Hierarchy-Aware Representation Learning for Industrial IoT Vulnerability Classification

Sicong Cao, Xiaobing Sun, Xinye Yang, Xiaoxue Wu, *Member, IEEE*, Wei Liu, and Bin Li

***Abstract*—As with anything connected to the internet, industrial Internet of Things (IIoT) devices are also subject to severe cybersecurity threats because an adversary could exploit vulnerabilities in their internal software to perform malicious attacks. Despite the promising results of deep learning-based approaches, most solutions can only detect the presence of a vulnerability but fail to pinpoint its corresponding type. Recently, TreeVul formalizes the task as a hierarchical multilabel classification problem to predict complete coarse-to-fine vulnerability type hierarchy. Yet, the TreeVul approach is still inaccurate and neglects samples labeled at coarse categories. In this article, we propose HierVul, a novel hierarchy-aware representation learning approach for IIoT vulnerability classification. Specifically, to make full use of vulnerable samples labeled at any granularity, HierVul constructs hierarchy-specific extractors as well as classifiers to disentangle level-wise vulnerability features from the code representation learning network backbone, and maximizes their marginal probability in the probability space constrained by the Common Weakness Enumeration tree hierarchy. Furthermore, considering that the distinction between two vulnerability types at the same level of abstraction becomes smaller and smaller as the refinement of classification granularity, HierVul leverages residual connections to add parent-level coarser-grained features to child-level finer-grained features to transfer hierarchical knowledge across levels. The experimental results show that HierVul achieves 15.25%, 45.16%, and 14.52% relative improvement over TreeVul on Weight F1, Macro F1, and PF, respectively, indicating the effectiveness of HierVul in the practical scenario.**

***Index Terms*—Deep learning (DL), hierarchical multilabel classification (HMC), industrial Internet of Things (IIoT), vulnerability type.**

Manuscript received 17 December 2023; revised 7 February 2024, 11 April 2024, and 19 May 2024; accepted 27 May 2024. This work was supported in part by the National Natural Science Foundation of China under Grant 62202414, in part by the Natural Science Foundation of Jiangsu Province under Grant BK20220562, in part by the Six Talent Peaks Project in Jiangsu Province under Grant RJFW-053, in part by the Jiangsu "333" Project and Yangzhou University Top-level Talents Support Program (2019), in part by the Postgraduate Research & Practice Innovation Program of Jiangsu Province under Grant KYCX22_3502, and in part by the China Scholarship Council Foundation under Grant 202308320436. Paper no. TII-23-5107.*(Corresponding author: Xiaobing Sun.)*

The authors are with the School of Information Engineering, Yangzhou University, Yangzhou 225127, China (e-mail: dx120210088@yzu.edu.cn; xbsun@yzu.edu.cn; mz120210889@stu.yzu.edu.cn; xiaoxuewu@yzu.edu.cn; weiliu@yzu.edu.cn; lb@yzu.edu.cn).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TII.2024.3413305.

Digital Object Identifier 10.1109/TII.2024.3413305

I. Introduction

INDUSTRIAL Internet of Things (IIoT) systems are improving service delivery and increasing productivity across a broad range of industries from simple applications on hand-held devices to complicated embedded systems [1]. In the meanwhile, as with anything connected to the internet, IIoT devices are also subject to security risks. One of the most prevalent attack vectors is vulnerabilities within the software deployed on IIoT devices.[1] These potential software vulnerabilities can be exploited by attackers to gain unauthorized access, manipulate device functionality, or inject malicious code into the system. What's worse, given that most IIoT devices are designed for decades of use, the older the device, the more likely it is to present a security risk as it lags in vulnerability awareness and fixing. For example, the Finite State[2] described a Huawei-manufactured device that included code using a version of OpenSSL that was released in 2003 and well known (and documented) as extremely vulnerable. Hence, automated detection of software vulnerabilities on IIoT devices has drawn increasing attention in recent years [2].

Conventional approaches primarily employ static analysis techniques or hand-crafted specifications to identify potential vulnerabilities. Recently, benefiting from the great success of deep learning (DL) in program comprehension, a number of learning-based approaches [3], [4], [5], [6], [7] have been proposed to leverage complex neural network models to learn automatically implicit patterns from prior vulnerable code instead of requiring expert involvement. They convert code snippets into abstract representations containing rich syntactic and semantic information, and employ off-the-shelf DL models to extract high-dimensional feature representations for classification.

While demonstrated superior performance, almost all of the DL-based approaches regard vulnerability detection as a binary classification task, i.e., determining whether the given source code contains vulnerabilities, but fail to point out fine-grained vulnerability types. As a matter of fact, pinpointing the vulnerability type is useful in reducing the workload of developers for vulnerability analysis, localization, and fixing to a certain extent. A straightforward way is to train a multiclass classifier based on well-labeled vulnerable samples, as μVulDeePecker [8] does. However, such a flat solution ignores

[1][Online]. Available: https://www.linkedin.com/pulse/industrial-iot-iiot-attacks-digialert

[2][Online]. Available: https://finitestate.io/

the strong correlation among vulnerability types at different levels of abstraction, and suffers from the severe class imbalance issue in practice. As reported in [9], 70% of vulnerabilities fall into 20% of types, and the top three types account for nearly 50% of vulnerabilities averagely. Such a long-tailed distribution hinders the learning process of deep learning-based models, where models could learn too well on the head vulnerability types while performing poorly on the tails. To alleviate this problem, TreeVul [10] formulated the task as a hierarchical multilabel classification (HMC) problem to infer a sequence of coarse-to-fine vulnerability types based on the Common Weakness Enumeration (CWE) tree structure information. However, conventional HMC solutions may not be applicable in vulnerability classification scenarios because manual collection and annotation of such datasets with complete hierarchical labels from the coarsest to the finest granularity (i.e., ground truth) are expensive and hard to scale. In practice, due to the uneven domain knowledge or limited vulnerability information, vulnerabilities are often labeled at different depths of the CWE tree structure. For example, according to the statistics [10], a large number of vulnerable samples are classified at a coarse-grained level (21%). As a result, examples labeled at coarse categories are often neglected, further exacerbating the scarcity of data available for training IIoT vulnerability classification model.

In this article, we propose HIERVUL, a novel **Hier**archy-aware representation learning-based approach to classify detected IIoT **Vul**nerabilities into multigranularity CWE types. The key insight underlying our proposed approach is that learning with vulnerable samples labeled at different levels of abstraction should transfer hierarchical knowledge across levels. Specifically, to make full use of vulnerable samples labeled at any granularity, HIERVUL constructs hierarchy-specific extractors as well as classifiers to disentangle level-wise vulnerability features from the code representation learning network backbone, and maximizes their marginal probability in the probability space constrained by the CWE tree hierarchy. The benefits of such marginalization are that, coarse-grained vulnerability features could impact decisions of fine-grained classifiers, while finer-grained label learning enhances the discriminability of coarser-grained classifiers. Furthermore, to reflect the feature interaction within the inherent coarse-fine hierarchical relationship among CWE vulnerability categories, HIERVUL leverages residual connections to add parent-level coarser-grained features to child-level finer-grained features to transfer hierarchical knowledge across levels.

To evaluate the effectiveness of our proposed HIERVUL, we conduct experiments on a popular real-world benchmark, Big-Vul [11], which consists of 8,783 C/C++ (a dominant language for implementing IIoT operating systems and embedded software that widely used as targets for vulnerability discovery [12] and analysis [13], [14]) vulnerable functions with CWE information. The experimental results show that HIERVUL outperforms all baselines with a substantial improvement (i.e., 54.55%–209.10% in terms of Weighted F1, 55.17%–221.43% in terms of Macro F1, and 51.06%–208.70% in terms of PF), indicating the effectiveness of HIERVUL in practical scenarios.

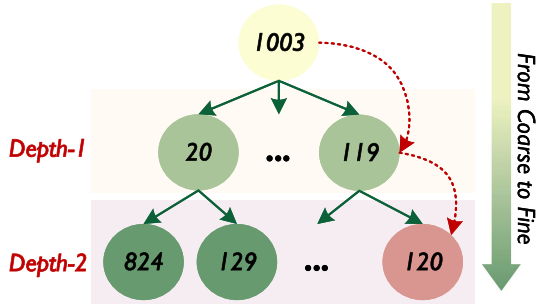


Fig. 1. Taxonomic hierarchy of the CWE tree.

In summary, this article makes the following contributions.

1) To the best of our knowledge, we are the first to promote multi-granularity IIoT vulnerability classification problem to a more practical scenario, in which samples are labeled at any level of the hierarchy.
2) We propose HIERVUL, a novel DL-based IIoT vulnerability type prediction approach, which classifies detected vulnerabilities into fine-grained CWE categories via hierarchy-aware representation learning. HIERVUL combines level-wise vulnerability feature disentanglement and residual connection to make full use of vulnerable samples labeled at any granularity.
3) Extensive experimental results show substantial improvements HIERVUL brings to IIoT vulnerability classification compared to state-of-the-art baselines.

The rest of this article is organized as follows. Section II introduces the background knowledge related to our proposed framework. Section III describes the details of our approach. Section IV presents the experimental setup and results. Section V discusses the possible threats to validity. Section VI reviews the related work. Finally, Section VII concludes this article.

II. BACKGROUND

In this section, we briefly introduce the basic concept of CWE and HMC.

A. Common Weakness Enumeration

CWE[3] provides a list of common software and hardware weakness types developed and maintained by security community. As shown in Fig. 1, vulnerability types are organized as a hierarchical tree structure of multiple levels of abstraction in CWE. Early CWE taxonomy mainly focus on software vulnerabilities, such as View-699 (Software Development), which organizes vulnerabilities around concepts that are frequently used or encountered in software development. Considering that hardware security issues are becoming increasingly important concerns for both enterprise IT, OT, and IoT in general, ranging from industrial control systems and medical devices to automobiles and wearable technologies, taxonomy for hardware vulnerabilities, i.e., View-1194 (Hardware Design), are supported by CWE since 2020.

[3][Online]. Available: https://cwe.mitre.org/index.html

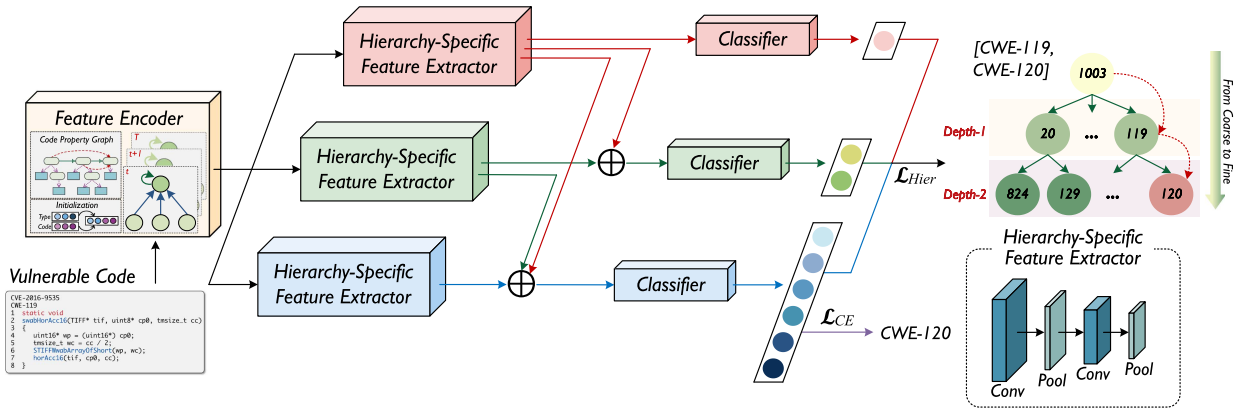


Fig. 2. Overall architecture of our proposed HIERVUL approach.

B. Hierarchical Multilabel Classification

HMC aims to predict the category of a given input (e.g., vulnerable function) in a predefined taxonomic hierarchy (e.g., tree or directed acyclic graph) from top (coarser-granularity) to down (finer-granularity). Concretely, suppose a code snippet c which is detected as vulnerable, we have a chain of $\mathcal{Y} = \{y^1, \ldots, y^K\}$, $y^k \in \{y_1^k, \ldots, y_m^k\}$ labels defined across different granularities, where m refers to the number of vulnerability types at depth-k.

Traditional HMC approach often leverages complete hierarchical labels from the coarsest to the finest granularity to construct K independent classifiers $\mathcal{G}(\cdot)$ for prediction, i.e., $\bar{\mathcal{Y}} = \{\bar{y}^1, \ldots, \bar{y}^K\}$, where $\bar{y}^k = \mathcal{G}_k(\mathcal{F}(c))$. $\mathcal{F}(\cdot)$ is an input-specific network backbone used for feature extraction. However, due to the lack of domain knowledge, vulnerabilities may be labeled at any level of the hierarchy. For example, LibTIFF,[4] an open-source Tag Image File Format (TIFF) processing library that is widely deployed on IIoT devices such as autonomous vehicles, has been exposed to have a heap buffer overflow vulnerability (CVE-2016-9535[5]) in release mode. According to the CWE taxonomy, it should have been labeled as CWE-120 (Classic Buffer Overflow), while it was labeled at its parent-level, i.e., CWE-119 (Improper Restriction of Operations within the Bounds of a Memory Buffer), by National Vulnerability Database (NVD[6]), as shown in Fig. 1. Therefore, a more practical HMC model should be able to leverage vulnerable samples labeled at various levels of the CWE tree hierarchy.

III. OUR APPROACH: HIERVUL

In this section, we present the details of our novel IIoT vulnerability type prediction approach that classifies detected vulnerabilities into hierarchical CWE vulnerability types.

A. Overview

Fig. 2 shows the overall architecture of our proposed HIERVUL approach, which consists of three main components: 1) feature encoder (Section III-B); 2) hierarchy-specific feature extractor (Section III-C); and 3) classifier (Section III-D). Specifically, given a detected vulnerable code snippet, the feature encoder transforms it into a numerical code property graph (CPG) [15] containing rich syntactic and semantic information of vulnerabilities through static analysis and program embedding, and leverages an attention-based Graph Neural Network (GAT) to iteratively update the feature embedding of each node in CPG. Then, the hierarchy-specific feature extractors are a set of CNN modules sharing the same structure and used for extract granularity-specific vulnerability features. Coarser-grained global features will be linearly added to the finer-grained local features via residual connections for information interaction across levels. Finally, to make full use of samples labeled at any level of the CWE tree hierarchy, the classifier integrates both hierarchical loss and categorical cross-entropy loss to transfer hierarchical knowledge to train the best performed classification model.

B. Feature Encoder

1) Graph Construction: Previous studies [4], [6], [7] have shown that combining diverse dimensional code representations is beneficial for DL models to capture the unique features of different types of vulnerabilities. To this end, we conduct intraprocedural program analysis to build CPG, a joint code graph representation which consists of AST, CFG, and data flow graph (DFG). AST organizes source code as a tree to reflect its syntax structure, while CFG and DFG provide the control- and data-flow information between statements. These structured code representations preserve rich syntactic and semantic information of programs beneficial to feature representation learning.

2) Code Embedding: After graph construction, we convert the code tokens of each CPG's node into low-dimensional vector representation for subsequent graph representation learning. Specifically, we apply CodeBERT [16], one of the most popular code-oriented language model that pretrained on 2.1 M bimodal comment-function pairs and 6.4 M unimodal functions across six programming languages, to capture the lexical information of vulnerable code. Code tokens (i.e., leaf nodes of ASTs) will be split and merged into meaningful subwords through

[4][Online]. Available: http://www.libtiff.org/
[5][Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2016-9535
[6][Online]. Available: https://nvd.nist.gov/

the BPE algorithm, and embedded into the fixed-size vector representations. The use of BPE subword tokenization will help reduce the vocabulary size, alleviating the problem of *Out-Of-Vocabulary* in code embedding. In addition, we also consider the abstract type of each node (e.g., *Identifier*, *Variable*) since it reflects the code property represented by each node, making the vulnerability patterns more general. We encode the abstract type of each node by one-hot encoding, which transforms text into numerical value. Finally, the textual code representation C_i is concatenated with the type representation T_i as the initial representation of each node i as follows:

$$\boldsymbol{h}_i^{(0)} = C_i || T_i \tag{1}$$

where $||$ denotes the concatenation operator.

3) Graph Representation Learning: To integrate the structured semantic information of vulnerable code into the embeddings of CPG nodes, we improve the attention-based graph neural network (hereafter GAT) to iteratively propagate and aggregate node information along with different edges (i.e., AST, CFG, and DFG edges). Formally, given a CPG node i, its node representation after tth iteration is updated as

$$\boldsymbol{h}_i^{(t+1)} = \frac{1}{|\mathcal{R}|} \sum_{r \in \mathcal{R}} \left(\sigma \left(\sum_{j \in \mathcal{N}_r} \boldsymbol{\alpha}_{i,j}^{(t)} \boldsymbol{z}_j^{(t)} \right) \right) \tag{2}$$

$$\boldsymbol{z}_j^{(t)} = \boldsymbol{W}_r^{(t)} \boldsymbol{h}_j^{(t)} \tag{3}$$

where $\mathcal{R}$ is the types of edges in CPG, and $|\cdot|$ represents the size of a set. σ denotes the activation function, which we use $LeakyReLU$ here. $\mathcal{N}_r$ represents the one-hop neighbors of node i under the edge r. $\boldsymbol{W}_r$ represents the weight matrix under the edge r. $\boldsymbol{\alpha}_{i,j}$ represents the attention weight between the node i and its neighbor j under the edge r

$$\boldsymbol{\alpha}_{i,j}^{(t)} = \frac{\exp(\boldsymbol{e}_{i,j}^{(t)})}{\sum_{q \in \mathcal{N}_r} \exp(\boldsymbol{e}_{i,q}^{(t)})} \tag{4}$$

$$\boldsymbol{e}_{i,j}^{(t)} = \sigma \left(\vec{\boldsymbol{a}_r}^{(t)^T} \left(\boldsymbol{z}_i^{(t)} || \boldsymbol{z}_j^{(t)} \right) \right) \tag{5}$$

where $\vec{\boldsymbol{a}_r}^T$ denotes the transposition of a learnable weight vector. $||$ denotes the concatenation operation. $\boldsymbol{e}_{i,j}$ can be regarded as the association degree between node i and its neighbor node j.

C. Hierarchy-Specific Feature Extractor

To capture important vulnerability features that are relevant to vulnerability types at a specific level of the CWE tree hierarchy, we construct K independent feature extractors $\mathcal{F}(\cdot)$. These feature extractors share the same structure that comprises d 1-D convolution layers with maxpooling

$$\boldsymbol{X}^{(d)} = \mathcal{F}(\boldsymbol{X}^{(d-1)}) \tag{6}$$

$$\mathcal{F}(\cdot) = MAX(ReLU(CONV(\cdot))) \tag{7}$$

where $\boldsymbol{X}^{(0)} = [\boldsymbol{h}_{n_1}^{(T)}, \boldsymbol{h}_{n_2}^{(T)}, \ldots, \boldsymbol{h}_{n_V}^{(T)}]$ is the node representation matrix of CPG after Tth iteration. The number of the convolution layer with maxpooling d is set to 2. $\boldsymbol{Z} = \boldsymbol{X}^{(d)}$ denotes the final node representation matrix which delineate important features specific to each hierarchy.

In addition, to reflect the hierarchical feature interaction, i.e., allowing coarser-grained features in jointly predicting a finer-grained label, we adopt residual connections to linearly add parent-level features to child-level features as follows:

$$\boldsymbol{Z'}_k = \boldsymbol{Z}_k + \boldsymbol{Z}_{k-1} \tag{8}$$

where $\boldsymbol{Z}_k$ represents the vulnerability features specific to CWE categories at depth-k ($k \in \{1, \ldots, K\}$) of the tree hierarchy.

Having obtained the final vulnerability features, we use a multilayer perceptron (MLP) with an average pooling layer to project the vulnerability features $\boldsymbol{Z'}$ specific to each level of the CWE tree into a latent space and aggregate them as a neural code representation $\boldsymbol{x}$ of the input code snippet c by

$$\boldsymbol{x}_c = AVG(MLP(\boldsymbol{Z'})). \tag{9}$$

D. Classifier

Given the hierarchy-specific neural code representation, we apply a classification layer to convert the predictive score into the probability distribution over multiple CWE vulnerability types. In particular, we set up two parallel output channels to compute the *Hierarchy Loss* ($\mathcal{L}_{\text{Hier}}$) and *Categorical Cross-Entropy Loss* ($\mathcal{L}_{\text{CE}}$) for model optimization. $\mathcal{L}_{\text{Hier}}$ aims to classify each code snippet c into multigranularity labels which comply with the hierarchical constraints of the CWE tree, while $\mathcal{L}_{\text{CE}}$ focuses on maximizing the separation between finest-grained types.

1) Hierarchy Loss: To effectively utilize vulnerable samples labeled at different levels of the CWE tree, we maximize the marginal probability of a labeled vulnerable sample in the probability space constrained by the CWE tree to calculate the hierarchy loss. The benefits of such marginalization is that coarse-grained vulnerability features can impact decisions of fine-grained classifiers, while finer-grained label learning enhances the discriminability of coarser-grained classifiers.

Specifically, given an input vulnerable code c, the joint probability of its hierarchical label $\boldsymbol{y}$ is computed as

$$\tilde{P}(\boldsymbol{y}|\boldsymbol{c}) = \prod_{a=1}^{n} \phi_a(\bar{c}_a, y_a) \prod_{a,b \in \{1,\ldots,n\}} \psi_{a,b}(y_a, y_b) \tag{10}$$

where $\boldsymbol{y} \in \{0,1\}^n$ is a binary vector which represents the assignment of all labels in the hierarchy. n denotes the total number of vulnerability types in the CWE tree. $\bar{c}_a = \text{sigmoid}(\boldsymbol{x}_c)$ is the probabilistic output of type a, and $\phi_a(\bar{c}_a, y_a) = e^{\bar{c}_a [y_a = 1]}$. $\psi_{a,b}(y_a, y_b)$ is the hierarchical constraint between any two labels in $\boldsymbol{y}$, which equals to 1 when y_a is the parent-level label of y_b, and vice versa. In other words, y_a and y_b should be vertically-correlated (reflecting the dependencies of vulnerability types at different levels of abstraction), and horizontally-exclusive (capturing the slight distinction between labels at the same hierarchy).

Then, if the input vulnerable code c is labeled at the ath label (i.e., $y_a = 1$) in the CWE tree hierarchy, the marginal probability

$Pr(y_a = 1|\boldsymbol{c})$ of label a is calculated by

$$Pr(y_a = 1|\boldsymbol{c}) = \frac{1}{Z(\boldsymbol{c})} \sum_{\bar{\boldsymbol{y}}:\bar{y}_a=1} \prod_a \phi_a(\bar{c}_a, \bar{y}_a) \prod_{a,b} \psi_{a,b}(\bar{y}_a, \bar{y}_b) \tag{11}$$

where $Z(\boldsymbol{c})$ is the partition function that sums over all legal assignments $\bar{\boldsymbol{y}}$.

With the hierarchy loss function, HIERVUL optimizes the classifier by maximizing the marginal likelihood of m training data $\mathcal{D} = \{c^{(l)}, y^{(l)}, g^{(l)}\}$ that labeled at any level (i.e., ground truth) of the CWE tree hierarchy as follows:

$$\mathcal{L}_{\text{Hier}}(\mathcal{D}) = -\frac{1}{m} \sum_{l}^{m} \ln(Pr(y^{(l)}_{g^{(l)}} = 1|c^{(l)})) \tag{12}$$

where $l = \{1, \ldots, m\}$ denotes the index of the training data. $y^{(l)}$ is the complete label vector and $g^{(l)} \in \{1, \ldots, n\}$ is the index of the label.

2) Categorical Cross-Entropy Loss: In addition, considering that the distinction between child-level labels belonging to the same parent becomes smaller with the refinement of classification granularity, which may seriously confuses the classifier, we also add *Categorical Cross-Entropy Loss* ($\mathcal{L}_{\text{CE}}$) to further improve the discriminative capability for the finest-grained CWE types.

Finally, the total loss used to train HIERVUL is defined as

$$\mathcal{L}_{\text{total}}(\mathcal{D}) = \sum_{l} \mathcal{L}_{\text{com}}(c^{(l)}, y^{(l)}_{g^{(l)}}) \tag{13}$$

where $\mathcal{L}_{\text{com}}$ is the combinatorial loss

$$\mathcal{L}_{\text{com}} = \begin{cases} \mathcal{L}_{\text{Hier}} + \mathcal{L}_{\text{CE}}, & \text{if } g^{(l)} \in g' \\ \mathcal{L}_{\text{Hier}}, & \text{otherwise} \end{cases} \tag{14}$$

where $g' \in \{u, \ldots, n\}$ is a set of finest-grained labels in the last level of the CWE tree.

IV. EXPERIMENTAL EVALUATION

In this section, we first introduce our research questions, dataset, baselines, evaluation metrics, and implementation details. Then, we show results for each research question.

A. Research Questions

In this article, we aim to answer the following research questions (RQs).

1) *RQ1:* How effective is HIERVUL in vulnerability classification as compared to state-of-the-art approaches?
2) *RQ2:* How does hierarchy-aware representation learning contribute to the overall performance of HIERVUL?
3) *RQ3:* Can HIERVUL be applied to vulnerability classification in real-world IIoT products?

B. Dataset

Given that IIoT-specific vulnerability samples with source code are either insufficient for training DL models [14] or lack of available vulnerability types (i.e., ground truth) [17], we employ **Big-Vul** [11], a large-scale C/C++ vulnerability dataset composed of vulnerable/benign code snippets crawled from 348 open-source GitHub projects, for our experiments. The evaluation on the Big-Vul dataset is reliable because: 1) a considerable portion of IIoT vulnerabilities are software vulnerabilities that introduced into the IIoT devices along with the integration of open-source software [18]; and 2) C/C++ is a dominant language for implementing IIoT operating systems and embedded software that widely used as targets for vulnerability discovery [12] and analysis [13].

Big-Vul adopts *Common Weakness Enumeration IDentifiers* (CWE-ID) as vulnerability classification standard to label vulnerable functions. In particular, these CWE-IDs are mapped to a vulnerability type tree based on a shallow two-level category view (View-1003[7]). Such a taxonomic hierarchy organizes vulnerability types according to the concepts commonly used or encountered in software development and are independent of any specific language or technology. While we evaluate HIERVUL on a simplified CWE tree with two layers, other views can also be done to support more complex IIoT vulnerability classification tasks. However, such taxonomy is beyond the scope of this research and we leave that for future research. In the released version of Big-Vul, there are 3754 vulnerabilities (including 11 823 vulnerable functions) spanning 91 different vulnerability types. In our experiments, we only retain vulnerable functions with CWE-IDs for model training and testing. As a result, we obtain a total of 8783 well-labeled vulnerable functions with 88 different kinds of CWE-IDs. Table I summarizes the descriptive statistics of our cleaned dataset. "Type-k" denotes the vulnerability type at depth-k of the CWE tree. "# Func" means the number of vulnerable functions in the dataset. Vulnerability families (depth-2 vulnerability types and their corresponding depth-1 parent types) types with a sample size of less than 30 (e.g., CWE-665 and its subtypes CWE-{1188,908,909}) or not mapped to the View-1003 are group together as "CWE-Other."

C. Baselines

To demonstrate the effectiveness of HIERVUL, we adopt the following two state-of-the-art DL-based vulnerability classifiers [8], [10] as our baselines.

1) **μVulDeePecker** [8] defines a set of type-specific syntax rules to extract *code attention* from *code gadget* (a group of statements which are control- or data-dependent on the actual vulnerable statements), and proposes an attention-based building-block BLSTM to fuse different levels of vulnerability features for type prediction.
2) **TreeVul** [10] proposes a hierarchical and chained classification model which leverages the hierarchy information of CWE tree-like taxonomy as prior knowledge to learn multilevel vulnerability features, and searches for the optimal label sequence based on a top-down inference algorithm. Given that TreeVul is designed for security commits, we directly feed the whole function (instead

[7][Online]. Available: https://cwe.mitre.org/data/definitions/1003.html

TABLE I
DESCRIPTIVE STATISTICS OF OUR CLEANED DATASET

Type-1	# Func	Type-2	# Func
CWE-119	2127	CWE-120 CWE-125 CWE-787 CWE-824	7 625 198 3
CWE-20	1142	CWE-129	4
CWE-200	503	CWE-209 CWE-532	1 2
CWE-287	26	CWE-290 CWE-295 CWE-522	1 5 2
CWE-269	38	–	–
CWE-362	278	CWE-367	–
CWE-400	48	CWE-770	8
CWE-404	62	CWE-763 CWE-772	2 46
CWE-672	–	CWE-415 CWE-416	81 330
CWE-682	11	CWE-190 CWE-191 CWE-369	307 3 34
CWE-706	–	CWE-22	35
CWE-732	66	CWE-281	7
CWE-74	3	CWE-77 CWE-78 CWE-79 CWE-89 CWE-94	11 11 55 5 11
CWE-754	10	CWE-252 CWE-476	1 215
CWE-834	10	CWE-835	34
CWE-Other	2415	–	–
Total	8783	–	–

of code changes) like previous Transformer-based approaches [19] into CodeBERT to support function-level vulnerability classification.

In addition, we also compare HIERVUL with the following four popular DL-based binary vulnerability detectors, including two sequence-based approaches [3], [5] and two graph-based approaches [4], [6].

1) **VulDeePecker** [3] extracts program slices based on data-flows between statements and leverages BLSTM to detect buffer error vulnerabilities (CWE-119) and resource management error vulnerabilities (CWE-399).
2) **SySeVR** [5] improves VulDeePecker by performing forward and backward program slicing on PDG to extract control- and data-flow-related code snippets as features and adopts several RNN-based models for training (BLSTM, BGRU, GRU, etc.).
3) **Devign** [4] combines multiple code representations (e.g., AST, CFG, and DDG) to model programs at the function-level, and adopts GGNN to learn implicit vulnerability semantics for classification.
4) **ReVeal** [6] proposes to leverage CPG and GGNN to automatically learn the graph properties of source code.

These baselines have been widely evaluated on the Big-Vul dataset by previous works [19], [20]. Their detailed comparison is shown in Table II. In addition, considering that the above four DL-based binary vulnerability detectors are not designed for multiclass classification scenarios, we make the minimal modifications to them by replacing the binary cross-entropy (CE) loss with categorical CE loss, and employs a *softmax* layer for multiclass prediction. For fair comparison, we retrain the four modified baselines on the Big-Vul dataset to focus on vulnerability classification.

D. Evaluation Metrics

Referring to previous works [8], [10], we used the following evaluation metrics to measure the performance of classifying detected vulnerabilities into different CWE categories.

1) ***Macro F1*** is the harmonic mean of ***Macro Recall*** (M_R) and ***Macro Precision*** (M_P), which reflects the global performance of the classifier over all vulnerability types. It is calculated as: $Macro\ F1 = 2 * \frac{M\_R * M\_P}{M\_R + M\_P}$, where $M\_R = \frac{1}{L}\sum_{l=1}^{L} Recall_l = \frac{1}{L}\sum_{l=1}^{L} \frac{TP_l}{TP_l + FN_l}$, and $M\_P = \frac{1}{L}\sum_{l=1}^{L} Precision_l = \frac{1}{L}\sum_{l=1}^{L} \frac{TP_l}{TP_l + FP_l}$. L indicates the total number of vulnerability types. TP_l, FP_l, and FN_l are the numbers of true positives, false positives, and false negatives for class l. The higher (ranging from 0 to 1) the Macro F1 is, the better the classifier is at multiclass classification.
2) ***Weighted F1*** represents the average F1-score of all classes weighted by the proportion of the number of each vulnerability class in the total number of vulnerabilities. It is defined as: $Weighted\ F1 = \sum_{l=1}^{L} W_l * \frac{2 * \text{Recall}_l * \text{Precision}_l}{\text{Recall}_l + \text{Precision}_l}$, where $W_l = \frac{|X_l|}{\sum_{l=1}^{L} |X_l|}$ is the weight factor of each vulnerability class. $|X_l|$ indicates the total number of samples for vulnerability type l. $|\cdot|$ denotes the size of a set.
3) ***Path Fraction*(PF)** measures the degree of overlap between the predicted label sequence and the observed CWE type. It is defined as: $PF = \frac{1}{N}\sum_{j=1}^{N} |\widetilde{l_j} \cap l_j| / |l_j|$, where $N = \sum_{l=1}^{L} |X_l|$ is the total number of samples in the dataset. $\widetilde{l_j}$ represents the predicted hierarchical label sequence of CWE categories, while l_j is the ground truth label.

These metrics are commonly used in the literature regarding vulnerability classification [8], [10], and suitable for our data where the vulnerability distribution is highly imbalanced [21].

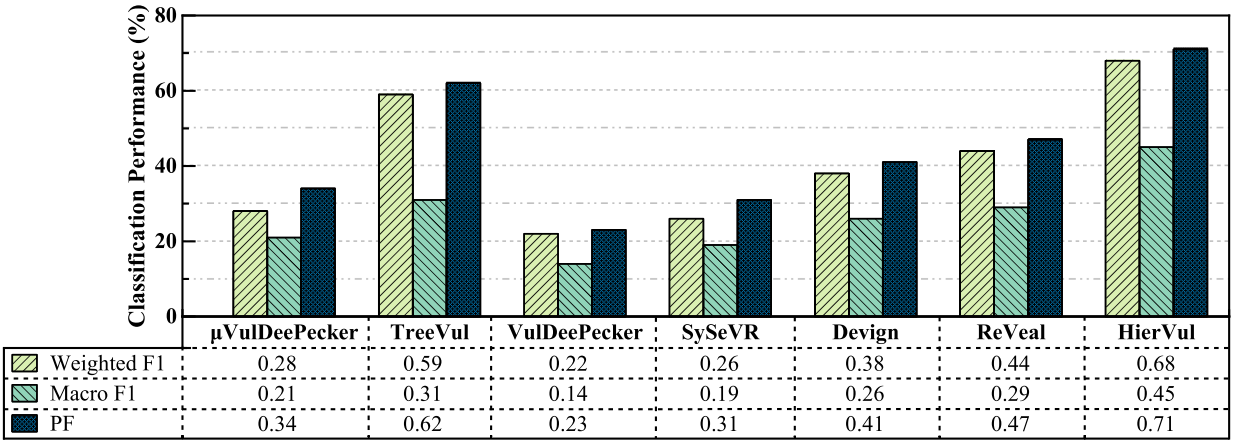
E. Implementation Details

Our experiments were performed on a computer with an Nvidia Graphics Tesla T4 GPU, installed with Ubuntu 18.04, CUDA 10.1. The cleaned dataset was randomly split into the ratio of 8:1:1 for training, validation, and testing. We implemented our approach in Python using PyTorch.[8] We extracted CFGs and PDGs of the code snippets based on the ASTs parsed by `tree-sitter`[9] to construct CPG. The dimension of the vector

[8][Online]. Available: https://pytorch.org/
[9][Online]. Available: https://tree-sitter.github.io/tree-sitter/

TABLE II
DETAILS OF DL-BASED VULNERABILITY CLASSIFICATION BASELINES

Approach	Category	Feature encoder		Model	Feature extractor	# Classifiers
		Input type	Node embedding			
μVulDeePecker	Flatten	Sequence	Word2Vec	BLSTM	–	Single
TreeVul	Hierarchical	Sequence	CodeBERT	BLSTM	–	Multiple
VulDeePecker	Flatten	Sequence	Word2Vec	BLSTM	–	Single
SySeVR	Flatten	Sequence	Word2Vec	BGRU	–	Single
Devign	Flatten	Graph	Word2Vec+Node Type	GGNN	CNN+MLP	Single
ReVeal	Flatten	Graph	Word2Vec+Node Type	GGNN	MLP	Single
HIERVUL	Hierarchical	Graph	CodeBERT+Node Type	GAT	CNN+MLP	Multiple



	µVulDeePecker	TreeVul	VulDeePecker	SySeVR	Devign	ReVeal	HierVul
Weighted F1	0.28	0.59	0.22	0.26	0.38	0.44	0.68
Macro F1	0.21	0.31	0.14	0.19	0.26	0.29	0.45
PF	0.34	0.62	0.23	0.31	0.41	0.47	0.71

Fig. 3. Performance of vulnerability classification regarding HIERVUL and baselines.

representation of each node/token in CPG is set to 128 and the dropout is set to 0.1. GGNN is trained in a batch-wise fashion until converging and the batch size is set to 64. ADAM [22] optimization algorithm is used to train the model with the learning rate of $1e$-4. Weight decay is set to $5e$-1. The other hyperparameters of our approach are tuned through grid search.

F. Effectiveness (RQ1)

To answer RQ1, we compare HIERVUL with two multiclass vulnerability classifiers and four modified DL-based vulnerability detectors on the Big-Vul dataset. Fig. 3 shows the performance comparison of HIERVUL with respect to six baselines in terms of the aforementioned evaluation metrics. Overall, HIERVUL generally outperforms all of the baselines, achieving 0.68 on Weighted F1, 0.45 on Macro F1, and 0.71 on PF.

Compared with DL-based approaches (i.e., μVulDeePecker, VulDeePecker, SySeVR, Devign, and ReVeal) which directly regard vulnerability type prediction as a multiclass classification problem, we can find that the average improvements of HIERVUL over each metric are significant, ranging from 54.55% to 209.10% on Weighted F1, from 55.17% to 221.43% on Macro F1, and from 51.06% to 208.70% on PF, respectively. These results verify the effectiveness of hierarchical classification strategy in predicting vulnerability types which are organized as a multilevel tree structure in CWE. The root cause for this performance gap is that such a flat solution ignores the strong correlation among vulnerability types at different levels of abstraction, and suffers from the severe class imbalance issue in practice. As a result, vulnerability types with similar features confuse the classifier seriously. By contrast, benefiting from the inherent coarse-fine hierarchical relationship among CWE vulnerability categories, HIERVUL can transfer hierarchical knowledge across levels and improve the discriminative capability among similar vulnerability types. In addition, HIERVUL also outperforms the most related baseline TreeVul on all metrics. In particular, HIERVUL achieves 15.25%, 45.16%, and 14.52% relative improvement on Weight F1, Macro F1, and PF, respectively. The key reason for HIERVUL to predict vulnerability types more correctly than TreeVul is that our hierarchy-aware representation learning can make full use of vulnerable samples labeled at any granularity by combining the hierarchy loss with categorical cross-entropy loss. On the contrary, TreeVul requires complete hierarchical labels from the coarsest to the finest granularity to construct multiple hierarchy-specific classifiers, further exacerbating the scarcity of data available for model training and overlooking samples labeled as coarse categories. As shown in Fig. 4, TreeVul behaves poorly on some fine-grained vulnerability types, such as CWE-129 and CWE-700, while our proposed approach achieves better performance on both coarse- and fine-grained vulnerability types.

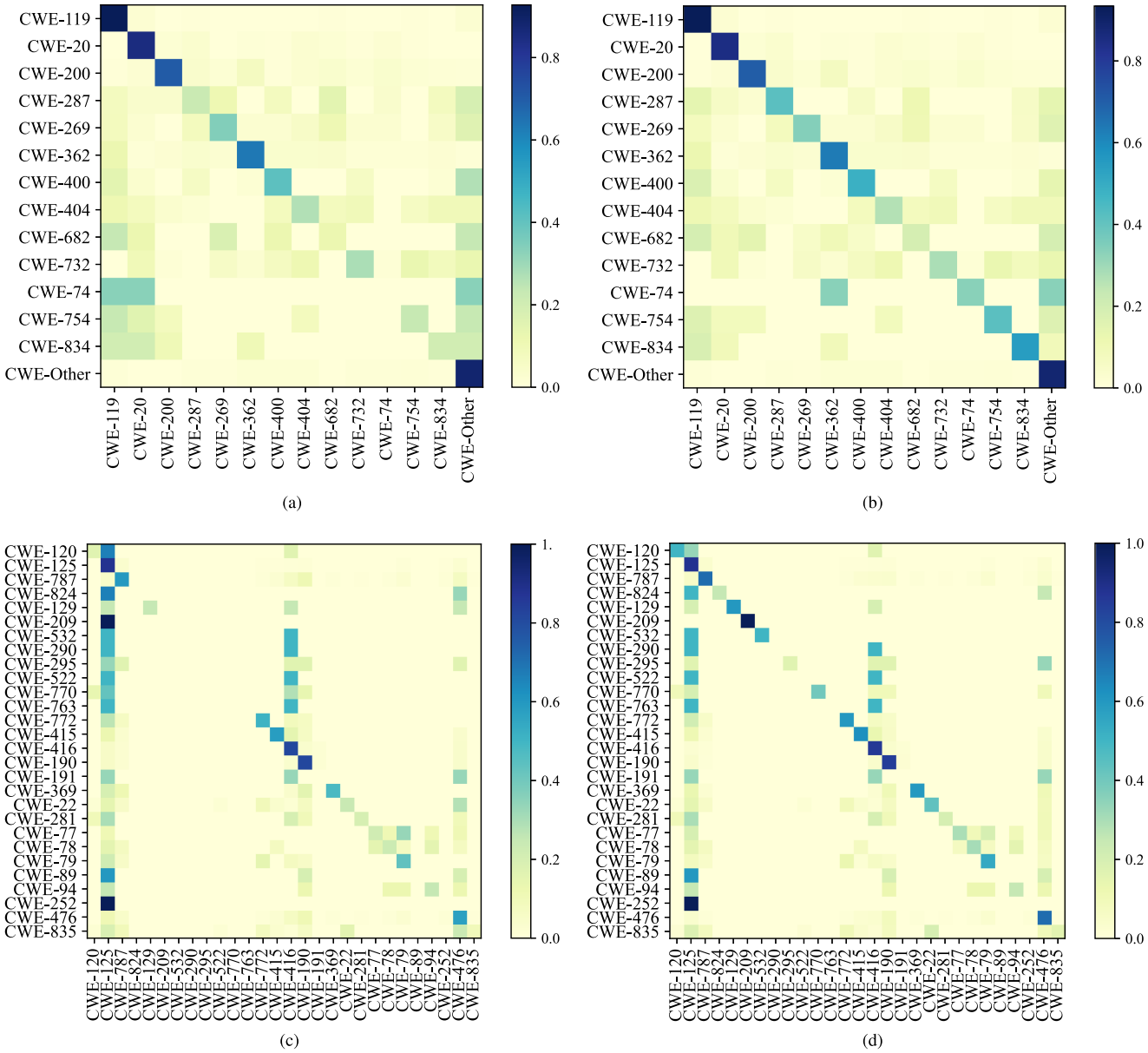


Fig. 4. Confusion matrices of TreeVul and HierVul at different levels of the CWE tree. (a) Depth-1 (TreeVul); (b) Depth-1 (HierVul); (c) Depth-2 (TreeVul); (d) Depth-2 (HierVul).

G. Ablation Study (RQ2)

To answer this RQ, we build three variants ($\mathcal{L}_{\text{CE}}$, $\mathcal{L}_{\text{Hier}}$, and $\mathcal{L}_{\text{Hier}}$ + $\mathcal{L}_{\text{CE}}$) of HierVul by removing each key element of our hierarchy-aware representation learning, including residual connections (RC) and combinatorial loss ($\mathcal{L}_{\text{com}}$), from the model one by one.

As shown in Table III, both residual connections and combinatorial loss significantly improve the performance of HierVul on each evaluation metric, indicating that our hierarchy-aware representation learning can promote the discriminative feature extraction and benefits the performance of vulnerability classification. Furthermore, we can find that compared to residual connections, the positive gains of introducing hierarchy loss are major. In particular, the Weighted F1, Macro F1, and PF are improved by 38.30%, 53.57%, and 31.37%, respectively. Such a significant improvement is foreseeable because the inherent

TABLE III
Impact of Two Key Designs of Our Proposed Hierarchy-Aware Representation Learning

Model	Weighted F1	Macro F1	PF
$\mathcal{L}_{\text{CE}}$	0.47	0.28	0.51
$\mathcal{L}_{\text{Hier}}$	0.58	0.36	0.54
$\mathcal{L}_{\text{Hier}}$ + $\mathcal{L}_{\text{CE}}$	0.65	0.43	0.67
$\mathcal{L}_{\text{Hier}}$ + $\mathcal{L}_{\text{CE}}$ + RC	**0.68**	**0.45**	**0.71**

The best result for each metric is highlighted in bold.

TABLE IV
VULNERABILITIES CLASSIFICATION PERFORMANCE OF TREEVUL AND HIERVUL IN REAL-WORLD IIOT PRODUCTS

Product	CVE-ID	CWE-ID	TreeVul	HIERVUL
LibTIFF	CVE-2010-2067	CWE-119	✓	✓
	CVE-2010-2483	CWE-119	✗	✓
	CVE-2010-4655	CWE-665	✗	✗
	CVE-2011-1167	CWE-119	✓	✓
	CVE-2012-1173	CWE-189	✗	✗
	CVE-2013-4231	CWE-119	✓	✓
	CVE-2013-4243	CWE-119	✓	✓
	CVE-2014-8129	CWE-787	✗	✓
	CVE-2014-8130	CWE-369	✗	✓
	CVE-2015-8668	CWE-119	✓	✓
	CVE-2015-8784	CWE-787	✗	✓
	CVE-2015-8870	CWE-190	✓	✓
	CVE-2016-3186	CWE-119	✓	✓
	CVE-2016-3619	CWE-125	✗	✓
	CVE-2016-3631	CWE-125	✓	✗
	CVE-2016-3658	CWE-125	✗	✓
	CVE-2017-10688	CWE-20	✗	✓
	CVE-2017-13727	CWE-617	✗	✗
	CVE-2018-15209	CWE-787	✗	✓
	CVE-2018-16335	CWE-787	✓	✓
	CVE-2018-19210	CWE-476	✓	✗
	CVE-2020-19131	CWE-787	✓	✓
	CVE-2022-2057	CWE-369	✗	✓
	CVE-2022-0562	CWE-476	✗	✓
VLC	CVE-2007-0017	CWE-134	✗	✗
	CVE-2008-0984	CWE-399	✓	✗
	CVE-2008-5276	CWE-189	✗	✗
	CVE-2009-2484	CWE-119	✓	✓
	CVE-2010-2937	CWE-20	✗	✓
	CVE-2011-2587	CWE-119	✓	✓
	CVE-2012-0023	CWE-399	✗	✗
	CVE-2013-4388	CWE-119	✓	✓
	CVE-2014-9598	CWE-20	✗	✗
	CVE-2015-5949	CWE-119	✓	✓
	CVE-2016-5108	CWE-119	✗	✓
	CVE-2018-19857	CWE-824	✓	✗
	CVE-2021-25801	CWE-125	✗	✓
Macro Precision			0.26	0.41
Macro Recall			0.34	0.46

coarse-fine hierarchical relationship among CWE vulnerability categories is ignored. As a result, the classifier struggles to distinguish a vulnerability type from its parent-level or child-level types sharing the similar features. In addition, the combinatorial loss consistently outperforms single $\mathcal{L}_{\text{Hier}}$ by adding $\mathcal{L}_{\text{CE}}$ imposed on the finest-grained vulnerability types, demonstrating the effectiveness of combining the probabilistic classification loss ($\mathcal{L}_{\text{Hier}}$) with the categorical cross-entropy loss ($\mathcal{L}_{\text{CE}}$).

H. Practicability (RQ3)

To answer this RQ, we perform a case study to investigate the practicability of HIERVUL on real-world IIoT vulnerability classification. We select two software, LibTIFF and VLC, as our targets because both of them are widely deployed in IIoT environments, such as smart healthcare [23] and grid [24], and open-sourced. We manually identify vulnerable code of these projects from NVD and filter out entries without explicit CWE-IDs. In total, we collect 37 vulnerabilities with CWE type information.

Table IV presents the effectiveness of the best-performing baseline TreeVul and HIERVUL in classifying real-world IIoT vulnerabilities. As seen, HIERVUL correctly classifies 26 of 37 (i.e., 0.70 in terms of Accuracy, 0.41 in terms of Macro Precision, and 0.46 in terms of Macro Recall) vulnerabilities as their corresponding CWE types, while TreeVul can only successfully classify 17 vulnerabilities (i.e., 0.46 in terms of Accuracy, 0.26 in terms of Macro Precision, and 0.34 in terms of Macro Recall). In addition, 13 of 26 vulnerabilities correctly classified by HIERVUL are missed by TreeVul. By contrast, HIERVUL only mislabels four vulnerabilities that can be found by TreeVul. Such results demonstrate the capability of HIERVUL in real-world IIoT vulnerability classification.

V. THREATS TO VALIDITY

Threats to Internal Validity come from the quality of our used Big-Vul dataset. The Big-Vul dataset gathers the type label (i.e., CWE-ID) of each vulnerable sample from the public CVE database. However, the vulnerability types provided by CVE have reported to exhibit quality issues such as erroneous and imprecise [9]. To reduce the likelihood of experiment biases, we employ two security experts with at least five years of experience in software security to manually confirm the correctness of vulnerability types.

Threats to External Validity refer to the generalizability of our approach. We only conduct our experiments on C/C++ projects, and thus our experimental results may not generalizable to IIoT projects developed by other programming languages such as Java and Python. To mitigate the threat, we employ `tree-sitter`, which supports a wide range of languages, to implement HIERVUL and baselines.

VI. RELATED WORK

The related prior works can be classified into two categories: 1) vulnerability detection; and 2) vulnerability classification.

A. Vulnerability Detection

The major breakthroughs in DL models along with the ever-increasing public datasets has opened up new opportunities to develop effective vulnerability detection techniques. Prior works [3], [5] focus on representing source code as sequences and use LSTM-like models to learn the syntactic and semantic information of vulnerabilities. Recently, a large number of works [4], [6], [20], [25], [26] turn to leveraging GNNs to extract rich and well-defined semantics of the program structure from graph representations for downstream vulnerability detection tasks. AMPLE [20] simplified the code graphs to alleviate the long-term dependency problems and fused local and global heterogeneous node relations for better representation learning.

Instead of exploring the best-performing DL models, we focus on classifying the detected vulnerabilities into fine-grained CWE types to facilitate vulnerability understanding and analysis. Thus, existing DL-based vulnerability detection approaches are orthogonal to our work.

B. Vulnerability Classification

Vulnerability classification is the first and a crucial step in vulnerability analysis and repair. Prior works [27], [28] focus

on constructing classifiers based on experts-curated vulnerability descriptions. Wang et al. [29] generated weighted word embeddings according to the category distribution, and fused the local and global features of vulnerability descriptions extracted by the TCNN-BiGRU model. Although text-based classification models have achieved relatively promising results, high-quality vulnerability descriptions are often hard to obtain in early vulnerability sensing. Thus, recent efforts [8], [10], [30] try to perform vulnerability classification after receiving an alert of a vulnerable code. Zhou et al. [30] proposed CoLeFunDa, a contrastive learning-based pretraining framework, which identifies silent vulnerability fixes and further provides OSS users with the CWE category and the exploitability rating information for explanation.

Different from the existing studies targeting at flat multiclass vulnerability type prediction, we leverage the CWE tree structure to perform hierarchical classification in which different vulnerabilities can be labeled at various levels of the label hierarchy due to the differences in domain knowledge.

VII. Conclusion

In this article, we proposed HierVul, a novel hierarchy-aware representation learning approach for IIoT vulnerability classification. The key insight of HierVul was that coarse-grained vulnerability features could impact decisions of fine-grained classifiers, while finer-grained label learning enhances the discriminability of coarser-grained classifiers. Based on this intuition, HierVul disentangled level-wise vulnerability features and maximized the marginal probability in the probability space constrained by the CWE tree hierarchy to make full use of vulnerable samples labeled at any granularity. The experimental results showed that HierVul significantly outperforms the state-of-the-art baselines in terms of all metrics. In the near future, we plan to generalize our approach to more programming languages like Java and Python, and investigate the explainability of our proposed approach.

References

[1] A. Gilchrist, *Industry 4.0: The Industrial Internet of Things.*, Berlin, Germany: Springer, 2016.

[2] X. Jiang, M. Lora, and S. Chattopadhyay, "An experimental analysis of security vulnerabilities in industrial IoT devices," *ACM Trans. Internet Techn.*, vol. 20, no. 2, pp. 16:1–16:24, 2020.

[3] Z. Li et al., "VulDeePecker: A deep learning-based system for vulnerability detection," in *Proc. 25th Annu. Netw. Distrib. Syst. Secur. Symp. (NDSS). Internet Soc.*, 2018, pp. 1–15.

[4] 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. 33 rd Annu. Conf. Neural Inf. Process. Syst. (NeurIPS)*, 2019, pp. 10197–10207.

[5] 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 Secur. Comput.*, vol. 19, no. 4, pp. 2244–2258, Jul.–Aug. 2022.

[6] 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.

[7] S. Cao, X. Sun, L. Bo, Y. Wei, and B. Li, "BGNN4VD: Constructing bidirectional graph neural-network for vulnerability detection," *Inf. Softw. Technol.*, vol. 136, 2021, Art. no. 106576.

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

[9] B. Liu et al., "A large-scale empirical study on vulnerability distribution within projects and the lessons learned," in *Proc. 42nd Int. Conf. Softw. Eng.*, 2020, pp. 1547–1559.

[10] S. Pan, L. Bao, X. Xia, D. Lo, and S. Li, "Fine-grained commit-level vulnerability type prediction by CWE tree structure," in *Proc. IEEE/ACM 45th Int. Conf. Softw. Eng.*, 2023, pp. 957–969.

[11] 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*, 2020, pp. 508–512.

[12] F. Xiao, L. Sha, Z. Yuan, and R. Wang, "VulHunter: A discovery for unknown bugs based on analysis for known patches in industry Internet of Things," *IEEE Trans. Emerg. Top. Comput.*, vol. 8, no. 2, pp. 267–279, Apr.-Jun. 2020.

[13] V. Lesi, Z. Jakovljevic, and M. Pajic, "Security analysis for distributed IoT-based industrial automation," *IEEE Trans Autom. Sci. Eng.*, vol. 19, no. 4, pp. 3093–3108, Oct. 2022.

[14] S. Figueroa, J. Añorga, and S. Arrizabalaga, "A survey of IIoT protocols: A measure of vulnerability risk analysis based on CVSS," *ACM Comput. Surv.*, vol. 53, no. 2, pp. 44:1–44:53, 2021.

[15] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Proc. IEEE 35th Symp. Secur. Privacy*, 2014, pp. 590–604.

[16] Z. Feng et al., "CodeBERT: A pre-trained model for programming and natural languages," in *Proc. Findings Assoc. Comput. Linguistics*, 2020, pp. 1536–1547.

[17] S. Liu, M. Dibaei, Y. Tai, C. Chen, J. Zhang, and Y. Xiang, "Cyber vulnerability intelligence for Internet of Things binary," *IEEE Trans. Ind. Informat.*, vol. 16, no. 3, pp. 2154–2163, Mar. 2020.

[18] R. A. Martin, "Visibility & control: Addressing supply chain challenges to trustworthy software-enabled things," in *Proc. IEEE Syst. Secur. Symp.*, 2020, pp. 1–4.

[19] M. Fu and C. Tantithamthavorn, "LineVul: A transformer-based line-level vulnerability prediction," in *Proc. IEEE/ACM 19th Int. Conf. Mining Softw. Repositories*, 2022, pp. 608–620.

[20] 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. IEEE/ACM 45th Int. Conf. Softw. Eng.*, 2023, pp. 2275–2286.

[21] A. Luque, A. Carrasco, A. Martńn, and A. de las Heras, "The impact of class imbalance in classification performance metrics based on the binary confusion matrix," *Pattern Recognit.*, vol. 91, pp. 216–231, 2019.

[22] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. 3 rd Int. Conf. Learn. Representations*, 2015, pp. 1–15.

[23] X. Yi, J. Wu, G. Li, A. K. Bashir, J. Li, and A. A. AlZubi, "Recurrent semantic learning-driven fast binary vulnerability detection in healthcare cyber physical systems," *IEEE Trans. Netw. Sci. Eng.*, vol. 10, no. 5, pp. 2537–2550, Sep.-Oct. 2023.

[24] B. P. Singh and M. Gore, "Smart DC microgrid: A cyber-physical system perspective," in *Technological Developments in Industry 4.0 for Business Applications*. Pennsylvania, USA: IGI Global, 2019, pp. 100–128.

[25] J. Gao, X. Yang, Y. Jiang, H. Song, K. R. Choo, and J. Sun, "Semantic learning based cross-platform binary vulnerability search for IoT devices," *IEEE Trans. Ind. Informat.*, vol. 17, no. 2, pp. 971–979, Feb. 2021.

[26] S. Cao, X. Sun, L. Bo, R. Wu, B. Li, and C. Tao, "MVD: Memory-related vulnerability detection based on flow-sensitive graph neural networks," in *Proc. IEEE/ACM 44th Int. Conf. Softw. Eng.*, 2022, pp. 1456–1468.

[27] Z. Han, X. Li, H. Liu, Z. Xing, and Z. Feng, "DeepWeak: Reasoning common software weaknesses via knowledge graph embedding," in *Proc. IEEE 25th Int. Conf. Softw. Anal., Evol. Reeng.*, 2018, pp. 456–466.

[28] Y. Dong, Y. Tang, X. Cheng, and Y. Yang, "Dekedver: A deep learning-based multi-type software vulnerability classification framework using vulnerability description and source code," *Inf. Softw. Technol.*, vol. 163, 2023, Art. no. 107290.

[29] Q. Wang, Y. Gao, J. Ren, and B. Zhang, "An automatic classification algorithm for software vulnerability based on weighted word vector and fusion neural network," *Comput. Secur.*, vol. 126, 2023, Art. no. 103070.

[30] J. Zhou et al., "CoLeFunDa: Explainable silent vulnerability fix identification," in *Proc. IEEE/ACM 45th Int. Conf. Softw. Eng.*, 2023, pp. 2565–2577.

Sicong Cao received the B.S. degree in software engineering from the Nanjing Institute of Technology, Nanjing, China, in 2019. He is currently working toward the Ph.D. degree in software engineering with the School of Information Engineering, Yangzhou University, Yangzhou, China.

He has authored or coauthored ten publications in top-tier conferences and journals like International Conference on Software Engineering, IEEE Symposium on Security and Privacy, *ACM Transactions on Software Engineering*, and *IEEE Internet of Things Journal*. His research interests include software security and deep learning.

Xiaobing Sun received the B.E. degree in computer science and technology from the Jiangsu University of Science and Technology, Zhenjiang, China, in 2007, and the Ph.D. degree in computer science and technology from the School of Computer Science and Engineering, Southeast University, Nanjing, China, in 2012.

He is currently a Professor with the School of Information Engineering, Yangzhou University, Yangzhou, China. He has authored and coauthored more than 80 papers in referred international journals and conferences, and has authorized more than 20 patents. His research interests include software maintenance and evolution, software repository mining, and intelligence analysis.

Xinye Yang received the B.E. degree in computer science and technology, in 2021, from Yangzhou University, Yangzhou, China, where he is currently working toward the M.Sc. degree in computer technique with the School of Information Engineering.

His research interests include software security and deep learning.

Xiaoxue Wu (Member, IEEE) received the Ph.D. degree in computer science and technology from Northwestern Polytechnical University, Xi'an, China, in 2021.

She is currently a Lecturer with the School of Information Engineering, Yangzhou University, Yangzhou, China. Her research interests include software testing and software security.

Wei Liu received the B.Sc. and M.Sc. degrees in computer science from Yangzhou University, Yangzhou, China, in 2004 and 2007, respectively, and the Ph.D. degree in computer science and technology from the Department of Computer Science, Nanjing University of Aeronautics and Astronautics, Nanjing, China, in 2010.

She is currently a Professor with the School of Information Engineering, Yangzhou University, Yangzhou, China. Her research interests include complex network and machine learning.

Bin Li received the B.S. degree in computer software from Fudan University, Shanghai, China, in 1986, and the M.S. and Ph.D. degrees in computer application technology from the Nanjing University of Aeronautics and Astronautics, Nanjing, China, in 1993 and 2001, respectively.

He is currently a Professor with Yangzhou University, Yangzhou, China. He has authored or coauthored more than 100 journal and conference papers. His main research interests include knowledge graph, software data mining, and multiagent system.