The Journal of Systems & Software 199 (2023) 111623

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

CSGVD: A deep learning approach combining sequence and graph embedding for source code vulnerability detection ☆

Wei Tang [a], Mingwei Tang [a,*], Minchao Ban [a], Ziguo Zhao [a], Mingjun Feng [b]

[a] School of Computer and Software Engineering, Xihua University, Chengdu, 610039, Sichuan, China
[b] State Grid Tibet Electric Power Co., Ltd., China

ARTICLE INFO

Article history:
Received 30 July 2022
Received in revised form 20 December 2022
Accepted 19 January 2023
Available online 31 January 2023

Dataset link: https://github.com/microsoft/CodeXGLUE

Keywords:
Graph neural networks
Vulnerability detection
Sequence embedding
Graph embedding
Pre-trained language model
Attention pooling

ABSTRACT

In order to secure software, it is critical to detect potential vulnerabilities. The performance of traditional static vulnerability detection methods is limited by predefined rules, which rely heavily on the expertise of developers. Existing deep learning-based vulnerability detection models usually use only a single sequence or graph embedding approach to extract vulnerability features. Sequence embedding-based models ignore the structured information inherent in the code, and graph embedding-based models lack effective node and graph embedding methods. As a result, we propose a novel deep learning-based approach, CSGVD (Combining Sequence and Graph embedding for Vulnerability Detection), which considers function-level vulnerability detection as a graph binary classification task. Firstly, we propose a PE-BL module, which inherits and enhances the knowledge from the pre-trained language model. It extracts the code's local semantic features as node embedding in the control flow graph by using sequence embedding. Secondly, CSGVD uses graph neural networks to extract the structured information of the graph. Finally, we propose a mean biaffine attention pooling, M-BFA, to better aggregate node information as a graph's feature representation. The experimental results show that CSGVD outperforms the existing state-of-the-art models and obtains 64.46% accuracy on the real-world benchmark dataset from CodeXGLUE for vulnerability detection.

1. Introduction

Modern software systems are often plagued by a wide variety of software vulnerabilities. From the vulnerability reports of the Common Vulnerabilities and Exposures (CVEs) (Anon, 2022a) in recent years, the number of software vulnerabilities has been increasing rapidly. With the increase in vulnerabilities, cybersecurity attacks will have more possibilities, resulting in serious economic and societal harm. So vulnerability detection is essential for ensuring the security of software systems and protecting social and economic stability.

Traditional static detection approaches (Anon, 2022c,b,d,f,i) were based on static analysis theory and pre-defined matching rules, which relied excessively on the expertise of developers and were difficult to detect unknown vulnerabilities. As a result, they often perform poorly in the real-world application environment.

Since deep learning techniques have become more advanced, researchers have increasingly focused on using deep learning techniques to detect vulnerabilities. Russell et al. (2018) regarded the source code as a natural language sequence and then used CNNs (Convolutional Neural Networks) to extract code semantic features for vulnerability detection. Li et al. (2018) employed the code gadget as input to train a BiLSTM (Hochreiter and Schmidhuber, 1997) network for vulnerability detection. Cheng et al. (2021) first generated the PDG by considering the control and data dependence. Then they conducted forward and backward slicing starting from a program point of interest (system API calls or arithmetic operators) to produce its corresponding XFG, a subgraph of PDG (Program Dependency Graph). Finally, they used the XFG as input to train a slice-level vulnerability model. Nguyen et al. (2022) created the relationship graphs based on a sliding window between tokens to conduct vulnerability detection tasks by using a residual graph neural network. Zhou et al. (2019) used the code property graph and gated graph neural network (Li et al., 2016) to perform vulnerability detection.

Existing deep learning approaches based on sequence embedding treat source code as a flat natural language sequence and apply natural language common approaches (e.g., RNN, TextCNN (Kim, 2014), BiLSTM (Hochreiter and Schmidhuber, 1997), BERT (Devlin et al., 2019), etc.) for vulnerability detection. Because these methods lack the inherent structure of source code (e.g., control flow graph, data flow graph, abstract syntax tree), they

☆ Editor: Dr. Alexander Serebrenik.

* Corresponding author.
E-mail addresses: tangwei@stu.xhu.edu.cn (W. Tang), tang4415@126.com (M. Tang), bmc@stu.xhu.edu.cn (M. Ban), ziguocs@gmail.com (Z. Zhao), fmj-520530@163.com (M. Feng).

https://doi.org/10.1016/j.jss.2023.111623
0164-1212/

perform poorly. Deep learning approaches based on graph embedding use Word2Vec (Mikolov et al., 2013), Doc2Vec (Le and Mikolov, 2014) to obtain node representations. Due to out-of-vocabulary problems, these approaches have many drawbacks in token embedding. Also, the existing approaches tend to ignore the code semantic information present within the statements of a node.

In this paper, we propose a simple yet effective neural network model, named CSGVD, to conduct vulnerability detection tasks. CSGVD formalizes the vulnerability detection for source code at the function level as a binary graph classification task, which accepts the control flow graph of the source code as input. Furthermore, we propose a novel efficient way of node embedding and graph embedding. Extensive experiments show that CSGVD significantly outperforms the existing state-of-the-art models on the benchmark defect detection dataset from CodeXGLUE (Lu et al., 2021).

In summary, this paper makes the following contributions:

- We propose a simple yet effective graph neural network model with residual connectivity, CSGVD. Extensive experiments show that CSGVD significantly outperforms the existing state-of-the-art models on the benchmark defect detection dataset from CodeXGLUE.
- We propose a PE-BL module instead of Word2Vec and Doc2Vec for node embedding, which uses the word embedding layer weights of a pre-trained language model to initialize the embedding layer of CSGVD and a BiLSTM to aggregate the local semantic information within a node.
- We propose the mean biaffine attention pooling (M-BFA) for graph embedding, which is inspired by the biaffine attention mechanism (Yu et al., 2020a). The results show that it can improve the model's performance.

2. The CSGVD framework

In this section, firstly, we summarize the problem definition. Then, we present the overall framework of CSGVD in detail. Finally, we demonstrate how we have incorporated the pre-trained language model and developed the graph construction method.

2.1. Problem definition

We formalize the vulnerability detection for source code at the function level as a binary graph classification task, which identifies whether a given function in raw source code is vulnerable or not. Let us define a sample of data as $\{(c_i, y_i) \mid c_i \in \mathbb{C}, y_i \in \mathbb{Y}\}_{i=1}^{N}$, where $\mathbb{C}$ is the set of raw source codes, $\mathbb{Y} = \{0, 1\}$ denotes the label sets with 1 for vulnerable and 0 otherwise, and N is the number of instances. We consider vulnerability detection as a graph classification problem and leverage the Control Flow Graph (CFG) of raw source code. Therefore, we construct the graph $cfg_i(\mathcal{V}, X, A) \in \mathcal{G}$ for each source code c_i, wherein $\mathcal{V}$ represents a set of n nodes in a graph; $X \in \mathbb{R}^{n \times d}$ is the node feature matrix, wherein each node $v_i \in \mathcal{V}$ is represented by a d-dimensional vector $\boldsymbol{x_i} \in \mathbb{R}^d$; $A \in \{0, 1\}^{n \times n}$ is the adjacency matrix. When $A_{i,j}$ equal 1, the edge between node i and node j exists, and 0 otherwise. Our goal is to learn a mapping function $f : \mathcal{G} \rightarrow \mathbb{Y}$ to determine whether a given source code is vulnerable or not. The mapping function f can be learned by minimizing the loss function with the regularization on model parameters θ as:

$$\min \sum \mathcal{L}(f(cfg_i(\mathcal{V}, X, A), y_i \mid c_i)) + \lambda \|\theta\|_2^2$$

where $\mathcal{L}(\cdot)$ is the cross-entropy loss function and λ is an adjustable weight.

Table 1
CFG Node.

Type	Content
ForInit	expression_1
Condition	expression_2
PostIncDecOperationExpression	expression_3

2.2. Approach overview

In this section, we present a combined sequence and graph embedding neural network model, named CSGVD, to identify whether a function-level code is vulnerable or not. In CSGVD, one fundamental insight is that identified control dependencies between statements could sufficiently serve as the contextual information for the function-level vulnerability detection task. Furthermore, we use the sequence embedding approach to extract local semantic features of the code as the node embedding representation of the graph. Meanwhile, we construct a graph neural network-based architecture to better leverage the structural relationships within the statement and between the statements, as shown in Fig. 1.

The overall structure is divided mainly into two parts, the feature extraction on the left and the neural network model on the right. The neural network architecture is demonstrated in Fig. 2. In the following chapters, we will describe in detail the layers of neural networks.

2.2.1. Feature extraction

To train a neural model, we need to transform the raw function-level source code into a data format acceptable to the neural network model. In our work, we use *Joern*[1] (Anon, 2022g) to parse the source code and extract the control flow graph we need. Since the graph extracted by *Joern* is too huge, we merge the extracted nodes by row. By compressing the graph and converting the original control flow graph into a statement-level control flow graph, the number of nodes in the compressed graph will be less than or equal to the number of lines of code, which will drastically reduce computing time and resources.

Example 1. When using Joern to parse a statement of sample c_i like

$$for\ (expression\_1; expression\_2; expression\_3) \quad (1)$$

Joern will split this statement into three different nodes (as shown in Table 1). It maybe makes the cfg_i corresponding to sample c_i to be very huge. To reduce the size of cfg_i, we merge these nodes located at the same line corresponding to the raw code file as a new node v_{new}. Meanwhile, we use the original statement (1) corresponding to the line as the content of node v_{new}. Furthermore, v_{new} will also inherit all the edge relations.

2.2.2. Node embedding

For subsequent tasks, it is important to generate an informative and comprehensive code representation of a given source code. As shown in Fig. 2, we propose a PE-BL module, which consists of an embedding layer and a BiLSTM layer. Specifically, instead of a traditional embedding layer, we introduce a pre-trained programming language model, CodeBERT (Feng et al., 2020), to initialize the embedding layer. We initialize self-embedding layer weights with its trained word embedding weights. In addition,

[1] *Joern* is a platform for robust analysis of source code, bytecode, and binary code. It generates code property graphs, a novel graph representation that exposes the code's syntax, control-flow, data-flow and type information in a joint data structure.

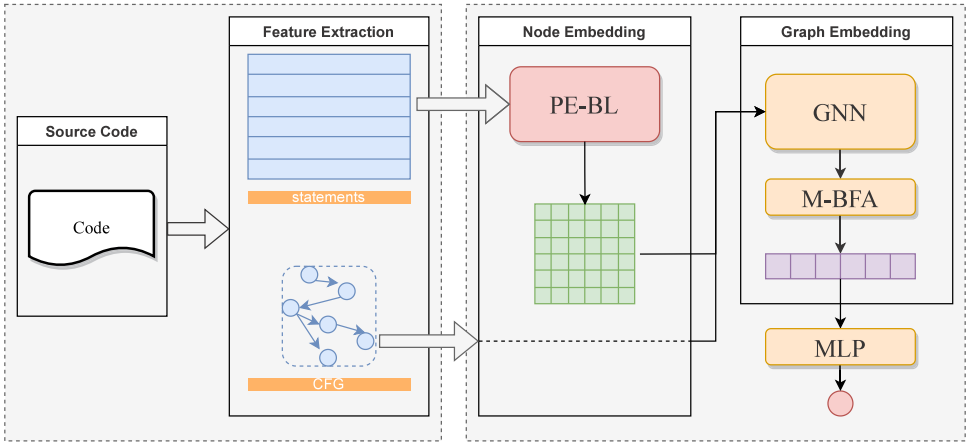

Fig. 1. Overall framework.

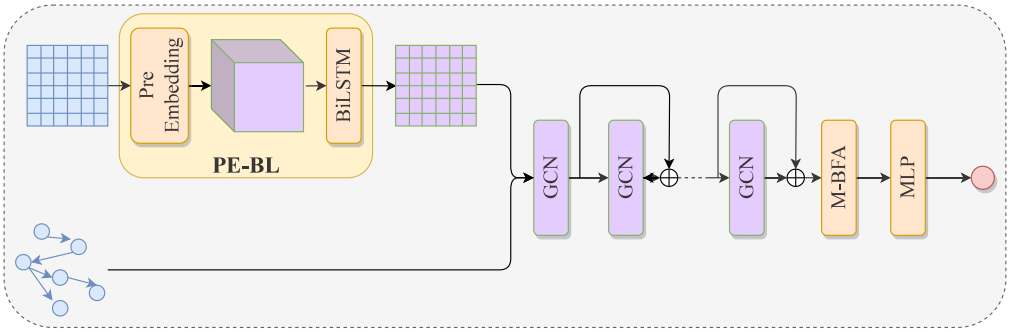

Fig. 2. Model architecture.

the byte pair encoding (BPE) (Sennrich et al., 2016) tokenizer used by CodeBERT can also mitigate the OOV problem.

We use a single function of source code as the raw input. We split the function into a collection of statements $\mathcal{S} = \{s_1, s_2, s_3, \ldots, s_n\}$ and statement-level CFG, where s_i corresponds to a node v_i of CFG. Each statement is firstly tokenized via CodeBERT's pretrained BPE tokenizer:

$$tokens_i = BPE - Tokenizer(s_i). \tag{2}$$

Then, CSGVD takes an embedding layer, which uses the word embedding layer weights of CodeBERT to initialize self weights, to obtain the vector representation $\boldsymbol{e_{ij}} \in E_i = \{\boldsymbol{e_{i1}}, \boldsymbol{e_{i2}}, \boldsymbol{e_{i3}}, \ldots, \boldsymbol{e_{in}}\}$ of each token:

$$E_i = Embedding(tokens_i). \tag{3}$$

Finally, a bidirectional LSTM is used to fuse the local semantic information of code and obtain the vector representation $\boldsymbol{x_i}$ of statement s_i corresponding to node v_i:

$$\overrightarrow{\boldsymbol{h_i}}, \overleftarrow{\boldsymbol{h_i}} = BiLSTM(E_i) \tag{4}$$

$$\boldsymbol{x_i} = Sum(\overrightarrow{\boldsymbol{h_i}}, \overleftarrow{\boldsymbol{h_i}}) \tag{5}$$

where $\overrightarrow{\boldsymbol{h_i}}, \overleftarrow{\boldsymbol{h_i}}$ is the final output of BiLSTM.

2.2.3. Residual graph neural network

In our works, we focus on the control dependency information, for which we construct a graph neural network model with residual connectivity. Unlike sequential neural networks, graph neural networks (GNNs) leverage the information diffusion mechanism to learn the graph-structured data, which updates the node states according to the graph connectivity. As shown in Fig. 2, the graph convolution network is used to learn control flow information from the graph. The GCN layer will firstly take the n output statement embedding from BiLSTM along with the edges between each node. The graph structure information of the source code, including the nodes and edges information, is extracted and fed into GCN. In particular, we add self-looping links to each node in the graph. CSGVD will propagate the information by nodes' state and the control flow relationship between neighboring nodes in an incremental manner. The model architecture consists of two graph convolution networks with residual connectivity. Overall, residual GCN is defined by its use of skip connectivity over different GCN layers in Eq. (6), wherein l indicates the current layer, $H^{(l)} = \{\boldsymbol{h_1^{(l)}}, \boldsymbol{h_2^{(l)}}, \boldsymbol{h_3^{(l)}}, \ldots, \boldsymbol{h_n^{(l)}}\}$ is the hidden state matrix of lth layer, and A is an adjacency matrix.

$$H^{(l+1)} = GCN(H^{(l)}, A) + H^{(l)} \tag{6}$$

Specially, in CSGVD,

$$H^{(0)} = X \tag{7}$$

$$H^{(1)} = GCN(H^{(0)}, A) \tag{8}$$

wherein $X = \{\boldsymbol{x_1}, \boldsymbol{x_2}, \boldsymbol{x_3}, \ldots, \boldsymbol{x_n}\}$ is the node embedding matrix.

2.2.4. Graph embedding

Using a multi-layer GNN with residual connectivity, we obtain the hidden representation of each node in the control flow graph while aggregating the node information of neighboring nodes. Before performing the graph classification task, we also need to aggregate the nodes' information to acquire the vector representation of the graph. Therefore, as shown in Fig. 3, inspired by the biaffine attention mechanism, we propose the mean biaffine attention (M-BFA) pooling. First, we assume the existence of a supernode v_s, which has a dominant role for the information aggregation of the graph. Second, we calculate the attention score of

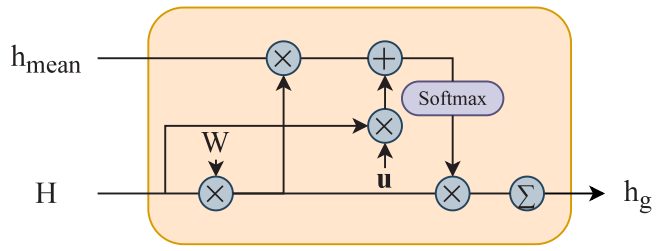

Fig. 3. M-BFA pooling.

Table 2
Dataset.

	Positive	Negative	Total
Train	10018	11836	21854
Valid	1187	1545	2732
Test	1255	1477	2732
Total	12460	14858	27318

v_s for each node. Finally, a weighted aggregation is performed for each node as a vector representation of the graph. The calculation steps are shown in the following equation:

$$\boldsymbol{h_{mean}} = \frac{\sum_{i=1}^{n} \boldsymbol{h_i}}{n} \tag{9}$$

$$\boldsymbol{h_{fi}} = \boldsymbol{h_i}^{\top} \cdot W \tag{10}$$

$$e_i = \boldsymbol{h_{fi}}^{\top} \cdot \boldsymbol{h_{mean}} + \boldsymbol{h_i}^{\top} \cdot \boldsymbol{u} \tag{11}$$

$$a_i = \frac{\exp(e_i)}{\sum_{j=0}^{n} \exp(e_j)} \tag{12}$$

$$\boldsymbol{h_g} = \sum_{i=0}^{n} a_i \cdot \boldsymbol{h_{fi}} \tag{13}$$

wherein W is a learnable weight matrix, $\boldsymbol{u}$ is a learnable weight vector, $\boldsymbol{h_i}$ is the final hidden state of node v_i, and $\boldsymbol{h_g}$ indicates the vector presentation of the graph. Specially, we choose the mean $\boldsymbol{h_{mean}}$ of all node' hidden representations as the hidden representation of supernode v_s.

2.2.5. Classifier learning

CSGVD uses a multilayer perceptron (MLP) as the classification layer, which is the best common classifier. In this work, our goal is to train a model that can learn graph representations from statement-level control flow graphs.

A two-layer MLP can be constructed as shown in Eqs. (14)–(15) to predict whether a given function is vulnerable or not.

$$\boldsymbol{o} = softmax(\sigma(U \cdot \boldsymbol{h_g}) \cdot V) \tag{14}$$

$$y = argmax(\boldsymbol{o}) \tag{15}$$

where U and V are two learnable weight matrix, $\sigma(\cdot)$ is an activation function and $y \in \{0, 1\}$ denotes the final predicted result.

3. Experiments evaluation

In this section, we will introduce the information of the dataset and the setting of our experiments. The results of the experiments will be reported in Section 4.

3.1. Dataset

We use the real-world dataset from CodeXGLUE (Lu et al., 2021) for vulnerability detection at the function level. The dataset is firstly created by Zhou et al. (2019), which includes 27318 samples from two large and popular C language open-source projects, QEMU (Anon, 2022h) and FFmpeg (Anon, 2022e). Then Lu et al. (2021) organize and re-split into the train/validation/test (8:1:1), as shown in Table 2. Fig. 4 shows the distribution of the number of nodes and edges per source code function in the training set.

3.2. Experimental setup

In this paper, we construct a 2-layer graph neural network model, set the batch size to 128 and the dropout to 0.5, and employ the Adam optimizer and a linear learning rate scheduler to train the model up to 100 epochs with the learning rate of 5e−4. In addition, the hidden size for all hidden layers, including LSTM, GNN, and MLP, is set to 768, which is equal to the dimensionality of pre-trained embedding.

We compare our model with strong and up-to-date baselines as follows:

- **BiLSTM** (Hochreiter and Schmidhuber, 1997) and **TextCNN** (Kim, 2014): two well-known models for text classification.
- **RoBERTa** (Liu et al., 2019): a pre-trained language model based on BERT (Devlin et al., 2019) by removing the next-sentence objective.
- **Code2Vec** (Alon et al., 2019; Coimbra et al., 2021): a neural network model for learning source code representation.
- **CodeBERT** (Feng et al., 2020): a pre-trained programming language model based on BERT for 6 programming languages, including Java, Python, JavaScript, PHP, Ruby and Go, using masked language (Devlin et al., 2019) model and replaced token detection objectives (Clark et al., 2020).
- **GraphCodeBERT** (Guo et al., 2021): a pre-trained programming language model, extending CodeBERT to incorporate the information of code data flow into the training objective.
- **CoTexT** (Phan et al., 2021): a transformer-based encoder–decoder model built on the same architecture as T5 (Raffel et al., 2020). It pre-trains the model with NL-PL data on the checkpoint of the original T5 model.
- **CodeT5** (Wang et al., 2021a): CodeT5 is also an encoder–decoder model following the same architecture as T5 (Raffel et al., 2020). CodeT5 proposed a novel identifier-aware pre-training task to enable the model to distinguish which code tokens are identifiers and to recover them when they are masked and a bimodal dual generation task for better NL-PL alignment.
- **PLBART** (Ahmad et al., 2021): a bidirectional and autoregressive transformer model inspired by BART (Lewis et al., 2020). It is pre-trained on an extensive collection of Java and Python functions and associated NL text via denoising autoencoding.
- Russell et al. (2018): using convolutional neural networks to extract source code features for vulnerability detection.
- **ReGVD** (Nguyen et al., 2022): a residual graph neural network by using a sliding window-based relationship between tokens.
- **Devign** (Zhou et al., 2019): a gate graph neural network-based model, which uses code property graph as input, and utilizes a 1-D convolution pooling to make a prediction.

3.3. Research questions

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

- **RQ1:** How does CSGVD perform?
- **RQ2:** How do different node embedding and initialization methods affect model performance?

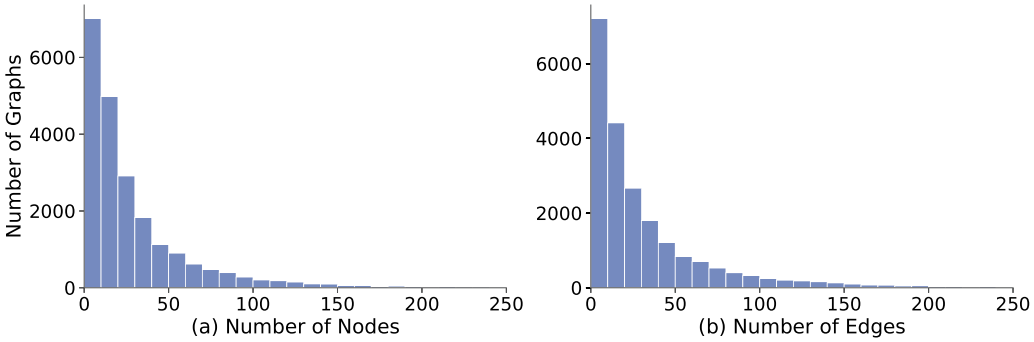

Fig. 4. Training set data distribution.

Table 3
The reproduced models in this paper and the URL of the code repository used.

Model	URL
CodeBERT	https://github.com/microsoft/CodeXGLUE/tree/main/Code-Code/Defect-detection
GraphCodeBERT	https://github.com/microsoft/CodeXGLUE/tree/main/Code-Code/Defect-detection
CoTexT	https://github.com/salesforce/CodeT5[a]
CodeT5	https://github.com/salesforce/CodeT5
PLBART	https://github.com/wasiahmad/PLBART
Russell et al.	https://github.com/VulDetProject/ReVeal
ReGVD	https://github.com/daiquocnguyen/GNN-ReGVD
Devign	https://github.com/saikat107/Devign

[a]Although the authors offer official code (https://github.com/justinphan3110/CoTexT), the code of CodeT5 performs better in our environment.

Table 4
RQ1: How does CSGVD perform?

	Accuracy	Precision	Recall	F1 score
BiLSTM	59.37[a]	/	/	/
TextCNN	60.69[a]	/	/	/
RoBERTa	61.05[a]	/	/	/
Code2Vec	62.48[a]	/	/	/
CodeBERT	63.36	**65.49**	42.79	51.76
GraphCodeBERT	62.81	61.75	50.03	55.28
CoTexT	63.84	62.10	54.58	58.10
CodeT5	64.09	63.00	52.91	57.51
PLBART	59.99	57.01	52.51	54.67
Russell et al.	59.40	59.10	37.74	46.07
ReGVD	61.71	64.41	37.21	47.17
Devign	60.05	59.79	39.92	47.87
CSGVD	**64.46**	61.87	**58.99**	**60.39**

[a]The result is from CodeXGLUE Leaderboard. https://microsoft.github.io/CodeXGLUE/.

- **RQ3:** How do different token fusion methods affect model performance?
- **RQ4:** How do different GNN models affect model performance?
- **RQ5:** How do different graph embedding methods affect model performance?

4. Results

Based on the five questions raised in Section 3.3, this section presents experimental results corresponding to the analysis. In the table below, the average score of 10 experiments is used for each experimental result. Our experiments are conducted on a Ubuntu 18.04 desktop workstation with an NVIDIA RTX 3090 GPU and an Intel(R) Core(TM) i9-10900X CPU operating at 3.70 GHz. Table 3 displays reproduced models in this paper and the URL of the code repository used.

4.1. RQ1: How does CSGVD perform?

In order to explore the performance of CSGVD, we compare it with two types of models based on sequence embedding and graph embedding respectively. According to the research of. Zeng et al. (2022), they found that the experimental results published in the CodeXGLUE Leaderboard were not precisely reproduced. For a fair comparison, this paper reproduces some of the models mentioned by Zeng et al. (2022). The experimental results are shown in the Table 4. In the above table, there are nine models based on sequence embedding: BiLSTM (Hochreiter and Schmidhuber, 1997), TextCNN (Kim, 2014), RoBERTa (Liu et al., 2019), CodeBERT (Feng et al., 2020), Russell (Russell et al., 2018), CoTexT (Phan et al., 2021), CodeT5 (Wang et al., 2021a), PLBART (Ahmad et al., 2021) and GraphCodeBERT (Guo et al., 2021). These sequence embedding-based approaches cannot effectively utilize the structured information of the code, such as control flow information, data flow information, etc. Insufficient vulnerability features are obtained due to the absence of structured information, resulting in poor prediction performance. In contrast, CSGVD uses PE-BL modules and graph neural networks to preserve the sequence information in the code as well as to better incorporate its inherent control flow information. CSGVD splits the code sequence c_i into multiple local blocks $\mathcal{S}$ based on the control flow information, and each block acts as a node in the control flow graph cfg_i. Inside the local blocks, CSGVD aggregates local context information using PE-BL, which preserves the sequence information within the local blocks. The graph neural network is then used to perform selective global information aggregation along the control flow direction, instead of every token aggregating information from all other tokens as in the previously mentioned model. As a result, CSGVD achieves higher accuracy, recall and f1 score which are 64.46%, 58.99% and 60.39%, respectively.

In terms of the two graph embedding-based models in the table, Devign incorporates code property graph information, which includes control flow and data flow information. It uses Word2Vec and the average of tokens to obtain the vector representation of nodes. The local semantic information of code within the node is

Table 5
RQ2: How do different node embedding and initialization methods affect model performance?

	Accuracy	Precision	Recall	F1 score
Word2Vec+BiLSTM	58.27	**62.23**	23.27	33.87
Doc2Vec	58.98	58.22	37.88	45.90
BPE+BiLSTM[a]	62.85	61.38	51.77	56.04
PE-BL with GCB[b]	63.30	59.31	**64.05**	**61.59**
PE-BL with CB[b]	**64.46**	61.87	58.99	60.39

[a]Word embedding layer with random initial weights.
[b]GCB and CB denote using the pre-trained BPE Tokenizer and word embedding weights of GraphCodeBERT and CodeBERT.

ignored, and the information of the node cannot be represented effectively. In ReGVD, edges are the tokens' relations based on sliding windows, which are still essentially natural language sequence relations and lack structured relations like control flow inherent in the code.

CSGVD uses the control flow information of the code as the model input. Using a BPE tokenizer is for tokenizing code and incorporating a pre-trained language model is for initializing weights. The BPE tokenizer alleviates the OOV problem and the pre-trained language model enhances the embedding performance. At the same time, CSGVD uses a bidirectional LSTM to aggregate the local semantic information of code within a node to better obtain the vector representation of nodes. Also, as shown in Section 4.2, the embedding knowledge inherited from the pre-trained model can better enhance the representation of local information.

From the results in Table 4, CSGVD achieves 4.41%, 2.02%, 19.07% and 12.52% improvement in accuracy, precision, recall, and F1 score over Devign. As compared to ReGVD, CSGVD achieves 2.75% accuracy, 21.78% recall, and 13.22% F1 score improvement in the results of our retraining. And CSGVD has 0.77% accuracy improvement over the result 63.69% provided in the original ReGVD paper.

4.2. RQ2: How do different node embedding and initialization methods affect model performance?

In view of the excellent results of pre-trained language models in natural language processing, a large number of researchers are now also focusing on pre-training (Guo et al., 2021; Feng et al., 2020; Phan et al., 2021; Wang et al., 2021a; Lachaux et al., 2021; Ahmad et al., 2021) in code processing. Following the application of pre-trained models to downstream tasks in natural language processing, we have also introduced pre-trained models into our work, and propose the PE-BL module for node embedding. Like ReGVD, the PE-BL module only uses the word embedding layer of the pre-trained language model in order to be able to compare with other models. Compared to ReGVD, it uses the word embedding layer of the pre-trained model as a token vectorization tool and does not incorporate it into the training model. In contrast, the PE-BL module uses the word embedding weights of the pre-trained language model to initialize the self-embedding layer and trains the introduced embedding weights again. In our experiments, we compare three different ways of parameter initialization. The first one is for training our character-level word tokenizer on the training set and initializing the embedding layer parameters using random initialization, and the other two are for initializing our embedding layer weights using CodeBERT and GraphCodeBERT. In addition, we also add Word2Vec and Doc2Vec methods to prove the effectiveness of our method.

As shown by the average results of multiple experiments in Table 5, the performance of initialization using pre-trained model weights is better than the performance of random initialization. Compared with the random initialization, the model initialized with embedding weights using CodeBERT has 1.61%, 0.49%, 7.22%, and 4.35% improvement in accuracy, precision, recall, and F1 score, respectively. Using GraphCodeBERT is also a significant improvement. The experimental results demonstrate that the pre-trained language model is equally effective in code processing. Compared with Word2Vec and Doc2Vec, the PE-BL module has significant improvements in several metrics, which shows the effectiveness of the PE-BL module.

Table 6
RQ3: How do different token fusion methods affect model performance?

	Accuracy	Precision	Recall	F1 score
Mean	61.89	65.14	36.62	46.88
Sum	59.48	60.27	35.76	44.38
Max	62.94	**65.27**	42.35	50.96
LSTM	61.60	59.42	51.76	55.32
BiLSTM	**64.46**	61.87	**58.99**	**60.39**

Table 7
RQ4: How do different GNN models affect model performance?

	Accuracy	Precision	Recall	F1 score
GCN	63.03	60.95	54.66	57.52
GAT	63.37	59.51	**63.97**	**61.55**
R-GCN	**64.46**	**61.87**	58.99	60.39

Meanwhile, considering the inconsistency of the length of each line of code, we verify the impact of different sequence lengths on the model performance through several experiments. Fig. 5 shows the distribution of the node sequence lengths after the BPE tokenizer of CodeBERT. Fig. 6 shows the results of the experiments with different sequence lengths, from which we can observe that some metrics increase as the sequence length increases.

4.3. RQ3: How do different token fusion methods affect model performance?

In our experiments, we consider a code statement as a node of the graph, and we need to combine multiple token vectors in order to obtain a vector representation of a code statement as the vector representation of the corresponding node in the graph. We compare five different fusion methods, including three common fusion methods, "mean", "sum" and "max", and two other methods requiring the neural network assistance, LSTM, and bidirectional LSTM.

From the results in Table 6, the bidirectional LSTM has higher metrics. Compared with other methods, bidirectional LSTM can better fuse the contextual information in the code statement. Furthermore, using bidirectional LSTM to obtain the vector representation of a single statement of code can improve performance on the subsequent graph classification tasks.

4.4. RQ4: How do different GNN models affect model performance?

To verify the impact of different graph neural networks on the vulnerability detection task, we compared three different graph neural networks, GCN, GAT (Velickovic et al., 2018), and residual GCN. To make a fair comparison, the graph neural network layer was unified into two layers, the head of GAT was set to 8, and the other parameters of the model are kept consistent. As shown in Table 7, the residual GCN has improvements in accuracy, precision, recall, and F1 score of 1.43%, 0.92%, 4.33%, and 2.87% compared to the GCN. Compared with GAT, the residual GCN has 1.09% and 2.36% improvement in accuracy and precision. The results show that residual connectivity can improve the performance of GCN and obtain better graph features.

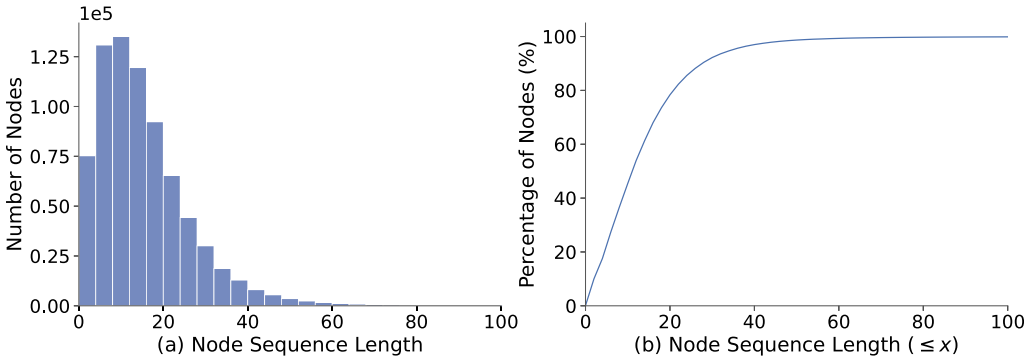

Fig. 5. Node sequence length distribution.

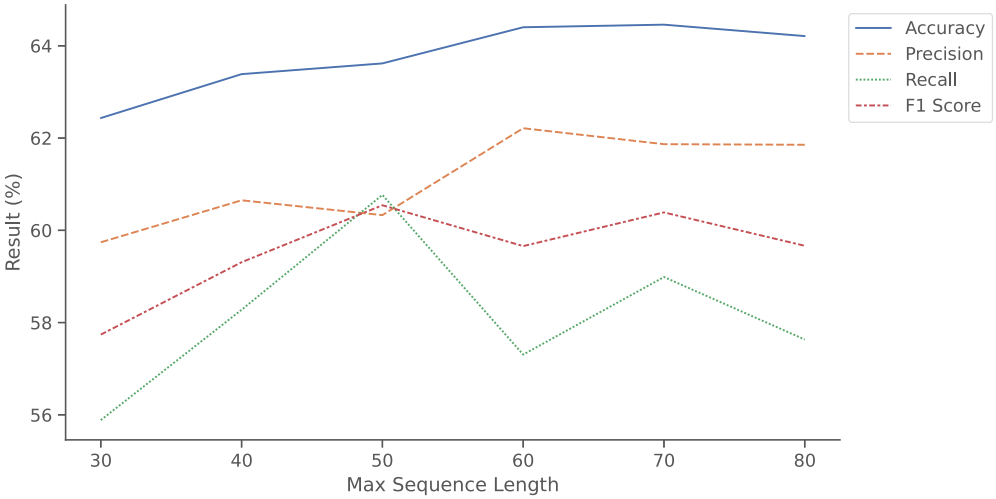

Fig. 6. Results for different sequence lengths.

Table 8
RQ5: How do different node fusion methods for graph embedding affect model performance?

	Accuracy	Precision	Recall	F1 score
Avg	63.16	60.72	56.71	58.54
Sum	62.27	61.25	49.30	54.39
Max	61.89	59.81	52.28	55.67
Sum×Max	60.00	61.07	35.36	43.02
M-BFA+max[a]	63.90	61.80	56.72	58.95
M-BFA+sum[a]	59.01	57.68	43.20	48.31
M-BFA	**64.46**	**61.87**	**58.99**	**60.39**

[a]Replace the mean with the max or sum of the nodes as the hidden representation of the supernode v_s.

4.5. RQ5: How do different graph embedding methods affect model performance?

In order to explore the effect of M-BFA for graph embedding, four different node fusion methods are selected for comparison, including graph-level average pooling, graph-level sum pooling, graph-level max pooling, and the node fusion method proposed in ReGVD. As shown in the average experimental results in the Table 8, the metric of M-BFA pooling is 64.46%, 61.87%, 58.99%, and 60.39%, respectively. It is much better than other fusion methods and demonstrates better nodes information fusion capability. Furthermore, it is experimentally demonstrated that the best results can be obtained by using the node mean as the vector representation of the supernode v_s. Compared with other approaches, the mean biaffine attention mechanism can learn the importance of the nodes and reinforce graph embedding.

5. Threats to validity

The first treat relates to the singularity of graph data. CSGVD only considers control flow graphs as input, lacking other information such as data flow graphs and abstract syntax trees, which may limit its ability to detect related vulnerabilities.

Another threat relates to the dataset. We use a C/C++ dataset built for function-level vulnerability detection, which is primarily derived from two open-source projects, FFmpeg and QEMU. The rare dataset may lead to poor generalization of the model. Furthermore, the function-level data samples focused on intraprocedural information and lacked interprocedural information, This flaw causes the model to fail to capture macro and interprocedural call information, making it difficult to understand the potential nature of the vulnerability.

6. Related work

6.1. Deep learning-based code processing

Since deep learning techniques have been widely applied to natural language processing, many researchers have also focused their efforts on applying them to code processing (Fu et al., 2019; Kanade et al., 2020; Svyatkovskiy et al., 2020; Jaffe et al., 2018; Sui et al., 2020; Alon et al., 2019; Lu et al., 2021; Guo et al., 2021; Feng et al., 2020; Yu et al., 2020b; Phan et al., 2021; Wang et al., 2021a; Jain et al., 2021; Raychev et al., 2019; Lachaux et al., 2021; Lacomis et al., 2019; Ahmad et al., 2021). Some researchers trained pre-trained programming language models on multiple

programming languages and achieved motivating results on multiple downstream tasks, such as clone detection, defect detection, code completion, code translation (Feng et al., 2020; Guo et al., 2021; Phan et al., 2021; Ahmad et al., 2021; Lachaux et al., 2021; Wang et al., 2021a). Lachaux et al. (2021) added a code deobfuscation task to the pre-trained language model, DOBF, for restoring the code after being obfuscated, while also achieving good results in several other downstream tasks. Lacomis et al. (2019) proposed DIRE, a model based on deep learning and structural information of the code, for recovering variable names in decompiled pseudocode and increasing the readability of decompiled code. Yu et al. (2020b) used DPCNN and several different LSTM neural networks for cross-modal function-level binary code matching. Fu et al. (2019) constructed a model named Coda for decompiling tasks instead of traditional decompilers.

6.2. Deep learning-based vulnerability detection

In recent years, due to the application of deep learning technology in vulnerability detection tasks, its detection performance has been continuously improved (Coimbra et al., 2021; Dam et al., 2017; Russell et al., 2018; Chakraborty et al., 2021; Zheng et al., 2021a; Li et al., 2018; Du et al., 2019; Cheng et al., 2021, 2019; Zhou et al., 2019; Li et al., 2021; Lin et al., 2020; Zheng et al., 2021b; Fan et al., 2020; Wang et al., 2021b; Hin et al., 2022; Xu, 2020; Nguyen et al., 2022). Russell et al. (2018) treated source code as a natural language sequence. They used Word2Vec for token embedding and CNN to extract code features for vulnerability detection tasks. Zhou et al. (2019) built the Devign model that used gate GNN to learn function-level code features from code property graphs, supplemented with 1-D convolution for node information fusion for function-level vulnerability detection. Guo et al. (2021) added edge prediction and node alignment tasks to train a cross-language pre-trained programming language model, named CodeBERT. Their goal was to learn structural information for code sequence. Meanwhile, CodeBERT was also applied to a downstream function-level defect detection task. Chakraborty et al. (2021) proposed the ReVeal model and tested it on several vulnerability detection datasets. They validated the impact of dataset composition in vulnerability detection tasks. Hin et al. (2022) proposed LineVD, a fine-grained vulnerability detection model. Its purpose was to perform statement-level vulnerability detection tasks and provide developers with a more user-friendly interface. Nguyen et al. (2022) proposed ReGVD for vulnerability detection by constructing graph relationships between tokens based on a sliding window mechanism. Li et al. (2018) extracted code gadgets from source code and fed them to a BiLSTM model to perform vulnerability detection tasks. Cheng et al. (2021) used the subgraph of PDG, named XFG, as input, then utilized a JK-Net style model with top-k pooling to predict vulnerabilities.

7. Conclusion

We consider vulnerability identification as a graph binary classification problem and propose a combined sequence and graph embedding neural network model, named CSGVD, to detect vulnerabilities in function-level source code. CSGVD uses the statement-level control flow graph of the source code as input. CSGVD then leverages a PE-BL module, a graph neural networks layers with residual connectivity, and a graph-level mean biaffine attention (M-BFA) pooling to learn graph presentation. To demonstrate the effectiveness of CSGVD, we conduct extensive experiments to compare CSGVD with other deep learning-based models on a real dataset from FFmpeg and QEMU. The experimental results show that CSGVD is significantly better than the baseline models and obtains the highest accuracy of 64.46% on the dataset. In the future, we will continue to explore better methods of node embedding and graph embedding.

CRediT authorship contribution statement

Wei Tang: Conceptualization, Methodology, Software, Writing – original draft. **Mingwei Tang:** Writing – review & editing, Supervision. **Minchao Ban:** Data curation, Visualization. **Ziguo Zhao:** Investigation, Validation. **Mingjun Feng:** Supervision.

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

Dataset from https://github.com/microsoft/CodeXGLUE.

Acknowledgments

This work is supported by the Scientic Research Funds project of Science and Technology Department of Sichuan Province, China (No. 2019YFG0508, 2019GFW131, 2023JY**), the National Natural Science Foundation of China (No. 61902324), Funds Project of Chengdu Science and Technology Bureau, China (No. 2023**, 2017-RK00-00026-ZF), Sichuan Youth Science and technology innovation research team, China (No. 2021**), Key Research and Development Program of Sichuan Province, China (No. 23ZDYF0121) and Science and Technology Planning Project of Guizhou Province, China (No. QianKeHeJiChu-ZK[2021]YiBan 319).

Appendix. Evaluation on the imbalanced dataset ReVeal

As a supplement, in this section we evaluate the performance of our model on the imbalanced dataset ReVeal. The ReVeal dataset was created by Chakraborty et al. (2021). It contains 18,169 samples from the Chromium and Debian projects, with 9.16% of the vulnerable samples. In this paper, we split the ReVeal dataset into train/validate/test according to 8:1:1. In the below section, this paper evaluates the performance of our model on different imbalanced dataset processing techniques. In addition, we also compare the performance difference between our model and other models.

A.1. Different imbalance processing techniques

In this section, we compare the performance of our model on the following imbalance processing techniques.

- **Weight random sampling**: Weighted random sampling is a data sampling technique that controls the balance of samples by setting sampling weights for different categories of data. The formula for calculating the weights used in this paper follows (A.1). $w_{y\in\{0,1\}}$ denotes the weights of different categories, and $N_{y\in\{0,1\}}$ is the number of samples of different categories.

$$w_{y\in\{0,1\}} = \frac{1}{N_{y\in\{0,1\}}} \tag{A.1}$$

- **Fixed weight cross-entropy loss**: Weighted cross-entropy loss deals with dataset imbalance from a loss learning perspective by increasing the loss weights of minority classes to make the model more sensitive to minority classes. Fixed weights mean that the weights used are fixed in the training. The formula for calculating the weights used in this paper follows (A.2). N is the total number of samples.

$$w_{y\in\{0,1\}} = \frac{N}{N_{y\in\{0,1\}}} \tag{A.2}$$

Table A.9
Performance of different imbalance techniques and graph neural network layers.

	Accuracy	Precision	Recall	F1 score
$R-GCN$	**90.08**	**51.14**	24.32	32.97
$R-GAT$	89.48	46.85	36.22	40.85
$R-GCN_{wrs}$	80.75	28.09	58.92	38.05
$R-GAT_{wrs}$	82.16	28.82	52.97	37.33
$R-GCN_{fw}$	79.83	27.03	59.46	37.16
$R-GAT_{fw}$	80.26	28.54	**64.32**	39.53
$R-GCN_{dw}$	84.71	33.56	53.51	41.25
$R-GAT_{dw}$	89.37	46.63	41.08	**43.68**

* wrs is "weight random sampling", fw is "fixed weight cross-entropy loss", and dw is "dynamic weight cross-entropy loss".
* "$R-$" means that residual connections are added to the model.

Table A.10
Different model performance.

	Accuracy	Precision	Recall	F1 score
CodeBERT	90.19	54.68	32.20	40.53
GraphCodeBERT	90.98	71.83	21.61	33.22
CoTexT	90.50	61.90	22.03	32.50
CodeT5	91.20	65.52	32.20	43.18
PLBART	90.54	62.65	22.03	32.60
Russell et al.	89.07	37.04	13.51	19.80
ReGVD	89.93	58.14	10.59	17.92
Devign	88.11	28.99	25.00	26.85
Ours	89.37	46.63	**41.08**	**43.68**

- **Dynamic weight cross-entropy loss**: Dynamic weights mean that the class weights are continually updated with the training procedure. We dynamically adjust the class weights based on the number of positive and negative samples in every batch. The formula for calculating the weights is shown in (A.3). N_b is the total number of samples in a batch. To avoid $w = 0$ when the number of samples in a class is 0, we modify the calculation formula as shown in (A.4).

$$w_{b0} = \frac{N_{b1}}{N_b}, w_{b1} = \frac{N_{b0}}{N_b} \tag{A.3}$$

$$w_{b0} = \exp(\frac{N_{b1}}{N_b}), w_{b1} = \exp(\frac{N_{b0}}{N_b}) \tag{A.4}$$

As shown in Table A.9, the model without imbalance handling has poor results in recall and f1 score despite having high accuracy and precision. The addition of weight random sampling and fixed weight cross-entropy loss brings some recall and f1 score improvement on the R-GCN model but sacrifices a large amount of accuracy and precision. In addition, for the R-GAT model, it also reduces the f1 score. When using dynamic weight cross-entropy loss, a smaller sacrifice in accuracy and precision brings a larger improvement in f1 score on the R-GCN model. Meanwhile, for the R-GAT model, the loss of accuracy and precision is negligible, but the recall and f1 scores are improved as well.

A.2. Different model performance

As shown in Table A.10, due to not dealing with the problem of imbalance of the dataset, most models have low recall and f1 score. In addition, as shown in Table A.11, we also compare the REVEAL model, which uses the SMOTE (Chawla et al., 2002) method to deal with the dataset imbalance problem. From Table A.11, we can know that REVEAL has a higher recall, but our model has more advantages in the other three metrics.

Table A.11
Comparison with REVEAL model.

	Accuracy	Precision	Recall	F1 score
REVEAL	84.37	30.91	60.91	41.25
Ours	**89.37**	**46.63**	41.08	**43.68**

The results of REVEAL are from the original paper.

References

Ahmad, W., Chakraborty, S., Ray, B., Chang, K.-W., 2021. 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, pp. 2655–2668. http://dx.doi.org/10.18653/v1/2021.naacl-main.211, URL https://aclanthology.org/2021.naacl-main.211.

Alon, U., Zilberstein, M., Levy, O., Yahav, E., 2019. Code2vec: Learning distributed representations of code. Proc. ACM Program. Lang. (PACMPL) 3, http://dx.doi.org/10.1145/3290353.

Anon, 2022a. National Vulnerability Database. American Information Technology Laboratory, URL https://www.nist.gov/.

Anon, 2022b. Clang static analyzer. URL https://clang-analyzer.llvm.org/scan-build.html.

Anon, 2022c. Checkmarx. URL https://checkmarx.com/.

Anon, 2022d. Coverity. URL https://scan.coverity.com/.

Anon, 2022e. FFmpeg. URL https://ffmpeg.org/.

Anon, 2022f. Flawfinder. URL https://dwheeler.com/flawfinder/.

Anon, 2022g. Joern - The bug Hunter's workbench. URL https://joern.io/.

Anon, 2022h. QEMU. URL https://www.qemu.org/.

Anon, 2022i. RATS. URL https://code.google.com/archive/p/rough-auditing-tool-for-security/.

Chakraborty, S., Krishna, R., Ding, Y., Ray, B., 2021. Deep learning based vulnerability detection: Are we there yet. IEEE Trans. Softw. Eng. 1. http://dx.doi.org/10.1109/TSE.2021.3087402.

Chawla, N.V., Bowyer, K.W., Hall, L.O., Kegelmeyer, W.P., 2002. SMOTE: synthetic minority over-sampling technique. J. Artificial Intelligence Res. 16, 321–357.

Cheng, X., Wang, H.Y., Hua, J.Y., Xu, G.A., Sui, Y.L., 2021. DeepWukong: Statically detecting software vulnerabilities using deep graph neural network. ACM Trans. Softw. Eng. Methodol. 30 (3), 33. http://dx.doi.org/10.1145/3436877.

Cheng, X., Wang, H., Hua, J., Zhang, M., Xu, G., Yi, L., Sui, Y., 2019. Static detection of control-flow-related vulnerabilities using graph embedding. In: 2019 24th International Conference on Engineering of Complex Computer Systems. ICECCS, IEEE Computer Society, pp. 41–50. http://dx.doi.org/10.1109/ICECCS.2019.00012, URL https://www.computer.org/csdl/proceedings-article/iceccs/2019/464600a041/1etdUjojcME.

Clark, K., Luong, M.-T., Le, Q.V., Manning, C.D., 2020. ELECTRA: Pre-training text encoders as discriminators rather than generators. In: 8th International Conference on Learning Representations. ICLR 2020, Addis Ababa, Ethiopia, April 26–30, 2020, OpenReview.net, URL https://openreview.net/forum?id=r1xMH1BtvB.

Coimbra, D., Reis, S., Abreu, R., Păsăreanu, C., Erdogmus, H., 2021. On using distributed representations of source code for the detection of C security vulnerabilities. arXiv:2106.01367.

Dam, H.K., Tran, T., Pham, T., Ng, S.W., Grundy, J., Ghose, A., 2017. Automatic feature learning for vulnerability prediction. arXiv:1708.02368.

Devlin, J., Chang, M.-W., Lee, K., Toutanova, K., 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In: Burstein, J., Doran, C., Solorio, T. (Eds.), Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Vol. 1 (Long and Short Papers). NAACL-HLT 2019, Minneapolis, MN, USA, June 2–7, 2019, Association for Computational Linguistics, pp. 4171–4186. http://dx.doi.org/10.18653/v1/n19-1423.

Du, X., Chen, B., Li, Y., Guo, J., Zhou, Y., Liu, Y., Jiang, Y., 2019. LEOPARD: Identifying vulnerable code for vulnerability assessment through program metrics. In: 2019 IEEE/ACM 41st International Conference on Software Engineering. ICSE, IEEE, http://dx.doi.org/10.1109/icse.2019.00024.

Fan, J., Li, Y., Wang, S., Nguyen, T.N., 2020. A C/C++ code vulnerability dataset with code changes and CVE summaries. In: Proceedings of the 17th International Conference on Mining Software Repositories. Association for Computing Machinery, pp. 508–512. http://dx.doi.org/10.1145/3379597.3387501.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., Zhou, M., 2020. CodeBERT: A pre-trained model for programming and natural languages. In: Cohn, T., He, Y., Liu, Y. (Eds.), Findings of the Association for Computational Linguistics. EMNLP 2020, Online Event, 16–20 November 2020, In: Findings of ACL, vol. EMNLP 2020, Association for Computational Linguistics, pp. 1536–1547. http://dx.doi.org/10.18653/v1/2020.findings-emnlp.139.

Fu, C., Chen, H., Liu, H., Chen, X., Tian, Y., Koushanfar, F., Zhao, J., 2019. Coda: An end-to-end neural program decompiler. In: Wallach, H.M., Larochelle, H., Beygelzimer, A., d'Alché- Buc, F., Fox, E.B., Garnett, R. (Eds.), Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019. NeurIPS 2019, December 8–14, 2019, Vancouver, BC, Canada, pp. 3703–3714, URL https://proceedings.neurips.cc/paper/2019/hash/093b60fd0557804c8ba0cbf1453da22f-Abstract.html.

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., Tufano, M., Deng, S.K., Clement, C.B., Drain, D., Sundaresan, N., Yin, J., Jiang, D., Zhou, M., 2021. GraphCodeBERT: Pre-training code representations with data flow. In: 9th International Conference on Learning Representations. ICLR 2021, Virtual Event, Austria, May 3–7, 2021, OpenReview.net, URL https://openreview.net/forum?id=jLoC4ez43PZ.

Hin, D., Kan, A., Chen, H., Babar, M.A., 2022. LineVD: Statement-level vulnerability detection using graph neural networks. In: 2022 IEEE/ACM 19th International Conference on Mining Software Repositories. MSR, pp. 596–607. http://dx.doi.org/10.1145/3524842.3527949.

Hochreiter, S., Schmidhuber, J., 1997. Long short-term memory. Neural Comput. 9 (8), 1735–1780. http://dx.doi.org/10.1162/neco.1997.9.8.1735.

Jaffe, A., Lacomis, J., Schwartz, E.J., Goues, C.L., Vasilescu, B., 2018. Meaningful variable names for decompiled code: A machine translation approach. In: Proceedings of the 26th Conference on Program Comprehension. pp. 20–30.

Jain, P., Jain, A., Zhang, T., Abbeel, P., Gonzalez, J., Stoica, I., 2021. Contrastive code representation learning. In: Moens, M.-F., Huang, X., Specia, L., Yih, S.W.-t. (Eds.), Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing. EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7–11 November, 2021, Association for Computational Linguistics, pp. 5954–5971. http://dx.doi.org/10.18653/v1/2021.emnlp-main.482.

Kanade, A., Maniatis, P., Balakrishnan, G., Shi, K., 2020. Learning and Evaluating Contextual Embedding of Source Code. PMLR, pp. 5110–5121.

Kim, Y., 2014. Convolutional neural networks for sentence classification. In: Moschitti, A., Pang, B., Daelemans, W. (Eds.), Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25–29, 2014, Doha, Qatar, a Meeting of SIGDAT, a Special Interest Group of the ACL. ACL, pp. 1746–1751. http://dx.doi.org/10.3115/v1/d14-1181.

Lachaux, M.-A., Rozière, B., Szafraniec, M., Lample, G., 2021. DOBF: A deobfuscation pre-training objective for programming languages. In: Ranzato, M., Beygelzimer, A., Dauphin, Y.N., Liang, P., Vaughan, J.W. (Eds.), Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021. NeurIPS 2021, December 6–14, 2021, Virtual, pp. 14967–14979, URL https://proceedings.neurips.cc/paper/2021/hash/7d6548bdc0082aacc950ed35e91fcccb-Abstract.html.

Lacomis, J., Yin, P., Schwartz, E., Allamanis, M., Le Goues, C., Neubig, G., Vasilescu, B., 2019. Dire: A Neural Approach to Decompiled Identifier Naming. IEEE, pp. 628–639.

Le, Q., Mikolov, T., 2014. Distributed representations of sentences and documents. In: Proceedings of the 31st International Conference on Machine Learning. PMLR, pp. 1188–1196, URL https://proceedings.mlr.press/v32/le14.html.

Lewis, M., Liu, Y., Goyal, N., Ghazvininejad, M., Mohamed, A., Levy, O., Stoyanov, V., Zettlemoyer, L., 2020. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics. Association for Computational Linguistics, pp. 7871–7880. http://dx.doi.org/10.18653/v1/2020.acl-main.703, Online, URL https://aclanthology.org/2020.acl-main.703.

Li, Y., Tarlow, D., Brockschmidt, M., Zemel, R.S., 2016. Gated graph sequence neural networks. In: Bengio, Y., LeCun, Y. (Eds.), 4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2–4, 2016, Conference Track Proceedings. URL http://arxiv.org/abs/1511.05493.

Li, Y., Wang, S., Nguyen, T.N., 2021. Vulnerability detection with fine-grained interpretations. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 292–303. http://dx.doi.org/10.1145/3468264.3468597, arXiv:2106.10478.

Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., Zhong, Y., 2018. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. Internet Society, http://dx.doi.org/10.14722/ndss.2018.23158.

Lin, G., Wen, S., Han, Q.-L., Zhang, J., Xiang, Y., 2020. Software vulnerability detection using deep neural networks: A survey. Proc. IEEE 108 (10), 1825–1848. http://dx.doi.org/10.1109/JPROC.2020.2993293.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., Stoyanov, V., 2019. RoBERTa: A robustly optimized BERT pretraining approach. http://dx.doi.org/10.48550/arXiv.1907.11692, arXiv:1907.11692.

Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C.B., Drain, D., Jiang, D., Tang, D., Li, G., Zhou, L., Shou, L., Zhou, L., Tufano, M., Gong, M., Zhou, M., Duan, N., Sundaresan, N., Deng, S.K., Fu, S., Liu, S., 2021. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. In: Vanschoren, J., Yeung, S.-K. (Eds.), Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1. NeurIPS Datasets and Benchmarks 2021, December 2021, Virtual, URL https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/c16a5320fa475530d9583c34fd356ef5-Abstract-round1.html.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G.S., Dean, J., 2013. Distributed representations of words and phrases and their compositionality. In: Advances in Neural Information Processing Systems, Vol. 26. Curran Associates, Inc., URL https://proceedings.neurips.cc/paper/2013/hash/9aa42b31882ec039965f3c4923ce901b-Abstract.html.

Nguyen, V.-A., Nguyen, D.Q., Nguyen, V., Le, T., Tran, Q.H., Phung, D., 2022. ReGVD: Revisiting graph neural networks for vulnerability detection. In: 2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings. ICSE-Companion, pp. 178–182. http://dx.doi.org/10.1109/ICSE-Companion55297.2022.9793807.

Phan, L., Tran, H., Le, D., Nguyen, H., Annibal, J., Peltekian, A., Ye, Y., 2021. CoTexT: Multi-task learning with code-text transformer. In: Proceedings of the 1st Workshop on Natural Language Processing for Programming. NLP4Prog 2021, Association for Computational Linguistics, pp. 40–47. http://dx.doi.org/10.18653/v1/2021.nlp4prog-1.5, URL https://aclanthology.org/2021.nlp4prog-1.5.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., Liu, P.J., 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. J. Mach. Learn. Res. 21 (140), 1–67, URL http://jmlr.org/papers/v21/20-074.html.

Raychev, V., Vechev, M.T., Krause, A., 2019. Predicting program properties from 'Big Code'. Commun. ACM 62 (3), 99–107. http://dx.doi.org/10.1145/3306204.

Russell, R., Kim, L., Hamilton, L., Lazovich, T., Harer, J., Ozdemir, O., Ellingwood, P., McConley, M., 2018. Automated vulnerability detection in source code using deep representation learning. In: 2018 17th IEEE International Conference on Machine Learning and Applications. ICMLA, pp. 757–762. http://dx.doi.org/10.1109/ICMLA.2018.00120.

Sennrich, R., Haddow, B., Birch, A., 2016. Neural machine translation of rare words with subword units. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, Vol. 1: Long Papers. ACL 2016, August 7–12, 2016, Berlin, Germany, The Association for Computer Linguistics, http://dx.doi.org/10.18653/v1/p16-1162.

Sui, Y., Cheng, X., Zhang, G., Wang, H., 2020. Flow2Vec: Value-flow-based precise code embedding. Proc. ACM Program. Lang. (PACMPL) 4, http://dx.doi.org/10.1145/3428301.

Svyatkovskiy, A., Deng, S.K., Fu, S., Sundaresan, N., 2020. Intellicode compose: Code generation using transformer. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 1433–1443.

Velickovic, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., Bengio, Y., 2018. Graph attention networks. In: 6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings. OpenReview.net, URL https://openreview.net/forum?id=rJXMpikCZ.

Wang, Y., Wang, W., Joty, S., Hoi, S.C., 2021a. 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. Association for Computational Linguistics, pp. 8696–8708. http://dx.doi.org/10.18653/v1/2021.emnlp-main.685, URL https://aclanthology.org/2021.emnlp-main.685.

Wang, H., Ye, G., Tang, Z., Tan, S.H., Huang, S., Fang, D., Feng, Y., Bian, L., Wang, Z., 2021b. Combining graph-based learning with automated data collection for code vulnerability detection. IEEE Trans. Inf. Forensics Secur. 16, 1943–1958. http://dx.doi.org/10.1109/TIFS.2020.3044773.

Xu, Z., 2020. Source code and binary level vulnerability detection and hot patching. In: 2020 35th IEEE/ACM International Conference on Automated Software Engineering. ASE, pp. 1397–1399.

Yu, J., Bohnet, B., Poesio, M., 2020a. Named entity recognition as dependency parsing. In: Jurafsky, D., Chai, J., Schluter, N., Tetreault, J.R. (Eds.), Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics. ACL 2020, Online, July 5–10, 2020, Association for Computational Linguistics, pp. 6470–6476. http://dx.doi.org/10.18653/v1/2020.acl-main.577.

Yu, Z., Zheng, W., Wang, J., Tang, Q., Nie, S., Wu, S., 2020b. CodeCMR cross-modal retrieval for function-level binary source code matching. In: Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M.-F., Lin, H.-T. (Eds.), Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020. NeurIPS 2020, December 6–12, 2020, Virtual, URL https://proceedings.neurips.cc/paper/2020/hash/285f89b802bcb2651801455c86d78f2a-Abstract.html.

Zeng, Z., Tan, H., Zhang, H., Li, J., Zhang, Y., Zhang, L., 2022. An extensive study on pre-trained models for program understanding and generation. In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA 2022, Association for Computing Machinery, New York, NY, USA, pp. 39–51. http://dx.doi.org/10.1145/3533767.3534390.

Zheng, W., Abdallah Semasaba, A.O., Wu, X., Agyemang, S.A., Liu, T., Ge, Y., 2021a. Representation vs. model: What matters most for source code vulnerability detection. In: 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering. SANER, IEEE, http://dx.doi.org/10.1109/saner50967.2021.00082.

Zheng, Y., Pujar, S., Lewis, B., Buratti, L., Epstein, E., Yang, B., Laredo, J., Morari, A., Su, Z., 2021b. D2A: A dataset built for AI-based vulnerability detection methods using differential analysis. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice. ICSE-SEIP, pp. 111–120. http://dx.doi.org/10.1109/ICSE-SEIP52600.2021.00020.

Zhou, Y., Liu, S., Siow, J., Du, X., Liu, Y., 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In: Wallach, H., Larochelle, H., Beygelzimer, A., dAlché Buc, F., Fox, E., Garnett, R. (Eds.), Advances in Neural Information Processing Systems, Vol. 32. Curran Associates, Inc., URL https://proceedings.neurips.cc/paper/2019/file/49265d2447bc3bbfe9e76306ce40a31f-Paper.pdf.

Wei Tang is a master student at the School of Computer and Software Engineering from Xihua University, Chengdu, China. His research interests include natural language processing and source code vulnerability detection.

Mingwei Tang is a professor in the school of computer and software engineering, Xihua University. He received the Ph.D. degree in the school of computer science and Engineering from University of Electronic Science and technology of China in 2012. His research interests include deep learning and natural language processing.

Minchao Ban is a master student at the School of Computer and Software Engineering from Xihua University, Chengdu, China. His research interests include natural language processing and aspect-based sentiment analysis.

Ziguo Zhao is a master student at the School of Computer and Software Engineering from Xihua University, Chengdu, China. His research interests include natural language processing and aspect-based sentiment analysis.

Mingjun Feng is a senior engineer at the State Grid Corporation of China. She received the Master's degree in the school of computer science and Engineering from Xihua University in 2008. Her research interests include information security, deep learning and natural language processing.