Hindawi
IET Information Security
Volume 2024, Article ID 4313185, 15 pages
https://doi.org/10.1049/2024/4313185

Research Article

VulMPFF: A Vulnerability Detection Method for Fusing Code Features in Multiple Perspectives

Xiansheng Cao[1] **Junfeng Wang**[2] **Peng Wu**[3] **and Zhiyang Fang**[1]

[1]*School of Cyber Science and Engineering, Sichuan University, Chengdu 610207, China*
[2]*College of Computer Science, Sichuan University, Chengdu 610065, China*
[3]*School of Information and Engineering, Sichuan Tourism University, Chengdu 610100, China*

Correspondence should be addressed to Junfeng Wang; wangjf@scu.edu.cn

Received 26 November 2023; Revised 25 February 2024; Accepted 4 March 2024; Published 22 March 2024

Academic Editor: Guowen Xu

Source code vulnerabilities are one of the significant threats to software security. Existing deep learning-based detection methods have proven their effectiveness. However, most of them extract code information on a single intermediate representation of code (IRC), which often fails to extract multiple information hidden in the code fully, significantly limiting their performance. To address this problem, we propose VulMPFF, a vulnerability detection method that fuses code features under multiple perspectives. It extracts IRC from three perspectives: code sequence, lexical and syntactic relations, and graph structure to capture the vulnerability information in the code, which effectively realizes the complementary information of multiple IRCs and improves vulnerability detection performance. Specifically, VulMPFF extracts serialized abstract syntax tree as IRC from code sequence, lexical and syntactic relation perspective, and code property graph as IRC from graph structure perspective, and uses Bi-LSTM model with attention mechanism and graph neural network with attention mechanism to learn the code features from multiple perspectives and fuse them to detect the vulnerabilities in the code, respectively. We design a dual-attention mechanism to highlight critical code information for vulnerability triggering and better accomplish the vulnerability detection task. We evaluate our approach on three datasets. Experiments show that VulMPFF outperforms existing state-of-the-art vulnerability detection methods (i.e., Rats, FlawFinder, VulDeePecker, SySeVR, Devign, and Reveal) in Acc and F1 score, with improvements ranging from 14.71% to 145.78% and 152.08% to 344.77%, respectively. Meanwhile, experiments in the open-source project demonstrate that VulMPFF has the potential to detect vulnerabilities in real-world environments.

1. Introduction

With the rapid development of the software supply chain, open-source software plays an increasingly important role in software development. In this context, developers are more likely to inadvertently introduce code vulnerabilities. Unlawful elements use these vulnerabilities to attack the fragile supply chain, which will not only bring threats to individuals and enterprises but also threaten national security. Security testing of source code during the software development cycle to reduce vulnerabilities in software at the source can help enhance software security and improve the security of the software supply chain.

To detect and eliminate vulnerabilities in software, techniques such as software static analysis [1], taint propagation analysis [2], symbolic execution [3], and fuzzy testing [4] have been applied to vulnerability detection. Unfortunately, the traditional techniques and methods are becoming less and less adaptable to software's increasing size and complexity. For example, static analysis methods have good adaptability but require experienced experts to manually define vulnerability patterns, which is usually costly and difficult to detect more complex triggering conditions, and has high false positives; dynamic analysis methods are good at capturing vulnerabilities during software runtime, but because they cannot reach a lot of code areas, and in combination with the current status quo of large-scale and high-complexity software, they often have high false negatives. Deep learning-based methods are also widely used in software vulnerability detection and have achieved relatively good results. Analyzing the source code to extract intermediate representation of code (IRC) containing more code information is the key to applying deep learning to code vulnerability detection. Some approaches view the code as a natural sequence and use recurrent neural networks (RNN) [5–7], convolutional neural networks (CNN) [8, 9], and their variant networks for vulnerability detection. This class of methods can effectively capture sequential and contextual information in the code. However, they tend to ignore the structural nature of the code, lose semantic information about vulnerability triggers, and fail to extract lexical and syntactic information adequately. There are also methods to represent code as code relationship graphs as IRC, such as abstract syntax trees (ASTs) [10], control flow graphs (CFG) [11], data flow graphs (DFG) [12], program dependency graphs (PDG) [13], and code property graphs (CPGs) [14], which in turn use graph neural networks (GNNs) to detect vulnerabilities [15–18]. This class of methods considers the structural characteristics of the code and can extract semantic information from the code. However, the contextual information of the vulnerability trigger and critical lexical and syntactic information will be lost. At the same time, different vulnerability-triggering patterns may be hidden in the semantic information of different subgraphs. Most existing graph-based approaches treat the code graph as a graph structure in which feature information is passed between nodes that do not distinguish a subgraph. Most existing graph-based approaches treat the code graph as a graph structure in which feature information is passed between nodes that do not distinguish a subgraph without sufficiently considering the heterogeneous nature of the graph.

This paper proposes a vulnerability detection method, VulMPFF, that fuses code features in multiple perspectives. IRC is extracted from three perspectives: code sequence, lexical and syntactic relations, and graph structure to capture vulnerability information in the code, which effectively realizes the complementary information of multiple IRCs. First, serialized AST and CPG are extracted on the preprocessed function code. After initialization, the serialized AST is fed into the Bi-LSTM model with a self-attention mechanism to learn the serialized AST features. At the same time, four subgraphs are extracted on CPG for different edges. The GNN with attention mechanism is used to update the node information on each subgraph to learn different semantic information about vulnerability triggering in the subgraphs, and then merge the node representations of the four subgraphs into the whole heterogeneous graph and aggregate the heterogeneous graph node information into the final CPG features. Finally, the serialized AST and graph features are fused into a fusion feature fed into the classifier for vulnerability detection. This method effectively combines the advantages of IRC extracted from different perspectives, extracts the contextual information of vulnerability triggering in the code, and captures the semantic information and the lexical and syntactic information of different complex streams in the code, which provides a better ability to recognize vulnerabilities.

The novelties and contributions of this paper are as follows. (1) A new vulnerability detection framework, VulMPFF, is proposed, which extracts IRC from three perspectives: code sequences, lexical and syntactic relations, and graph structures, effectively realizing the complementarity of information from different IRCs and having a stronger capability of capturing code vulnerability information. (2) We improve the way of extracting code information in heterogeneous graphs by updating the respective node features in different subgraphs and aggregating them into CPGs, which fully considers the heterogeneity of code relationship graphs. (3) We use serialized ASTs instead of directly extracting code sequences to extract code information, fully preserving the contextual information in the code while compensating for the missing lexical and syntactic information in the AST subgraphs of CPG. (4) Experiments based on this design on multiple datasets show that VulMPFF outperforms six state-of-the-art vulnerability detection methods (i.e., Rats [19], FlawFinder [20], VulDeePecker [5], SySeVR [7], Devign [15], and Reveal [16]) in terms of Acc and F1 score are improved from 14.71% to 145.78% and 152.08% to 344.77%, respectively, with the ability to detect actual open source projects.

In terms of structure, this paper is divided into the following sequence: Section 2 presents the related work. Section 3 presents the preliminaries of this paper. Section 4 presents the overall architecture of VulMPFF. The experimental evaluation is given in Section 5. Section 6 presents a discussion of the limitations of VulMPFF. Section 7 concludes this paper.

2. Related Work

This section presents the background and related work in terms of both code intermediate representation and existing vulnerability detection methods.

2.1. Intermediate Representation of Source Code. Converting source code into a suitable IRC is the core problem of source code-oriented vulnerability detection models [21]. Generally, the richer the code information in IRC, the better the detection performance. Existing IRCs include treating source code as natural sequences and converting to ASTs and graph structures.

Source code sequence is a sequence or a set of tokens containing key elements such as keywords, identifiers, and operators obtained through lexical analysis of the source code IRC [9], which can be fully extracted to the contextual and lexical information in the code sequence, effectively identifying the vulnerabilities of remote dependencies and lexical change triggering characteristics.

AST is an IRC that converts source code into a tree structure [10]. It is an abstract representation of the syntactic structure of the source code, which represents the syntactic structure of the programing language in a tree form; each node in the tree represents a structure in the source code, corresponding to the main code elements. AST contains syntactic information in the source code and lexical and syntactic information in the source code, and vulnerabilities triggered by lexical and syntactic errors can be effectively identified using AST.

CFG is an internal control graph that converts source code into a directed graphical structure describing all possible traversal paths during code execution through control dependencies in an AST [11]. Each of its nodes corresponds to a statement in the source code, and it usually requires the code to have complete functional logic and to be compiled before it can be generated. CFG contains the functional logic relationships in the code, and each of its edges represents the direction of the control flow within the code. Control-dependency-related vulnerabilities can be effectively identified using CFG.

DFG is an internal control graph representing the logical flow of data through the code and the transformation process [12]. It can represent the data dependencies of a series of operations in source code from the data flow and processing perspective, and data-dependency-related vulnerabilities can be identified using DFG.

PDG is an IRC that converts source code into a graphical structure consisting of a data dependence graph (DDG) and a control dependence graph (CDG) [13]. It extracts data dependence and control dependence relationships between code elements based on AST, and each node can correspond to a statement of the source code; using PDG, more code details can be abstracted, and vulnerabilities related to data dependence and control dependence can be identified.

CPG is a type of IRC that converts source code into a graph structure, integrating multiple relational graphs of AST, CFG, and PDG into a single graph containing multiple complex flow relationships [14]. Multiple semantic information can be extracted using CPG, which synthesizes the advantages of multiple graph-level IRCs and identifies multiple types of vulnerabilities.

There are other approaches to transforming code into an extended graph structure IRC, such as extending the AST to a code-pointing relationship graph containing four kinds of edges (AST, CFG, DFG, and NCS) [15], and treating the source code as a simplified code property graph (SCPG) [22], which start from the goal of extracting a more comprehensive semantics and identifying a greater variety of types of vulnerabilities.

2.2. Vulnerability Detection Methods for Source Code

2.2.1. Pattern-Based Methods. Pattern-based vulnerability detection methods usually require experienced experts to manually define vulnerability patterns, which form large databases of vulnerability rules [1]. Since the pattern-based is predefined, detection tools such as Rats [19], CppCheck [23], Flawfinder [20], and Checkmarx [24], which usually use this approach, can quickly detect known vulnerabilities in the pattern-based. However, this approach is labor-intensive and dependent on vulnerability rules, and discovering new and relatively complex vulnerabilities is often challenging. In addition, developing vulnerability rules with more complex triggering conditions is costly, and building a comprehensive vulnerability pattern database is even more challenging. Thus, pattern-based approaches are limited to the vulnerabilities in the vulnerability rule database and cannot do anything about emerging vulnerabilities.

2.2.2. Code Cloning-Based Methods. Code clone-based methods are methods for discovering vulnerabilities introduced by code clones. This approach generally extracts IRC rather than performing similarity detection directly with code. Vulpecker [25] devised a set of definitions for software patches and proposed a similarity detection method that can automatically identify vulnerabilities and enable localization for remediation when vulnerabilities and source code are given. Vuddy [26] is an extensible method that can quickly detect code cloning vulnerabilities by calculating the hash of a sequence of strings and comparing the hashes. VDSimilar [27] is implemented based on vulnerability code and vulnerability repair patch code, and it chooses the Bi-LSTM model with an attention mechanism to learn the difference between a pair of patches and the homogeneity of classes between a pair of vulnerabilities for vulnerability detection. Effective IRC is the key to this type of approach, and although very efficient, its detection of unknown vulnerabilities is weak, often only detecting vulnerabilities introduced by code cloning, and usually requires a larger dataset of cloned vulnerabilities to learn as many vulnerability patterns as possible, which is a significant limitation.

2.2.3. Deep Learning-Based Methods. Deep learning-based methods have powerful modeling and intelligent pattern-learning capabilities to extract source code information automatically. Russell et al. [9] used a lexical analyzer to extract the sequence tokens of the source code and then used a CNN model to learn the source code features and later used a random forest classifier to identify the vulnerabilities. VulDeePecker [5] constructs a series of code gadgets for a collection of code statements based on heuristics and then processes these for vulnerability detection using Bi-LSTM models. SySeVR [7] can detect multiple types of vulnerabilities by customizing SyVCs and SeVCs, which represent syntactic and semantic information of the code, respectively, and designing automatic extraction methods. Many other researchers consider source code as natural sequences [28, 29], usually extracting the code sequence information and then using RNN or CNN correlation models for vulnerability detection.

In recent years, many researchers have made significant progress using GNNs [30] for vulnerability detection. Devign [15] extended the AST of function code into a directed graph of the code containing four types of edges (AST, CFG, DFG, and NCS) and used a GGNN model to efficiently extract the semantic information in the source code to identify vulnerabilities. Reveal [16] extracted the function code as CPG and used the GGNN model to identify vulnerabilities. Wu et al. [22] considered the source code as a SCPG and used GGNN for vulnerability detection. Despite the progress, the existing GNN-based methods still have limitations and cannot effectively fuse heterogeneous information in different graphs.

3. Preliminaries

3.1. Problem Formulation. The goal of VulMPFF is to detect vulnerabilities at function-level granularity. We denote codes as $C = \{c_1, c_2, c_3, \ldots, c_n\}$. Their labels are $L = \{0, 1\}^n$, where n represents the number of function codes, 0 illustrates a vulnerable code, and 1 denotes a nonvulnerable code. Our approach aims to discover the optimal mapping of the codes to their corresponding labels $\varphi : C \rightarrow L$. We extract the serialized ASTs and CPGs of the codes as IRCs from different perspectives, and they can be defined as follows:

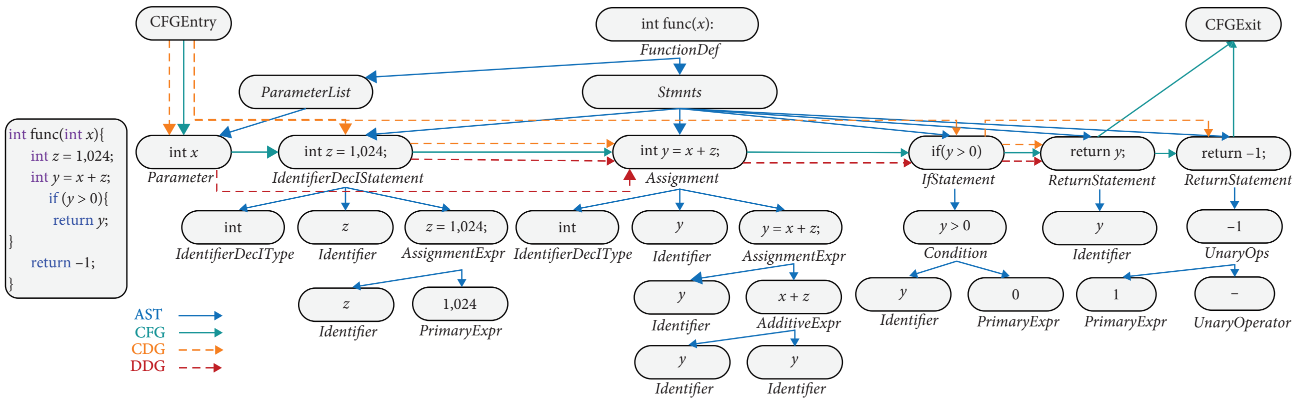


FIGURE 1: CPG of the example function *func()*.

Serialized AST: The serialized AST can be represented as $S = s(T)$, where $T = \{t_1, t_2, t_3, \ldots, t_m\}$ denotes all tokens, and m represents the number of tokens.

CPG: CPG can be represented as $G = g(V, E, A)$, where $V \in X$ denotes all nodes, $E \in Y$ denotes all edges and A means all node attributes, X and Y represent the types of all nodes and edges, respectively. The CPG is a heterogeneous graph with different subgraphs distinguished by different edge types, and then there is $|X| + |Y| > 2$, and in particular, $|Y| = 4$ in the CPG.

Then VulMPFF finds the optimal mapping by minimizing the loss function with the following definition:

$$\min \sum_{i=1}^{n} \ell(\varphi(g_i(V, E, A) || s_i(T), y_i | g_i, s_i)) + \lambda\omega(\varphi), \tag{1}$$

where $\ell(\cdot)$ denotes the loss function, $(\cdot \parallel \cdot)$ denotes the concatenated operation, $\lambda(\cdot)$ denotes the adaptive weight parameter, and $\omega(\cdot)$ denotes the regularization term.

3.2. The Intermediate Representation of the Code Used.

VulMPFF chooses serialized AST as IRC under the code sequence, lexical and syntactic relation perspective, and CPG as IRC under the graph structure perspective. Figure 1 is the schematic diagram of CPG for the function *func*(), where the subgraphs of the AST are consistent with the nodes and structure of the tree-structured AST of *func*().

Serialized AST is derived from a planar transformation based on the tree-structured AST, which consists of nodes with parent–child mapping relationships at different depths [31]. We extract the serialized AST on the tree-structured AST, and Table 1 shows the serialized AST node information table of *func*() function in Figure 1. According to the serialized AST node information table, the DFT algorithm is used to traverse the serialized AST to obtain a planarized sequence as the final serialized AST representation. For node values, a lexical analyzer is used to extract all the tokens along with the traversed node types to form the serialized AST. Taking the *func*() function as an example, we extracted the serialized AST as: *{functiondef, int, func, parameterlist, parameter, int,x, stmnts, identifierdecistatement, int, z, …}*. The serialized AST is very different from pure code sequences. It

TABLE 1: Serialized AST node information table.

Node type	Depth	Values
FunctionDef	0	int func(x)
ParameterList	1	—
Parameter	2	int x
Stmnts	1	—
IdentifierDeclStatement	3	int z = 1,024;
IdentifierDeclType	4	int
…	…	…

extracts the keyword method and syntax information related to vulnerability triggering based on tree-structured AST and, at the same time, retains the contextual information in the original sequence, which has better expressive ability.

CPG contains enough semantic information about the code and can encode the complex flow relationship in the code into the model. It is based on the AST extension to heterogeneous graphs and contains four kinds of relational graphs, AST, CFG, CDG, and DDG, corresponding to four different semantic relations.

4. Methodology

In this section, we discuss in detail our proposed vulnerability detection method, VulMPFF, that fuses code features under multiple perspectives, including the general framework of the method, the data preprocessing process, the extraction methods of serialized AST and CPG features, and the design of the detection classifier.

4.1. Overview.

The research goal of VulMPFF is to detect vulnerabilities at function-level granularity. The general framework of our proposed method is shown in Figure 2, divided into three main parts.

(i) *Data preprocessing*: Multilevel generalization operations are performed on the function code to improve the model's adaptability to code variants, followed by serialized AST and CPG extraction.

(ii) *Source code feature extraction*: It includes two parts: serialized AST feature extraction and CPG feature

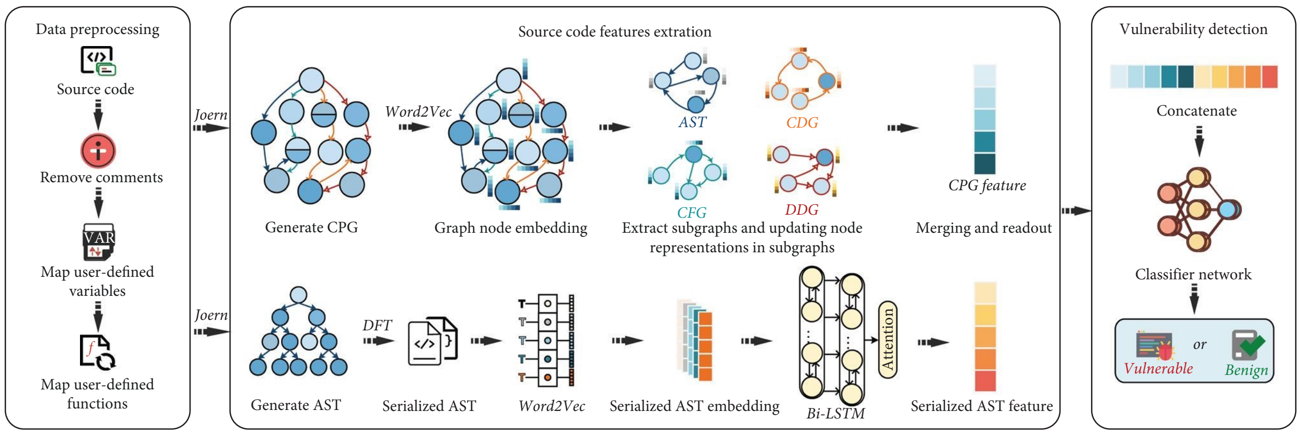


Figure 2: Overall framework of VulMPFF.

extraction. A dual-attention mechanism is introduced to extract serialized AST and CPG features by the Bi-LSTM model and GNN, respectively.

(iii) *Vulnerability detection*: Serialized AST and CPG features are fused to obtain fusion features and vulnerability detection of fusion features at function-level granularity is performed using classifiers.

4.2. Data Processing. Before extracting the IRC of the source code, some generalized preprocessing operations are required to reduce unwanted noise and lower the vectorized dimensionality. The preprocessing improves the adaptability of VulMPFF to common code variants while preserving the rich information in the code. A schematic of the preprocessing operations we perform on the function code is shown in Figure 3, which includes three levels of generalization processing operations.

The first step is to remove comments in the code that do not make sense for vulnerability triggering; while removing the comments, it does not affect various information in the code.

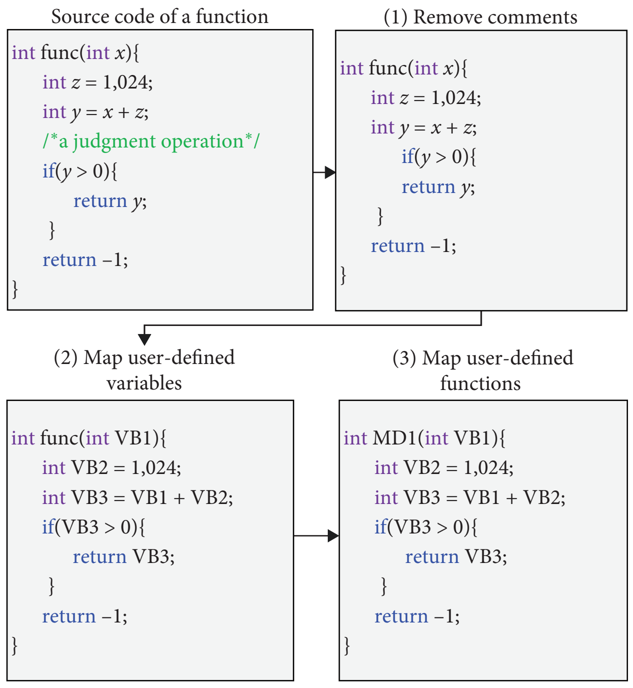
The second step is to map user-defined variable names uniformly, which can reduce the interference due to the program developer's habits and, at the same time, can improve the model's adaptability to code modifications.

The third step is to map user-defined function names.

The mapping rule in steps 2 and 3 replaces the developer-defined variable and function names with VB and MD with a numeric suffix, where the number counts from 1. If there is more than one variable and function, the number is incremented in the order of occurrence. For example, the variable names x, y, and z are mapped to VB1, VB2, and VB3 in step 2, and the function name *func* is mapped to MD1 in step 3.

4.3. Source Code Features Extraction. After preprocessing the function code, VulMPFF uses the powerful C/C++ code analysis tool Joern [32] to generate AST and CPG as IRCs, after which serialized AST features and CPG features are extracted on both IRCs.

4.3.1. Serialized AST and Graph Node Embedding.
(1) Serialized AST Embedding. After obtaining the serialized ASTs, we embedded them into the low-latitude space using a


Source code of a function
(1) Remove comments
(2) Map user-defined variables
(3) Map user-defined functions

```
int func(int x){
    int z = 1,024;
    int y = x + z;
    /*a judgment operation*/
    if(y > 0){
        return y;
    }
    return −1;
}
```

```
int func(int x){
    int z = 1,024;
    int y = x + z;
        if(y > 0){
        return y;
    }
    return −1;
}
```

```
int func(int VB1){
    int VB2 = 1,024;
    int VB3 = VB1 + VB2;
    if(VB3 > 0){
        return VB3;
    }
    return −1;
}
```

```
int MD1(int VB1){
    int VB2 = 1,024;
    int VB3 = VB1 + VB2;
    if(VB3 > 0){
        return VB3;
    }
    return −1;
}
```

Figure 3: Data preprocessing.

pretrained Word2vec [33] model to convert the serialized ASTs into feature vectors that the model could process. We trained the Word2vec model individually for each dataset, and the corpus used was derived from the serialized ASTs of all function samples in each dataset. Considering that the lengths of the serialized ASTs extracted from the function samples were highly inconsistent, we chose 1,000 as the length threshold for the serialized ASTs after examining the datasets used. Short sequences are padded with zeros, and long sequences are truncated.

(2) Graph Node Embedding. After obtaining the CPG, the node information in the graph needs to be embedded into quantizable vectors in the low-dimensional space, which will be used as the initial feature vectors of the nodes that the model can process. We use a pretrained Word2vec model to obtain the initial feature vectors of CPG nodes. Specifically,

the code segments in the nodes are taken, and all tokens are extracted using a lexical splitter. For the case of multiple tokens, the multiple tokens are averaged to obtain the initialized feature vector of the node. Node types also provide a lot of hidden information. We count all the node types in each dataset that participated in the training and finally encode each node type as an integer using one-hot. Finally, the node type encoding is concatenated behind the above initial node feature vector as the final node initialization feature vector.

4.3.2. Serialized AST Feature Extraction. After obtaining the initialized feature vectors of the serialized AST, we use a Bi-LSTM model with an attentional mechanism to capture the contextual, lexical, and syntactic information in which the vulnerability triggers the key. The serialized AST passes through the embedding, Bi-LSTM, and attention layers before accessing the flatten and fully connected layers. We added a Dropout layer to the model for randomly disconnecting a certain percentage of neurons, which can enhance the model's generalization ability and reduce the complexity to some extent. In this section, we choose the Relu function as the activation function to update the feature vectors of the serialized AST.

We denote S as a serialized AST containing n tokens, and each token is mapped to a global vector. Then, S can be expressed by Formula (2):

$$S = [\vec{e}_1 \parallel \vec{e}_2 \parallel \vec{e}_3 \parallel \dots \parallel \vec{e}_n], \tag{2}$$

where vector $\vec{e}_m$ denotes the vector of the mth token in the serialized AST S.

The initialized feature vector of the serialized AST is fed into the Bi-LSTM layer to get the representation of $H = \mathrm{BiLSTM}(S)$. After that, the output of the Bi-LSTM layer is fed into the attention layer. Adding the attention layer computes the interactions of different units in the serialized AST, solves the data dependency problem in long sequences, and highlights the critical information for vulnerability triggering. The computation of the attention layer is as follows.

We denote h_t^T and $h_{t'}^T$ as the outputs of the Bi-LSTM layer and first obtain the implicit representation through an MLP, as shown in Formula (3):

$$m_{t,t'} = \tanh\left(h_t^T W_q + h_{t'}^T W_k + b\right), \tag{3}$$

where W_q and W_k are the key and attention parameters, respectively, and b is the bias parameter. It is then activated by a Sigmoid function as shown in Formula (4):

$$n_{t,t'} = \sigma\left(W_a m_{t,t'} + b'\right), \tag{4}$$

where W_a denotes the attention parameter and b' is the bias parameter. After that, it goes through the Softmax function to get the normalized weights, as shown in Formula (5):

$$u_{t,t'} = \mathrm{softmax}\left(n_{t,t'}\right), \tag{5}$$

where $u_{t,t'}$ denotes the self-attention value, then the output of the last self-attention layer can be expressed as Formula (6):

$$v = \sum_{t'} u_{t,t'} h_{t'}. \tag{6}$$

4.3.3. CPG Feature Extraction. VulMPFF first extracts four subgraphs by edge type on the CPG that has completed node initialization, which are AST:0, CFG:1, CDG:2, and DDG:3. Different subgraphs have different vulnerabilities triggering critical semantic information. Then, update the respective node features in each subgraph separately and aggregate the node features on each subgraph into CPG. Finally, the problem of aggregating all node representations of the heterogeneous graph is viewed as a structured feature classification problem to read out the CPG features over the whole heterogeneous graph.

(1) Updating Node Representations in Subgraphs. We denote the graph-level IRC as $G = \bigcup_{e \in E} G_{\mathrm{sub}}^{\mathrm{e}}$, where e denotes the four edge types and $G_{\mathrm{sub}}^{\mathrm{e}}$ denotes the subgraph. We denote the node in the subgraph $G_{\mathrm{sub}}^{\mathrm{e}}$ as v_m, where m is the node serial number. Then, denote the state of node v_m at the moment $k-1$ as $h_{m,e}^{k-1}$, then node v_m at the moment k as $h_{m,e}^{k}$. The state of node v_m at the moment k is aggregated for its neighboring nodes in subgraph $G_{\mathrm{sub}}^{\mathrm{e}}$, denoted as Formula (7):

$$h_{m,e}^{k} = \mathrm{Aggregate}\left(h_{n,e}^{k}, \forall v_n \in N_{m,e}\right), \tag{7}$$

where $N_{m,e}$ denotes the neighbor node of node v_m in subgraph $G_{\mathrm{sub}}^{\mathrm{e}}$, when subgraph $G_{\mathrm{sub}}^{\mathrm{e}}$ updates each node representation, we also introduce the attention mechanism to distinguish between the importance of a node's neighboring nodes. We chose the multihead mechanism to enhance the expressiveness and generalization of the model. We calculate the correlation coefficient $a_{\mathrm{mn}}^{\mathrm{e}}$ between node v_m and neighbor node v_n in subgraph $G_{\mathrm{sub}}^{\mathrm{e}}$, according to Formula (8):

$$a_{mn}^{e} = M\left(\left[W h_{m,e}^{k-1} \parallel W h_{n,e}^{k-1}\right]\right), n \in N_{m,e}, \tag{8}$$

where $[\cdot \parallel \cdot]$ denotes the high-dimensional features of the merged node v_m with its neighbor node v_n, $\mathrm{M}(\cdot)$denotes a mapping relationship that maps high-dimensional data into a low-dimensional space, and W is the shared parameter matrix. Then, we calculate the attention coefficient a_{mn}^{e} of neighbor node v_n to node v_m, according to Formula (9):

$$\alpha_{mn}^{e} = \frac{\exp(\sigma(a_{mn}^{e}))}{\sum_{n \in N_{m,n}} \exp(\sigma(a_{mn}^{e}))}, \tag{9}$$

where σ is the activation function, the representation of the node v_m can be updated after obtaining the individual attention coefficients of the neighboring nodes of the node v_m to that node, according to Formula (10):

$$h_{m,e}^{k} = \|_{z=1}^{Z} \sigma \left(\sum_{n \in N_{\mathrm{m},e}} \alpha_{mn}^{e^z} W^z h_{n,e}^{k-1} \right), \quad (10)$$

where $\alpha_{mn}^{e^z}$ denotes the zth head of α_{mn}^{e}, and W^z corresponds to the zth head of W. We repeat steps Formulas (2)–(4) to update the node representations to aggregate the information of the node's multistep neighbors and extend the sense field.

(2) Merging Node Representations of Subgraphs. After updating the node representations in different subgraphs, it is necessary to merge the representations of the nodes of different subgraphs over the whole graph. Typical operations include averaging, summing, cascading, maximizing, and minimizing [34]. We use the averaging method to aggregate subgraph node representations, represented by Formula (11):

$$h_{m}^{'} = \frac{\sum_{e \in E} h_{m,e}}{|E|}, \quad (11)$$

where h_m' denotes the feature of node v_m updated by aggregating elements from all neighboring nodes in different edges.

(3) Readout CPG Representation. After the subgraph nodes have finished merging, we read out the graph representation by averaging all the node features in the CPG, as shown in Formula (12):

$$H_{\mathrm{CPG}} = \frac{\sum_{m \in M} h_{m}^{'}}{|M|}, \quad (12)$$

where M denotes the set of all nodes in the CPG.

4.3.4. Vulnerability Detection. After extracting the serialized AST and CPG feature vectors, we fused them to obtain fused feature vectors, determining whether a function is a vulnerability function.

We choose to directly concatenate serialized AST feature vectors and CPG feature vectors to fuse the two feature vectors. The serialized AST feature vector contains lexical and syntactic information and remote dependencies critical for vulnerability triggering. In contrast, the CPG features contain different complex flow semantic information critical for vulnerability triggering, and the fusion of the two can effectively complement each other's code information from multiple perspectives, which can be used to classify and detect vulnerability functions.

We choose multilayer perceptron (MLP) as the classifier to perform the linear transformation on the fused feature vector to extract the abstract features of the code function. The MLP classifier contains three implicit layers and accesses a Dropout layer. The Sigmoid is chosen as the activation function for the classification, denoted as Formula (13):

Table 2: Overview of the datasets.

Dataset	Vul.	No-Vul.	Total	Ratio (Vul.:No-vul.)
Big-Vul$_{\mathrm{sub}}$	10,094	11,027	21,121	1 : 1.09
Big-Vul$_{\mathrm{full}}$	10,094	159,354	169,448	1 : 15.79
Devign	9,824	11,856	21,680	1 : 1.21
Reveal	1,664	18,169	19,833	1 : 10.92

$$\widetilde{y} = \mathrm{Sigmoid}(\mathrm{MLP}(H_{\mathrm{SAST}} \parallel H_{\mathrm{CPG}})), \quad (13)$$

where $\tilde{y}$ is the final detection result and H_{SAST} and H_{CPG} are the function code's serialized AST and CPG features, respectively.

5. Evaluation

In this section, we discuss in detail our proposed vulnerability detection method, VulMPFF, that fuses code features under multiple perspectives, including the general framework of the method, the data preprocessing process, the extraction methods of serialized AST and CPG features, and the design of the detection classifier.

5.1. Experimental Setup

5.1.1. Datasets. To verify the validity of VulMPFF, three different public datasets are collected in this paper. The three datasets are the Big-Vul dataset [35], the Devign dataset [15], and the Reveal dataset [16], as shown in Table 2, for an overview of our collected datasets. We chose Joern [32] to extract AST and CPG in our experiments, so we removed the samples that Joern could not process in each dataset.

The Big-Vul dataset collects 348 real open-source projects from 2002 to 2019. It covers all CVE entries and over 90 different vulnerability types, totaling over 10,000 vulnerability function samples and over 170,000 nonvulnerability function samples. Due to the uneven distribution of vulnerability samples and nonvulnerability samples in this dataset, to validate our method more comprehensively, we extract a balanced dataset based on the Big-Vul dataset, which we call the Big-Vul$_{\mathrm{sub}}$ dataset, with a sample ratio of 1 : 1.09 (negative samples:positive samples). The original Big-Vul dataset, which we call the Big-Vul$_{\mathrm{full}}$ dataset, has a sample ratio 1 : 15.79 (negative samples: positive samples).

The Devign dataset was collected from two open-source projects, FFmpeg and Qemu, at function-level granularity and hand-labeled. It contains about 10,000 vulnerability function samples and 12,000 nonvulnerability function samples, with a sample ratio 1 : 1.21 (negative samples:positive samples), making it a relatively balanced dataset.

The Reveal dataset was collected from two open-source projects, Linux Debian Kernel and Chromium, at function-level granularity. It includes about 1,700 vulnerable function samples and over 18,000 nonvulnerable function samples. The sample ratio is 1 : 10.92 (negative and positive samples), a nonunbalanced dataset.

5.1.2. Evaluation Metrics. In evaluating VulMPFF, we define the confusion matrix, as shown in Table 3.

Table 3: Confusion matrix.

Actual/predicted	Vul.	No-vul.
Vul.	T_p	F_n
No-vul.	F_p	T_n

Table 4: Software and hardware environment for the experiment.

Environment	Version or size
Software	*Ubuntu 18.04.6*
	python = = 3.8.12
	torch = = 0.11.1
	dgl = = 1.1.0
	keras = = 2.9.0
	nltk = = 3.6.5
	gensim = = 3.5.0
	numpy = = 1.22.4
	scikit-learn = = 1.1.2
Hardware	*cpu = = Intel(R) Xeon(R) Gold 6144@3.50 GHz*
	gpu = = 2 × NVIDIA GeForce RTX 4,070 Ti
	ram = = 512 GB
	disk = = 22 TB

The formulas we used to calculate the assessment indicators are as follows:

Accuracy: $\text{Accuracy} = \frac{T_p+T_n}{(T_p+F_n)+(T_n+F_p)}$.

Recall: $\text{Recall} = \frac{T_p}{T_p+F_n}$.

Precision: $\text{Precision} = \frac{T_p}{T_p+F_p}$.

F1 score: $\text{F1} = \frac{2\,\text{precision}\times\text{recall}}{\text{precision}+\text{recall}}$.

5.1.3. Baseline Methods. There are many excellent detection methods and tools for C code. We choose the following six state-of-the-art tools and methods to compare with our approach, including two static detectors, Rats [19] and FlawFinder [20]; two sequence-based methods, VulDeepecker [5] and Sysevr [7]; and two graph structure-based methods, Devign [15] and Reveal [16].

5.1.4. Implementation Details. The software and hardware environments in the experimental setting are shown in Table 4. We assigned each dataset as mutually disjoint training, validation, and test sets using a ratio of 8 : 1 : 1 and randomly shuffled the data at each epoch. We set the model epoch to 100, the batch size to 128, and the Word2Vec word vector dimension to 100 when initializing the serialized AST and CPG node vectors, and to prevent the model from overfitting, we set all the dropout parameters to 0.2, and set the number of bi-LSTM model units to 128. We choose the cross-entropy loss function to compute the parameter loss and the Adam optimizer with a learning rate of 0.0001.

5.2. Results and Analysis. To better evaluate the performance of VulMPFF in detecting vulnerabilities at function-level granularity and its effectiveness in real-world applications, we designed and provided detailed answers to four research questions.

RQ1: How effective is VulMPFF at detecting vulnerabilities at function-level granularity?

To answer this question, we compare the performance of VulMPFF on four datasets with the chosen baseline method. The comparison results are shown in Figure 4. Based on the performance comparison results, we have the following observations.

VulMPFF outperforms existing static vulnerability detection systems Rats and FlawFinder. Rats and FlawFinder have F1 score below 40% on the Big-Vul$_{\text{sub}}$ dataset and perform even worse on the Big-Vul$_{\text{full}}$ dataset, where their F1 score, Pre and Rec, are below 15%. This class of methods is used to detect vulnerabilities by manually defining vulnerability rules by human experts, which is limited by the size of the vulnerability pattern rule database.

VulMPFF outperforms two methods that use code sequences IRC, VulDeepecker and Sysevr. VulDeepecker and Sysevr outperform two static vulnerability detection systems, Rats and FlawFinder. Their Pre, Rec, and F1 score on all datasets outperform Rats and FlawFinder. Using code sequence as IRC combined with deep learning techniques can extract the sequential and contextual information of the code. Still, it cannot effectively extract the critical semantic information of the vulnerability triggers in the code and has limited ability to extract lexical and syntactic information.

The two methods, Devign and Reveal, that use the code graph structure as IRC, are superior to the other four methods. Their F1 score exceeds the other methods, with the Reveal method outperforming the Devign method, but VulMPFF equally outperforms both methods. We observe that VulMPFF's Acc and F1 score achieve an absolute improvement from 4.12% to 9.05% and 9.2% to 16.38% on the four datasets compared to the two methods, respectively, which is attributed to the fact that VulMPFF fuses the features in different perspectives and realizes the complementarity of the various code information. Also, introducing a dual attention mechanism helps the model pay more attention to the critical information of vulnerability triggering.

RQ2: Does fusing features in multiple views improve vulnerability detection performance?

To answer this question, we subtract the IRCs involved in fusion in multiple views and perform experiments to compare the performance changes on each of the four datasets. Meanwhile, in order to further investigate the different roles of each subgraph in the CPG feature fusion process, we subtracted each of the four subgraphs and compared their performance changes. In order to verify the advantages of serialized AST compared to code sequences in extracting code information, we also conducted experiments on code sequences as well as code sequences fused with CPG features, which are denoted as SEQ and SEQ + CPG, respectively. The experimental results are shown in Table 5. To highlight the experimental results, we counted the percentage reduction of the F1 score after removing a specific IRC in Table 5. The results are shown in Table 6. From the experimental results, we have the following observations.

When CPG is removed, at which point only serialized ASTs are used for vulnerability detection, all four evaluation

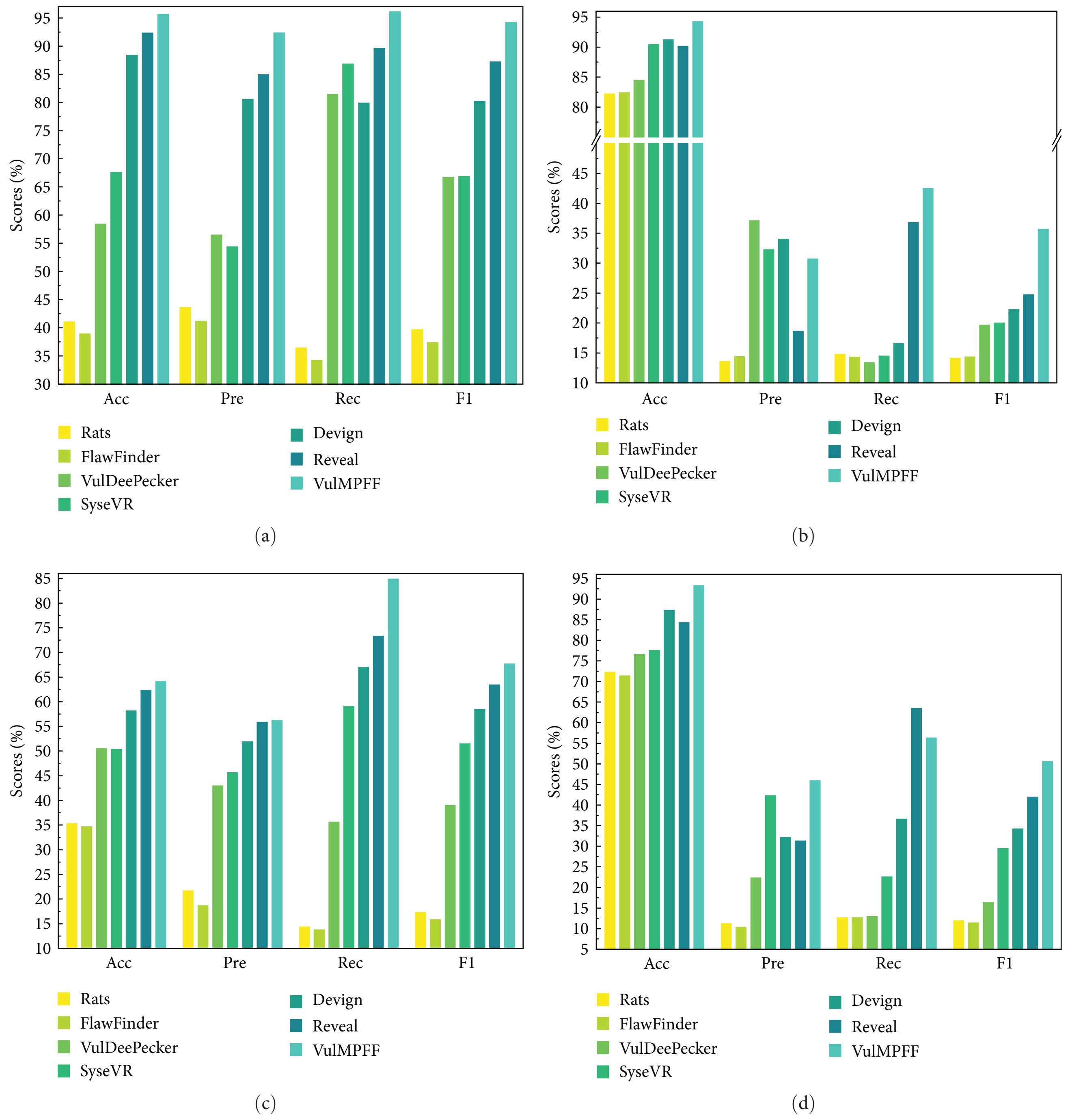


Figure 4: Performance of different detection methods on multiple datasets: (a) on Big-Vul$_{sub}$ dataset, (b) on Big-Vul$_{full}$ dataset, (c) on Devign dataset, and (d) on Reveal dataset.

metrics show substantial decreases across the datasets. Specifically, on the Big-Vul$_{sub}$ dataset the F1 score decreased by 24.17%, Pre even more by 33.92%, and Acc and Rec decreased by 25.07% and 8.76%, respectively; on the Big-Vul$_{full}$ dataset the F1 score decreased by 13.67%, and its Acc, Pre, and Rec, respectively, decreased by 4.54%, 2.31%, and 24.60%; on the Devign dataset Acc was 53.26%, a decrease of 10.87%, and Pre, Rec, and F1 score decreased by 7.4%, 22.68%, and 12.94%, respectively; on the Reveal dataset F1 score decreased by 15.32%, and Acc, Pre, and Rec decreased by 9.81%, 4.48%, and 25.61%, respectively. After removing CPG, the model cannot extract the complex flow semantic information critical for vulnerability triggering, and the detection performance decreases dramatically, indicating that the semantic information in the code contributes a lot to vulnerability detection.

When the serialized AST is removed, at which point only the CPG is used for vulnerability detection, the four evaluation metrics show a slight decrease in magnitude across the datasets, with very few values showing an increase. Specifically, in the Big-Vul$_{sub}$ dataset, Big-Vul$_{full}$ dataset, Devign dataset, and Reveal

Table 5: Results of multifeature fusion ablation experiments (%), I_{rem} indicates removal of a IRC.

	Big-Vul$_{sub}$				Big-Vul$_{full}$				Devign				Reveal			
	Acc	Pre	Rec	F1	Acc	Pre	Rec	F1	Acc	Pre	Rec	F1	Acc	Pre	Rec	F1
SEQ	65.62	59.44	84.71	69.86	90.78	34.52	14.44	20.36	51.83	46.49	60.75	52.67	81.26	42.71	25.68	32.07
CPG$_{rem}$	70.57	58.42	87.34	70.01	89.72	28.38	17.85	21.95	53.26	48.83	62.15	54.69	83.48	41.44	30.67	35.25
SAST$_{rem}$	93.22	92.28	95.82	94.02	91.14	29.17	40.34	33.86	63.28	**56.31**	81.02	66.44	93.06	43.89	56.15	49.27
AST$_{rem}$	91.11	91.15	92.24	91.69	90.03	21.62	32.53	25.98	61.63	52.24	76.22	61.96	88.03	33.28	53.57	41.05
CFG$_{rem}$	92.37	**92.35**	93.13	92.74	90.35	27.76	30.28	28.97	61.34	52.19	79.53	63.06	90.22	35.46	54.68	44.72
DDG$_{rem}$	93.21	92.16	95.65	93.87	91.2	23.35	35.08	28.04	62.34	52.18	80.8	63.5	91.37	38.98	56.27	46.06
CDG$_{rem}$	94.24	92.38	95.5	93.91	92.22	23.33	34.99	27.99	62.39	52.26	80.91	63.41	92.35	38.9	**56.33**	46.02
SEQ + CPG	93.39	92.62	95.69	94.13	92.84	30.17	40.68	34.65	63.27	56.21	81.45	66.52	93.04	44.83	56.09	49.83
VulMPFF	**95.64**	92.34	**96.1**	**94.18**	**94.26**	**30.69**	**42.45**	**35.62**	**64.13**	56.23	**84.83**	**67.63**	**93.29**	**45.92**	56.28	**50.57**

The bolded values in the table indicate the best result values obtained.

Table 6: Multifeature fusion ablation experiment F1 score reduction value statistics (%), I_{rem} indicates removal of a representation.

	Big-Vul$_{sub}$	Big-Vul$_{full}$	Devign	Reveal
CPG$_{rem}$	24.17	13.67	12.94	15.32
SAST$_{rem}$	0.16	1.76	1.19	1.30
AST$_{rem}$	2.49	9.64	5.67	9.52
CFG$_{rem}$	1.44	6.65	4.57	5.85
DDG$_{rem}$	0.31	7.58	4.13	4.51
CDG$_{rem}$	0.27	7.63	4.22	4.55

dataset their F1 score decreased by 0.16%, 1.76%, 1.19%, and 1.30%, respectively, and in the Devign dataset Pre increased slightly by 0.08%, presumably due to experimental perturbations. After removing the serialized AST, the model cannot capture the critical contextual information of vulnerability triggers in the code, and its ability to extract lexical and syntactic information is weakened. The performance degradation is smaller than the removal of CPG, indicating that serialized AST makes a smaller contribution than CPG in vulnerability detection.

When removing any of the subgraphs in the CPG, there were varying degrees of decline in the evaluation metrics across the datasets. The impact is most significant when the AST subgraph is removed, with the F1 score decreasing by 2.49%, 9.64%, 5.67%, and 9.52% on the Big-Vul$_{sub}$ dataset, Big-Vul$_{full}$ dataset, Devign dataset, and Reveal dataset, respectively. The three metrics of Acc, Pre, and Rec are also decreasing the most on each dataset compared to the removal of other subgraphs also all decreased the most. This is because the CPG is expanded based on AST, and more code information is lost when AST is removed. The F1 score decreases by 1.44%, 6.65%, 4.57%, and 5.85% when CFG is removed, 0.31%, 7.58%, 4.13%, and 4.51% when DDG is removed, and 0.27%, 7.63%, 4.22%, and 4.55% when CDG is removed. The experimental results fully illustrate that different subgraphs contain different vulnerabilities triggering critical semantic information. The extraction of code information on IRC of graph structure needs to fully consider the heterogeneity of graph structure and extract the code semantic information at a fine-grained level.

Serialized AST has a significant improvement in detection performance compared to code sequences. To highlight

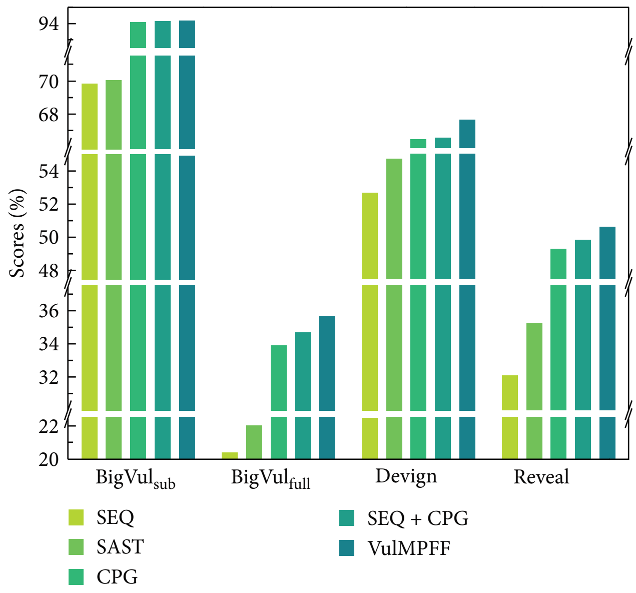


Figure 5: F1 score of SEQ, SAST, CPG, SEQ + CPG, and VulMPFF on the four datasets.

the experimental results, we counted the F1 scores of SEQ, SAST (corresponding to the CPG$_{rem}$ term), CPG (corresponding to the SAST$_{rem}$ term), SEQ + CPG, and VulMPFF on the four datasets in Table 5, and the results are shown in Figure 5. It can be observed that the F1 scores of SAST are better than SEQ on all four datasets, which is because the serialized AST retains the lexical and syntactic information in the code sequences while also including the syntactic information in the code as well as the remote dependencies between the code elements; the F1 scores of SEQ + CPG are better than those of CPG, which indicates that fusing the code features in a dual perspective extracts more code than single feature information; the F1 score of VulMPFF outperforms SEQ + CPG, indicating that fused code features in multiple views perform better than fused code features in dual views, which further illustrates that serialized ASTs contain more code information than code sequences.

RQ3: What is the impact of different feature extraction methods on model performance?

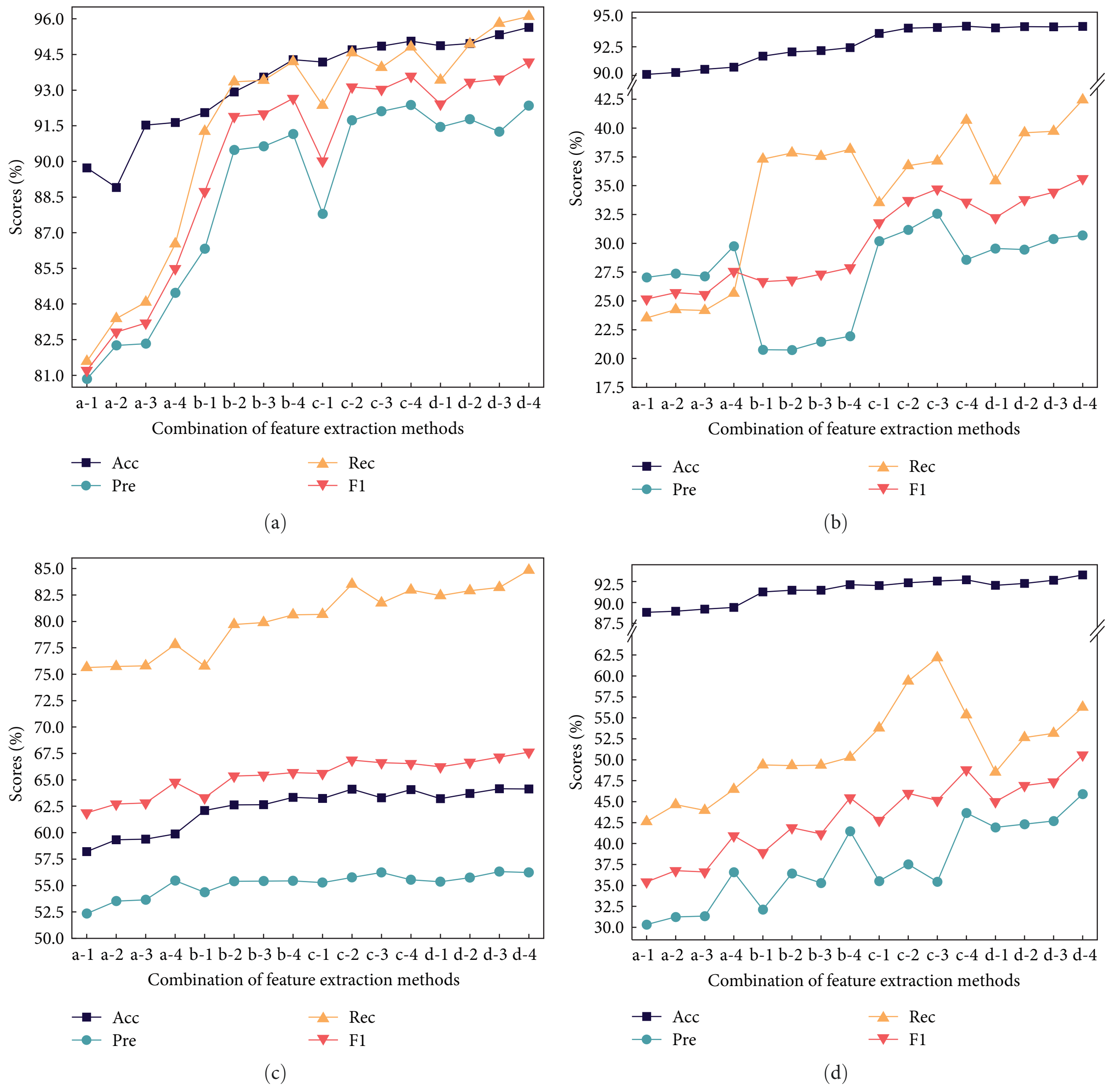


FIGURE 6: Performance of 16 feature extraction combination schemes on four datasets: (a) on Big-Vul$_{sub}$ dataset, (b) on Big-Vul$_{full}$ dataset, (c) on Devign dataset, and (d) on Reveal dataset.

To answer this question, we selected four networks on serialized AST and CPG feature extraction, respectively, and combined them two by two to obtain 16 feature extraction schemes and shared the same experimental configurations for the experiments. The experimental results are shown in Figure 6.

The networks corresponding to serialized AST feature extraction are LSTM [36], TextCNN [37], Bi-LSTM [38], and Bi-LSTM with attention mechanism, denoted as LSTM:1, TextCNN:2, Bi-LSTM:3, and Bi-LSTM + Attention:4, respectively. The corresponding networks for CPG feature extraction are GCN [39], GIN [40], GGNN [41], and GAT [42], denoted as GCN: a, GIN: b, GGNN: c, and GAT: d, respectively. The 16 feature extraction schemes are denoted as (a–d)–(1–4). For example, a-1 represents the scheme of the GCN model combined with the LSTM model. Based on the experimental results, we have the following observations.

Among the 16 feature extraction combination schemes, our scheme, i.e., the d-4 scheme, achieves the best results on several datasets. In contrast, the a-1 scheme performs poorly on several datasets. The d-4 scheme achieved the optimal F1 score on the Big-Vul$_{sub}$, Big-Vul$_{full}$, Devign, and Reveal datasets, which were 94.18%, 35.62%, 67.63%, and 50.57%, respectively. The a-1 scheme had the lowest F1 score on several datasets, which were 81.21%, 25.16%, 61.87%, and 35.43%. Overall, extracting CPG features using the GAT

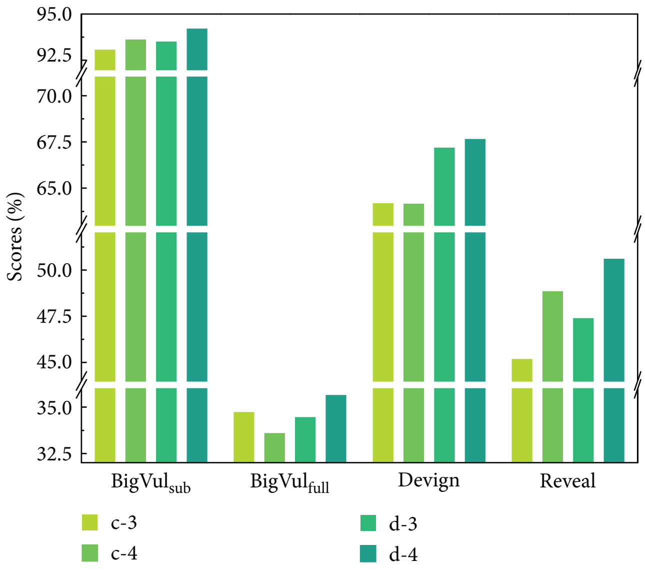


Figure 7: F1 score of four feature extraction combination schemes introducing attention mechanism on four datasets.

model outperforms the other GNN models. The scheme using the GGNN model outperforms the scheme using the GIN model, and using the GCN model has the worst performance among the four GNN models. The scheme using the Bi-LSTM model with attention mechanism outperforms the others on serialized AST, the scheme using the LSTM model has the worst performance among several schemes, and the schemes using the TextCNN model or Bi-LSTM model are close in performance.

To verify whether the introduction of the dual-attention mechanism facilitates the model to pay more attention to the critical information of vulnerability triggering in the code, we, respectively, counted the F1 score performance results of the four scenarios c-3, c-4, d-3, and d-4 in Figure 6, as shown in Figure 7. From the results, we can see that the introduction of the dual-attention mechanism achieves the best performance on each dataset, and the introduction of the dual-attention mechanism facilitates the model to pay more attention to the critical information of vulnerability triggering and improves the model vulnerability detection performance. When only one attention mechanism is introduced in the feature extraction process, introducing an attention mechanism in CPG feature extraction has more significant and stable performance improvement. If the attention mechanism is introduced only in the feature extraction of the serialized AST, a performance degradation of tiny magnitude occurs on the bigfull and Devign datasets, presumably due to the perturbation of noise in the experiments.

RQ4: How effective does VulMPFF perform in real-world applications?

To answer this question, we further evaluate our approach in real-world open-source projects. We select five popular open-source projects: Linux, Libav, Qemu, OpenSSL, and FFmpeg.

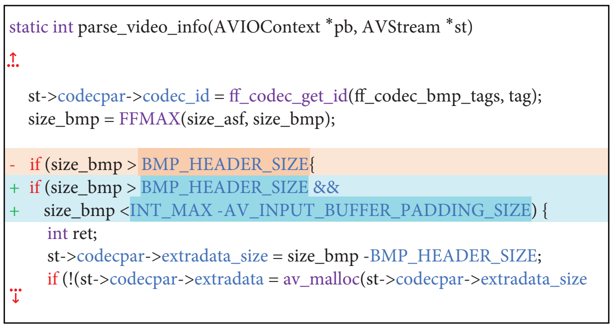
Precisely, we extract function samples on the chosen version of the open-source project and keep the samples that can be extracted using Joern [32] for AST and CPG. The extracted function samples are subjected to the same data preprocessing as the experimental dataset, fed into the VulMPFF pretrained model that has already completed training on the experimental dataset, and the detection results are collected. We manually compare the detected vulnerability functions with the real vulnerabilities we collected. If they belong to the same vulnerability pattern, the detected vulnerability functions are categorized as real vulnerability functions. Some of the vulnerabilities detected by VulMPFF in five open-source projects are shown in Table 7. These results demonstrate that VulMPFF can detect vulnerabilities in real environments.

For example, CVE-2018-1999011 is a buffer overflow vulnerability in the FFmpeg project in the *asf_o* format decoder. This vulnerability can lead to remote code execution due to heap buffer overflow. We extracted the code of the *parse_video_info*() function where the vulnerability is located in the *libavf_ormat/asf_dec_o.c* file of the FFmpeg-n6.0 version project, where the lack of necessary boundary determination for the *size_bmp* parameter may trigger the buffer overflow vulnerability. The schematic diagram for fixing the vulnerability in this function is shown in Figure 8(a). Similarly, we have detected a CVE-2023-28772 vulnerability in the Linux-4.20-rc6 release project. The vulnerability is located in the seq_buf_putmem_hex() function in the *lib/seq_buf.c* file, where variable *i* and variable *j* follow the original data stream and the data stream written to the buffer, respectively, in a *for* loop, which triggers a buffer overflow vulnerability when *j* is greater than *HEX_CHARS*. The schematic for fixing this function vulnerability is shown in Figure 8(b). Vulnerabilities such as CVE-2018-999011 and CVE-2023-28772 are closely related to semantic information about data flow and control flow in code. VulMPFF can effectively detect similar vulnerabilities by extracting the function code into CPGs containing rich data flow and control flow semantic information as IRCs and extracting the critical semantic information triggering the vulnerability at a fine granularity in different subgraphs.

We have detected a CVE-2020-35965 vulnerability in the FFmpeg-n6.0 release project. The vulnerability is located in the *decode_frame*() function in the *libavcodec/exr.c* file. An out-of-bounds write error results when performing a memset operation in a multiple *for* loop. A schematic for fixing this vulnerability is shown in Figure 8(c). Similarly, we have detected the CVE-2023-0401 vulnerability in the OpenSSL-3.0.4 release project. The vulnerability is in the *pkcs7_bio_add_digest*() function in the *crypto/pkcs7/pk7_doit.c* file, and is a classic null pointer dereference vulnerability. The vulnerability is caused by the lack of necessary checks on the return value of the initialization function, as Figure 8(d) shows a schematic diagram for fixing the vulnerability in this function. Detecting vulnerabilities like CVE-2020-35965 and CVE-2023-0401 requires models that can extract contextual information related to vulnerability triggers, data flow and control flow information in code, and a strong ability to capture lexical and syntactic information. VulMPFF extracts serialized ASTs and CPGs as IRCs in sequence, lexical and syntactic, and graph structure perspectives,

Table 7: Some of the vulnerabilities detected by VulMPFF in the open-source project.

Project	CVEID	Vulnerable file in the target product	Release date	Vulnerability type
Linux-4.20-rc6	CVE-2021-43975	drivers/net/ethernet/aquantia/atlantic/hw atl/hw atl utils.c	2021-11-17	CWE-787
	CVE-2023-28772	lib/seq buf.c	2023-03-23	CWE-120
Libav-12.3	CVE-2018-19128	libavcodec/lcldec.c	2018-11-09	CWE-125
	CVE-2019-14443	libavcodec/apedec.c	2019-07-30	CWE-369
Qemu-4.2.0	CVE-2021-20203	hw/net/vmxnet3.c	2021-02-25	CWE-190
	CVE-2021-3507	hw/block/fdc.c	2021-05-06	CWE-119 CWE-787
OpenSSL-3.0.4	CVE-2023-0401	crypto/pkcs7/pk7_doit.c	2023-02-08	CWE-476
	CVE-2022-2274	crypto/bn/rsaz_exp_x2.c	2022-07-01	CWE-787
FFmpeg-n6.0	CVE-2020-35965	libavcodec/exr.c	2021-01-04	CWE-787
	CVE-2018-1999011	libavformat/asfdec_o.c	2018-07-23	CWE-119

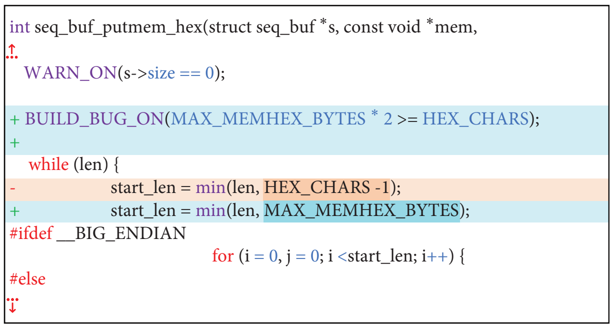


```
static int parse_video_info(AVIOContext *pb, AVStream *st)

    st->codecpar->codec_id = ff_codec_get_id(ff_codec_bmp_tags, tag);
    size_bmp = FFMAX(size_asf, size_bmp);

-   if (size_bmp > BMP_HEADER_SIZE{
+   if (size_bmp > BMP_HEADER_SIZE &&
+       size_bmp <INT_MAX -AV_INPUT_BUFFER_PADDING_SIZE) {
        int ret;
        st->codecpar->extradata_size = size_bmp -BMP_HEADER_SIZE;
        if (!(st->codecpar->extradata = av_malloc(st->codecpar->extradata_size
```

(a)

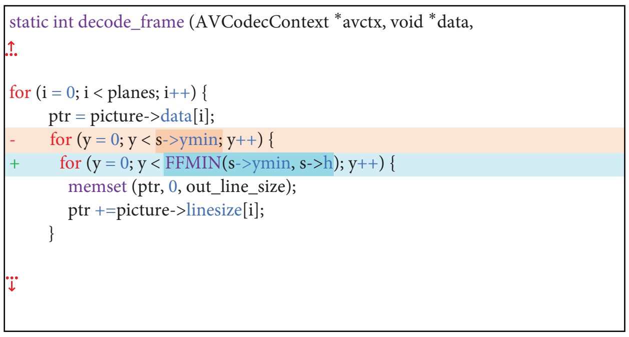


```
int seq_buf_putmem_hex(struct seq_buf *s, const void *mem,

  WARN_ON(s->size == 0);

+ BUILD_BUG_ON(MAX_MEMHEX_BYTES * 2 >= HEX_CHARS);
+
  while (len) {
-               start_len = min(len, HEX_CHARS -1);
+               start_len = min(len, MAX_MEMHEX_BYTES);
#ifdef __BIG_ENDIAN
                                for (i = 0, j = 0; i <start_len; i++) {
#else
```

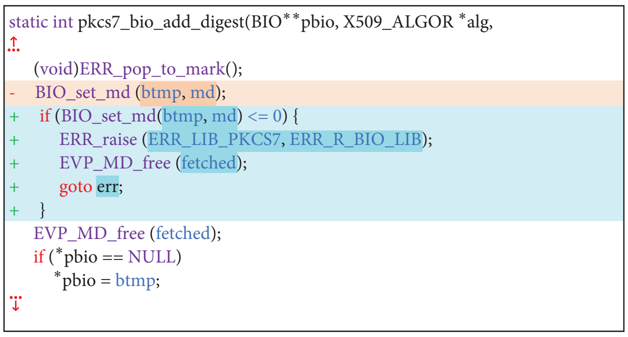
(b)



```
static int decode_frame (AVCodecContext *avctx, void *data,

for (i = 0; i < planes; i++) {
       ptr = picture->data[i];
-      for (y = 0; y < s->ymin; y++) {
+        for (y = 0; y < FFMIN(s->ymin, s->h); y++) {
          memset (ptr, 0, out_line_size);
          ptr +=picture->linesize[i];
       }
```

(c)



```
static int pkcs7_bio_add_digest(BIO**pbio, X509_ALGOR *alg,

    (void)ERR_pop_to_mark();
-   BIO_set_md (btmp, md);
+    if (BIO_set_md(btmp, md) <= 0) {
+       ERR_raise (ERR_LIB_PKCS7, ERR_R_BIO_LIB);
+       EVP_MD_free (fetched);
+       goto err;
+    }
    EVP_MD_free (fetched);
    if (*pbio == NULL)
       *pbio = btmp;
```

(d)

Figure 8: Function fix patch code: (a) CVE-2018-1999011, (b) CVE-2023-28772, (c) CVE-2020-35965, and (d) CVE-2023-0401.

and the fusion of features extracted from multiple perspectives can effectively capture multiple code information related to vulnerability triggering and has better detection capability for similar vulnerabilities.

6. Discussion and Limitations

Our proposed VulMPFF vulnerability detection method still has some limitations that can be further investigated. The first limitation concerns the datasets used to validate our approach. We validated our method on three public datasets: Big-Vul [35], Devign [15], and Reveal [16]. Although they already contain a large amount of high-quality function code, these function codes containing vulnerabilities will gradually become obsolete as computer technology continues to evolve. In the future, we will use more high-quality datasets to evaluate our algorithm further.

The second limitation is that our approach only supports code vulnerability detection at function-level granularity. While function-level granularity is a common detection granularity for most vulnerability detection methods, finer-grained vulnerability detection, such as line-of-code vulnerability detection, is more likely to be applied in practice. In the future, we will further explore the possibility of extending our approach to line-of-code granularity.

The third limitation is that our approach is specific to C code, our data processing process is not yet extensible, and the datasets we chose are all C code datasets. In the future, we will further explore extending our algorithm to more programing language types (e.g., Java, JavaScript, Python, Go, PHP, etc.).

7. Conclusions

Aiming at the problems of existing vulnerability detection methods, a vulnerability detection method, VulMPFF, that fuses code features under multiple perspectives is proposed. This method extracts serialized ASTs and CPGs as IRCs to capture code information under various perspectives, which can effectively realize the complementary information of multiple IRCs. The dual-attention mechanism is introduced in feature extraction to highlight the critical information related to vulnerability triggering, the heterogeneous nature of the code relationship graph is considered on CPG, and the regional subgraph extracts the semantic information of different complex flows. The fused features in multiple perspectives contain critical context, lexical and syntactic, and different complex flow semantic information of vulnerability triggers. Compared with other state-of-the-art vulnerability detection methods, VulMPFF significantly improves the F1 score on multiple datasets from 152.08% to 344.77%. Experiments in open-source projects show that the algorithm can detect vulnerabilities in real environments.

Data Availability

The datasets used in this paper are public, free, and available at Big-Vul (https://github.com/ZeoVan/MSR_20_Code_vulnerability_CSV_Dataset), Reveal (https://drive.google.com/drive/folders/1KuIYgFcvWUXheDhT–cBALsfy1I4utOy), and Devign (https://drive.google.com/file/d/1x6hoF7G-tSYxg8AFybggypLZgMGDNHfF/edit).

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported in part by the National Key Research and Development Program under Grant 2022YFB3305203, in part by the National Natural Science Foundation of China under Grants U2133208 and 62101368, and in part by the Sichuan Youth Science and Technology Innovation Team under Grant 2022JDTD0014 and in part by the Major Science and Technology Special Project of Sichuan Province under Grant 2022ZDZX0008.

References

[1] X. Du, B. Chen, Y. Li et al., "Leopard: identifying vulnerable code for vulnerability assessment through program metrics," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 60–71, IEEE, 2019.

[2] W. Niu, X. Zhang, X. Du, L. Zhao, R. Cao, and M. Guizani, "A deep learning based static taint analysis approach for IoT software vulnerability location," *Measurement*, vol. 152, Article ID 107139, 2020.

[3] S. Guo, Y. Chen, P. Li et al., "SpecuSym: speculative symbolic execution for cache timing leak detection," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pp. 1235–1247, ACM, 2020.

[4] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, "Large language models are zero-shot fuzzers: fuzzing deep-learning libraries via large language models," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 423–435, ACM, 2023.

[5] Z. Li, D. Zou, S. Xu et al., "Vuldeepecker: a deep learning-based system for vulnerability detection," arXiv preprint arXiv: 1801.01681, 2018.

[6] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, "μVulDeePecker: a deep learning-based system for multiclass vulnerability detection," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 5, pp. 2224–2236, 2019.

[7] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "SySeVR: a framework for using deep learning to detect software vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 4, pp. 2244–2258, 2021.

[8] Y. Wu, D. Zou, S. Dou, W. Yang, D. Xu, and H. Jin, "VulCNN: an image-inspired scalable vulnerability detection system," in *Proceedings of the 44th International Conference on Software Engineering*, pp. 2365–2376, ACM, 2022.

[9] R. Russell, L. Kim, L. Hamilton et al., "Automated vulnerability detection in source code using deep representation learning," in *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pp. 757–762, IEEE, 2018.

[10] I. Neamtiu, J. S. Foster, and M. Hicks, "Understanding source code evolution using abstract syntax tree matching," in *Proceedings of the 2005 International Workshop on Mining Software Repositories*, pp. 1–5, ACM, 2005.

[11] F. E. Allen, "Control flow analysis," *ACM SIGPLAN Notices*, vol. 5, no. 7, pp. 1–19, 1970.

[12] E. Yourdon and L. L. Constantine, *Structured Design. Fundamentals of a Discipline of Computer Program and Systems Design*, Yourdon Press, Englewood Cliffs, 1979.

[13] H. Nasirloo and F. Azimzadeh, "Semantic code clone detection using abstract memory states and program dependency graphs," in *2018 4th International Conference on Web Research (ICWR)*, pp. 19–27, IEEE, Tehran, Iran, 2018.

[14] W. Xiaomeng, Z. Tao, W. Runpu, X. Wei, and H. Changyu, "CPGVA: code property graph based vulnerability analysis by deep learning," in *2018 10th International Conference on Advanced Infocomm Technology (ICAIT)*, pp. 184–188, IEEE, Stockholm, Sweden, 2018.

[15] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *International Conference on Neural Information Processing Systems*, pp. 10197–10207, Vancouver, BC, Canada, 2019.

[16] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: are we there yet?" *IEEE Transactions on Software Engineering*, vol. 48, no. 9, pp. 3280–3296, 2022.

[17] D. Hin, A. Kan, H. Chen, and M. A. Babar, "LineVD: statement-level vulnerability detection using graph neural networks," in *Proceedings of the 19th International Conference on Mining Software Repositories*, pp. 596–607, ACM, 2022.

[18] X. Cheng, H. Wang, J. Hua, G. Xu, and Y. Sui, "DeepWukong: statically detecting software vulnerabilities using deep graph neural network," *ACM Transactions on Software Engineering and Methodology*, vol. 30, no. 3, pp. 1–33, 2021.

[19] Secure Software Inc., “Rough audit tool for security,” 2023, {https://code.google.com/archive/p/rough-auditing-tool-for-security/}.

[20] D. A. Wheeler, “Flawfinder,” 2023, {https://dwheeler.com/flawfinder/}.

[21] T. Ben-Nun, A. S. Jakobovits, and T. Hoefler, “Neural code comprehension: a learnable representation of code semantics,” *Advances in Neural Information Processing Systems*, vol. 31, pp. 1–13, 2018.

[22] Y. Wu, J. Lu, Y. Zhang, and S. Jin, “Vulnerability detection in C/C++ source code with graph representation learning,” in *2021 IEEE 11th Annual Computing and Communication Workshop and Conference (CCWC)*, pp. 1519–1524, IEEE, NV, USA, 2021.

[23] Cppcheck, “Cppcheck,” 2023, {https://github.com/danmar/cppcheck}.

[24] Checkmarx, “Checkmarx,” 2023, {https://www.checkmarx.com/}.

[25] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, “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, ACM, 2016.

[26] S. Kim, S. Woo, H. Lee, and H. Oh, “Vuddy: a scalable approach for vulnerable code clone discovery,” in *2017 IEEE Symposium on Security and Privacy (SP)*, pp. 595–614, IEEE, San Jose, CA, USA, 2017.

[27] H. Sun, L. Cui, L. Li et al., “VDSimilar: vulnerability detection based on code similarity of vulnerabilities and patches,” *Computers & Security*, vol. 110, Article ID 102417, 2021.

[28] A. Xu, T. Dai, H. Chen, Z. Ming, and W. Li, “Vulnerability detection for source code using contextual LSTM,” in *2018 5th international conference on systems and informatics (ICSAI)*, pp. 1225–1230, IEEE, Nanjing, China, 2018.

[29] H. Feng, X. Fu, H. Sun, H. Wang, and Y. Zhang, “Efficient vulnerability detection based on abstract syntax tree and deep learning,” in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pp. 722–727, IEEE, Toronto, ON, Canada, 2020.

[30] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, “The graph neural network model,” *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2009.

[31] G. Lin, J. Zhang, W. Luo et al., “Cross-project transfer representation learning for vulnerable function discovery,” *IEEE Transactions on Industrial Informatics*, vol. 14, no. 7, pp. 3289–3297, 2018.

[32] Joern, “Open-source code analysis platform for C/C++ based on code property graphs,” 2023, {https://joern.io/}.

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

[34] N. Navarin, D. Van Tran, and A. Sperduti, “Universal readout for graph convolutional neural networks,” in *2019 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–7, IEEE, Budapest, Hungary, 2019.

[35] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, “A C/C++ code vulnerability dataset with code changes and CVE summaries,” in *Proceedings of the 17th International Conference on Mining Software Repositories*, pp. 508–512, ACM, 2020.

[36] K. Greff, R. K. Srivastava, J. Koutnik, B. R. Steunebrink, and J. Schmidhuber, “LSTM: a search space odyssey,” *IEEE Transactions on Neural Networks and Learning Systems*, vol. 28, no. 10, pp. 2222–2232, 2017.

[37] Y. Kim, “Convolutional neural networks for sentence classification,” arXiv preprint arXiv: 1408.5882, 2014.

[38] T. Chen, R. Xu, Y. He, and X. Wang, “Improving sentiment analysis via sentence type classification using BiLSTM-CRF and CNN,” *Expert Systems with Applications*, vol. 72, pp. 221–230, 2017.

[39] T. N. Kipf and M. Welling, “Semi-supervised classification with graph convolutional networks,” arXiv preprint arXiv: 1609.02907, 2016.

[40] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, “How powerful are graph neural networks?” arXiv preprint arXiv: 1810.00826, 2018.

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

[42] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, “Graph attention networks,” arXiv preprint arXiv: 1710.10903, 2017.