LIVABLE: Exploring Long-Tailed Classification of Software Vulnerability Types

Xin-Cheng Wen, Cuiyun Gao, Feng Luo, Haoyu Wang, Ge Li, and Qing Liao

***Abstract*—Prior studies generally focus on software vulnerability detection and have demonstrated the effectiveness of Graph Neural Network (GNN)-based approaches for the task. Considering the various types of software vulnerabilities and the associated different degrees of severity, it is also beneficial to determine the type of each vulnerable code for developers. In this paper, we observe that the distribution of vulnerability type is long-tailed in practice, where a small portion of classes have massive samples (i.e., head classes) but the others contain only a few samples (i.e., tail classes). Directly adopting previous vulnerability detection approaches tends to result in poor detection performance, mainly due to two reasons. First, it is difficult to effectively learn the vulnerability representation due to the over-smoothing issue of GNNs. Second, vulnerability types in tails are hard to be predicted due to the extremely few associated samples. To alleviate these issues, we propose a Long-taIled software VulnerABiLity typE classification approach, called LIVABLE. LIVABLE mainly consists of two modules, including (1) vulnerability representation learning module, which improves the propagation steps in GNN to distinguish node representations by a differentiated propagation method. A sequence-to-sequence model is also involved to enhance the vulnerability representations. (2) adaptive re-weighting module, which adjusts the learning weights for different types according to the training epochs and numbers of associated samples by a novel training loss. We verify the effectiveness of LIVABLE in both type classification and vulnerability detection tasks. For vulnerability type classification, the experiments on the Fan et al. dataset show that LIVABLE outperforms the state-of-the-art methods by 24.18% in terms of the accuracy metric, and also improves the performance in predicting tail classes by 7.7%. To evaluate the efficacy of the vulnerability representation learning module in LIVABLE, we further compare it with the recent vulnerability detection approaches on three benchmark datasets, which shows that the proposed representation learning module improves the best baselines by 4.03% on average in terms of accuracy.**

***Index Terms*—Software vulnerability, deep learning, graph neural network.**

Manuscript received 16 June 2023; revised 11 March 2024; accepted 11 March 2024. Date of publication 11 April 2024; date of current version 14 June 2024. This research was supported in part by the National Key R & D Program of China under Grant 2022YFB3103900, in part by the Natural Science Foundation of Guangdong Province under Project 2023A1515011959, in part by Shenzhen-Hong Kong Jointly Funded Project under Category A SGDX20230116091246007, in part by Shenzhen Basic Research under General Project JCYJ20220531095214031, in part by Shenzhen International Science and Technology Cooperation Project under Grant GJHZ20220913143008015, in part by the Major Key Project of PCL under Grant PCL2022A03, in part by the National Natural Science Foundation of China under Grant 62072007, Grant 62192733, Grant 61832009, Grant 62192731, and Grant 62192730, and in part by the Key Program of Hubei under Grant JD2023008. Recommended for acceptance by J. Sun. *(Corresponding author: Cuiyun Gao.)*

Xin-Cheng Wen, Feng Luo, and Qing Liao are with Harbin Institute of Technology, Shenzhen 518055, China (e-mail: xiamenwxc@foxmail.com; 190110308@stu.hit.edu.cn; liaoqing@hit.edu.cn).

Cuiyun Gao is with Harbin Institute of Technology, Shenzhen 518055, China, and also with Peng Cheng Laboratory and Guangdong Provincial Key Laboratory of Novel Security Intelligence Technologies, Shenzhen 518055, China (e-mail: gaocuiyun@hit.edu.cn).

Haoyu Wang is with Huazhong University of Science and Technology, Wuhan 430074, China (e-mail: haoyuwang@hust.edu.cn).

Ge Li is with Peking University, Beijing 100871, China (e-mail: lige@pku.edu.cn).

Digital Object Identifier 10.1109/TSE.2024.3382361

I. Introduction

SOFTWARE vulnerabilities are important and common security threats in software systems. These vulnerabilities can be easily exploited by attackers and have the potential to cause irreparable damage to software systems [1]. For example, buffer overflow issues [2] allow attackers to exploit vulnerabilities to tamper with memory data or gain control of the system. Vulnerabilities are inevitable for many reasons, e.g. the complexity of software and the steady growth in the size of the Internet [3]. Vulnerability detection has received intensive attention in the software community recently.

With the development of deep learning (DL) techniques, various DL-based vulnerability detection methods have been proposed [4], [5], [6], [7], [8], [9], which aim to detect whether a code function is vulnerable or not. For example, SyseVR [6] combines multiple sequence features to generate code slices and uses a bidirectional Recursive Neural Network (RNN) [10] to detect vulnerabilities. Recent studies [7], [8], [9] have demonstrated that Graph Neural Networks (GNNs) [11] are effective in vulnerability detection by capturing the structural information of source code. For example, Devign [7] applies the Gated Graph Neural Network (GGNN) [12] to learn the representation of code structure graph for vulnerability detection, where the code structure graph is the combination of Abstract Syntax Tree (AST), Control Flow Graph (CFG), Data Flow Graph (DFG) and Natural Code Sequence (NCS).

Despite that the previous studies [7], [8], [9] can inform developers of the vulnerable functions in source code, the developers may still feel difficult to fix the vulnerabilities. Considering the various types of vulnerabilities and the associated different degrees of severity [14], [15], predicting the vulnerability types can assist developers in arranging the maintenance priority and localizing the vulnerable cause. Recently, Zou et al. [16] leverage function call information to conduct multi-class vulnerability detection, which requires project-level information and can hardly be applied to predict the vulnerability type

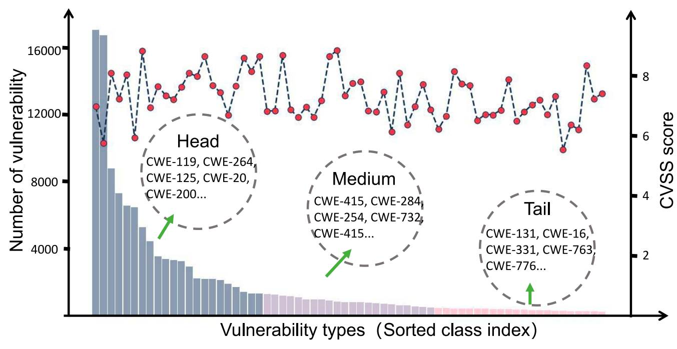


Fig. 1. The label distribution (bar chart) and the corresponding CVSS score (line chart) of National Vulnerability Database (NVD) data [13] during the period of 2012 to 2022.The head, medium, and tail classes respectively contain 27, 45 and 253 classes in NVD respectively.

of source code in function-level. To our best knowledge, no prior research has explored the type classification problem for vulnerable functions.

For facilitating analysis, we first study the collected National Vulnerability Database (NVD) [13] data during the period of 2012 to 2022, as illustrated in Fig. 1. We observe that the sample size of different vulnerability types generally presents a long-tailed distribution. Specifically, only a small portion of classes have massive samples (i.e., head classes) while the others contain extremely few samples (i.e., tail classes). Although the vulnerable code in tails is limited in number, the degree of severity would be high. According to *Common Vulnerability Scoring System (CVSS)* [17], which characterizes the degree of severity into a score of 1-10, the average CVSS score[1] of all the classes in tails is 7.01, indicating a high-level severity [19]. Some of the most threatening vulnerability types in tails such as CWE-507 and CWE-912 are with a CVSS score of 9.8. For example, the CWE-507 (Trojan Horse) [20] is a virus that is added to supply chain software by malware, leading to serious security risks. Therefore, effectively classifying the vulnerable types including the tail classes is important for software security.

Directly applying the existing vulnerable detection methods [7], [8] is one possible solution. However, they tend to fail in the long-tailed scenario, because: (1) They are difficult to effectively learn the vulnerability representation due to the over-smoothing issue [21] of GNNs. Prior studies [22] show that the performance of GNNs will degrade as the number of layers increases, leading to similar representations for different nodes. However, few layers in GNNs will make the model hard to capture the structural information of vulnerable code, due to the deep levels in the code structure graph [23], [24]. (2) It is hard to predict the vulnerability types in tails which are associated with extremely few samples. The serious imbalance between the numbers of different vulnerability types renders the models biased towards head classes, leading to poor representations of tail classes. The models are prone to producing head classes while ignoring the tails.

To mitigate the above challenges, we propose a **L**ong-ta**I**led software **V**ulner**AB**i**L**ity typ**E** classification approach, called LIVABLE. LIVABLE mainly consists of two modules: (1) **a vulnerability representation learning module**, which involves a differentiated propagation-based GNN for alleviating the over-smoothing problem. For further enhancing the vulnerability representation learning, a sequence-to-sequence model is also involved to capture the semantic information. (2) **an adaptive re-weighting module**, which adaptively updates the learning weights according to the vulnerable types and training epochs. The module is designed to well learn the representations of tail classes based on the limited samples.

The experimental evaluation is performed on both vulnerability type classification and vulnerability detection tasks. For vulnerability type classification, we prepare the evaluation dataset by extracting the vulnerable samples from Fan et al. [25]. The results demonstrate that the LIVABLE outperforms the state-of-the-art methods by 24.18% in terms of the accuracy metric, by improving the performance in predicting medium and tail class by 14.7% and 7.7%, respectively. To evaluate the efficacy of the vulnerability representation learning module in LIVABLE, we further compare with the recent vulnerability detection approaches on FFMPeg+Qemu [7], Reveal [8], and Fan et al. [25] datasets, which shows that the proposed representation learning module improves the best-performing baselines by 4.03% on average in terms of accuracy.

The major contributions of this paper are as follows:

1) We are the first to approach to explore the long-tailed classification of vulnerable functions.
2) We propose LIVABLE, a long-tailed software vulnerability type classification approach, including: 1) a vulnerability representation learning module for alleviating the over-smoothing issue of GNNs and enhancing the vulnerability representations; and 2) an adaptive re-weighting module that involves a novel training objective for balancing the weights of different types.
3) Extensive experiments show the effectiveness of LIVABLE in vulnerability type classification and the efficacy of vulnerability representation learning module in vulnerability detection.

The rest of this paper is organized as follows. Section II describes the background. Section III details the two components in the proposed framework of LIVABLE, including the vulnerability representation learning module and adaptive re-weighting module. Section IV describes the evaluation methods, including the datasets, baselines, implementation and metrics. Section V presents the experimental results. Section VI discusses some cases and threats to validity. Section VIII concludes the paper.

II. Background

A. Graph Neutral Networks

Graph Neural Networks (GNNs) have demonstrated superior performance at capturing the structural information of source code in the software vulnerability detection task [7], [8], [9]. The widely-used GNN methods include Graph Convolutional Network (GCN) [26], Graph Attention Network (GAT) [27], GGNN. The two-layer of GCN model can calculate as:

$$H = softmax(\hat{\hat{A}}ReLU(\hat{\hat{A}}XW_0)W_1) \quad (1)$$

[1]In this paper, we use the *Common Vulnerability Scoring System V3 Score [18].*

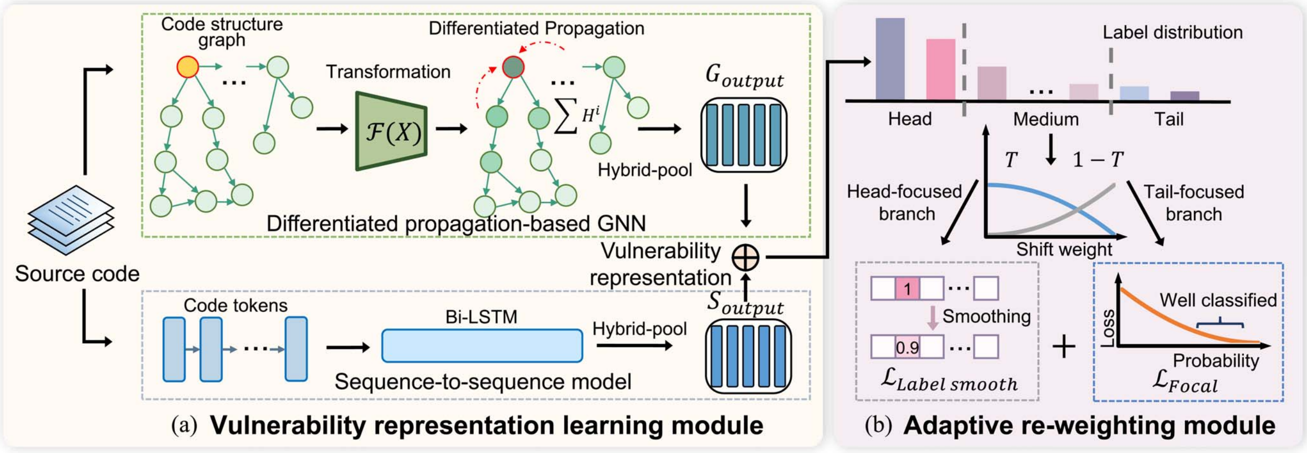


Fig. 2. The architecture of LIVABLE, which mainly contains two components: (A) a vulnerability representation learning module, and (B) an adaptive re-weighting module.

where $\hat{\hat{A}}$ denotes the adjacency matrix and W_0, W_1 are trainable weight matrices. X and H denote the initial and output node representations respectively. The over-smoothing [22] problem means that each node vector converges to a similar vector in the multi-layer GNN. It is hard to distinguish the node from the whole graph [28]. However, due to the depth of the code structure graph [23], [24], a two-layer GNN is hard to well capture the code node representations. While methods such as adding layers without message passing have been proposed to enhance the discriminative capacity of GNNs, these approaches fall short for vulnerability detection and type classification tasks. This is because vulnerable nodes typically constitute a small part of the entire node set yet wield great influence over the results, necessitating a focus on enhancing the initial node vector representations. To mitigate the over-smoothing issue and distinguish the node representations, a vulnerability representation learning module is proposed by improving the node feature.

B. Long-Tailed Learning Methods

In real-world scenarios, data typically exhibit a long-tailed distribution [29], where a small portion of classes contain massive samples while the others are associated with only a few samples. Long-tailed learning methods have been widely studied in the computer vision field [29], [30], [31]. There are some popular methods to alleviate long-tailed problems, such as class-balanced strategies [32], [33] and smoothing strategies [30], [34]. For example, one popular method Focal Loss [33] adjusts the standard cross-entropy loss [35] by reducing the learning weight for well-classified samples and focuses more on misclassified samples during model training, calculated as:

$$\mathcal{L}_{FL} = -\sum_{i=1}^{n} (1-\hat{y_i})^{\gamma} y_i log(\hat{y_i}) \tag{2}$$

where y_i and $\hat{y_i}$ denote the label distribution in the ground truth and the predicted output, respectively. γ is the modulating factor for focusing on misclassified samples.

Label smooth cross-entropy loss [30] is also a widely used long-tailed solution. It uses smoothing strategies to encourage the model to be less confident in head classes:

$$\mathcal{L}_{LSCE} = -\sum_{i=1}^{n} log(\hat{y_i}) \left((1-\epsilon)y_i + \epsilon\delta_i\right) \tag{3}$$

where ϵ denotes a smoothing parameter, and δ_i denotes the uniform distribution to smooth the ground-truth distribution y_i.

Although the existing methods alleviate the problem of long-tailed distributions in computer vision, no studies have explored the performance of these methods in software vulnerability classification. To fill the gap, we experimentally evaluate the popular long-tailed learning methods, and propose a novel adaptive learning method for vulnerability type classification.

III. Proposed Framework

In this section, we first formulate the long-tailed problem and then describe the overall framework of the proposed LIVABLE. As shown in Fig. 2, LIVABLE consists of two main components: (1) a vulnerability representation learning module for enhancing the vulnerability representations; and (2) an adaptive re-weighting module that involves a novel training objective for balancing the weights of different types.

A. Problem Formulation

The problem of the long-tailed vulnerability type classification is formulated below. We denote the unbalanced dataset with training instances collected from the real world as $D = \{x_i, y_i\}$, where x_i denotes a vulnerable function in raw source code and y_i denotes its corresponding vulnerability type. Assume that all classes are ordered by cardinality, i.e., if $N_a \geq N_b$ when the class index $a < b$, where N_a indicates the number of training samples for the class a. According to the typical division ratio in long-tailed scenario [34], [36], we classify the vulnerable types into three classes: head (>200 samples per type), medium (50-200 samples per type), and tail (<50 samples per type). As a result, the head, medium, and tail classes contain 12, 11, and 46 types, respectively. The goal of LIVABLE is to learn a mapping, $f : x_i \mapsto y_i, y_i$ denotes the label distribution, to predict the class of vulnerabilities function. Typically, we use the Cross-Entropy (CE) loss [35] function as follows:

$$\mathcal{L}_{CE} = -\sum_{i=1}^{n} y_i log(\hat{y_i}) \tag{4}$$

where n denotes the number of categories and $\hat{y}_i$ denotes the output predicted by the model.

B. Vulnerability Representation Learning Module

In this section, we elaborate on the proposed vulnerability representation learning module, which involves differentiated propagation-based GNN for capturing the structural information of source code, and combines the sequence-to-sequence approach for further enhancing the vulnerability representations.

1) Differentiated Propagation-Based GNN: Following the prior studies [7], we extract the same code structure graph of source code for vulnerability representation. To alleviate the over-smoothing issue of GNN, we design a differentiated propagation method to distinguish node representations.

We denote the code structure graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ as the input, where $\mathcal{V}$ denotes the set of nodes v and $\mathcal{E}$ denotes the set of edges e in the graph. The differentiated propagation-based GNN is decoupled into two steps, including node feature transformation and propagation. The representation for each node v is first initialized as a 128-dimensional vector by the Word2Vec [37] model. To enhance the representation of each node vector, the feature transformation step then encodes the vector feature based on a Gated Recurrent Unit (GRU) layer:

$$H^{(0)} = \mathcal{F}(X) \tag{5}$$

where X is the initial node representation of node v, and $\mathcal{F}$ denotes a GRU layer.

In the domain of software vulnerability detection and classification, the initial representation of vulnerability nodes is crucial. It is attributed to the fact that the initial node vectors, representing source code, directly contribute to the emergence of vulnerabilities. Furthermore, the edge between nodes denotes the structural relationships in the source code. To avoid convergent node vectors during propagation, we propose to explicitly involve the initial node representations. The node representation H^l at the l-th layer is defined as:

$$H^l = \frac{1}{l}\sum_{i=0}^{l-1}\left((1-\alpha)\left(\widetilde{D}^{-\frac{1}{2}}\widetilde{A}\widetilde{D}^{-\frac{1}{2}}\right)^i H^0 + \alpha H^{(0)}\right) \tag{6}$$

where $H^{(0)}$ is the transformed feature representation computed by Equation (5). $\widetilde{A}$ is the adjacency matrix of the code structure graph with self-connections and $\widetilde{D}$ is the degree matrix of $\widetilde{A}$. α denotes a teleport hyperparameter, which determines the importance of the initial node features during the propagation. Specifically, LIVABLE integrates an initial vector representation into the node vector at each propagation step. This integration is pivotal in fostering distinctiveness among nodes. In the process of establishing global connections, LIVABLE adopts a strategic approach by moderatively attenuating the full values of the node vector. This decrease is progressively implemented, with an increased reduction in vector weight correlating with increasing node distance. Such a strategy enhances the discriminative ability of the nodes, thereby enhancing the overall effectiveness for vulnerability detection and classification.

Finally, the graph representation G_{output} of the code structure graph $\mathcal{G}$ is calculated through a hybrid pooling layer:

$$G_{output} = \mathcal{C}\left(AvgPool(H_v) + MaxPool(H_v)\right), \forall v \in \mathcal{V} \tag{7}$$

where $AvgPool(\cdot)$ and $MaxPool(\cdot)$ indicate the average pooling operation [38] and maximum pooling operation [39], for better capturing the local information and global information of the code structure graph [40], respectively. $\mathcal{C}(\cdot)$ denotes two Multi-Layer Perceptron (MLP) layers.

2) Sequence-to-Sequence Model: To further enhance the vulnerability representations, we involve a sequence-to-sequence model to capture the semantics information. Specifically, given a code snippet S and its composed code token sequence $S = \{t_1, ..., t_i, ..., t_n\}$, where t_i denotes the i-th token and n denotes the total number of tokens, the sequence representation S_{output} is calculated as follows:

$$S_{output} = \mathcal{C}\left(AvgPool(s_{t_i}) + MaxPool(s_{t_i})\right), \forall t_i \in S \tag{8}$$

$$s_{t_1}, ..., s_{t_n} = \left(\overrightarrow{LSTM}\,(t_1, ..., t_n)\right) \mid\mid \left(\overleftarrow{LSTM}\,(t_1, ..., t_n)\right) \tag{9}$$

where the symbol $||$ is the concatenation operation. $\rightarrow$ and $\leftarrow$ denote the Bi-directional Long Short-Term Memory (LSTM) [41] operations respectively.

Finally, the representation for each vulnerable code is computed by combining the two outputs:

$$\mathcal{O} = G_{output} + S_{output}. \tag{10}$$

C. Adaptive Re-Weighting Module

The adaptive re-weighting module is proposed to better learn the representations for vulnerability types with different numbers of samples. A novel training objective is designed to adjust the learning weights for different types of vulnerabilities according to the training epochs and a number of associated samples.

Before the module design, we first investigate the popular long-tailed learning methods in vulnerability type classification, such as class-balanced strategies [32], [33] and smoothing strategies [30], [34]. We experimentally evaluate the performance of these methods. And the results are illustrated in Table II and will be detailed in Section V-A. We find that focal loss [33] adds a modulating factor to focus more on tail samples. The label smooth CE loss [30] uses the smoothing strategies to reduce the focus on head classes. The experiment results also demonstrate that focal loss performs better for classifying tails, while the label smooth CE loss helps to improve the performance of head classes. Based on the observation, we propose a novel training objective that involves two training

TABLE I
THE SPECIFIC VULNERABILITY TYPES AND THEIR CORRESPONDING PROPORTION AND GROUP OF THE TYPES OF VULNERABILITIES IN THIS PAPER. CLASSES WITH A SAMPLE SIZE OF LESS THAN 20 ARE GROUPED TOGETHER IN THE REMAIN CLASS. "NONE TYPE" MEANS THE VULNERABILITY IS NOT CLASSIFIED INTO ANY CLASS. AS THESE VULNERABILITIES EXIST IN THE REAL WORLD AS WELL, THEY ARE ALSO CONSIDERED TO BE A VULNERABILITY TYPE

Types	Ratio	Group	Types	Ratio	Group
CWE-119	19.94%		CWE-415	0.76%	
None type	19.85%		CWE-732	0.62%	
CWE-20	10.71%		CWE-404	0.58%	
CWE-399	6.90%		CWE-79	0.52%	Medium
CWE-125	5.86%		CWE-19	0.52%	
CWE-264	4.76%		CWE-59	0.49%	
CWE-200	4.72%	Head	CWE-17	0.48%	
CWE-189	3.16%		CWE-400	0.45%	
CWE-416	3.09%		CWE-772	0.43%	
CWE-190	2.88%		CWE-269	0.36%	
CWE-362	2.61%		CWE-22	0.33%	Tail
CWE-476	2.02%		CWE-369	0.32%	
CWE-787	1.86%		CWE-18	0.32%	
CWE-284	1.66%	Medium	CWE-835	0.32%	
CWE-254	1.15%		Remaining class	1.57%	
CWE-310	0.88%				

branches, i.e., a "*tail-focused branch*" for tail classification and "*head-focused branch*" for head classification.

Specifically, these two branches take $\mathcal{O} = \{\hat{y}_i | i = 1, 2, .., n\}$ as input, where n denotes the number of types and $\hat{y}_i$ denotes the output predicted by the model. The adaptive re-weighting method $\mathcal{L}$ is calculated as follows:

$$\mathcal{L} = T \cdot \mathcal{L}_{FL}(\mathcal{O}) + (1 - T) \cdot \mathcal{L}_{LSCE}(\mathcal{O}) \tag{11}$$

where $\mathcal{L}_{FL}$ denotes the focal loss which focuses more on learning the representations of tail classes. $\mathcal{L}_{LSCE}$ denotes the label smooth CE loss focuses more on learning the representations of head classes. T denotes a learning weight for the two branches, which are calculated according to the training epochs:

$$T = 1 - \left(\frac{E_{now}}{E_{max}}\right)^2 \tag{12}$$

where E_{now} and E_{max} denote the current training epoch and the total number of training epochs, respectively. The design of T is based on the assumption that the representations of head classes are more easily learnt in the long-tailed scenario [42]. In the early training stage (i.e, with a smaller E_{now}), the design will enable the model to focus on learning the representations of tail classes. As the training epoch increases, T will gradually decrease, shifting the model' learning focus to head classes. T ensures that both branches are updated continuously throughout the training process, avoiding learning conflict between the two branches.

IV. EXPERIMENTAL SETUP

A. Research Questions

In order to evaluate LIVABLE, we answer the following research questions:

RQ1: How well does LIVABLE perform in classifying vulnerability types under the long-tailed distribution?

RQ2: How effective is the vulnerability representation learning module in vulnerability detection?

RQ3: What is the impact of different components on the type classification performance of LIVABLE?

RQ4: What is the influence of hyper-parameters on the performance of LIVABLE?

B. Datasets

Vulnerability type classification. To obtain the labels of vulnerabilities for type classification, we extract a new dataset from Fan et al. [25], which consists of different types of vulnerabilities from 2002 to 2019 and provides the *Common Weakness Enumeration IDentifier* (CWE ID) [43] for each vulnerable function.

We obtain a total of 10,667 vulnerable functions. Among these, 8,783 functions are associated with 91 different kinds of direct CWE-IDs. The remaining functions, which do not align with any specific CWE class, are classified under the "None Type" class. This classification is also recognized as a distinct category of vulnerability type in the training process. Specifically, as shown in Table I, we demonstrate the involved vulnerability types and the proportion distributions of head, medium, and tail classes in our dataset. The percentage of samples in the head, medium, and tail classes amount to 86.50%, 9.52%, and 4.00% respectively. Moreover, we group categories with a sample size of less than 20 as a new class named *Remaining class*.

Vulnerability detection. Our study employs three representative vulnerability datasets: FFMPeg+Qemu [7], Reveal [8], and Fan et al. [25]. The FFMPeg+Qemu dataset, previously utilized in Devign [7] with +22K data, exhibits a vulnerability rate of 45.0% among all instances. The Reveal dataset consists of +18K instances, and the proportion of vulnerable to non-vulnerable data is 1:9.9. The Fan et al. dataset includes a total of 188,636 C/C++ function samples, among which 5.7% are vulnerable. We do not use FFMPeg+Qemu [7] and Reveal [8] datasets for vulnerability type classification since these two datasets do not contain type information of vulnerability.

C. Baseline Methods

In vulnerability detection task, we compare LIVABLE with three sequence-based methods [4], [5], [6] and three state-of-the-art graph-based methods [7], [8], [9].

1) **VulDeePecker** [4]: VulDeePecker embeds the data flow dependency to construct the code slices. Then, it adopts the BiLSTM to detect buffer error vulnerabilities and resource management error vulnerabilities.
2) **Russell et al.** [5]: Russell et al. involves the Convolutional Neural Network (CNN), integrated learning and random forest to vulnerability detection.
3) **SySeVR** [6]: SySeVR constructs the program slices by combining control flow dependency, data flow dependency, and code statement. It embeds program slices as input and uses Recursive Neural Networks (RNN).

4) **Devign** [7]: Devign constructs code structure graph from functions and leverages Gated Graph Neural Network (GGNN) for classification.
5) **Reveal** [8]: Reveal leverages code property graph (CPG) as input and also adopts GGNN for extracting features. Then it utilizes a multi-layer perceptron for vulnerability detection.
6) **IVDetect** [9]: IVDetect constructs Program Dependency Graph (PDG) and designs the feature attention GCN for vulnerability detection.

In vulnerability type classification, we compare LIVABLE with graph-based methods: Devign and Reveal. These methods are designed for the software vulnerability detection task and achieve the best-performing results. We also use the focal loss [33] and label smooth CE (LSCE) loss [30] to replace CE loss, which has been introduced in Section III-C. Furthermore, we compare with label aware smooth loss [34], class-balanced loss [32], and class-balanced focal loss [32] that mitigates the imbalance of long-tailed distribution. Label aware smooth loss adds a probability distribution factor based on LSCE, which considers the predicted probability distributions of different classes. Class-balanced loss and class-balanced focal loss are modified from CE loss and focal loss, respectively. They add a weight inversely related to the number of class samples to tackle long-tailed distribution.

D. Implementation Details

In the experiment section, we randomly split the dataset into training, validation, and testing sets by the number of classes in a ratio of 8:1:1. To ensure the fairness of the experiments, we use the same data split in all experiments. We train the vulnerability detection model for 100 epochs and the vulnerability type classification model for 50 epochs on a server with an NVIDIA GeForce RTX 3090.

Following the previous work [8], [44], we use Joern [45] to create the code structure graph in the graph branch. We leverage Word2Vec [46] to initialize the node representation in the graph branch and the token representation in the sequence branch, where both vectors have a dimension of $d = 128$. The number of layers in the graph branch is $L = 16$ and the limitation and the source code length in the sequence branch is $n = 512$. The dimension of the hidden vector in the sequence branch is set as 512.

E. Performance Metrics

We use the following four widely-used performance metrics in our evaluation:

Accuracy: $Accuracy = \frac{TP+TN}{TP+FP+TN+FN}$. Accuracy is the proportion of correctly classified instances to all instances. *TP* is the number of true positives, *TN* is the number of true negatives. and $TP + FP + TN + FN$ represents the number of all instances. To evaluate the classification performance of different classes of vulnerabilities, we use $Head$, $Medium$, $Tail$ to denote the proportion of the head, medium, and tail CWE samples that are detected correctly, respectively.

Precision: $Precision = \frac{TP}{TP+FP}$. Precision is the proportion of relevant instances among those retrieved. *TP* is the number of true positives and *FP* is the number of false positives.

Recall: $Recall = \frac{TP}{TP+FN}$. Recall is the proportion of relevant instances retrieved. *TP* is the number of true positives and *FN* is the number of false negatives.

F1 score: $F1score = 2 * \frac{Precision*Recall}{Precision+Recall}$. F1 score is the geometric mean of precision and recall and indicates the balance between them.

V. Experimental Results

A. RQ1: Evaluation on Vulnerability Types Classification

To answer this research question, we compare our approach with the previous methods of vulnerability type classification under long-tailed distribution.

1) Compared With Baselines: As shown in Table II, the proposed LIVABLE consistently outperforms all the baseline methods in the long-tailed distribution. Our approach achieves an accuracy of 64.01%, which is an absolute improvement of 26.74% and 24.18% over Devign and Reveal, respectively. Compared with the baseline methods, we focus on modeling vulnerability patterns from two perspectives based on the graph branch and the sequence branch. The results show that our approach can learn vulnerabilities more efficiently from different perspectives. For the head, medium and tail classes, we can see that LIVABLE achieves the best performance compared to the previous methods, with the improvement of 25.40%, 16.17% and 7.7% respectively. It demonstrates that our approach is better at capturing the differences between the different vulnerability types.

Fig. 3 shows the results of LIVABLE with Devign and Reveal in terms of vulnerability type classification, with respect to each vulnerability type. In the head class, LIVABLE performs better in the detection of unknown types (i.e. None type) of vulnerabilities. This may be due to that the vulnerability representation learning module captures more information from the code snippets and obtains a more discriminative vulnerability representation. In the medium and tail classes, LIVABLE also performs better than the previous method, especially for CWE-787, CWE-284 and CWE-25, CWE-22, etc. This is due to the fact that LIVABLE gives higher weighted attention to the tail class samples. In general, LIVABLE outperforms baselines for classifying vulnerability types.

Furthermore, the experimental results indicate an enhanced performance of Reveal in tail classes compared to head and medium CWE classes. This improvement may stem from the application of resampling methods during the two-step phase of the training process, directing greater focus toward the tail training data. However, in vulnerability type detection, the long-tail situation inherent in multi-class classification leads to data loss for certain classes under the resampling approach, consequently impeding the learning process for several mid-tier and header classes. The proposed adaptive re-weighting module is demonstrated to mitigate the issue, enhancing accuracy by 1.61% and 4.41% for the head and medium classes, respectively.

TABLE II
UNDER THE LONG-TAIL DATA DISTRIBUTION, COMPARISON RESULTS BETWEEN LIVABLE AND THE BASELINES IN VULNERABILITY TYPE CLASSIFICATION. VRL MODULE DENOTES THE VULNERABILITY REPRESENTATION LEARNING MODULE

Baseline / Metrics(%)	Devign				Reveal				VRL Module in LIVABLE			
Re-Weighting	Head	Medium	Tail	Accuracy	Head	Medium	Tail	Accuracy	Head	Medium	Tail	Accuracy
CE loss	38.26	7.35	23.08	34.99	39.39	42.65	46.15	39.83	60.61	51.47	38.46	59.32
Label aware smooth loss	37.14	25.00	23.08	35.70	38.26	42.65	46.15	38.83	63.50	52.94	46.15	62.16
Label smooth loss	37.78	4.41	23.08	34.28	39.07	42.65	46.15	39.54	63.50	55.88	46.15	62.45
Class-balanced loss	27.97	36.76	38.46	29.02	37.30	44.12	46.15	38.12	60.45	54.41	53.85	59.74
Class-balanced focal loss	24.44	7.35	23.08	22.76	27.17	42.65	46.15	29.02	54.82	54.41	53.85	54.77
Focal loss	38.91	14.71	38.46	37.27	38.59	41.18	46.15	38.98	61.09	52.94	53.85	60.17
Adaptive re-weighting module	39.27	37.29	50.00	39.39	41.00	47.06	46.15	41.68	**64.79**	**58.82**	**53.85**	**64.01**

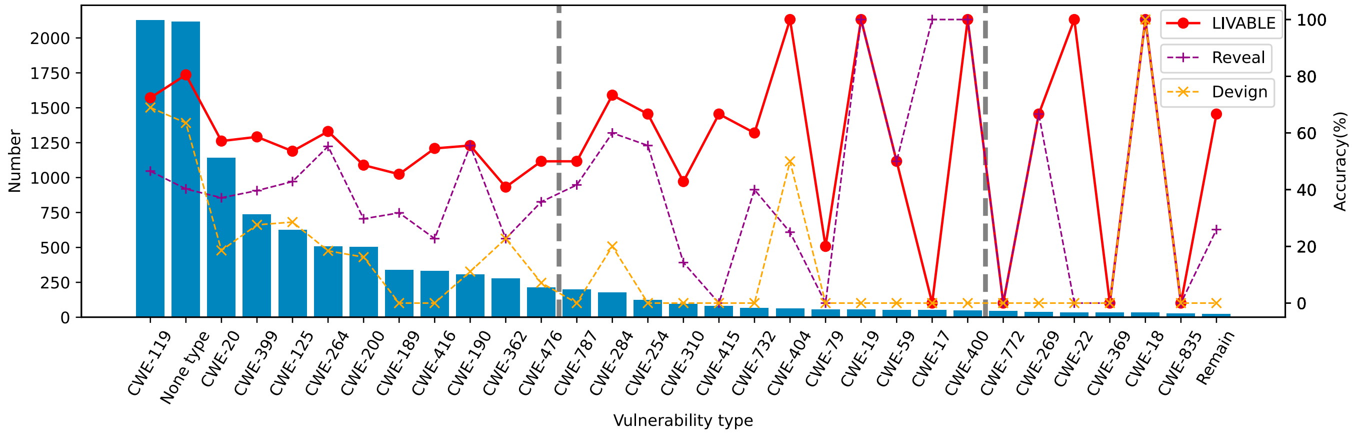


Fig. 3. The accuracy of each vulnerability type in LIVABLE. The X-axis denotes the vulnerability type. The Y-axis on the left and right indicate the number of vulnerabilities and accuracy respectively. The lines in yellow, purple, and red represent the accuracy of devign, reveal, and LIVABLE respectively.

TABLE III
EXPERIMENT RESULTS FOR REVEAL AND LIVABLE. "*" DENOTES STATISTICAL SIGNIFICANCE IN COMPARISON TO REVEAL IN TERMS OF ACCURACY (I.E., TWO-SIDED T-TEST WITH P-VALUE < 0.05)

Dataset	Metrics	Reveal	LIVABLE	P-Value
Fan et al.	Head	39.39	**64.79***	1.76e-18
	Medium	42.65	**58.82***	2.26e-08
	Tail	46.15	**53.85***	1.07e-02
	Accuracy	39.83	**64.01***	6.21e-17

The superior performance of the Devign method in tail classes, as opposed to medium classes, might be attributed to the vulnerability types that are easy to learn, such as CWE-18. Such type vulnerabilities are consistently identified across all baseline models. Nevertheless, the limited sample of vulnerabilities in the tail category leads to an inflated accuracy rate for tail class vulnerabilities in the Devign.

Following the previous work [44], [47], we use the statistical significance tests (t-test) to validate that the significance the proposed LIVABLE is better than the baselines. The results are listed in Table III. Our results show that LIVABLE outperforms the Reveal in terms of accuracy at a significance level of 0.05 (p-value is 1.76e-18, 2.26e-8, and 1.07e–2** on the head, medium, and tail classes, respectively).

2) Compared With Re-Weighting Methods: In this subsection, we experiment with different re-weighting methods to solve the long-tailed problem of vulnerability type classification.

Cross-entropy Loss: When using the CE-loss, our LIVABLE is more accurate in identifying head classes than tail classes. CE-Loss encourages the whole model to be overconfident in the head classes for massive data. The experiment shows that the head class samples performed 9.14% and 22.15% better than the medium and tail classes, respectively, in terms of accuracy.

Smoothing strategies: Smoothing strategies include label smooth loss and label-aware smooth loss. They are another regularization technique that encourages the model to be less over-confident in the head classes. We use these methods to replace CE loss and they both improve the accuracy of the head class by 2.89% in LIVABLE. The use of the smoothing strategies in other methods did not work, which is probably due to the poor vulnerability representation they captured.

Class-balanced strategies: The class-balanced strategies consist of the class-balanced loss and class-balanced focal loss, which assigns different weights for classes and instances. They lead to an average increase in accuracy of 2.94% and 15.39% for the medium and tail classes respectively, but an average decrease of 2.98% for the head classes. It shows that class-balanced strategies focus more on the tail data but will cause a decrease in the accuracy of the head classes.

TABLE IV
COMPARISON RESULTS BETWEEN VULNERABILITY REPRESENTATION LEARNING (VRL) MODULE AND THE BASELINES ON THE THREE DATASETS IN VULNERABILITY DETECTION. "-" MEANS THAT THE BASELINE DOES NOT APPLY TO THE DATASET IN THIS SCENARIO. THE BEST RESULT FOR EACH METRIC IS HIGHLIGHTED IN BOLD. THE SHADED CELLS REPRESENT THE PERFORMANCE OF THE TOP-3 BEST METHODS IN EACH METRIC. DARKER CELLS REPRESENT BETTER PERFORMANCE. "*" DENOTES STATISTICAL SIGNIFICANCE IN COMPARISON TO REVEAL IN TERMS OF ACCURACY (I.E., TWO-SIDED T-TEST WITH P-VALUE < 0.05)

Dataset / Metrics(%)	FFMPeg+Qemu [7]				Reveal [8]				Fan et al. [25]			
Baseline	Accuracy	Precision	Recall	F1 score	Accuracy	Precision	Recall	F1 score	Accuracy	Precision	Recall	F1 score
VulDeePecker	49.61	46.05	32.55	38.14	76.37	21.13	13.10	16.17	81.19	38.44	12.75	19.15
Russell et al.	57.60	54.76	40.72	46.71	68.51	16.21	52.68	24.79	86.85	14.86	26.97	19.17
SySeVR	47.85	46.06	58.81	51.66	74.33	40.07	24.94	30.74	90.10	30.91	14.08	19.34
Devign	56.89	52.50	64.67	57.95	87.49	31.55	36.65	33.91	92.78	30.61	15.96	20.98
Reveal	61.07	55.50	70.70	62.19	81.77	31.55	**61.14**	41.62	87.14	17.22	34.04	22.87
IVDetect	57.26	52.37	57.55	54.84	-	-	-	-	-	-	-	-
VRL module	**64.84***	**57.87**	**80.67***	**67.39***	**93.53***	**52.27***	50.55	**51.50***	**95.05***	**40.04***	**37.27***	**38.60***

Adaptive re-weighting module: Compared with other re-weighting methods, the adaptive re-weighting module achieves the highest accuracy in all four metrics. Specifically, the adaptive re-weighting module achieves an absolute improvement of 4.24% in overall accuracy.The tail-focused branch leverages the advantages of class-balanced strategies, which assign higher weights to the more challenging tail samples in the early stages of training. It improves the average accuracy in medium and tail classes by 5.15% and 5.13% respectively. In the later phase of training, the head-focused branch involves the smoothing strategy, which calculates the loss of the soft value of the label. It is effective as a means of coping with label noise [48], [49] in the vulnerability types classification. The shifting weight ensures that the head and tail classes can be constantly learned in the whole training process, which avoids conflicts with each other. In summary, the adaptive re-weighting module is able to learn representations of tail data efficiently based on a limited number of samples.

Answer to RQ1: LIVABLEoutperforms all the baseline methods in terms of accuracy for head classes, medium classes, and tail classes, as well as for all samples. In particular, the adaptive re-weighting module achieves better performance than existing re-weighting methods.

B. RQ2: Performance on Vulnerability Detection

To answer this research question, we explore the performance of the vulnerability representation learning (VRL) module in LIVABLE and compare it with other baseline methods. Table IV shows the overall results.

The proposed VRL module outperform all the baseline methods: Overall, VRL module achieves better results on the FFMPeg+Qemu, Reveal and Fan et al. datasets. When considering all the performance metrics regarding the three datasets (12 combination cases), VRL module obtains the best performance in 11 out of 12 cases. On all three datasets, VRL module outperforms all baseline methods in terms of accuracy and F1 score metrics. Especially in terms of F1 score, the relative improvements are 8.36%, 23.74%, and 68.78%, respectively. Compared to the best-performing baseline method Reveal, VRL module achieves an average improvement of 9.88%, 67.49%, 2.09%, and 33.63% on the four metrics across all datasets. This indicates that the VRL module performs better on vulnerability detection than previous methods, obtaining a more discriminative code representation.

In the statistical tests for vulnerability detection, we conduct experiments across three datasets, encompassing four metrics, thereby resulting in twelve cases. We observe that LIVABLE demonstrated significant improvements in eleven out of these twelve metrics, each exhibiting a p-value of less than 0.05. This finding substantiates the statistically significant enhancement in the performance of LIVABLE.

The combination of the graph-based branch and the token-based branch can achieve better performance: The experimental results also show that the graph-based methods perform better in most cases. The token-based methods perform well on some metrics. For example, in Fan et al., the token-based methods obtain better performance than graph-based methods on precision metrics overall. The reason may be attributed to that token-based methods are better at capturing semantic information in the code, whereas graph-based methods are more attentive to the structure information in the code. Thus, VRL module combines the advantages of both graph-based and token-based methods to capture more information and obtain a more discriminating code representation in vulnerability detection.

Answer to RQ2: The vulnerability representation learning module outperforms all baseline methods in terms of accuracy and F1 scores in vulnerability detection. In particular, it improves the accuracy of the three datasets by 3.77%, 6.04% and 2.27% respectively over the best baseline method.

C. RQ3: Ablation Study

To answer this research question, we explore the effect of vulnerability representation learning module and adaptive re-weighting module on the performance of LIVABLE by performing an ablation study on vulnerability type classification.

1) Vulnerability Representation Learning Module: To explore the contribution of the vulnerability representation

TABLE V
RESULTS OF ABLATION STUDY IN VULNERABILITY TYPE CLASSIFICATION

Metrics(%) \ Methods	w/o Sequence Representation				w/o Graph Representation				LIVABLE			
Methods	Head	Medium	Tail	Accuracy	Head	Medium	Tail	Accuracy	Head	Medium	Tail	Accuracy
w/o adaptive	46.95	25.00	38.46	44.67	56.59	32.35	38.46	53.91	60.61	51.47	38.46	59.32
w/o tail-focused branch	49.20	44.12	30.77	48.36	60.29	47.06	38.46	58.61	63.50	55.88	46.15	62.45
w/o head-focused branch	48.71	33.82	30.77	46.94	62.38	51.47	46.15	61.02	61.41	45.59	38.46	59.46
LIVABLE	-	-	-	-	-	-	-	-	**64.79**	**58.82**	**53.85**	**64.01**

learning module, we construct the following two variants in the module for comparison: (1) only using differentiated propagation-based GNN (denoted as w/o sequence representation) to validate the impact of the sequence-to-sequence model (2) only using sequence-to-sequence model (denoted as w/o graph representation) to verify the effectiveness of differentiated propagation-based GNN.

As shown in Table V, all variations contribute positively to the overall performance. This indicates that the combination of differentiated propagation-based GNN and sequence-to-sequence model can better help the model to capture more discriminative representations from the source code. Overall, the impact of combining sequence-to-sequence model is greater, which achieves a 14.65% improvement while combining differentiated propagation-based GNN gives only a 5.41% performance improvement.

2) Adaptive Re-Weighting Module: Then we explore the contribution of the adaptive re-weighting module to the performance of vulnerable type classification. We construct three variations, including (1) using the cross-entropy loss instead of the adaptive re-weighting module (denoted as w/o adaptive) (2) removing the tail-focused branch while keeping the head-focused branch (denoted as w/o tail-focused branch) (3) removing head-focused branch while keeping tail-focused branch (denoted as w/o head-focused branch).

The results of ablation studies are shown in Table V. We find that the performance of all the variants is lower than LIVABLE, which indicates that all the components contribute to the overall performance of LIVABLE. The head-focused branch has more influence on the differentiated propagation-based GNN, while the tail-focused branch enables the sequence-to-sequence model to have a greater performance improvement.

Answer to RQ3: The vulnerability representation learning module combines the differentiated propagation-based GNN and sequence-to-sequence model, in which both contributes significantly to the performance of LIVABLE, with an improvement of 5.41% and 14.65% respectively in terms of accuracy. The adaptive re-weighting module also has a positive effect on model performance.

D. RQ4: Parameter Analysis

To answer this research question, we explore the impact of hyper-parameters in the LIVABLE, including the layer number of the differentiated propagation-based GNN, and the hidden size in the sequence-to-sequence model.

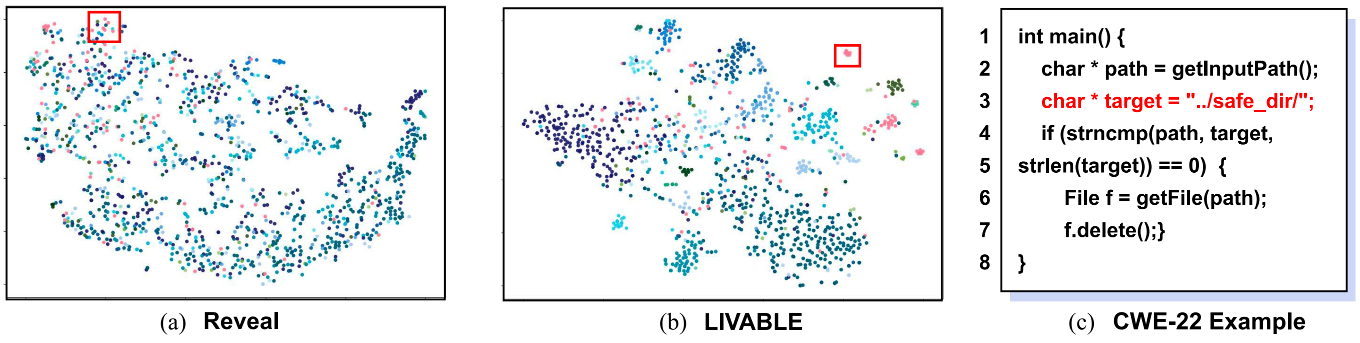
TABLE VI
THE IMPACT OF THE NUMBER OF GNN LAYERS IN THE GRAPH BRANCH AND HIDDEN SIZE IN THE SEQUENCE BRANCH ON THE PERFORMANCE OF LIVABLE

Graph Branch		Sequence Branch	
Layer Number	Accuracy (%)	Hidden Size	Accuracy (%)
12	62.07	128	56.53
14	62.78	256	59.23
16	**64.01**	**512**	**64.01**
18	62.92	768	62.07
20	61.50	1024	62.93

1) Number of GNN Layers: Table VI shows the accuracy of LIVABLE with different layers of GNN on the vulnerability type classification. In the previous work, GNNs usually achieve the best performance at 2 layers. However, with the number of layers set to 16, LIVABLE achieves a best performance of 64.01%. This indicates that LIVABLE can learn the relationships between more distant nodes than traditional methods, effectively alleviating the problem of over-smoothing. In addition, due to the limited number of nodes in the code structure graph, the number of layers of the GNN model cannot be increased indefinitely, and the accuracy rate starts to decrease when the number of layers equals 18. In summary, the model benefits from the use of more layers to capture relationships between distant nodes, enhancing the model's ability to capture information about the graph structure.

2) Size of Hidden Layers: We also experiment with MLPs with different sizes of hidden neurons. We use the same setup in the previous experiments. We experiment with the task of vulnerability type classification. The results are shown in Table VI and show that the LIVABLE performs best when using a hidden size of 512. This may be due to the fact that the task requires the detection of a larger number of vulnerability types. When more dimensions are used (more than 512), the performance starts to drop, which may be due to the over-fitting problem.

Answer to RQ4: The hyper-parameter settings of GNN layer number and hidden size can impact the performance of LIVABLE. The proposed differentiated propagation-based GNN can use more layers for learning to distinguish node representations.



```
1 int main() {
2     char * path = getInputPath();
3     char * target = "../safe_dir/";
4     if (strncmp(path, target,
5 strlen(target)) == 0)  {
6         File f = getFile(path);
7         f.delete();}
8 }
```

Fig. 4. The case study of LIVABLE, which mainly contains three parts: (a) T-SNE visualization of reveal, (b) T-SNE visualization of LIVABLE, and (c) a CWE-22 [51] example in tail class. In (a) and (b), the blue dots represent head samples, the green dots represent mid samples and the pink dots represent tail samples. In (c), the red-colored code is vulnerable code.

VI. Discussion

A. Why Does LIVABLE Work?

In this section, we identify the following two advantages of LIVABLE, which can explain its effectiveness in vulnerability type classification. We visualize the vulnerability representations learnt by LIVABLE and the best baseline Reveal via the popular T-SNE technique [50], as shown in Fig. 4(a) and 4(b), respectively.

(1) The ability to learn the vulnerability representation. The proposed vulnerability representation learning module helps LIVABLE to learn the vulnerability representation. More specifically, the proposed differentiated propagation GNN branch alleviates the over-smoothing problem and the sequence branch enhances the model's ability for capturing semantics information. As shown in Fig. 4(a) and 4(b), compared with the Reveal, LIVABLE can enhance the discriminability of representation on the type classification, in which head and tail classes are clustered more tightly.

(2) The ability to capture the representation of the tail classes. Although Reveal also identifies some samples from the tail class, they are usually mixed and clustered with samples from other classes. In comparison, LIVABLE is more effective to detect samples of the tail class. Specifically, we focus on CWE-22 in tail classes as depicted in Fig. 4(c). When considering the Devign and Reveal benchmarks, the accuracy for classifying CWE-22 vulnerabilities stands at 0% accuracy. In comparison, LIVABLE achieves a 100% accuracy for classifying CWE-22. This example implies LIVABLE's effectiveness in detecting vulnerabilities in the tail classes.

One reason why this CWE-22 type of vulnerability is difficult to detect by previous vulnerability types is that the number of CWE-22 is limited, accounting for only 0.33% of the entire dataset, and the previous methods can not focus on vulnerabilities in tail classes thus limiting the classification of vulnerability types.

Another reason is that CWE-22 is very similar to some vulnerability types. the example code attempts to validate a given input path by checking it against an allowlist and once validated delete the given file. But the "../" sequence in line 3 will cause the program to delete the important file in the parent directory. This type of vulnerability usually has only a small number of statements related to the vulnerability, and is an access rights-related vulnerability. However, CWE-284 (Improper Access Control), for example, is also a related type of vulnerability, and CWE-284 is more concerned with incorrect privileges, permissions, ownership, etc. that are explicitly specified for either the user or the resource. either the user or the resource. This can lead to CWE-22 being easily obfuscated.

To alleviate the above problems, in this paper, we propose a novel adaptive learning method for unbalanced situations and a vulnerability representation learning module to improve the discriminative ability of GNNs.

TABLE VII
Comparison Results Between LIVABLE and Baselines on the Precision, Recall, Macro F1, Weighted F1 and Matthews Correlation Coefficient (MCC) Metrics

Metrics	Precision	Recall	Macro F1	Weighted F1	MCC
Devign	28.63	30.66	27.62	33.53	25.83
Reveal	39.30	32.86	37.40	40.93	33.70
LIVABLE	64.64	54.90	57.00	63.54	59.02

B. Performance for Long Tail Classes

In this section, we further use other metrics to compare our approach with the previous methods of vulnerability type classification under long-tailed distribution. Specifically, we compare the performance of LIVABLE with Devign and Reveal in precision, recall, macro F1 score, weighted F1 score and Matthews Correlation Coefficient (MCC) metrics [52], as summarized in Table VII.

The results demonstrate that LIVABLE surpasses both Devign and Reveal across all metrics in the vulnerability type classification. LIVABLE achieves superior experimental results when compared to the best-performing baseline in long-tail setting, which exhibits improvements of 25.3%, 22.04%, 19.6%, 22.61%, and 25.32% in terms of precision, recall, macro F1 score, weighted F1 score, and MCC, respectively. Notably, these enhancements are more substantial than those observed in accuracy metrics. This indicates that LIVABLE is particularly adept at identifying tail class's vulnerability types, thereby enhancing its performance in vulnerability type classification tasks.

C. Advantages of Adding a Seqeuence Model

First, prior research has shown that GNNs are good at capturing local information through neighboring nodes. However, their tendency towards over-smoothing limits their effectiveness by neglecting global information from the code structure graph. To alleviate this limitation, LIVABLE incorporates a sequence-to-sequence model to effectively capture global information from a source code perspective. Specifically, the integration of a Bi-LSTM branch enhances the model's capability to capture global information.

Second, in the vulnerability type classification, it is observed that understanding code semantics is more feasible than comprehending code syntax structure, especially when dealing with limited training data in long-tail situations. Structural information, due to its abstract relations, poses a greater challenge in interpretation as opposed to code semantics. Our empirical findings, as detailed in Table V, highlight the impact of integrating the sequence-to-sequence model, particularly in vulnerability type classification tasks. This enhancement yields a noteworthy 14.65% improvement, whereas combining differentiated propagation-based GNNs yields a comparatively modest 5.41% performance boost.

Finally, incorporating the sequence-to-sequence model into vulnerability analysis can be considered as adopting a multi-view approach, which substantially enriches the understanding of vulnerability patterns by LIVABLE. The optimal performance of the model will be achieved when it efficiently discerns vulnerability patterns from various perspectives.

D. Compared With Previous GNNs

To validate the effectiveness of the differentiated propagation-based GNN (DPGNN), we compare DPGNN with the previous GNN models. In recent years, GNNs have recently been applied to vulnerability detection tasks, such as Devign and Reveal utilize the Gated Graph Neural Network (GGNN), and LineVD employs the Graph Attention Network (GAT) for statement-level vulnerability detection. Moreover, approaches like Approximation Personalized Propagation of Neural Prediction (APPNP), which incorporates Personalized PageRank to enhance GNN's discriminative capabilities, and GCNII, employing initial residual and identity mapping, address over-smoothing problems in GNNs. We compare the performance of DPGNN with these four variants and set default parameters for all GNNs.

Table VIII shows the experimental results, where we present the Accuracy, Precision, Recall, Macro F1, Weighted F1 and MCC metrics in vulnerability type classification. We observe that DPGMM outperforms all the baseline methods, by 0.66% for Accuracy, 4.49% for Precision, 4.32% for Recall, 4.01% for Macro F1, 1.48% for weighted and 0.74% for MCC. These results demonstrate that the LIVABLE leverage with DPGNN is more effective than previous GNNs. For the ong-tail metric, DPGNN demonstrates even more pronounced improvements, averaging of increase 3.11% in Accuracy and 5.87% across the other five metrics when compared to the four previously mentioned GNN models.

TABLE VIII
COMPARISON RESULTS BETWEEN DIFFERENTIATED PROPAGATION-BASED GNN AND OTHER GNNS ON THE VULNERABILITY TYPE CLASSIFICATION TASK

Metrics	Accuracy	Precision	Recall	Macro F1	Weighted F1	MCC
GGNN	58.38	52.27	45.97	46.29	57.61	52.93
APPNP	60.09	58.26	48.87	49.51	60.05	55.19
GAT	61.79	55.76	46.26	47.93	61.09	56.95
GCN2	63.35	60.15	50.58	52.99	62.06	58.28
LIVABLE	64.01	64.64	54.90	57.00	63.54	59.02

TABLE IX
COMPARISON RESULTS BETWEEN LIVABLE AND REVEAL ON THE PRECISION, RECALL AND MACRO F1 METRICS

Metrics	Accuracy	Precision	Recall	Macro F1
Reveal	19.30	3.39	5.66	3.65
LIVABLE	36.38	2.59	7.14	3.81

E. Performance on the Dataset Split by Time

Following the previous work [4], [5], [7], [8], [9], we have used the random split to experiment. Furthermore, we employ the chronological partition way to conduct experiments in Fan et al. [25], i.e., splitting data by time. Specifically, we use the order of commit submission of vulnerability data in the Fan et al. dataset as a criterion for data partitioning. To be precise, we considered 80% of the data submitted first as the training set, while the data submitted in the last 10% as the testing set, and the remaining as the validation set. We compared with the best-performing baseline Reveal to evaluate the influence of the time factor. The detailed experimental results are shown in Table IX.

It can be seen that based on the division of time, LIVABLE gets three better performances out of a total of four cases. This illustrates that LIVABLE can obtain better experimental results than the best-performing baseline when considering the time factor. Specifically, LIVABLE improves the performance of 16.85% in Accuray, 1.48% in Recall and 0.16% in F1 scores, respectively. However, the random-split dataset also suffers from data leakage. In the dataset of split by time, all four metrics degrade to different degrees. The degradation in the accuracy metric reaches 27.63%.

F. Comparison With CWE Classification Methods

The recent approach TREEVUL [53] also aims to automatically predict the fine-grained CWE category for vulnerability commits. It utilizes the structure information of the CWE tree as prior knowledge for predicting the vulnerability types. Our proposed LIVABLE is essentially different from TREEVUL in the following aspects: **(1) Training input.** TREEVUL and LIVABLE exhibit disparities in the input. TREEVUL utilizes a training set that comprises code changes to construct a classifier, aiming at investigating the distinctions in code changes

TABLE X
COMPARISON RESULTS BETWEEN LIVABLE AND REVEAL ON THE PRECISION, RECALL AND MACRO F1 METRICS IN DIVERSEVUL DATASET

Metrics	Accuracy	Precision	Recall	Macro F1
Reveal	18.03	12.02	9.30	7.26
LIVABLE	19.83	23.63	13.14	11.41

to predict the vulnerability type. In contrast, LIVABLE only uses the source code as training input for vulnerability type classification. **(2) Vulnerability type.** Owing to constraints by the CVE recording system, vulnerability types exhibit both coarse and fine-grained mappings, with each vulnerability type potentially encompassing multiple child vulnerability types. Therefore, vulnerability types can be organized as a tree of multiple levels. TREEVUL defines three different levels to represent this hierarchical relationship, in which the lower and high levels denote the rough and precise mappings of vulnerability type, respectively. For example, a vulnerability example has the CWE-74 at the second level and also has the label CWE-707 at the first level, which results in multiple labels for each sample during training. TREEVUL is a hierarchical multi-label classification system. In contrast, LIVABLE is directly a single-label classification framework, which adopts the CWE type as the predicted type, which enables the prediction of a broader range of commits and CWE types.

In addition, as DiverseVul [54] claims, the Fan et al. dataset contains many irrelevant functions. We also experiment with DiverseVul. The experimental configuration remained consistent with the preceding experiment for vulnerability type prediction. We compare with the best-performing baseline Reveal to evaluate the DiverseVul. The detailed experimental results are shown in Table X.

LIVABLE exhibits superior performance when compared with Reveal, achieving improvements of 1.8%, 11.61%, 3.84%, and 4.15% in Accuracy, Precision, Recall, and F1 score, respectively. Compared with the Fan et al. dataset, DiverseVul has more vulnerability types and therefore lower accuracy in vulnerability type prediction.

G. Venn Diagram of LIVABLE

In addition to the accuracy of vulnerability type classification by each approach, we further evaluate the number of unique correct classifications. The figure presents the unique correct classifications among Devign, Reveal and LIVABLE, which can correctly predict in the format of Venn diagrams. Our analysis revealed that LIVABLE successfully identified 139 unique correct classifications, outperforming with Reveal and Devign securing 88 and 24 unique instances, respectively. It indicates that the contribution of LIVABLE cannot be replaced by the combination of existing baselines.

H. Limitations

Despite LIVABLE achieving performance surpassing previous methods in vulnerability type classification, it still only

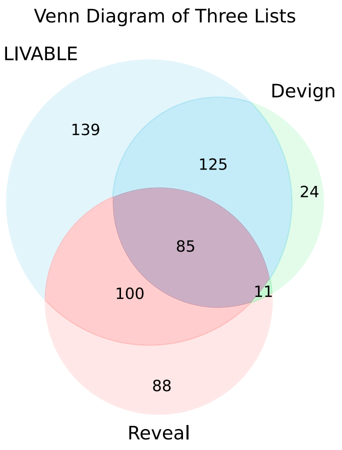
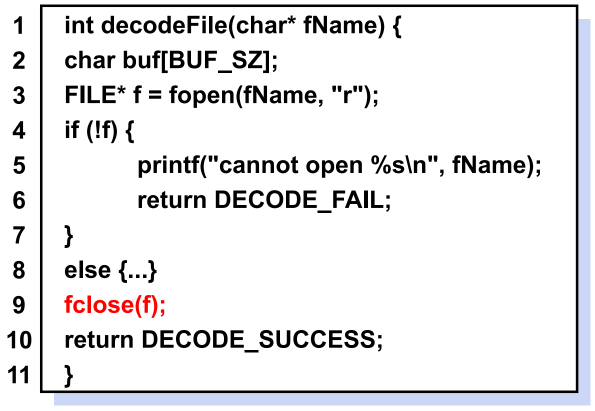


Fig. 5. Venn diagram of correct samples provided by LIVABLE and baselines.



```
int decodeFile(char* fName) {
char buf[BUF_SZ];
FILE* f = fopen(fName, "r");
if (!f) {
        printf("cannot open %s\n", fName);
        return DECODE_FAIL;
}
else {...}
fclose(f);
return DECODE_SUCCESS;
}
```

Fig. 6. A source code of CWE-772 example. The red-colored code is vulnerable code.

attains an accuracy rate of 65%. This limitation may stem from several factors: (1) The sample size of tail types is limited and easily confused with other head classes, such as CWE-772 (Missing Release of Resource after Effective Lifetime), which have shown a consistent zero accuracy rate in vulnerability type classification across Devign, Reveal, and LIVABLE. It only constitutes merely 0.43% of the entire dataset. It is easily coupled with potential confusion with other classes, which makes it challenging to capture these types of vulnerabilities, even using re-weighting strategies. (2) The vulnerability pattern is difficult to capture. As illustrated in Fig. 6, the example of CWE-772 does not close the file handle it opens in case of an error occurrence. This oversight can lead to the exhaustion of file handles for long-lived processes. The error in this instance is implicit, resulting from a rearrangement of statements that could fix the vulnerability. Therefore, these vulnerabilities are prone to misclassification due to their implicit vulnerability patterns.

I. Threats to Validity

One threat to validity comes from the size of our constructed dataset. Following the previous methods, we use a C/C++

dataset built for vulnerability detection. In the task of vulnerability type classification, we extract 91 types of vulnerability functions from this dataset and construct the new dataset. However, in the real world, there are more than 300 types of vulnerabilities. In the future, we will collect a more realistic benchmark for evaluation.

The second threat to validity is that our proposed LIVABLE has only experimented on the C/C++ dataset. Experiments have not been conducted on datasets from other programming languages, such as Java and Python. In the future, we will select more programming languages and corresponding datasets to further evaluate the effectiveness of LIVABLE.

The third threat comes from the implantation of baselines. Since Devign [7] does not publish their implementation and hyper-parameters. So we reproduce Devign based on the Reveal's [8] implementation and re-implement the method to the best of our abilities.

An additional threat arises from employing function-level granularity in vulnerability detection and classification tasks. In vulnerability detection, LIVABLE faces difficulty in identifying inter-function vulnerabilities when there are no obvious vulnerabilities within a singular function. In the vulnerability type classification, LIVABLE is restricted to detecting solitary types of vulnerabilities, precluding the identification of more intricate, composite vulnerability types. We plan to further explore this issue in the future.

Finally, in this study, following the previous research, we adopt function-level granularity for both vulnerability detection and type classification. Then, the function-level granularity detection facilitates effective model learning. Previous studies [7], [8], [25] have established that in the process of vulnerability detection, extensive code segments significantly degrade performance. Hence, we refrain from employing larger granularities, such as file-level, for tasks related to vulnerabilities. Moreover, the dataset is typically collected from developers' commits. During the submission process, numerous unchanged functions exist. This situation introduces a substantial risk of erroneous labeling in cross-function scenarios.

VII. Related Work

A. Learning-Based Vulnerability Detection

In recent years, learning-based methods have been widely used for vulnerability detection tasks. The researcher first uses the Machine Learning (ML)-based method [55], [56], [57], [58], [59] in vulnerability detection. For example, Neuhaus et al. [60] extract the dependency information matrix and vulnerability vectors from the source code, and use the support vector machine to detect them. Grieco et al. [61] abstract features from the C-standard library and use multiple machine learning models, such as logistic regression and random forest.

ML-based methods rely heavily on manual feature extraction, which is time-consuming and may require much effort. Therefore, Deep Learning (DL)-based methods [62], [63], [64], [65], [66] learn the input representation from the source code, which can better capture the vulnerability patterns. Depending on the input generated from source code and training model types, DL-based approaches can group into two different types: sequence-based and graph-based methods. Sequence-based approaches utilize source code tokens as their model inputs. VulDeePecker [4] extracts data flow information from source code and adapts BiLSTM to detect buffer error vulnerabilities. SySeVR [6] combines both control flow and data flow to generate program slices and utilizes a bidirectional RNN for code vulnerability detection. Graph-based methods capture more structural information than sequence-based methods, which achieve better performance on vulnerability detection tasks. Devign [7] builds graphs combining AST, CFG, DDG and NCS edges and uses the GGNN model for vulnerability detection. IVDetect [9] constructs the Program Dependency Graph (PDG) and proposes feature attention GCN to learn the graph representation.

However, directly adopting these approaches tends to result in poor performance in vulnerability type classification, which is limited by class-imbalanced and over-smoothing problems in GNNs. In this paper, we propose a vulnerability representation learning module to boost the model to learn the vulnerability representation and capture vulnerability types.

B. Class Re-Weighting Strategies

Long-tailed classification has attracted increasing attention due to the prevalence of imbalanced data in real-world applications [29], [30], [31], [42], [67]. It leads to a small portion of classes having massive sample points but the others contain only a few samples, which makes the model ignore the identification of tail classes. Recent studies have mainly pursued re-weighting strategies. For example, Lin et al. propose Focal Loss [33], which adjusts the standard CE-loss to reduce the relative loss for well-classified samples and focus more on rare samples that are misclassified during model training. Cui et al. [32] design the class-balanced loss, which adds a weight related to the number of samples in the class to make the model focus on the tail class. Another effective approach is to reduce the model's excessive focus on head classes. Zhong et al. [34] use a label smoothing strategy and combines the probability distributions of different classes to mitigate the overweighting of the head classes. In this paper, we propose an adaptive re-weighting module to learn the vulnerability representation in the tail classes.

VIII. Conclusion

In this paper, we show the distribution of vulnerability types in the real world belongs to the long-tailed distribution and the importance of the tail class vulnerabilities. We propose LIVABLE, a long-tailed software vulnerability type classification approach, which involves the vulnerability representation learning module and adaptive re-weighting module. It can distinguish node representation and enhance vulnerability representations. It can also predict vulnerability types in the long-tailed distribution. Our experimental results on vulnerability type classification validate the effectiveness of LIVABLE, and the results in vulnerability detection and the ablation studies further demonstrate the advantages of LIVABLE. The future works include combining the vulnerability repair with type and generating human-readable or explainable reports due to the vulnerability type.

Data Availability

Our source code as well as experimental data are available at: https://github.com/Xin-Cheng-Wen/LIVABLE.

References

[1] "Key statistics of the Google bug bounty program." Google. Accessed: 2023. [Online]. Available: https://bughunters.google.com/about/key-stats

[2] "Common weakness enumeration." MITRE. Accessed: 2023. [Online]. Available: https://cwe.mitre.org/data/definitions/119.html

[3] "Microsoft bug bounty programs year in review: $13.6M in rewards." Microsoft. Accessed: 2021. [Online]. Available: https://msrc-blog.microsoft.com/2021/07/08/microsoft-bug-bounty-programs-year-in-review-13-6m-in-rewards/

[4] Z. Li et al., "VulDeePecker: A deep learning-based system for vulnerability detection," in *Proc. 25th Annu. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, San Diego, CA, USA. San Diego, CA, USA: Internet Society, 2018.

[5] R. L. Russell et al., "Automated vulnerability detection in source code using deep representation learning," in *Proc. 17th IEEE Int. Conf. Mach. Learn. Appl. (ICMLA)*, Orlando, FL, USA, M. A. Wani, M. M. Kantardzic, M. S. Mouchaweh, J. Gama, and E. Lughofer, Eds., Piscataway, NJ, USA: IEEE Press, 2018, pp. 757–762.

[6] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "SySeVR: A framework for using deep learning to detect software vulnerabilities," *IEEE Trans. Dependable Secure Comput.*, vol. 19, no. 4, pp. 2244–2258, Jul./Aug. 2022.

[7] Y. Zhou, S. Liu, J. K. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Proc. Adv. Neural Inf. Process. Syst. 32, (NeurIPS)*, Vancouver, BC, Canada, H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. B. Fox, and R. Garnett, Eds., 2019, pp. 10197–10207.

[8] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet?" *IEEE Trans. Softw. Eng.*, vol. 48, no. 9, pp. 3280–3296, Sep. 2022.

[9] Y. Li, S. Wang, and T. N. Nguyen, "Vulnerability detection with fine-grained interpretations," in *Proc. 29th ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE '21)*, Athens, Greece, D. Spinellis, G. Gousios, M. Chechik, and M. D. Penta, Eds. New York, NY, USA: ACM, 2021, pp. 292–303.

[10] J. L. Elman, "Finding structure in time," *Cognit. Sci.*, vol. 14, no. 2, pp. 179–211, 1990.

[11] M. Gori, G. Monfardini, and F. Scarselli, "A new model for learning in graph domains," in *Proc. IEEE Int. Joint Conf. Neural Netw.*, vol. 2, 2005, pp. 729–734.

[12] Y. Li, D. Tarlow, M. Brockschmidt, and R. S. Zemel, "Gated graph sequence neural networks," in *Proc. 4th Int. Conf. Learn. Representations (ICLR)*, 2016.

[13] "National vulnerability database." Accessed: 2023. [Online]. Available: https://nvd.nist.gov/

[14] B. Shuai, H. Li, M. Li, Q. Zhang, and C. Tang, "Automatic classification for vulnerability based on machine learning," in *Proc. IEEE Int. Conf. Inf. Automat. (ICIA)*, Yinchuan, China. Piscataway, NJ, USA: IEEE Press, 2013, pp. 312–318.

[15] S. Na, T. Kim, and H. Kim, "A study on the classification of common vulnerabilities and exposures using naïve Bayes," in *Proc. 11th Int. Conf. Broad-Band Wireless Comput. Commun. Appl.*, vol. 2, New York, NY, USA: Springer-Verlag, 2016, pp. 657–662.

[16] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, "μVulDeePecker a deep learning-based system for multiclass vulnerability detection," *IEEE Trans. Dependable Secure Comput.*, vol. 18, no. 5, pp. 2224–2236, Sep./Oct. 2021.

[17] "Common vulnerability scoring system sig." CVSS v4.0. Accessed: 2023. [Online]. Available: https://www.first.org/cvss/

[18] "Common vulnerability scoring system version 3.0." CVSS v3.1. Accessed: 2023. [Online]. Available: https://www.first.org/cvss/v3-0/

[19] "Common vulnerability scoring system version 3.0 qualitative-severity-rating-scale." CVSS v3.0. Accessed: 2023. [Online]. Available: https://www.first.org/cvss/v3.0/specification-document#Qualitative-Severity-Rating-Scale

[20] "Common weakness enumeration." MITR. Accessed: 2023. [Online]. Available: https://cwe.mitre.org/data/definitions/507.html

[21] D. Lukovnikov and A. Fischer, "Improving breadth-wise backpropagation in graph neural networks helps learning long-range dependencies," in *Proc. 38th Int. Conf. Mach. Learn. (ICML)*, Virtual Event, M. Meila and T. Zhang, Eds., vol. 139., PMLR, 2021, pp. 7180–7191.

[22] U. Alon and E. Yahav, "On the bottleneck of graph neural networks and its practical implications," in *Proc. 9th Int. Conf. Learn. Representations (ICLR),* Virtual Event, Austria, OpenReview.net, 2021.

[23] D. Guo et al., "GraphCodeBERT: Pre-training code representations with data flow," in *Proc. 9th Int. Conf. Learn. Representations (ICLR)*, Virtual Event, Austria, OpenReview.net, 2021.

[24] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng. (ICSE)*, Montreal, QC, Canada, J. M. Atlee, T. Bultan, and J. Whittle, Eds., 2019, pp. 783–794.

[25] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "A C/C++ code vulnerability dataset with code changes and CVE summaries," in *Proc. 17th Int. Conf. Mining Softw. Repositories (MSR '20)*, Seoul, Republic of Korea, S. Kim, G. Gousios, S. Nadi, and J. Hejderup, Eds., New York, NY, USA: ACM, 2020, pp. 508–512.

[26] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *Proc. 5th Int. Conf. Learn. Representations, (ICLR)*, Toulon, France, OpenReview.net, 2017.

[27] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," 2017, *arXiv:1710.10903*.

[28] K. Xu, C. Li, Y. Tian, T. Sonobe, K. Kawarabayashi, and S. Jegelka, "Representation learning on graphs with jumping knowledge networks," in *Proc. 35th Int. Conf. Mach. Learn. (ICML)*, Stockholmsmässan, Stockholm, Sweden, J. G. Dy and A. Krause, Eds., vol. 80, PMLR, 2018, pp. 5449–5458.

[29] Y. Zhang, B. Kang, B. Hooi, S. Yan, and J. Feng, "Deep long-tailed learning: A survey," *IEEE Trans. Pattern Anal. Mach. Intell.,* vol. 45, no. 9, pp. 10795–10816, Sep. 2023.

[30] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Las Vegas, NV, USA. Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 2016, pp. 2818–2826.

[31] J. Tan et al., "Equalization loss for long-tailed object recognition," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Seattle, WA, USA, 2020, pp. 11659–11668.

[32] Y. Cui, M. Jia, T. Lin, Y. Song, and S. J. Belongie, "Class-balanced loss based on effective number of samples," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Long Beach, CA, USA, Jun. 16–20, 2019, Piscataway, NJ, USA: IEEE Press, 2019, pp. 9268–9277.

[33] T. Lin, P. Goyal, R. B. Girshick, K. He, and P. Dollár, "Focal loss for dense object detection," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, Venice, Italy. Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 2017, pp. 2999–3007.

[34] Z. Zhong, J. Cui, S. Liu, and J. Jia, "Improving calibration for long-tailed recognition," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR), Virtual*, , Piscataway, NJ, USA: IEEE Press, 2021, pp. 16489–16498.

[35] Z. Zhang and M. R. Sabuncu, "Generalized cross entropy loss for training deep neural networks with noisy labels," in *Proc. Adv. Neural Inf. Process. Syst. 31, Annu. Conf. Neural Inf. Process. Syst. (NeurIPS)*, Montréal, QC, Canada, S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., 2018, pp. 8792–8802.

[36] H. Guo and S. Wang, "Long-tailed multi-label visual recognition by collaborative training on uniform and re-balanced samplings," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Virtual, Piscataway, NJ, USA: IEEE Press, 2021, pp. 15089–15098. Accessed: 2021. [Online]. Available: https://openaccess.thecvf.com/content/CVPR2021/html/Guo_Long-Tailed_Multi-Label_Visual_Recognition_by_Collaborative_Training_on_Uniform_and_CVPR_2021_paper.html

[37] K. W. Church, "Word2vec," *Natural Lang. Eng.*, vol. 23, no. 1, pp. 155–162, 2017.

[38] Y. Boureau, J. Ponce, and Y. LeCun, "A theoretical analysis of feature pooling in visual recognition," in *Proc. 27th Int. Conf. Mach. Learn. (ICML-10)*, Haifa, Israel, J. Fürnkranz and T. Joachims, Eds., Omnipress, 2010, pp. 111–118. Accessed: 2010. [Online]. Available: https://icml.cc/Conferences/2010/papers/638.pdf

[39] K. Yue, F. Xu, and J. Yu, "Shallow and wide fractional max-pooling network for image classification," *Neural Comput. Appl.*, vol. 31, no. 2, pp. 409–419, 2019.

[40] S. Woo, J. Park, J. Lee, and I. S. Kweon, "CBAM: Convolutional block attention module," in *Comput. Vis. (ECCV) 15th Eur. Conf.*, Munich, Germany, V. Ferrari, M. Hebert, C. Sminchisescu, and Y. Weiss, Eds., vol. 11211, New York, NY, USA: Springer-Verlag, 2018, pp. 3–19.

[41] P. Zhou et al., "Attention-based bidirectional long short-term memory networks for relation classification," in *Proc. 54th Annu. Meeting Assoc. Comput. Linguistics (ACL)*, Berlin, Germany, vol. 2 (Short Papers), Assoc. Comput. Linguistics, 2016, pp. 207-212.

[42] B. Zhou, Q. Cui, X. Wei, and Z. Chen, "BBN: Bilateral-branch network with cumulative learning for long-tailed visual recognition," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Seattle, WA, USA. Piscataway, NJ, USA: IEEE Press, 2020, pp. 9716–9725.

[43] "Common weakness enumeration." MITRE. Accessed: 2023. [Online]. Available: http://cwe.mitre.org/

[44] X. Wen, Y. Chen, C. Gao, H. Zhang, J. M. Zhang, and Q. Liao, "Vulnerability detection with graph simplification and enhanced graph representation learning," in *Proc. 45th IEEE/ACM Int. Conf. Softw. Eng. (ICSE)*, Melbourne, Australia. Piscataway, NJ, USA: IEEE Press, 2023, pp. 2275–2286.

[45] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Proc. IEEE Symp. Secur. Privacy,* Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 2014, pp. 590–604.

[46] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *Proc. 1st Int. Conf. Learn. Representations (ICLR)*, 2013.

[47] S. Gao et al., "Code structure-guided transformer for source code summarization," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 1, pp. 23:1–23:32, 2023.

[48] M. Lukasik, S. Bhojanapalli, A. K. Menon, and S. Kumar, "Does label smoothing mitigate label noise?" in *Proc. 37th Int. Conf. Mach. Learn. (ICML)*, Virtual Event, vol. 119, PMLR, 2020, pp. 6448–6458.

[49] R. Croft, M. A. Babar, and M. M. Kholoosi, "Data quality for software vulnerability datasets," in *Proc. IEEE/ACM 45th Int. Conf. Softw. Eng. (ICSE)*, 2023, pp. 121–133.

[50] V. der Maaten, Laurens, and H. Geoffrey, "Visualizing data using t-SNE." *J. Mach. Learn. Res.*, vol. 9, no. 11, pp. 2579–2605, 2008.

[51] "Common weakness enumeration," MITRE Corporation. Accessed: 2023. [Online]. Available: https://cwe.mitre.org/data/definitions/22.html

[52] C. Davide and J. Giuseppe, "The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation," *BMC Genomics*, vol. 21, no. 1, pp. 1–13, 2020.

[53] S. Pan, L. Bao, X. Xia, D. Lo, and S. Li, "Fine-grained commit-level vulnerability type prediction by CWE tree structure," in *Proc. 45th IEEE/ACM Int. Conf. Softw. Eng. (ICSE)*, Melbourne, Australia. Piscataway, NJ, USA: IEEE Press, 2023, pp. 957–969.

[54] Y. Chen, Z. Ding, L. Alowain, X. Chen, and D. A. Wagner, "DiverseVul: A new vulnerable source code dataset for deep learning based vulnerability detection," in *Proc. 26th Int. Symp. Res. Attacks Intrusions Defenses (RAID)*, Hong Kong, China. New York, NY, USA: ACM, 2023, pp. 654–668.

[55] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for unix processes," in *Proc. IEEE Symp. Secur. Privacy,* Oakland, CA, USA, Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 1996, pp. 120–128.

[56] F. Yamaguchi, F. F. Lindner, and K. Rieck, "Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning," in *Proc. 5th USENIX Workshop Offensive Technol. (WOOT'11)*, San Francisco, CA, USA, D. Brumley and M. Zalewski, Eds., San Francisco, CA, USA: USENIX Association, 2011, pp. 118–127.

[57] I. Santos, J. Devesa, F. Brezo, J. Nieves, and P. G. Bringas, "OPEM: A static-dynamic approach for machine-learning-based malware detection," in *Proc. Int. Joint Conf. CISIS'12-ICEUTE'12-SOCO'12 Special Sessions,* Ostrava, Czech Republic, Á. Herrero, V. Snásel, A. Abraham, I. Zelinka, B. Baruque, H. Quintián-Pardo, J. L. Calvo-Rolle, J. Sedano, and E. Corchado, Eds., vol. 189. New York, NY, USA: Springer-Verlag, 2012, pp. 271–280.

[58] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *Proc. ACM Conf. Comput. Commun. Secur. (CCS)*, Alexandria, VA, USA, P. Ning, S. D. C. di Vimercati, and P. F. Syverson, Eds., New York, NY, USA: ACM, 2007, pp. 529–540.

[59] Y. Shin, A. Meneely, L. A. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE Trans. Softw. Eng.*, vol. 37, no. 6, pp. 772–787, Nov./Dec. 2011.

[60] S. Neuhaus and T. Zimmermann, "The beauty and the beast: Vulnerabilities in red hat's packages," in *Proc. USENIX Annu. Tech. Conf.*, San Diego, CA, USA, G. M. Voelker and A. Wolman, Eds., San Diego, CA, USA: USENIX Association, 2009.

[61] G. Grieco, G. L. Grinblat, L. C. Uzal, S. Rawat, J. Feist, and L. Mounier, "Toward large-scale vulnerability discovery using machine learning," in *Proc. 6th ACM Conf. Data Appl. Secur. Privacy (CODASPY)*, New Orleans, LA, USA, E. Bertino, R. S. Sandhu, and A. Pretschner, Eds., New York, NY, USA: ACM, 2016, pp. 85–96.

[62] G. Lin, J. Zhang, W. Luo, L. Pan, and Y. Xiang, "POSTER: Vulnerability discovery with function representation learning from unlabeled projects," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, Dallas, TX, USA, B. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds., New York, NY, USA: ACM, 2017, pp. 2539–2541.

[63] J. Li, P. He, J. Zhu, and M. R. Lyu, "Software defect prediction via convolutional neural network," in *Proc. IEEE Int. Conf. Softw. Qual., Rel. Secur., (QRS)*, Prague, Czech Republic, Piscataway, NJ, USA: IEEE Press, 2017, pp. 318–328.

[64] J. Harer et al., "Learning to repair software vulnerabilities with generative adversarial networks," in *Proc. Adv. Neural Inf. Process. Syst. 31, Annu. Conf. Neural Inf. Process. Syst. (NeurIPS)*, Montréal, QC, Canada, S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., 2018, pp. 7944–7954.

[65] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proc. 38th Int. Conf. Softw. Eng. (ICSE)*, Austin, TX, USA, L. K. Dillon, W. Visser, and L. A. Williams, Eds., New York, NY, USA: ACM, 2016, pp. 297–308.

[66] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proc. 31st IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Singapore, D. Lo, S. Apel, and S. Khurshid, Eds., New York, NY, USA: ACM, 2016, pp. 87–98.

[67] S. Park, J. Lim, Y. Jeon, and J. Y. Choi, "Influence-balanced loss for imbalanced visual classification," in *Proc. IEEE/CVF Int. Conf. Comput. Vis., (ICCV)*, Montreal, QC, Canada. Piscataway, NJ, USA: IEEE Press, 2021, pp. 715–724.