Information and Software Technology 177 (2025) 107581

Contents lists available at ScienceDirect

Information and Software Technology

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

A dual graph neural networks model using sequence embedding as graph nodes for vulnerability detection

Miaogui Ling, Mingwei Tang [*], Deng Bian, Shixuan Lv, Qi Tang

School of Computer and Software Engineering, Xihua University, Chengdu 610039, China

ARTICLE INFO

Keywords:
Vulnerability detection
Graph neural network
Sequence embedding

ABSTRACT

Context: Detecting critical to ensure software system security. The traditional static vulnerability detection methods are limited by staff expertise and perform poorly with today's increasingly complex software systems. Researchers have successfully applied the techniques used in NLP to vulnerability detection as deep learning has developed. The existing deep learning-based vulnerability detection models can be divided into sequence-based and graph-based categories. Sequence-based embedding models cannot use structured information embedded in the code, and graph-based embedding models lack effective node representations.
Objective: To solve these problems, we propose a deep learning-based method, DGVD (Double Graph Neural Network for Vulnerability Detection).
Methods: We use the sequential neural network approach to extract local semantic features of the code as nodes embedded in the control flow graph. First, we propose a dual graph neural network module (DualGNN) that consists of GCN and GAT. The altered module utilizes two different graph neural networks to obtain the global structural information of the control flow and the relationship between the nodes and fuses the two. Second, we propose a convolution-based feature enhancement module (TC-FE) that uses different convolution kernels of different sizes to capture information at different scales so that subsequent readout layers can better aggregate node information.
Results: Experiments demonstrate that DGVD outperforms existing models, obtaining 64.23% vulnerability detection accuracy on CodeXGLUE's real benchmark dataset.
Conclusion: The proposed DGVD achieves better performance than the state-of-the-art DGVD has a more effective source code feature extraction capability on real-world datasets.

1. Introduction

As Internet technology has developed rapidly, various software systems have become more popular. These systems allow people to work more efficiently, communicate better, and access information more easily. Software vulnerabilities can be exploited and lead to data breaches and other digital security incidents. It is important to pay close attention to the security of software systems. Software vulnerabilities are defects in software that could allow an attacker to gain control of a system. Because of this, organizations need to ensure that their software systems are regularly reviewed and updated with the latest security patches. When exploited by malicious attackers, software vulnerabilities put software systems at risk, resulting in significant economic and social consequences. From the vulnerability reports of Common Vulnerabilities and Exposures (CVEs) [1], the number of software vulnerabilities has grown rapidly. As a result, vulnerability detection has been a crucial part of software security.

Traditional static detection methods have some limitations, as they require expertise from the developer and use predefined matching rules. These tools perform poorly when dealing with vulnerabilities that are hard to define or unknown.

With the development of deep learning, more and more researchers are applying it to various fields, which also include vulnerability detection. Russell et al. [2] were the first to consider source code as a natural language sequence and then used a convolutional neural network to extract code semantic features for vulnerability detection. BiLSTM, a common method in NLP, was also used for vulnerability detection by Li et al. [3]. The intrinsically structured information of source code plays an important role in static analysis theory. So, some researchers have considered this and started using this structural information to detect vulnerabilities. Using the specified concerns as a starting point, Cheng et al. [4] extracted the forward and backward relationships in a program dependency graph. A vulnerability detection model was constructed using a graph neural network model in the

* Corresponding author.
E-mail address: tang4415@126.com (M. Tang).

https://doi.org/10.1016/j.infsof.2024.107581
Received 14 June 2024; Received in revised form 29 August 2024; Accepted 5 September 2024
Available online 7 September 2024

style of JK-Net, coupled with the Top-K method of pruning the graph during learning. Zhou et al. [5] used code attribute graphs and gated graph neural networks [6] for vulnerability detection. Hin et al. [7] used graph attention networks to learn program dependencies of the code supplemented with sequential features of functions for statement-level vulnerability detection tasks. However, they lack the utilization of sequence information in the source code itself.

Existing deep learning-based vulnerability detection techniques are categorized into two main types: sequence-based and graph-based. Sequence-based vulnerability detection methods treat the source code as a flattened natural language sequence that is classified using common text classification models. This lack of information leads to a poor model. While there are sequence neural models such as GraphCodeBERT [8] that attempt to incorporate graph-structured information about the code into the sequence model, the actual enhancement is not significant enough. Graph embedding-based deep learning methods can effectively utilize the structured information embedded in the source code to obtain node representations using Word2Vec [9] and Doc2Vec [10], though which often fail to effectively utilize the sequence information embedded in the nodes, thus affecting the subsequent feature extraction of the full graph.

We propose a neural network model for performing vulnerability detection tasks to address these issues. DGVD has formalized function-level source code vulnerability detection as a binary classification task that accepts a control flow graph of source code as input. DGVD uses two different graph neural networks to extract global structural information and the relationships between the nodes. In addition, we propose a feature enhancement method to improve detection further. Extensive experiments have shown that DGVD significantly outperforms existing state-of-the-art models on the benchmark defect detection dataset of CodeXGLUE [11].

In summary, the main contributions of this paper are as follows:

- We designed a sequence-based node embedding module where we introduced a pre-trained model to utilize the semantic information of source code statements.
- We constructed a dual graph neural network that can focus on both the global structural information of the control flow and the relationships between the nodes and fused the feature information extracted from both, and the results show that the performance of the model is improved.
- We design and propose a convolution-based feature enhancement module called TC-FE, which enables subsequent pooling layers to better aggregate information, and results show that it improves the model's performance.
- We propose a dual graph neural network model for vulnerability detection, DGVD. Extensive experiments show that DGVD significantly outperforms existing state-of-the-art models on the benchmark defect detection dataset of CodeXGLUE.

2. Related work

2.1. Vulnerability detection

Software vulnerabilities are vulnerabilities in a software system that may cause security problems. There is a possibility that the defect is caused by a developer's error in coding or a system configuration error, or it may be caused by an unreasonable design of the user-system interaction process. These vulnerabilities are widely found in all corners of cyberspace and threaten the security of cyberspace at all times. As long as vulnerabilities exist, they can be exploited and jeopardize today's society. Therefore, timely and effective detection of hidden vulnerabilities can protect cyberspace security.

Traditional static analysis tools mainly rely on the theory of static analysis and vulnerability matching rules formulated by security experts. Although these tools currently have good performance in vulnerability detection there are very obvious limitations. First, the effectiveness of vulnerability detection depends on the expertise and experience of security experts. Second, security experts manually formulate vulnerability matching rules, which consumes high labor and time costs. Traditional static analysis tools also perform poorly when faced with vulnerabilities that cannot be clearly defined.

With deep learning taking off in the field of natural language processing, some researchers have set their sights on the technology. Researchers have migrated natural language processing techniques to source code-related fields since there is a similarity between programming languages and natural languages. For example, CNN has been used for software defect prediction [12] and defective source code localization [13]. Software defect prediction is done using DBN [14, 15]. RNN is used for vulnerability detection [3,16], software traceability [17] and code clone detection [18].

Now, deep learning-based source code vulnerability detection techniques are mainly categorized into two types: the first is a source code vulnerability detection method based on sequential neural network models, and the second is a source code vulnerability detection method based on graph neural network models and graph-structured data.

Sequence-based vulnerability detection methods [19–22] convert code into token sequences. For example, Russell et al. [2] treat source code as natural language sequences, use Word2Vec for token embedding, and utilize CNN to extract code features for the vulnerability detection task.VulDeepecker [3] uses code gadgets as a granularity to train classifiers using a bidirectional BiLSTM network.

Graph-based vulnerability detection models [23–28] use graph data of the code as input for software vulnerability detection using GNN. Most of them utilize structured information embedded in the source code. CPGVA [29] combines multiple code property graphs and generates a CPG (Code Property Graph) for vulnerability detection. Devign [5] learns function-level code features through code attribute graphs and uses 1D convolution-based node information fusion to obtain full graph feature representations. Notably, ReGVD [30] creates a graph of relationships between tokens based on a sliding window mechanism and uses a residual graph neural network to accomplish the vulnerability detection task. This also provides a new scheme for vulnerability detection models based on graph neural networks.

Although graph-based vulnerability detection models have achieved promising results, demonstrating their superiority in utilizing the structural information of source code, current research still faces difficulties. In terms of training data, high-quality training data is crucial for the performance and generalization ability of the model, but high-quality vulnerability datasets are scarce and difficult to obtain, especially labeled data. Additionally, the graph structure of program code can be very large, containing thousands or even tens of thousands of nodes and edges. Handling such large-scale graphs poses computational and storage challenges.

Program vulnerabilities may manifest differently across various programming languages and platforms. Designing generalizable vulnerability detection models across languages and platforms is a major challenge for researchers. Additionally, many vulnerabilities may only be exposed at runtime, making static analysis insufficient. Effectively integrating dynamic analysis information into graph neural network-based vulnerability detection is also an important issue.

To address these challenges, researchers are exploring various approaches, such as developing more efficient graph representation methods, constructing more extensive and more diverse vulnerability datasets, designing more interpretable model architectures, and combining static and dynamic analysis methods.

2.2. Graph Neural Networks

Graph Neural Networks (GNN) is a class of neural network models for processing graph-structured data. Unlike traditional neural networks with regular data structures, graph neural networks specialize in irregular graph-structured data, such as social networks and knowledge graphs. Graph-structured data is a complex relational network composed of nodes and edges, where nodes represent entities and edges represent relationships between entities.

The core idea of graph neural networks is to utilize information propagation between graph nodes to enhance node feature representation. Graph neural networks were first applied to directed acyclic graphs as early as 1997 by Sperduti et al. [31]. The original concept of Grph Neural Networks was introduced by Gori et al. [32] in 2005. Scarseli et al. [33] extended and further clarified graph neural networks on classical deep learning methods such as convolutional networks and recurrent neural networks, and later Gallicchio et al. [34] further elaborated graph neural networks gradually became a popular direction of research. Mikolov et al. [35] proposed the idea and word embedding based on representation learning, Perozzi et al. [36] proposed the deep wandering algorithm in 2014, which is a graph-structured data mining algorithm combining the randomized wandering algorithm and Word2vec, and was tasked to be the first graph embedding method based on representation learning, which can efficiently vectorize the information about node features. Similar breakthroughs have been made in randomized wandering algorithm approaches and their variants, such as Node2vec proposed by Grover and Leskovec [37], LINE model proposed by Tang et al. [38], and TADW proposed by Yang et al. [39]. The success of graph embedding has promoted the research and development of graph neural networks in non-Euclidean space and has also given rise to numerous variant models with excellent performance.

The structure of graph neural networks usually consists of multiple layers, each containing operations such as node embedding, message passing, and pooling. In the node embedding operation, the features of each node are converted into a low-dimensional vector representation for learning and processing by the neural network. In the message passing operation, each node receives information from its neighboring nodes and aggregates this information into a new node representation. In the pooling operation, the node representations are merged into a representation of the entire graph to facilitate prediction for graph-level tasks.

Compared to ordinary graph neural networks, GCN is able to utilize the features of nodes and the features of their neighbors for information transfer and feature updating through graph convolution operations. This approach is able to capture the structure and dependencies between nodes to a certain extent, which leads to a better representation of the node's features. At the same time, it can retain the global structural information of the whole graph, which can better capture the relationships and dependencies between nodes. This retention of global structural information can make the model more stable and avoid overfitting. The advantages of GCN applied to vulnerability detection are mainly the powerful modeling ability, the ability to capture the relationship between nodes and edges effectively, and the overall understanding of the network topology and evolution process. In vulnerability detection, source code is typically represented as complex graph structures, such as Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), and Data Flow Graphs (DFGs). GCN is able to model this graph structure better, thus improving detection accuracy and performance. It can also interpret the weights of each node or edge, improving the interpretability and comprehensibility of vulnerability detection.

GAT [40] introduces an attention mechanism and a nonlinear feature transformation, which makes the network more flexible and interpretable, better able to capture the relationships between nodes, and with better generalization and representation capabilities. GAT uses an adaptive attention mechanism to assign different weights to each node, which can select the most useful neighbor nodes for each node to deliver information. This attention mechanism can fully utilize the information in the graph data to more accurately capture the temporal and spatial evolutionary features in the network. Meanwhile, GAT supports a multi-head attention mechanism, which can share node representations among different attention heads. This helps to improve the generalization and learning ability of the model and can better capture the complex relationships in the network. Since GAT uses the degree matrix of the adjacency matrix and the normalization matrix to handle sparse matrices, it can handle graph data of different sizes and sparsities. This makes GAT highly efficient and accurate in processing large-scale network data. GAT has obvious advantages in processing graph data, especially applied in vulnerability detection can better capture the temporal evolution characteristics and abnormal events of network flow, and improve the detection accuracy and efficiency of the model.

3. Methods

In this section, we first outline the definition of the problem. Then, we detail the overall framework of the DGVD. Finally, we detail the various parts of the model.

3.1. Problem definition

We define function-level source code vulnerability detection as a binary classification task for graphs. First the source code text set is defined as $\left\{(c_i, y_i) \mid c_i \in C, y_i \in Y\right\}_{i=1}^{N}$, where C is the set of source code, $Y = \{0, 1\}$ is the tag set of the source code, where 1 denotes the vulnerable code, 0 denotes the normal code, and N is the number of sample. We construct the control flow graph $CFG_i = (V, X, A) \in G$ for the source code c_i. Where V is the set of all nodes in a graph. $X \in R^{n\times d}$ is the feature matrix of the graph nodes, consisting of the d-dimensional vectors $x \in R^d$ of all nodes; A is an $n \times n$ adjacency matrix, where the value of A_{ij} is 1 or 0. If there is an edge between node i and node j, then the value of A_{ij} is 1; otherwise, it is 0.

Our final goal is to learn a mapping function $f : G \to Y$, and to use it to detect whether the source code is vulnerable. The mapping function f is learned by minimizing a loss function as follows:

$$\min \sum \mathcal{L}\left(f\left(cfg_i(\mathcal{V}, X, A), y_i \mid c_i\right)\right) \quad (1)$$

Where $\mathcal{L}(\cdot)$ is the cross-entropy loss function.

3.2. Framework proposed

In this subsection, we will briefly describe the overall architecture of our model. DGVD is a neural network model that uses two different graph neural networks simultaneously to extract feature information and use it for vulnerability detection. It uses sequence embedding methods to extract local semantic features of the code as node embeddings of the graph, considers control dependencies between statements, and utilizes contextual information about the statements. The overall working framework is shown in Fig. 1, and the architecture of the model part is shown in Fig. 2. We will describe the various parts of the model in detail in the subsequent parts of this section.

3.2.1. Source code preprocessing

Neural networks cannot directly accept the data format of the source code of the original function level. So first, we need to convert the original function-level source code into a data format that is acceptable to the neural network model. Here, we use the control flow graph of the source code. The Control Flow Graph (CFG) builds a preliminary diagram by considering structured control statements and then modifies the CFG with unstructured control statements. The CFG shows the order in which source code statements are executed, describing all the paths a program may take during execution and the conditions that need to be met. Fig. 3(b) shows a simple example of a CFG. We take as raw input a single function from the source code and split it into a statement-level CFG and its corresponding set of statements S. S is then input to the node embedding module. Where $S = \{s_1, s_2 \ldots s_n\}$, s_i corresponds to the node v_i in the CFG.

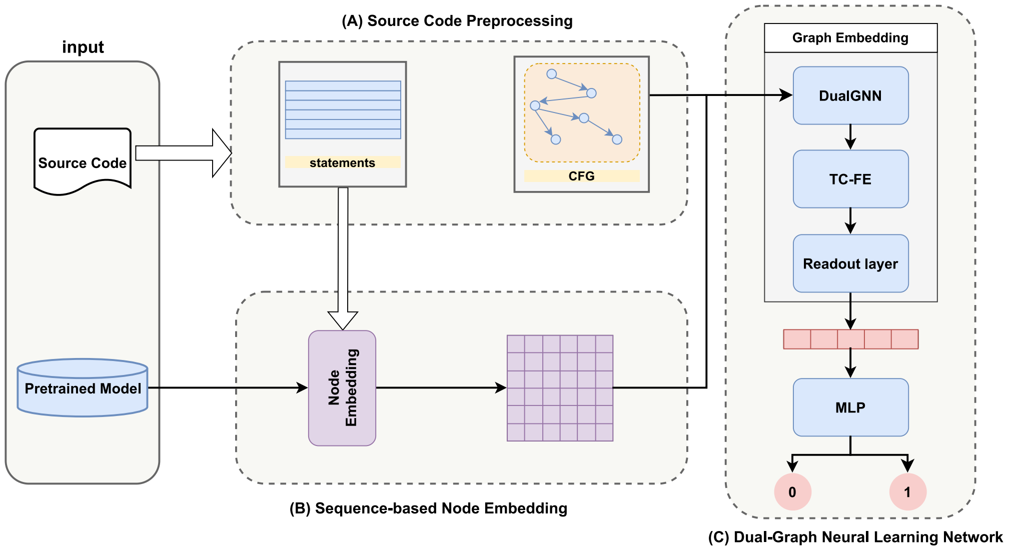


Fig. 1. The architecture of DGVD, which consists of three main parts:(A) Source Code Preproc(B) Sequence-based Node Embedding and (C) Dual-Graph Neural Learning Network.

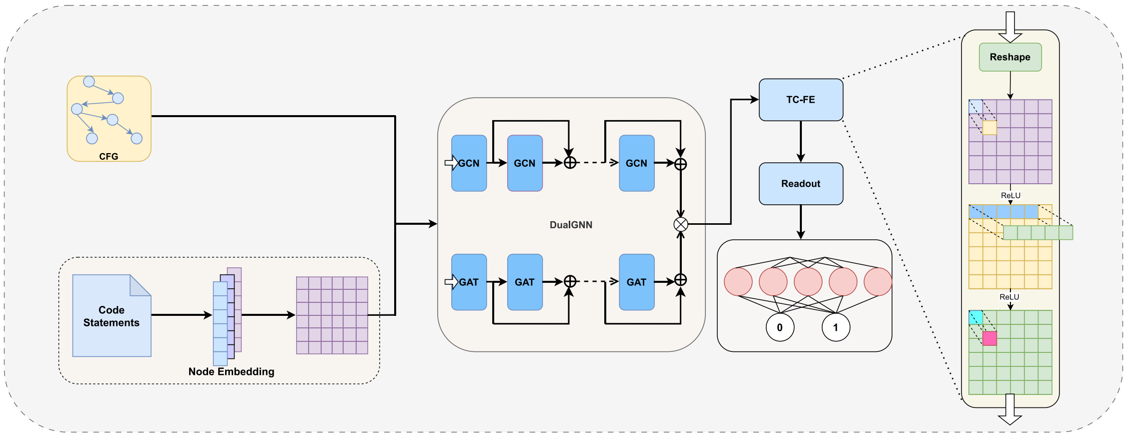
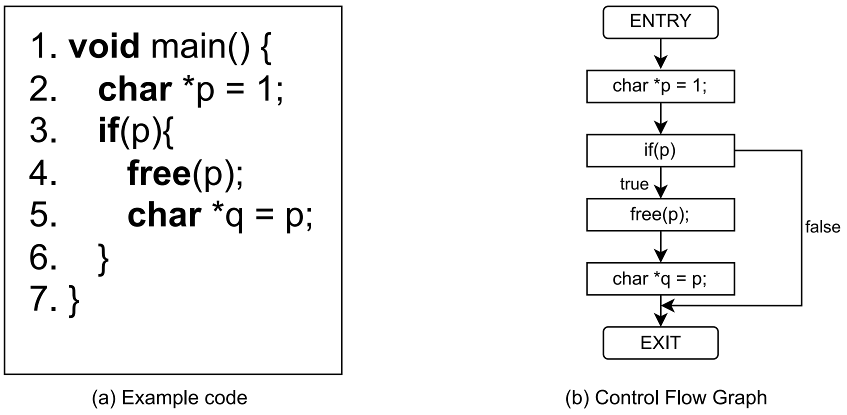


Fig. 2. Model architecture.



```
void main() {
    char *p = 1;
    if(p){
        free(p);
        char *q = p;
    }
}
```

Fig. 3. A sample code with its corresponding CFG.

Here we use Joern [41] to parse the source code and extract the control flow graph that we need.

Joern parses the statement as three different nodes in Table 1. The CFG_i obtained after Joern parsing can be very large, so a simple compression process is applied to the extracted control flow graph in this paper. The multiple node relationships parsed from a single line of code are merged into a new node v_{new}, which makes the number of nodes less than or equal to the number of lines of source code. Meanwhile, we use the original statement corresponding to the merged node as the content of the node v_{new}. In addition, v_{new} will inherit the original nodes' edge relations. This converts the original control flow graph to a statement-level control flow graph, which will significantly reduce computation time and resources.

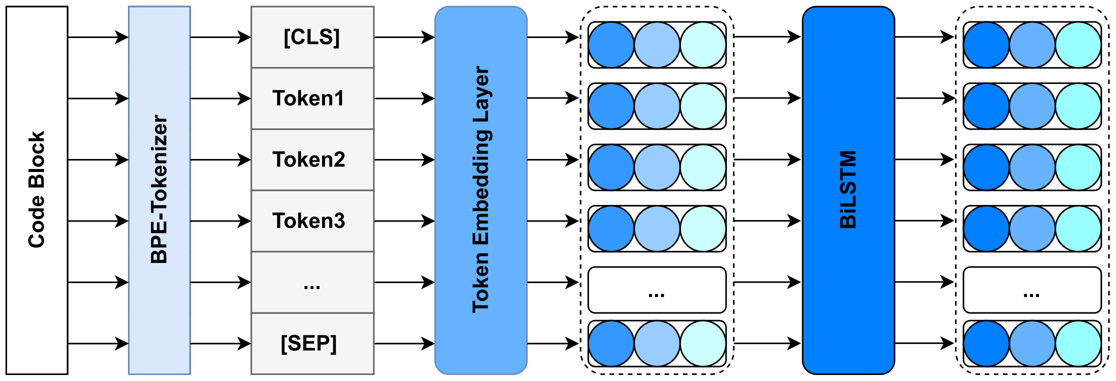


Fig. 4. Sequence-based Node Embedding.

Table 1
CFG node.

Node type	Node content
ForInit	$expression_1$
Condition	$expression_2$
PostIncDecOperationExpression	$expression_3$

3.2.2. *Sequence-based node embedding*

An informative and comprehensive code representation plays an important role in vulnerability detection. Code statements have some similarity to natural language, and using a sequence-based approach will be able to fully utilize the semantic information contained in the source code. We designed a sequence-based node embedding module that combines semantic information from source code with structured information.

First, we use the pre-trained model CodeBERT [42] trained word embedding weights to initialize the embedding layer weights. The node embedding layer is shown in Fig. 4. We employed the byte pair encoding (BPE) [43] tokenizer from CodeBERT because it can mitigate the out-of-vocabulary (OOV) problem. After S is input to the node embedding module, we first tokenized each statement in S using the pre-trained BPE tokenizer from CodeBERT.

$$\text{BPE-Tokenizer}\left(s_i\right) = \left\{CLS, Token_1, Token_2, Token_3, \ldots, Token_n, SEP\right\} \tag{2}$$

DGVD then uses the embedding layer, which has been initialized with weights, to obtain a vector representation of each token e_{ij}, and the set of vectors of tokens corresponding to the statement s_i is $E_i = \{e_{i1}, e_{i2}, e_{i3}, \ldots, e_{in}\}$.

Finally, the fusion of the local semantic information of the code is performed by BiLSTM to obtain the vector representation h_i of the utterance s_i corresponding to node v_i.

$$\text{B}iLSTM\left(E_i\right) \rightarrow \left\{\overleftarrow{h_i}, \overrightarrow{h_i}\right\} \tag{3}$$

$$h_i = Sum\left(\overleftarrow{h_i}, \overrightarrow{h_i}\right) \tag{4}$$

3.2.3. *Dual graph neural network*

In the DGVD model, we focus on the control flow's global structural information and the relationships between the nodes, for which we construct a dual graph neural network consisting of GCN and GAT. The dual graph neural network first obtains n output statement embeddings from the final output of the previous layer as well as the edges between each node. The graph structure information (including node and edge information) extracted from the source code is input to both GCN and GAT.

To prevent losing information about itself during node state updates, we add self-loop links to each node in the graph. DGVD propagates information about the state of a node and the control flow relationships between neighboring nodes in an incremental manner to improve the representation of nodes by the graph neural network. We also added residual links, defined by using jump connections on the previous GNN layer to the next GNN layer. The information is transmitted and updated as follows :

$$\text{H}_{l+1} = GNN\left(H_l, A\right) + H_l \tag{5}$$

A denotes the adjacency matrix of the graph, l represents the current number of layers, and H denotes the hidden layer state of the node. Specifically:

$$\text{H}_0 = \left\{h_1, h_2, \ldots, h_n\right\} \tag{6}$$

$$\text{H}_1 = GNN\left(H_0, A\right) \tag{7}$$

Finally, we fused the outputs of GCN and GAT to get a more informative H. Where H_{GCN} and H_{GAT} are the output of GCN and GAT, respectively. α and β is the weighting factor at the time of fusion, the sum of α and β is 1.

$$H = \alpha \cdot H_{GCN} + \beta \cdot H_{GAT} \tag{8}$$

3.2.4. *Feature enhancement module*

Yoon Kim [44] proposed multiple convolutional kernel sizes to capture different ranges of semantic features to enhance the characterization of text representation. Whereas source code shares certain similarities with natural language, some methods of natural language can also be migrated to source code related tasks. For example, Allamanis et al. [45] proposed a text summarization model based on convolution and attention mechanisms for the extreme summarization task for source code. The model utilizes convolutional operations to extract local features of the source code and focuses on key code fragments through an attention mechanism to generate a refined summary.

Convolutional operations can effectively extract localized features from the input data, and by stacking multiple convolutional layers, the model is able to capture higher-level feature representations layer by layer. And using different sizes of convolution kernels to capture information at different scales also enables the model to have better robustness.

We have designed a convolution-based feature enhancement module as shown in Fig. 2, named TC-FE and inspired by the fact that convolution is able to extract localized features in the input data by sliding a convolution kernel over the input data and calculating the result of the convolution at each position to perform the task better. In triple convolution, the first layer of convolution extracts the basic features of the input data. The second convolution layer can further extract more advanced features based on the first convolution layer. The third layer of convolution can further extract more abstract features, as well as global structures and relationships. Through this multi-level feature extraction process, the model can capture more features from the data, thus improving the model's performance.

We transform the input data and pass it sequentially into three different convolutions, nonlinearly transforming the result with each passing convolution and then feeding it into the next convolution layer. The computational steps are as follows:

$$H^1 = ReLU\left(Convolution\left(H, K_1\right)\right) \tag{9}$$

$$H^2 = ReLU\left(Convolution\left(H^1, K_2\right)\right) \tag{10}$$

$$H^3 = ReLU\left(Convolution\left(H^2, K_3\right)\right) \tag{11}$$

Where H denotes the input data and K denotes the convolution kernel.

3.2.5. Readout module

After learning the graph neural network and enhancing the feature enhancement module, we need to aggregate the final states of all nodes to obtain the final representation of the full graph. This aggregation process compresses and refines the rich information in the input feature map to obtain a more concise and representative representation of the features, thus providing better inputs to the subsequent classifier.

Mean bisimulation attention [46] is a more complex and flexible attention mechanism. This attention mechanism is able to take into account different sources of information when calculating attention weights. It is able to better capture the information in the input sequence and is commonly used in various tasks of Natural Language Processing (NLP), especially in tasks such as machine translation, text summarization, and question-answer systems.

Global attention pooling is a technique that applies the attention mechanism to a pooling operation, which enables the application of the attention mechanism over the entire input sequence or feature map, thus integrating all the information in the input sequence into a global representation. This helps the model to better understand the semantic content of the entire sequence instead of focusing only on information in localized regions. Compared to the traditional pooling approach, global attention pooling can also reduce information loss by weighting the information by the importance of each position in the whole sequence or feature map. In contrast, traditional pooling operations (e.g., maximum pooling or average pooling) tend to lose some of the information because they focus only on the most significant features or average features of a local region.

So, we design a global attention pooling readout layer based on the bisimulation attention mechanism. It is assumed that a super node exists that can play a major role in the fusion of node information on the map. First, we compute the attention score between the super node and each node and use this score as the node's weight when performing weighted aggregation of all nodes. The final full graph feature representation is the result of the weighted aggregation of all the nodes. The actual calculation process is shown in Eqs. (12) through (16).

$$h_s = \frac{\sum_{i=0}^{n} h_i}{n} \tag{12}$$

$$h_f = h_i^{\tau} \cdot W \tag{13}$$

$$e_i = h_f^{\tau} \cdot h_s + h_i^{\tau} \cdot w \tag{14}$$

$$a_i = \frac{exp(e_i)}{\sum_{j=0}^{n}(e_j)} \tag{15}$$

$$h_g = \sum_{i=0}^{n} a_i \cdot h_i^{w} \tag{16}$$

where W is a learnable weight matrix, w is a learnable weight vector, h_i is the final hidden layer state of the node, and h_g denotes the vector representation of the graph.

3.2.6. Classifier

We have used the MLP as the classification layer of DGVD to determine whether a given sample contains a vulnerability or not. The MLP is currently the best general-purpose classifier for the classification task.

$$h_{out} = softmax\left(\sigma\left(U \cdot h_g\right)\right) \tag{17}$$

$$y = argmax\left(h_{out}\right) \tag{18}$$

where U are two learnable weight matrices, $\sigma(\cdot)$ is an activation function, $y \in (0, 1)$ indicates final prediction results.

4. Experimental setting

In this section, we present information about the dataset and the relevant settings for the experiment.

4.1. Data set

We chose a widely used vulnerability dataset, the Devign dataset, from the defect detection part of the CodeXGLUE [11] project. Devign consists of two open-source C-based projects, FFMPeg and Qemu, with a total of 22k samples, 10k of which are vulnerable.

It was first collected and organized by Zhou et al. [5]. Later, Lu et al. [11] reorganized the Devign dataset and divided the training, validation, and test sets in the ratio of 8:1:1. Finally, the CodeXGLUE project includes the delineated dataset and provides it to source code processing researchers for model evaluation.

The specifics of the dataset are shown in Table 2

Table 2
Devign dataset.

	Positive	Negative	Total
Train	10 018	11 836	21 854
Valid	1187	1545	2732
Test	1255	1477	2732
Total	12 460	14 858	27 318

4.2. Experimental setting

The specific experimental environment is shown in Table 3

Table 3
Runtime environment.

Configuration name	Releases
CPU	Intel(R) Core(TM) i9-10900X
GPU	NVIDIA RTX 3090 24G
RAM	64G
Hard Drive	512G M.2 SSD
Operating System	Ubuntu 18.04 desktop workstation
Python	3.7.6
Pytorch	1.10.0
Pytorch Lighting	1.8.3
Pytorch Geometric	2.1.0

The experimental parameters were set as follows: the number of layers of the graph neural network was set to 2, the batch size was set to 128, the dropout was set to 0.5, the parameters were managed, and the model was trained at a learning rate of 5e - 4 using the Adam optimizer and the linear learning rate scheduler. We set the maximum epoch to 100 and used the early stop method to complete the training.

Table 4
DGVD performance comparison with existing sequential neural network models.

	Accuracy	Precision	Recall	F1 score
BiLSTM	59.37[a]	\	\	\
TextCNN	60.69[a]	\	\	\
RoBERTa	61.05[a]	\	\	\
CodeBERT	63.36	**65.49**	42.79	51.76
Graph CodeBERT	62.81	61.75	50.03	55.28
CodeT5	64.09	63.02	52.91	57.53
PLBART	59.99	57.01	52.51	54.67
Russell et al.	59.41	59.10	37.74	46.07
DGVD	**64.23**	61.45	**59.33**	**60.37**

[a] Denotes the result from CodeXGLUE [11].

4.3. Baselines

In this paper, we compare the performance of DGVD with eight sequence-based models, two graph-based models, and a state-of-the-art vulnerability detection method for PU learning, respectively.

Sequence-based model:

• **BiLSTM** [47]: an improved recurrent neural network (RNN) model that is able to consider both forward and backward information of a sequence.

• **TextCNN** [44]: a convolutional neural network (CNN) model for text categorization tasks, where local features in text are extracted by convolutional operations.

• **RoBERTa** [48]: a pre-trained language model based on BERT with enhanced robustness.

• **CodeBERT** [42]: a pre-trained programming language model based on BERT that uses a masked language model and replaces the token detection target.

• **Graph CodeBERT** [8]: a pre-trained programming language model that extends CodeBERT to incorporate information from code data streams into the training objectives.

• **CodeT5** [49]: CodeT5 is a pretrained model for code understanding and generation, following the same architecture as T5. It is specifically designed for handling programming languages, aiming to improve the performance of code-related tasks.

• **PLBART** [50]: a bidirectional and autoregressive transformer model that uses denoised sequence-to-sequence pre-training to exploit unlabeled data from PL and NL.

• **Russell et al.** [2]: token embedding using Word2Vec and vulnerability detection using convolutional neural networks.

Graph-based model:

• **Devign** [5]: a model based on gate graph neural networks that uses code attribute graphs as input and one-dimensional convolutional pooling for prediction.

• **ReGVD** [30]: vulnerability detection using residual graph neural network, the data used is a graph of relationships between tokens created through a sliding window mechanism.

• **Reveal** [51]: Reveal uses GGNN to extract features, and training is accomplished by multilayer perceptron and ternary loss.

Finally, we also compare it with a new and advanced method:

•**PILOT** [52]: a PU learning-based vulnerability detection model focusing on positive examples and unlabeled learning problems in software vulnerability detection.

4.4. Research question

Our evaluation was designed to answer the following five research questions:

•**RQ1**: How did the DGVD perform?

•**RQ2**: How do different token fusion methods affect model performance?

Table 5
Performance comparison of DGVD with existing graph neural network models.

	Accuracy	Precision	Recall	F1 score
Devign	58.50	55.32	55.78	54.34
ReGVD	61.71	**64.41**	37.21	47.17
Reveal	59.18	55.53	56.34	55.96
DGVD	**64.23**	61.45	**59.33**	**60.37**

•**RQ3**: How do different graph neural structures affect model performance?

•**RQ4**: How does the feature enhancement module affect the model's performance?

•**RQ5**: What effect do the hyperparameters have on the performance of DGVD?

5. Experimental results

In response to the question set forth in Section 4.4, this chapter conducts the appropriate experiments and analyzes the results of the experiments.

5.1. RQ1: how did the DGVD perform?

To answer RQ1, we compare DGVD with 12 different vulnerability detection methods on the Devign dataset for 4 performance metrics (Accuracy, Precision, Recall, F1). These include eight sequence-based vulnerability detection methods (BiLSTM, TextCNN, RoBERTa, CodeBERT, Graph CodeBERT, CodeT5, PLBART, Russell et al.), three graph-based vulnerability detection methods (Devign, ReGVD, Reveal), and the PU learning-based of PILOT. BiLSTM, TextCNN, and RoBERTa lack official code, so results from CodeXGLUE [11] are used. The other remaining models were reproduced on the same experimental equipment.

Sequence-based deep learning models are good for vulnerability detection tasks though. However, unlike natural language, the structured information of the source code contains a lot of feature information which is very important for the vulnerability extraction task. Therefore, the lack of such information in the sequence-based model leads to the lack of perfect vulnerability feature extraction, and thus limited vulnerability detection performance. As can be seen from Table 4, we achieved the highest accuracy, recall, and F1 score of 64.23%, 59.33%, and 60.37% respectively.

Compared with previous graph-based vulnerability detection models, we take the control dependencies between statements as contextual information, which makes the model better access to the contextual representation of nodes in the control flow graph during the learning and training process. Also the knowledge of weights inherited from pre-trained programming language models can well enhance the vector representation of words and strengthen the model performance.

Based on the results from Table 5, as compared to Devign, DGVD achieved 5.73% accuracy, 6.13% precision, 3.55% recall, and 6.03% F1 score improvement in the training results. On the experimental dataset, DGVD outperforms the best baseline method by 2.52%, 2.99%, and 4.41% in terms of accuracy, recall, and F1 score, respectively. In special, DGVD also achieved an improvement of 0.84% compared to the original ReGVD results provided by the ReGVD with an accuracy of 63.69%.

PILOT is a vulnerability detection model based on PU learning. PILOT first chooses the largest distance difference as the high-quality negative samples, and then continuously updates the selection of high-quality negative sample set and high-quality positive sample set by progressive fine-tuning, and utilizes the Mixed-supervision Representation Learning Module to reduce the noise interference so as to achieve the goal of vulnerability detection. Compared to PILOT, DGVD has better results in accuracy, precision (see Table 6).

Table 6
Performance comparison of DGVD with advanced deep learning models.

	Accuracy	Precision	Recall	F1 score
PILOT	63.31	57.95	**65.18**	**61.21**
DGVD	**64.23**	**61.45**	59.33	60.37

Table 7
How do different token fusion methods affect model performance?

	Accuracy	Precision	Recall	F1 score
Mean	62.04	**65.14**	37.42	47.53
Sum	60.15	60.31	36.85	45.21
Max	62.87	64.29	43.21	51.68
LSTM	61.58	59.43	51.73	55.31
BiLSTM	**64.23**	61.45	**59.33**	**60.37**

Table 8
How do different graph neural structures affect model performance?

	Accuracy	Precision	Recall	F1 score
R-GCN	63.61	60.87	58.19	59.51
R-GAT	62.15	58.31	**61.90**	60.05
GCN+GAT	63.63	60.76	58.62	59.67
R-GCN+R-GAT	**64.23**	**61.45**	59.33	**60.37**

5.2. RQ2: how do different token fusion methods affect model performance?

In our experiments, we embed code statements as nodes of the graph, and a complete code statement is a node in the graph. Therefore, before feeding it into a graph neural network for full graph feature extraction, we need to obtain a vector representation of that code statement and use it as the vector representation of the corresponding node in the graph. The vector representation of a code statement is aggregated from a vector of multiple tokens. In the experiments in this subsection, we compare five different fusion methods.

From the results of Table 7, although the "mean", "sum" and "max" methods have good performance in accuracy and precision, they can only extract the salient features in the sequence, and thus do not perform well in recall and F1 score. However, they can only extract the salient features in the sequence, and thus do not perform well in recall and F1 score. LSTM and BiLSTM take into account the semantic information of the code statements, thus improving the performance of the model on two evaluation metrics, recall and F1 score, while thereby maintaining good performance in accuracy and precision.

However, LSTM only considers the above information, while BiLSTM considers the contextual information at the same time so it performs better. We can conclude that using BiLSTM to obtain vector representations of individual code statements can further improve the performance of subsequent graph classification tasks.

5.3. RQ3: how do different graph neural structures affect model performance?

In order to verify the impact of different graph neural networks on the vulnerability detection task, we compared four different graph neural network structures. For a fair comparison, the graph neural network layers were standardized to two layers, the head of GAT was set to 8, and the other parameters of the model were kept the same.

From the experimental results as Table 8, it can be seen that the graph neural network with residual links improves in all four performance metrics compared to the original graph neural network. This also proves the effectiveness of residual links in graph neural networks. Compared with the single graph neural network, the DualGNN, which consists of residual GCN and residual GAT, achieves a lead in terms of. However, residual GAT performed well on recall and F1. However, our goal is not only to find more vulnerabilities but also to ensure that the output results have a higher confidence level. So, by combining the four performance metrics, DualGNN performs the best.

Table 9
How does the feature enhancement module affect the model's performance?.

	Accuracy	Precision	Recall	F1 score
R-GCN(w/o TC-FE)	62.27	60.11	56.25	58.11
R-GCN	**63.61**	**60.87**	**58.19**	**59.51**
R-GAT(w/o TC-FE)	61.68	57.89	60.77	59.31
R-GAT	**62.15**	**58.31**	**61.90**	**60.05**
DGVD(w/o TC-FE)	63.36	**61.45**	57.09	59.24
DGVD	**64.23**	**61.45**	**59.33**	**60.37**

Table 10
Comparison of model performance on training, validation and test sets.

	Accuracy	Precision	Recall	F1 score
Train	**64.37**	61.33	**60.37**	**60.85**
Valid	63.87	**61.54**	60.12	60.84
Test	64.23	61.45	59.33	60.37

5.4. RQ4: how does the feature enhancement module affect the performance of the model?

The aim of this subsection is to validate the effectiveness of the TC-FE module by comparing it with a model that does not use the TC-FE module. We conduct experiments using three different graph structures. The experiments demonstrate that through this multi-level feature extraction process, the model is able to learn richer and more useful representations from the data, thus improving the performance of the model. From the results in Table 9, it can be seen that the TC-FE module improves the model's accuracy by 0.87%, recall by 2.24%, and F1 score by 1.13%.

5.5. RQ5: what effect do the hyperparameters have on the performance of dgvd?

To answer RQ5, we explore the weighting of the output results when fusing GCN and GAT. In the model, we use GCN and GAT to focus on global structural information and relationships between nodes, respectively. Finally, we fused the outputs of both to obtain a more informative feature matrix as shown in Eq. (8).

Here, we are using a weighted average to fuse the outputs between two graphical nerves. Weighted averaging makes certain data points have a greater or lesser impact on the final result by taking into account the relative importance of each data point by assigning different importance or weights to different data points in the calculation process. So who has the greatest influence on the fused results, H_{GCN} or H_{GAT}, and in what ways do they mainly play a role?

Fig. 5 shows the folded lines of changes in the four performances of the model under different combinations of weights. We use dotted lines of the same color to indicate the trend of the corresponding folds. We can see an overall decreasing trend in Recall as α increases, while the opposite is true for precision. This also shows that H_{GCN} mainly affects precision for the fused results,while H_{GAT} mainly affects Recall, and our goal is to detect as many vulnerabilities as possible with high confidence. That is, we need both Precision and Recall performance metrics to be as high as possible. From Fig. 5, we can see that the relatively best results are obtained when both α and β are 0.5.

6. Threats to validity

From Table 10, we can see that there is no significant degradation in the performance of the model on the validation set compared to the performance of the model on the training set. And the performance of the model on the validation and test sets is close. This shows that the model has no overfitting and underfitting phenomenon and has good generalization ability.

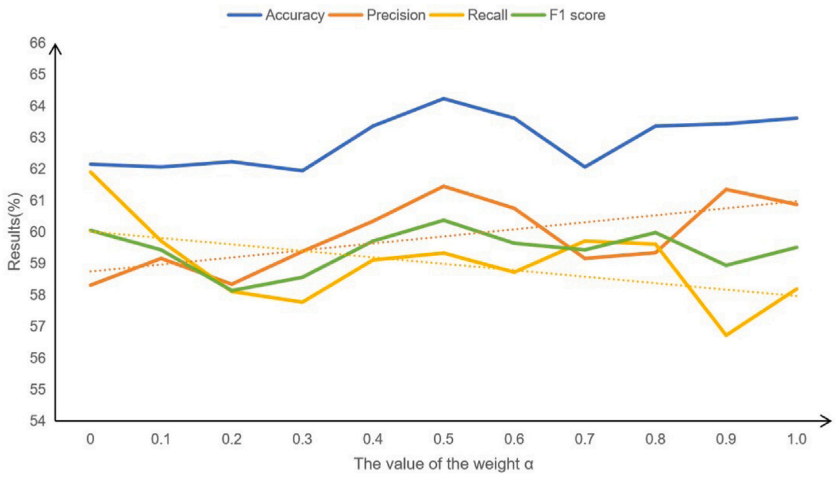


Fig. 5. Results for different weights.

But the threat to effectiveness remains. First, we use datasets mainly from two open source projects FFmpeg and QEMU. This may lead to poor generalization of the model. Second, DGVD only utilizes control flow graphs and lacks the use of structural information about the rest of the source code, which may limit its ability to detect relevant vulnerabilities. Third, the functional-level data samples lack inter-process information, so the model is unable to capture macro and inter-process call information, making it difficult to understand the potential nature of the vulnerabilities.

7. Conclusion

We view vulnerability identification as a graph binary classification problem and design a deep learning model DGVD to detect vulnerabilities in function-level source code. DGVD uses an utterance-level control flow graph of the source code as input. DGVD uses a pre-trained model to exploit the semantic information of the source code by embedding the utterances of the source code as nodes of the graph. And then a bi-graph neural network layer with residual connectivity, a feature enhancement module is utilized to learn the graph representation. To demonstrate the effectiveness of DGVD, we conducted extensive experiments on real datasets from FFmpeg and QEMU, comparing DGVD with other deep learning-based models. The experimental results show that DGVD significantly outperforms the baseline model, obtaining the highest accuracy of 64.23% on the dataset. In the future, we will continue to explore more about deep learning approaches to detect vulnerabilities.

CRediT authorship contribution statement

Miaogui Ling: Writing – original draft, Software, Methodology, Conceptualization. **Mingwei Tang:** Writing – review & editing, Supervision. **Deng Bian:** Visualization, Data curation. **Shixuan Lv:** Investigation. **Qi Tang:** Validation.

Declaration of competing interest

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

Data availability

The data that has been used is confidential.

Acknowledgments

This work is supported by the Scientific Research Funds project of Science and Technology Department of Sichuan Province (Nos. 2019YFG0508, 2019GFW131, 2024JY**, 24NSFJQ0015), Sichuan Key R&D project (No. 2023YFG0354), the National Natural Science Foundation of China (No. 61902324), Funds Project of Chengdu Science and Technology Bureau, China (Nos. 2017-RK00-00026-ZF, 2022-YF04-00065-JH, 2023-JB00-00020-GX), the Xihua University Education and teaching reform project (No: xjjg2021049, xjjg2021115), Science and Technology Planning Project of Guizhou Province (Nos. QianKeHeJiChu-ZK[2021]YiBan319) and Xihua University Science and Technology Innovation Competition Project for Postgraduate Students (Nos: YK2024 0133, YK20240137, YK20240143)

References

[1] National Vulnerability Database, American Information Technology Laboratory, 2023, URL https://www.nist.gov/.
[2] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, M. McConley, Automated vulnerability detection in source code using deep representation learning, in: 2018 17th IEEE International Conference on Machine Learning and Applications, ICMLA, IEEE, 2018, pp. 757–762.
[3] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, Y. Zhong, Vuldeepecker: A deep learning-based system for vulnerability detection, 2018, arXiv preprint arXiv:1801.01681.
[4] X. Cheng, H. Wang, J. Hua, G. Xu, Y. Sui, Deepwukong: Statically detecting software vulnerabilities using deep graph neural network, ACM Trans. Softw. Eng. Methodol. (TOSEM) 30 (3) (2021) 1–33.
[5] Y. Zhou, S. Liu, J. Siow, X. Du, Y. Liu, Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks, Adv. Neural Inf. Process. Syst. 32 (2019).
[6] Y. Li, D. Tarlow, M. Brockschmidt, R. Zemel, Gated graph sequence neural networks, 2015, arXiv preprint arXiv:1511.05493.
[7] D. Hin, A. Kan, H. Chen, M.A. Babar, Linevd: Statement-level vulnerability detection using graph neural networks, in: Proceedings of the 19th International Conference on Mining Software Repositories, 2022, pp. 596–607.
[8] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S.K. Deng, C.B. Clement, D. Drain, N.S. Sundaresan, J. Yin, D. Jiang, M. Zhou, GraphCodeBERT: Pre-training code representations with data flow, in: 9th International Conference on Learning Representations, OpenReview, Virtual Event, Austria, 2021.
[9] T. Mikolov, I. Sutskever, K. Chen, G.S. Corrado, J. Dean, Distributed representations of words and phrases and their compositionality, Adv. Neural Inf. Process. Syst. 26 (2013).
[10] Q. Le, T. Mikolov, Distributed representations of sentences and documents, in: International Conference on Machine Learning, PMLR, 2014, pp. 1188–1196.
[11] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S.K. Deng, S. Fu, S. Liu, CodeXGLUE: A machine learning benchmark dataset for code understanding and generation, in: Thirty-Fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1), 2021.

[12] J. Li, P. He, J. Zhu, M.R. Lyu, Software defect prediction via convolutional neural network, in: 2017 IEEE International Conference on Software Quality, Reliability and Security, QRS, IEEE, 2017, pp. 318–328.
[13] X. Huo, M. Li, Z.-H. Zhou, et al., Learning unified features from natural and programming languages for locating buggy source code, in: IJCAI, Vol. 16, No. 2016, 2016, pp. 1606–1612.
[14] S. Wang, T. Liu, L. Tan, Automatically learning semantic features for defect prediction, in: Proceedings of the 38th International Conference on Software Engineering, 2016, pp. 297–308.
[15] S. Wang, T. Liu, L. Tan, Automatically learning semantic features for defect prediction, in: Proceedings of the 38th International Conference on Software Engineering, 2016, pp. 297–308.
[16] G. Lin, J. Zhang, W. Luo, L. Pan, Y. Xiang, POSTER: Vulnerability discovery with function representation learning from unlabeled projects, in: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, 2017, pp. 2539–2541.
[17] J. Guo, J. Cheng, J. Cleland-Huang, Semantically enhanced software traceability using deep learning techniques, in: 2017 IEEE/ACM 39th International Conference on Software Engineering, ICSE, IEEE, 2017, pp. 3–14.
[18] M. White, M. Tufano, C. Vendome, D. Poshyvanyk, Deep learning code fragments for code clone detection, in: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, 2016, pp. 87–98.
[19] H.K. Dam, T. Tran, T. Pham, S.W. Ng, J. Grundy, A. Ghose, Automatic feature learning for vulnerability prediction, 2017, arXiv preprint arXiv:1708.02368.
[20] F. Wu, J. Wang, J. Liu, W. Wang, Vulnerability detection with deep learning, in: 2017 3rd IEEE International Conference on Computer and Communications, ICCC, IEEE, 2017, pp. 1298–1302.
[21] X. Li, L. Wang, Y. Xin, Y. Yang, Y. Chen, Automated vulnerability detection in source code using minimum intermediate representation learning, Appl. Sci. 10 (5) (2020) 1692.
[22] C. Wang, Y. Yang, C. Gao, Y. Peng, H. Zhang, M.R. Lyu, No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence, in: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2022, pp. 382–394.
[23] Y. Wu, D. Zou, S. Dou, W. Yang, D. Xu, H. Jin, Vulcnn: An image-inspired scalable vulnerability detection system, in: Proceedings of the 44th International Conference on Software Engineering, 2022, pp. 2365–2376.
[24] X. Duan, J. Wu, S. Ji, Z. Rui, T. Luo, M. Yang, Y. Wu, VulSniper: Focus your attention to shoot fine-grained vulnerabilities, in: IJCAI, 2019, pp. 4665–4671.
[25] H. Wang, G. Ye, Z. Tang, S.H. Tan, S. Huang, D. Fang, Y. Feng, L. Bian, Z. Wang, Combining graph-based learning with automated data collection for code vulnerability detection, IEEE Trans. Inf. Forensics Secur. 16 (2020) 1943–1958.
[26] S. Cao, X. Sun, L. Bo, Y. Wei, B. Li, Bgnn4vd: Constructing bidirectional graph neural-network for vulnerability detection, Inf. Softw. Technol. 136 (2021) 106576.
[27] X.-C. Wen, Y. Chen, C. Gao, H. Zhang, J.M. Zhang, Q. Liao, Vulnerability detection with graph simplification and enhanced graph representation learning, in: 2023 IEEE/ACM 45th International Conference on Software Engineering, ICSE, IEEE, 2023, pp. 2275–2286.
[28] X.-C. Wen, C. Gao, J. Ye, Y. Li, Z. Tian, Y. Jia, X. Wang, Meta-path based attentional graph learning model for vulnerability detection, IEEE Trans. Softw. Eng. (2023).
[29] W. Xiaomeng, Z. Tao, W. Runpu, X. Wei, H. Changyu, CPGVA: Code property graph based vulnerability analysis by deep learning, in: 2018 10th International Conference on Advanced Infocomm Technology, ICAIT, IEEE, 2018, pp. 184–188.
[30] V.-A. Nguyen, D.Q. Nguyen, V. Nguyen, T. Le, Q.H. Tran, D. Phung, Regvd: Revisiting graph neural networks for vulnerability detection, in: Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings, 2022, pp. 178–182.
[31] A. Sperduti, A. Starita, Supervised neural networks for the classification of structures, IEEE Trans. Neural Netw. 8 (3) (1997) 714–735.
[32] M. Gori, G. Monfardini, F. Scarselli, A new model for learning in graph domains, in: Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005, Vol. 2, IEEE, 2005, pp. 729–734.
[33] F. Scarselli, M. Gori, A.C. Tsoi, M. Hagenbuchner, G. Monfardini, The graph neural network model, IEEE Trans. Neural Netw. 20 (1) (2008) 61–80.
[34] C. Gallicchio, A. Micheli, Graph echo state networks, in: The 2010 International Joint Conference on Neural Networks, IJCNN, IEEE, 2010, pp. 1–8.
[35] T. Mikolov, K. Chen, G. Corrado, J. Dean, Efficient estimation of word representations in vector space, 2013, arXiv preprint arXiv:1301.3781.
[36] B. Perozzi, R. Al-Rfou, S. Skiena, Deepwalk: Online learning of social representations, in: Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2014, pp. 701–710.
[37] A. Grover, J. Leskovec, node2vec: Scalable feature learning for networks, in: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2016, pp. 855–864.
[38] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, Q. Mei, Line: Large-scale information network embedding, in: Proceedings of the 24th International Conference on World Wide Web, 2015, pp. 1067–1077.
[39] C. Yang, Z. Liu, D. Zhao, M. Sun, E.Y. Chang, Network representation learning with rich text information, in: IJCAI, Vol. 2015, 2015, pp. 2111–2117.
[40] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Lio, Y. Bengio, et al., Graph attention networks, stat 1050 (20) (2017) 10–48550.
[41] Joern - The bug Hunter's workbench, 2023, URL https://joern.io/.
[42] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, et al., Codebert: A pre-trained model for programming and natural languages, 2020, arXiv preprint arXiv:2002.08155.
[43] R. Sennrich, B. Haddow, A. Birch, Neural machine translation of rare words with subword units, 2015, arXiv preprint arXiv:1508.07909.
[44] Y. Kim, Convolutional neural networks for sentence classification, 2014, pp. 1746–1751, arXiv:1408.5882.
[45] M. Allamanis, H. Peng, C. Sutton, A convolutional attention network for extreme summarization of source code, in: International Conference on Machine Learning, PMLR, 2016, pp. 2091–2100.
[46] J. Yu, B. Bohnet, M. Poesio, Named entity recognition as dependency parsing, 2020, arXiv preprint arXiv:2005.07150.
[47] S. Hochreiter, J. Schmidhuber, Long short-term memory, Neural Comput. 9 (8) (1997) 1735–1780.
[48] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, V. Stoyanov, Roberta: A robustly optimized bert pretraining approach, 2019, arXiv preprint arXiv:1907.11692.
[49] Y. Wang, W. Wang, S. Joty, S.C. Hoi, CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, in: Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, 2021, pp. 8696–8708.
[50] W.U. Ahmad, S. Chakraborty, B. Ray, K.-W. Chang, Unified pre-training for program understanding and generation, in: Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Association for Computational Linguistics, 2021, pp. 2655–2668.
[51] S. Chakraborty, R. Krishna, Y. Ding, B. Ray, Deep learning based vulnerability detection: Are we there yet?, vol. 48, 2022, pp. 3280–3296.
[52] X.-C. Wen, X. Wang, C. Gao, S. Wang, Y. Liu, Z. Gu, When less is enough: Positive and unlabeled learning model for vulnerability detection, in: 2023 38th IEEE/ACM International Conference on Automated Software Engineering, ASE, IEEE, 2023, pp. 345–357.