REGVD: REVISITING GRAPH NEURAL NETWORKS FOR VULNERABILITY DETECTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Identifying vulnerabilities in the source code is essential to protect the software systems from cyber security attacks. It, however, is also a challenging step that requires specialized expertise in security and code representation. Inspired by the successful applications of pre-trained programming language models such as CodeBERT and graph neural networks (GNNs) for natural language processing, we propose ReGVD, a general and novel graph neural network-based model for vulnerability detection. In particular, ReGVD views a given source code as a flat sequence of tokens and then examines two effective methods of utilizing unique tokens and indexes respectively to construct a single graph as an input, wherein node features are initialized only by the embedding layer of a pre-trained PL model. Next, ReGVD leverages a practical advantage of residual connection among GNN layers and explores a beneficial mixture of graph-level sum and max poolings to return a graph embedding for the given source code. Experimental results demonstrate that ReGVD outperforms the existing state-of-the-art models and obtain the highest accuracy on the real-world benchmark dataset from CodeXGLUE for vulnerability detection.

1 INTRODUCTION

The software vulnerability problems have been rapidly grown recently, either reported through publicly disclosed information-security flaws and exposures (CVE) or exposed inside privately-owned source codes and open-source libraries. These vulnerabilities are the main reasons for cyber security attacks on the software systems that cause substantial damages economically and socially (Neuhaus et al., 2007; Zhou et al., 2019). Therefore, vulnerability detection is an essential yet challenging step to identify vulnerabilities in the source codes to provide security solutions for the software systems.

Early approaches (Neuhaus et al., 2007; Nguyen & Tran, 2010; Shin et al., 2010) have been proposed to carefully design hand-engineered features for machine learning algorithms to detect vulnerabilities. These early approaches, however, suffer from two major drawbacks. First, designing good features requires prior knowledge, hence needs domain experts, and is usually time-consuming. Second, hand-engineered features are impractical and not straightforward to adapt to all vulnerabilities in numerous libraries evolving over time.

To reduce human efforts on feature engineering, recent approaches (Li et al., 2018; Russell et al., 2018) consider each raw source code as a flat natural language sequence and explore deep learning architectures applied for natural language processing (NLP) (such as LSTMs (Hochreiter & Schmidhuber, 1997) and CNNs (Kim, 2014)) in vulnerability detection. Besides, it is worth noting that pre-trained language models such as BERT (Devlin et al., 2018) have recently emerged as a significantly trending learning paradigm, offering numerous successful applications in NLP. Inspired by the successes of BERT-style models, pre-trained programming language (PL) models such as CodeBERT (Feng et al., 2020) have also made a significantly improvement for PL downstream tasks including vulnerability detection. However, as mentioned in (Nguyen et al., 2019), all interactions among all positions in the input sequence inside the self-attention layer of the BERT-style model build up a complete graph, i.e., every position has an edge to all other positions. Hence, this limits learning local structures within the source code to differentiate vulnerabilities.

Graph neural networks (GNNs) have recently become a central method to embed nodes and graphs into low-dimensional continuous vector spaces (Hamilton et al., 2017; Wu et al., 2019). GNNs

provide faster and practical training, higher accuracy, and state-of-the-art results for downstream tasks such as text classification (Yao et al., 2019). Inspired by this advanced architecture, Devign (Zhou et al., 2019) is proposed to utilize GNNs for vulnerability detection, using a complex pre-process to extract multi-edged graph information such as Abstract Syntax Tree (AST), data flow, and control flow from the source code. This complex pre-process, however, is difficult of being practiced for many programming languages and numerous open-source codes and libraries.

In this paper, we propose a general and novel graph neural network-based model, named ReGVD, for vulnerability identification. In particular, we also consider programming language as natural language. Hence, ReGVD treats a given source code as a flat sequence of tokens and leverages two effective graph construction methods to build a single graph. The first method is to consider unique tokens as nodes and co-occurrences between nodes (within a fixed-size sliding window) as edges. The second is to consider indexes as nodes and also co-occurrences between indexes as edges. To make a fair comparison with pre-trained PL models such as CodeBERT, ReGVD employs only the embedding layer of these PL models to initialize node feature vectors. Then, ReGVD examines GNNs, but with a novelty of using residual connection among GNN layers. Next, ReGVD exploits the sum and max poolings and utilizes a beneficial mixture between these poolings to produce a graph embedding for the given source code. This graph embedding is finally fed to a single fully-connected layer followed by a softmax layer to predict the code vulnerabilities. To sum up, our main contributions are as follows:

- We are inspired by pre-trained programming language models and graph neural neural networks to introduce ReGVD – a novel GNN-based model for vulnerability detection.
- ReGVD makes use of effective code representation through two graph construction methods to build a graph for each given source code, wherein node features are initialized only by the embedding layer of a pre-trained PL model. ReGVD then introduces a novel adaptation of residual connection among GNN layers and an advantageous mixture of the sum and max poolings to enhance learning better code graph representation.
- Extensive experiments show that ReGVD significantly outperforms the existing state-of-the-art models and produces the highest accuracy of 63.69%, gaining absolute improvements of 1.61% and 1.39% over CodeBERT and GraphCodeBERT respectively, on the benchmark vulnerability detection dataset from CodeXGLUE (Lu et al., 2021).

2 THE PROPOSED REGVD

2.1 PROBLEM DEFINITION

We formalize vulnerability detection as an inductive binary classification problem for source code at the function level, i.e., we aim to identify whether a given function in raw source code is vulnerable or not (Zhou et al., 2019). We define a data sample as $\{(\mathsf{c}_i, \mathsf{y}_i)|\mathsf{c}_i \in \mathbb{C}, \mathsf{y}_i \in \mathbb{Y}\}_{i=1}^{n}$, where $\mathbb{C}$ represents the set of raw source codes, $\mathbb{Y} = \{0, 1\}^n$ denotes the label set with 1 for vulnerable and 0 otherwise, and n is the number of instances. As graph neural networks (GNNs) provide faster and practical training, higher accuracy, and state-of-the-art results for many downstream tasks (Kipf & Welling, 2017), we leverage GNNs for vulnerability detection. Therefore, we construct a graph $\mathsf{g}_i(\mathcal{V}, \boldsymbol{X}, \boldsymbol{A}) \in \mathcal{G}$ for each given source code c_i, wherein $\mathcal{V}$ is a set of m nodes in the graph; $\boldsymbol{X} \in \mathbb{R}^{m \times d}$ is the node feature matrix, wherein each node v_j in $\mathcal{V}$ is represented by a d-dimensional real-valued vector $\boldsymbol{x}_j \in \mathbb{R}^d$; $\boldsymbol{A} \in \{0, 1\}^{m \times m}$ is the adjacency matrix, where $\boldsymbol{A}_{\mathsf{v},\mathsf{u}}$ equal to 1 means having an edge between node v and node u, and 0 otherwise. We aim 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 $\boldsymbol{\theta}$ as:

$$\mathsf{min} \sum_{i=1}^{n} \mathcal{L}(f(\mathsf{g}_i(\mathcal{V}, \boldsymbol{X}, \boldsymbol{A}), \mathsf{y}_i|\mathsf{c}_i)) + \lambda \|\boldsymbol{\theta}\|_2^2 \tag{1}$$

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

Note that Devign (Zhou et al., 2019) uses a complex pre-process to build a multi-edged graph for each given source code. Hence it is impractical for many programming languages (PLs) and numerous open-source codes and libraries. It is also worth noting that recently, pre-trained PL models such

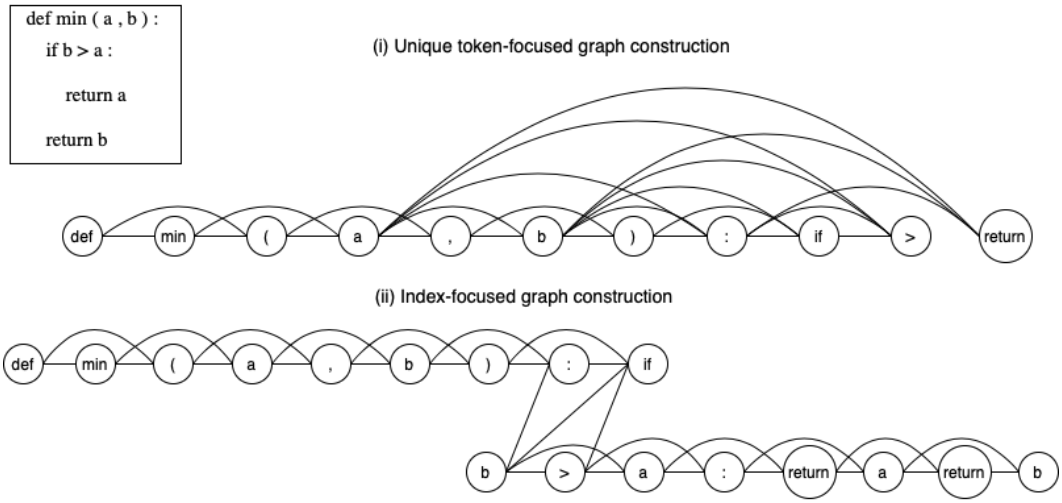


Figure 1: An illustration for two graph construction methods with a fixed-size sliding window of length 3 (the window size equals 3).

as CodeBERT (Feng et al., 2020) have significantly improved the performance of PL downstream tasks such as vulnerability detection. However, these BERT-style PL models limit learning local and logical structures inside the source code to differentiate vulnerabilities. To this end, we propose ReGVD – a novel and general GNN-based model using effective code representation for vulnerability detection as follows: (i) ReGVD views a given raw source code as a flat sequence of tokens and transforms this sequence into a single graph. (ii) ReGVD examines GCNs (Kipf & Welling, 2017) and Gated GNNs (Li et al., 2016), with a novel use of residual connection among GNN layers. (iii) ReGVD utilizes a new and beneficial mixture between the sum and max poolings to produce a graph embedding for the given source code.

In what follows, we first introduce two effective methods to construct a graph for each raw source code in Section 2.2, then describe our ReGVD in utilizing graph neural networks with residual connection in Section 2.3, finally focus on presenting a graph-level readout layer in Section 2.4 to obtain the graph embedding to perform the classification task.

2.2 GRAPH CONSTRUCTION

We consider a given source code as a flat sequence of tokens and illustrate two graph construction methods in Figure 1 to keep the local programming logic of source code. Note that we omit self-loops in these two methods since the self-loops do not help to improve performance in our pilot experiments. A possible reason is that source code is more structural than natural language where the self-loops can contribute useful graph information (Yao et al., 2019; Huang et al., 2019; Zhang et al., 2020).

Unique token-focused construction We represent unique tokens as nodes and co-occurrences between tokens (within a fixed-size sliding window) as edges, and the obtained graph has an adjacency matrix $\boldsymbol{A}$ as:

$$\boldsymbol{A}_{\mathsf{v},\mathsf{u}} = \begin{cases} 1 & \text{if nodes v and u co-occur within a sliding window and } \mathsf{v} \neq \mathsf{u} \\ 0 & \text{otherwise} \end{cases}$$

As the size of the graph is much smaller than the actual length of the source code, this method can consume less GPU memory.

Index-focused construction Given a flat sequence of l tokens $\{t_{\mathsf{i}}\}_{\mathsf{i}=1}^{l}$, we represent all tokens as the nodes, i.e., treating each index i as a node to represent token t_{i}. The number of nodes equals the sequence length. We also consider co-occurrences between indexes (within a fixed-size sliding

window) as edges, and the obtained graph has an adjacency matrix $\boldsymbol{A}$ as:

$$\boldsymbol{A}_{\mathsf{i},\mathsf{j}} = \begin{cases} 1 & \text{if indexes i and j co-occur within a sliding window and i} \neq \text{j} \\ 0 & \text{otherwise} \end{cases}$$

Node feature initialization To attain the advantage of pre-trained PL models such as CodeBERT and make a fair comparison, we use only *the embedding layer* of the pre-trained PL model to initialize node feature vectors.

2.3 Graph neural networks with residual connection

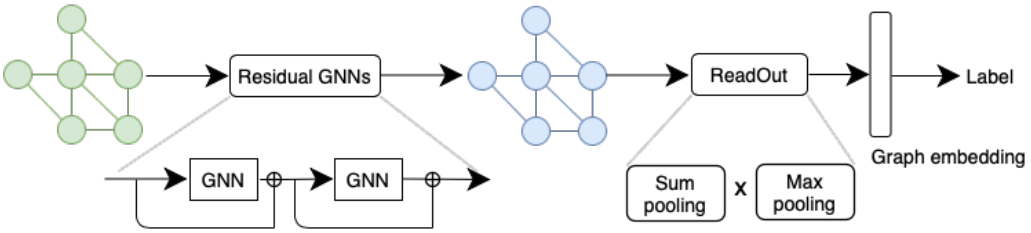


Figure 2: An illustration for our proposed ReGVD.

GNNs aim to update vector representations of nodes by recursively aggregating vector representations from their neighbours (Scarselli et al., 2009; Kipf & Welling, 2017). Mathematically, given a graph $\mathsf{g}(\mathcal{V}, \boldsymbol{X}, \boldsymbol{A})$, we formulate GNNs as follows:

$$\mathbf{h}_{\mathsf{v}}^{(k+1)} = \textsc{Aggregation}\left(\left\{\mathbf{h}_{\mathsf{u}}^{(k)}\right\}_{\mathsf{u}\in\mathcal{N}_{\mathsf{v}}\cup\{\mathsf{v}\}}\right) \tag{2}$$

where $\mathbf{h}_{\mathsf{v}}^{(k)}$ is the vector representation of node v at the k-th iteration/layer; $\mathcal{N}_{\mathsf{v}}$ is the set of neighbours of node v; and $\mathbf{h}_{\mathsf{v}}^{(0)} = \boldsymbol{x}_{\mathsf{v}}$ is the node feature vector of v.

There have been many GNNs proposed in recent literature (Wu et al., 2019), wherein Graph Convolutional Networks (GCNs) (Kipf & Welling, 2017) is the most widely-used one, and Gated graph neural networks ("Gated GNNs" or "GGNNs" for short) (Li et al., 2016) is also suitable for our data structure. Our ReGVD leverages GCNs and GGNNs as the base models.

Formally, GCNs is given as follows:

$$\mathbf{h}_{\mathsf{v}}^{(k+1)} = \phi\left(\sum_{\mathsf{u}\in\mathcal{N}_{\mathsf{v}}\cup\{\mathsf{v}\}} a_{\mathsf{v},\mathsf{u}}\boldsymbol{W}^{(k)}\mathbf{h}_{\mathsf{u}}^{(k)}\right), \forall \mathsf{v} \in \mathcal{V} \tag{3}$$

where $a_{\mathsf{v},\mathsf{u}}$ is an edge constant between nodes v and u in the Laplacian re-normalized adjacency matrix $\mathbf{D}^{-\frac{1}{2}}\boldsymbol{A}\mathbf{D}^{-\frac{1}{2}}$ (as we omit self-loops), wherein $\mathbf{D}$ is the diagonal node degree matrix of $\boldsymbol{A}$; $\boldsymbol{W}^{(k)}$ is a weight matrix; and ϕ is a nonlinear activation function such as ReLU.

GGNNs adopts GRUs (Cho et al., 2014), unrolls the recurrence for a fixed number of timesteps, and removes the need to constrain parameters to ensure convergence as:

$$\begin{aligned}
\mathbf{a}_{\mathsf{v}}^{(k+1)} &= \sum_{\mathsf{u}\in\mathcal{N}_{\mathsf{v}}} a_{\mathsf{v},\mathsf{u}}\mathbf{h}_{\mathsf{u}}^{(k)} \\
\mathbf{z}_{\mathsf{v}}^{(k+1)} &= \sigma\left(\mathbf{W}^{z}\mathbf{a}_{\mathsf{v}}^{(k+1)} + \mathbf{U}^{z}\mathbf{h}_{\mathsf{v}}^{(k)}\right) \\
\mathbf{r}_{\mathsf{v}}^{(k+1)} &= \sigma\left(\mathbf{W}^{r}\mathbf{a}_{\mathsf{v}}^{(k+1)} + \mathbf{U}^{r}\mathbf{h}_{\mathsf{v}}^{(k)}\right) \\
\widetilde{\mathbf{h}_{\mathsf{v}}^{(k+1)}} &= \phi\left(\mathbf{W}^{o}\mathbf{a}_{\mathsf{v}}^{(k+1)} + \mathbf{U}^{o}\left(\mathbf{r}_{\mathsf{v}}^{(k+1)} \odot \mathbf{h}_{\mathsf{v}}^{(k)}\right)\right) \\
\mathbf{h}_{\mathsf{v}}^{(k+1)} &= \left(1 - \mathbf{z}_{\mathsf{v}}^{(k+1)}\right) \odot \mathbf{h}_{\mathsf{v}}^{(k)} + \mathbf{z}_{\mathsf{v}}^{(k+1)} \odot \widetilde{\mathbf{h}_{\mathsf{v}}^{(k+1)}}
\end{aligned} \tag{4}$$

where $\mathbf{z}$ and $\mathbf{r}$ are the update and reset gates; σ is the sigmoid function; and $\odot$ is element-wise multiplication.

The residual connection (He et al., 2016) is used to incorporate information learned in the lower layers to the higher layers, and more importantly, to allow gradients to directly pass through the layers to avoid vanishing gradient or exploding gradient problems. The residual connection is employed in many architectures in computer vision and NLP. Motivated by that, ReGVD presents a novel adaptation of residual connection among the GNN layers, where we fix the same hidden size for the different layers. In particular, ReGVD redefines Equation 3 as:

$$\mathbf{h}_{\mathsf{v}}^{(k+1)} = \mathbf{h}_{\mathsf{v}}^{(k)} + \phi\left(\sum_{\mathsf{u}\in\mathcal{N}_{\mathsf{v}}\cup\{\mathsf{v}\}} a_{\mathsf{v},\mathsf{u}}\boldsymbol{W}^{(k)}\mathbf{h}_{\mathsf{u}}^{(k)}\right) \tag{5}$$

Similarly, ReGVD also redefines Equation 4 as follows:

$$\mathbf{h}_{\mathsf{v}}^{(k+1)} = \mathbf{h}_{\mathsf{v}}^{(k)} + \left(\left(1-\mathbf{z}_{\mathsf{v}}^{(k+1)}\right)\odot\mathbf{h}_{\mathsf{v}}^{(k)} + \mathbf{z}_{\mathsf{v}}^{(k+1)}\odot\widetilde{\mathbf{h}_{\mathsf{v}}^{(k+1)}}\right) \tag{6}$$

2.4 Graph-level readout pooling layer

The graph-level readout layer is used to produce a graph embedding for each input graph. This layer can be built more complex poolings such as hierarchical pooling (Cangea et al., 2018), differentiable pooling (Ying et al., 2018), and *Conv* pooling (Zhou et al., 2019). As the simple sum pooling produces better results for graph classification (Xu et al., 2019), ReGVD leverages the sum pooling to obtain the graph embedding. Besides, ReGVD utilizes the max pooling to exploit more information on the key nodes. ReGVD defines a beneficial mixture between the sum and max poolings to produce the graph embedding as follows:

$$\mathbf{e}_{\mathsf{v}} = \sigma\left(\mathbf{w}^{\mathsf{T}}\mathbf{h}_{\mathsf{v}}^{(K)} + \mathsf{b}\right)\odot\phi\left(\mathbf{W}\mathbf{h}_{\mathsf{v}}^{(K)} + \mathbf{b}\right), \forall \mathsf{v}\in\mathcal{V} \tag{7}$$

$$\mathbf{e}_{\mathcal{G}} = \mathsf{MIX}\left(\sum_{\mathsf{v}\in\mathcal{V}}\mathbf{e}_{\mathsf{v}}, \textsc{MaxPool}\left\{\mathbf{e}_{\mathsf{v}}\right\}_{\mathsf{v}\in\mathcal{V}}\right) \tag{8}$$

where $\mathbf{e}_{\mathsf{v}}$ is the final vector representation of node v, wherein $\sigma\left(\mathbf{w}^{\mathsf{T}}\mathbf{h}_{\mathsf{v}}^{(K)} + \mathsf{b}\right)$ acts as soft attention mechanisms over nodes (Li et al., 2016), and $\mathbf{h}_{\mathsf{v}}^{(K)}$ is the vector representation of node v at the last K-th layer; and $\mathsf{MIX}(.)$ denotes an arbitrary function. ReGVD examines three MIX functions consisting of SUM, MUL, and CONCAT as:

$$\mathsf{SUM} : \mathbf{e}_{\mathcal{G}} = \sum_{\mathsf{v}\in\mathcal{V}}\mathbf{e}_{\mathsf{v}} + \textsc{MaxPool}\left\{\mathbf{e}_{\mathsf{v}}\right\}_{\mathsf{v}\in\mathcal{V}} \tag{9}$$

$$\mathsf{MUL} : \mathbf{e}_{\mathcal{G}} = \sum_{\mathsf{v}\in\mathcal{V}}\mathbf{e}_{\mathsf{v}}\odot\textsc{MaxPool}\left\{\mathbf{e}_{\mathsf{v}}\right\}_{\mathsf{v}\in\mathcal{V}} \tag{10}$$

$$\mathsf{CONCAT} : \mathbf{e}_{\mathcal{G}} = \left[\sum_{\mathsf{v}\in\mathcal{V}}\mathbf{e}_{\mathsf{v}} \;\|\; \textsc{MaxPool}\left\{\mathbf{e}_{\mathsf{v}}\right\}_{\mathsf{v}\in\mathcal{V}}\right] \tag{11}$$

After that, ReGVD feeds $\mathbf{e}_{\mathcal{G}}$ to a single fully-connected layer followed by a $\mathsf{softmax}$ layer to predict whether a given source code is vulnerable or not as:

$$\hat{\mathbf{y}}_{\mathcal{G}} = \mathsf{softmax}\left(\mathbf{W}_1\mathbf{e}_{\mathcal{G}} + \mathbf{b}_1\right) \tag{12}$$

Finally, ReGVD is trained by minimizing the cross-entropy loss function. We illustrate our proposed ReGVD in Figure 2 and briefly present the learning process in ReGVD in Algorithm 1.

3 Experimental setup and results

In this section, we evaluate the benefits of our proposed ReGVD and address the following questions:

Algorithm 1: The learning process in ReGVD.

Input: A source code c with its label y
$\mathsf{g}(\mathcal{V}, \boldsymbol{X}, \boldsymbol{A}) \leftarrow$ BUILD_GRAPH (c)
$\mathbf{H}^{(0)} \leftarrow \boldsymbol{X}$
for $k = 0, 1, ..., K-1$ **do**
  $\mathbf{H}^{(k+1)} \leftarrow \mathbf{H}^{(k)} + \mathsf{GNN}\left(\boldsymbol{A}, \mathbf{H}^{(k)}\right)$
$\mathbf{e}_{\mathsf{v}} \leftarrow \sigma\left(\mathbf{w}^{\mathsf{T}}\mathbf{h}_{\mathsf{v}}^{(K)} + \mathsf{b}\right) \odot \mathsf{g}\left(\mathbf{W}\mathbf{h}_{\mathsf{v}}^{(K)} + \mathbf{b}\right)$
$\mathbf{e}_{\mathcal{G}} \leftarrow \mathsf{MIX}\left(\sum_{\mathsf{v}\in\mathcal{V}} \mathbf{e}_{\mathsf{v}}, \textsc{MaxPool}\left\{\mathbf{e}_{\mathsf{v}}\right\}_{\mathsf{v}\in\mathcal{V}}\right)$
$\mathsf{y} \leftarrow \mathsf{softmax}\left(\mathbf{W}_1\mathbf{e}_{\mathcal{G}} + \mathbf{b}_1\right)$

Q1 How does ReGVD compare to other state-of-the-art vulnerability detection methods?

Q2 Can the graph-level readout pooling layer proposed in our ReGVD work better than the more complex *Conv* pooling layer employed in Devign (Zhou et al., 2019)?

Q3 How is the influence of the residual connection, the mixture function, and the sliding window size on the GNNs performance?

Q4 Can ReGVD obtain satisfactory accuracy results even with limited training data?

3.1 Experimental setup

Dataset We use the standard benchmark dataset from CodeXGLUE (Lu et al., 2021) for vulnerability detection at the function level.[1] The dataset was firstly created by Zhou et al. (2019), including 27,318 manually-labeled vulnerable or non-vulnerable functions extracted from security-related commits in two large and popular C programming language open-source projects (i.e., QEMU and FFmpeg) and diversified in functionality. Since Zhou et al. (2019) did not provide official training/validation/test sets, Lu et al. (2021) combined these projects and then split into the training/validation/test sets. Table 1 reports the statistics of this benchmark dataset.

Table 1: Statistics of the dataset.

Dataset	#Instances
Training set	21,854
Validation set	2,732
Test set	2,732

Training protocol We construct a 2-layer model, set the batch size to 128, and employ the Adam optimizer (Kingma & Ba, 2014) to train our model up to 100 epochs. As mentioned in Section 2.3, we set the same hidden size ("hs") for the hidden GNN layers, wherein we vary the size value in {128, 256, 384}. We vary the sliding window size ("ws") in {2, 3, 4, 5} and the Adam initial learning rate ("lr") in $\{1e^{-4}, 5e^{-4}, 1e^{-3}\}$. The final accuracy on the test set is reported for the best model checkpoint, which obtains the highest accuracy on the validation set. Table 2 shows the optimal hyper-parameters for each setting in our ReGVD.

Baselines We compare our ReGVD with up-to-date strong baselines as follows:

- **BiLSTM** (Hochreiter & Schmidhuber, 1997) and **TextCNN** (Kim, 2014) are two well-known standard models applied for text classification. Since we consider a given source code as a flat sequence of tokens, we can adapt these two models for vulnerability detection.
- **RoBERTa** (Liu et al., 2019) is built based on BERT (Devlin et al., 2018) by removing the next-sentence pre-training objective and training on a massive dataset with larger mini-batches and learning rates.

[1] https://github.com/microsoft/CodeXGLUE/tree/main/Code-Code/Defect-detection

Table 2: The optimal hyper-parameters on the validation set for each ReGVD setting. "Const" denotes the graph construction method, wherein "Idx" and "UniT" denote the index-focused construction and the unique token-focused one, respectively. "Init" denotes the feature initialization, wherein "CB" and "G-CB" denote using only *the embedding layer* of CodeBERT and GraphCodeBERT to initialize the node features, respectively.

Const	Init	Base	lr	ws	MIX	hs
Idx	CB	GGNN	$1e^{-4}$	2	SUM	384
		GCN	$5e^{-4}$	2	MUL	384
	G-CB	GGNN	$1e^{-4}$	2	MUL	256
		GCN	$1e^{-4}$	2	SUM	128
UniT	CB	GGNN	$5e^{-4}$	2	MUL	256
		GCN	$5e^{-4}$	5	MUL	256
	G-CB	GGNN	$5e^{-4}$	3	MUL	384
		GCN	$5e^{-4}$	5	MUL	128

- **Devign** (Zhou et al., 2019) builds a multi-edged graph from a raw source code, then uses Gated GNNs (Li et al., 2016) to update node representations, and finally utilizes a 1-D CNN-based pooling ("*Conv*") to make a prediction.
- **CodeBERT** (Feng et al., 2020) is a bimodal pre-trained model also based on BERT for 6 programming languages (Python, Java, JavaScript, PHP, Ruby, Go), using the objectives of masked language model (Devlin et al., 2018) and replaced token detection (Clark et al., 2020).
- **GraphCodeBERT** (Guo et al., 2021) is a new pre-trained PL model, extending CodeBERT to consider the inherent structure of code data flow into the training objective.

We note that Zhou et al. (2019) did not release the official implementation of Devign. Thus, we re-implement Devign using our two graph construction methods and the same training protocol w.r.t. the optimizer, the window sizes, the initial learning rate values, and the number of training epochs.

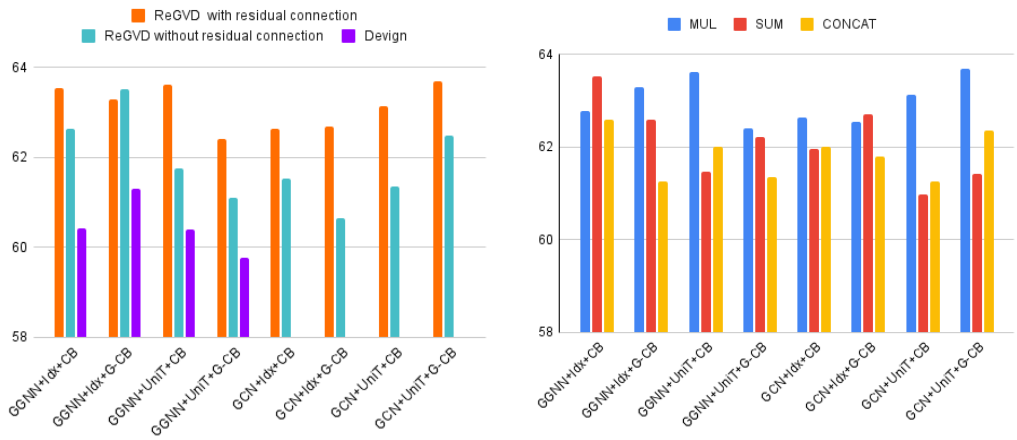
Table 3: Vulnerability detection accuracies (%) on the test set. The best scores are in bold, while the second best scores are in underline. The results of BiLSTM, TextCNN, RoBERTa, and CodeBERT are taken from (Lu et al., 2021). $\star$ denotes that we report our own results for other baselines.

Model	Accuracy (%)
BiLSTM	59.37
TextCNN	60.69
RoBERTa	61.05
CodeBERT	62.08
GraphCodeBERT*	62.30
Devign (Idx + CB)*	60.43
Devign (Idx + G-CB)*	61.31
Devign (UniT + CB)*	60.40
Devign (UniT + G-CB)*	59.77
ReGVD (GGNN + Idx + CB)	63.54
ReGVD (GGNN + Idx + G-CB)	63.29
ReGVD (GGNN + UniT + CB)	<u>63.62</u>
ReGVD (GGNN + UniT + G-CB)	62.41
ReGVD (GCN + Idx + CB)	62.63
ReGVD (GCN + Idx + G-CB)	62.70
ReGVD (GCN + UniT + CB)	63.14
ReGVD (GCN + UniT + G-CB)	**63.69**

3.2 MAIN RESULTS

Table 3 presents the accuracy results of the proposed ReGVD and the other up-to-date strong baselines on the standard benchmark dataset from CodeXGLUE for vulnerability detection regarding **Q1**. We note that TextCNN and RoBERTa outperform Devign, except the Devign setting (Idx+G-CB). Both the recent models CodeBERT and GraphCodeBERT obtain competitive performances and perform better than Devign, indicating the effectiveness of the pre-trained PL models. More importantly, ReGVD gains absolute improvements of 1.61% and 1.39% over CodeBERT and GraphCodeBERT, respectively. This shows the benefit of ReGVD in learning the local structures inside the source code to differentiate vulnerabilities (w.r.t using only the embedding layer of the pre-trained PL model). Hence, our ReGVD significantly outperforms the up-to-date baseline models. In particular, ReGVD produces the highest accuracy of 63.69% – a new state-of-the-art result on the CodeXGLUE vulnerability detection dataset.

In our pilot studies, we achieve higher results for our ReGVD by feeding the flat sequence of tokens as an input for the pre-trained PL model to obtain the contextualized embeddings, which are then used to initialize the node feature vectors. But for a fair comparison, we use only the embedding layer of the pre-trained PL model to initialize the feature vectors.



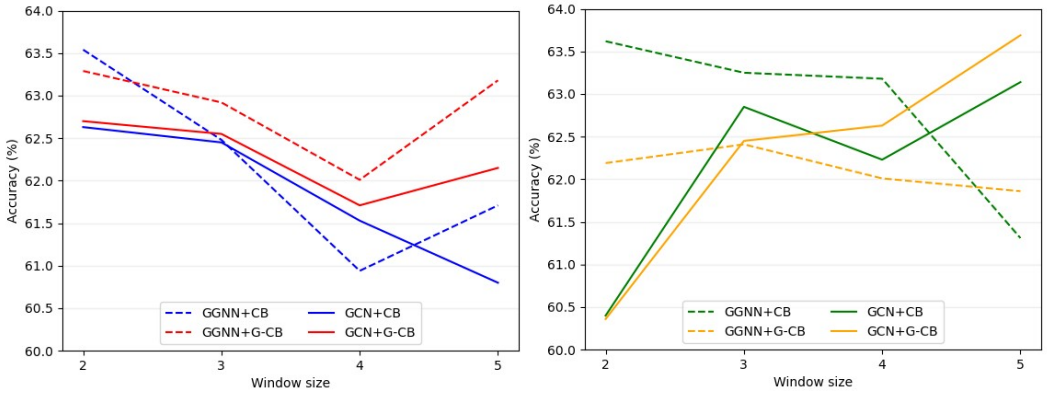
(a) Accuracy with and without residual connection.

(b) Accuracy w.r.t the MIX functions.

Figure 3: Accuracy with different settings.

We look at Figure 3a to address **Q2** to investigate whether the graph-level readout layer proposed in ReGVD performs better than the more complex *Conv* pooling layer utilized in Devign. Since Devign also uses Gated GNNs to update the node representations and gains the best accuracy of 61.31% for the setting (Idx+G-CB); thus, we consider the ReGVD setting (GGNN+Idx +G-CB) without using the residual connection for a fair comparison, wherein ReGVD achieves an accuracy of 63.51%, which is 2.20% higher accuracy than that of Devign. More generally, we get a similar conclusion from the results of three remaining ReGVD settings (without using the residual connection) that the graph-level readout layer utilized in ReGVD outperforms that used in Devign.

We analyze the influence of the residual connection, the mixture function, and the sliding window size to answer **Q3**. We first look back Figure 3a for the ReGVD accuracies w.r.t with and without using the residual connection among the GNN layers. It demonstrates that the residual connection helps to boost the GNNs performance on seven settings, where the maximum accuracy gain is 2.05% for the ReGVD setting (GCN+Idx+G-CB). Next, we look at Figure 3b for the ReGVD results w.r.t the MIX functions. We find that ReGVD generally gains the highest accuracies on six settings using the MUL operator and on two remaining settings using the SUM operator. But it is worth noting that the ReGVD setting (GGNN+Idx+CB) using the CONCAT operator obtains an accuracy of 62.59%, which is still higher than that of Devign, CodeBERT, and GraphCodeBERT. Then, we check the results shown in Figure 4 to explore the influence of the sliding window size on the accuracy performance w.r.t the graph construction method. We see that using smaller sizes pro-



(a) Index-focused graph construction.

(b) Unique token-focused graph construction.

Figure 4: Accuracy with varying window sizes.

duces better accuracies than using the larger ones regarding the index-focused construction method, otherwise regarding the unique token-focused construction method. We also find that the window size 3 gives stable results and the highest average accuracy of 62.67% over all eight settings.

We test the best performing settings with different percents of the training data regarding **Q4**. Figure 5 shows the test accuracies when training ReGVD with 20%, 40%, 60%, 80%, and full training sets. Our model achieves satisfactory performance with limited training data, compared to the baselines using the full training data. For example, ReGVD obtains an accuracy of 61.68% with 60% training set, which is higher than that of BiLSTM, TextCNN, RoBERTa, and Devign. It also achieves an accuracy of 62.55% with 80% training set, which is better than that of CodeBERT and GraphCodeBERT.

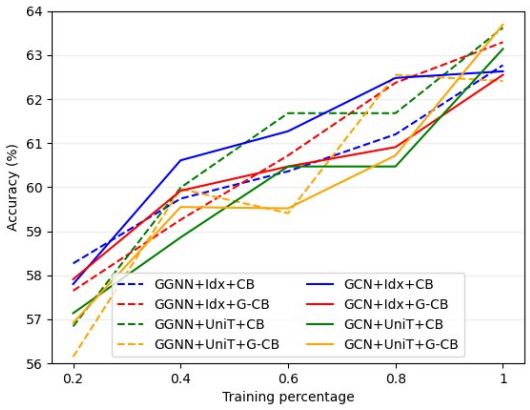


Figure 5: Accuracy with different percents of the training set.

4 Conclusion

We introduce a novel graph neural network-based model ReGVD to detect vulnerabilities in source code. ReGVD transforms each raw source code into a single graph to benefit the local structures inside the source code, through two effective graph construction methods, wherein ReGVD utilizes only the embedding layer of the pre-trained programming language model to initialize node feature vectors. ReGVD then makes a novel use of residual connection among GNN layers and an useful mixture of the sum and max poolings to learn better code graph representation. To demonstrate the effectiveness of ReGVD, we conduct extensive experiments with to compare ReGVD with the up-to-date strong baselines on the benchmark vulnerability detection dataset from CodeXGLUE. Experimental results show that that the proposed ReGVD significantly performs better than the baseline models and obtains the highest accuracy of 63.69% on the standard dataset.

ReGVD can be seen as a general, unified and practical framework. Future work can adapt ReGVD for similar classification tasks such as clone detection. Furthermore, we plan to extend and combine ReGVD using the index-focused graph construction with a BERT-style model to build an encoder for other programming language tasks such as code completion and code translation.

REFERENCES

Cătălina Cangea, Petar Veličković, Nikola Jovanović, Thomas Kipf, and Pietro Liò. Towards sparse hierarchical graph classifiers. *arXiv preprint arXiv:1811.01287*, 2018.

Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder–decoder for statistical machine translation. In *EMNLP*, 2014.

Kevin Clark, Minh-Thang Luong, Quoc V Le, and Christopher D Manning. Electra: Pre-training text encoders as discriminators rather than generators. *arXiv preprint arXiv:2003.10555*, 2020.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. Graphcodebert: Pre-training code representations with data flow. In *International Conference on Learning Representations*, 2021.

William L. Hamilton, Rex Ying, and Jure Leskovec. Representation learning on graphs: Methods and applications. *arXiv:1709.05584*, 2017.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, pp. 770–778, 2016.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9:1735–1780, 1997.

Lianzhe Huang, Dehong Ma, Sujian Li, Xiaodong Zhang, and Houfeng Wang. Text level graph neural network for text classification. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pp. 3444–3450, 2019.

Yoon Kim. Convolutional Neural Networks for Sentence Classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, pp. 1746–1751, 2014.

Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *ICLR*, 2017.

Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated Graph Sequence Neural Networks. *ICLR*, 2016.

Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation. *CoRR*, abs/2102.04664, 2021.

Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. Predicting vulnerable software components. In *Proceedings of the 14th ACM conference on Computer and communications security*, pp. 529–540, 2007.

Dai Quoc Nguyen, Tu Dinh Nguyen, and Dinh Phung. Universal graph transformer self-attention networks. *arXiv preprint arXiv:1909.11855*, 2019.

Viet Hung Nguyen and Le Minh Sang Tran. Predicting vulnerable software components with dependency graphs. In *Proceedings of the 6th International Workshop on Security Measurements and Metrics*, pp. 1–8, 2010.

Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. 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, 2018.

Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.

Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE transactions on software engineering*, 37(6):772–787, 2010.

Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S Yu. A comprehensive survey on graph neural networks. *arXiv:1901.00596*, 2019.

Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How Powerful Are Graph Neural Networks? *ICLR*, 2019.

Liang Yao, Chengsheng Mao, and Yuan Luo. Graph convolutional networks for text classification. In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, pp. 7370–7377, 2019.

Rex Ying, Jiaxuan You, Christopher Morris, Xiang Ren, William L. Hamilton, and Jure Leskovec. Hierarchical graph representation learning with differentiable pooling. In *NeurIPS*, pp. 4805–4815, 2018.

Yufeng Zhang, Xueli Yu, Zeyu Cui, Shu Wu, Zhongzhen Wen, and Liang Wang. Every document owns its structure: Inductive text classification via graph neural networks. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 334–339, 2020.

Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Advances in Neural Information Processing Systems*, 2019.