, we defined the confusion matrix as shown in Table 2.

In order to better accurately evaluate the detection ability of the model, we use the following evaluation indicators:

$$Precision = TP/(TP + FP) \tag{15}$$

Precision indicates the proportion of negative samples in all samples detected by the model. The higher the precision, the better the model.

$$Recall = TP/(TP + FN) \tag{16}$$

Recall rate means that the vulnerability samples with correct classification account for the proportion of all actual vulnerability samples. The higher the recall, the higher the detection ability of the model.

$$FPR = FP/(FP + TN) \tag{17}$$

The accuracy rate represents the proportion of samples that the model detects correctly to all samples.

$$F1 = 2 * Precision * Recall/(Precision + Recall) \tag{18}$$

F1 represents the harmonic average of Precision and Recall, and its value is close to the minimum of precision and recall. The higher the F1, the better the performance of the model.

5.3. Experimental setups

In the experiment, we used high-performance computing resources. Table 3 shows our experimental environment. All experimental results are based on this environment.

We randomly shuffle the dataset, with 60% of the training set, 20% of the test set, and 20% of the verification set. To prevent overfitting during training, we set Dropout Rate to 0.3, Learning Rate to 0.0001. Through experimental comparison, we finally choose Cross_Entropy as the loss function to accelerate the convergence of the network, the Adamax is the optimizer.

Table 3
Experimental environment description.

No	Hardware or software	Version or size
1	Operating system	Ubuntu 18.04.1
2	Programming language	Python3.6.9
3	Development environment	Pycharm 2020.2.2
4	CPU	Intel E5-2680 v4@2.4GHz
5	GPU	NVIDIA RTX 2080TI
6	RAM	128 GB
7	Disk	8 T

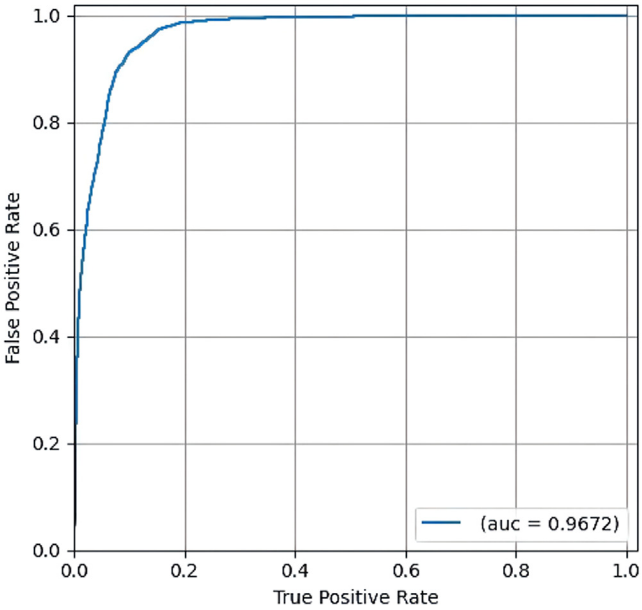


Fig. 9. ROC curve of the HyVulDect.

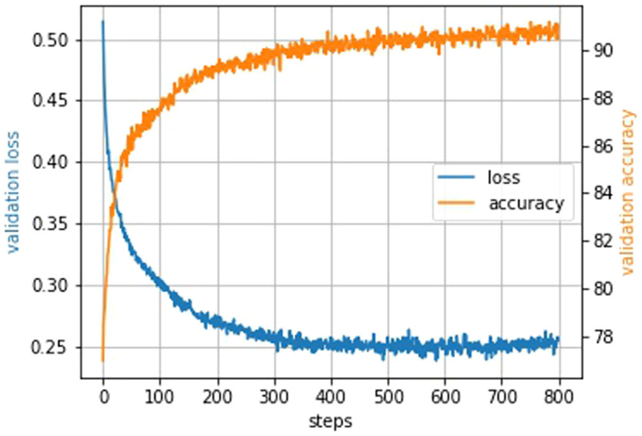


Fig. 10. Loss and accuracy curves during training phase.

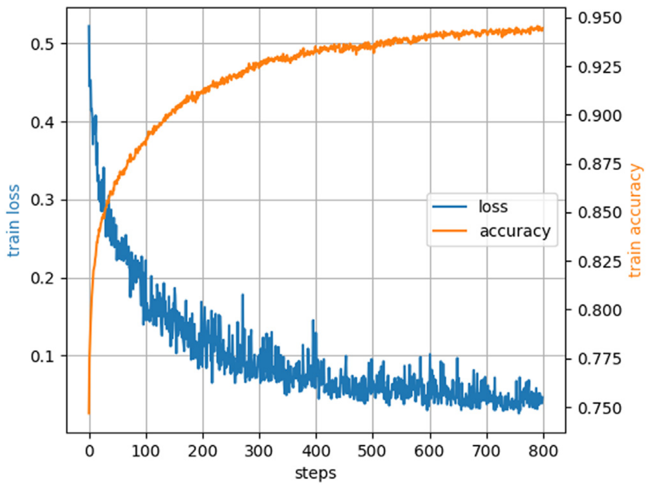


Fig. 11. Loss and accuracy curves during validation phase.

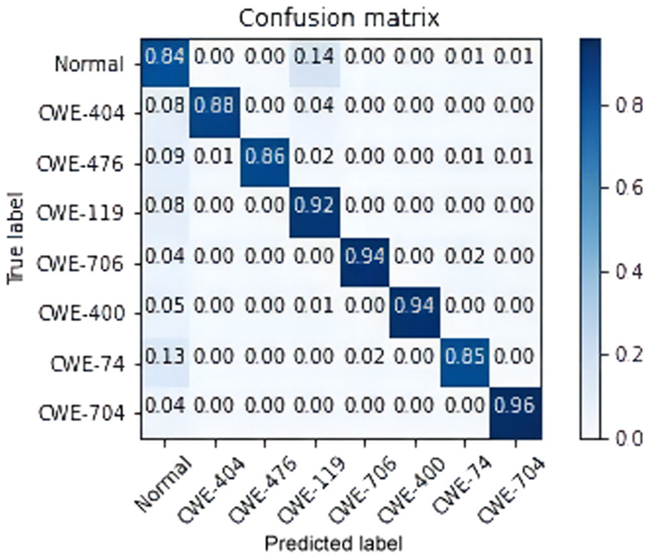


Fig. 12. HyVulDect confusion matrix for different types of classification results.

Table 4
Classification report of HyVulDect(%).

	Precision	Recall	F1	Support
non-vulnerabilities	0.95	0.85	0.90	2452
vulnerabilities	0.89	0.96	0.93	3,252
accuracy	-	-	0.91	5,704
macro avg	0.92	0.91	0.91	5,704
weighted avg	0.92	0.91	0.91	5,704

5.4. Result & analysis

In order to explain the detection ability of the HyVulDect more clearly and intuitively, we have designed five questions and given detailed answers.

Q1: How is HyVulDect system detection capability?

In the experiment, we used the ROC curve to evaluate the performance of the HyVulDect. Fig. 9 shows the ROC curve of the HyVulDect. It can be seen that the area under the curve (AUC) has reached 0.9672, indicating that the model has good detection capability.

Figs. 10 and 11 show the trends of loss and accuracy during the training of the model. During the training process, as the number of training steps increases, accuracy gradually increases and finally stabilizes, and the accuracy reaches 95% when the step is 800. At the same time, the loss gradually decreased from 0.4 to about 0.03.

As shown in Table 4 for the classification report of HyVulDect, it can be seen that the accuracy of the model can reach 92%.

The HyVulDect performs very well on the two-classification task, with an accuracy rate of 92%. In order to further verify the model's ability to detect different types of vulnerabilities, we generated a multi-class confusion matrix as shown in Fig. 12. It can be found that HyVulDect's four types of vulnerability detection capabilities for CWE-704, CWE-400, CWE-706, and CWE-119 can reach more than 93%.

Q2: Is the fusion of source code graph-level features and tokens-level features beneficial to vulnerability detection?

Table 5
Comparison of ablation experiment results(%).

Model	Precision	Recall	F1-Score
HyVulDect-Graph	0.8847	0.8828	0.8832
HyVulDect-Tokens	0.8767	0.8746	0.8744
HyVulDect	0.9154	0.9129	0.9122

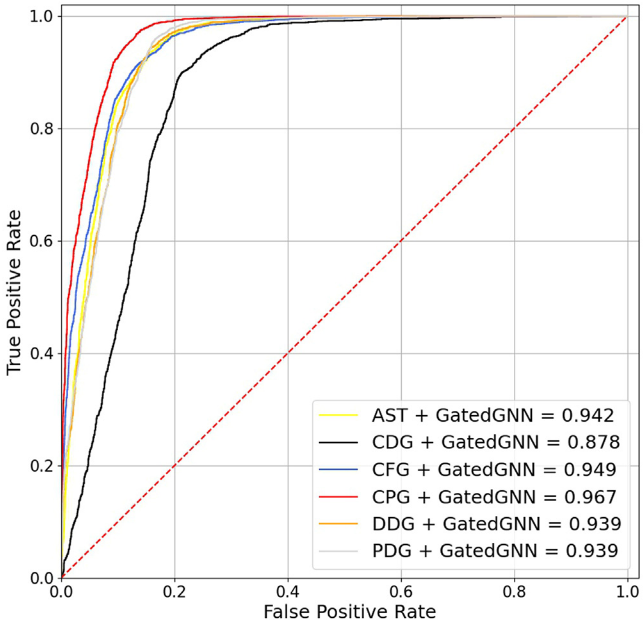


Fig. 13. ROC comparison of different code representations.

HyVulDect is a detection system that fuses source code graph-level features and tokens-level features of taint propagation chain. In order to explore the effectiveness of the two features for vulnerabilities detection, we set up an ablation experiment for comparison to prove the effectiveness of our model. Among them, HyVulDect-Grpah represents only the graph-level features extracted by graph neural networks, and HyVulDect-Tokens represents only the tokens-level features of the taint propagation chain. Table 5 shows the comparison results of the HyVulDect model ablation experiment. It can be seen that the accuracy of HyVulDect-Tokens, which only considers the Tokens-level features of the source code, is only 87%, which is 4% lower than the method of fusion features. Similarly, the accuracy of the HyVulDect-Grpah model that only considers graph-level features is only 88%. This experiment shows that the model detection effect of fusion features is better because it can extract richer semantic information in the source code.

HyVulDect uses CPG to represent source code, which can compound a variety of semantic information. From Fig. 13, it can be found that the CPG-based model detection ability is better because it combines the control dependence and data dependence in the source code. At the same time, we found that the CDG-based model based on control dependence are lower than the DDG-based model based on data dependence. It can also be found from the side that in source code vulnerability detection, the contribution of data flow is greater.

Q3: Is XGboost the best classifier?

In order to achieve the optimal detection effect, we compared several different classifiers. As Table 6 shows the experimental results of different classifiers, we found that XGboost has the best classification ability, so we chose XGboost as the features classifier.

Q4: How does the detection capability compare with other models and tools?

Table 6
Classification effects of different classifiers(%).

Classifier	Precision	Recall	F1-Score	Acc
RF	0.91	0.90	0.90	0.90
SVM	0.90	0.89	0.89	0.89
BILSTM	0.90	0.90	0.90	0.90
XGboost	0.92	0.91	0.91	0.91

Table 7
Comparison of existing methods and tools(%).

System	Precision	Recall	F1-Score
Devign	0.6205	0.6442	0.5733
VUDDY	0.7547	0.7400	0.7362
BGNN4VD	0.8472	0.8460	0.8459
Flawfinder	0.5477	0.5095	0.4547
Cppcheck	0.5805	0.5707	0.5672
RATS	0.4291	0.4400	0.4288
HyVulDect	0.8964	0.8937	0.8935

There are many existing detection models and mature tools for C/C++. We have selected the following advanced models for comparison.

Devign (Zhou et al., 2019): This is a model based on a general graph neural network, which uses the graph embedding layer to encode the function source code into a joint graph structure with comprehensive program semantics, a gated graph recurrent layer to aggregate and pass information about neighboring nodes in the graph to learn the features of the nodes, and finally a Conv module is used to extract meaningful node representations for graph-level prediction.

VUDDY (Kim et al., 2017): A method of scalable detection of vulnerable code clones, using the similarity of the vulnerable code to realize the automatic discovery of the vulnerable code. Also clones with common modifications (e.g. variable names) can be detected.

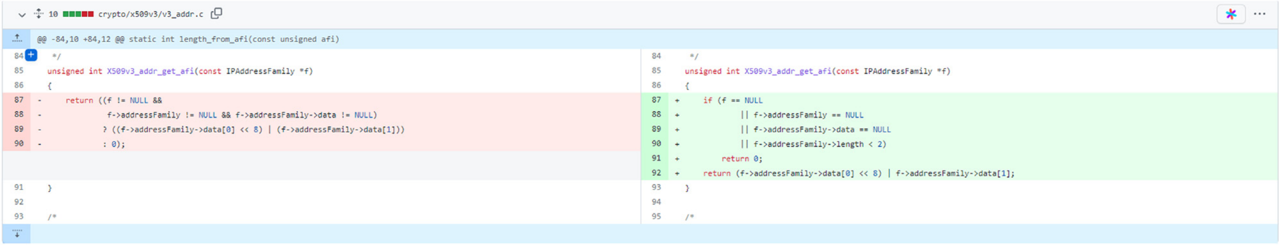
BGNN4VD (Cao et al., 2021): The syntactic and semantic information of the source code is extracted by AST, CFG and DFG, and then the vectorized source code is used as input to a bi-directional graph neural network (BGNN) to learn the different functions between vulnerable and non-vulnerable code. Finally a convolutional neural network (CNN) is used to further extract features and detect vulnerabilities by a classifier.

Flawfinder (Ferschke et al., 2012): This is an open source static scanning and analysis tool for C/C++, which performs static search based on the internal dictionary database to match simple defects and vulnerabilities. The Flawfinder tool does not need to compile C/C++ code, and can directly scan and analyze the code.

Cppcheck (Cppcheck, 2021): This is a static check tool for C/C++ code defects. Unlike the C/C++ compiler and many other analysis tools, it does not check for syntax errors in the code. Cppcheck only checks the types of bugs that the compiler cannot check.

RATS (RAST, 2014): A code security audit tool that can scan C, C++, Perl, PHP and Python source code to check out some common security issues.

It can be seen from Table 7 that the detection ability of HyVulDect is stronger than the existing open source detection tools Flawfinder, Cppcheck and RATS. The main reason is that these tools generally use simple parsers and vulnerability rules, the generation of which relies on expert knowledge. However, it is difficult to consider the triggering rules of all kinds of vulnerabilities due to the manual defined vulnerability rules, and the imperfect rules lead to high false positives and false negatives in vulnerability detection. At the same time, in comparison with the detection model based on deep learning and graph neural network, we found that HyVulDect is also better than these methods. HyVulDect uses



```
10 crypto/x509v3/v3_addr.c
@@ -84,10 +84,12 @@ static int length_from_afi(const unsigned afi)
```

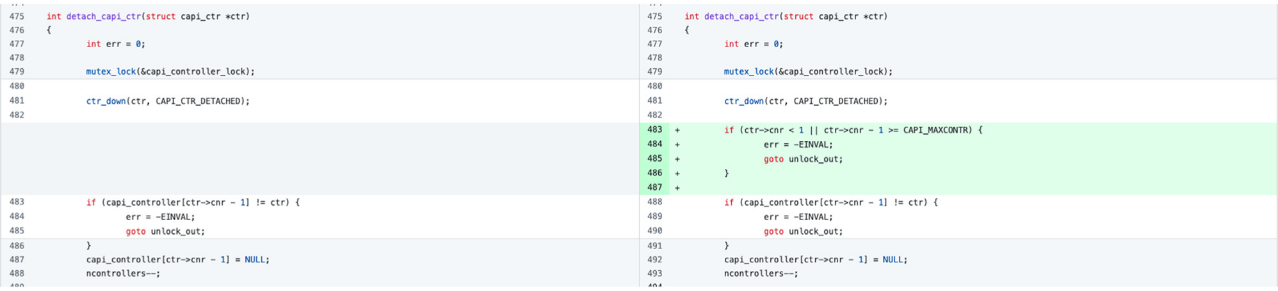
```
 */
unsigned int X509v3_addr_get_afi(const IPAddressFamily *f)
{
-    return ((f != NULL &&
-             f->addressFamily != NULL && f->addressFamily->data != NULL)
-            ? ((f->addressFamily->data[0] << 8) | (f->addressFamily->data[1]))
-            : 0);
}

/*
```

```
 */
unsigned int X509v3_addr_get_afi(const IPAddressFamily *f)
{
+    if (f == NULL
+            || f->addressFamily == NULL
+            || f->addressFamily->data == NULL
+            || f->addressFamily->length < 2)
+        return 0;
+    return (f->addressFamily->data[0] << 8) | f->addressFamily->data[1];
}

/*
```

Fig. 14. CVE-2017-3735 vulnerability and patch source code.



```
int detach_capi_ctr(struct capi_ctr *ctr)
{
        int err = 0;

        mutex_lock(&capi_controller_lock);

        ctr_down(ctr, CAPI_CTR_DETACHED);

        if (capi_controller[ctr->cnr - 1] != ctr) {
                err = -EINVAL;
                goto unlock_out;
        }
        capi_controller[ctr->cnr - 1] = NULL;
        ncontrollers--;
```

```
int detach_capi_ctr(struct capi_ctr *ctr)
{
        int err = 0;

        mutex_lock(&capi_controller_lock);

        ctr_down(ctr, CAPI_CTR_DETACHED);

+       if (ctr->cnr < 1 || ctr->cnr - 1 >= CAPI_MAXCONTR) {
+               err = -EINVAL;
+               goto unlock_out;
+       }
+
        if (capi_controller[ctr->cnr - 1] != ctr) {
                err = -EINVAL;
                goto unlock_out;
        }
        capi_controller[ctr->cnr - 1] = NULL;
        ncontrollers--;
```

Fig. 15. CVE-2021-43389 vulnerability and patch source code.

Table 8
Model performance comparison experimental results.

System	Training time(s)	Predicted time(s)
Devign	3589.810	3.063
VUDDY	/	0.180
BGNN4VD	6125.166	3.121
Flawfinder	/	0.069
Cppcheck	/	0.127
RATS	/	0.008
HyVulDect	5221.945	3.116

gated graph neural network to extract graph-level features, including control flow and data flow information, and extracts tokens-level features from source code as a semantic supplement, which makes its detection ability better.

The performance comparison results are shown in Table 8. The experimental results show that HyVulDect is smaller than BGNN4VD in both training and prediction time. (1): BGNN4VD has limited preprocessing of source code, and many useless codes aggravate the model performance. HyVulDect can extract the critical part of the code by program slicing, which reduces the time overhead of the model. (2): HyVulDect adopts a parallel structure. In the process of graph-level feature and tokens-level feature extraction, it can be executed in parallel to synchronize training and feature extraction, which greatly improves the model's efficiency. In contrast, BGNN4VD adopts a sequence structure. It must first train BGNN to get the state vector-matrix before using the convolution module for feature compression and extraction. Finally, use the classifier to complete the vulnerability classification, which will undoubtedly increase the training and prediction time of the model. At the same time, BGNN4VD only considers the graph-level features of the source code, which results in a lower accuracy than HyVulDect.

Devign is a simple model, it only uses GCN to extract the graph-level features of the source code. It directly constructs the graph structure of the source code and makes prediction. Lots of useless code results in a larger scale of the code property graph, which is more time consuming in the learning process and not much different from HyVulDect in prediction time.

Devign, BGNN4VD, and HyVulDect systems need to build the source code as a code property graph, which consume more time in this process. The four tools of the VUDDY, Flawfinder, Cppcheck, and RATS are purely static analyses without any machine learning models. Therefore they do not require training and spend less time in the prediction process, but the results are poor.

Q5: How effective is the HyVulDect in actual application?

Although many existing methods have good detection capabilities on public datasets, they do not work well in real scenarios. The main reason is that the existing vulnerabilities are complex and the trigger conditions are harsh. In order to further test the detection ability of the HyVulDect system in real scenarios, we select the latest leaked CVE vulnerability to evaluate our model.

As shown in Table 9, HyVulDect has detected multiple vulnerabilities in the open source projects PHP, FFmpeg, Linux, QEMU, and OpenSSL. For example, CVE-2020-7059 is a buffer error vulnerability in versions of PHP prior to 7.2.27 that could be exploited to cause an information leak or denial of service. The main reason is that the *fgets*() function is not handled properly. This type of vulnerability is closely related to data flow, and HyVulDect can effectively detect this type of vulnerability by modeling its semantic features through taint analysis and code property graphs. The patch source code for CVE-2017-3735 is shown in Fig. 14, While parsing an IPAddressFamily extension in an X.509 certificate, it is possible to do a one-byte overread. This would result in an incorrect text display of the certificate. Fig. 15 shows the vulnerability and patch source code of CVE-2021-43389, An issue was discovered in the Linux kernel before 5.14.15. There is an array-index-out-of-bounds flaw in the *detach_capi_ctr* function in *drivers/isdn/capi/kcapi.c*. The vulnerabilities are closely related to data flow and control flow. HyVulDect integrates control dependency and data dependency semantic information, which can detect this type of vulnerabilities well.

HyVulDect can detect CVE vulnerabilities that already exist. It can also detect some unknown vulnerabilities. As shown in Table 10, we found many undisclosed vulnerabilities not publicly published on the NVD and were secretly fixed by developers in the version iterations.

Table 9
A part of vulnerabilities detected by HyVulDect on the real world.

Project	CVE ID	Vulnerability file	Release date
OpenSSL	CVE-2017-3735	crypto/x509v3/v3_addr.c	08/28/2017
	CVE-2021-23839	crypto/rsa/rsa_ssl.c	02/16/2021
	CVE-2021-3711	crypto/sm2/sm2_crypt.c	08/24/2021
Linux	CVE-2020-9391	mm/mmap.c,mm/mremap.c	02/25/2020
	CVE-2020-8648	drivers/tty/vt/selection.c	02/05/2020
	CVE-2021-33624	kernal/bpf/verifier.c	06/23/2021
	CVE-2021-43389	drivers/isdn/capi/kcapi.c	11/09/2021
QEMU	CVE-2019-14378	slirp/src/ip_input.c	07/29/2019
	CVE-2021-3546	contrib/vhost-user-gpu/virgl.c	06/02/2021
	CVE-2021-20255	hw/net/eepro100.c	03/09/2021
PHP	CVE-2020-7059	ext/standard/string.c	02/10/2020
	CVE-2018-7584	ext/standard/http_fopen_wrapper.c	03/01/2018
FFmpeg	CVE-2021-33815	libavcodec/exr.c	06/03/2021
	CVE-2021-38114	libavcodec/dnxhddec.c	08/04/2021

Table 10
Instances of identified undisclosed vulnerabilities.

Project	Upload date	Vulnerability file	Vulnerability type
OpenSSL	14/02/2017 (7d061···99fbc)	ssl/statem/extensions.c	null point dereference
PHP	24/09/2016 (70973···7faa2)	ext/gd/libgd/gd_gd2.c	integer overflow
Wireshark	06/09/2016 (9b39d···3adfe)	extcap.c	memory leak

6. Conclusion

Aiming at the issues of existing vulnerability mining methods, we proposed a graph neural network vulnerability mining system named HyVulDect based on hybrid semantics, which slices the source code to extract vital lines of code, and further constructs code property graphs with composite semantics, and uses gated graph neural network to extract deep features of the code. At the same time, we found that most of the vulnerabilities come from data flow dependencies. For this reason, we use taint analysis methods to extract tokens-level features of the code context. Finally, we use the classifier to classify the fusion features for source code vulnerability detection. Experiments on the SARD and CVE datasets found that the F1-score of the HyVulDect reached 91%, and its detection ability was superior to existing methods and open source detection tools. It can also detect actual CVE vulnerabilities, which have a particular application value. The method proposed in this paper still has some space for improvement. At present, it can only detect whether there are vulnerabilities in the source code, but it can not locate the specific location of vulnerabilities. In the future, our work will also be devoted to solving these problems.

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.

CRediT authorship contribution statement

Wenbo Guo: Conceptualization, Methodology, Software, Data curation, Writing – original draft. **Yong Fang:** Conceptualization, Methodology, Investigation. **Cheng Huang:** Conceptualization, Methodology, Validation, Writing – review & editing. **Haoran Ou:** Investigation, Software, Data curation. **Chun Lin:** Investigation, Data curation, Writing – review & editing. **Yongyan Guo:** Investigation, Software, Data curation, Writing – review & editing.

Acknowledgments

This research is funded by the National Natural Science Foundation of China (U20B2045, 61902265).

References

Ben-Nun, T., Jakobovits, A.S., Hoefler, T., 2018. Neural code comprehension: a learnable representation of code semantics. In: Proceedings of the 32nd International Conference on Neural Information Processing Systems, pp. 3589–3601.

Cao, S., Sun, X., Bo, L., Wei, Y., Li, B., 2021. BGNN4VD: constructing bidirectional graph neural-network for vulnerability detection. Inf Softw Technol 136, 106576.

Chen, T., Xu, R., He, Y., Wang, X., 2017. Improving sentiment analysis via sentence type classification using BiLSTM-CRF and CNN. Expert Syst Appl 72, 221–230.

Church, K.W., 2017. Word2vec. Nat Lang Eng 23 (1), 155–162.

Cppcheck, 2021. https://github.com/danmar/cppcheck.

Dey, R., Salem, F.M., 2017. Gate-variants of gated recurrent unit (gru) neural networks. In: 2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS). IEEE, pp. 1597–1600.

Duan, X., Wu, J., Ji, S., Rui, Z., Luo, T., Yang, M., Wu, Y., 2019. VulSniper: Focus your attention to shoot fine-grained vulnerabilities. In: IJCAI, pp. 4665–4671.

Fang, Y., Han, S., Huang, C., Wu, R., 2019. Tap: a static analysis model for PHP vulnerabilities based on token and deep learning technology. PLoS ONE 14 (11), e0225196.

Ferschke, O., Gurevych, I., Rittberger, M., 2012. Flawfinder: A modular system for predicting quality flaws in wikipedia. In: CLEF (Online Working Notes/Labs/-Workshop), pp. 1–10.

Guo, W., Huang, C., Niu, W., Fang, Y., 2021. Intelligent mining vulnerabilities in python code snippets. Journal of Intelligent & Fuzzy Systems 41 (2), 3615–3628.

Ji, Y., Cui, L., Huang, H.H., 2021. Buggraph: Differentiating source-binary code similarity with graph triplet-loss network. In: Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security, pp. 702–715.

Joern, 2019. https://github.com/joernio/joern.

Ke, G., Xu, Z., Zhang, J., Bian, J., Liu, T.-Y., 2019. Deepgbm: A deep learning framework distilled by GBDT for online prediction tasks. In: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, pp. 384–394.

Kim, S., Woo, S., Lee, H., Oh, H., 2017. VUDDY: a scalable approach for vulnerable code clone discovery. In: 2017 IEEE Symposium on Security and Privacy (SP). IEEE, pp. 595–614.

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

Li, Y., Wang, S., Nguyen, T.N., Van Nguyen, S., 2019. Improving bug detection via context-based code representation learning and attention-based neural networks. Proceedings of the ACM on Programming Languages 3 (OOPSLA), 1–30.

Li, Z., Zou, D., Xu, S., Jin, H., Qi, H., Hu, J., 2016. Vulpecker: an automated vulnerability detection system based on code similarity analysis. In: Proceedings of the 32nd Annual Conference on Computer Security Applications, pp. 201–213.

Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., Chen, Z., 2021. Sysevr: a framework for using deep learning to detect software vulnerabilities. IEEE Trans Dependable Secure Comput.

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

Lin, G., Wen, S., Han, Q.-L., Zhang, J., Xiang, Y., 2020. Software vulnerability detection using deep neural networks: a survey. Proc. IEEE 108 (10), 1825–1848.

Lopez, M.M., Kalita, J., 2017. Deep learning applied to nlp. arXiv preprint arXiv:1703.03091.

Minaee, S., Boykov, Y.Y., Porikli, F., Plaza, A.J., Kehtarnavaz, N., Terzopoulos, D., 2021. Image segmentation using deep learning: a survey. IEEE Trans Pattern Anal Mach Intell.

Nasirloo, H., Azimzadeh, F., 2018. Semantic code clone detection using abstract memory states and program dependency graphs. In: 2018 4th International Conference on Web Research (ICWR). IEEE, pp. 19–27.

Neamtiu, I., Foster, J.S., Hicks, M., 2005. Understanding source code evolution using abstract syntax tree matching. In: Proceedings of the 2005 International Workshop on Mining Software Repositories, pp. 1–5.

NIST, 1901. https://www.nist.gov/.

Ponta, S.E., Plate, H., Sabetta, A., 2018. Beyond metadata: Code-centric and usage-based analysis of known vulnerabilities in open-source software. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp. 449–460.

Rabheru, R., Hanif, H., Maffeis, S., 2020. A hybrid graph neural network approach for detecting php vulnerabilities. arXiv preprint arXiv:2012.08835.

RAST, 2014. https://code.google.com/archive/p/rough-auditing-tool-for-security/.

SARD, 2017. https://samate.nist.gov/SARD/.

Scarselli, F., Gori, M., Tsoi, A.C., Hagenbuchner, M., Monfardini, G., 2008. The graph neural network model. IEEE Trans. Neural Networks 20 (1), 61–80.

Sun, H., Cui, L., Li, L., Ding, Z., Hao, Z., Cui, J., Liu, P., 2021. Vdsimilar: vulnerability detection based on code similarity of vulnerabilities and patches. Computers & Security 110, 102417.

Sundermeyer, M., Schlüter, R., Ney, H., 2012. Lstm neural networks for language modeling. In: Thirteenth Annual Conference of the International Speech Communication Association.

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., Bengio, Y., 2017. Graph attention networks. arXiv preprint arXiv:1710.10903.

Wang, D., Chen, J., 2018. Supervised speech separation based on deep learning: an overview. IEEE/ACM Trans Audio Speech Lang Process 26 (10), 1702–1726.

Wang, H., Ye, G., Tang, Z., Tan, S.H., Huang, S., Fang, D., Feng, Y., Bian, L., Wang, Z., 2020. Combining graph-based learning with automated data collection for code vulnerability detection. IEEE Trans. Inf. Forensics Secur. 16, 1943–1958.

Wang, M., Zheng, D., Ye, Z., Gan, Q., Li, M., Song, X., Zhou, J., Ma, C., Yu, L., Gai, Y., Xiao, T., He, T., Karypis, G., Li, J., Zhang, Z., 2019. Deep graph library: a graph-centric, highly-performant package for graph neural networks. arXiv preprint arXiv:1909.01315.

Xiaomeng, W., Tao, Z., Runpu, W., Wei, X., Changyu, H., 2018. Cpgva: Code property graph based vulnerability analysis by deep learning. In: 2018 10th International Conference on Advanced Infocomm Technology (ICAIT). IEEE, pp. 184–188.

Zhang, H., Wang, S., Li, H., Chen, T.-H.P., Hassan, A.E., 2021. A study of C/C++ code weaknesses on stack overflow. IEEE Trans. Software Eng..

Zheng, L., Yuan, H., Peng, X., Zhu, G., Guo, Y., Deng, G., 2019. Research and implementation of web application system vulnerability location technology. In: The International Conference on Cyber Security Intelligence and Analytics. Springer, pp. 937–944.

Zhou, Y., Liu, S., Siow, J., Du, X., Liu, Y., 2019. Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks. Advances in Neural Information Processing Systems 2019. Neural Information Processing Systems (NIPS).

Wenbo Guo received the B.Eng. degree in cyberspace security from Sichuan University, Chengdu, China, in 2020, where he is currently pursuing the masters degree with the School of Cyber Science and Engineering. His current research interests include Web security and artificial intelligence.

Yong Fang received the Ph.D. degree from Sichuan University, Chengdu, China, in 2010. He is currently a Professor with School of Cyber Science and Engineering, Sichuan University, China. His research interests include network security, Web security, Internet of Things, Big Data and artificial intelligence.

Cheng Huang received the Ph.D. degree from Sichuan University, Chengdu, China, in 2017. From 2014 to 2015, he was a visiting student at the School of Computer Science, University of California, CA, USA. He is currently an Associate Professor at the School of Cyber Science and Engineering, Sichuan University, Chengdu, China. His current research interests include Web security, attack detection, artificial intelligence.

Haoran Ou received the B.Eng. degree in cyberspace security from Sichuan University, Chengdu, China, in 2020, where he is currently pursuing the masters degree with the School of Cyber Science and Engineering. His current research interests include Web security and artificial intelligence.

Chun Lin is currently pursuing the masters degree with the School of Cyber Science and Engineering. His current research interests include Web security and artificial intelligence.

Yongyan Guo received the B.Eng. degree in cyberspace security from Sichuan University, Chengdu, China, in 2020, where he is currently pursuing the masters degree with the School of Cyber Science and Engineering. His current research interests include Web security and artificial intelligence.