The ISC Int'l Journal of Information Security

July 2026, Volume 18, Number 2 (pp. 85–102)

http://www.isecure-journal.org

A Multi-Class Function/Line-Level Vulnerability Detection using Graph Neural Networks

Hamidreza M. Taheri [1]
Alireza Shafieinejad [1,*]
URL]https://www.modares.ac.ir/ shafieinejad
[1] *Department of Electrical and Computer Engineering, Tarbiat Modares University, Jalal AleAhmad Nasr, P.O.Box 14115-111, Tehran, Iran*

ARTICLE INFO.

Keywords:
Vulnerability Detection, Line-Level Vulnerability Detection, Multi-granularity Detection, Static Analysis, Deep Learning, Graph Neural Network

Abstract

One of the challenging issues for software developers is detecting vulnerabilities at different development stages. Security researchers are always seeking new methods to detect vulnerabilities more precisely in a short time. While there are many static and dynamic methods for detecting and discovering vulnerabilities, many of these approaches come with a high computational cost, which leads to inefficiencies, particularly in large-scale codebases. In recent years, deep learning has gained prominence in extracting vulnerability features from code without requiring direct intervention from cybersecurity experts. This paper proposes a multi-class vulnerability detection scheme at both the line-level (LLVD) and function-level (FLVD) using graph neural networks, based on node-level and graph-level prediction models, respectively. Moreover, by combining LLVD as a fine-grained approach with FLVD as a coarse-grained one, we propose a multi-granularity scheme called Function/Line-Level Vulnerability Detection (FLLVD) scheme. More specifically, it uses FLVD to detect the type of vulnerability while employing LLVD to identify its location in the source code. Our scheme's variants work with any abstraction graph extracted from incoming source code, such as Data Dependency Graph (DDG) and Program Dependency Graph (PDG). We evaluate our schemes using both man-made and real-world datasets: SARD and BigVul. Particularly, LLVD and FLLVD achieve performance gains of 0.90 and 0.94, respectively, in terms of F_1 metrics for a subset of SARD with 20 vulnerability types. In contrast, for the combination of SARD and BigVul with 6 vulnerability types, LLVD and FLLVD have F_1 scores of approximately 0.76 and 0.82, respectively.

1 Introduction

Software vulnerabilities are the major source of security flaws and cyber attack. Thus, finding potential software vulnerabilities is an important step in creating defensive layer against cyber attacks. It is dif-

* Corresponding author.
Email addresses: h.moallem@modares.ac.ir, shafieinejad@modares.ac.ir [
ISSN: 2008-2045

ficult and likely infeasible for developers to exactly determine the vulnerable parts of the source code in large-scale software systems. Thus, the main problem is to design an automated tool to detect efficiently and accurately vulnerabilities of a given source code. Traditional solutions for automated detection are mainly rule-based obtained by human experts. Recently, numerous research has been dedicated to using data-driven solutions for automated detection tools that take advantage of data mining and machine learning to predict the presence of software vulnerabilities. However, this approach is not straightforward and encounters many challenges in algorithm design and implementation such as selection of machine learning algorithm, proper representation of source code and fitting it to a constant size vector for feeding to the input layer of learning model.

1.1 Problem motivations

An important issue is detection level of vulnerability that refers to the extent of vulnerability localization. The first option is a coarse-grain algorithm that detects vulnerable functions in the given code. Other interesting options include fine-grain algorithms that perform detection at the code segment, statement, or line-level. The concept of line or statement level detection may be criticized by researchers who define vulnerability as a group of statements, neither a single statement nor a line, which their execution provides a situation that can be exploited by attackers. However, line-level detection is interesting as it aims to help developers identify vulnerabilities through appropriate warnings from the vulnerability verification system. There is a trade-off between accuracy and localization. That is the detection accuracy in a fine-grained algorithm is likely higher than a coarse-grained algorithm due to the likelihood of incorrect detection of line or statement.

Regarding this trade-off, we seek for a collaborative solution that takes the benefits of both coarse grain approach such as function-level and a fine grain approach such as line-level to find the optimum point for precision and localization. The former makes us to find the type of vulnerability with a high probability while the latter enable us to close more identify the code segment that contains the vulnerability.

1.2 Limitations of existing work

The most important challenges in the state-of-the-art of vulnerability detection using machine learning are as follows:

- **Detection level:** The schemes in [1–7] are function-level, i.e., only identifies vulnerable functions. While the schemes in [8–11] detect a code segment with vulnerability, the work in [12–16] perform detection at the line-level and the schemes in [17–19] aim to detect vulnerability at statement level. The detection accuracy in a fine-grained algorithm is likely reduced compared to a coarse-grained algorithm due to the likelihood of incorrect detection of line or statement.
  Further, some schemes can detect vulnerabilities at multiple level, e.g., both line and function-level. This feature referred as multi-granularity aims to detect the type of vulnerability at the function-level while simultaneously identifying the location of the vulnerability through line or statement-level detection. Although, most prior schemes are single-granularity, some of them such as [12–14, 18] are multi-granularity.
- **Detection the type of vulnerability:** Most of the prior researches focus on identifying a function as vulnerable or not-vulnerable regardless of vulnerability type. For instance, [1, 7–9, 15] are simple binary classification methods while the schemes in [2, 4, 10] are multi-class vulnerability detection algorithms. The performance of multi-class algorithms is usually lower compared to binary classification due to factors such as unbalanced vulnerability datasets and the similarity between different vulnerabilities.
- **Efficient processing in large scale source code :** Due to efficiency reason, it is necessary to focus on important parts of the code for embedding and passing to the neural network and thereby ignoring unnecessary details. For example VulDeePecker [8] uses *code attention* concept for choosing important parts of source code. It focuses only on parts of a given code that is related to code attentions. This refinement removes excessive parts of code and thereby restricting the input size of training model as well as decreasing noise during training.
  However, the extraction of code attention is not straightforward. Currently, VulDeePecker only detects vulnerabilities caused by C/C++ library/API function calls, as it generates code attention specifically around an API system call. Identifying the rest of code attentions has been left as an open problem. Similarly, the schemes in [3, 8–14, 17, 19] perform some type code slicing prior to word embedding or vectorization phase.
- **Inter-function detection:** Inter-function detection refers to identifying vulnerabilities that originate from flaws in multiple functions. Indeed, this vulnerability type cannot be detected by examining just a single function. Thus, a given inter-function algorithm should extract features from different functions and aggregate them to classify the sam-

ple as either vulnerable or non-vulnerable.
Although most of the prior work, such as [1, 4, 7, 8, 13, 15, 17, 19], focuses on intra-function vulnerability detection, the schemes presented in [2, 9–12, 14, 18] offer an inter-function approach for vulnerability detection.

- **Transforming to an intermediate language:** Intermediate language, e.g., Java bytecodes, is more closer to the hardware than a high level language like C++ and thus has more environment and runtime information which can help to detect software vulnerabilities more accurate or at the opposite increase noise in source data. Most of the work, such as [1, 7–9], does not utilize intermediate languages and instead analyzes the main source code directly. However, some approaches do employ intermediate languages as the basis for feature extraction during training. For example, ISVSF [17] utilizes a custom intermediate representation of the source code, as does [12].

Table 1 summarizes the state-of-the-art literature on vulnerability detection along with our scheme concerning the aforementioned features.

1.3 Contributions

In this paper, we focus on a multi-class and fine grained vulnerability detection algorithm. To solve this problem, we utilized a node level prediction model of graph neural networks in addition to graph perdition model. Further, we take a combination of node and graph prediction model into account to get higher precision. Below are our contributions.

- We propose a multi-class Line-Level Vulnerability Detection (LLVD) scheme using GNN node prediction level, which not only detects the type of vulnerability but also locates its line within the source code.
- Further, we propose a multi-class Function-Level Vulnerability Detection (FLVD) by means of GNN graph prediction level. By combination of LLVD and FLVD through an algorithm so-called *update prediction*, we propose FLLVD that outperforms LLVD in terms of accuracy for detection both type and location of vulnerability without any redesign or retrain of learning models.
- Our schemes operate on any desired abstraction level of the source code like CFG, DDG and PDG. However, our evaluations show that DDG obtains the best results in terms of precision, recall and F_1-score.

Our work is different from the schemes in [4, 8–10, 12, 15] which we use a graph neural network for learning model.
Unlike most schemes, such as Devign [1], which function as binary classifiers, our scheme addresses multi-class vulnerability detection.

Further, our algorithm tends to detect the vulnerability at line-level of source code while most of the schemes like [4] find it at function or code segment level. More specifically, few schemes including [2, 4, 10] which address multi-class vulnerability detection, are coarse grained algorithms where the vulnerability is identified at function or code segment level. Moreover, like [12–14, 18], our scheme has multi-granularity in which vulnerability is detected at both function and line level .

1.4 Paper organization

The rest of this paper is organized as follows: Section 2 provides a comprehensive literature review for vulnerability detection using deep learning approach. Section 3 introduces the system model and problem formulation. We present three algorithms which respectively detect vulnerability at function-level (FLVD), line-level (LLVD) and combination of function and line-level (FLLVD). The evaluation results are presented in Section 4. Finally, Section 5 concludes the paper.

2 Related work

In this section, we review the related work in four subsections: Code slicing, graph neural network (GNN), Image processing as well as Lexical analysis and large language models (LLMs). This categorization is mainly originated from source code embedding.

2.1 Code slicing approaches

Li et al. introduced VulDeePecker [8], a DL-based multi-class vulnerability detection approach using code gadgets to represent programs. This method extracts code slices from source codes based on code representation graphs like CFG. Finally code slices are converted to code gadgets and feed to BLSTM neural network.

Saccente et al. developed a prototype tool [20] using Long-Short Term Memory Recurrent Neural Networks (LSTM-RNN) to automate vulnerability detection in source code.

Fidalgo et al. proposed a deep learning method [21] for classifying PHP SQL injection (SQLi) vulnerabilities. It utilizes an intermediate language to represent code slices and applied NLP techniques.

Zou et al. introduced μVulDeePecker [10], a deep-learning-based system for multi-class vulnerability detection. It focuses on a specific part of source code, namely code gadget, and then transfers it to a binary

string to feed relevant information to a BLSTM networks. Each code gadget is mainly created around a line of source code containing vulnerable system call such as strcpy.

Li et al. introduced SySeVR [9], a framework for DL-based multi-class vulnerability detection in C/C++ programs. It focuses on extracting semantic and syntax features from the source code and combining them before feeding to neural network. They used 5-fold cross-validation to train eight standard models, including LR (Logistic Regression), MLP (Multi-Layer Perception), DBN (Deep Belief Network), CNN (Convolutional Neural Network), LSTM (Long Short-Term Memory), GRU (Gated Recurrent Unit), BLSTM, and BGRU (Bidirectional GRU), thereby identifying the model with the highest F_1 measure.

Li et al. introduced VulDeeLocator [12] a fine grained vulnerability detection system works at statement level. This work uses intermediate code for feature extraction. It can find the approximate location of a vulnerability across multiple files.

Zhang et al. proposed ISVSF [17], a framework for self-detecting statement level vulnerabilities in Java. Each sentence is defined as an intermediate representation of the sub-block obtained from control flow abstract syntax tree (CFAST). It considers the syntax characteristics of Java and adopts sentence-level method representation and pattern exploration.

2.2 GNN based approaches

Zhou et al. proposed Devign [1], a graph neural network model tailored for graph-level classification tasks for multi-class vulnerability detection. It leverages diverse code semantic representations and uses a Conv module to effectively extract valuable features from node representations. Devign detects vulnerability at function-level as a result of using a graph-level prediction model.

Haojie et al. introduced VULMG [2], a vulnerability detection approach framed as graph classification. It utilizes VecG as a code property graph to extract lexical, grammatical, and semantic features from source code as well as adjacency matrix capturing structural, control, and dependency information.

DeepWukong [22], proposed by Cheng et al., is another GNN-based approach vulnerability detection in C/C++ programs. It embeds code fragments into compact representations preserving programming logic.

Renjith and Aji focused on Android Java vulnerabilities from 2015 to June 2021 [23]. Both vulnerable and fixed code snippets are transformed into graphical representations using intermediate graph formats and a GNN-based vulnerability detection mechanism is developed to analyze them.

BGNN4VD [24] extracts syntax and semantic information from source code using abstract syntax trees (AST), CFG and data flow graphs (DFG). It employs a Convolutional Neural Networks (CNNs) to classify vulnerabilities.

Hin et al. proposed LineVD [18], a statement-level vulnerability detection, utilizing GNNs and transformers to identify vulnerable statements. The feature extraction component utilizes CodeBERT, a transformer-based model, to generate a total of $n+1$ feature embeddings: one for the overall function and n embeddings for each statement.
Yang et al. [3] proposed an algorithm to detect vulnerable sentences within a function using different code representation graphs including CDG, AST and DDG. The initial vectors, constructed using a pre-trained Word2Vec model, are fed into a vulnerability detection model based on a heterogeneous graph transformer.

Zou et al. introduced mVulPreter [13], a multi-granularity vulnerability detection model that combines function-level (coarse-grained) and slice-level (fine-grained) detection approaches. It tries to provide interpretable results.

SedSVD, proposed by Dong et al. [19], is a graph-based statement-level detection which leverages a Code Property Graph (CPG) to capture semantic and syntactic information. The important parts of the CPG, extracted by so-called central nodes, are used to construct subgraphs. They are embedded and trained using a Relational Graph Convolutional Network (RGCN) and a MLP.

Wu et al. introduced SlicedLocator [14], a dual-grained model detection that predicts both program-level and statement-level vulnerabilities. It incorporates a sliced dependence graph to preserve inter-procedural relations while filtering out vulnerability-irrelevant statements. It creates attention-based code embedding networks and employs a new LSTM-GNN model for fusion of semantic and structural modeling.

Zhang et al. introduced SDV [6], a static method for detecting function-level vulnerabilities in C/C++ programs. It utilizes a code property graph with Jump Graph Attention Network layers and convolutional pooling for feature extraction.

Tang et al. introduced CSGVD [5], another function-level vulnerability detection. It integrates a PE-BL module for semantic feature extraction which

consists of CodeBERT as embedding layer and a BiLSTM layer with GNNs to extract the structured information.

RGAN [7], proposed by Tang et al., addresses the issue of code imbalance in vulnerability detection, particularly in node-level graph prediction, where the proportion of non-vulnerable nodes is relatively high. RGAN utilizes a residual graph attention network with a mean bi-affine attention pooling mechanism to eliminate this issue.

Nguyen et al. introduced CodeJIT [11], a code-centric approach for just-in-time vulnerability detection that focuses on code changes, specifically commits in Git. It employs a graph-based representation, along with GNNs, to distinguish dangerous code changes from safe ones based on semantic encoding.

2.3 Image processing approaches

The schemes in [15, 25] employ DL-based image classification in vulnerability detection. The former, called VulCNN, addresses scalability and accuracy in detecting vulnerabilities within large-scale source code. It converts each function of source code into an image while preserving program intricacies. The latter, called VulGAI, extracts degree, closeness and second order metrics from the source code to build an image. Both schemes use a CNN model to detect vulnerabilities. performance.

Russell et al. proposed a vulnerability detection using deep representation learning that directly interprets lexed source code[26]. The scheme uses a custom C/C++ lexer to create a generic representation of function and explores both CNNs and RNNs for feature extraction from the embedded source representation.

2.4 Lexical analysis and LLM based approaches

Wartschinski et al. introduced VUDENC [27], a DL-based vulnerability detection tool that automatically learns features of vulnerable code from a large Python codebase. It applies a Word2Vec model to identify semantically similar code tokens and to provide a vector representation. Then, these vectors are fed into a LSTM network to classify vulnerable code token sequences.

Fu et al. introduced VulExplainer [4], a transformer based vulnerability classification for addressing diversity challenges. First, based on CWE abstract types, it redistributes labels into sub-distributions to reach a set of groups with more balanced label distribution. Then, it trains TextCNN teachers on simplified distributions and a Transformer student model for generalization across different CWE-IDs.

Steenhoek et al. [28] conducted an empirical study replicating state-of-the-art deep learning models on vulnerability detection datasets. They analyzed model capabilities, training data impacts, and interpretability challenges across different vulnerability types and dataset compositions.

Li et al. [16] proposed a DL-based approach for line-level vulnerability detection. It uses Clang to tokenize each line of function to get the tokens and their types. Further, position information for each token is added using positional encoding. Then, these vectors are fed into a hybrid neural network consisting of memory networks and multi-head attention mechanisms.

Zhang et al. worked on ChatGPT prompts [29] to develop a vulnerability detection engine based on Large language model (LLM). It integrates structural and sequential auxiliary information, optimizes prompt configurations for vulnerability detection tasks, and leverages ChatGPT's ability of memorizing multi-round dialogue.
Cho et al. [30] proposed LLM based vulnerability detection called RoS-Dex which consists of two main components: Code Understanding (CU) and Vulnerability Encoder (VE). The former is developed using code embedding techniques and a transformer-based architecture, captureing the semantic features of source code, while the former focuses on encoding vulnerability-related features, thereby improving classification performance.
Abdechakour et al. [31] introduced SecureQwen, an LLM based vulnerability detection in large-scale Python codebases. It Utilizes a decoder-only transformer architecture to captures complex relationships between code tokens, enabling classification of vulnerable code sequences across 14 CWEs.
Aleksei et. al [32] leveraged WizardCoder, a recent LLM StarCoder, and adapt it for vulnerability detection through further finetuning. They modified WizardCoder's training procedure to overcome imbalanced dataset and further explored different techniques to improve classification performance.

3 Proposed algorithms for vulnerability detection

3.1 Problem Formulation

We need to design a scheme that identifies the type of vulnerability among a set of possible vulnerability types, as well as its location within a given source code. Let N_{class} denote the number of different vulnerability types. Further, assume that the target program source code contains a set of functions or procedures.

Table 1. Comparison summary of existing DL-based vulnerability detection schemes (F: Function, C: Code segment, L: Line and S: Statement)

Feature	ours	[8]	[1]	[9]	[12]	[10]	[2]	[17]	[3]	[18]	[13]	[19]	[14]	[15]	[4]	[5]	[16]	[6]	[11]	[7]
Graph neural networks	✓	×	✓	×	×	×	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓
Detection level	L	C	F	C	L	C	F	C	F	S	L	S	L	L	F	F	L	F	C	F
Multi-class	✓	×	×	×	×	✓	✓	×	×	×	×	×	×	×	✓	×	×	×	×	×
Inter-function	×	×	×	✓	✓	✓	✓	×	×	✓	×	×	✓	×	×	×	×	×	✓	×
Code slicing	×	✓	×	✓	✓	✓	×	✓	✓	×	✓	✓	✓	×	×	×	×	×	✓	×
Intermediate language	×	×	×	×	✓	×	×	✓	×	×	×	×	×	×	×	×	✓	×	×	×
Multi-granularity	✓	×	×	×	✓	×	×	×	×	✓	✓	×	✓	×	×	×	×	×	×	×

The objective is to label each line of any function with a number from set 0, 1, 2, . . . , N_{class} , where type-0 identifies non-vulnerable code while type-i ($1 \leq i \leq N_{class}$) corresponds to a Common Weakness Enumeration IDentifier (CWE-ID).

Note that both type and location of vulnerability must be predicted accurately. That is we will encounter a false alarm by detecting either a wrong type or an incorrect location. The problem is quite general, but under specific assumptions, it becomes a well-known problem. For example, in the case of $N_{class} = 2$, it reduces to a binary classification problem that distinguishes between vulnerable lines and non-vulnerable ones. Moreover, if we assign the same label to each line of a function in the given source code, it transforms into a file-level vulnerability detection problem.

3.2 System Model

The overview of the proposed framework is shown in Figure 1. The primary objective of design is to ensure a high degree of flexibility. For instance, the framework supports both fine-grained and coarse-grained vulnerability detection by operating at the line and function levels, respectively. Further, our scheme is independent of programming language, supporting various source codes, including Java and C++. It converts these source codes to graph abstraction levels, such as Control Flow Graphs (CFG) or Data Dependency Graphs (DDG), preparing them for input into the neural network model.

According to Figure 1, our scheme composed of the following steps occurred in sequence:

- Preprocessing
- Embedding
- Training
- Test and Update prediction

The first step deals with primary input as a latest patch of software package and its previous vulnerable version to create a clean dataset as labeled source code. It contains a set of source files categorized into vulnerable and non-vulnreable files. It is supposed that each vulnerable file at least contains a line of code that is labeled as a specific vulnerability. Basically, these files are gathered from the latest patch of a software package which eliminate a set of specific vulnerabilities. Thus, for each vulnerable file, there is a non-vulnerable version which leads to a balanced dataset.

For efficiency, in each vulnerable file, we select only a set of functions that contain at least one vulnerable line, removing the rest of the functions. In summary, we will have a set of function pairs (f_s, f_v), where f_s is the secure version and f_v is the vulnerable version. Now, we extract a desired abstraction graph model for each function pair like CFG or DDG depending on our configuration with a suitable software tool. The output of this stage is graph pairs (G_s, G_v), where the former is the secure version and the latter is the vulnerable version. The label of each node in G_s is set to 0 while some nodes of G_v is marked with 1 denoting that these nodes are corresponding to a vulnerable line.

The second stage performs graph embedding in which each G_s and G_v generated in previous step are transformed to a proper format which is suitable for feeding to neural network model. Basically, this step encounters word embedding where each node or edge containing string labels, i.e., part of source code, will be converted to a vector using a word embedding technique. The embedded has a specific data structure representing nodes, edges, adjacency matrix and nodes vulnerability labels.

The third step involves training where in each embedded graph of training set is fed into a customized neural network. More specifically, *GatedGraphConv* [33] model is used for training CFG graphs while *EC-*

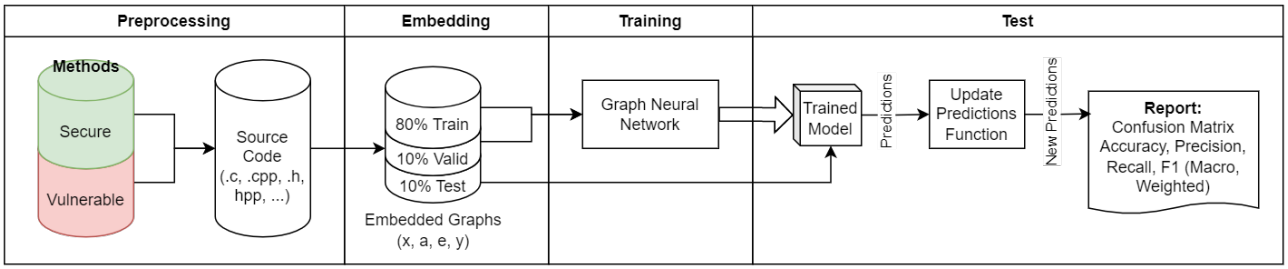


Figure 1. System Model Big Picture.

CConv [34] is used for the graphs with edge-label like DDG. The result of this step is a trained model.

The final step performs testing to evaluate the trained model's performance. First, each graph of test set is fed into model to predict node or graph label. Then update prediction is done in which predicted node labels are further processed. This function alters the predicted output in order to improve the evaluation metrics. The basic idea of this phase is that unified the prediction labels to a set of nodes relating to a single line of source code. We will show that this step increase the precision of the model even for real-world applications.

3.3 Line-Level Vulnerability Detection

The first scheme (LLVD) is a fine-grain method which predicts vulnerabilities at node level using GNN.

3.3.1 Preprocessing

Since this step is highly tied with the incoming dataset, we will describe it in more detail in Section 4.

3.3.2 Embedding

As mentioned earlier, we deal with a set of pair (f_s, f_v) denoting the safe and vulnerable version of a given function, respectively. Without loss of generality, we use the following abstraction graph models for converting a desired f_s or f_v into a graph model:

- CFG (Control Flow Graph)
- DDG (Data Dependency Graph)
- PDG (Program Dependency Graph)

The first item focuses on the flow of control through a program while the second concentrates on the flow of data and dependencies between operations. Further, PDG maintain some control dependency information in addition to data dependencies. The output of CFG is denoted by a pair (**x**, **a**) while the output of DDG and PDG is represented by a triple (**x**, **a**, **e**) where :

- **x** denotes the set of nodes,
- **a** denotes the adjacency matrix which illustrates the transition between nodes and
- **e** denotes the set of edges which describes data dependency between nodes.

Each statement in source code converts to a single or multiple nodes, based on its complexity. A simple assignment statement is mapped to a single node while complex instruction like if-else or function call is transformed to multiple nodes. Each node has a *code* field which corresponds to a whole or part of a statement. Further, as each edge label in DDG corresponds to data dependency, it contains a variable or an expression.

To feed x and e into the learning model, the source code related to each node or edge, represented as an ASCII string, must be converted into an array of floats. This process, known as word/code embedding, is mainly implemented by means of either Word2Vec or BERT.

We examined both methods, but we disregarded Word2Vec due to its poor results. The second solution, which is based on the pre-trained model CodeBERT for the C++ language [35], yields better results. The output length of this model is 768. Let n and m denotes the number of nodes and edges, respectively. Thus, **x** and **e** are embedding to array of $n\times768$ and $m\times768$ float numbers. Moreover, **a** is embedding into array of n^2 elements. In CFG each element is 0 or 1 while in DDG it can 0 or a positive integer larger than 1 denoting the number of edges between two distinct nodes. In both cases,**a** will be a sparse matrix which is efficiently handled by suitable structure to optimize memory usage.

3.3.3 Training

We design two distinct neural networks for LLVD shown in Figure 2 and Figure 3, respectively. The former is used for graphs without edge like CFG while the latter is used for graphs with edge label like DDG. In the first model, the input **x** after normalization along with **a** are feeding to a 4-layer gated graph convolution networks. The input part a is not updated and is feeding to each hidden layer without any change while the input part x is gradually update in each hidden layer. After the hidden layers, the out-

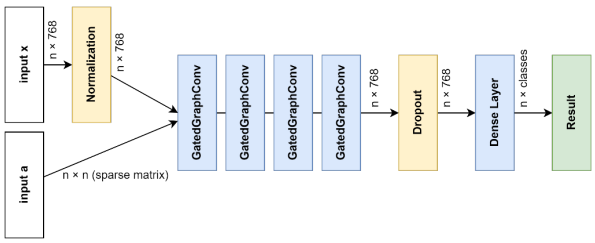


Figure 2. The node level prediciton model for graphs without edge labels like CFG.

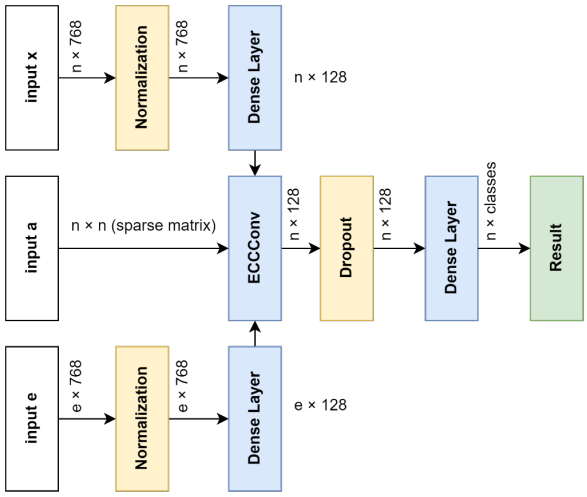


Figure 3. The node level prediciton model for graphs with edge labels like DDG.

put is forwarded into a Dropout layer with rate 0.5 to avoid over-fitting. After that, the output is feeding into a Dense layer which reduces the output to $n \times N_{class}$ where N_{class} denotes the number of vulnerability classes.
In the second model, the input **x** as well as input **e** is passed through a normalization layer and a Dense layer with scale factor 768 to 128. Then, both of them along with **a** are feeding to an ECCConv layer. After that, the output is forwarded into a Dropout layer with rate 0.5 to avoid over-fitting. Finally, the output is feeding into a Dense layer which reduces the output to $n \times N_{class}$.

3.3.4 Update Predictions

As mentioned above, during CFG or DDG extraction, a single statement can be mapped into multiple nodes. Since LLVD detects vulnerabilities at the node level, it can be very fine-grained; that is, it may mark a part of a statement as vulnerable while leaving the rest as non-vulnerable. To address this issue, we consider a new stage as *Update-prediction* which aggregates predicted label for multiple nodes corresponding to a single statement of source code. It ensures that the

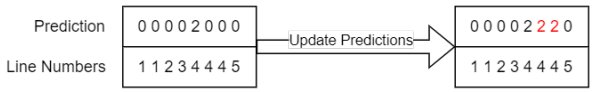


Figure 4. A simple example for update-prediction.

nodes extracted from a single statement share the same prediction.

For better understanding, consider the example in Figure 4, where a single vulnerability was detected in a node extracted from line 4 of the source code. Since two more nodes (6 and 7) are associated with line 4 of source code, *UpdatePrediction* will replace their labels with vulnerability class 2 which is currently assigned to node 5.

Algorithm 1 outlines the procedure for updating node-level predictions. It begins by initializing an index variable, α, at the first node. Then, a while loop iterates through the nodes, aggregating predictions for contiguous nodes associated with a specific line of code. Once these nodes are identified, the algorithm checks for the maximum predicted value among them. If a non-zero value is found, it will be assigned to all nodes in this segment. Then, α is updated to the beginning of next segment, and the loop iterates until all nodes have been processed.

Algorithm 1 UpdatePrediction
Input : $P_1, .., P_n, L_1, .., L_n$
Output : $P_1, .., P_n$

$\alpha \leftarrow 1$
while $(\alpha < n)$ **do**
  $\beta \leftarrow \alpha$
  while $(\beta < n$ and $L_{\beta+1} = L_\alpha)$ **do**
    $\beta{+}{+}$
  end while
  $\gamma \leftarrow \max_{i=\alpha}^{\beta} P_i$
  if $(\gamma > 0)$ **then**
    $(P_\alpha, .., P_\beta) \leftarrow (\gamma, .., \gamma)$
  end if
  $\alpha \leftarrow \beta + 1$
end while
return $(P_1, .., P_n)$

Algorithm 1 describes the procedure for updating node-level predictions in the FLLVD model. Similar to the LLVD algorithm, this method also begins by initializing the index α at the first node. The algorithm iterates over the nodes to identify contiguous segments corresponding to the same line. However, instead of summing the prediction scores, this algorithm assigns each node the maximum prediction score within the segment. This approach ensures that all nodes in a segment reflect the highest level of predicted vulnerability, thereby aligning with the function-level predictions of the FLLVD model.

3.4 Function-Level Vulnerability Detection

Function-Level Vulnerability Detection (FLVD) is our second scheme based on graph-level prediction. It is a coarse-grain method and useful for enhancing other methods to create a multi-granularity vulnerability detection. It predicts the vulnerability class for the entire source code of a function without any localization. The training model for FLVD is the same as that for LLVD, except that the final Dense layer reduces the output to N_{class} bits in a one-hot encoding format. Note that FLVD can utilize both CFG and DDG abstractions during the embedding and training phases.

3.5 Function/Line-Level Vulnerability Detection

The third scheme is Function/Line-Level Vulnerability Detection (FLLVD. It is a multi-granularity method obtained through the combination of FLVD and LLVD models. This scheme prioritizes FLVD in order that if FLVD does not detect any vulnerabilities in a function, all nodes will be labeled as non-vulnerable, regardless of the LLVD predictions.

Otherwise, if FLVD detects a specific vulnerability y_g, we will take the LLVD prediction into account. More specifically, the nodes that LLVD indicates as vulnerable are marked with y_g, regardless of the type of vulnerability predicted by LLVD. Briefly, LLVD is used for the localization of vulnerabilities while the type of vulnerability is identified by FLVD. The details of FLLVD is depicted in Figure 5 as well as explained in Algorithm 2.

Algorithm 2 FLLVD-Prediction
Input : $G, L_1, .., L_n$
Output : $P_1, .., P_n$

$y_G \leftarrow$ FLVD-Prediction(G)
$(P_1, .., P_n) \leftarrow (0, .., 0)$
if $(y_G = 0)$ **then**
  return $(P_1, .., P_n)$
end if
$(Q_1, .., Q_n) \leftarrow$ LLVD-Predict(G)
$(q_1, .., q_n) \leftarrow$ UpdatePrediction$(Q_1, .., Q_n, L_1, .., L_n)$
for $i \leftarrow 1$ to n **do**
  if $(q_i > 0)$ **then**
    $P_i \leftarrow y_G$
  end if
end for
return $(P_1, .., P_n)$

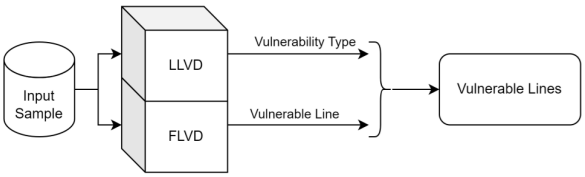


Figure 5. FLLVD as the combination of node-level and graph-level prediction model.

4 Evaluations

4.1 Data preparation

4.1.1 Data gathering

We examined different variants of our scheme with both man-made and real world datasets : SARD and BigVul. SARD (Software Assurance Reference Dataset) is a collection of test programs with documented vulnerability which vary from small synthetic programs to large applications in C, C++, Java, PHP, and Csharp, and cover over 150 classes of weaknesses. We selected a relatively balanced set of SARD datases which summarized in Table 2. We will explain more details of process in Section 4.1.2. For BigVul, the available samples are very limited in both vulnerability types and the number of samples. Therefore, we combined BigVul samples with SARD to achieve a sufficient number of samples while also ensuring a balanced set of different vulnerabilities

4.1.2 Graph generation

The preperation of dataset along with graph generation is show in Figure 6.In SARD, both vulnerable and non-vulnerable code are included in a single file, with comments detailing each vulnerable statement. Therefore, we first separate the vulnerable code from the non-vulnerable code by creating two distinct file, effectively doubling the number of samples in the dataset. Each sample is then processed with the Joern tool [36] to extract the desired graph. The output of Joern can be further processed and converted into various abstraction graphs. By default, the CFG graph is extracted for each function and is used to ensure that the given code is a patched version of the existing code. It is done by applying the DotDiff algorithm to extracted graphs of both safe and vulnerable code (f_s and f_v) to verify that f_v is a patched version of f_s by comparing respective nodes and edges. It detects changes by comparing DOT files of each function in the vulnerable and non-vulnerable versions while ignoring non-affecting characters such as spaces and tabs during comparison.
Although CFG is generated by default, other abstraction formats, such as DDG and PDG can be created upon request. All extracted graphs are saved in DOT

format. Each non-trivial line of code can be mapped to one or more nodes in the code graph. Since comments are omitted in the extracted graph, we label the nodes corresponding to a vulnerable line as vulnerable nodes and leave the others as non-vulnerable. Against SARD, in BigVul, patched functions are located in separate files. By comparing a desired function in the patched file with the one in previous file version, we can identify the vulnerable function.

After identifying patched functions, we can mark a specific node in the extracted graph as either vulnerable or non-vulnerable, as well as labeling the whole graph accordingly. Now, labeled graphs are ready for the next stage, i.e., embedding. The output of SARD preparation and graph generation is show in Table 2. The second column denotes the number of samples for each vulnerability class. The number of function extracted from each class samples is shown in third column. The fourth column identifies the number of duplicated functions in each class. Subsequent columns G_v and G_s denote the number of vulnerable and non-vulnerable graphs, respectively.
Figure 7 shows extracted dot files for a simple C++ program. The source code, shown in Figure 7a, defines a main function consists of 9 lines. For simplicity and efficient use of space we removed some unnecessary details from DOT files. The DOT file for CFG, DDG and PDG are showin in Figure 7b, Figure 7c, Figure 7d, receptively. Moreover, the corresponding graphs are depicted in Figure 7e, Figure 7f, Figure 7g, receptively. DOT file consists of two main parts : nodes and edges definitions. Each node corresponds to a statement in source file. The related line of source code is identified by the phrase L_i at the end of node definition. We see that trivial line such as "", "else" and "" does not mapped to any node. Against, complex source codes, such as if-condition in line 5, are mapped to more than one nodes. Further, most of non-trivial code lines (such as 4, 6 and 8) are mapped to a single node. Nodes in DDG and PDG is almost the same as CFG with a slight difference that some declarations such as argument passing are considered as nodes.
The edges in a CFG strictly represent control dependencies between nodes, while in a DDG, they reflect data dependencies among nodes. In contrast, edges in a PDG may represent both control and data dependencies among nodes.The control and data dependencies for edges in the PDG DOT file are discriminated by the keywords CDG and DDG, respectively. Further, in graphs represented in Figure 7, control and data dependency are distinguished by solid and dashed line, respectively. Each data dependency edge is labeled by a variable or expressions which caused data dependency among nodes. There can be multiple edges between the same pair of nodes due to various variables that create data dependencies.

4.1.3 Data leakage prevention

After creating dot file, we do an additional checking for finding duplicated functions:

- The hash for each dot file is computed using SHA-1.
- We seek for the similar items in calculated hash values.
- For each set of dot files with with the same hash value, all instances except one of them are removed form datasets.

This process ensures that no duplicated functions are fed into learning model and accordingly avoid data leakage between train and test sets. Note that we can do this checking in previous stage directly on source file. However, we ignore it since working on functions may create false result. First, renaming any variable, function name or argument, string literal creates a different function. Second, difference in style of programming like as additional space, tab or end of line generates a different function body text. As stated before, the number of duplicated functions for each class of vulnerability in SARD is shown in Table 2. In average, each class has 75 duplicated functions out of them, 19 entries are vulnerable while 56 items are corresponding to non-vulnerable functions. It is worth noting that in the BigVul dataset, no duplicated functions are found. Just, we checked the potion of SARD records adding to BigVul. Regarding these policies, ensures prevention of data leakage in which a given sample does not appear in both train and test sets.

4.2 Training setup

To demonstrate the efficiency and accuracy of our scheme, we implemented it as python program. For graph and node prediction, we utilized library tensorflow??. The program runs on a laptop system with Intel Core i7 12700 CPU and 16 GB of RAM. Further, the system utilizes an Nvidia GeForce RTX 3060 GPU to accelerate training process. The learning parameters are summarized in Table 3.
The number of convolution layers is empirically set for both models. For the first model, Figure 2, it is set to 4 as the best choice. For 1, 2 or 3 layers the accuracy of the model is significantly lower than the model with 4 layer. However it is faster in training phase. Against, for the models with 5 or higher number of convolution layers no significant gain is obtained although they are very slow in training phase. For the second model, Figure 3, which convolution layer is set to ECCConv with a single layer. We

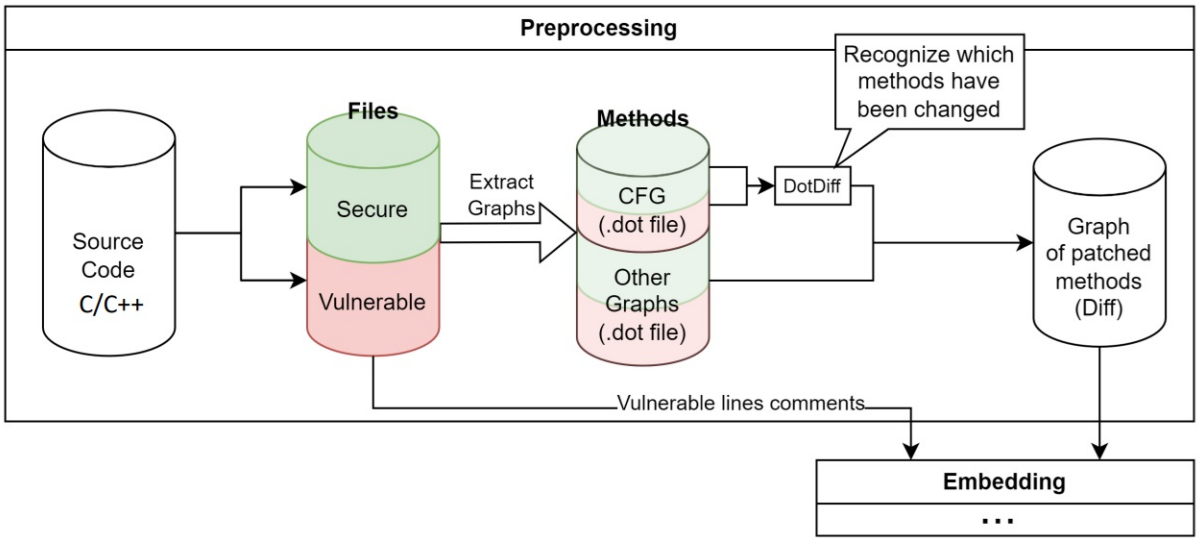


Figure 6. Preparation of dataset.

Table 2. SARD dataset used in our evaluation.

CWE	Samples	Functions	#Dup	$\|G_v\|$	$\|G_s\|$
23	4100	11850	104	3680	8066
36	4100	11850	101	3686	8063
78	8200	23700	44	7395	16261
121	8177	24503	41	7287	17175
122	10116	30452	62	9068	21322
124	3456	10500	102	3057	7341
126	2388	7518	63	2113	5342
127	3456	10500	104	3047	7349
134	5160	24630	13	5092	19525
190	6660	27140	19	5885	21236
191	4854	20211	36	4350	15825
194	1968	5688	92	1752	3844
195	1968	5688	106	1750	3832
369	1548	5940	45	1315	4580
401	2812	10917	36	1643	9238
415	1724	6581	147	1465	4969
590	4355	12663	159	3011	9493
690	1640	4740	59	1454	3227
762	6388	24552	110	5471	18971
789	1720	6600	57	1465	5078
Average	3784	14311	75	3700	10537

checked more than one layer of ECCConv. But surprisingly, we found that no significant gain is obtained. Further, we examined other type of convolution such as CrystalConv. But, all of them has accuracy less than ECCConv. Thus, we fix it to single layer of ECCConv. The running time of the algorithm for training phase is about 75 minutes for SARD dataset. Moreover, for SARD+BigVul, it takes about 20 minutes. During the testing phase, processing a set of 10 batches took approximately 1 second. Since each batch consists of 32 embedded graphs, the average inference time for a single function is approximately 3.1 milliseconds.

We use the following metrics for evaluations:

$$Recall = \frac{\text{TP}}{\text{TP+FN}} \tag{1}$$

$$Precsion = \frac{\text{TP}}{\text{TP+FP}} \tag{2}$$

$$F_1 = \frac{2Precision \times Recall}{Recall + Precsion} = \frac{2\text{TP}}{2\text{TP+FP+FN}} \tag{3}$$

Where TP, FP and FN denote the number of true positive, false positive and false negative, respectively. For multi-class evaluation, we use macro average F_1 which is computed by taking the arithmetic mean of all the per-class F_1 scores as :

$$\overline{F_1} = \frac{1}{N_{class}} \sum_{i=1}^{N_{class}} F_1(c) \tag{4}$$

Where $F_1(c)$ denotes F_1 score for class c.This ensures that classes with more samples contribute more significantly to the overall weighted F_1 score, reflecting a realistic performance measure across imbalanced classes.

4.3 Results for SARD

The result of binary classification for SARD is shown in Figure 8. Note that all F_1 scores were evaluated over multiple runs using the average results from 5-fold cross-validation. For each vulnerability type, we trained a specific model with a dataset containing

```
1. int main(int argc,
            char *argv[])
2. {
3.   char input[128];
4.   scanf("%s", input);
5.   if (strlen(input) > 10)
6.     printf("%s", input);
7.   else
8.     puts("Nothing");
9. }
```

(a) C++ Source code

```
#nodes
"1" (METHOD,main)                L1
"2" (scanf,scanf("%s";, input)) L4
"3" (strlen,strlen(input))       L5
"4" (strlen(input) > 10)         L5
"5" (printf,printf("%s", input)) L6
"6" (puts,puts("Nothing"))       L8
"7" (METHOD_RETURN,int) L1

#edges
"1" -> "2"
"2" -> "3"
"3" -> "4"
"4" -> "5"
"4" -> "6"
"5" -> "7"
"6" -> "7"
```

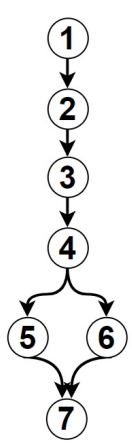
(b) CFG DOT file

```
#nodes
"1" (METHOD,main)                L1
"2" (METHOD_RETURN,int) L1
"3" (PARAM,int argc)             L1
"4" (PARAM,char *argv)           L1
"5" (scanf,scanf("%s", input))   L4
"6" (strlen(input) > 10)         L5
"7" (printf,printf("%s", input)) L6
"8" (strlen,strlen(input))       L5
"9" (puts,puts("Nothing"))       L8

#edges
"3" -> "2"  "argc"
"4" -> "2"  "argv"
"6" -> "2"  "strlen(input)"
"6" -> "2"  "strlen(input) > 10"
"7" -> "2"  "input"
"7" -> "2"  "printf("%s", input)"
"1" -> "3"
"1" -> "4"
"1" -> "5"
"8" -> "6"  "input"
"1" -> "6"
"5" -> "8"  "input"
"1" -> "8"
"1" -> "7"
"8" -> "7"  "input"
"1" -> "9"
```

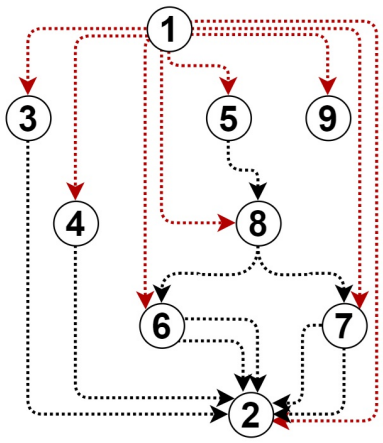
(c) DDG DOT file

```
#nodes
"1" (METHOD,main)                L1
"2" (METHOD_RETURN,int) L1
"3" (PARAM,int argc)             L1
"4" (PARAM,char *argv)           L1
"5" (scanf,scanf("%s", input))   L4
"6" (strlen(input) > 10)         L5
"7" (printf,printf("%s", input)) L6
"8" (strlen,strlen(input))       L5
"9" (puts,puts("Nothing"))       L8

#edges
"3" -> "2"  "DDG: argc"
"4" -> "2"  "DDG: argv"
"6" -> "2"  "DDG: strlen(input)"
"6" -> "2"  "DDG: strlen(input) > 10"
"7" -> "2"  "DDG: input"
"7" -> "2"  "DDG: printf("%s", input)"
"1" -> "3"  "DDG: "
"1" -> "4"  "DDG: "
"1" -> "5"  "DDG: "
"8" -> "6"  "DDG: input"
"1" -> "6"  "DDG: "
"5" -> "8"  "DDG: input"
"1" -> "8"  "DDG: "
"1" -> "7"  "DDG: "
"8" -> "7"  "DDG: input"
"1" -> "9"  "DDG: "
"6" -> "7"  "CDG: "
"6" -> "9"  "CDG: "
```

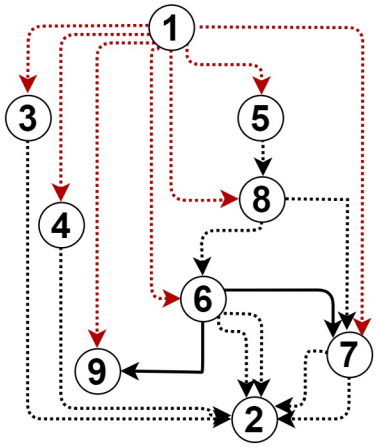
(d) PDG DOT file



(e) CFG



(f) DDG



(g) PDG

Figure 7. A simple C++ source file, corresponding DOT files and schematics for CFG, DDG and PDG configurations.

that vulnerability along with its non-vulnerable samples. We examined both CFG and DDG with LLVD and FLLVD. The results indicates that:

- In most cases, FLLVD has better performance in terms of F_1 for both code representation graph. More specifically, the improvement is about 1.6% for DDG and 6.2% for CFG. Since FLLVD only updates the type of prediction made by LLVD, it demonstrates that graph-level prediction is more accurate than node-level prediction in detecting vulnerability types.
- The performance of DDG is better than CFG regardless the detection algorithm. More specifically, the gain of DDG over CFG is about 15% for LLVD as well as 10.3% for FLLVD. It demonstrates that data dependency provides more information about vulnerabilities than control dependency.

However, some exceptions are observed. Ftimeor example, $\text{FLLVD}_{\text{DDG}}$ outperforms other methods except for CWE-127. Further, the performance for DDG is approximately higher than CFG in terms of F_1 metric. $\text{FLLVD}_{\text{CFG}}$ exhibits the same performance as

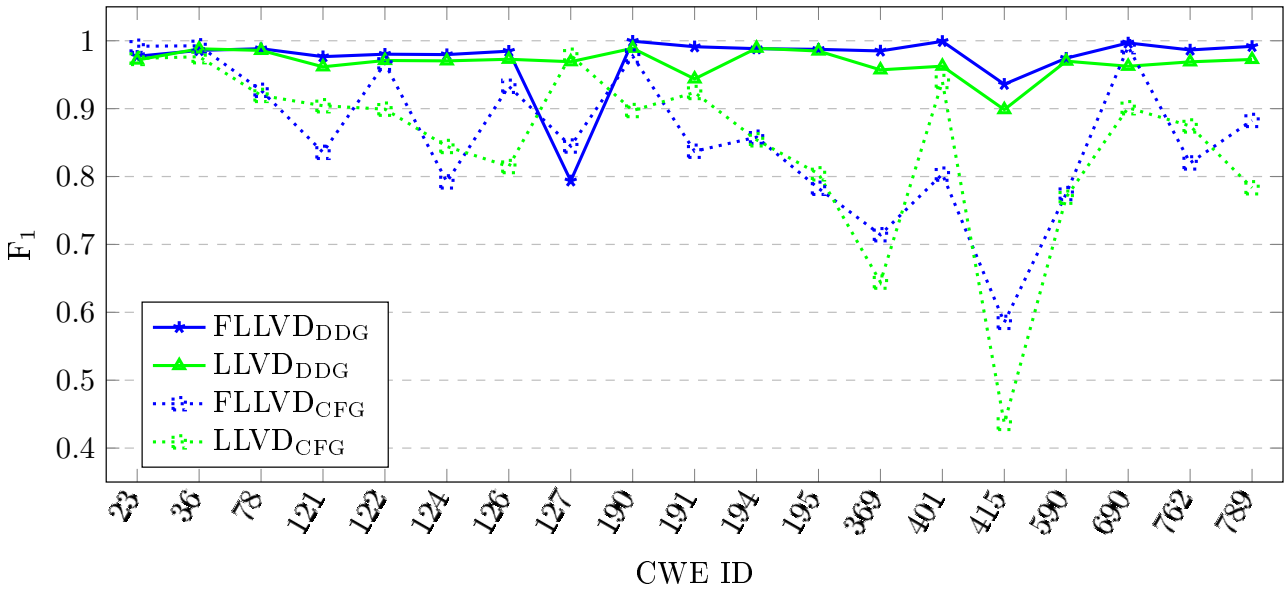


Figure 8. Results of our binary classification in terms of F_1 metric

Table 3. Recommended parameters for training

Parameter	Value
Learning rate	10^{-5}
Weight decay	1.3×10^{-6}
Loss Lambada	1.3×10^{-6}
Activation	ReLU
Epochs	100
Batch size	32
Drop-out	0.5
Optimization	Adams
Aggregate	Mean
loss	CategoricalCrossEntropy

FLLVD$_{DDG}$ for CWE 23 and 36. These vulnerabilities correspond to relative and absolute 'Path Traversal. It seems that the code embedding for file-system calls (e.g., open) is sufficient to catch this vulnerability, regardless of its dependency on previous or subsequent statements.

We examined our multi-class vulnerability detection algorithms using DDG and PDG on a subset of SARD, as shown in Table 2. For each vulnerability class, 1,000 graphs were randomly chosen, resulting in a balanced dataset across all vulnerability classes. However, the imbalance between vulnerable and non-vulnerable instances still remains. The results are shown in Figure 9. In contrast to binary classification, this experiment analyzes all vulnerabilities simultaneously. In addition to the 20 vulnerability types, a separate class for non-vulnerable cases (referred to as 'Benign') is also included in the detection process. The overall F_1 (macro average) is 0.833, 0.902, 0.787 and 0.938 for LLVD$_{DDG}$, FLLVD$_{DDG}$, LLVD$_{PDG}$ and FLLVD$_{PDG}$, respectively. The important issues that we observe are :

- Except a few cases, FLLVD has better performance in terms of F_1 for both code representation graph. Further, the improvement for PDG is higher than DDG. More specifically, the gap between LLVD$_{DDG}$ and FLLVD$_{DDG}$ is about 0.07 while the gap between LLVD$_{PDG}$ and FLLVD$_{PDG}$ reachs to 0.15. Similar to binary classification, we find that graph-level prediction operates more precisely than node-level prediction in detecting vulnerability types.
- The performance of PDG is better than DDG for FLLVD and vice versa for LLVD. More specifically, the gain of PDG over DDG is about 3.6% for FLLVD while it becomes -1.5% for LLVD. It demonstrates that control dependency in PDG improves FLLVD performance comparing to DDG while it provides some noise for LLVD.
- The detection rate of FLLVD for non-vulnerable code is approximately 100%, while for LLVD, it is about 99%. It is mainly originated from the fact that non-vulnerable portion of the dataset is larger than the vulnerable portion. Table 2 shows that vulnerable functions in SARD account for approximately 26% of the total functions. This imbalance ratio worsens at the node level, with the proportion of vulnerable nodes constituting about 2% of the total number of nodes

In summary, we can argue that FLLVD$_{PDG}$) outperforms other variants except for CWE 369, which corresponds to 'divide by zero' where FLLVD$_{DDG}$

demonstrates better performance by about 2%.

4.4 Results for SARD+BigVul

As mentioned above, BigVul samples are very limited in both types and number. More specifically the available vulnerability types are CWE-ID 78, 134, 191, 369 and 415. Therefore, we combined BigVul samples with SARD to achieve a sufficient number of samples while also ensuring a balanced set of different vulnerabilities. In this way, we arrive at a dataset containing approximately 600 graphs for each vulnerability type. It is worth noting that the graphs are randomly selected from both vulnerable and non-vulnerable samples. Further, F_1 scores were obtained using the average results from 5-fold cross-validation.

The results for both LLVD and FLLVD, are represented at Table 4. We can see that :

- Similar to SARD result, Except CWE 369, FLLVD has better performance in terms of F_1 for both DDG and PDG. More specifically, the gap between LLVD_{DDG} and $\text{FLLVD}_{\text{DDG}}$ is about 0.06 while the gap between LLVD_{PDG} and $\text{FLLVD}_{\text{PDG}}$ reachs to 0.1.
- In contrast to SARD results, the performance of $\text{FLLVD}_{\text{DDG}}$ is better than $\text{FLLVD}_{\text{PDG}}$ in terms of overall F_1. It is mainly originated from the ppor performance of $\text{FLLVD}_{\text{PDG}}$ in detection of CWE 415 corresponding to "Double free". It seems that control dependency in PDG provides some noise for FLLVD in detecting CWE 415.

Table 4. Results for SARD + BigVul dataset in terms of F_1.

CWE ID	LLVD_{DDG}	$\text{FLLVD}_{\text{DDG}}$	LLVD_{PDG}	$\text{FLLVD}_{\text{PDG}}$
0	0.99	0.99	0.99	0.99
78	0.83	0.85	0.79	0.87
134	0.69	0.96	0.68	0.88
191	0.73	0.82	0.27	0.78
369	0.78	0.62	0.84	0.76
415	0.56	0.73	0.36	0.26
$\overline{F_1}$	0.76	**0.82**	0.66	0.76

4.5 Comparison to previous schemes

Table 5 presents comparative results for the state of the art in vulnerability detection. Although this is not a fair comparison, mainly due to the use of different datasets in the evaluations, it provides a brief overview of the specifications, strengths, and limitations of each scheme. This table highlights the respective datasets, classification variables, detection levels, and average F1 scores. The third column indicates the datasets on which the respective schemes are trained or tested, including various combinations of existing datasets such as SARD, NVD, and BigVul, or a customized dataset, demonstrating the breadth and contextual focus of the datasets. The forth column indicates the number of distinct classes used in each scheme. A higher value suggests a more complex classification task that the model is designed to handle.

We can see that most of the work involves binary classification, just distinguishing between vulnerable and non-vulnerable code, regardless of the type of vulnerability. This limitation may restrict their applicability in scenarios with diverse codebases or various vulnerability types that must be accounted for. VulDeePecker+, μVolDeePecker, and VULEXPLAINER are the only schemes that provide multi-class vulnerability detection. However, these schemes are coarse-grained, detecting vulnerabilities at the function or code segment level and leaving ambiguity about the exact location of the vulnerabilities.

Here, $\textbf{FLLVD}_{\textbf{PDG}}$ demonstrates a superior performance with an F_1 score of 0.938, indicating a strong balance between precision and recall. This performance is very close to the performance of VulDeePecker+ and μVolDeePecker as multi-class detection schemes by regarding that they are more coarse grained than our scheme.

The analysis distinctly emphasizes the trade-off between the number of classes, detection level and detection performance. Our proposed methods demonstrate that a higher number of classes can still achieve high detection accuracy relative to others in the same category. Overall, our schemes provide evidence of state-of-the-art effectiveness in vulnerability detection when compared to various alternatives across different detection levels and datasets, particularly in the context of software vulnerability detection, where precise and contextual detection is crucial.

5 Conclusions

We proposed a novel multi-class vulnerability detection scheme leveraging graph neural networks to address the pressing need for effective vulnerability identification in software systems. By integrating Line-Level Vulnerability Detection (LLVD) and Function-Level Vulnerability Detection (FLVD), we established a comprehensive multi-granularity approach known as the Function/Line-Level Vulnerability Detection (FLLVD) scheme. This method not only enhances detection accuracy but also enables us to find both the type and location of each vulnerability within the given source code. The evaluation demonstrates promising results, with LLVD and FLLVD achieving

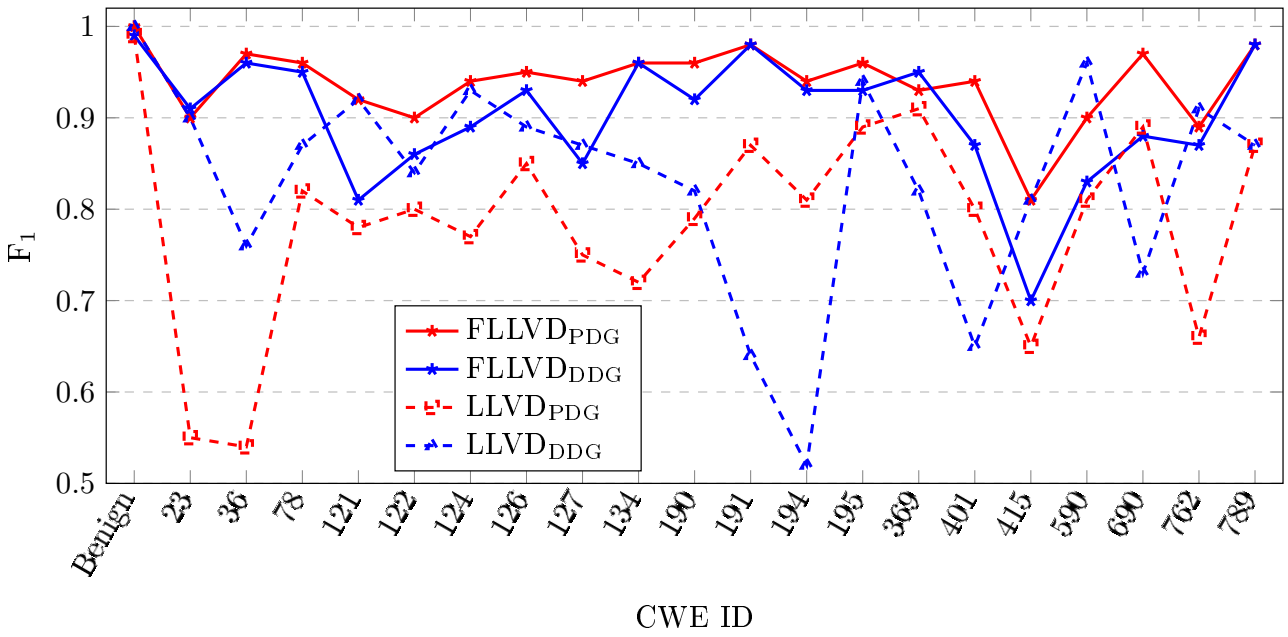


Figure 9. Performance of our multi-class detection in terms of F_1 metric

Table 5. Comparison of our schemes with prior work in terms of N_{class}, detection level and F_1 metric.

Scheme	Cite	Dataset	N_{class}	Detection level	$\overline{F_1}$
FLLVD$_{\mathbf{PDG}}$	This work	SARD	21	Line	0.938
FLLVD$_{\mathbf{DDG}}$	This work	SARD+BigVul	6	Line	0.828
SlicedLocator	[14]	SARD+NVD+CVEFixes[37]	2	Line	0.687
MAVLD	[16]	SARD+Draper	2	Line	0.99
VULGAI	[15]	SARD+NVD	2	Line	0.845
LineVD	[18]	BigVul	2	Statement	0.360
ISVSF	[17]	SARD+NVD	2	Statement	0.965
SedSVD	[19]	SARD+NVD	2	Statement	0.951
mVulPreter	[13]	NVD	2	Code segment	0.581
VulDeePecker	[8]	SARD	2	Code segment	0.934
VulDeePecker	[8]	NVD	2	Code segment	0.905
VulDeePecker+	[8]	SARD+NVD	41	Code segment	0.936
μVulDeePecker	[10]	SARD+NVD	41	Code segment	0.946
CODEJIT	[20]	Custom	2	Code segment	0.84
Devign	[1]	NVD	2	Function	0.557
VULMG	[2]	SARD	2	Function	0.944
VDRG	[3]	SARD+Deving+Reveal	2	Function	0.812
VULEXPLAINER	[4]	BigVul	45	Function	0.639
CS-GVD	[5]	CodeXGLUE	2	Function	0.603
SDV	[6]	CHRDB	2	Function	0.396
RGAN	[7]	Reveal	2	Function	0.473

substantial improvements in their F_1 metrics.

Our findings underscore the significance of employing a multi-granularity perspective in vulnerability detection, effectively bridging the gap between fine-

grained and coarse-grained analyses. The proposed schemes can serve as foundational tools for further advancements in the field, paving the way for more reliable and secure software development practices. Future work should focus on expanding the dataset variations, refining the graph representations, and exploring the adaptation of our scheme to emerging vulnerability types, thereby enhancing its applicability in a continuously evolving threat landscape.

Finally, this work contains numerous assumptions in design model that provides an opportunity for improvement regarding practicality. We suggest to consider different conditions to eliminate some assumption of the current work:

- **Detection of multiple vulnerabilities within a function :** Although FLLVD sets a single label on all detected vulnerable lines, LLVD can set multiple lines as vulnerable with a different set of labels. It needs a new dataset containing functions with multiple point of vulnerabilities.
- **Multi-granularity with different set of labels :** Although FLLVD supports multi-granularity, it lacks from setting different labels on vulnerable detected lines. Fully supporting of multi-granularity demands for redesigning FLLVD and accordingly improving update-prediction stage.

It is worth noting that each of the above assumptions changes the existing model and will be subject of an independent work.

Appendix A. My Appendix

CRediT authorship contribution statement

Hamidreza M. Taheri: Algorithm Implementation, Software, Data curation, Writing - Original draft preparation. **Alireza Shafieinejad:** Conceptualization of this study, Methodology, Verification.

References

[1] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*, 32, 2019.

[2] Zhang Haojie, Li Yujun, Liu Yiwei, and Zhou Nanxin. Vulmg: A static detection solution for source code vulnerabilities based on code property graph and graph attention network. In *2021 18th International Computer Conference on Wavelet Active Media Technology and Information Processing (ICCWAMTIP)*, pages 250–255. IEEE, 2021.

[3] Hongyu Yang, Haiyun Yang, Liang Zhang, and Xiang Cheng. Source Code Vulnerability Detection Using Vulnerability Dependency Representation Graph. In *2022 IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 457–464. IEEE, 2022.

[4] Michael Fu, Van Nguyen, Chakkrit Kla Tantithamthavorn, Trung Le, and Dinh Phung. Vulexplainer: A transformer-based hierarchical distillation for explaining vulnerability types. *IEEE Transactions on Software Engineering*, 2023.

[5] Wei Tang, Mingwei Tang, Minchao Ban, Ziguo Zhao, and Mingjun Feng. Csgvd: A deep learning approach combining sequence and graph embedding for source code vulnerability detection. *Journal of Systems and Software*, 199:111623, 2023.

[6] Chunyong Zhang and Yang Xin. Static vulnerability detection based on class separation. *Journal of Systems and Software*, 206:111832, 2023.

[7] Mingwei Tang, Wei Tang, Qingchi Gui, Jie Hu, and Mingfeng Zhao. A vulnerability detection algorithm based on residual graph attention networks for source code imbalance (rgan). *Expert Systems with Applications*, 238:122216, 2024.

[8] 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.

[9] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 19(4):2244–2258, 2021.

[10] Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li, and Hai Jin. μVulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection. *IEEE Transactions on Dependable and Secure Computing*, 18(5):2224–2236, 2021. .

[11] Son Nguyen, Thu-Trang Nguyen, Thanh Trong Vu, Thanh-Dat Do, Kien-Tuan Ngo, and Hieu Dinh Vo. Code-centric learning-based just-in-time vulnerability detection. *Journal of Systems and Software*, page 112014, 2024.

[12] Zhen Li, Deqing Zou, Shouhuai Xu, Zhaoxuan Chen, Yawei Zhu, and Hai Jin. Vuldeelocator: a deep learning-based fine-grained vulnerability detector. *IEEE Transactions on Dependable and Secure Computing*, 19(4):2821–2837, 2021.

[13] Deqing Zou, Yutao Hu, Wenke Li, Yueming Wu, Haojun Zhao, and Hai Jin. mvulpreter: A multi-granularity vulnerability detection system with

interpretations. *IEEE Transactions on Dependable and Secure Computing*, 2022.

[14] Bolun Wu, Futai Zou, Ping Yi, Yue Wu, and Liang Zhang. Slicedlocator: Code vulnerability locator based on sliced dependence graph. *Computers & Security*, 134:103469, 2023.

[15] Chunyong Zhang and Yang Xin. VulGAI: vulnerability detection based on graphs and images. *Computers & Security*, 135:103501, 2023.

[16] Miles Q Li, Benjamin CM Fung, and Ashita Diwan. A Novel Deep Multi-head Attentive Vulnerable Line Detector. *Procedia Computer Science*, 222:35–44, 2023.

[17] Haibin Zhang, Yifei Bi, Hongzhi Guo, Wen Sun, and Jianpeng Li. ISVSF: Intelligent vulnerability detection against java via sentence-level pattern exploring. *IEEE Systems Journal*, 16(1): 1032–1043, 2021.

[18] David Hin, Andrey Kan, Huaming Chen, and M Ali Babar. Linevd: Statement-level vulnerability detection using graph neural networks. In *Proceedings of the 19th international conference on mining software repositories*, pages 596–607, 2022.

[19] Yukun Dong, Yeer Tang, Xiaotong Cheng, Yufei Yang, and Shuqi Wang. SedSVD: Statement-level software vulnerability detection based on Relational Graph Convolutional Network with subgraph embedding. *Information and Software Technology*, 158:107168, 2023.

[20] Nicholas Saccente, Josh Dehlinger, Lin Deng, Suranjan Chakraborty, and Yin Xiong. Project achilles: A prototype tool for static method-level vulnerability detection of Java source code using a recurrent neural network. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, pages 114–121. IEEE, 2019.

[21] Ana Fidalgo, Ibéria Medeiros, Paulo Antunes, and Nuno Neves. Towards a deep learning model for vulnerability detection on web application variants. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 465–476. IEEE, 2020.

[22] Xiao Cheng, Haoyu Wang, Jiayi Hua, Guoai Xu, and Yulei Sui. Deepwukong: Statically detecting software vulnerabilities using deep graph neural network. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(3):1–33, 2021.

[23] G Renjith and S Aji. Vulnerability Analysis and Detection Using Graph Neural Networks for Android Operating System. In *Information Systems Security: 17th International Conference, ICISS 2021, Patna, India, December 16–20, 2021, Proceedings 17*, pages 57–72. Springer, 2021.

[24] Sicong Cao, Xiaobing Sun, Lili Bo, Ying Wei, and Bin Li. Bgnn4vd: Constructing bidirectional graph neural-network for vulnerability detection. *Information and Software Technology*, 136:106576, 2021.

[25] Yueming Wu, Deqing Zou, Shihan Dou, Wei Yang, Duo Xu, and Hai Jin. VulCNN: An Image-inspired Scalable Vulnerability Detection System. 2022.

[26] 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)*, pages 757–762. IEEE, 2018.

[27] Laura Wartschinski, Yannic Noller, Thomas Vogel, Timo Kehrer, and Lars Grunske. VUDENC: vulnerability detection with deep learning on a natural codebase for python. *Information and Software Technology*, 144:106809, 2022.

[28] Benjamin Steenhoek, Md Mahbubur Rahman, Richard Jiles, and Wei Le. An empirical study of deep learning models for vulnerability detection. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2237–2248. IEEE, 2023.

[29] Chenyuan Zhang, Hao Liu, Jiutian Zeng, Kejing Yang, Yuhong Li, and Hui Li. Prompt-enhanced software vulnerability detection using chatgpt. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, pages 276–277, 2024.

[30] Cho Do Xuan, Dat Bui Quang, and Vinh Dang Quang. A novel approach for software vulnerability detection based on ensemble learning model. *Computers and Electrical Engineering*, 130:110848, 2026.

[31] Abdechakour Mechri, Mohamed Amine Ferrag, and Merouane Debbah. Secureqwen: Leveraging llms for vulnerability detection in python codebases. *Computers & Security*, 148:104151, 2025.

[32] Aleksei Shestov, Rodion Levichev, Ravil Mussabayev, Evgeny Maslov, Pavel Zadorozhny, Anton Cheshkov, Rustam Mussabayev, Alymzhan Toleu, Gulmira Tolegen, and Alexander Krassovitskiy. Finetuning large language models for vulnerability detection. *IEEE Access*, 2025.

[33] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

[34] Martin Simonovsky and Nikos Komodakis. Dynamic edge-conditioned filters in convolutional

neural networks on graphs. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3693–3702, 2017.

[35] https://huggingface.co/neulab/codebert cpp. Hugging face: Codebert c++ pretrained model, 2024. URL `https://huggingface.co/neulab/codebert-cpp`. Accessed: 2024-9-20.

[36] https://joern.io. Joern, the bug hunter's workbench, 2024. URL `https://joern.io`. Accessed: 2024-9-20.

[37] Guru Bhandari, Amara Naseer, and Leon Moonen. Cvefixes: automated collection of vulnerabilities and their fixes from open-source software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 30–39, 2021.

Hamidreza M. Taheri was born in Tehran, Iran in 1999. He received the B.S. degree in computer engineering from Isfahan University of Technology, in 2021. He was also received M.Sc. degree in computer engineering from Tarbiat Modares University, Tehran, Iran, in 2024. His research area includes vulnerability detection and graph neural networks. He also works as a security engineer.

Alireza Shafieinejad is an Assistant Professor in the Department of Electrical and Computer Engineering, Tarbiat Modares University, Tehran, Iran. He received the B.S. degree in computer engineering from the Sharif University of Technology, Tehran, in 1999. He was also received M.Sc. and Ph.D. degrees in electrical engineering from Isfahan University of Technology, Isfahan, Iran, respectively in 2002 and 2013. Since 2015, he has been an Assistant Professor at Tarbiat Modares University, Tehran, Iran. His research interests are in the area of Wireless Networks, Network, cloud and Social Media Security. At Tarbiat Modares University, he leads research in online social media security in SNS-Lab (Social Network Security Laboratory).