ISBN: 978-981-18-1791-5
2021 the 11th International Workshop on Computer Science and Engineering (WCSE 2021)
doi: 10.18178/wcse.2021.06.002

Static Detection of Vulnerabilities via Graph Attention hierarchically

Yuhan Zhang [1,2], Xueyang Liu[1+], Dongdong Du[3+]

[1] National Engineering Research Center for Software Engineering, Peking University, China

[2] School of Software and Microelectronics, Peking University, China

[3] China Academy of Industrial Internet, Beijing, China

Abstract. With the rapid growth of the software industry, the risks of vulnerabilities are inevitably increasing. Deep learning based methods have been widely used in vulnerability detection in recent years. Since the inherent graph structure of source code contains rich semantics, many deep learning works have exploited graph neural networks to enhance code representation. Despite their novel design, learning the structural information in the graph hierarchically and focusing on important nodes are still problems to better capture vulnerability semantics. To tackle this bottleneck, we propose a novel neural model for vulnerability detection. A SAGPool module is designed to automatically chooses important nodes to retain hierarchically in each graph convolution layer. Our model is trained and tested over the REVEAL dataset built on two popular and well-maintained open-source projects. The experimental results demonstrate that our model outperforms the state-of-the-art methods.

Keywords: code vulnerabilities, graph neural network, attention, etc.

1. Introduction

Internet and software have gradually become indispensable tools for human society. Due to the massively growing number of software users and more prosperous software functions, the software's complexity increases dramatically, which inevitably increases the security risks of software systems.

However, it is challenging and tedious to detect vulnerabilities even for developers with specialized security expertise. Thus, automatic detection of vulnerabilities in source code has attracted a great research focus. Traditional techniques like static analysis, dynamic analysis, and symbolic execution rely on expert knowledge, resulting in high labor costs. Deep learning and machine learning have demonstrated their extraordinary ability to summarize from samples and deal with noise. The feature mining and representation capabilities of deep neural networks also provide an effective technical approach to detect software vulnerabilities automatically.

Several studies have explored the potential of applying deep learning techniques to detect vulnerabilities. However, most of these works have limitations in learning the semantic information. On one hand, there are many substructures such as paths, trees and cycles which give the meaningful information, message-passing in graph neural network may lost some semantic information. On the other hand, different nodes in the code representation graphs contribute differently in the representation of the code graph. Thus, they do not learn the hierarchical representations which are crucial for capturing structural information of graphs.

To this end, we propose a novel model based on the graph attention neural network, which concentrates on hierarchically learning the representation of the code graph. The key innovation is leveraging Self Attention Graph Pool(SAGPool)[1] with the graph neural network, which automatically distinguishes between nodes that should be dropped and the nodes that should be retained in each layer. We choose the REVEAL[2] dataset collected from the realistic data source to make the model more generalizable. Besides, the code property graph (CPG) is leveraged to build the graph representation of the input code, for capturing more semantics of the source code. CPG combines multiple semantics dimensions, like control flow dependencies, data flow dependencies and so on. So it is suitable for the vulnerability detection task.

+ Corresponding author.
E-mail address: liuxueyang@pku.edu.cn. dudongdong@china-aii.com.

In summary, our contributions in this paper are:

We propose the graph neural network model with SAGPool[1] for graph-level classification. The SAGPool module chooses important nodes to retain in each layer. Thus, the graph's representation is learned hierarchically, which will focus more on the crucial nodes and capture the structure information.

We have conducted our experiments by comparing our model with the state-of-the-art vulnerability detection approaches based on deep learning. The results show the effectiveness of our approach in terms of both F1 score and accuracy.

2. Related Work

Deep learning based vulnerability detection methods learn different vulnerability patterns from a train dataset, all samples of which are labelled as vulnerable (label 1) or benign (label 0). Most current approaches consist of three steps: (1) generate the intermediate representations of the code from the source code (2) design suitable model to transform the intermediate representations into vectors (3) output the prediction from the classifier according to the vector representations. In the light of different intermediate representations of the code, there are two main kinds of approaches:

2.1. Token-based Models

Token-based models just regard source code as a token sequence. Russel et al.[3] directly input the code sequence of a function into the model, learning the embedding of the code by the Convolution Neural Network (CNN) combined with a Random Forest (RF) module. The simple way to process the code lead to the length of the sequence affecting the performance of the model. To deal with it, Li et al. propose VulDeePecker[4] and SySeVR[5] to extract the code slices as the input of the model. A code slice composes of some program statements. These statements are semantically related in data dependency or control dependency. VulDeePecker applies a Bidirectional Long Short Term Memory (Bi-LSTM) [6] and SySeVR uses a Bidirectional Gate Recurrent Unit (Bi-GRU)[7] to capture the semantics of the context in the code slice. Even though the code slice contains some semantic relations, statements semantically related to each other may be far away, which makes the model hard to learn their relations.

2.2. Graph-based Models

Graph-based models leverage graph representation of code, such as control flow graph (CFG), program dependency graph (PDG), code property graph (CPG). Devign[8] get four subgraphs from the CPG with respect to different edge types to capture multi-dimensions of semantics. Accordingly, they use gated graph neural network (GGNN)[9] to learn the representation of the subgraphs, and combine them by a convolution module. Saikat et al. [2] make use of similar model to learn the representation of the CPG. On top of this, they employ the representation learning[10] to increase the class separation between vulnerable samples and benign samples. All these methods make full use of the semantic graph information of the code, but learn the ‘flat’ representation of the graph.

3. Proposed Model

In this section, we first present an overview of the architecture of our model. Then we introduce each component of the model in detail.

3.1. Overview

In our model, we formalize the prediction of vulnerable code as a binary classification problem. Figure 1 shows the architecture of our proposed model. Firstly, we extract the features from the code property graph (CPG) of the function code. Then, we learn the embedding of the CPG by the model. Finally, the model predicts whether there are vulnerabilities in the code by a Multi-layer Perception (MLP) classifier.

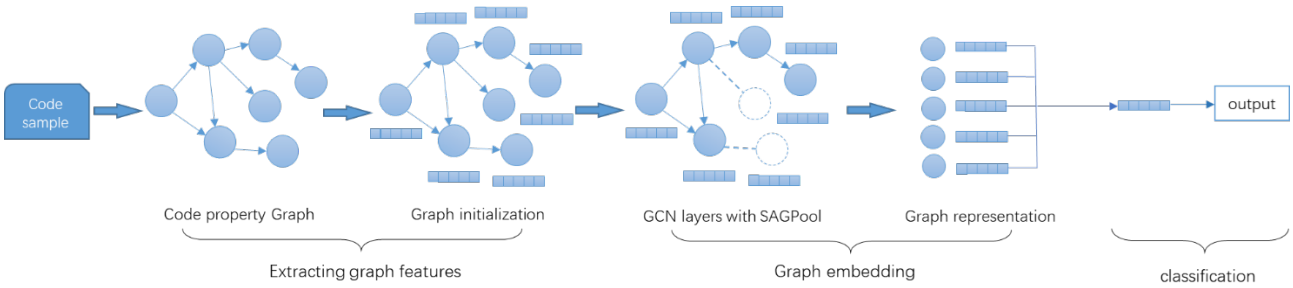

Fig. 1: Overview Architecture of the Model

3.2. Extracting Graph Features

To get the syntax and semantics features of the code, we choose CPG as the intermediate representation, which contains the structure information of the whole code. Figure 2 shows an example source code of a function and its corresponding CPG, and the statement highlighted in red contains vulnerability.

In the situation that every node has both type attribute (i.e. *Patameter*, *Assignment*, *Ifstatement*) and source code attribute, for node i, we use one-hot to encode the type of the node as T_i and use word2vec to encode the source code segment as C_i. Specifically, C_i is calculated by adding the vectors of all tokens in the code segment. The finally representation of node i is denoted as $h_i^0 = [T_i \,||C_i]$, where $||$ means concatenate operation. It is used to initialize the input node features $H \in \mathbb{R}^{N\times F}$ for GNN module, where N denotes the number of nodes in the CPG and F denotes the size of the node feature vectors.

```
static void cpu_request_exit(void* opaque, int irq, int level) {
    CPUPPCState *env = cpu_single_env;
    if (env && level) {
        cpu_exit(CPU(ppc_env_get_cpu(env)));
    }
}
```

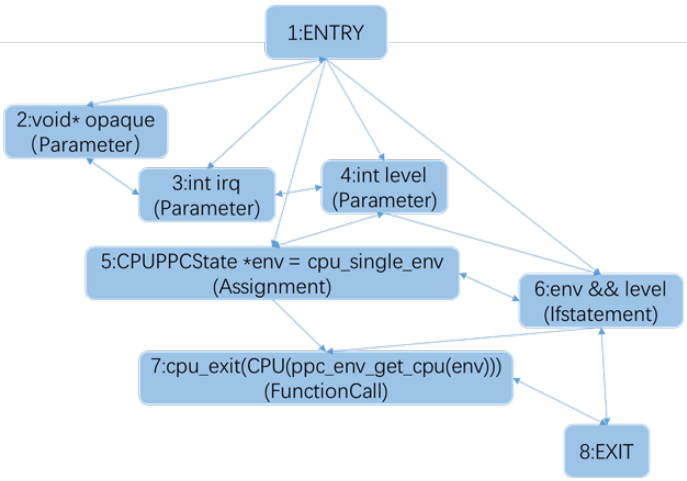

Fig. 2: An Example of a Function and the Corresponding CPG

3.3. Graph Embedding with SAGPool

We use Graph Convolution Network (GCN) with SAGPool[1] to learn the embedding of the graph. The process is shown in Figure 3, and the CPG in Figure 2 is simplified via numbers. The key point of SAGPool is that it uses GCN to calculate self-attention scores. Attention mechanisms make models focus more on important features and less on unimportant features. Specifically, self-attention allows input features to be the criteria for the attention itself. SAGPool obtains self-attention scores using graph convolution. The GCN formula[13] used in the model is:

$$GNN(H, A) = \widetilde{D}^{-\frac{1}{2}} A \widetilde{D}^{-\frac{1}{2}} H \tag{1}$$

Accordingly, attention score $Z \in \mathbb{R}^{N\times 1}$ is calculated using the node features output from the GCN layer:

$$Z = \sigma(GNN(H, A)\Theta_{att}) \tag{2}$$

where $A \in \mathbb{R}^{N\times N}$ is the adjacency matrix, $\widetilde{D} \in \mathbb{R}^{N\times N}$ is the degree matrix of A, and $\Theta_{att} \in \mathbb{R}^{F\times 1}$ is the parameter for SAGPool layer.

SAGPool selects nodes by retaining a portion of nodes of the input graph. The hyperparameter $k \in (0,1]$,which is called pooling ratio, determines the number of nodes to keep, and the top $\lceil kN \rceil$ nodes are selected according to the value of attention scores in Z.

$$idx = topRank(Z, \lceil kN \rceil), Z_{mask} = Z_{idx} \tag{3}$$

Where $topRank$ is the function that returns the indices of the top $\lceil kN \rceil$ values, $\cdot_{idx}$ is the indexing operation and Z_{mask} is the feature attention mask. In Figure 3, the model chooses nodes 1,5,6,7,8, which contain more semantic information of the function.

After getting the self-attention mask, SAGPool masks the input graph for graph pooling. The equation for calculating the pooled graph is:

$$H' = H_{idx,:}, H_{out} = H' \odot Z_{mask}, A_{out} = A_{idx,idx} \tag{4}$$

Where $H_{idx,:}$ is the row-wise indexed feature matrix, $\odot$ is the broadcasted elementwise product, and $A_{idx,idx}$ is the row-wise and col-wise indexed adjacency matrix. H_{out} and A_{out} are the new feature matrix and the corresponding adjacency matrix.

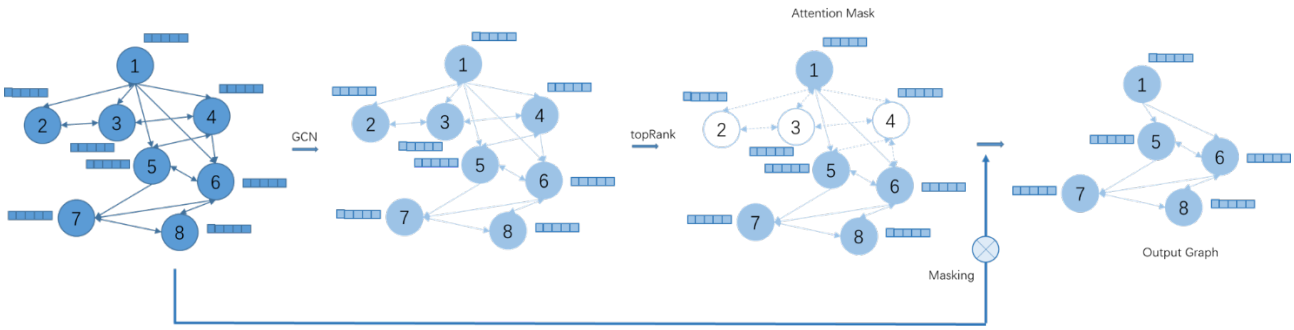

Fig. 3: An Illustration of the GCN Layer and SAGPool Layer

After T GCN layers with SAGPool, we get the features of all nodes in the CPG, and the embedding of the graph will be calculated by the sum of all node features in last layer, denoted as h_{graph}.

3.4. Training

In the code dataset from the real scene, the number of vulnerable samples is far smaller than that of non-vulnerable samples. In REVEAL[2], only 9.16% of the samples are vulnerable. To improve the performance, we over-sample vulnerable samples to alleviate the data imbalance. In addition, the triplet loss[11] is adopted to increase the discrimination between positive and negative samples.

In the classification layer, the embedding of CPG, h_{graph}, is firstly projected to a latent space by a dense layer $f(\cdot)$. The total loss is composed of three loss functions:(1) cross entropy loss $\mathcal{L}_{ce}$ (2) triplet loss $\mathcal{L}_{tri}$ (3) regularization loss $\mathcal{L}_{reg}$, i.e.:

$$\mathcal{L} = \mathcal{L}_{ce} + \alpha\mathcal{L}_{tri} + \beta\mathcal{L}_{tri} \tag{5}$$

$$\mathcal{L}_{ce} = -\sum\left(\hat{y} \cdot log(y) + (1-\hat{y}) \cdot log(1-y)\right) \tag{6}$$

$$\mathcal{L}_{tri} = \left|\mathbb{D}\left(f(h_{graph}), f(h_s)\right) - \mathbb{D}\left(f(h_{graph}), f(h_d)\right) + \gamma\right| \tag{7}$$

$$\mathcal{L}_{reg} = \| f(h_{graph}) \| + \| f(h_s) \| + \| f(h_d) \| \tag{8}$$

$$\mathbb{D}(v_1, v_2) = 1 - \left|\frac{v_1 \cdot v_2}{\| v_1 \| * \| v_1 \|}\right| \tag{9}$$

$\mathcal{L}_{ce}$ penalizes miss-classifications between the predict label $\hat{y}$ and the true label y. $\mathcal{L}_{tri}$ makes examples from the same class close and examples from different class far away from each other. Here, h_s is graph embedding from the same class, while h_d is graph embedding from different class. $\mathbb{D}(\cdot)$ represents the

cosine distance between two vectors. γ defines the minimum separation boundary between classes. The goal of our task is to learn parameters of the model by minimizing the loss function.

4. Experiment and Analysis

4.1. Experimental Setup

Dataset and Evaluate Metrics

In order to evaluate our model more realistically, we select the dataset REVEAL gathered from past vulnerabilities of open-source projects Linux Debian Kernel and Chromium. The total number of samples in the dataset is 18169 with 9.16% vulnerable samples. Every sample in the dataset is a function collected from vulnerability-fix commits. To get the graph representations of the code samples, we use Joern[12] to extract CPGs. We evaluate the approaches on four widely used metrics for binary classification tasks, i.e., Accuracy, Precision, Recall, and F1-score.

Experimental Settings

We use Pytorch 1.5.1 with Cuda 10.1 to implement our experiment. We follow the implementation of baseline models in [2]. For the GCN layer, the input embedding size is 169, and we set the hidden size to 256. The number of GCN and SAGPool layers is 3, and the learning rate is 0.001. The hyperparameters in the loss function, i.e., α, β, γ are set to 0.5, 0.001, 0.5. We use the Adam optimizer for all models.

4.2. Research Question

A. How does our proposed approach perform when compared with state-of-the-art models?

To answer this question, we compare the proposed model with the state-of-the-art deep learning-based approaches:

VulDeePecker: the model extracts code gadgets according to library/API functions and employs the Bi-LSTM model to learn the representation of them for classification.

SySeVR: a state-of-the-art token-based model that extracts code slices via PDGs and employs a Bi-GRU model to learn the representation of them for classification.

Devign: the first graph-based deep learning model that leverages GGNN to learn the embedding of the CPG of a function and uses a Convolution module to classify.

REVEAL: the state-of-the-art deep learning model that uses GGNN to learn the embedding of the CPG of a function and uses representation learning to classify.

Table 1 Classification Accuracies, Precisions, Recalls and F1 Scores of Baseline Models in Percentages

method	Acc	Prec	Recall	F1
VulDeePecker	89.05	17.68	13.87	15.7
SySeVR	84.22	24.46	40.11	30.25
Devign	88.41	34.61	26.67	29.87
REVEAL	84.37	30.91	60.91	41.25
Our model	84.90	32.43	60.25	**42.14**

As indicated in Table 1, our proposed model achieves the highest score in F1-score and precision. Compared to REVEAL, a robust latest method, the relative accuracy gain is 0.53%, the F1-score gain is 0.89%, and the relative precision gain is 1.52%. At the same time, the accuracy and F1 are relatively high compared to other baseline models. Overall, our proposed model demonstrates significant performance superiority over the baseline models.

B. Does SAGPool module learn more extra structural information?

To investigate this question, we choose the GCN module without SAGPool as the baseline. We also compared the model without the sample module and triplet module to explore their impacts.

Table 2 Impact of SAGPool, Sample and Triplet Loss

approach	Acc	Prec	Recall	F1
Complete model	84.90	32.43	60.25	42.14
w/o SAGPool	78.12	40.13	17.72	24.58
w/o sample	80.41	36.69	33.93	35.26
w/o triplet loss	82.37	36.84	42.04	39.27
w/o sample & triplet loss	83.30	38.71	28.83	33.05

Table 2 summarizes all the experiment results. The model without SAGPool module declines sharply, with 17.56% in F1 score and 6.78% in accuracy. We can safely conclude that SAGPool module helps learn more structural information because it pays more attention to code graphs' critical nodes. We observed that oversampling positive samples improves the performance a lot in the imbalanced dataset in general. The gains in F1-score is 6.88% and the gains in accuracy is 4.49%. The results also show that the triplet loss helps discriminate positive samples and negative samples. These comparisons clearly demonstrate the effectiveness of the SAGPool module, the sampling mechanism, and the triplet loss.

5. Conclusion and Threats to Validity

This study put forward a GCN model with SAGPool to detect code vulnerabilities and verify the superiority over other state-of-art models in the experiment. In a word, the model can learn more structural information about the vulnerability code, which will perform well in realistic scenes.

As mentioned in the experiment, alleviating the data imbalance may improve the model's performance, which is a problem in line with reality in the static vulnerability detection task. We only take a standard method to solve it, which may result in overfitting. So it needs more improvements for imbalanced data specific to the task itself. Finally, the size of the dataset is not big enough for the task, so the model cannot learn the vulnerability patterns well. Thus, datasets collected from realistic data sources are needed. However, such a dataset is hard to collect for the requirements for expert knowledge.

6. Acknowledgements

This research was supported by the 2019 Industrial Internet Innovation and Development Project No. TC190H46G/1.

7. References

[1] Lee J, Lee I, Kang J. Self-attention graph pooling[C]//International Conference on Machine Learning. PMLR, 2019: 3734-3743.

[2] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray. Deep learning based vulnerability detection: Are we there yet? CoRR, abs/2009.07235, 2020.

[3] R. L. Russell, L. Y. Kim, L. H. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. M. Ellingwood, and M. W. McConley. Automated vulnerability detection in source code using deep representation learning. In M. A. Wani, M. M. Kantardzic, M. S. Mouchaweh, J. Gama, and E. Lughofer, editors, 17th IEEE International Conference on Machine Learning and Applications, ICMLA 2018, Orlando, FL, USA, December 17-20, 2018, pages 757–762. IEEE, 2018.

[4] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. In 25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018. The Internet Society, 2018.

[5] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, Z. Chen, S. Wang, and J. Wang. Sysevr: A framework for using deep earning to detect software vulnerabilities. CoRR, abs/1807.06756, 2018.

[6] S. Hochreiter and J. Schmidhuber. Long short-term memory. Neural Comput.,9(8):1735–1780, 1997.

[7] K. Cho, B. van Merrienboer, C¸. G¨ulc¸ehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In A. Moschitti, B. Pang,

and W. Daelemans, editors, 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, pages 1724–1734. ACL, 2014.

[8] Y. Zhou, S. Liu, J. K. Siow, X. Du, and Y. Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alch´e-Buc, E. B. Fox, and R. Garnett, editors, Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada, pages 10197–10207, 2019.

[9] Y. Li, D. Tarlow, M. Brockschmidt, and R. S. Zemel. Gated graph sequence neural networks. In Y. Bengio and Y. LeCun, editors, 4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings, 2016.

[10] B. Rozi`ere, M. Lachaux, L. Chanussot, and G. Lample. Unsupervised translation of programming languages. In H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, editors, Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual, 2020.

[11] F. Schroff, D. Kalenichenko, and J. Philbin. Facenet: A unified embedding for face recognition and clustering. In IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015, pages 815–823. IEEE Computer Society, 2015.

[12] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. In 2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014, pages 590–604. IEEE Computer Society, 2014.

[13] Kipf, T. N. and Welling, M. Semi-supervised classification with graph convolutional networks. arXiv preprint,arXiv:1609.02907, 2016.