Vulnerability Detection in C/C++ Source Code With Graph Representation Learning

Yuelong Wu*†, Jintian Lu*, Yunyi Zhang*
, Shuyuan Jin*†
*School of Computer Science and Engineering, Sun Yat-sen University, Guangzhou, China
†GuangDong Key Laboratory of Information Security Technology, Sun Yat-sen University, Guangzhou, China
Email: {wuylong3, lujd6, zhangyy333}@mail2.sysu.edu.cn, jinshuyuan@mail.sysu.edu.cn,

Abstract—**An open challenge in software vulnerability detection is how to identify potential vulnerabilities of source code at a fine-grained level automatically. This paper proposes an approach to automate vulnerability detection in source code at the software function level based on graph representation learning without the efforts of security experts. The proposed approach firstly represents software functions as Simplified Code Property Graphs (SCPG), which can conserve syntactic and semantic information of source code while keeping itself small enough for computing. It then utilizes graph neural network and multi layer perceptrons to learn graph representations and extract features automatically, saving efforts of feature engineering. The comparison experiments demonstrate the effectiveness of the proposed approach.**

Index Terms—**Vulnerability Detection, Source Code, Graph Neural Network**

I. Introduction

Software vulnerabilities are always emerging with more and more software programs coming into practice in our daily life. Vulnerabilities once exploited by malicious attackers can possibly trigger global cyberattacks such as WannaCry, Heartbleed attack and so on. Due to software vulnerabilities, it was estimated that the WannaCry attack alone had caused 4 billion dollars economic losses across the globe [1]. How to detect vulnerabilities at an early stage of software development is crucial to mitigate or even prevent such kinds of worldwide attacks.

The existing vulnerability detection techniques can be mainly divided into two categories: static and dynamic techniques, according to whether a target program is executed during detection. The static detection techniques [2], [3] attempt to identify vulnerabilities in the source code of a program without actually running it. Generally these techniques require considerable time and effort of security experts. For instance, [3] tried to precisely detect vulnerabilities of PHP code by simulating nearly all built-in features in PHP language, which undoubtedly entailed tremendous analyses of PHP. Dynamic techniques such as fuzzing, dynamic taint analysis or symbolic execution [4], identity vulnerabilities through executing programs. These techniques have the abilities of detecting vulnerabilities with relatively high precision, however, much manual labor is needed to figure out how a program exactly work and how to handle path explosion problems [5].

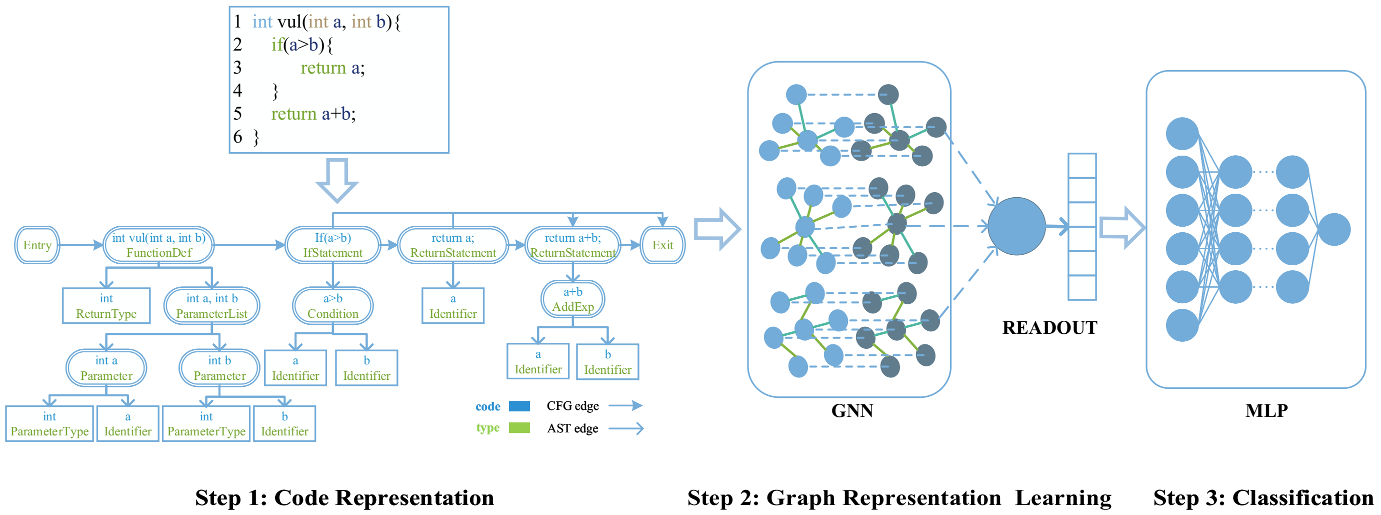
This paper focuses on utilizing deep learning to detect software vulnerabilities in source code. In comparison to existing techniques identifying vulnerabilities [6]–[8] at component or file level, the proposed approach identifies vulnerabilities at function—a fine-grained level of source code. It has advantages of detecting vulnerabilities at functions of source code, reducing the efforts of security experts to find and fix vulnerabilities.

To achieve that, there are two main problems to be solved: 1) how to represent source code so that meaningful information of code can be preserved as much as possible; 2) how to extract useful features from the source code representation to make detection effective. Note that graphs are suitable to represent complex relations among programs. There are several graph representations for software in literature, e.g. Abstract Syntax Tree (AST), Control Flow Graph (CFG) and Data Flow Graph (DFG). This paper proposes Simplified Code Property Graph (SCPG) to represent software programs effectively. SCPG is a kind of Code Property Graph (CPG) [9] which only consists of AST and CFG. It can conserve both syntactic and semantic information of source code while it is small enough for efficient computing. The paper utilizes two deep learning algorithms to automatically extract features from SCPGs. A Graph Neural Network (GNN) [10] is first used to learn graph-level representations for SCPGs, then a Multi Layer Perceptron (MLP) is employed to extract features automatically.

The contributions of this paper are as follows:

- The SCPG is proposed to represent the source code at the software function level, which enables fine-grained vulnerability detection. The SCPG representation approach has the advantages of conserving both syntactic and semantic information of source code while keeping itself small enough for computing.
- An effective and automatic feature extraction approach is proposed. It utilizes GNN and MLP to automatically learn graph representation and extract features. In comparison to Convolutional Neural Network (CNN) and Recurrent

Supported by the key research and development program of GuangDong Province(Grant No.2019B010136001) and the National Natural Science Foundation of China (Grant No.61672494). Corresponding author: Shuyuan Jin (jinshuyuan@mail.sysu.edu.cn).



```
int vul(int a, int b){
    if(a>b){
        return a;
    }
    return a+b;
}
```

Fig. 1. The overview of the proposed approach.

Neural Network (RNN) approaches, the proposed approach is suitable to learn representations for SCPGs because of the intrinsic ability of GNNs to learn from graphs.

The rest of the paper is organized as follows. Section II presents related work. Section III gives a detailed demonstration of the proposed function-level vulnerability detection approach. Section IV presents the evaluation of the proposed approach. Section V summarizes the paper and discusses limitations and future work.

II. Related Work

With an intention to relieve security experts from intense and tedious code auditing to find bugs, many researchers have applied deep learning methods to detecting vulnerabilities in source code.

Following fashion in natural language processing, some researches [11]–[13] just simply treated tokens in source code as words in natural language and used word embedding models (e.g. word2vec [14]) to transform source code into numeric vectors. Then the vectors can be further used as input to deep learning methods.

However, such a kind of method can not leverage much of specific innate information of source code since programming language is more structural than natural language [15]. Therefore, other works have utilized some typical representations of source code such as AST and CFG. Lin et al. [16] proposed a method using AST to represent source code. The method first parsed ASTs for each function and then serialized all tokens in ASTs. After tokens were embedded by word2vec model, the serialized ASTs were encoded into vectors by concatenating vectorized tokens. Finally, the method chose Bidirectional Long-Short Term Memory(BLSTM) model for feature extraction and applied transfer learning to cross-project vulnerability detection.

In [17], a novel intermediate code presentation “code gadget” was proposed. A code gadget consists of multiple lines of code that are semantically related and can be extracted via program slicing technique. The method named VulDeePecker vectorized and labeled the code gadgets as input to BLSTM to train a binary classifier, which achieved better false negative rate and false positive rate than existing detection systems. Further, Zou at et. [18] extended VulDeePecker to achieve multi-class vulnerability detection by introducing the concept of code attention and refining code gadget with control dependence.

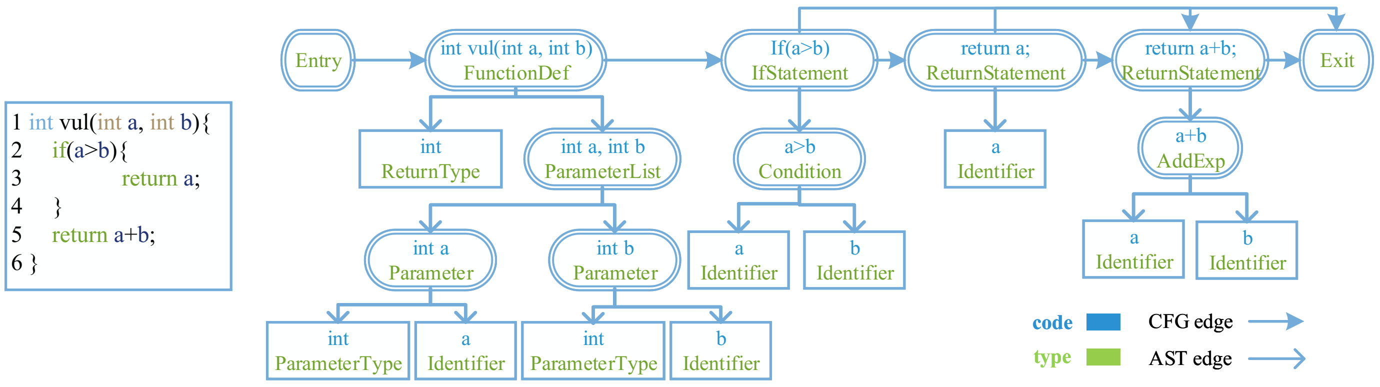
As indicated in [19], learning from different representations of code fragments together, a model can be more effective to detect code similarity, which also applies to other software engineering tasks (e.g. vulnerability detection). Inspired by CPG, Zhou et al. [20] extended CPG by adding natural sequence edges to it, and used gated graph recurrent network to embed graphs of functions in source code. Then the embedding graphs were input to a Conv model designed to select node features. Instead of extending CPG, our work simplifies CPG to make it smaller for graph learning. Based on CPG as well, Duan et al. [21] manually designed a feature encoder to transform the CPG of a function into a numeric matrix. The matrices were then used as input to an attention neural network followed by a classification model to finally achieve vulnerable function discovery. Compared with [21], this paper utilizes GNN and MLP to automatically extract features from SCPGs, saving time and labor for designing feature encoders.

III. Vulnerability Detection with Graph Representation Learning

In the section, we first present the overview of the proposed vulnerability detection approach. Then we describe how source code is represented as graph and how deep learning methods are deployed to vulnerability detection.

A. Overview

The proposed approach mainly consists of three steps as illustrated in Fig. 1. In the code representation, SCPGs



```
int vul(int a, int b){
    if(a>b){
            return a;
    }
    return a+b;
}
```

(a) sample code

(b) SCPG of sample code

Fig. 2. Code representation of sample code.

are extracted from the source code of target functions after which SCPGs are transformed into numeric representations as demonstrated in III-C. In the graph representation learning, numeric representations of SCPGs are first used as input of GNN to learn vector representations of nodes in SCPGs. Node vectors of each SCPG are then aggregated by a readout function to obtain a vector representation for the entire SCPG. In the classification, an MLP network is utilized to further extract abstract features from the graph-level representations and finally output classification results through a *sigmoid* function.

B. Code Representation

To utilize deep learning algorithms, source code has to be transformed into numeric representation. However, both syntactic and semantic information of source code will to some extent decrease after transformation, which hinders deep learning algorithms to learn more useful features. Hence, how to preserve as much information as possible during transformation is crucial. Although source code can be transformed into many distinct intermediate representations, such as tokens [12], trees (AST [16]), graphs (CFG [22], CPG [9]), representations that are effective in vulnerability detection are preferred. Inspired by [9], we simplify CPG to create SCPG for source code representation, since CPG has been used to effectively detect different kinds of vulnerabilities. SCPG only consists of AST and CFG relations in CPG, since data dependency can be deduced from AST and CFG [21], so it is smaller than CPG and more efficient for computing.

Given a target function, we use *Joern* [23], an open source platform for robust C/C++ code analysis, to extract the CPG and further process it to obtain the SCPG. Fig. 2 shows a simple example of how the source code of a function is represented as the SCPG. Each node in SCPG contains two parts, *code* and *type*. The *code* denotes the source code tokens the node represents and the *type* denotes the node type defined by the CPG schema in *Joern*. To transform the SCPG into numeric representation, an adjacency matrix **A** is used to store the connectivity between nodes. Each element in **A** is one of three values 0, 1, 2, which denote no connection, AST edge connection, CFG edge connection between two nodes respectively. As for node representation, we encode *type* using one-hot encoding and embed tokens in *code* using a word2vec model, which is trained with all code tokens from source code used in experiment. The final representation of *code* is the average of vectors of tokens in *code*. Eventually, the numeric representation of a node is a vector concatenated from the vectors of *type* and *code*; the SCPG is represented by an $n \times n$ adjacency matrix **A** and an $n \times d$ node feature matrix **X**, where n is the number of nodes in SCPG and d is the embedding dimension of code tokens.

C. Graph Representation Learning

Once we obtain the initial numeric representations of SCPGs, we use them as input of deep learning algorithms to learn graph representations and extract features automatically. As we represent source code as SCPGs which in essence are graphs, we need algorithms that fit such specific data. Graph Neural Network (GNN) is a preferred choice compared with CNN and RNN. CNN is more suitable for multi-array data like images and RNN is more suitable for sequential data like audio and text, while GNN is originally designed for data that can be represented as graphs [10].

GNN can learn node representations in a graph through message passing between neighbor nodes. Thus, the output of GNN can be used for graph-related tasks at various levels: node level, edge level, graph level. As for graph-level classification, the learned representations of nodes need to be further aggregated by a readout function, which can be a simple mean, min or max operation on all node representations of a graph.

Let $\mathcal{G}(\mathbb{V}, \mathbb{E})$ denote a graph, where $\mathbb{V}$ is the set of all nodes and $\mathbb{E}$ is the set of all edges in the graph; then the graph representation learning step using GNN can be formulated as follows:

$$\mathbf{h}_v^{(k)} = \bigodot_{t \in \mathbb{T}} AGG_N^{(k)}(\mathbf{x}_u^{(k-1)} : u \in \mathbb{N}_t(v)) \tag{1}$$

$$\mathbf{x}_v^{(k)} = UPDATE^{(k)}(\mathbf{x}_v^{(k-1)}, \mathbf{h}_v^{(k)}) \tag{2}$$

$$\mathbf{x}_{\mathcal{G}} = READOUT(\mathbf{x}_v^{(k)} : v \in \mathbb{V}) \tag{3}$$

where $\mathbf{h}_v^{(k)}$, $\mathbf{x}_v^{(k)}$ respectively denote the hidden and final representation of node v at the k_{th} layer or iteration of GNN and $\mathbf{x}_{\mathcal{G}}$ denotes final representation of graph $\mathcal{G}$; $\mathbb{N}_t(v)$ denotes neighbor nodes that can be reached by v through an edge of type t.

The learning step actually contains only two parts: node state updating (1) (2) and graph representation aggregating(3). In the node state updating, the node state $\mathbf{h}_v^{(k)}$ of node v at k_{th} layer in GNN is calculated as (1), where $\circledcirc$ is a function like *sum*, *mean*, *max* or *concat*. For each edge type t, a hidden node state $\mathbf{h}_v^t$ is aggregated from $u \in \mathbb{N}_t(v)$ using the node aggregation function $AGG_N^{(k)}(\cdot)$, which can be formulated as:

$$\mathbf{h}_v^t = \underset{u \in \mathbb{N}_t(v)}{\circledast} \phi(\mathbf{x}_v^{(k-1)}, \mathbf{x}_u^{(k-1)}) \tag{4}$$

where $\phi(\cdot)$ denotes a function like MLP and $\circledast$ denotes a node aggregation schema which is a differentiable, permutation invariant function like *sum*, *mean* or *max*; $\mathbf{h}_v^{(k)}$ is computed by aggregating all hidden node states $\{\mathbf{h}_v^t : t \in \mathbb{T}\}$ via $\circledcirc$. Then, $\mathbf{x}_v^{(k)}$ is gained through the node state updating function $UPDATE^{(k)}(\cdot)$, which can be functions such as MLP and RNN. In the graph representation aggregating, $\mathbf{x}_{\mathcal{G}}$ can be computed via $READOUT(\cdot)$, which can be functions like *sum*, *mean*, *max*. In our experiments, besides the simple functions, we also use the soft attention mechanism as (5) to assign different weights to nodes to obtain the representation of a graph, which helps focus more on relevant nodes.

$$\mathbf{x}_{\mathcal{G}} = \sum_{v \in \mathbb{V}} \mathbf{softmax}(MLP_1(\mathbf{x}_v)) \odot MLP_2(\mathbf{x}_v) \tag{5}$$

There exist numerous GNN variants since diverse implementations of $AGG_N(\cdot)$, $UPDATE(\cdot)$. We use three representative GNNs in our experiments: Graph Attention Network(GAT) [24], Graph Convolutional Network(GCN) [25] and Gated Graph Network(GGN) [26]. The major differences between these GNNs are how states of nodes are aggregated and updated. GAT updates node states using masked version of *self-attention* with *multi-head attention* while GGN uses an RNN cell to update node states. GCN in essence uses a weighted sum schema to update node states with weights computed from node degrees. For all of them, *sum* schema is used in $AGG_N^{(k)}(\cdot)$ in the experiments, which is a better schema than *mean* or *max* in most cases [27].

D. Classification

In the classification, the learned representation of a graph $\mathbf{x}_{\mathcal{G}}$ is directly input to a classification layer, so that we can detect vulnerabilities of source code at function level in an end-to-end fashion. An MLP is used as the classification layer to further extract abstract features and make classification through a *sigmoid* function. We evaluate how the choice for different number of layers of MLP and different activation functions affects classification performance.

TABLE I
DETAILED DESCRIPTION OF DATASET

Category	**Counts**	**Avg Nodes**	**Avg Edges**
Good SCPG	73694	26	43
Bad SCPG	34849	45	78
Total	108543	32	54

IV. EVALUATION

To evaluate the proposed approach, we conduct experiments to answer the following research questions(**RQ**s):

- **RQ1**: How do various factors of the proposed approach affect its performance?
- **RQ2**: Is the proposed approach competitive with existing approaches?

A. Dataset

We use Juliet Test Suite for C/C++ version 1.3 from SARD [28] as the raw data. We choose it as the source of dataset mainly for two reasons. On the one hand, it provides us with enough data that can be easily labeled, which is crucial for deep learning based approaches; the test suite contains tens of thousands of synthetic test cases for C/C++ language, which are specifically designed according to different CWEs. On the other hand, the ground truth of the test suit is solid enough since it is manually designed and has been used to evaluate many vulnerability detection methods [18], [21], [22], [29].

Since we aim to detect vulnerabilities at the function level, we first extract functions from source code files and label a function with 0 if the function is vulnerable, or 1 otherwise. The ground truth of a function can be acquired from the documentation of the test suit. As the test suite is originally designed for testing, it contains some information indicative of vulnerable code, e.g. the function names with "bad" and "good" are indicative of whether a function contains vulnerable code or not. To diminish such kind of bias, following the code normalization step in [17], we normalize user-defined function names and variables as **FUN#** and **VAR#**, where **#** is an integer corresponding to the order of a function/variable in the source code. We then use *Joern* to extract CPGs from normalized source code of functions from which we further obtain SCPGs and embed them as described in III-C. The detailed description of the ultimate dataset of SCPGs used in the experiments is shown in Table I.

B. Benchmarks

Two approaches [13], [16] are selected as benchmarks, which respectively utilize CNN and RNN to detect vulnerabilities in source code at function level; we simply refer to them as CNN and RNN approaches in our experiments. For both approaches, we first normalize the source code of a function as the proposed approach for a fair comparison. Then we follow the steps as described in the original papers as much as possible.

TABLE II
RESULTS TO EVALUATE EFFECTS OF GNN FACTORS

GNN Factors		ACC (%)	P (%)	R (%)	F1 (%)
GAT	*attention*	88.9	**96.3**	69.2	80.5
	mean	88.8	91.9	72.6	81.1
	max	88.5	91.9	71.7	80.5
	sum	89.1	82.2	**85.7**	**83.9**
GCN	*attention*	89.1	92.0	73.5	81.7
	mean	89.1	95.8	70.1	80.9
	max	88.9	95.8	69.6	80.6
	sum	**89.2**	83.5	84.2	83.8
GGN	*attention*	88.9	94.9	70.3	80.8
	mean	89.1	83.7	83.3	83.5
	max	88.7	94.7	69.9	80.4
	sum	88.8	92.5	72.1	81.0

C. Experiments

As for code tokens embedding, we use word2vec model from python package *gensim*. As for GNN, we use *pytorch geometric* [30] to implement several variants of GNN that fit SCPGs, which eventually are graphs with multi-type edges; the most implementations of GNNs in *pytorch geometric* do not support graphs with multi-type edges. The whole model is mostly implemented based on *pytorch* framework.

In the experiments, we split the dataset into train, validation and test set with a split ratio of 8:1:1, where we use the train set for model training, validation set for parameters tuning and test set for the final evaluation. To evaluate the overall performance of different models, we select four widely-used metrics: *accuracy* (**ACC**), *recall* (**R**), *precision* (**P**) and *F1-score* (**F1**). After parameters tuning, we run our experiments for **RQs** setting a value of 100 for epochs, batch size and embedding dimension of code tokens, 0.0001 for learning rate and 2 for number of GNN layers and 0.3 for dropout ratio in hidden layers. We use binary cross entropy as loss function and Adam with $L2$ penalty as the optimizer.

D. Results

For **RQ1**, we evaluate the effects of GNN Factors and MLP factors. GNN factors include GNN variants and the $READOUT(\cdot)$ function in (3). We select GAT, GCN and GGN as the GNN variants and *max*, *mean*, *sum* and *attention* (5) as $READOUT(\cdot)$. We use an MLP of 1 hidden layer with *gelu* activation for these experiments and the results are displayed in Table II.

Overall, GAT is the preferred choice in the candidates of GNN variants because it achieves best scores on three metrics with different $READOUT(\cdot)$ and *sum* is the preferred choice as $READOUT(\cdot)$ since it results in best scores on three metrics with different GNNs. The model combining GAT with *sum* achieves best *F1-score* and *recall*, which makes it the best combination. The model using *attention* with GAT achieves the best *precision*, which could be explained as that attention mechanism helps focus on more relevant features but reduces many ambiguous features; the model therefore tends to recognize samples that are representative but fails to identify samples that are less representative.

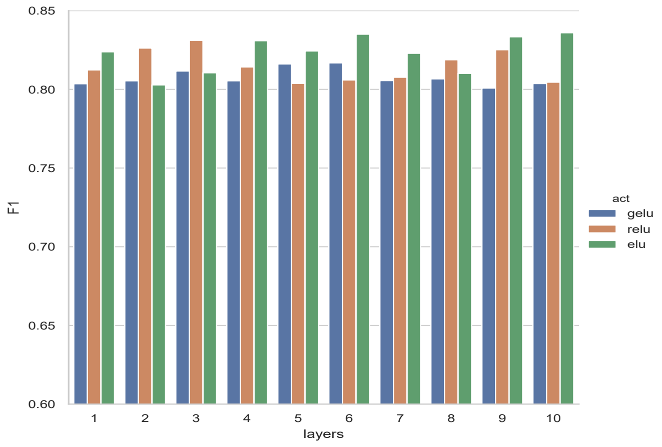


Fig. 3. F1-scores of different MLP factors.

TABLE III
COMPARISON BETWEEN THE PROPOSED APPROACH AND BENCHMARK

Approach	ACC (%)	P (%)	R (%)	F1 (%)
Ours	**89.2**	84.0	**83.1**	**83.6**
CNN	89.1	**99.8**	67.2	80.3
RNN	**89.2**	99.5	67.5	80.5

MLP factors concerned are the number of hidden layers of MLP and activation function used in the classification step. We vary the hidden layers in MLP from 1 to 10 with *relu*, *gelu*, *elu* as activation function respectively. For this experiment, we use GAT with *sum* as $READOUT(\cdot)$ with other settings unchanged and results are shown in Fig. 3. We measure results using *F1-score* since it can depict the overall performance of a model. In general, using *elu* can gain better *F1-scores*, gaining best scores in seven out of ten layers; the number of hidden layers of MLP does not have an obvious influence on the performance of models.

For **RQ2**, we use GAT with *sum* as $READOUT(\cdot)$, *elu* as activation function keeping other settings the same as for **RQ1**. The comparisons with benchmarks are indicated in Table III. Our approach achieves the best *recall* (83.1%) and *F1-score* (83.6%); its score on *accuracy* is also the best, though all approaches achieve nearly the same. Both the CNN and RNN approach achieve much higher scores (nearly 100%) on *precision* but lower on *recall* which may be an indication that directly utilizing source code tends to miss out some helpful information for vulnerability detection. The proposed approach also achieves an acceptable score (84.0%) on *precision*, so it is more effective overall.

V. CONCLUSION

This paper proposes a graph representation learning approach to automate vulnerability detection at the software function level. We first represent source code as SCPGs aiming to preserve source code information as much as possible while keeping itself small enough for computing. Then we utilize GNN to learn abstract representations of source code followed by an MLP layer to make classification automatically. The comparison experimental results indicate that the proposed approach achieves scores of 83.6%, 83.1% on *F1-score*, *recall* respectively, which are higher than existing CNN, RNN approaches; it also achieves the best *accuracy* of 89.2% and an acceptable *precision* of 84.0%.

Nonetheless, the present approach has some limitations. First, the paper uses synthetic datasets rather than real datasets in experiments. Second, the proposed approach is binary classification that can only detect whether the source code of a function is vulnerable or not. How to detect multiple and different types of vulnerabilities is still an open problem.

REFERENCES

[1] James Lewis, "Economic Impact of Cybercrime—No Slowing Down Report," McAfee, Tech. Rep., 2018.

[2] G. Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities," *Proceedings - International Conference on Software Engineering*, pp. 171–180, 2008. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1368088.1368112

[3] J. Dahse and T. Holz, "Simulation of Built-in PHP Features for Precise Static Code Analysis," in *Proceedings 2014 Network and Distributed System Security Symposium*, vol. 14, no. 2. Reston, VA: Internet Society, jun 2014, pp. 23–26. [Online]. Available: https://www.ndss-symposium.org/ndss2014/programme/simulation-built-php-features-precise-static-code-analysis/

[4] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," *Proceedings - IEEE Symposium on Security and Privacy*, pp. 317–331, 2010.

[5] T. N. Brooks, "Survey of automated vulnerability detection and exploit generation techniques in cyber reasoning systems," in *Intelligent Computing*, K. Arai, S. Kapoor, and R. Bhatia, Eds. Cham: Springer International Publishing, 2019.

[6] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2007, pp. 529–540. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1315245.1315311

[7] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE Transactions on Software Engineering*, vol. 37, no. 6, pp. 772–787, 2011.

[8] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Predicting vulnerable software components via text mining," *IEEE Transactions on Software Engineering*, vol. 40, no. 10, pp. 993–1006, 2014.

[9] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Proceedings - IEEE Symposium on Security and Privacy*, 2014, pp. 590–604.

[10] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, jan 2009.

[11] B. Chernis and R. Verma, "Machine learning methods for software vulnerability detection," in *IWSPA 2018 - Proceedings of the 4th ACM International Workshop on Security and Privacy Analytics, Co-located with CODASPY 2018*, vol. 2018-Janua, 2018, pp. 31–39. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3180445.3180453

[12] H. K. Dam, T. Tran, T. T. M. Pham, S. W. Ng, J. Grundy, and A. Ghose, "Automatic feature learning for predicting vulnerable software components," *IEEE Transactions on Software Engineering*, 2018.

[13] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, "Automated Vulnerability Detection in Source Code Using Deep Representation Learning," in *Proceedings - 17th IEEE International Conference on Machine Learning and Applications, ICMLA 2018*, 2018.

[14] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *1st International Conference on Learning Representations, ICLR 2013 - Workshop Track Proceedings*, 2013, pp. 1–12. [Online]. Available: http://arxiv.org/abs/1301.3781

[15] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2Vec: Learning Distributed Representations of Code," in *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, 2019, pp. 1–29. [Online]. Available: http://arxiv.org/abs/1803.09473

[16] G. Lin, J. Zhang, W. Luo, L. Pan, Y. Xiang, O. De Vel, and P. Montague, "Cross-Project Transfer Representation Learning for Vulnerable Function Discovery," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 7, pp. 3289–3297, 2018.

[17] 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 *Proceedings 2018 Network and Distributed System Security Symposium*, no. February. Internet Society, 2018, pp. 1–1.

[18] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, "μVulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection," *IEEE Transactions on Dependable and Secure Computing*, vol. PP, no. c, pp. 1–1, 2019.

[19] D. Poshyvanyk, M. White, M. Tufano, G. Bavota, C. Watson, M. Di Penta, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "Deep learning similarities from different representations of source code," in *Proceedings of the 15th International Conference on Mining Software Repositories - MSR '18*, 2018, pp. 542–553. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3196398.3196431

[20] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks," in *In Advances in Neural Information Processing Systems*, 2019, pp. 10 197–10 207. [Online]. Available: https://sites.google.com/view/devign http://arxiv.org/abs/1909.03496

[21] X. Duan, J. Wu, S. Ji, Z. Rui, T. Luo, M. Yang, and Y. Wu, "Vulsniper: Focus your attention to shoot fine-grained vulnerabilities," in *IJCAI International Joint Conference on Artificial Intelligence*, vol. 2019-Augus. International Joint Conferences on Artificial Intelligence, 2019, pp. 4665–4671.

[22] X. Cheng, H. Wang, J. Hua, M. Zhang, G. Xu, L. Yi, and Y. Sui, "Static Detection of Control-Flow-Related Vulnerabilities Using Graph Embedding," in *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*, vol. 2019-Novem. IEEE, nov 2019, pp. 41–50. [Online]. Available: https://ieeexplore.ieee.org/document/8882745/

[23] "ShiftLeftSecurity/joern: Open-source code analysis platform for C/C++ based on code property graphs." [Online]. Available: https://github.com/ShiftLeftSecurity/joern/

[24] P. Veličković, A. Casanova, P. Liò, G. Cucurull, A. Romero, and Y. Bengio, "Graph attention networks," in *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*, 2018.

[25] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings*, 2017.

[26] Y. Li, R. Zemel, M. Brockschmidt, D. Tarlow, M. Brockschmidt, R. Zemel, M. Brockschmidt, and D. Tarlow, "Gated graph sequence neural networks," in *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings*, no. 1, nov 2016, pp. 1–20. [Online]. Available: http://arxiv.org/abs/1511.05493

[27] K. Xu, S. Jegelka, W. Hu, and J. Leskovec, "How powerful are graph neural networks?" in *7th International Conference on Learning Representations, ICLR 2019*, 2019, pp. 1–17.

[28] "Software Assurance Reference Dataset." [Online]. Available: https://samate.nist.gov/SRD/testsuite.php

[29] Y. Zhang, D. Jin, Y. Xing, and Y. Gong, "Automated defect identification via path analysis-based features with transfer learning," *The Journal of Systems & Software*, vol. 166, p. 110585, 2020. [Online]. Available: https://doi.org/10.1016/j.jss.2020.110585

[30] M. Fey and J. E. Lenssen, "Fast Graph Representation Learning with PyTorch Geometric," in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, no. 1, 2019, pp. 1–19.