

Vulnerability Detection via Multiple-Graph-Based Code Representation

Fangcheng Qiu , Zhongxin Liu , *Member, IEEE*, Xing Hu , Xin Xia , *Member, IEEE*,
Gang Chen , *Member, IEEE*, and Xinyu Wang 

Abstract—During software development and maintenance, vulnerability detection is an essential part of software quality assurance. Even though many program-analysis-based and machine-learning-based approaches have been proposed to automatically detect vulnerabilities, they rely on explicit rules or patterns defined by security experts and suffer from either high false positives or high false negatives. Recently, an increasing number of studies leverage deep learning techniques, especially Graph Neural Network (GNN), to detect vulnerabilities. These approaches leverage program analysis to represent the program semantics as graphs and perform graph analysis to detect vulnerabilities. However, they suffer from two main problems: (i) Existing GNN-based techniques do not effectively learn the structural and semantic features from source code for vulnerability detection. (ii) These approaches tend to ignore fine-grained information in source code. To tackle these problems, in this paper, we propose a novel vulnerability detection approach, named MGVD (MULTIPLE-GRAPH-BASED VULNERABILITY DETECTION), to detect vulnerable functions. To effectively learn the structural and semantic features from source code, MGVD uses three different ways to represent each function into multiple forms, i.e., two statement graphs and a sequence of tokens. Then we encode such representations to a three-channel feature matrix. The feature matrix contains the structural feature and the semantic feature of the function. And we add a weight allocation layer to distribute the weights between structural and semantic features. To overcome the second problem, MGVD constructs each graph representation of the input function using multiple different graphs instead of a single graph. Each graph focuses on one statement in the function and its nodes denote the related statements and their fine-grained code elements. Finally, MGVD leverages CNN to identify whether this function is vulnerable based on such feature matrix. We conduct experiments on 3 vulnerability datasets with a total of 30,341 vulnerable functions and 127,931 non-vulnerable functions. The experimental results show

that our method outperforms the state-of-the-art by 9.68% – 10.28% in terms of F1-score.

Index Terms—Vulnerability detection, deep learning, code representation, graph neural network.

I. INTRODUCTION

SOFTWARE vulnerabilities are a crucial reason for the prevalence of cyber attacks and have caused substantial damage to society's software infrastructures [1]. Vulnerability detection has long been an important problem in software security. Nevertheless, it is time-consuming and labor-intensive to manually detect vulnerabilities. To help developers reduce manual efforts regarding the software debugging process, program-analysis-based [2], [3], [4] and machine-learning-based [5], [6] approaches have been proposed for automatically detecting vulnerabilities. Program-analysis-based tools often rely on security domain experts to define vulnerability rules and patterns. To better capture the vulnerability features, traditional Machine Learning (ML) has been used to automatically learn the patterns of vulnerabilities from prior vulnerable code. However, due to the limited capabilities of traditional machine learning models, these techniques did not achieve satisfactory results. With the widespread application of Deep Learning (DL), a growing number of studies have leveraged deep learning to detect software vulnerabilities [7], [8], [9], [10], [11]. They differ from each other in how to represent source code and extract vulnerability-related features. For example, Li et al. [7] represent the source code as text and use natural language processing techniques to capture features. But these text-based methods do not achieve satisfactory results since they mainly rely on the text of source code and suffer from capturing the features related to the structure and semantics of source code.

To address the issue, researchers conduct program analysis to distill the semantics of source code into a graph representation, such as Abstract Syntax Tree (AST), Control Flow Graph (CFG) and Program Dependence Graph (PDG), then perform graph analysis to detect vulnerabilities. At present, Graph Neural Network (GNN) have achieved great success in the field of analyzing graph-structured data, such as social networks [12] and traffic networks [13]. Unlike traditional neural networks, which are designed for grid-like data such as images or sequences, GNN is a type of neural network designed specifically for graph structured data. Inspired by this and the graph representations of source code, researchers also adopt GNN to

Manuscript received 1 October 2023; revised 28 June 2024; accepted 9 July 2024. Date of publication 12 July 2024; date of current version 15 August 2024. This work was supported in part by the National Natural Science Foundation of China under Grant 62202420 and in part by Zhejiang Province “JianBingLingYan+X” Research and Development Plan under Grant 2024C01114. Recommended for acceptance by L. Pasquale. (*Corresponding authors: Zhongxin Liu; Xinyu Wang.*)

Fangcheng Qiu and Zhongxin Liu are with the State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, Zhejiang 310027, China (e-mail: fangchengq@zju.edu.cn; liu_zx@zju.edu.cn).

Xing Hu is with the School of Software Technology, Zhejiang University, Ningbo, Zhejiang 315103, China (e-mail: xinghu@zju.edu.cn).

Xin Xia is with the Software Engineering Application Technology Lab, Huawei, Hangzhou, Zhejiang 310051, China (e-mail: xin.xia@acm.org).

Gang Chen and Xinyu Wang are with the College of Computer Science and Technology, Zhejiang University, Hangzhou, Zhejiang 310027, China (e-mail: cg@zju.edu.cn; wangxinyu@zju.edu.cn).

Digital Object Identifier 10.1109/TSE.2024.3427815

develop effective tools for software vulnerability detection [14], [15], [16], [17].

These GNN-based techniques are more effective than text-based methods, but still suffer from two main problems: (i) Existing GNN-based techniques, e.g., Devign, REVEAL, IVDETECT and FUNDED, do not **effectively** capture the structural and semantic features of source code for vulnerability detection [14], [15], [16], [17]. Some vulnerabilities are solely related to the structural information of the code, such as NULL Pointer Dereference, while others require consideration of both the structural and semantic information of the code, such as Integer Overflow. Some methods, such as Devign, REVEAL and FUNDED, only consider the structural information of source code and overlook semantic information. For instance, Devign and REVEAL abstract the user-defined variables and function names in the given code snippet and represent it as the Code Property Graph of its abstract version. Therefore, these GNN-based approaches face challenges in detecting the second type of vulnerabilities. Other methods, such as IVDETECT, consider both the structural and semantic information of source code but only integrate these two types of information into one graph-based representation. When dealing with the first type of vulnerabilities, they may be misled by irrelevant semantic information in the code [18], [19]. (ii) Existing GNN-based approaches, e.g., Devign, REVEAL, IVDETECT and FUNDED, tend to ignore fine-grained information (such as the data type of a specific variable) in source code. Specifically, they represent a function as a single graph where each node denotes a statement, ignoring the fine-grained information within statements [20], [21]. Although some methods, e.g., Devign, REVEAL and FUNDED, incorporate AST nodes and edges into the graph to mitigate this problem, their constructed graphs become large and complex. Prior work has shown that it is easy for GNN to lose some detailed information when processing large graphs [22], [23]. Therefore, existing GNN-based approaches face challenges in detecting vulnerabilities related to some fine-grained information.

In this study, we propose a novel GNN-based vulnerability detection approach, named MGVD (**M**ULTIPLE-**G**RAPH-BASED **V**ULNERABILITY **D**ETECTION), to tackle the problems mentioned above. For the first problem, MGVD represents each input function in three different forms, *raw text*, *abstract graph* and *raw graph*. Raw text refers to the text of the input function and can help capture semantic information. An abstract graph is constructed from the abstract version of a function (referred to as *abstract function* by us) and can ease the capture of structural information. The raw graph is constructed from the raw version of a function and it encodes both the structure and the text of the function, providing a more comprehensive perspective. We collectively refer to the raw graph and abstract graph as *function graph*. This design provides three different perspectives, i.e., semantic information alone, structural information alone, and a combination of semantic and structural information, which can help address the issue of neglecting semantic information and alleviate the problem of deep learning models being easily influenced by irrelevant semantic information. In addition, MGVD leverages the Convolution Neural Network (CNN) model and

Squeeze-and-Excitation Network (SENet) layer to adaptively fuse the features learned from the three forms. In this way, MGVD can effectively utilize both the structural and semantic features of source code. To tackle the second problem, MGVD constructs the function graph by combining the PDG and the AST of the function. This allows MGVD to not only consider the relationship between statements, such as data dependency and control dependency, from the PDG, but also to capture the internal relationships within statements, such as syntactic dependencies from the AST. Next, to ease the capture of fine-grained information, MGVD converts every function graph, which is usually large and complex, into multiple small sub-graphs to represent the function. Each sub-graph focuses on one statement in the function and contains the graph nodes and edges related to the statement and its context. We refer to such a sub-graph as a *statement graph*. GNN is used to process each statement graph separately. In this way, the size of the graph input into the GNN is greatly reduced, which can reduce the detailed information being overlooked by the GNN and help MGVD to better capture vulnerability-related fine-grained information.

MGVD consists of three stages: function representation, feature matrix generation, and classification. In the function representation stage, MGVD represents the function into three different forms. In the feature matrix generation stage, MGVD fuses the three representation forms to construct the feature matrix. In the classification part, MGVD determines whether the input function contains vulnerabilities based on the feature matrix.

To demonstrate the effectiveness of our approach, we evaluate MGVD on three vulnerability datasets, i.e., SARD [24], FFmpeg+QEMU [16], Big-Vul [25]. We measure the performance of MGVD using accuracy, precision, recall, F1-score and MCC. The experimental results show that MGVD outperforms seven state-of-the-art vulnerability detection tools by substantial margins. The materials of this work are released at Zenodo [26].

In summary, this paper makes the following contributions:

- We propose a multi-dimensional code representation, which represents a function as two sequences of small graphs and one sequence of texts.
- We propose a novel function-level vulnerability detection model based on our proposed code representation, which first encodes the input function as a three-dimensional feature matrix and then leverages a CNN model to perform detection.
- We perform extensive experiments to evaluate MGVD and analyze the effectiveness of MGVD across different vulnerability types and different code complexities. Our results demonstrate the superiority of MGVD over the state-of-the-art vulnerability detection tools.

The organization of this paper is as follows. Section II describes the motivation of MGVD. Section III describes the background of this work. Section IV introduces the detailed design of our approach. Section V describes our experiment setup. Section VI reports the experimental results. Section VII discusses further experiments. Section VIII shows threats to

```

1 void dostor(char *name, const int append, const int autorename)
2 {
3     ...
4     off_t max_filesize = (off_t) -1;
5     ...
6     #ifdef QUOTAS
7     if (quota_update(&quota, 0LL, 0LL, &overflow) == 0 &&
8         (overflow > 0 || quota.files >= user_quota_files || quota.size > user_quota_size ||
9         (max_filesize >= (off_t) 0 &&
10          (max_filesize = user_quota_size - quota.size) < (off_t) 0))) {
11         overflow = 1;
12         (void) close(f);
13         goto afterquota;
14     }
15     #endif
16     ...
17 }

```

Fig. 1. A vulnerable function in pure-ftpd exposed by CVE-2021-40524 (most of which are not presented).

validity. Section XI describes the related work. We conclude the paper and discuss the future work in Section X.

II. MOTIVATION

Fig. 1 shows a function collected from the pure-ftpd project¹. It contains the vulnerability exposed by CVE-2021-40524. In detail, in line 4, the initial value of `max_filesize` is set to `-1`. In line 7, `max_filesize >= (off_t) 0` evaluates to false, and the following condition `(max_filesize = user_quota_size - quota.size) < (off_t) 0` will not be evaluated. Hence `max_filesize = -1` will always be `-1`. This leads to the problem that there is no overflow even if the quota is exceeded.

From this example, we have the following observations: (i) Both the structural and semantic information are helpful for understanding this vulnerability. For example, if we replace `max_filesize`, `user_quota_size` and `quota` with `VAR1`, `VAR2` and `VAR3`, it would not be easy to understand whether initializing `VAR1` as `-1` is proper. (ii) This vulnerability is only related to a few statements, i.e., line 4 and line 7, but the function contains a large number of statements (109 lines, most of which are not presented). (iii) This vulnerability is related to fine-grained code elements, e.g., the two clauses circled in red in line 7. To detect the vulnerability in this function, some existing GNN-based approaches do not effectively leverage the structural and semantic features of source code, which may lose the semantic information in these identifiers and prevent vulnerability detection. In addition, existing GNN-based approaches usually represent the function as one large single function graph, leading to a potential oversight of the fine-grained information. For example, line 7 in Fig. 1 contains multiple clauses and complex conditions. The internal computations in this statement constitute only a small part of the function graph, and tend to be overlooked by existing approaches. Thus, it is difficult for existing approaches to detect this vulnerable function.

The first observation mentioned above motivates us to effectively leverage both the structural and semantic information. Therefore, given an input function, our approach first uses

three different ways to represent it to better capture diverse information in it. Then, our approach adopts the SENet layer to learn to adaptively assign weights to three representations, so that we can effectively leverage both structural and semantic features of source code. The last two observations motivate us to better identify and focus on fine-grained vulnerability-related information in source code. To this end, our approach first combines PDG with AST to construct the graph representations, i.e., the raw graph and the abstract graph mentioned in Section I, of this function to explicitly incorporate fine-grained code elements into the graphs. Then, we propose to extract statement graphs for each statement from the raw graph and the abstract graph, respectively. We use two sequences of statement graphs instead of the raw graph and the abstract graph to represent the function. In this way, the input of GNN becomes a sub-graph, and the number of nodes and edges that GNN needs to process greatly reduces, making it less challenging for GNN to extract vulnerability-related features from fine-grained code elements. In our used datasets (c.f. Section V-A), the mean numbers of nodes and edges in a function graph are 145.5 and 198.7, respectively, with a median of 124 and 149, respectively. In contrast, the mean numbers of nodes and edges in a sub-graph are 27.8 and 30.2, respectively, with a median of 24 and 26, respectively. Note that, the sequence of sub-graphs keeps all the information in the function. Thus, our approach also has the ability to extract vulnerability-related features across multiple statements. Based on these techniques, our approach can identify that the initial value of `max_filesize` is `-1` and can better understand the complex conditions within the statement at line 7. As a result, it successfully detects the vulnerability in Fig. 1.

III. BACKGROUND

Our approach represents source code as graphs and leverages Graph Neural Network to encode source code. In this section, we introduce the background knowledge of the related techniques.

A. Graph Representations of Source Code

A code snippet can be represented to an **Abstract Syntax Tree (AST)**. Each node in an AST denotes a construct occurring in the source code. **Control Flow Graph (CFG)** is a graph representation of source code where all paths might be traversed through a program during the execution of the code. In a CFG, each node represents a basic block, and each path represents a jump in the control flow. **Data Flow Graph (DFG)** is a graph that tracks the usage of variables throughout the program. Its nodes are places where variables are assigned or used, and its paths show the relationship between these places. **Program Dependency Graph (PDG)** is a graph representation that makes data dependencies and control dependencies explicit. In a PDG, each node represent program statement, and each edge represent dependencies between these statements.

Given a function, our approach first combines its PDG and AST to construct a function graph. We use PDG because it contains the information of both the control flow and the data flow. Leveraging AST can add more fine-grained code elements

¹<https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-40524>

to the function graph. We leverage Joern [27] to extract the AST and the PDG of each function.

B. Source Code Embedding

To process the constructed graph of each function using GNN, we need to transform each node into a feature vector. Prior studies have utilized text embedding methods like Doc2Vec [28], or word embedding methods such as GloVe [29], Word2Vec [30] and fasttext [31] to generate pre-trained vectors for singular tokens. These embedding methods obtain vector representations of words by mapping words into a meaningful vector space where the distance between two words is related to their semantic similarity. Because we are working with sentences, we use Sent2Vec [32], a lightweight text embedding model, to convert graph nodes into feature vectors. Sent2Vec is an unsupervised model that allows composing sentence embeddings using word vectors along with n-gram embeddings. An n-gram is a contiguous sequence of n items from a given sample of text or speech. Sent2Vec simultaneously trains the word vectors and the composition component.

C. Graph Neural Network

In recent years, Graph Neural Network (GNN) [33], such as Graph Convolutional Network (GCN) [34], Graph Attention Network (GAT) [35], Gated Graph Sequence Neural Networks (GGNN) [36], has been shown to be effective in modeling graph data, e.g., social networks [12], traffic networks [13]. GNN has also been applied to software engineering tasks such as code summarization [37], clone detection [38], and bug localization [39], and has achieved impressive performance. The goal of GNN is to train a parametric function via message passing between the nodes of graphs for further tasks, i.e., graph classification, node classification, and edge prediction.

GNN is a kind of neural network specifically designed to operate over data structured as graphs. It has gained popularity due to its ability to leverage the rich relational information in graph data. GNN accepts graph-structured data as input. A graph consists of nodes (or vertices) and edges. Each node represents an entity, and each edge represents a relationship between two entities. In addition to the graph structure, each node and edge can have associated attributes or features, represented as vectors. The input of GNN is usually a graph with some associated features for each node and edge. These features could be binary (indicating the presence or absence of a certain attribute), categorical (discrete values from a certain set), or numerical (continuous values). The graph structure is represented by an adjacency matrix that defines the connections between different nodes. The output of a GNN can be different depending on the task at hand. For node classification, the output would be a label or a set of labels for each node in the graph. For graph classification, the output would be a label or a set of labels for the entire graph. For link prediction, the output would be a set of predicted links (edges) in the graph. For graph generation, the output would be a new graph, generated based on learned patterns from the input graph.

The general architecture of a GNN involves a few key components: 1. Node Representation Learning: GNNs learn a vector representation (embedding) for each node in the graph. 2. Aggregation Function: The function used to aggregate the neighbors' representations for each node in each iteration. 3. Update Function: After aggregating the neighbors' representations, an update (or transformation) function is applied to compute the new representation for each node based on the node's current representation and its neighbors' aggregated representations. 4. Readout Function: After several iterations of node representation learning (also known as message passing), a readout function is used to aggregate all of the node representations in the graph to form a graph-level representation.

GNN differs from traditional neural networks in the following aspects: First, traditional neural networks like CNN and RNN typically deal with grid data, such as images (2D grids of pixels) or texts (1D sequences of words). In contrast, GNN is specifically designed to work with graph data, where entities (nodes) can have complex, non-grid relationships with each other. Second, in traditional neural networks, the process of feature extraction is done via predefined layers, and the connections between these layers are fixed. In GNN, the feature extraction process is dynamic and is dependent on graph structure.

For the function-level vulnerability detection task, representing a function as one or more graphs can better capture its structural information than representing it as plain text. Considering GNN is effective in encoding graph data, we also adopt GNN to capture vulnerability features from the graph representations of functions.

IV. APPROACH

We propose a novel model, named MGVD (**M**ULTIPLE-**G**RAPH-BASED **V**ULNERABILITY **D**ETECTION), to detect vulnerable functions. As shown in Fig. 2, MGVD consists of three stages, i.e., function representation, feature matrix generation, and classification. The function representation stage represents each function in three different forms, namely, statement texts, raw statement graphs and abstract statement graphs. The feature matrix generation stage encodes the three representations of the function into a feature matrix. The classification stage predicts whether the function contains vulnerabilities.

A. Function Representation

Given a function, this stage aims to transform it into three different representations, i.e., statement texts, raw statement graphs, and abstract statement graphs. Statement texts refer to the code texts of the statements in the function. We use this representation because code texts can help the model capture the semantic information in code [40], and many existing code representation methods also leverage code texts to represent source code [41], [42]. In addition, prior work shows that both semantic and structural information in code is important for vulnerability detection [43]. Considering this, we first transform the function into two graphs, namely *raw graph* and *abstract*

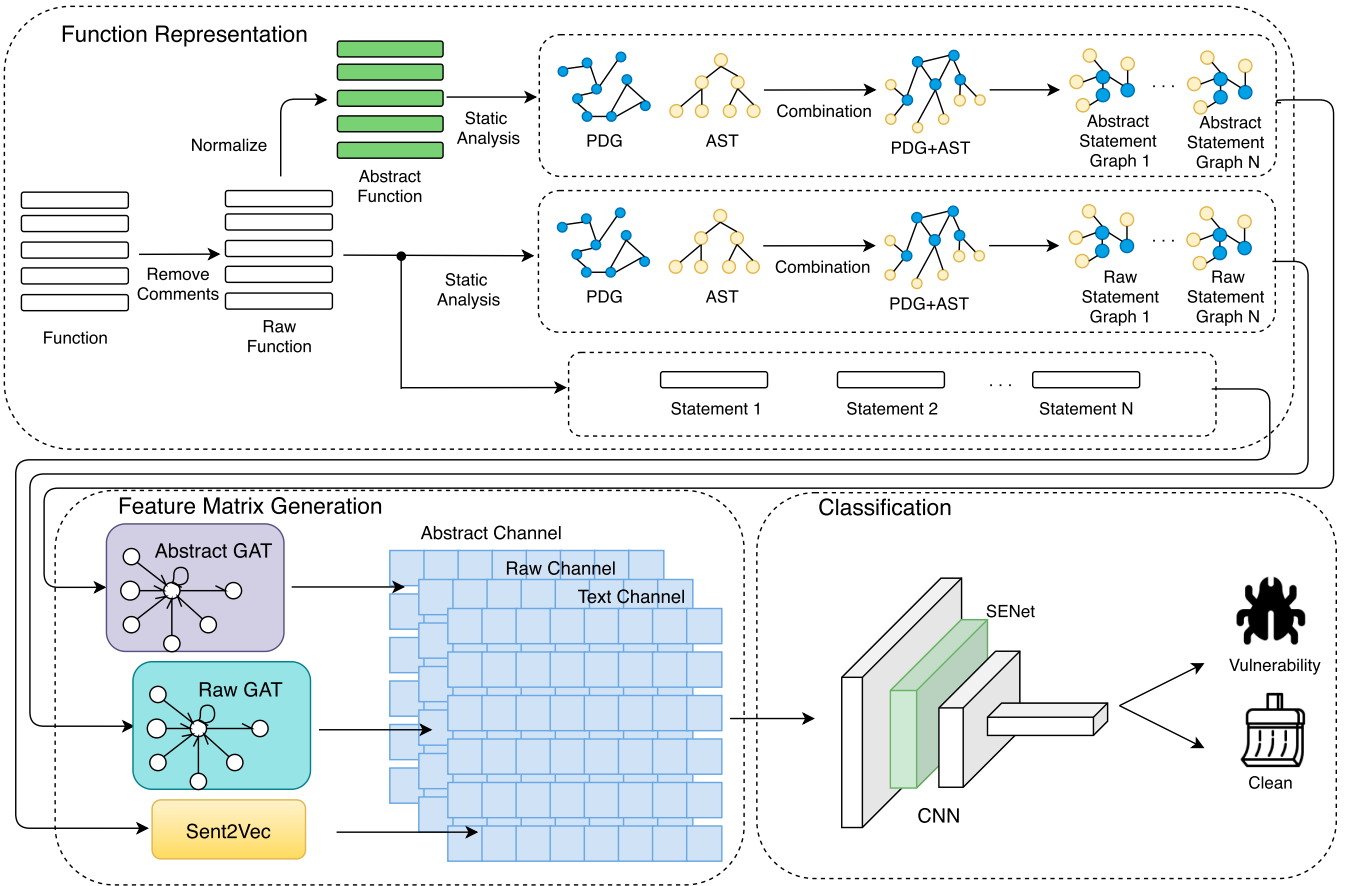


Fig. 2. Overall framework of MGVD.

graph. The raw graph is the combination of the PDG and the AST of this function. The abstract graph is the abstract version of the raw graph where user-defined variables, function names, and string literals are abstracted. The abstract graph captures the structural information in code, which can be useful for vulnerabilities only related to the structure of code. The raw graph provides the structural and semantic information in an integrated way. The Statement text, the abstract graph and the raw graph provide three different perspectives. By adaptively fusing the features extracted from them, we can effectively capture the structural and semantic information in code and improve vulnerability detection.

However, as we mentioned in Section I, the function graph is usually large and complex, preventing GNNs from identifying and capturing fine-grained information [22], [23]. Therefore, if we directly encode the raw graph and abstract graph using GNNs, we may fail to detect the vulnerabilities only related to a few fine-grained code elements, like the example shown in Section II. To overcome this challenge, we further extract two sequences of statement graphs from the raw graph and the abstract graph, respectively, based on the statements in the function. We refer to the statement graph extracted from the raw graph as *raw statement graph*, and the statement graph extracted from the abstract graph as *abstract statement graph* respectively.

Raw Function:

```

1 void badfun()
2 {
3   char * data;
4   char data_buf[100] = FULL_COMMAND;
5   data = data_buf;
6   badStatic = 1;
7   data = badSource(data);
8   if (SYSTEM(data) <= 0)
9   {
10    printLine("command execution failed!");
11    exit(1);
12  }
13 }

```

Abstract Function:

```

1 void FUN0()
2 {
3   char * VAR1;
4   char VAR2[100] = FULL_COMMAND;
5   VAR1 = VAR2;
6   VAR3 = 1;
7   VAR1 = FUN1(VAR1);
8   if (SYSTEM(VAR1) <= 0)
9   {
10    printLine(STR);
11    exit(1);
12  }
13 }

```

Fig. 3. Example of raw function and abstract function.

Specifically, given a function, we first remove the comments in it, as we would like to detect vulnerabilities only based on the information in source code. We refer to the obtained function as *raw function*. The code text of each statement is extracted as its *statement text*. For constructing the abstract graph later, we follow the practice of existing studies [7], [10], [44] to construct the abstract version of this function, namely *abstract function*. Fig. 3 shows an example of abstract functions. Specifically, given a raw function, we map each user-defined variable and function name in it with a symbol, having the form TYPE#. TYPE is set to “VAR” and “FUN” for variables and function names, respectively. # is a numerical ID generated sequentially

for each distinct type (i.e., variables or functions). For example, the first variable in a function will be normalized as VAR1 and the third unique function name will be normalized as FUN3. The variables (function names) using the same identifiers will use the same placeholders. In addition, the string contents do not affect source code structure either, so we map them into a special symbol “STR”. We abstract these tokens because they are unrelated to the structure of the function and may even introduce noise when we focus on encoding the code structure. Please note that this abstraction only guarantees that the same identifiers in a function share the same symbol, while the same identifiers across different functions can be replaced with different symbols.

Then, we use a static analysis tool named Joern [27] to construct the PDGs and the ASTs of both the raw function and the abstract function. The PDG contains the relationships between statements, and the AST contains the internal relationships within statements. We combine the PDG and the AST of the raw function to construct a graph and refer to it as *raw graph*. Similarly, we construct a graph from the abstract function and refer to it as *abstract graph*.

Specifically, we use PDG as the backbone, where each node denotes a statement, and replace PDG node with the AST of its corresponding statement. The raw graph contains the complete semantic and structural information of the function, and the abstract graph contains structural information of the function. The use of the raw graph can help MGVD learn structural and semantic information from a comprehensive perspective without neglecting semantic details. The use of the abstract graph can help MGVD focus on structural information while reducing interference from semantic information [18], [45].

Next, we extract raw statement graphs and abstract statement graphs from the raw graph and the abstract graph, respectively. Each statement graph targets representing one statement and the graphs in each sequence are sorted according to the order in which the corresponding statements appear in the function. The statements in a function are usually related to each other and the context of a statement plays an important role in understanding and encoding this statement. Hence, we construct statement graph for each statement by extracting not only its nodes and edges but also the nodes and edges of the statements in its context. Intuitively, if there is a path between statements f and s in PDG, we can include s in f 's context. We refer to the length of the path as context length. The maximum context length should be set to construct each statement graph. We would like to set this length to a small number. Because firstly, the longer the context length, the smaller the correlation between the corresponding statement pair. Secondly, each statement graph aims to capture the unique features of its focal statement, instead of all of the context. The complete context of each statement can be captured later from the feature matrix (described in Section IV-B). In addition, we found that if the maximum context length is greater than 2, the statement graphs of multiple statements within a single code block could be completely identical, making it unable to capture their unique features. For instance, when the maximum context length is set to 3 for the function shown in Fig. 3, the statement graphs generated for `data=data_buf`

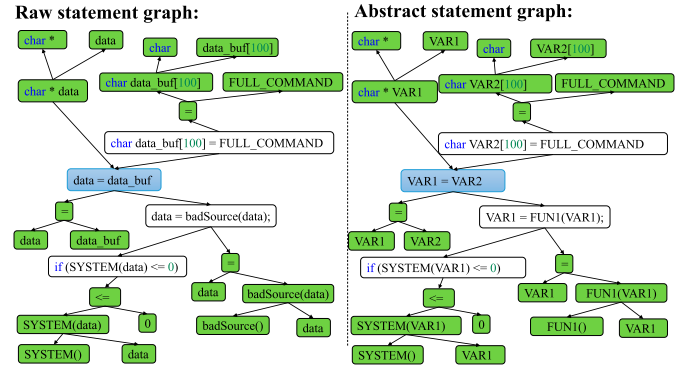


Fig. 4. Statement graph examples.

and `data=badSource(data)` are the same. Therefore, in our work, we set the maximum context length as 2, which means that each statement graph involves the nodes and edges of its focal statement and the statements with a distance of 2 from this focal statement in PDG. Fig. 4 shows the statement graph of the statement `data = data_buf` in Fig. 3. We have marked the PDG nodes in white and marked the AST nodes in green. Compared to the function graph, the statement graph is smaller in scale and contains the information in PDG and AST that is most related to a specific statement. Each statement graph contains the data dependency and control dependency relationships between the target statement and its neighbor statements. Each statement in the statement graph is an AST, which contains syntactic dependencies within the statement.

Finally, we sort the constructed sequences of statement texts, raw statement graphs and abstract statement graphs according to the order in which the statements occur in the function. By combining PDG with the order of statements, MGVD can obtain enough control-flow information from the representation.

B. Feature Matrix Generation

This stage takes as input the three representations of each function and encodes them into a feature matrix for further classification. Specifically, we first encode each statement in the three representations into a fixed-dimensional feature vector, i.e., Statement Encoding. Then we construct a 3D feature matrix based on these feature vectors, Matrix Generation.

1) *Statement Encoding*: After Feature Representation, each statement in the function is transformed into a statement text and two statement graphs. To encode them into feature vectors, we first use a Byte Pair Encoding (BPE) tokenizer [46] to tokenize the statement text and the code text of each graph into tokens. We choose BPE because it can mitigate the out-of-vocabulary problem and is widely used [20], [41]. BPE is a type of subword tokenizer that splits words into a sequence of subwords or character n-grams. An n-gram is a contiguous sequence of n items from a given sample of text or speech. This approach enables dealing with out-of-vocabulary words and reduces the size of the vocabulary needed for language models. BPE starts by tokenizing at the character level and iteratively merges the most frequent pairs of characters or character

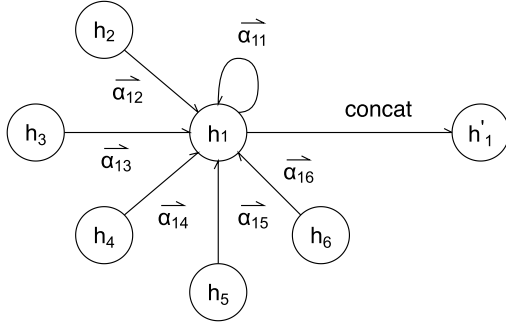


Fig. 5. The mechanism of GAT.

sequences to form a new token. This continues until a specified number of merges have been made, creating a balance between the granularity of tokenization and the size of the vocabulary. For example, the node “VAR1 = FUN1 (VAR1) ;” in Fig. 4 will be tokenized into a sequence of tokens [VAR1, =, FUN1, (, VAR1,), ;].

Next, we use Sent2Vec [32], a widely used sentence embedding technique, to convert each tokenized text into a fixed-dimensional feature vector. Although we can also achieve this goal by aggregating the word embedding of tokens produced by word-level embedding techniques, such as Word2Vec, this method ignores the context and word order and may be ineffective in encoding code texts. Sent2Vec is a language representation model that provides dense vector representations for sentences by training them in a supervised manner. It encodes the context and the order of tokens. The vectors produced by Sent2Vec can capture semantic meanings and can be used in various NLP tasks. Thus, Sent2Vec is more suitable for this task than word-level embedding techniques. We also compare Sent2Vec and Word2Vec in Section VI.B.2. In our approach, the Sent2Vec model is trained using all files in our training set.

Then, for each statement, we compute its feature vectors based on its statement text, abstract statement graph, and raw statement graph, respectively. The feature vector of its statement text is the corresponding embedding produced by Sent2Vec. To compute the feature vectors of its statement graphs, we first employ a type of Graph Neural Network named GAT (Graph Attention Network) to encode each graph and compute the features of each graph node. GAT uses the attention mechanism to compute the importance of neighboring nodes when updating the features of a node. It has been shown to allow for more nuanced feature updates and can lead to better performance on tasks involving graph-structured data [35], [36]. Then, we use average pooling to obtain the feature vector of each graph based on node features.

Specifically, the input of the GAT is a set of node features, $\mathbf{h} = \{h_1, h_2, h_3, \dots, h_N\}$, $h_i \in R^F$, where N is the number of nodes, and F is the dimension of node features. The output of the GAT is a new set of node features, $\mathbf{h}' = \{h'_1, h'_2, h'_3, \dots, h'_N\}$, $h'_i \in R^{F'}$. The number of nodes N remains unchanged, and the dimension of node features F can be changed to F' . As shown in Fig. 5, taking h_1 for example,

a shared linear transformation ($\vec{\alpha}_{11}, \vec{\alpha}_{12}, \vec{\alpha}_{13}, \vec{\alpha}_{14}, \vec{\alpha}_{15}, \vec{\alpha}_{16}$) is applied to h_1 and its neighbors (h_2, h_3, h_4, h_5, h_6) to aggregate features from the neighbors. Then, the aggregated features are concatenated to obtain h'_1 . Now that we have obtained the features of all nodes, but we want to get the graph feature that captures the information of the entire statement graph. Therefore, we add an average-pooling layer after GAT as follows:

$$y'_G = \text{Mean}(\{h_v \in R^d, \forall v \in G\}) \quad (1)$$

Here, y'_G is the feature vector of the statement graph.

2) *Matrix Generation*: After encoding statement graphs and statement texts, each statement is encoded into three feature vectors, i.e., the feature vectors of its raw statement graph, its abstract statement graph and its statement text. We refer to the three vectors as *raw vector*, *abstract vector* and *text vector*, respectively. Each row in these three vectors corresponds to a statement in the function, and the statements within the function also have their natural order. Therefore, we concatenate these three vectors into a feature matrix. This approach not only facilitates further processing but also takes into account the natural order of the source code. Specifically, for each type of vector, we first stack such vectors of all statements according to the order in which they appear in the function to construct a one-dimensional feature matrix. Then, we stack the feature matrix of the three types of vectors to construct a three-dimensional feature matrix for the function, as shown in Fig. 2.

Since different functions consist of different numbers of statements, the feature matrix may have different numbers of rows. Based on our experiments, we find that 98% of the functions in the datasets used by us (we will describe them in Section V-A) are at most 128 lines. Therefore, to make the feature matrix of different functions the same size, we set the number of matrix rows to a fixed value of 128. When the number of statements in a function is less than 128, we add empty statements to pad the rows. When the number of statements in a function is larger than 128, we only use the feature vectors of the first 128 statements to construct the matrix.

C. Classification

This stage takes as input the three-dimensional feature matrix produced by the feature matrix generation stage and aims to predict whether the input function is vulnerable or not based on this matrix. Inspired by the concept of channel in image processing, we treat the three types of statement feature vectors, i.e., text vector, raw vector and abstract vector, as three channels, referred to as *text channel*, *raw channel* and *abstract channel*. CNN has been extensively used and shown to be effective in learning from multi-channel inputs [47]. Therefore, we employ a CNN model as the classifier. Kim proposed a variant of CNN that can handle to text data, called TextCNN [48]. TextCNN can capture the spatial dependencies between words in a sentence, making it capable of recognizing semantic patterns in text data. Thus, we build our CNN model according to the architecture of TextCNN. In addition, in the feature matrix, different channels focus on different types of information, i.e., structural information, semantic information, and both. To adaptively fuse

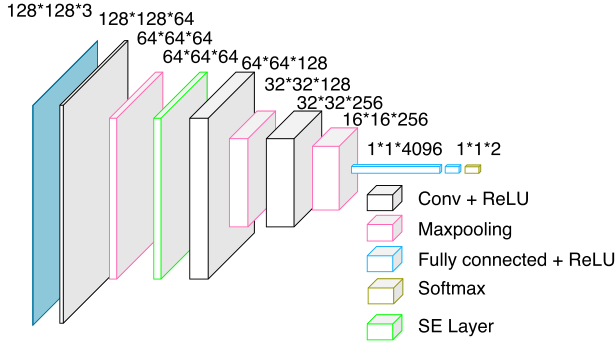


Fig. 6. Overall structure of classifier.

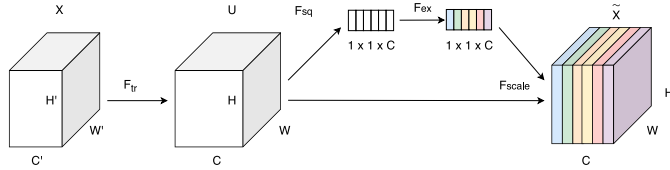


Fig. 7. Overall Structure of SENet Layer.

the weights among the three channels and boost the feature interaction, we add a Squeeze-and-Excitation Network (SENet) layer [49] between the first conv layer and the second conv layer in our CNN model. SENet can adaptively recalibrate channel-wise feature responses by explicitly modeling the interdependencies between channels. The classification model processes the entire matrix simultaneously, meaning that it considers all the statements in the function and can capture the relationship between statements.

The structure of the classifier is shown in Fig. 6. The size of the input feature matrix is $128 \times 128 \times 3$. It passes through a convolutional (conv) layer and a max pooling layer with 2×2 pixel window, stride 2, first. The convolutional (conv) layer consists of a conv [50] and a ReLU [51] function. The filter size is 3×3 , the conv stride is 1 pixel, and the padding is 1 pixel. Then, it goes through a SENet layer. The next level is two stack of conv layer and max pooling layer. Finally, it is passed through two fully connected layers: the first has 4096 channels each, and the second performs 2-way classification and thus contains 2 channels (one for each class). The last layer is the softmax layer. The output of the classifier is a probability distribution vector vulnerable and not vulnerable. Using this vector, MGVD predicts whether the input function is vulnerable.

Specifically, the structure of the SENet layer is shown in Fig. 7. The SENet layer is a computational layer that can be constructed for any given transformation $F_{tr}: X \rightarrow U$, $X \in \mathbb{R}^{H' \times W' \times C'}$, $U \in \mathbb{R}^{H \times W \times C}$. F_{tr} is a convolutional operator. The outputs of F_{tr} is $U = [u_1, u_2, \dots, u_C]$. In our model, F_{tr} is the conv layer before the SENet layer. SENet consists of two parts, squeeze and excitation. In the squeeze operation, the SENet squeezes global spatial information into a channel descriptor by using global average pooling to generate channel-wise statistics. This enables our model to exploit channel dependencies. The squeeze operation is shown as F_{sq} in Fig. 7

and defined as follows:

$$z_c = F_{sq}(u_c) = \frac{1}{H \times W} \sum_{i=1}^H \sum_{j=1}^W u_c(i, j), \quad (2)$$

where U is the output of the convolutional operator. A statistics $z \in \mathbb{R}^C$ is generated by shrinking U through spatial dimensions $H \times W$. Here, z_c is the c -th element of z .

In the excitation operation, the SENet captures channel-wise dependencies. It uses two fully connected layers to limit model complexity and aid generalisation. The first one is followed by a ReLU activation, and the second one is followed by a sigmoid activation.

$$s = F_{ex}(z, W) = \sigma(g(z, W)) = \sigma(W_2 \delta(W_1 z)), \quad (3)$$

where δ is the ReLU function and σ is the sigmoid function. The final output of the SENet layer is obtained by rescaling the transformation output U with the activations:

$$\tilde{x}_c = F_{scale}(u_c, s_c) = s_c \cdot u_c, \quad (4)$$

where $\tilde{X} = [\tilde{x}_1, \tilde{x}_2, \dots, \tilde{x}_C]$ and F_{scale} refers to channel-wise multiplication between the feature map $u_c \in \mathbb{R}^{H \times W}$ and the scalar s_c . $\tilde{X}$ is the output of the SENet layer. After training, the SENet layer can adaptively assign weights to the three representations and help MGVD to focus on the information that is related to vulnerabilities.

V. EXPERIMENT SETUP

In this section, we first introduce the datasets used to evaluate MGVD. Then, we present the detailed parameters and experiment environment for training our approach. Finally, we introduce the evaluation metrics.

A. Datasets

In this work, we focus on detecting C/C++ vulnerabilities at the function level and conduct our study on three well-known C/C++ vulnerability datasets. The first dataset is collected from Software Assurance Reference Dataset (SARD) [24] which is a project maintained by the National Institute of Standards and Technology (NIST)². This dataset has a large number of vulnerable functions with different types. The reason we use SARD as our data source is that the functions in SARD have both vulnerable versions and not vulnerable versions, which implies that it is suitable for being used to evaluate the vulnerability-detection ability of a model. Based on the Top25 CWE in 2021 [52] and previous studies [17], we collect 14,503 vulnerable functions and 28,451 not vulnerable functions from SARD as shown in Table II. Each test case in the SARD contains one or more vulnerable code functions along with their corresponding fixed, not vulnerable versions. We collect these functions to form our SARD dataset from SARD. Notably, when partitioning this dataset into training, validation, and test sets, we stratified by each type of CWE, following an 8:1:1 ratio. This strategy ensured the presence of each CWE type in all subsets, thereby

²<https://www.nist.gov/>

TABLE I
THE STATISTICS OF DATASETS

Dataset	Vul	Not-Vul	Ratio	Total
SARD	14,503	28,451	1:1.96	42,954
FFmpeg+QEMU	9,637	10,557	1:1.10	20,194
Big-Vul	6,201	88,923	1:14.34	95,124

TABLE II
THE CWES IN SARD DATASET

CWE	Description	Total	Train	Eval	Test
476	NULL Pointer Dereference	591	473	59	59
369	Divide By Zero	1446	1158	144	144
195	Signed to Unsigned Conversion Error	1230	984	123	123
122	Heap Based Buffer Overflow	3889	3109	390	390
127	Buffer Under-read	2462	1970	246	246
121	Stack Based Buffer Overflow	7638	6112	763	763
78	OS Command Injection	6275	5019	628	628
126	Buffer Over-read	2343	1875	234	234
134	Uncontrolled Format String	4183	3347	418	418
124	Buffer Underwrite	2461	1969	246	246
416	Use After Free	324	258	33	33
190	Integer Overflow	4659	3727	466	466
758	Undefined Behavior	980	782	99	99
191	Integer Underflow	4473	3581	446	446

avoiding situations where a CWE type is absent in the training set but present in the test set. Considering that the functions in SARD are often modified to isolate vulnerabilities from their original context, we also use two vulnerability datasets that are directly constructed from real-world vulnerabilities, i.e., FFmpeg+QEMU, which was created by Zhou et al. [16] and published by CodeXGLUE, and Big-Vul dataset, which was created by Fan et al. [25]. In these two datasets, both functions and their labels are collected from real-world projects. The FFmpeg+QEMU and Big-Vul datasets have been extensively utilized in prior research [14], [15]. Our approach uses a static analysis tool Joern to construct PDGs and ASTs for the functions in each dataset. Due to the limitation of Joern, some functions in the datasets cannot be properly analyzed. Thus, we filter out the functions to ensure the PDG of each used function is correctly constructed. And we remove the functions longer than 128 statements from the dataset. The statistics of the three datasets are shown in Table I.

B. Training Details

To train our approach, we set the initial learning rate to 0.01 with a momentum of 0.5 and clip the gradients norm by 5. The learning rate decay is set to 0.99. The batch size is 16. Our model is trained using the Stochastic Gradient Descent (SGD) algorithm [53]. We use cross entropy as the loss function. Cross entropy is commonly used in classification tasks. It measures the dissimilarity between the predicted probability distribution and the true distribution. Our task is a binary classification problem, so the cross entropy can be calculated as follows:

$$H(Y) = -\frac{1}{N} \sum_{i=1}^N [y_i \log(p(y_i)) + (1 - y_i) \log(1 - p(y_i))], \quad (5)$$

where N is the number of instances, y_i is the true label (i.e., 0 or 1) of the i -th instance (0 or 1), $p(y_i)$ is the predicted probability that the i -th instance is of class 1. Since the number of vulnerable functions is much less than that of not vulnerable functions in the three datasets (especially in the Big-Vul dataset), we apply the oversampling technique [54], [55] to mitigate the impact of data imbalance during training. Specifically, we randomly copy vulnerable functions in the training set to make the number of vulnerable functions equal to the number of not vulnerable functions. For the SARD dataset and the Big-Vul dataset, we randomly select 80%, 10% and 10% of the data as the training, validation and test sets following prior work [14], [15], [17]. For the FFmpeg+QEMU dataset, considering that the CodeXGLUE benchmark [56] has incorporated the FFmpeg+QEMU dataset and has been widely used, we follow CodeXGLUE's data division for the training, validation, and testing. In addition, for each dataset, we train our approach for 100 epochs, save the model after each epoch, and then select the model with the best performance according to F1-score on the validation set as our final model. On the SARD, FFmpeg+QEMU and Big-Vul validation sets, MGVD achieved its highest F1-score at the 39th, 45th, and 28th epochs, respectively. Consequently, we selected the model from these epochs as our final model on each dataset.

C. Evaluation Metrics

We adopt five widely-used classification metrics, i.e., accuracy, precision, recall, F1-score and Matthews Correlation Coefficient (MCC) to evaluate the performance of MGVD. Accuracy is the ratio of correctly predicted functions to all functions. Precision measures the proportion of functions that are truly vulnerable among those predicted by the model to contain vulnerabilities. It is calculated as $Precision = \frac{TP}{TP+FP}$ where TP is the number of functions that are predicted by the model to be vulnerable and indeed contain vulnerabilities. FP is the number of functions that are predicted to be vulnerable but do not contain vulnerabilities. Recall measures the proportion of all actual vulnerable functions that are correctly predicted by the model as vulnerable. It is calculated as $Recall = \frac{TP}{TP+FN}$ where FN is the number of functions that actually contain vulnerabilities but are predicted to be non-vulnerable. F1-score takes both precision and recall into account, which can be regarded as their harmonic mean. It is calculated as $F1 - score = \frac{2 * Precision * Recall}{Precision + Recall}$. MCC is a measure of the quality of binary (two-class) classifications. It takes into account true and false positives and negatives and is generally regarded as a balanced measure which can be used even if the classes are of very different sizes.

VI. EVALUATION

We evaluate the performance of MGVD on the three vulnerability datasets. We attempt to answer the following research questions:

- RQ1: How effective is our approach compared with the state-of-the-art baselines?
- RQ2: How effective are the techniques in our approach?

- RQ3: What is the impact of different CNN parameters on the effectiveness of our approach?

A. RQ1: Effectiveness Evaluation

1) *Baselines*: We compare MGVD with seven state-of-the-art vulnerability detection techniques. We train the baselines with hyperparameters set according to their original paper and select the models with the best F1-score on the validation set for evaluation.

- Devign [16]. Devign is a GNN-based model. It leverages AST, CFG, DFG and code sequences for graph learning. It uses AST as the backbone and combines CFG and DFG into a joint graph. Then its Gated Graph Neural Network model learns the node features for graph-level classification.
- FUNDED [17]. FUNDED is a GNN-based model that operates on graph representations of the input function and has the capability to learn and aggregate multiple code relationships. FUNDED encodes different code relationships in AST, CFG and DFG, but adds more edges to AST such as “GuardedBy”, “ComputeFrom” to capture richer intra-program relations.
- REVEAL [14]. REVEAL is a GNN-based model for function-level vulnerability detection. It leverages representation learning techniques to detect vulnerabilities. REVEAL first extracts the Code Property Graph (the combination of AST, CFG and PDG) [27] of the function. Then it encodes nodes into vectors and inputs them into GGNN to detect vulnerability.
- SySeVR [10]. SySeVR is a systematic framework for using deep learning to detect vulnerabilities. It leverages sliced programs as basic units to train the model and adopts multiple kinds of neural networks to detect vulnerabilities.
- SEVulDet [44]. SEVulDet is a Semantics-Enhanced learnable Vulnerability Detector that employs a path-sensitive code slicing approach to extract sufficient path semantics and control flow logic into code gadgets. It inserts a spatial pyramidal pooling layer into the CNN with a well-designed multilayer attention mechanism.
- GraphCodeBERT [57]. GraphCodeBERT is a state-of-the-art language model specifically designed for programming and coding tasks. It utilizes graph-based pre-trained tasks to capture the structural information of code.
- UniXcoder [58]. UniXcoder is a unified cross-modal pre-trained model for programming languages that incorporates semantic and syntax information from code comment and AST.
- LineVul [59]. LineVul is a transformer-based model, which first uses CodeBERT to predict vulnerable functions and then leverages the attention mechanism to locate vulnerable code lines. We use its function-level vulnerability detection part as our baseline.

2) *Experimental Results*: In this research question, we first compare the performance of our approach with the baselines across the three datasets. Secondly, we construct a hybrid dataset by merging the training, validation, and test sets of the three datasets to create the training, validation, and test sets

of the hybrid dataset. Then we train and evaluate MGVD and baselines on this dataset. In this way, we can evaluate the performance of various models in handling functions of different complexities, in general instead of within a specific dataset. Besides, this hybrid dataset to some extent helps evaluate the generality of each model. Finally, we partition the functions in the test set into four quartiles based on the McCabe’s Cyclomatic Complexity [60], ranging from 1 to 5, 6 to 10, 11 to 20, and 21 to 487 to assess the performance of all approaches across the functions of different complexities. We employ Joern to extract CFGs of the functions and then compute the McCabe’s Cyclomatic Complexity of each function following McCabe [60]. McCabe’s Cyclomatic Complexity is a software metric that provides a quantitative measure of a program’s complexity. McCabe’s Cyclomatic Complexity for source code refers to the count of the maximum number of linearly independent paths that exist in the code. The McCabe’s Cyclomatic Complexity M of a program’s source code is calculated using the formula:

$$M = E - N + 2 * P, \quad (6)$$

where E is the number of edges in the CFG, N is the number of nodes in the CFG and P is the number of connected components. When only considering one function, the value of P is 1. A higher McCabe’s Cyclomatic Complexity value indicates that the function is more complex. Subsequently, we compared the performance of our method with the baselines across these four quartiles.

Table III presents the accuracy, precision, recall, F1-score and MCC of our approach and the baselines. We can make the following observations from the table:

1) **Our approach outperforms all baselines in terms of precision F1-score and MCC on all the datasets.** In detail, on the three datasets, the improvements of our approach over the best-performing baseline are 10.28%, 9.87% and 9.68% in terms of F1-score, 9.46%, 5.65% and 12.75% in terms of precision and 25.62%, 36.79% and 9.14% in terms of MCC. GNN-based techniques, e.g., Devign, FUNDED, REVEAL, represent a function as a single graph, which is usually large, and process this graph using GNNs, making them prone to overlooking detailed information. For path-based techniques, e.g., SySeVR and SEVulDet, because analyzing all possible paths in complex functions can be highly resource-intensive, they perform poorly when processing complex functions. GraphCodeBERT, UniXcoder, and LineVul are pre-trained models. LineVul does not use the graph structure of source code, and thus may overlook the structure information of code. GraphCodeBERT and UniXcoder utilize PDG and AST respectively. However, they do not preserve the graph structure during processing, which makes them less effective in learning structural information. Our approach uses three different forms to represent function to better capture structural and semantic information in it. Then, our approach adopts the SENet layer to learn to adaptively assign weights to three representations, so that we can effectively leverage both structural and semantic features of source code. Besides, we combine PDG with AST to construct the graph representations and extract a sub-graph for each

TABLE III
THE PERFORMANCE OF MGVD AND THE BASELINES

Approach	SARD					FFmpeg+QEMU					Big-Vul				
	Accuracy	Precision	Recall	F1-score	MCC	Accuracy	Precision	Recall	F1-score	MCC	Accuracy	Precision	Recall	F1-score	MCC
FUNDED	0.6850	0.5244	0.7186	0.6063	0.3664	0.5763	0.5119	0.5358	0.5236	0.1426	0.8733	0.1031	0.1226	0.1120	0.0445
SySeVR	0.6261	0.4598	0.6159	0.5265	0.2347	0.5895	0.5252	0.5770	0.5499	0.1748	0.7227	0.1215	0.5226	0.1972	0.1425
DeSign	0.6310	0.4668	0.6524	0.5442	0.2582	0.5966	0.5324	0.5889	0.5592	0.1899	0.6414	0.1014	0.5726	0.1723	0.1120
SEVulDet	0.7162	0.5623	0.7198	0.6314	0.4145	0.5858	0.5202	0.5998	0.5572	0.1733	0.6787	0.1101	0.5548	0.1838	0.1273
GraphCodeBERT	0.6396	0.4742	0.6214	0.5379	0.2573	0.6122	0.5601	0.5000	0.5284	0.2018	0.8244	0.2137	0.6323	0.3195	0.2941
UniXcoder	0.6549	0.4912	0.6152	0.5462	0.2780	<u>0.6211</u>	<u>0.5683</u>	0.5325	0.5498	<u>0.2237</u>	0.8029	0.2131	<u>0.7516</u>	<u>0.3320</u>	<u>0.3274</u>
LineVul	0.6731	0.5121	0.6697	0.5804	0.3281	0.6159	0.5603	0.5390	0.5495	0.2151	0.8103	<u>0.2140</u>	0.7145	0.3294	0.3180
Reveal	0.6754	0.5141	0.7055	0.5948	0.3467	0.6023	0.5401	0.5694	0.5544	0.1959	0.7656	0.1920	0.8097	0.3105	0.3164
MGVD	0.7641	0.6155	0.7992	0.6963	0.5207	0.6569	0.6004	0.6291	0.6144	0.3060	0.8310	0.2413	0.7425	0.3642	0.3573
Improvement	6.69%	9.46%	11.03%	10.28%	25.62%	5.77%	5.65%	4.88%	9.87%	36.79%	-4.85%	12.75%	-8.30%	9.68%	9.14%

statement, which helps our approach more effectively capture fine-grained code elements from graphs. Thus, these technical enhancements lead to the better performance of our approach. 2) **Only on the Big-Vul dataset, the accuracy and recall of our approach are lower than the baselines.** Although FUNDED achieves the best performance in terms of accuracy on the Big-Vul dataset, it performs worst among all the techniques in terms of precision, recall and F1-score. This is because FUNDED tends to predict functions as not vulnerable instead of effectively identifying vulnerable functions. Since there are many more not vulnerable functions than vulnerable ones in the Big-Vul dataset, FUNDED achieves a high accuracy but cannot perform well in terms of other metrics. Our approach not only achieves a high accuracy (the second-best accuracy) but also performs well in terms of other metrics. REVEAL and UniXcoder achieve higher recall than our approach on the Big-Vul dataset, but their precision is lower than our approach. F1-score takes into account both the precision and recall of the classification model. According to the F1-score, our approach achieves better performance. 3) **Some baselines do not perform consistently across different datasets.** For instance, SEVulDet and FUNDED rank second and third in terms of F1-score on the SARD dataset, but they are both at the bottom of the rankings on the Big-Vul dataset. The main reasons include: First, the Big-Vul dataset is collected from real-world projects and is highly imbalanced, while SEVulDet and FUNDED's training methods are not well-suited for highly imbalanced datasets. Second, both methods normalize the function into an abstract form and employ a single representation strategy, causing them to overlook some semantic information. Compared to the synthetic SARD dataset, the semantic information of the functions in the Big-Vul dataset can be more helpful for vulnerability detection. This results in SEVulDet and FUNDED being less effective on the Big-Vul dataset than on the SARD dataset. In contrast, MGVD utilizes multiple representation strategies and fuses raw function, abstract function, and text together, leading to its better robustness. As a result, MGVD achieves the best F1-score on all three datasets.

Table IV shows the performance of MGVD and the baselines on the hybrid dataset. We observe that MGVD outperforms the best-performing baseline, i.e., REVEAL, by 11.86% in terms of F1-score and 21.56% in terms of MCC on the hybrid dataset. Tables V and VI display the performance of MGVD and the baselines on functions with different McCabe's Cyclomatic Complexity ranges. According to Table VI, MGVD

TABLE IV
THE PERFORMANCE OF MGVD AND THE BASELINES ON HYBRID DATASET

Approach	Hybrid				
	Accuracy	Precision	Recall	F1-score	MCC
DeSign	0.6326	0.2816	0.6163	0.3866	0.2006
SySeVR	0.6789	0.3112	0.5846	0.4061	0.2332
GraphCodeBert	0.6776	0.3154	0.6116	0.4161	0.2472
SEVulDet	0.6778	0.3223	0.6491	0.4307	0.2686
FUNDED	0.7782	<u>0.4251</u>	0.5134	0.4651	0.3291
LineVul	0.7351	0.3760	0.6217	0.4686	0.3233
UniXcoder	0.7398	0.3813	0.6183	0.4717	0.3278
REVEAL	0.7152	0.3688	0.7256	<u>0.4890</u>	<u>0.3547</u>
MGVD	<u>0.7755</u>	0.4404	<u>0.7216</u>	0.5470	0.4312
Improvement	-0.35%	3.60%	-0.55%	11.86%	21.56%

outperforms the baselines in most metrics in the range of McCabe's Cyclomatic Complexity values from 11 to 21. MGVD outperforms the best-performing baseline by 20.20% in terms of F1-score and by 45.73% in terms of MCC. In the range of McCabe's Cyclomatic Complexity values from 22 to 487, MGVD slightly outperforms the best-performing baseline by 4.20% in terms of F1-score. These functions, based on their McCabe's Cyclomatic Complexity values, can be considered of high complexity. These results confirm that our approach effectively addresses the challenge of poor performance on complex functions that current approaches face. As evident from Table V, although MGVD does not outperform the baselines in the range of McCabe's Cyclomatic Complexity values from 1 to 5, the performance difference between MGVD and the best-performing baseline is small. For example, the best-performing baseline has an F1-score of 0.5484, while MGVD's is 0.5339. But in the range of McCabe's Cyclomatic Complexity values from 6 to 10, MGVD slightly outperforms the best-performing baseline by 2.19% in terms of F1-score and by 2.81% in terms of MCC. According to the McCabe's Cyclomatic Complexity, we conclude that MGVD can also achieve comparable performance on functions of low to medium complexity.

In summary, MGVD outperforms the baselines in terms of Precision, F1-score and MCC on the three datasets and it is more effective in handling complex functions.

B. RQ2: Ablation Studies

1) *Experimental Setup:* Our approach enhances existing approaches in several ways: 1) We use three ways, i.e., abstract graph, raw graph and raw text, to represent each function, which results in three channels of statement features. 2) We

TABLE V
THE PERFORMANCE OF MGVD AND THE BASELINES IN MCCABE'S CYCLOMATIC COMPLEXITY VALUES RANGE FROM 1–10

Approach	MCC 1-5					MCC 6-10				
	Accuracy	Precision	Recall	F1-score	MCC	Accuracy	Precision	Recall	F1-score	MCC
SySeVR	0.6956	0.3986	<u>0.6841</u>	0.5037	0.3288	0.6271	0.3049	0.6084	0.4062	0.1984
Devign	0.6346	0.3426	0.6730	0.4540	0.2494	0.6821	0.3628	<u>0.6826</u>	0.4738	0.3036
GraphCodeBERT	0.6996	0.4017	0.6752	0.5037	0.3291	<u>0.7971</u>	<u>0.5168</u>	0.4982	0.5073	0.3797
REVEAL	0.7027	0.4047	0.6730	0.5054	0.3318	0.7632	0.4524	0.6144	0.5211	0.3762
SEVulDet	<u>0.7433</u>	<u>0.4502</u>	0.6185	0.5211	0.3599	0.7572	0.4435	0.6204	0.5172	0.3700
LineVul	0.7270	0.4354	0.7041	0.5380	0.3802	0.7966	0.5153	0.5030	0.5091	0.3809
UniXcoder	0.7310	0.4404	0.7063	<u>0.5425</u>	<u>0.3867</u>	0.7973	0.5175	0.4970	0.5070	0.3797
FUNDED	0.7717	0.4955	0.6140	0.5484	0.4021	0.7840	0.4878	0.5988	<u>0.5376</u>	<u>0.4022</u>
MGVD	0.7361	0.4440	0.6696	0.5339	0.3750	0.7504	0.4420	0.7257	0.5494	0.4135
Improvement	-4.62%	-10.41%	-5.20%	-2.65%	-6.74%	-5.89%	-14.58%	6.32%	2.19%	2.81%

TABLE VI
THE PERFORMANCE OF MGVD AND THE BASELINES IN MCCABE'S CYCLOMATIC COMPLEXITY VALUES RANGE FROM 11 – 487

Approach	MCC 11–21					MCC 22–487				
	Accuracy	Precision	Recall	F1-score	MCC	Accuracy	Precision	Recall	F1-score	MCC
FUNDED	0.6891	0.4095	0.4597	0.4332	0.2206	0.8730	0.0914	0.1354	0.1092	0.1037
Devign	0.6193	0.3477	0.5403	0.4231	0.1671	0.5945	0.0701	0.4934	0.1228	0.1406
GraphCodeBERT	0.5128	0.3003	0.6657	0.4139	0.1108	0.7010	0.1012	0.5328	0.1700	0.1920
SEVulDet	0.5181	0.3137	0.7279	0.4384	0.1544	0.6924	0.0964	0.5197	0.1627	0.1839
SySeVR	0.7069	0.4384	0.4772	0.4570	0.2572	0.6859	0.1045	0.5895	0.1775	0.2028
UniXcoder	<u>0.7127</u>	<u>0.4616</u>	0.6725	<u>0.5475</u>	<u>0.3607</u>	0.7183	0.0974	0.4716	0.1614	0.1796
LineVul	0.6848	0.4287	0.6599	0.5197	0.3161	0.7194	0.1170	0.5590	0.1935	0.2143
REVEAL	0.6065	0.3821	0.8474	0.5267	0.3274	0.7884	<u>0.1855</u>	0.7904	0.3004	0.3372
MGVD	0.7941	0.5763	0.7668	0.6580	0.5257	<u>0.8215</u>	0.2010	0.7074	0.3130	0.3396
Improvement	11.42%	24.84%	-9.52%	20.20%	45.73%	-5.90%	8.38%	-10.50%	4.20%	0.71%

use Sent2Vec to embed source code. 3) We use a sequence of statement graphs instead of a single function graph as the graph representation of each function. 4) We use an attention-based GCN, i.e., GAT, to encode each statement graph. 5) We use CNN as the classifier. 6) We use SENet in the CNN model to capture channel-wise dependencies. To evaluate the effectiveness of these techniques in our approach, we also perform ablation studies. Specifically, we construct several variants of MGVD and compare the performance between MGVD and these variants:

The effect of different feature channels. To explore the impact of each feature channel in our approach, we compare MGVD with its variants, as follows:

MGVD-R (Raw graph channel): MGVD-R only keeps the raw graph channel in the function feature matrix.

MGVD-A (Abstract graph channel): MGVD-A only keeps the abstract graph channel in the function feature matrix.

MGVD-T (Text channel): MGVD-T only keeps the text channel in the function feature matrix.

MGVD-RA (Raw graph channel and Abstract graph channel): MGVD-RA drops the text channel but keeps the raw graph channel and the abstract graph channel in the function feature matrix.

MGVD-RT (Raw graph channel and Text channel): MGVD-RT drops the abstract graph channel but keeps the raw graph channel and the text channel in the function feature matrix.

MGVD-AT (Abstract graph channel and Text channel): MGVD-AT drops the raw graph channel but keeps the abstract graph channel and the text channel in the function feature matrix.

The effect of Sent2Vec. To investigate the impact of Sent2Vec, we compare MGVD with another popular word embedding method Word2Vec. Compared to Sent2Vec, Word2Vec represents another category of embedding methods, embedding

each word in the sentence. We will compare these two types of embedding methods.

MGVD-Word2Vec: Compared to MGVD, MGVD-Word2Vec leverages Word2Vec instead of Sent2Vec to embed source code instead of Sent2Vec. Given that the dimension of the feature matrix is $128 \times 128 \times 3$, we set the length of the output vector to 128. For each sentence, we adopted the aggregation strategy from the work of Le et al. [61], where we take the average of all token vectors to represent the sentence.

The effect of statement graphs. To investigate the impact of statement graphs, we compare MGVD with a variant as follows:

MGVD-SG (Single Graph): Compared to MGVD, MGVD-SG does not extract statement graphs from the raw graph and the abstract graph of the input function. Instead, it directly processes the raw graph and the abstract graph using two GATs, respectively, and obtains two feature vectors of the function. The code text of the function is embedded as a feature vector using Sent2Vec. Then, we use the three feature vectors of the function to construct a feature matrix of which the size is $128 \times 1 \times 3$ instead of $128 \times 128 \times 3$ because each graph representation used by MGVD-SG contains only one function graph instead of multiple statement graphs. Since the height of the function matrix is only 1, in the classifier stage, we use a fully connected layer and instead of the CNN.

The effect of GAT. We also perform an ablation study to investigate the effectiveness of GAT in our approach. In the fields of NLP and program representation, there are three types of commonly used GNNs: Recurrent Graph Neural Network, Spectral-based Convolutional Graph Neural Network, and Spatial-based Convolutional Graph Neural Network [62]. Their representative models are respectively Gated Graph Sequence Neural Network (GGNN), Graph Convolutional Network (GCN), and Graph Attention Network (GAT). In our

TABLE VII
THE PERFORMANCE OF MGVD'S VARIANTS

Approach	SARD					FFmpeg+QEMU					Big-Vul				
	Accuracy	Precision	Recall	F1-score	MCC	Accuracy	Precision	Recall	F1-score	MCC	Accuracy	Precision	Recall	F1-score	MCC
MGVD-R	0.6186	0.4511	0.5979	0.5142	0.2160	0.5740	0.5102	0.4892	0.4994	0.1290	0.6423	0.0960	0.5323	0.1627	0.0937
MGVD-A	0.5718	0.3910	0.4814	0.4315	0.0953	0.5556	0.4900	0.5597	0.5225	0.1112	0.6504	0.0916	0.4883	0.1542	0.0778
MGVD-T	0.5804	0.3880	0.4207	0.4037	0.0810	0.5471	0.4804	0.5195	0.4992	0.0873	0.6504	0.0939	0.5048	0.1584	0.0856
MGVD-RA	0.7218	0.5733	0.6847	0.6235	0.4103	0.6225	0.5603	0.6095	0.5839	0.2403	0.7817	0.1798	0.6577	0.2823	0.2596
MGVD-RT	0.6552	0.4912	0.6036	0.5416	0.2736	0.5895	0.5232	0.6247	0.5695	0.1857	0.7605	0.1688	0.6819	0.2707	0.2510
MGVD-AT	0.6690	0.5074	0.6740	0.5789	0.3238	0.6183	0.5665	0.5174	0.5408	0.2160	0.7719	0.1819	0.7129	0.2899	0.2769
MGVD-Word2Vec	0.7314	0.5831	0.7162	0.6428	0.4371	0.6470	0.5927	0.5998	0.5962	0.2827	0.7912	0.1639	0.5431	0.2517	0.2106
MGVD-SG	0.5161	0.3620	0.5683	0.4422	0.0547	0.5669	0.5017	0.4675	0.4840	0.1119	0.6660	0.0934	0.4734	0.1560	0.0802
MGVD-GCN	0.6466	0.4826	0.6497	0.5538	0.2798	0.5490	0.4814	0.4913	0.4863	0.0845	0.7363	0.1572	0.6968	0.2565	0.2374
MGVD-GGNN	0.7452	0.5991	0.7403	0.6621	0.4681	0.6395	0.5827	0.5998	0.5911	0.2690	0.8126	0.2108	0.6835	0.3222	0.3053
MGVD-RNN	0.5008	0.3543	0.5821	0.4404	0.0395	0.5829	0.5212	0.4935	0.5070	0.1462	0.7197	0.1317	0.5903	0.2154	0.1732
MGVD-LSTM	0.5314	0.3826	0.6322	0.4767	0.1068	0.5877	0.5262	0.5119	0.5190	0.1583	0.7080	0.1252	0.5806	0.2060	0.1600
MGVD-WS	0.7484	0.5998	0.7669	0.6731	0.4832	0.6385	0.5862	0.5716	0.5788	0.2624	0.8126	0.2147	0.7060	0.3293	0.3164
MGVD	0.7641	0.6155	0.7992	0.6963	0.5207	0.6569	0.6004	0.6291	0.6144	0.3060	0.8306	0.2409	0.7427	0.3638	0.3570

work, MGVD employed GAT. Thus, we use GCN and GGNN for comparison. We construct the following variants:

MGVD-GCN: MGVD-GCN replaces GAT with GCN to encode statement graphs.

MGVD-GGNN: MGVD-GGNN replaces GAT with GGNN to encode statement graphs.

The effect of CNN. To investigate the impact of using a CNN as the classifier, we include a comparison with RNN and LSTM, as these types of neural networks are typically used for encoding text data. Specifically, we compare MGVD with two variants as follows:

MGVD-RNN: MGVD-RNN replaces CNN with vanilla RNN as the classifier. We arrange the feature matrix into a sequence row by row and input it into the RNN.

MGVD-LSTM: MGVD-LSTM replaces CNN with LSTM as the classifier. We arrange the feature matrix into a sequence row by row and input it into the LSTM.

The effect of the SENet layer. To explore the impact of the SENet layer in our approach, we compare MGVD with a variant as follows:

MGVD-WS (Without SENet): MGVD-WS drops the SENet layer in our approach.

2) Experimental Results: The effect of different feature channels. Table VII demonstrates the performance of MGVD-R, MGVD-A, MGVD-T, MGVD-RA, MGVD-RT and MGVD-AT compared with our approach. From this table, we see that **MGVD outperforms the six variants by substantial margins**. In detail, MGVD outperforms the single-channel variants by over 30% in terms of each metric and performs better than the two-channel variants by about 10% in terms of all evaluated metrics. The performance of the single-channel variants, i.e., MGVD-R, MGVD-A, and MGVD-T, is relatively weak. With more feature channels being used, the performance improves significantly. The two-channel variants outperform the single-channel variants by over 5% in terms of F1-score. We notice that there are no substantial performance differences among MGVD-R, MGVD-A and MGVD-T, which does not provide evidence to determine which channel is more useful. By comparing the two-channel variants, i.e., MGVD-RA, MGVD-RT, MGVD-AT, with MGVD, we observe that each channel plays a role in enhancing the performance. Hence, it is apparent that the multi-channel approach in MGVD is effective. By comparing MGVD-RT with MGVD, we find that the graph structure is

helpful for vulnerability detection. By comparing MGVD-RT and MGVD-AT with MGVD, we find that using raw statement graphs or abstract statement graphs can achieve better performance. The performance difference between MGVD-RA and MGVD demonstrates that the classifier can benefit from more semantic information. These results confirm the effectiveness and necessity of the three feature channels incorporated in MGVD.

The effect of Sent2Vec. Table VII shows the performance of MGVD-Word2Vec compared with our approach. We observe that **MGVD, which uses Sent2Vec, slightly outperforms MGVD-Word2Vec in all evaluated metrics on all used datasets**. But the relative improvement of MGVD over MGVD-Word2Vec on the Big-Vul dataset in F1-score reach 44.54%. Although Word2Vec and Sent2Vec are both text embedding tools, their operations and aims are different. Word2Vec is a tool that embeds individual words. It trains its model by predicting a word from the context or predicting the context from a word. The advantage of this method is that it can capture similarities and semantic relationships between words, such as synonyms and antonyms. However, Word2Vec doesn't handle phrases or sentences directly. If we want to get a vector for a sentence, we need to aggregate the vectors of the individual words, often by averaging them, but this might not always effectively capture the semantic meaning of the sentence. Sent2Vec is trained in a way that is similar to Word2Vec but adapted to operate at the sentence level. Its embeddings can capture more semantic information and are more powerful when the order of the words is important. Since our task requires understanding the meaning of each statement, Sent2Vec is more suitable.

The effect of statement graphs. Table VII demonstrates the performance of MGVD-SG compared with our approach. From this table, we find that **MGVD outperforms MGVD-SG by over 16% in all evaluated metrics on all included datasets**. This indicates that using statement graphs is effective and beneficial. One possible reason for this is that compared to large function graphs, it is easier for GNN to analyze relatively small statement graphs and capture vulnerability-related features from statement graphs.

The effect of CNN. From Table VII, we observe that **MGVD, which uses CNN as the classifier, consistently outperforms both MGVD-RNN and MGVD-LSTM by over 12% in all evaluated metrics on all included datasets**.

TABLE VIII
PERFORMANCE BASED ON VARYING NUMBERS OF CONVOLUTIONAL LAYERS

Approach	SARD					FFmpeg+QEMU					Big-Vul				
	Accuracy	Precision	Recall	F1-score	MCC	Accuracy	Precision	Recall	F1-score	MCC	Accuracy	Precision	Recall	F1-score	MCC
MGVD-1Conv	0.5139	0.3679	0.6126	0.4597	0.0726	0.5475	0.5235	0.5659	0.5437	0.0966	0.6568	0.1204	0.6762	0.2043	0.1699
MGVD-2Conv	0.6164	0.4557	0.7010	0.5523	0.2597	0.5470	0.5227	0.5774	0.5487	0.0967	0.6577	0.1171	0.6496	0.1984	0.1582
MGVD-4Conv	0.7083	0.5519	0.7240	0.6262	0.4041	0.6368	0.5984	0.7255	0.6558	0.2846	0.7721	0.1734	0.6629	0.2749	0.2526
MGVD-5Conv	0.7017	0.5463	0.6867	0.6085	0.3790	0.5549	0.5263	0.6836	0.5939	0.1251	0.7904	0.2094	0.7972	0.3316	0.3356
MGVD	0.7641	0.6155	0.7992	0.6963	0.5207	0.6569	0.6004	0.6291	0.6144	0.3060	0.8306	0.2409	0.7427	0.3638	0.3570
Improvement	7.87%	11.52%	10.40%	11.20%	28.87%	4.98%	5.26%	1.99%	3.75%	21.08%	5.09%	15.04%	-6.83%	9.69%	6.38%

CNN is primarily used for processing grid-structured data. CNN operates by performing convolutions over the local neighborhood of input data, capturing local and translational invariant features. On the other hand, RNN and LSTM are typically used for processing sequential data. They can capture temporal dependencies in the sequence. In our task, we are dealing with a feature matrix rather than a sequence of data. Therefore, employing a CNN is more suited to our requirements.

The effect of the SENet layer. Table VII shows the performance of WS compared with our approach. From this table, we find that **MGVD achieves slightly better performance than MGVD-WS in all evaluated metrics on all included datasets.** The improvements in terms of accuracy and F1-score are relatively small, i.e., about 2%, while the improvement in MCC exceeds 7%. This indicates that better modeling interdependencies between channels can improve the performance of the CNN model.

The effect of GAT. Table VII demonstrates the performance of MGVD-GCN, MGVD-GGNN and MGVD. From the table, we see that **MGVD outperforms MGVD-GCN by substantial margins and MGVD also outperforms MGVD-GGNN on all evaluation metrics.** In detail, MGVD improves GCN in terms of accuracy, precision, recall and F1-score by over 15% on average. We attribute this to the following reason: Compared to GCN, GGNN can better capture the dependencies among nodes, hence GGNN performs better than GCN. Compared to GCN and GGNN, GAT adapts the attention mechanism in a shared manner to all edges in the graph, which allows GAT to assign different importance to nodes that share the same neighbors. In contrast, GCN and GGNN treat all neighbor nodes equally during the aggregation process, which may lead to less optimal performance. According to the structure and semantics of the source code, the influence of the statement is different. Thus, GAT is more suitable for encoding program graphs.

In summary, each techniques used in MGVD is beneficial and can boost the effectiveness of MGVD.

C. RQ3: Parameter Sensitivity

1) *Experimental Setup:* Our approach employs CNN as the classifier to identify vulnerable functions. The number of convolutional layers is a key parameter in CNN, which affects the performance of CNN. MGVD uses 3 convolutional layers. To investigate the effect of different numbers of convolutional layers on our approach's performance, we compare the performance of MGVD with four variants as follows:

- **1 conv:** The CNN model adapts 1 convolutional layer.

- **2 conv:** The CNN model adapts 2 convolutional layers.
- **4 conv:** The CNN model adapts 4 convolutional layers.
- **5 conv:** The CNN model adapts 5 convolutional layers.

2) *Experimental Results:* Table VIII presents the performance of our approach with respect to the four different numbers of convolutional layers in the CNN model. From the table, we can see that the MGVD (which uses 3 convolutional layers) achieves better performance in most of the cases. Specifically, as the number of convolutional layers increases, the performance of the model first gets better and then becomes worse. The model using 3 convolutional layers, i.e., MGVD, achieves the peak performance. We attribute this to the following reasons: Too few convolutional layers make the model unable to extract complex features. Kaiming et al. have demonstrated that in the computer vision area, with the network depth increasing, accuracy gets saturated and then degrades rapidly [63]. Since the sizes of our used datasets are smaller than those of the datasets in the computer vision area, the rapid degrades appear earlier. According to these results, we think it is reasonable to set the number of convolution layers in our approach as three.

In summary, using a CNN model with three convolutional layers can achieve the best performance.

VII. DISCUSSION

A. Performance on Different Types of Vulnerabilities

To delve deeper into the performance of MGVD and the baselines in detecting various vulnerability types, we first divide the SARD test set into smaller subsets based on CWE types and evaluate the performance of MGVD and the baseline on each subset.

Fig. 9 shows the performance of MGVD and the baselines across 14 distinct CWE types. As can be seen from Fig. 9, all approaches except REVEAL perform very well in identifying CWE-758 vulnerabilities. CWE-758 refers to undefined behavior. The root cause of these vulnerabilities is accessing a variable before it is defined. Most approaches can learn the structural information of the function to track the definition and usage of variables within the function. Thus, they can effectively detect these vulnerabilities. From Fig. 9, we observe that all approaches except GraphCodeBERT and LineVul perform the worst on CWE-190 vulnerabilities. CWE-190 refers to integer overflow. It arises when erroneous calculations are performed within the function. It is challenging to accurately understand the effect and results of the numerical computations in a function without execution. Therefore, all the techniques face challenges in detecting CWE-190 vulnerabilities.

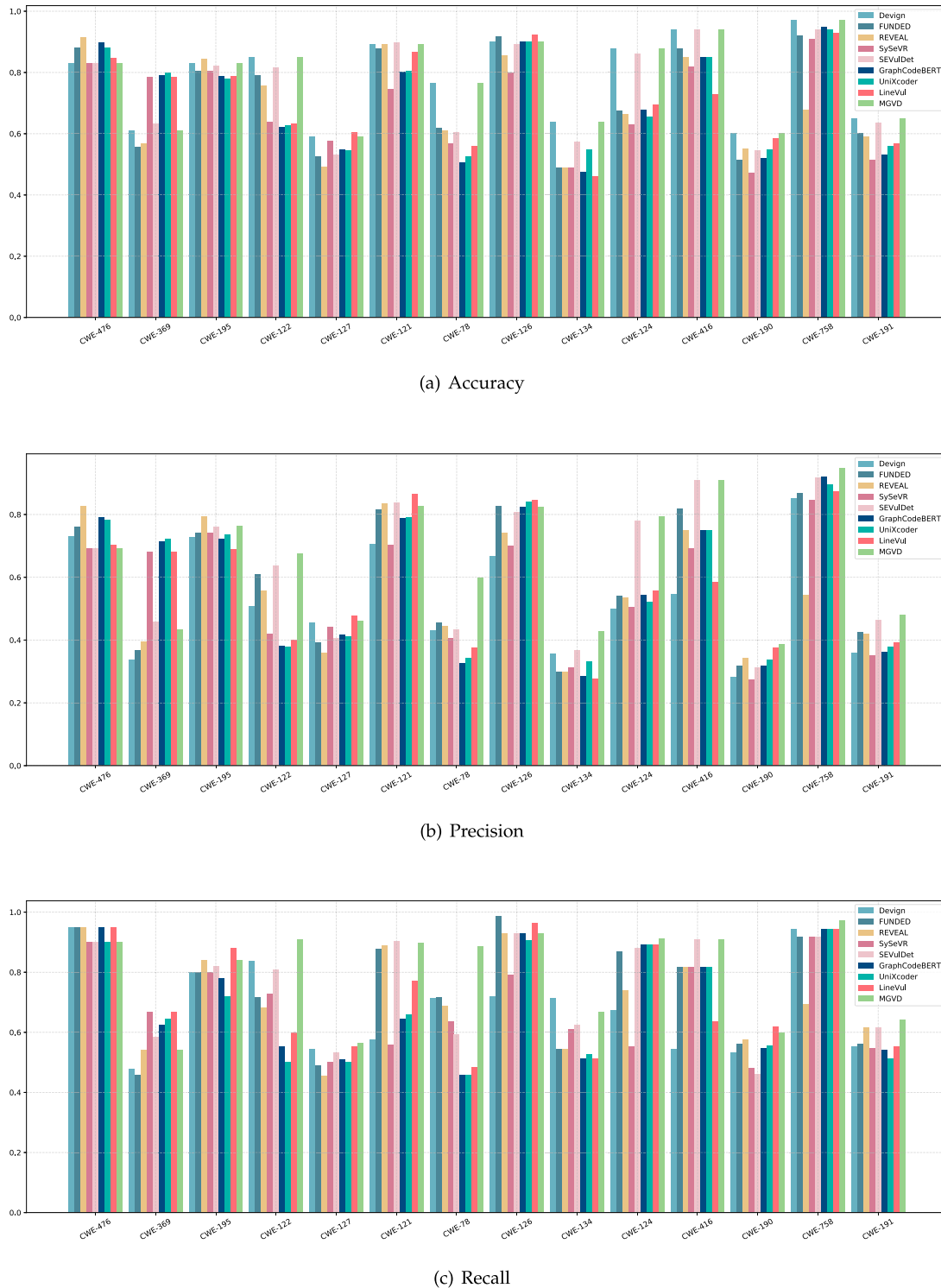
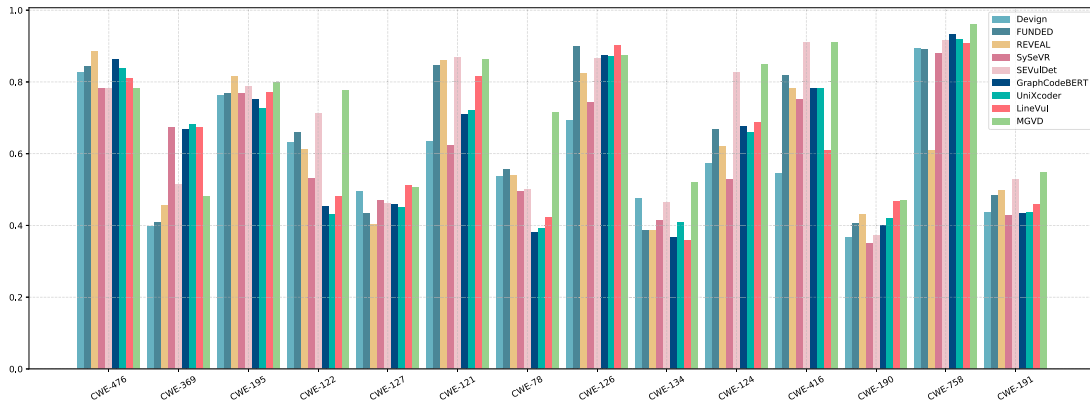


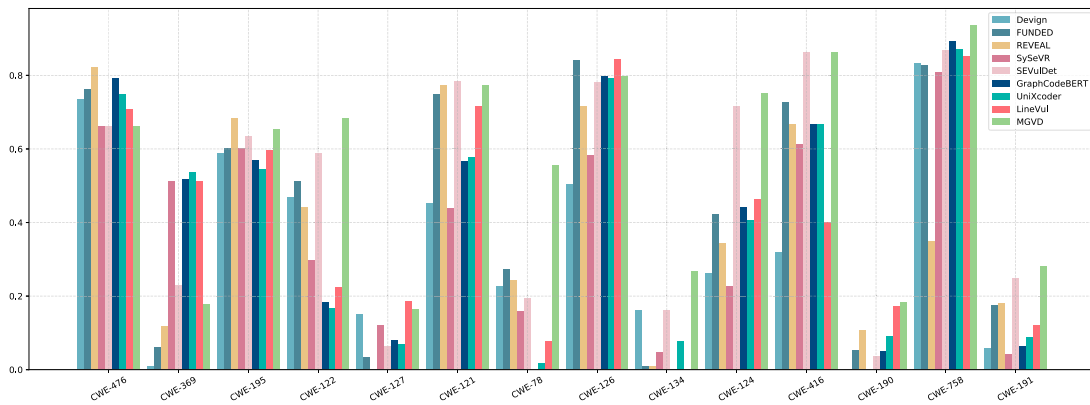
Fig. 8. The performance of MGVD and the baselines on different types of vulnerabilities in SARD test set.

For MGVD, it demonstrates considerable strength in identifying CWE-121, CWE-126, CWE-416, and CWE-758 vulnerabilities. On each of these types, MGVD achieves an accuracy exceeding 0.89, with precision, recall, F1-score all surpassing 0.8. These types of vulnerabilities are primarily introduced by the logical errors within the code, which the model can

effectively detect by learning the code's structure and content. Taking CWE-758 as an example, MGVD performs the best on this type of vulnerability, achieving an accuracy of 0.9697 and an F1-score of 0.9589. CWE-758 refers to a programming vulnerability related to relying on behavior that is not defined, specified, or consistent across different implementations or



(d) F1-score



(e) MCC

Fig. 9. The performance of MGVD and the baselines on different types of vulnerabilities in SARD test set.

platforms. Fig. 10 shows a vulnerable function with a CWE-758 type of vulnerability from the SARD dataset. The root cause of this vulnerability is related to line 9. This statement tries to access the wide character that the pointer points to. Even though memory was allocated for the pointer itself (`pointer`), no memory was allocated for the object that the pointer is meant to point to (`*pointer`). As a result, when data is assigned the value of `*pointer`, it tries to access uninitialized memory, which leads to undefined behavior. MGVD, by constructing a statement graph, especially through combining PDG and AST, can effectively track the definitions and usage of each variable. Thus, MGVD is proficient at detecting such type of vulnerabilities.

However, MGVD's performance in identifying CWE-134, CWE-190, CWE-191, and CWE-78 vulnerabilities falls short of expectations. This is primarily because the characteristics of these vulnerabilities are less conspicuous, which provides challenges for the model to learn. These vulnerabilities also involve intricate operations such as parameter value calculations and external calls, which are currently beyond the model's learning capability. Take CWE-134 as an example. CWE-134 refers to a programming vulnerability related to the improper

handling of format strings in certain programming languages. Fig. 11 shows a function with a CWE-134 vulnerability from the SARD dataset. The string data is passed as an argument to the function and used in the `SNPRINTF` function call (line 6), which could potentially introduce a format string vulnerability. Though MGVD can learn the internal call relations within a function, it is hard to capture external function calls. As a result, it is challenging to determine whether a variable originates from an external source. Thus, MGVD still holds potential for improvements in these areas.

We also conduct the same experiments on the Big-Vul dataset. Please refer to <https://sites.google.com/view/appendixformgvd/> for the experimental results and analysis.

B. Time Efficiency

We further investigate the time efficiency of MGVD for training and inference. To measure the training time, because we conduct our study on three datasets, we randomly selected 1,000 functions from all datasets to serve as our training set, and computed the time required to train the model using this training set. For each approach, we timed the training for 100 epochs, with other hyperparameters set according to its original

```

1 void CWE758_Undefined_Behavior__wchar_t_pointer_malloc_use_17_bad()
2 {
3     int j;
4     for(j = 0; j < 1; j++)
5     {
6         {
7             wchar_t ** pointer = (wchar_t **) malloc(sizeof(wchar_t *));
8             if(pointer == NULL) {exit(-1);}
9             wchar_t * data = *pointer;
10            free(pointer);
11            printWLine(data);
12        }
13    }
14 }

```

Fig. 10. A CWE-758 vulnerable function in SARD.

```

1 void badSink(map<int, wchar_t*> dataMap)
2 {
3     wchar_t * data = dataMap[2];
4     {
5         wchar_t dest[100] = L"";
6         SNPRINTF(dest, 100-1, data);
7         printWLine(dest);
8     }
9 }

```

Fig. 11. A CWE-134 vulnerable function in SARD.

paper. To measure the inference time, we similarly choose 1,000 functions at random from all the datasets to constitute our test set. We sequentially fed these functions into the model, timed each prediction, and calculated the average prediction time per function. To reduce the bias in the experimental results, we repeat the training and testing process 5 times and calculate the average time. All models are trained and tested on the same machine, which contains an Intel i9-9900k CPU and RTX 3080Ti GPU.

Table IX shows our experimental results. We observe that: (i) It takes 1657.8 seconds to train MGVD on the training set, which is neither the best nor the worst. Considering that the training process only needs to be conducted offline once, the training time cost of MGVD is acceptable. (ii) The average inference time for MGVD is 0.089 seconds, which is worse than most of the baselines. The main reason is that it takes more time for our approach to build multiple statement graphs for each function compared to the baselines which only build a single graph for each function. Considering that our approach is only 0.037 seconds slower than the fastest baseline, we argue that such time difference is acceptable in practice.

C. Handle Functions With Over 128 Statements

Our model limits the number of statements in a function to 128 statements. MGVD is trained and tested on the datasets where the functions with more than 128 statements have been excluded. To investigate the impact of such functions on the performance of MGVD, we first tested the trained MGVD on the original test sets, i.e., the test sets without removing the function with over 128 statements. Besides, we establish a variant named MGVD-Truncate. MGVD-Truncate is trained on the original datasets and handles the functions with over

TABLE IX
TIME COST ANALYSIS

Approach	Training Time /s	Inference Time /s
UniXcoder	1950.8	0.081
GraphCodeBERT	1875.6	0.075
LineVul	1731.1	0.068
REVEAL	1522.2	0.052
FUNDED	1396.3	0.058
SySeVR	1375.5	0.066
SEVulDet	1268.3	0.055
Devign	1253.9	0.072
MGVD	1657.8	0.089

TABLE X
PERFORMANCE ON THE TEST SET THAT INCLUDES THE FUNCTIONS WITH MORE THAN 128 STATEMENTS

Approach	Test Set	Accuracy	Precision	Recall	F1-score	MCC
MGVD	Big-Vul RM 128	0.8310	0.2413	0.7425	0.3642	0.3573
MGVD	Big-Vul K 128	0.8313	0.2342	0.7296	0.3546	0.3476
MGVD-Truncate	Big-Vul K 128	0.8354	0.2353	0.7344	0.3565	0.3503

128 statements through truncation. Among the three datasets we use, only the Big-Vul dataset contains the functions with more than 128 statements. Therefore, we conduct experiments on this dataset.

As shown in Table X, *Big-Vul RM 128* represents the test set with the functions exceeding 128 statements removed, while *Big-Vul K 128* stands for the test set that includes the functions with more than 128 statements. MGVD's performance is very close on both test sets, and the performance differences between MGVD-Truncate and MGVD on *Big-Vul K 128* are slight. The possible reasons for these results include: i) There are only 2% of functions that have more than 128 statements, which do not affect the performance significantly. ii) For such 2% of functions, vulnerabilities often occur within the first 128 statements, so truncating the last few statements does not make a substantial impact.

D. The Effects of Filtering Incorrect PDGs

In Section VI-A, we made two modifications to the three datasets. First, we eliminate the functions of which the PDGs cannot be correctly constructed by Joern. Second, we remove the functions with more than 128 statements. To further compare MGVD with the baselines in a public dataset, we train and evaluate them using the FFmpeg+QEMU dataset included in CodeXGLUE without any modifications. The results of which are presented in Table XI. We observe that MGVD outperforms all the baselines in terms of F1-score and Recall by at least 3.29% and 5.83%, respectively. The performance of MGVD in terms of Accuracy and MCC is comparable to that of the best-performing baseline, i.e., UniXcoder. However, MGVD is inferior to UniXcoder in terms of Precision. We find that the performance of MGVD is significantly affected by whether it can correctly generate the function graph. This is because if the function graph is generated incorrectly, the features learned by MGVD from such graph can be incorrect, making it hard to perform accurate predictions.

TABLE XI
THE PERFORMANCE OF MGVD AND THE BASELINES ON CODEXGLUE
FFmpeg+QEMU DATASET

Approach	FFmpeg+QEMU				
	Accuracy	Precision	Recall	F1-score	MCC
SySeVR	0.5406	0.5000	0.4892	0.4946	0.0737
FUNDED	0.5505	0.5106	0.5203	0.5154	0.0964
Devign	0.5600	0.5204	0.5394	0.5297	0.1167
REVEAL	0.5860	0.5504	0.5394	0.5449	0.1653
GraphCodeBERT	0.6365	0.6326	0.4980	0.5573	0.2616
SEVulDet	0.6029	0.5706	0.5474	0.5588	0.1981
LineVul	0.6497	0.6408	0.5402	0.5863	0.2895
UniXcoder	0.6665	0.6744	0.5299	0.5935	0.3243
MGVD	0.6640	0.6509	0.5793	0.6130	0.3195
Improvement	-0.38%	-3.49%	5.83%	3.29%	-1.48%

E. Limitation

Our approach still has the following limitations: i) At present, our approach is only applicable to functions where the number of statements is at most 128. With more powerful hardware, we could potentially increase the size of the feature matrix to accommodate longer functions. Additionally, we would explore the strategy of partitioning longer functions for vulnerability detection in the future. ii) Our approach is currently restricted to function-level vulnerability detection and is not suitable for vulnerabilities that involve multiple functions. We may explore linking multiple function graphs through function call relationships to construct novel representation techniques. This can ultimately lead to the capability to detect vulnerabilities involving multiple functions. iii) The accuracy of the function graph extraction can still be improved. In the future, we could consider integrating static analysis tools with deep learning to enhance the extraction of function graphs. Specifically, first, we could employ static analysis tools to generate a function graph and manually revise it. The original graph generated by static analysis and the source code serves as training data, and the manually revised graph serves as the label, thus training a graph-generation model. During application, we can extract a function graph using static analysis tools, and then employ the trained graph-generation model to revise the function graph.

VIII. THREATS TO VALIDITY

Threats to internal validity refer to errors in our experiments. For each baseline, we carefully reuse its existing implementation. The baselines we selected have publicly accessible replication packages available online, which we downloaded from GitHub. For each baseline, we follow the instructions described in its README file to replicate it, and we also use the settings and parameters in its replication package without making any modifications. The experimental results of REVEAL reproduced by us differs from the original paper, mainly due to the following reasons: As mentioned in Section V-A, we used the FFmpeg+QEMU dataset provided in CodeXGLUE, which differs from the FFmpeg+QEMU dataset used in the REVEAL paper. In addition, we eliminate the functions of which the PDGs cannot be correctly constructed by Joern in the dataset and the functions with more than 128 statements in the dataset. Sections VII-C and VII-D discuss the effects of the two preprocessing steps. However, some bugs were encountered while

extracting function graphs with certain baselines due to their use of an older version of Joern. To address this, we adjust the relevant scripts to make them compatible with the newer version of Joern. Our approach uses the static analysis tool Joern to construct PDGs and ASTs. Different versions of Joern may lead to different PDGs and ASTs. To mitigate this threat, we choose the v1.1.7 version of Joern because it has fixed many bugs in older versions. In addition, there are flaws and limitations even in the v1.1.7 version of Joern. For example, it cannot correctly analyze approximately 13% of the functions in the datasets. To mitigate this threat, we only use the functions that can be correctly handled by Joern in the datasets for evaluation.

Threats to external validity concern the generalizability of our approach. In this work, we only consider C/C++ vulnerabilities. Although the effectiveness of our approach has been demonstrated on three C/C++ vulnerability datasets, we cannot claim that our approach can achieve the same or similar performance on the datasets of other programming languages. However, it is worth mentioning that our approach can be extended to support other programming languages by normalizing source code, extracting program graphs and tokenizing source code according to the syntax and grammar of the target language.

Threats to construct validity concern the evaluation metrics used in our study. To reduce this threat, we adopt five classification metrics, i.e., accuracy, precision, recall, and F1-score, which are widely used in prior vulnerability detection studies.

IX. RELATED WORK

Our work adopts several advanced techniques from the fields of graph neural networks, natural language processing and deep learning. In this section, we discuss the related work in these fields.

A. Traditional Machine-Learning-Based Vulnerability Detection

During the rise of traditional machine learning, numerous studies employed these techniques for vulnerability detection. For example, Hovsepyan et al. implemented a methodology that leveraged Support Vector Machine (SVM) on a bag-of-words representation, which is derived from a simple tokenization of Java source code, to predict static analyzer labels. Nevertheless, the applicability of their approach was constrained, as it was only trained and evaluated on a single software repository [64]. Yamaguchi et al. proposed a system named Chucky [65] to detect vulnerability. Chucky combines machine-learning techniques with static program analysis to determine vulnerabilities. It leveraged k-Nearest neighbors and bag-of-words techniques. However, it could only detect two types of vulnerabilities and had a relatively high false-positive rate. Sadeghi et al. proposed a probabilistic rule ranking approach based on the information contained in categorized software repositories to improve the efficiency and scalability of static vulnerability analysis tools [66]. However, their approach is a supplement to static analysis tools and is therefore constrained by the inherent performance of these tools, which results in relatively low efficiency and sub-optimal detection accuracy in their method. Shar et al. proposed

a vulnerability prediction tool using hybrid program analysis and machine learning to detect SQLI and XSS vulnerabilities in PHP web-applications [67]. They tested two classification models, i.e., MLP and Logistic Regression in the study. However, they conducted tests only on small datasets and lacked experiments on a larger scale.

B. Deep-Learning-Based Vulnerability Detection

With the progression of deep learning, an increasing number of studies have begun to employ deep learning techniques for vulnerability detection. Wang et al. leveraged Deep Belief Network (DBN) to automatically learn semantic features from token vectors extracted from programs' Abstract Syntax Trees, and then utilize these features to train a defect prediction model [68]. However, their approach works at the file level and does not allow code inspectors to pinpoint the vulnerabilities related to specific code lines. Li et al. proposed a methodology which uses code gadgets generated through data dependence and leverages Bi-directional Long Short-Term Memory (BiLSTM) to detect vulnerabilities [7]. However, this approach is limited to detecting vulnerabilities related to library or API function calls. Russell et al. proposed a deep-learning based method to detect vulnerability. They leveraged feature-extraction approaches similar to those used for sentence sentiment classification with CNN and RNN for function-level source vulnerability classification [69]. Li et al. presented a lightweight-assisted vulnerability discovery method using a deep neural network called LAVDNN. LAVDNN uses function names as important semantics features for constructing a deep neural network-based classifier to distinguish functions in source code. Given a target function, not only the source code of the function but also the source code of the whole project is required by LAVDNN for vulnerability detection. There is still substantial room for improvement in the results achieved by these methods. Wu et al. proposed VulCNN [70], which leverages three different centralities to convert the source code of a function into an image. Then they used a CNN to detect vulnerabilities. Although our method also makes use of the concept of image channels like VulCNN, our key idea is different. We build three feature channels based on three types of statement feature vectors, but they transform one type of statement feature vectors into multiple feature channels based on centralities. Fu et al. proposed a transformer-based model named LineVul [59], which leverages CodeBERT and the attention scores in CodeBERT to perform line-level vulnerability detection.

C. Graph Neural Network

Recently, many attempts have been made to generalize neural networks on graph-structured data, such as social networks [12] and traffic networks [13], and use Graph Neural Network (GNN) to analyze them. GNN has also been applied in several NLP tasks. For example, Li et al. [71] used GNN in machine translation, where GNN is used to encode the syntactic structure of sentences. Liang et al. [72] applied GNN to represent word

co-occurrence and document-word relations in text classification tasks. Some researchers also have leveraged GNN to encode source code and tackle various software engineering tasks, such as clone detection [38] and code summarization [37].

D. GNN-Based Software Vulnerability Detection

Recent advances have shown that GNN and its variants are effective in function-level vulnerability detection. For example, Zhou et al. proposed a model called Devign [16], which applies a general graph neural network to detect vulnerability. Devign contains a novel convolutional component that can effectively extract useful features from the learned rich node representations for graph-level classification tasks. Cheng et al. proposed DeepWukong [73], which extracts program slices based on PDG and feeds these slices into GCN for vulnerability detection. Wang et al. proposed a GGNN-based model named FUNDED [17], which operates on the graph representations of the input function and has the capability to learn and aggregate multiple code relationships (including data, control, operation order, and operand values). Chakraborty et al. proposed a GNN-based model named REVEAL [14] for function-level vulnerability detection. REVEAL translates real-world code into a graph, then leverages representation learning techniques for model building, which makes the model can automatically learn to represent features of vulnerable and benign code in the feature space. Li et al. proposed IVDETECT [15], which is also a GNN-based model for function-level vulnerability detection. It leverages a PDG to represent a function then inputs the PDG into a GNN for classification to determine whether the function is vulnerable. Recently, Hin et al. proposed a statement-level vulnerability detection model named LineVD [20]. It leverages CFG and DFG to represent a function. It formulates statement-level vulnerability detection as a graph node classification task and uses GNN to identify whether the function is vulnerable.

Although the techniques mentioned above achieve promising performance in vulnerability detection, they still suffer from some problems. IVDETECT retains most of the semantic information by retain user-defined variables and functions name in source code, but its vulnerability detection results is easily influenced by irrelevant semantic information [19], [45]. Although, Devign, FUNDED and REVEAL choose to abstract user-defined variables and function names in source code to avoid being influenced by irrelevant semantic information, they end up losing most of the semantic information. It leads them to ignore the semantic features related to vulnerabilities. In contrast, our approach retains both the abstract and the raw versions of the function and we innovatively use three different ways to represent a function and encode such representation into a three-channel feature matrix. Thus, our approach can adequately balance learning both structural and semantic features. Besides, they represent a function as a single graph and incorporate AST nodes and edges into the graph, thereby making the graph large and complex when dealing with complex functions. Prior work has shown that it is easy for GNN to lose some detailed information when processing large graphs [22], [23]. It is difficult for these techniques to detect vulnerabilities when the vulnerable

function is complex and the vulnerability is related to some fine-grained information. In contrast, our method transforms the function graph into a sequence of sub-graphs, reducing the size of the graph and thereby capturing more detailed information that aids in vulnerability identification.

X. CONCLUSION AND FUTURE WORK

We propose a novel approach named MGVD to detect function-level vulnerabilities. MGVD leverages multiple statement graphs to represent a function, then uses Graph Neural Network to embed these graphs and fuses the text of the statement as a three-channel feature matrix, and adapts CNN to identify whether the function is a vulnerable function. Our experimental results show that MGVD outperforms the state-of-the-art approaches and achieves better results in terms of F1-score and MCC classification metrics.

In future work, we plan to improve the effectiveness of our approach by adding more information (e.g., source code history) to the model. We also plan to extend our approach to detect vulnerabilities in other programming languages (e.g., C# and Python). The deployment of deep learning models is still relatively expensive and complex. Techniques such as knowledge distillation can be explored to simplify models and facilitate deployment. We could first attempt to distill our model into a smaller one that can be deployed swiftly, and then compare the miniaturized model with the original. If the performance gap is minimal, it would validate the utility of the distilled, smaller model. MGVD utilizes three representations, i.e., raw graph, abstract graph and raw text, to represent a function. Although we have empirically shown that each representation is beneficial for vulnerability detection, it is still an open question whether we can improve MGVD by replacing the three representations with other sets of representations or adding more representations to MGVD. Future work can investigate this question by proposing more novel code representations and incorporating them into MGVD for vulnerability-related tasks. Currently, some vulnerability detection approaches use PDG to represent functions [15], [21], while others use CPG [14], [16], with both achieving good performance. Future work can explore the impact of using different graph representations of code on vulnerability detection. The detection of vulnerabilities that involve multiple functions is indeed an essential aspect of software vulnerability detection. While our current MGVD works on a function-level, its foundation in GNN provides potential avenues for expansion. The relationships between functions, which are defined by function calls, can be represented as edges in the graph. By extracting the function graphs of each individual function and linking them through these call relationships, we can create a larger, multi-function graph. This allows us to adopt the function-level approach for modeling the multi-function graph, thereby enabling the detection of vulnerabilities that span multiple functions.

ACKNOWLEDGMENT

Zhongxin Liu gratefully acknowledges the support of Zhejiang University Education Foundation Qizhen Scholar Foundation.

REFERENCES

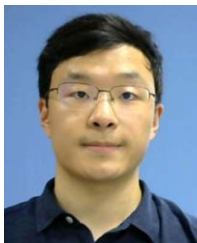
- [1] J. Jang-Jaccard and S. Nepal, "A survey of emerging threats in cyber-security," *J. Comput. Syst. Sci.*, vol. 80, no. 5, pp. 973–993, 2014.
- [2] Z. Xu, B. Chen, M. Chandramohan, Y. Liu, and F. Song, "Spain: Security patch analysis for binaries towards understanding the pain and pills," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng. (ICSE)*, Piscataway, NJ, USA: IEEE Press, 2017, pp. 462–472.
- [3] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan, "Bingo: Cross-architecture cross-OS binary search," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2016, pp. 678–689.
- [4] J. Newsome and D. X. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Proc. Netw. Distrib. Syst. Secur. (NDSS)*, vol. 5, 2005, pp. 3–4.
- [5] R. Scandariato, J. Walden, A. Hovsepian, and W. Joosen, "Predicting vulnerable software components via text mining," *IEEE Trans. Softw. Eng.*, vol. 40, no. 10, pp. 993–1006, Oct. 2014.
- [6] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *Proc. 14th ACM Conf. Comput. Commun. Secur.*, 2007, pp. 529–540.
- [7] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "VulDeePecker: A deep learning-based system for vulnerability detection," 2018, *arXiv:1801.01681*.
- [8] R. Russell et al., "Automated vulnerability detection in source code using deep representation learning," in *Proc. 17th IEEE Int. Conf. Mach. Learn. Appl. (ICMLA)*, Piscataway, NJ, USA: IEEE Press, 2018, pp. 757–762.
- [9] X. Duan et al., "VulSniper: Focus your attention to shoot fine-grained vulnerabilities," in *Proc. 28th Int. Joint Conf. Artif. Intell. (IJCAI-19)*, 2019, pp. 4665–4671.
- [10] 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 Trans. Dependable Secure Comput.*, vol. 19, no. 4, pp. 2244–2258, Jul./Aug. 2022.
- [11] Z. Li, M. Pan, Y. Pei, T. Zhang, L. Wang, and X. Li, "Empirically revisiting and enhancing automatic classification of bug and non-bug issues," *Frontiers Comput. Sci.*, vol. 18, no. 5, pp. 185–207, 2023.
- [12] Y. Liu, K. Zeng, H. Wang, X. Song, and B. Zhou, "Content matters: A GNN-based model combined with text semantics for social network cascade prediction," in *Proc. Pacific-Asia Conf. Knowl. Discovery Data Mining*, New York, NY, USA: Springer-Verlag, 2021, pp. 728–740.
- [13] K.-H. N. Bui, J. Cho, and H. Yi, "Spatial-temporal graph neural network for traffic forecasting: An overview and open research issues," *Appl. Intell.*, vol. 52, no. 3, pp. 2763–2774, 2022.
- [14] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet," *IEEE Trans. Softw. Eng.*, vol. 48, no. 9, pp. 3280–3296, Sep. 2022.
- [15] Y. Li, S. Wang, and T. N. Nguyen, "Vulnerability detection with fine-grained interpretations," in *Proc. 29th ACM Joint Meeting Eur. Softw. Eng. Conf./Symp. Found. Softw. Eng.*, 2021, pp. 292–303.
- [16] 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 *Proc. Adv. Neural Inf. Process. Syst.*, 32, 2019, pp. 10197–10207.
- [17] H. Wang et al., "Combining graph-based learning with automated data collection for code vulnerability detection," *IEEE Trans. Inf. Forensics Secur.*, vol. 16, pp. 1943–1958, 2020.
- [18] H. Zhang, Z. Li, G. Li, L. Ma, Y. Liu, and Z. Jin, "Generating adversarial examples for holding robustness of source code processing models," in *Proc. AAAI Conf. Artif. Intell.*, vol. 34, 2020, pp. 1169–1176.
- [19] H. Zhang et al., "Towards robustness of deep program processing models—Detection, estimation, and enhancement," *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, vol. 31, no. 3, pp. 1–40, 2022.
- [20] D. Hin, A. Kan, H. Chen, and M. A. Babar, "LineVD: Statement-level vulnerability detection using graph neural networks," in *Proc. 19th IEEE/ACM Int. Conf. Mining Softw. Repositories (MSR)*, (Pittsburgh, PA, USA), New York, NY, USA: IEEE Press, 2022, pp. 596–607.

- [21] S. Cao, X. Sun, L. Bo, R. Wu, B. Li, and C. Tao, "MVD: Memory-related vulnerability detection based on flow-sensitive graph neural networks," in *Proc. 44th Int. Conf. Softw. Eng.*, 2022, pp. 1456–1468.
- [22] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proc. Adv. Neural Inf. Process. Syst.*, 30, 2017, pp. 1024–1034.
- [23] W.-L. Chiang, X. Liu, S. Si, Y. Li, S. Bengio, and C.-J. Hsieh, "Cluster-GCN: An efficient algorithm for training deep and large graph convolutional networks," in *Proc. 25th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2019, pp. 257–266.
- [24] "Software assurance reference dataset," NIST Software Assurance Reference Dataset. [Online]. Available: <https://samate.nist.gov/SARD>
- [25] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "AC/C++ code vulnerability dataset with code changes and CVE summaries," in *Proc. 17th Int. Conf. Mining Softw. Repositories*, 2020, pp. 508–512.
- [26] MGVD, 2023, doi: 10.5281/zenodo.8130972.
- [27] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Proc. IEEE Symp. Secur. Privacy*, Piscataway, NJ, USA: IEEE Press, 2014, pp. 590–604.
- [28] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *Proc. Int. Conf. Mach. Learn.*, PMLR, 2014, pp. 1188–1196.
- [29] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proc. Conf. Empirical Methods Natural Lang. Process. (EMNLP)*, 2014, pp. 1532–1543.
- [30] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proc. Adv. Neural Inf. Process. Syst.* 26, 2013, pp. 3111–3119.
- [31] A. Joulin et al., "Fasttext.zip: Compressing text classification models," 2016, *arXiv:1612.03651*.
- [32] M. Pagliardini, P. Gupta, and M. Jaggi, "Unsupervised learning of sentence embeddings using compositional n-gram features," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics (NAACL)*, 2018, pp. 528–540.
- [33] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Trans. Neural Netw.*, vol. 20, no. 1, pp. 61–80, Jan. 2009.
- [34] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *Proc. 5th Int. Conf. Learn. Representations (ICLR)*, Toulon, France, 2017, pp. 1–14.
- [35] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," in *Proc. 6th Int. Conf. Learn. Representations (ICLR)*, Vancouver, BC, Canada, 2018, pp. 1–12.
- [36] Y. Li, R. Zemel, M. Brockschmidt, and D. Tarlow, "Gated graph sequence neural networks," in *Proc. Int. Conf. Learn. Representations (ICLR'16)*, 2016, pp. 1–20.
- [37] A. LeClair, S. Haque, L. Wu, and C. McMillan, "Improved code summarization via a graph neural network," in *Proc. 28th Int. Conf. Program Comprehension (ICPC'20)*, Seoul, Republic Korea. New York, NY, USA: ACM 2020, pp. 184–195.
- [38] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin, "Detecting code clones with graph neural network and flow-augmented abstract syntax tree," in *Proc. IEEE 27th Int. Conf. Softw. Anal., Evolution Reeng. (SANER)*, Piscataway, NJ, USA: IEEE Press, 2020, pp. 261–271.
- [39] Y. Lou et al., "Boosting coverage-based fault localization via graph-based representation learning," in *Proc. 29th ACM Joint Meeting Eur. Softw. Eng. Conf./Symp. Found. Softw. Eng.*, 2021, pp. 664–676.
- [40] J. Dong et al., "FIRA: fine-grained graph-based code change representation for automated commit message generation," in *Proc. 44th IEEE/ACM 44th Int. Conf. Softw. Eng. (ICSE)*, Pittsburgh, PA, USA. New York, NY, USA: ACM, 2022, pp. 970–981.
- [41] Z. Feng et al., "CodeBERT: A pre-trained model for programming and natural languages," in *Proc. Findings Assoc. Comput. Linguistics (EMNLP)*, 2020, pp. 1536–1547.
- [42] M. White, M. Tufano, C. Vendome, and D. Poshvanyk, "Deep learning code fragments for code clone detection," in *Proc. 31st IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2016, pp. 87–98.
- [43] G. Lin, S. Wen, Q.-L. Han, J. Zhang, and Y. Xiang, "Software vulnerability detection using deep neural networks: A survey," *Proc. IEEE*, vol. 108, no. 10, pp. 1825–1848, Oct. 2020.
- [44] Z. Tang, Q. Hu, Y. Hu, W. Kuang, and J. Chen, "SEVulDet: A semantics-enhanced learnable vulnerability detector," in *Proc. 52nd Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Piscataway, NJ, USA: IEEE Press, 2022, pp. 150–162.
- [45] N. Yefet, U. Alon, and E. Yahav, "Adversarial examples for models of code," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 1–30, 2020.
- [46] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," in *Proc. 54th Annu. Meeting Assoc. Comput. Linguistics (ACL)*, vol. 1, Berlin, Germany, Long Papers. The Association for Computer Linguistics, 2016, pp. 1715–1725.
- [47] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.
- [48] Y. Kim, "Convolutional neural networks for sentence classification," in *Proc. Conf. Empirical Methods Natural Lang. Process. (EMNLP)*, A. Moschitti, B. Pang, and W. Daelemans, Eds., Doha, Qatar, ACL, 2014, pp. 1746–1751.
- [49] J. Hu, L. Shen, and G. Sun, "Squeeze-and-excitation networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2018, pp. 7132–7141.
- [50] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.
- [51] G. E. Dahl, T. N. Sainath, and G. E. Hinton, "Improving deep neural networks for LVCSR using rectified linear units and dropout," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process.*, Piscataway, NJ, USA: IEEE Press, 2013, pp. 8609–8613.
- [52] "The 2021 common weakness enumeration top 25 most dangerous software weaknesses," MITRE Corporation. [Online]. Available: https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html
- [53] H. Robbins and S. Monro, "A stochastic approximation method," *Ann. Math. Statist.*, vol. 22, no. 3, pp. 400–407, Sep. 1951.
- [54] R. Mohammed, J. Rawashdeh, and M. Abdullah, "Machine learning with oversampling and undersampling techniques: Overview study and experimental results," in *Proc. 11th Int. Conf. Inf. Commun. Syst. (ICICS)*, Piscataway, NJ, USA: IEEE Press, 2020, pp. 243–248.
- [55] R. Yedida and T. Menzies, "On the value of oversampling for deep learning in software defect prediction," *IEEE Trans. Softw. Eng.*, vol. 48, no. 8, pp. 3103–3116, Aug. 2022.
- [56] S. Lu et al., "CodeXGLUE: A machine learning benchmark dataset for code understanding and generation," 2021, *arXiv:2102.04664*.
- [57] D. Guo et al., "GraphCodeBERT: Pre-training code representations with data flow," 2020, *arXiv:2009.08366*.
- [58] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "UniXcoder: Unified cross-modal pre-training for code representation," in *Proc. 60th Annu. Meeting Assoc. Comput. Linguistics*, vol. 1, Long Papers, 2022, pp. 7212–7225.
- [59] M. Fu and C. Tantithamthavorn, "LineVul: A transformer-based line-level vulnerability prediction," in *Proc. 19th Int. Conf. Mining Softw. Repositories*, 2022, pp. 608–620.
- [60] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. 4, pp. 308–320, Dec. 1976.
- [61] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *Proc. Int. Conf. Mach. Learn.*, PMLR, 2014, pp. 1188–1196.
- [62] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Philip, "A comprehensive survey on graph neural networks," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 32, no. 1, pp. 4–24, Jan. 2021.
- [63] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.
- [64] A. Hovsepian, R. Scandariato, W. Joosen, and J. Walden, "Software vulnerability prediction using text analysis techniques," in *Proc. 4th Int. Workshop Secur. Meas. metrics*, 2012, pp. 7–10.
- [65] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck, "Chucky: Exposing missing checks in source code for vulnerability discovery," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2013, pp. 499–510.
- [66] A. Sadeghi, N. Esfahani, and S. Malek, "Mining the categorized software repositories to improve the analysis of security vulnerabilities," in *Proc. Fundam. Approaches Softw. Eng.: 17th Int. Conf. (FASE), Held as Part Eur. Joint Conf. Theory Pract. Softw. (ETAPS)*, vol. 17, Grenoble, France, New York, NY, USA: Springer-Verlag, 2014, pp. 155–169.
- [67] L. K. Shar, L. C. Briand, and H. B. K. Tan, "Web application vulnerability prediction using hybrid program analysis and machine learning," *IEEE Trans. Dependable Secure Comput.*, vol. 12, no. 6, pp. 688–707, Nov./Dec. 2015.

- [68] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 297–308.
- [69] R. Russell et al., "Automated vulnerability detection in source code using deep representation learning," in *Proc. 17th IEEE Int. Conf. Mach. Learn. Appl. (ICMLA)*, Piscataway, NJ, USA: IEEE Press, 2018, pp. 757–762.
- [70] Y. Wu, D. Zou, S. Dou, W. Yang, D. Xu, and H. Jin, "VulCNN: An image-inspired scalable vulnerability detection system," in *Proc. 44th IEEE/ACM 44th Int. Conf. Softw. Eng. (ICSE)*, Pittsburgh, PA, USA. New York, NY, USA: ACM, 2022, pp. 2365–2376.
- [71] J. Bastings, I. Titov, W. Aziz, D. Marcheggiani, and K. Sima'an, "Graph convolutional encoders for syntax-aware neural machine translation," in *Proc. Conf. Empirical Methods Natural Lang. Process. (EMNLP)*, M. Palmer, R. Hwa, and S. Riedel, Eds., Copenhagen, Denmark. Association for Computational Linguistics, 2017, pp. 1957–1967.
- [72] L. Yao, C. Mao, and Y. Luo, "Graph convolutional networks for text classification," in *Proc. AAAI Conf. Artif. Intell.*, vol. 33, 2019, pp. 7370–7377.
- [73] X. Cheng, H. Wang, J. Hua, G. Xu, and Y. Sui, "DeepWukong: Statically detecting software vulnerabilities using deep graph neural network," *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, vol. 30, no. 3, pp. 1–33, 2021.



Fangcheng Qiu is currently working toward the Ph.D. degree with the College of Computer Science and Technology, Zhejiang University, China. His research interests include the intersection of software engineering and deep learning, focused on defect prediction and localization.



Zhongxin Liu (Member, IEEE) received the Ph.D. degree in computer science and technology from Zhejiang University, China, in 2021. He is currently a Distinguished Research Fellow with the College of Computer Science and Technology, Zhejiang University, China. His research interests include AI for software engineering and mining software repositories, especially helping developers understand, write, and test source code, and make informed development decisions by learning from software "big data". For more information, see <https://zhongxin-liu.github.io/>.



Xing Hu received the Ph.D. degree from the School of Electronics Engineering and Computer Science (EECS), Peking University, in 2020. She is an Associate Professor with the School of Technology, Zhejiang University. Her research interests include mining software repositories and exploring AI techniques for software engineering. Her work aims to improve software quality, increase developer productivity, and reduce development and maintenance costs.



to uncover interesting and actionable information. For more information, see <https://xin-xia.github.io/>.



Gang Chen (Member, IEEE) received the Ph.D. degree in computer science from Zhejiang University, Hangzhou, China. He is a Professor with the College of Computer Science, Zhejiang University. He is also the Director of Key Laboratory of Intelligent Computing Based Big Data of Zhejiang Province. His research interests include database management technology, intelligent computing based Big Data and massive internet systems. He is a Member of ACM and a Standing Member of the China Computer Federation Database Professional Committee.



Xinyu Wang received the bachelor's and Ph.D. degrees in computer science from Zhejiang University, in 2002 and 2007, respectively. He is currently a Professor with the College of Computer Science and Technology, Zhejiang University, China. His research interests include real-time intelligent data processing, artificial intelligence, and software engineering.