REGVD: REVISITING GRAPH NEURAL NETWORKS FOR VULNERABILITY DETECTION

Van-Anh Nguyen[1,*], **Dai Quoc Nguyen**[2,†,*], **Van Nguyen**[3], **Trung Le**[3], **Quan Hung Tran**[4], **Dinh Phung**[3]
[1]VNU - University of Science, Vietnam; [2]Oracle Labs, Australia
[3]Monash University, Australia; [4]Adobe Research, San Jose, CA, USA
[1]vananhnt57@gmail.com; [2]dai.nguyen@oracle.com
[3]{van.nk,trunglm,dinh.phung}@monash.edu; [4]qtran@adobe.com

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. To this end, we aim to develop a general, practical, and programming language-independent model capable of running on various source codes and libraries without difficulty. Therefore, we consider vulnerability detection as an inductive text classification problem and propose ReGVD, a simple yet effective graph neural network-based model for the problem. In particular, ReGVD views each raw source code as a flat sequence of tokens to build a graph, wherein node features are initialized by only the token embedding layer of a pre-trained programming language (PL) model. ReGVD then leverages residual connection among GNN layers and examines a mixture of graph-level sum and max poolings to return a graph embedding for the source code. ReGVD outperforms the existing state-of-the-art models and obtains the highest accuracy on the real-world benchmark dataset from CodeXGLUE for vulnerability detection. Our code is available at: https://github.com/daiquocnguyen/GNN-ReGVD.

1 INTRODUCTION

The software vulnerability problems have 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, creating 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 open-source codes and libraries evolving over time.

To reduce human efforts on feature engineering, some 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 detecting vulnerabilities. Recently, pre-trained language models such as BERT (Devlin et al., 2018) have emerged as a trending learning paradigm, achieving significant success in NLP applications. Inspired by this BERT-style trending paradigm, pre-trained programming language (PL) models such as CodeBERT (Feng et al., 2020) have improved

[*]The first two authors contributed equally to this work. [†]Corresponding author. This work was done before Dai Quoc Nguyen joined Oracle Labs, Australia. The final authenticated version is available online at https://doi.org/10.1145/3510454.3516865.

the performance of PL downstream tasks such as 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; thus, this limits learning local structures within the source code to differentiate vulnerabilities.

Graph neural networks (GNNs) have recently become a primary method to embed nodes and graphs into low-dimensional continuous vector spaces (Hamilton et al., 2017; Wu et al., 2019; Nguyen, 2021). 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; Huang et al., 2019; Zhang et al., 2020; Nguyen et al., 2021). Devign (Zhou et al., 2019) is proposed to utilize Gated GNNs (Li et al., 2016) for vulnerability detection, wherein Devign uses a PL parser to extract multi-edged graph information. However, Devign is difficult of being practiced in reality. The main reason is that there is not a perfect parser in reality for each PL, which can successfully parse a variety of source codes and libraries without any internal compile errors and exceptions.

In this paper, our goal is to develop a general, practical, and programming language-independent model capable of running on various source codes and libraries without difficulty. Hence, we consider vulnerability detection as an inductive text classification problem and introduce ReGVD – a simple yet effective GNN-based model for vulnerability detection as follows: (i) ReGVD views each raw source code as a flat sequence of tokens to construct a graph (in Section 2.2), wherein node features are initialized by only the token embedding layer of a pre-trained PL model. (ii) ReGVD leverages GNNs (such as GCNs (Kipf & Welling, 2017) or Gated GNNs (Li et al., 2016)) using residual connection among GNN layers (in Section 2.3). (iii) ReGVD examines a mixture between the sum and max poolings to produce a graph embedding for the source code (in Section 2.4). This graph embedding is fed to a single fully-connected layer followed by a softmax layer to predict the code vulnerabilities. Extensive experiments show that ReGVD significantly outperforms the existing state-of-the-art models on the benchmark vulnerability detection dataset from CodeXGLUE (Lu et al., 2021). ReGVD produces the highest accuracy of 63.69%, gaining absolute improvements of 1.61% and 1.39% over CodeBERT and GraphCodeBERT, respectively; thus, ReGVD can act as a new strong baseline for future work.

2 The proposed ReGVD

2.1 Problem definition

We consider vulnerability detection 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, $\mathbb{Y} = \{0, 1\}$ denotes the label set with 1 for vulnerable and 0 otherwise, and n is the number of instances. In this work, we consider vulnerability detection as an inductive text classification problem and leverage GNNs for the problem. Therefore, we construct a graph $\mathsf{g}_i(\mathcal{V}, \boldsymbol{X}, \boldsymbol{A}) \in \mathcal{G}$ for each 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 $\mathsf{v}_j \in \mathcal{V}$ 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 $f : \mathcal{G} \to \mathbb{Y}$ 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:

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

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

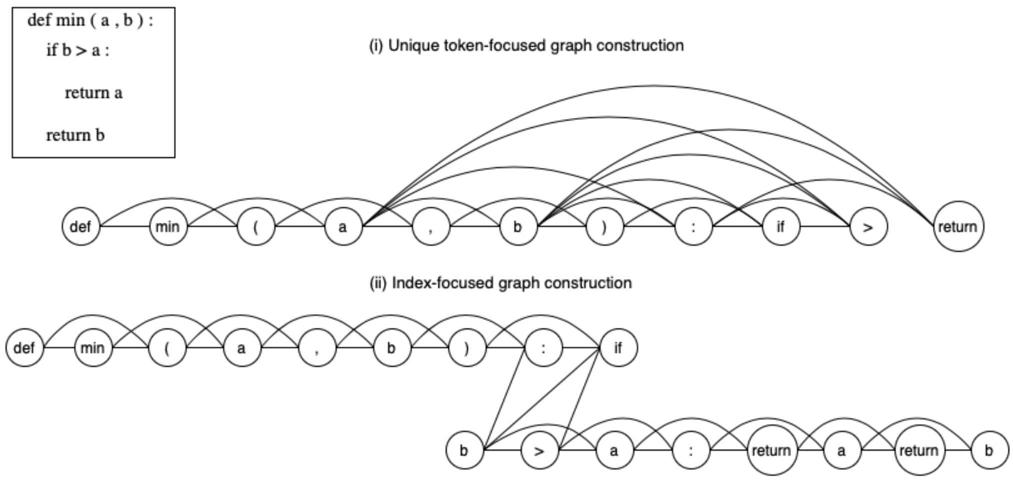


Figure 1: An illustration for two graph construction methods with a fixed-size sliding window of length 3.

2.2 GRAPH CONSTRUCTION

We consider a raw source code as a flat sequence of tokens and illustrate two graph construction methods (Huang et al., 2019; Zhang et al., 2020) in Figure 1, wherein we omit self-loops in these two methods since the self-loops do not help to improve performance in our pilot experiments.[1]

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:

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

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:

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

Node feature initialization It is worth noting that pre-trained programming language (PL) models such as CodeBERT (Feng et al., 2020) have recently improved the performance of PL downstream tasks such as vulnerability detection. To make *a fair comparison*, we use *only the token embedding layer* of the pre-trained PL model to initialize node feature vectors for reporting our final results.

2.3 GRAPH NEURAL NETWORKS WITH RESIDUAL CONNECTION

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 simply formulate GNNs as follows:

$$\mathbf{H}^{(k+1)} = \mathsf{GNN}\left(\boldsymbol{A}, \mathbf{H}^{(k)}\right)$$

[1]In our implementation, we firstly tokenize the source code using the corresponding tokenizer of the pre-trained PL model, and then we construct the graph from the tokenized sequence.

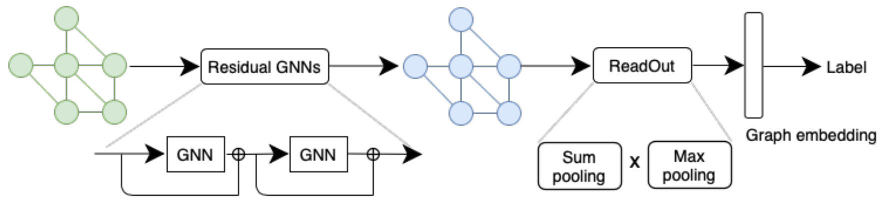


Figure 2: An illustration for our proposed ReGVD.

where $\mathsf{H}^{(k)}$ is the matrix representation of nodes at the k-th iteration/layer; and $\mathsf{H}^{(0)} = \boldsymbol{X}$. 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:

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

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:

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

where $\mathbf{z}$ and $\mathbf{r}$ are the update and reset gates; σ is the sigmoid function; and $\odot$ is the 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. Motivated by that, we follow (Bresson & Laurent, 2017) to adapt residual connection among the GNN layers, with fixing the same hidden size for the different layers. In particular, ReGVD redefines GNNs as:

$$\mathsf{H}^{(k+1)} = \mathsf{H}^{(k)} + \mathsf{GNN}\left(\boldsymbol{A}, \mathsf{H}^{(k)} \right)$$

2.4 GRAPH-LEVEL READOUT POOLING LAYER

The graph-level readout layer is used to produce a graph embedding for each input graph. ReGVD leverages the sum pooling as it produces better results for graph classification (Xu et al., 2019).[2] Besides, ReGVD utilizes the max pooling to exploit more information on the node representations.

[2]In our pilot studies, using the sum pooling $\sum_{\mathsf{v} \in \mathcal{V}} \mathbf{e}_{\mathsf{v}}$ also provides higher accuracies than using the mean pooling $\frac{1}{|\mathcal{V}|} \sum_{\mathsf{v} \in \mathcal{V}} \mathbf{e}_{\mathsf{v}}$ employed in (Zhang et al., 2020).

ReGVD then considers a mixture between the sum and max poolings to produce the graph embedding $\mathbf{e}_\mathsf{g}$ as:

$$\begin{aligned}\mathbf{e}_\mathsf{v} &= \sigma\left(\mathbf{w}^\mathsf{T}\mathbf{h}_\mathsf{v}^{(K)} + \mathsf{b}\right) \odot \phi\left(\mathbf{W}\mathbf{h}_\mathsf{v}^{(K)} + \mathbf{b}\right) \\ \mathbf{e}_\mathsf{g} &= \mathsf{MIX}\left(\sum_{\mathsf{v}\in\mathcal{V}} \mathbf{e}_\mathsf{v}, \textsc{MaxPool}\left\{\mathbf{e}_\mathsf{v}\right\}_{\mathsf{v}\in\mathcal{V}}\right)\end{aligned}$$

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:

$$\begin{aligned}\mathsf{SUM} &: \mathbf{e}_\mathsf{g} = \sum_{\mathsf{v}\in\mathcal{V}} \mathbf{e}_\mathsf{v} + \textsc{MaxPool}\left\{\mathbf{e}_\mathsf{v}\right\}_{\mathsf{v}\in\mathcal{V}} \\ \mathsf{MUL} &: \mathbf{e}_\mathsf{g} = \sum_{\mathsf{v}\in\mathcal{V}} \mathbf{e}_\mathsf{v} \odot \textsc{MaxPool}\left\{\mathbf{e}_\mathsf{v}\right\}_{\mathsf{v}\in\mathcal{V}} \\ \mathsf{CONCAT} &: \mathbf{e}_\mathsf{g} = \left[\sum_{\mathsf{v}\in\mathcal{V}} \mathbf{e}_\mathsf{v} \;\|\; \textsc{MaxPool}\left\{\mathbf{e}_\mathsf{v}\right\}_{\mathsf{v}\in\mathcal{V}}\right]\end{aligned}$$

After that, ReGVD feeds $\mathbf{e}_\mathsf{g}$ to a single fully-connected layer followed by a $\mathsf{softmax}$ layer to predict whether the source code is vulnerable or not as: $\hat{\mathbf{y}}_\mathsf{g} = \mathsf{softmax}\left(\mathbf{W}_1\mathbf{e}_\mathsf{g} + \mathbf{b}_1\right)$ Finally, ReGVD is trained by minimizing the cross-entropy loss function as mentioned in Section 2.1. We illustrate the proposed ReGVD in Figure 2.

3 Experimental setup and results

3.1 Experimental setup

Dataset We use the real-world benchmark from CodeXGLUE (Lu et al., 2021) for vulnerability detection at the function level.[3] 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. Then Lu et al. (2021) combined these projects and then split into the training/validation/test sets.

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 $\left\{1e^{-4}, 5e^{-4}, 1e^{-3}\right\}$. The final accuracy on the test set is reported for the best model checkpoint, which obtains the highest accuracy on the validation set. Table 1 shows the optimal hyper-parameters for each setting in our ReGVD.

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

- **BiLSTM** (Hochreiter & Schmidhuber, 1997) and **TextCNN** (Kim, 2014) are two well-known standard models applied for text classification.
- **RoBERTa** (Liu et al., 2019) is built based on BERT (Devlin et al., 2018) by removing the next-sentence objective and training on a massive dataset with larger mini-batches and learning rates.
- **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

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

Table 1: The optimal hyper-parameters on the validation set for each ReGVD setting. "Const" denotes the 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 token 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

CNN-based pooling ("*Conv*") to make a prediction. We note that Zhou et al. (2019) did not release the official implementation of Devign. Thus, we re-implement Devign using the same training and evaluation protocols.

- **CodeBERT** (Feng et al., 2020) is a pre-trained model also based on BERT for 6 programming languages (Python, Java, JavaScript, PHP, Ruby, Go), using masked language model (Devlin et al., 2018) and replaced token detection (Clark et al., 2020) objectives.
- **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.

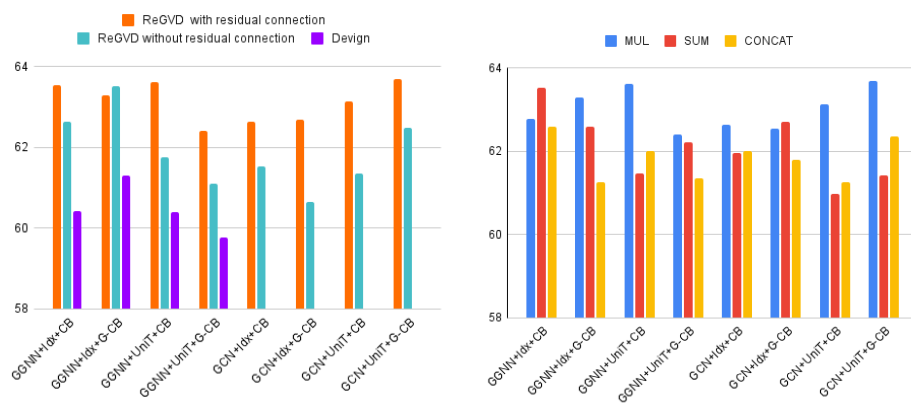
3.2 Main results

Table 2: 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**

Table 2 presents the accuracy results of the proposed ReGVD and the strong and up-to-date baselines on the real-world benchmark dataset from CodeXGLUE for vulnerability detection. We note that 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 token 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.

We look at Figure 3a to investigate whether the graph-level readout layer proposed in ReGVD performs better than the *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.



(a) Accuracy with and without residual connection.

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

Figure 3: Accuracy with different settings.

We analyze the influence of the residual connection and the mixture function. 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.

4 Conclusion

We consider vulnerability identification as an inductive text classification problem and introduce a simple yet effective graph neural network-based model, named ReGVD, to detect vulnerabilities in source code. ReGVD transforms each raw source code into a graph, wherein ReGVD utilizes only the token embedding layer of the pre-trained programming language model to initialize node feature vectors. ReGVD then leverages residual connection among GNN layers and a mixture of the sum and max poolings to learn graph representation. To demonstrate the effectiveness of ReGVD, we conduct extensive experiments to compare ReGVD with the strong and up-to-date baselines on the benchmark vulnerability detection dataset from CodeXGLUE. Experimental results show that the proposed ReGVD is significantly better than the baseline models and obtains the highest accuracy of 63.69% on the benchmark dataset. ReGVD can be seen as a general, practical, and programming language-independent model that can run on various source codes and libraries without difficulty.

REFERENCES

Xavier Bresson and Thomas Laurent. Residual gated graph convnets. *arXiv preprint arXiv:1711.07553*, 2017.

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*, pp. 1724—-1734, 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, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 1536–1547, 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 *ICLR*, 2021.

William L. Hamilton, Rex Ying, and Jure Leskovec. Representation learning on graphs: Methods and applications. *preprint 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 *EMNLP-IJCNLP*, 2019.

Yoon Kim. Convolutional neural networks for sentence classification. In *EMNLP*, 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. In *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. *arXiv preprint arXiv:2102.04664*, 2021.

Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. Predicting vulnerable software components. In *ACM CCS*, pp. 529–540, 2007.

Dai Quoc Nguyen. *Representation Learning for Graph-Structured Data*. PhD thesis, Monash University, 2021.

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

Dai Quoc Nguyen, Tu Dinh Nguyen, and Dinh Phung. Quaternion graph neural networks. In *Asian Conference on Machine Learning*, 2021.

Viet Hung Nguyen and Le Minh Sang Tran. Predicting vulnerable software components with dependency graphs. In *MetriSec*, 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 *ICMLA*, 2018.

Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20:61–80, 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, 2010.

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

Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *ICLR*, 2019.

Liang Yao, Chengsheng Mao, and Yuan Luo. Graph convolutional networks for text classification. In *AAAI*, pp. 7370–7377, 2019.

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 *ACL*, 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 *NeurIPS*, 2019.