EXVUL: Toward Effective and Explainable Vulnerability Detection for IoT Devices

Sicong Cao, Xiaobing Sun, *Member, IEEE*, Wei Liu, *Member, IEEE*, Di Wu, *Member, IEEE*, Jiale Zhang, *Member, IEEE*, Yan Li, *Senior Member, IEEE*, Tom H. Luan, *Senior Member, IEEE*, and Longxiang Gao, *Senior Member, IEEE*

***Abstract*—As with anything connected to the Internet, Internet of Things (IoT) 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 (DL)-based approaches, the lack of well-labeled IoT vulnerability samples available for training and explainability pose a critical challenge to deploy them in practice. In this article, we propose EXVUL, a novel DL-based approach for Effective and eXplainable IoT VULnerability detection. Specifically, inspired by recent advances of self-supervised learning in label-expensive tasks, we propose a new combinatorial contrastive loss to combine the strengths of large-scale unlabeled code corpus and limited IoT vulnerability samples. Then, given a binary detection result, EXVUL provides a set of faithful and stable code statements positively contributing to the model's predictions as understandable explanations. Experimental results indicate that EXVUL outperforms state-of-the-art baselines by 33.44%-72.91% and 19.52%-98.78% with respect to the accuracy and F1 score metrics, respectively. For vulnerability explanation, EXVUL improves over the best-performing baseline explainer PGExplainer by 22.97% in mean statement precision, 49.55% in mean statement recall, and 48.40% in mean intersection over union, demonstrating that the explanations provided by EXVUL can correctly point out the vulnerable statements relevant to the detected vulnerabilities.**

Manuscript received 30 November 2023; revised 1 March 2024; accepted 19 March 2024. Date of publication 27 March 2024; date of current version 7 June 2024. This work was supported in part by the National Natural Science Foundation of China under Grant 62206238; 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 China Postdoctoral Science Foundation under Grant 2023M732985; in part by the State Key Laboratory of Massive Personalized Customization System and Technology under Grant H&C-MPC-2023-02-05; in part by the Postgraduate Research and Practice Innovation Program of Jiangsu Province under Grant KYCX22_3502; and in part by the China Scholarship Council Foundation under Grant 202308320436. *(Corresponding author: Xiaobing Sun.)*

Sicong Cao, Xiaobing Sun, Wei Liu, and Jiale Zhang are with the School of Information Engineering, Yangzhou University, Yangzhou 225127, China (e-mail: dx120210088@yzu.edu.cn; xbsun@yzu.edu.cn; weiliu@yzu.edu.cn; jialezhang@yzu.edu.cn).

Di Wu and Yan Li are with the School of Mathematics, Physics and Computing, University of Southern Queensland, Toowoomba, QLD 4350, Australia (e-mail: di.wu@unisq.edu.au; yan.li@unisq.edu.au).

Tom H. Luan is with the School of Cyber Engineering, Xidian University, Xi'an 710126, China (e-mail: tom.luan@xidian.edu.cn).

Longxiang Gao is with the Key Laboratory of Computing Power Network and Information Security, Ministry of Education, Shandong Computer Science Center, Qilu University of Technology (Shandong Academy of Sciences), Jinan 250353, China, also with the Shandong Provincial Key Laboratory of Computer Networks, Shandong Fundamental Research Center for Computer Science, Jinan 250014, China, and also with the School of Mathematics, Physics and Computing, University of Southern Queensland, Toowoomba, QLD 4350, Australia (e-mail: gaolx@sdas.org).

Digital Object Identifier 10.1109/JIOT.2024.3381641

***Index Terms*—Contrastive learning (CL), explainability, Internet of Things (IoT), stability.**

I. INTRODUCTION

THE Internet of Things (IoT) landscape and its smart devices are spreading in all aspects of our society, such as autonomous vehicles and smart grids, improving service delivery and increasing productivity. As reported in [1], the global number of connected IoT devices is expected to grow by 16%, to 16.7 billion active endpoints by 2023. In the meanwhile, as with anything connected to the Internet, IoT devices are also subject to security threats, leading to severe economic loss and equipment damage. One of the most common IoT security threats is software vulnerabilities [2], which arise from bugs in code as well as from insecure system settings that can be exploited by threat actors for a variety of malicious ends.

Benefiting from the great success of deep learning (DL) on IoT security [3], an increasing number of learning-based vulnerability detection approaches [4], [5], [6], [7] have been proposed. Compared to conventional approaches [8], [9], [10] that heavily rely on hand-crafted vulnerability specifications, DL-based approaches focus on constructing complex neural network (NN) models to automatically learn implicit vulnerability patterns from source code without human intervention. Recently, inspired by the ability to effectively capture structured semantic information (e.g., control- and data-flows) of source code, graph NNs (GNNs) have been widely adopted by state-of-the-art neural vulnerability detectors [11], [12], [13], [14].

While demonstrated superior performance, these approaches face two challenges that limit their potential when applied to detecting vulnerabilities on IoT devices:

1) *Insufficient Labeled Data Set:* Almost all DL-based vulnerability detection approaches follow the supervised learning paradigm, i.e., training a best-performing detection model over a well-labeled vulnerability datasets. However, collecting such a large-scale dataset with human annotations for software vulnerabilities in practice is *time-consuming* and *error-prone*, let alone for IoT vulnerabilities. For example, one of the most popular benchmarks, `FFmpeg+QEMU` [12], which contains +22K functions with 45.66% of the vulnerable ones, was manually labeled by four professional security researchers for *600 man-hours*. What is worse, as reported in a

File: net/qrtr/tun.c ***Commit:*** https://github.com/torvalds/linux/commit/a21b7f0cff1906a93a0130b74713b15a0b36481d ***Vulnerability Type:*** Missing Release of Memory after Effective Lifetime (CWE-401)	**Run 1**	**Run 2**
1 static ssize_t qrtr_tun_write_iter(struct kiocb *iocb, struct iov_iter *from)	0	0
2 {	0	0
3 kbuf = kzalloc(len, GFP_KERNEL);	1	0
4 if (!kbuf)	1	1
5 return -ENOMEM;	0	1
6 if (!copy_from_iter_full(kbuf, len, from))	0	1
7 return -EFAULT;	1	0
8 ret = qrtr_endpoint_post(&tun->ep, kbuf, len);	0	1
9 return ret < 0 ? ret : len;	1	0
10 }	0	0

Fig. 1. Explanation results (i.e., vulnerability-related contexts highlighted by "1") achieved by a leading vulnerability explainer IVDetect on the same vulnerable code in `Linux Kernel`.

recent work [15], existing vulnerability datasets are prone to varying degrees of quality issues such as noisy labels and duplication, which may seriously degrade the reliability of detection models.

2) *Lack of Explainability:* Due to the *black-box* nature of NN models, GNN-based approaches fall short in the capability to explain why a given code is predicted as vulnerable [13], [16]. Such a lack of *explainability* could hinder their adoption when applied to real-world usage as substitutes for traditional security analyzers [17]. To reveal the decision logic behind the binary detection results (vulnerable or not), several approaches have been proposed to provide additional explanatory information [18]. For example, IVDetect [14] leverages a model-agnostic explanation approach, named GNNExplainer [19], to simplify the detected vulnerable code to a *minimal* program dependence subgraph consisting of a set of crucial statements along with program dependencies while retaining the initial model prediction as explanations. Unfortunately, due to the complexity of code structures and the diversity of candidate program subsets, existing instance-based explanation approaches may break the criteria of *stability*, i.e., such extracted explanations may not be consistent with the same input for different runs. A failure case is shown in Fig. 1, a *memory leak* vulnerability occurs when the allocated `buf` (at line 3) is not released in case of error or success return, allowing attackers to cause a denial of service.[1] In the first run, IVDetect identifies statements at line 3, 4, 7, 9 as vulnerable, while in the second run, it turns to pinpoint statements at line 4, 5, 6, 8 as explanations. As a result, explanations provided by existing approaches fail to faithfully reflect the decision mechanism of the detection model, making the security practitioners quite confused and not trust the explanation results.

To tackle the above two challenges, we propose a novel DL-based approach, named EXVUL, for Effective and eXplainable IoT VULnerability detection. The key insights underlying our approach include (❶) combining the strengths of large-scale unlabeled code corpus and limited labeled data to train an effective IoT vulnerability detection model, as well as (❷) providing both *faithful* (reflecting the decision mechanism of the to-be-explained detection model) and *stable* (explanation results are consistent with the same input for different runs) explanations. Specifically, to solve the first issue, EXVUL adopts a novel combinatorial contrastive learning (CL) paradigm to facilitate learning better code representations in a self-supervised manner for the downstream detection task, while making use of limited label information to distinguish IoT vulnerable code from benign ones. To address the second issue, we propose a deviation-aware strategy, which aligns the feature embedding of the input code snippet with its explanatory candidate set in the latent space to improve inconsistency, and incorporate it into GNNExplainer to obtain more faithful and stable explanations.

To evaluate the effectiveness of our proposed EXVUL, we conduct experiments on a real-world IoT vulnerability dataset composed of 1471 vulnerable functions. The experimental results show that EXVUL significantly outperforms the state-of-the-art baselines from 33.44% to 72.91% in terms of Accuracy, and from 19.52% to 98.78% in terms of F1, indicating the effectiveness of EXVUL in IoT vulnerability detection. Besides, EXVUL improves over the best-performing baseline explainer PGExplainer by 22.97% in mean statement precision (MSP), 49.55% in mean statement recall (MSR), and 48.40% in mean intersection over union (MIoU), demonstrating that the explanations provided by EXVUL can correctly point out the vulnerable statements relevant to the detected vulnerabilities. Finally, this article makes the following contributions.

1) We propose a research problem that the lack of labeled data and explainability pose a critical challenge to migrate existing DL-based approaches to IoT vulnerability detection and need to be treated together.
2) We propose EXVUL, a novel DL-based approach for effective and explainable IoT vulnerability detection. EXVUL adopts a combinatorial CL paradigm to train a well-performing detection model over limited IoT vulnerability samples, and incorporates a novel deviation-aware alignment strategy into the state-of-the-art explanation approach GNNExplainer to provide both faithful and stable explanations.
3) Extensive experimental results and user study show substantial improvements EXVUL brings to IoT vulnerability detection and explainability.

The rest of this article is organized as follows. Section II introduces the background knowledge related to our problem. 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 and outlines our future research agenda.

II. Background

In this section, we briefly introduce the general pipeline of DL-based vulnerability detection and explanation. Then, we discuss related techniques used in our approach, including CL and GNN-specific explanation framework.

A. Problem Definition

Following [14] and [20], explainable vulnerability detection (EVD) is generally expanded from a well-trained binary

[1] https://nvd.nist.gov/vuln/detail/CVE-2019-19079

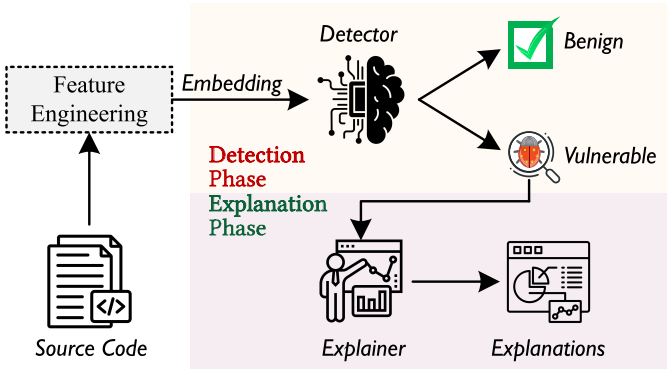


Fig. 2. Workflow of DL-based vulnerability detection and explanation.

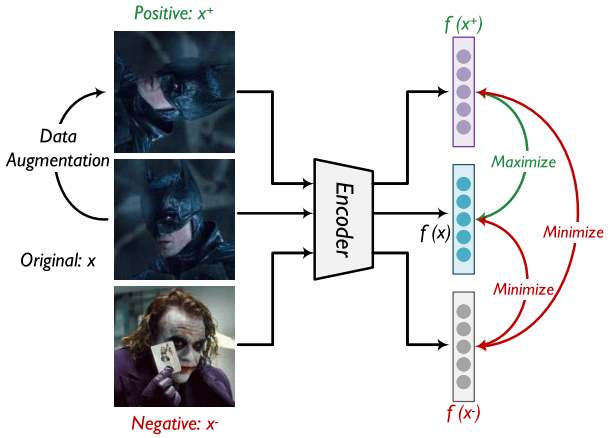


Fig. 3. Self-supervised CL pipeline.

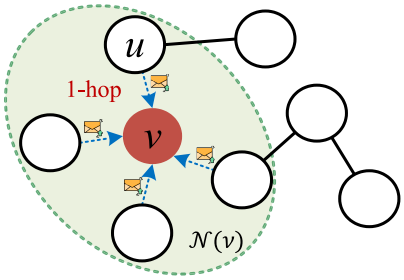


Fig. 4. Neighborhood aggregation scheme in GNNs.

classifier by appending a *post-hoc* explainer, and the "classification with explanation" workflow for an individual instance can be illustrated as Fig. 2. The definitions of EVD and its two components (the detector and explainer) are formalized as:

DEFINITION 1 *(EVD):* Given a code snippet $\mathcal{C}$, EVD first performs feature engineering to embed it into a feature representation $\mathcal{H}$, and then generates two outputs, respectively, from its detector and explainer: a predicted label $\mathcal{Y} \in \{0, 1\}$ with 1 for vulnerable and 0 otherwise, and an explanation $\mathcal{E}$ indicating why the sample is predicted as vulnerable.

DEFINITION 2 *(DETECTOR):* The detection pipeline can be further decoupled into two components, a feature encoder and a classifier. It is defined as $\mathcal{Y} = g(f(\mathcal{H}))$, where the feature encoder $f(\cdot)$ learns to capture vulnerability-related features from $\mathcal{H}$, and the classifier $g(\cdot)$ assigns it a binary label $\mathcal{Y}$.

DEFINITION 3 *(EXPLAINER):* Given a code snippet $\mathcal{C}$ detected as vulnerable, i.e., $\mathcal{Y} = 1$, the explanation $\mathcal{E}$ is a set of important features $\mathcal{H}' \in \mathcal{H}$ positively (or above a certain threshold) contributing to the model's prediction. These important features imply the risky behaviors of the vulnerable code.

B. Contrastive Learning

Given that the limited labeled data in downstream tasks, CL, a popular self-supervised learning paradigm, has emerged as a promising approach in computer vision (CV) [21] and natural language processing (NLP) [22] for learning better feature representations without supervision from labels [23]. The goal of CL is to maximize the agreement between original sample and its positive (i.e., similar) variant while minimizing the agreement between original sample and a negative (i.e., dissimilar) sample. Positive sample x^+ is a semantically equivalent (SE) variant derived from the anchor x by applying built-in pretext tasks (also known as *data augmentation*), while negative sample x^- is the other sample different from x. The general pipeline of CL is shown in Fig. 3. The positive sample x^+ of an image x is constructed by data augmentation such as rotation and cropping. Then, x^+ and x will be fed into the feature encoder with other images x^- (labeled as negatives) to produce better embeddings via minimizing the contrastive loss function.

C. GNN-Specific Explanation Framework

Due to the outstanding representation learning ability for structured graph data, GNNs [24] have been applied to a variety of research domains such as natural science [25], knowledge graphs [26], and blockchain [27]. As shown in Fig. 4, modern GNNs mostly follow a neighborhood aggregation scheme, where the node feature is updated by iteratively aggregating message from its κ-hop neighbors, to capture the semantic features from the graph structure. This procedure can be formulated by

$$\boldsymbol{h}_v^{(t)} = \sigma\left(\boldsymbol{h}_v^{(t-1)}, \mathrm{AGG}^{(t)}\left(\left\{\boldsymbol{h}_u^{(t-1)} : u \in \mathcal{N}(v)\right\}\right)\right) \quad (1)$$

where $\boldsymbol{h}_v^{(t)}$ is the feature representation of node $v \in \mathcal{V}$ at the tth iteration, $u \in \mathcal{N}(v)$ is the neighbors of v, and $\mathrm{AGG}(\cdot)$ and $\sigma(\cdot)$ denote aggregation (e.g., *MEAN*) and activation (e.g., ReLU) functions for node feature computation. After iterating T time steps, the final node representation matrix $H_k^{(T)} = \{h_v^{(T)}\}_{v=1}^{\mathcal{V}}$ of graph $\mathcal{G}_k$ is used for downstream tasks.

Despite their effectiveness, the lack of explainability creates key barriers to the adoption of GNNs in practice. Recently, several studies [19], [28] have attempted to explain the decisions of GNNs by applying a perturbation-based strategy, a representative effort is GNNExplainer [19]. Typically, it formulates the problem by maximizing the mutual information (MI), which quantifies the consistency between original predictions and prediction of candidate explanation, between

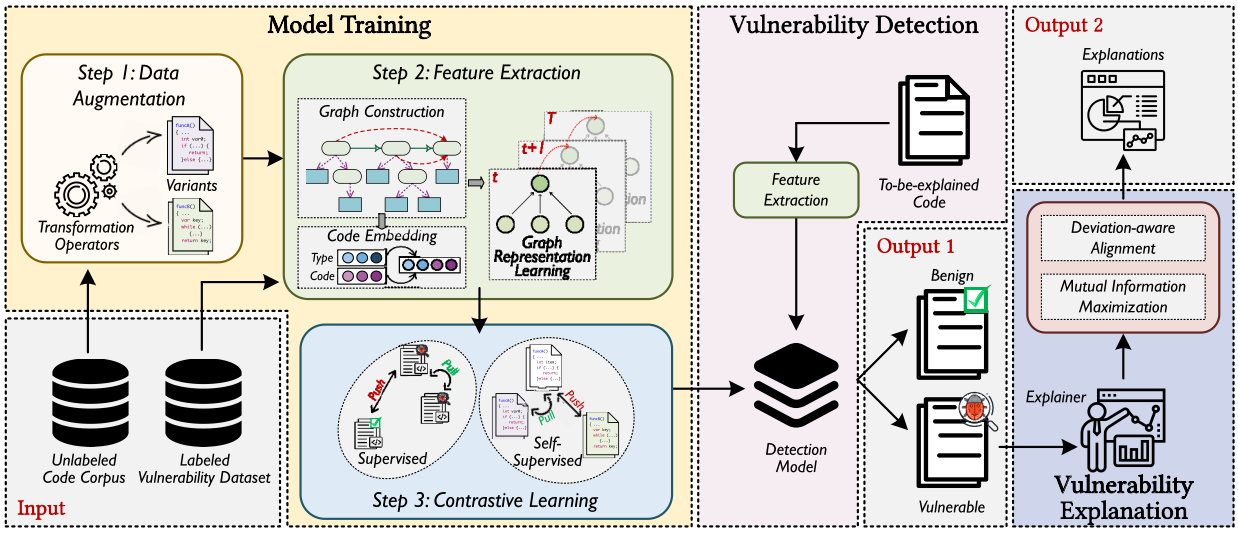


Fig. 5. Overall architecture of our proposed EXVUL.

the minimal explanatory substructure $\mathcal{G}'_k$ of the kth graph $\mathcal{G}_k$ and its predicted label $\hat{\mathcal{Y}}$:

$$\max_{\mathcal{G}'_k} MI(\hat{\mathcal{Y}}, \mathcal{G}'_k) = H(\hat{\mathcal{Y}}) - H\left(\hat{\mathcal{Y}}|G = \mathcal{G}'_k\right) \tag{2}$$

where $H(\hat{\mathcal{Y}})$ is the entropy term and $H(\hat{\mathcal{Y}}|G = \mathcal{G}'_k)$ is the conditional entropy term.

For optimization, GNNExplainer treats the subgraph $\mathcal{G}'_k$ as a random graph variable $\mathcal{G}$ and leverages edge mask to find a subgraph that can best predict the original output $\hat{\mathcal{Y}}$ as follows:

$$\min_{\mathcal{G}} \mathbb{E}_{\mathcal{G}'_k \sim \mathcal{G}} H\left(\hat{\mathcal{Y}}|G = \mathcal{G}'_k\right). \tag{3}$$

After optimization, the subgraph $\mathcal{G}'_k$ is the generated explanations for the prediction of the input graph $\mathcal{G}_k$.

III. Our Approach: EXVUL

In this section, we present the details of our novel approach named EXVUL, which integrates accurate binary results and understandable explanations for effective and explainable IoT vulnerability detection.

A. Overview

Fig. 5 shows the overall architecture of our proposed EXVUL approach, which consists of three main phases: model training, vulnerability detection, and vulnerability explanation.

The model training phase includes three steps. In *Step 1* (Section III-B), EXVUL conducts multiple semantically-preserving transformation on unlabeled code corpus to construct equivalent variants and arranges them into a mini-batch for self-supervised CL. The labeled dataset (including vulnerable code and their corresponding patched version) is fed into another mini-batch for supervised CL. Then, in *Step 2* (Section III-C), both unlabeled and labeled samples are embedded into numerical graph representations through graph construction and code embedding, and then fed into an attention-based GNN to extract representative features. Finally, in *Step 3* (Section III-D), a well-performing IoT vulnerability detection model is produced by performing our novel combinatorial CL over the representations of both unlabeled and labeled code samples in the latent feature space.

In the vulnerability detection phase (Section III-E), EXVUL first splits the target program into functions and repeats feature extraction (*Step 2*) to obtain corresponding vector representations. Then, for each code snippet, both unstructured node embeddings and structured relations are feed into the well-trained detection model for classification.

In the vulnerability explanation phase (Section III-F), EXVUL incorporates a novel deviation-aware alignment strategy into the state-of-the-art explanation approach GNNExplainer to provide both *faithful* and *stable* explanations.

TABLE I
SEMANTIC-PRESERVING TRANSFORMATIONS WE ADOPTED FOR DATA AUGMENTATION

No.	Name	Description
1	Identifier Renaming	Substitute the function/variable name with a random token.
2	Operand Swap	Swap the operands of binary logical operations.
3	Statement Permutation	Swap two lines of statements that have no dependency.
4	Loop Exchange	Replace for loops with while loops or vice versa.
5	Block Swap	Swap `then` block of a chosen `if` statement with its corresponding `else` block.
6	Switch to If	Replace a `switch` statement with its equivalent `if` statement.

B. Data Augmentation

In order to construct positive variants of unlabeled programs for self-supervised CL, we first perform static analysis to parse each source code into an abstract syntax tree (AST) and traverse it to search for potential injection locations. Following [29], once an injection location is found, an applicable augmentation operator $\Phi \in \{\phi_1, \phi_2, \ldots, \phi_6\}$ (shown in Table I) will be randomly selected and applied to get the transformed node. We then adapt the context accordingly, and translate it to the positive variants. Subsequently, we arrange original code samples along with their SE variants (i.e., positives) as inputs in a mini-batch. In this way, augmented samples originated from one pair are negatively correlated to any sample from other pairs within a mini-batch during self-supervised CL. For supervised CL, we directly regard samples with the same label as positives and the others as negatives.

C. Feature Extraction

After data augmentation, both unlabeled original-variant sample pairs and labeled vulnerability dataset should be converted into feature embeddings acceptable for DL models. In particular, feature extraction includes three main steps, including graph construction, code embedding, and graph representation learning.

1) Graph Construction: To model the discriminative features beneficial to distinguishing vulnerable and benign code, we first perform static analysis to generate a joint graph structure, code property graph (CPG) [30], for each input code at the *function-level*. CPG is a classic abstract representation, which integrates AST, control flow graph (CFG), and program dependence graph (PDG), in security-related program analysis and has been proved to be a powerful tool for vulnerability discovery [13], [31]. Fig. 6 shows a simple code sample and its corresponding CPG. 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 beneficial to feature representation learning. The node set $\mathcal{N}$ of CPG is composed of statement nodes in CFG (or PDG) and leaf nodes in AST, and the edge set $\mathcal{R}$ is composed of three types of relations.

2) Code Embedding: After graph construction, we convert the code tokens of each CPG's node into low-dimensional vector representation for subsequent feature representation learning. Specifically, to effectively alleviate the vocabulary explosion problem, we first perform abstraction on user-defined identifiers by replacing formal parameters and local variables defined by developers with a normalized symbol `PARAM_i` and `VAR_j`, respectively. Then, we use Word2Vec [32] to encode leaf nodes in AST. Considering some important structure information may be lost when converting graphs into low-dimensional vectors with Word2Vec, we use Node2Vec [33] as an alternative to encode statement nodes in CFG and PDG. Node2Vec can capture data dependency and control dependency between statements because it transfers the information of two nodes bidirectionally and encodes a node with the information from its surrounding structures. In addition, we 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 label encoding, which transforms text into numerical value. Finally, we concatenate the node representation C_v of each node $v \in \mathcal{N}$ with the type representation T_v as the initial representation as follows:

$$\boldsymbol{h}_v^{(0)} = C_v || T_v \tag{4}$$

where $||$ denotes the concatenation operator.

3) Graph Representation Learning: To capture global vulnerability semantics for classification, we employ GAT [34], a state-of-the-art GNN with multihead attention, to iteratively propagate and aggregate node information along with different edges. Formally, given a CPG node v, its node representation after tth iteration is updated as

$$\boldsymbol{h}_v^{(t+1)} = \frac{1}{|\mathcal{R}|} \sum_{r \in \mathcal{R}} \left(\sigma \left(\sum_{u \in \mathcal{N}_r} \boldsymbol{\alpha}_{v,u}^{(t)} \boldsymbol{z}_u^{(t)} \right) \right) \tag{5}$$

$$\boldsymbol{z}_u^{(t)} = \boldsymbol{W}_r^{(t)} \boldsymbol{h}_u^{(t)} \tag{6}$$

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 1-hop neighbors of node v under the edge r. $\boldsymbol{W}_r$ represents the weight matrix under the edge r. $\boldsymbol{\alpha}_{v,u}$ represents the attention weight between the node v and its neighbor u under the edge r

$$\boldsymbol{\alpha}_{v,u}^{(t)} = \frac{\exp\left(\boldsymbol{e}_{v,u}^{(t)}\right)}{\sum_{p \in \mathcal{N}_r} \exp\left(\boldsymbol{e}_{v,p}^{(t)}\right)} \tag{7}$$

$$\boldsymbol{e}_{v,u}^{(t)} = \sigma\left(\vec{\boldsymbol{a}}_r^{(t)^{\mathrm{T}}} \left(\boldsymbol{z}_v^{(t)} || \boldsymbol{z}_u^{(t)} \right) \right) \tag{8}$$

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

D. Contrastive Learning

To alleviate the labeled data scarcity issue, we propose a new combinatorial CL-based training strategy to combine the strengths of large-scale unlabeled code corpus and limited IoT vulnerability dataset. Specifically, for unlabeled code samples along with their SE variants (positives), we compute self-supervised contrastive (SupCon) loss to learn better code presentations. Meanwhile, for labeled IoT vulnerability dataset, we calculate SupCon loss based on vulnerable code and their patched version. Below, we elaborate on each component of our CL with more technical details.

1) Self-Supervised Contrastive Loss: Specifically, given a set of N randomly sampled unlabeled code pairs $\{\tilde{c}_i\}$, where $\tilde{c}_{2d-1}$ and $\tilde{c}_{2d}$ are the original and augmented view of $\{c_d\}_{d=1,\ldots,N}$, respectively, in the mini-batch $\mathcal{B} \equiv \{1, \ldots, 2N\}$,

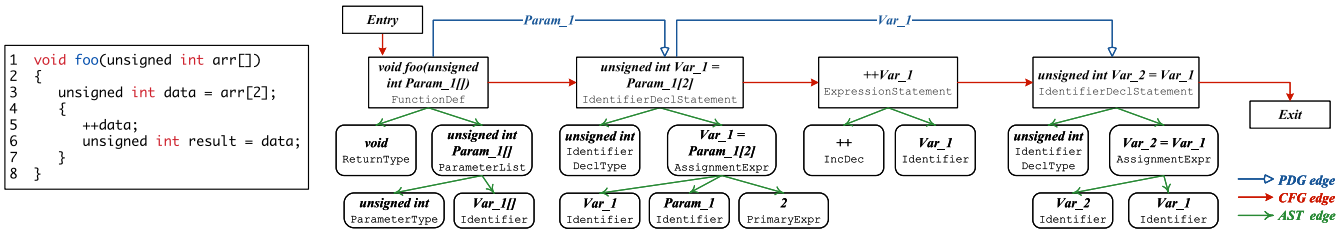


Fig. 6. Exemplary code sample (left) and its corresponding CPG (right).

we employ the noise contrastive estimate (NCE) [21] to compute the self-supervised loss $\mathcal{L}_{\text{con}}^{\text{self}}$

$$\mathcal{L}_{\text{con}}^{\text{self}} = \frac{1}{|\mathcal{B}|}\sum_{i\in\mathcal{B}} -\log\frac{\exp(H_i'\cdot H_{j(i)}'/\tau)}{\sum_{a\in\mathcal{A}(i)}\exp(H_i'\cdot H_a'/\tau)} \tag{9}$$

where H_i' represents the low-dimensional embedding of the graph-level representation $H_i^{(T)} = \{\boldsymbol{h}_v^{(T)}\}_v^{\mathcal{V}}$ of an arbitrary (original or augmented) sample $\tilde{c}_i$ via a projection head consisting of a MLP layer with a single hidden layer. $j(i)$ is the index of the other view originating from the same source. $\tau \in \mathcal{R}^+$ is the temperature parameter to scale the loss, and $\mathcal{A}(i) \equiv \mathcal{B}\backslash\{i\}$.

2) Supervised Contrastive Loss: In addition, to effectively leverage limited IoT vulnerability samples with human annotations, we employ an additional SupCon loss term [35] during the training process. The the use of label information encourages the feature encoder to closely aligns all samples from the same class in the latent space to learn more accurate (in terms of samples with the same label) cluster representations for vulnerable and benign samples. Formally, the SupCon loss $\mathcal{L}_{\text{con}}^{\text{sup}}$ is written as

$$\mathcal{L}_{\text{con}}^{\text{sup}} = \frac{1}{|\mathcal{B}^l|}\sum_{i\in\mathcal{B}^l}\frac{-1}{|\mathcal{Q}(i)|}\sum_{q\in\mathcal{Q}(i)}\log\frac{\exp(H_i'\cdot H_q'/\tau)}{\sum_{a\in\mathcal{A}(i)}\exp(H_i'\cdot H_a'/\tau)} \tag{10}$$

where $\mathcal{B}^l$ corresponds to the used IoT vulnerability dataset, and $\mathcal{Q}(i) \equiv \{q \in \mathcal{A}(i) : \tilde{y}_q = \tilde{y}_i\}$ is the set of indices of all other *positives* that hold the same label. Particularly, the patched version of the vulnerable code in $\mathcal{B}^l$ can serve as *hard negatives* to capture more subtle yet discriminative vulnerability features. $1/|\mathcal{Q}(i)|$ is the positive normalization factor which serves to remove bias present in multiple positives samples and preserve the summation over negatives in the denominator to increase performance.

Finally, the total loss used to train a robust feature encoder over the batch is defined as

$$\mathcal{L}_{\text{total}} = (1-\lambda)\mathcal{L}_{\text{con}}^{\text{self}} + \lambda\mathcal{L}_{\text{con}}^{\text{sup}} \tag{11}$$

where λ is a weight coefficient to balance the two loss terms.

E. Vulnerability Detection

In the detection phase, given a code snippet, we aim to apply a well-trained detection model (more specifically, a binary classifier) to identify potential IoT vulnerabilities. Similar to the feature extraction phase (Section III-C) in model training, program semantics reflected in the CPG of source code are first captured through static analysis. Next, each node in CPG is embedded into low-dimensional vectors through code embedding. Then, both unstructured node embeddings and structured relations in CPGs are feed into the well-trained feature encoder [i.e., the GAT model trained with (11)] for feature extraction. Finally, the prediction (i.e., vulnerable or not) is made by a classifier composed of a one-layer fully connected layer.

F. Vulnerability Explanation

To derive explanations on why the detection model has decide on the vulnerability, we follow the most related work IVDetect [14], which aims to find a subgraph $\mathcal{G}_k'$, which covers the key nodes (tokens/statements) and edges (program dependencies) that are most decisive to the prediction label, from the graph representation $\mathcal{G}_k$ of the detected vulnerable code c_k via GNNExplainer [19]. The main difference lies in that we aim to seek both *faithful* (reflecting the decision mechanism of the to-be-explained detection model) and *stable* (explanation results should be consistent with the same input for different runs) explanations. Hence, we incorporate a novel deviation-aware loss term $\mathcal{L}_{\text{Align}}$ into GNNExplainer to identify explanatory CPG subgraphs while preserving their alignment with original inputs.

Specifically, we first leverage the CPGs $\{\mathcal{G}_k\}_{k=1}^n$ of other vulnerable code from the dataset to obtain a global view of the graph representation $\{\sum_{v\in\mathcal{V}_k'}\boldsymbol{h}_{v,k}^{l+1}/|\mathcal{V}_k'|\}_{k=1}^n$, where $\boldsymbol{h}_{v,k}^l$ denotes embeddings of node v in graph $\mathcal{G}_k$ at layer l, and $\mathcal{V}_k'$ is a set of selected nodes after graph pooling. Then, a clustering algorithm is applied to divide the latent representations of samples in the embedding space into $\mathcal{X}$ groups. The representative graph embeddings are assigned as anchors $\{\boldsymbol{h}^{l+1,x}\}_{x=1}^{\mathcal{X}}$ to measure the distance between $\mathcal{G}_k$ and $\mathcal{G}_k'$ at lth layer for alignment

$$\mathcal{L}_{\text{Align}}(\mathcal{H}_{\mathcal{G}_k}, \mathcal{H}_{\mathcal{G}_k'}) = \sum_l\sum_{v\in\mathcal{V}_k'}\left\|\boldsymbol{s}_v^l - \hat{\boldsymbol{s}}_v^l\right\|_2^2 \tag{12}$$

where $\boldsymbol{s}_{v,x}^l = ||\boldsymbol{h}_v^{l+1} - \boldsymbol{h}_v^{l+1,x}||_2$ represents the relative distance to kth anchor (i.e., the clustering center of each group).

By comparing relative positions, our deviation-aware alignment loss $\mathcal{L}_{\text{Align}}$ provides a simple yet effective strategy to encode the varying importance of each dimension for evaluating alignments in the embedding distribution manifold. Finally, in order to generate both *faithful* and *stable* explanations, we incorporate the deviation-aware alignment loss term $\mathcal{L}_{\text{Align}}$ into GNNExplainer (3) as

$$\min_{\mathcal{G}} \mathbb{E}_{\mathcal{G}_k'\sim\mathcal{G}} H(\hat{\mathcal{Y}}|G = \mathcal{G}_k') + \eta\cdot\mathcal{L}_{\text{Align}} \tag{13}$$

TABLE II
DESCRIPTIVE STATISTICS OF OUR USED DATA SET

Data Source	Dataset	# Vul	# Non-vul
General Software	ReVeal	1,664	16,505
	Devign	11,888	14,149
	Big-Vul	10,547	168,752
IoT Software	Asterisk	94	17,755
	FFmpeg	249	5,552
	Httpd	57	3,850
	LibPNG	45	577
	LibTIFF	123	731
	OpenSSL	159	7,068
	Pidgin	29	8,626
	VLC Player	44	6,115
	Xen	671	9,023
	Total	**1,471**	**59,297**

where η controls the balance between prediction preservation and embedding alignment.

As a result, the code snippet corresponding to the extracted subgraph $\mathcal{G}'_k$ is the explanation for the detected vulnerability sample c_k.

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).

- RQ1: How effective is EXVUL in IoT vulnerability detection as compared to other state-of-the-art approaches?
- RQ2: How well does EXVUL perform on explaining the detection results?
- RQ3: How does combinatorial CL contribute to the performance of EXVUL?
- RQ4: How does deviation-aware alignment contribute to the performance of EXVUL?
- RQ5: What is the influence of hyper-parameters on the performance of EXVUL?

B. Data Set

Given that most DL model-oriented vulnerability datasets are not tailored for IoT vulnerabilities, following Mei et al. [36], we adopt a small-scale vulnerability dataset consisting of 1,471 real-world vulnerable functions crawled from nine open-source software deployed on IoT devices. For example, `VLCPlayer` is a multimedia playback software which is widely integrated in smart home devices for multimedia file playing. In addition, we also employ three general software vulnerability datasets, including ReVeal [13], Devign [12], and Big-Vul [37], to train baseline models.

Table II reports the statistics of two datasets. Column 1 reports the data source of each dataset. Column 2 lists the concrete projects selected to collect vulnerability samples. Columns 3-4 denote the function-level statistics of each project, including the number of vulnerable functions (Column 3) and non-vulnerable functions (Column 4). The Reveal dataset [13] is collected by tracking the history vulnerability fixes in two popular open-source projects: Linux Debian Kernel and Chromium. In total, ReVeal dataset includes 18 169 functions, in which 9.9% of them are vulnerable (1664 vulnerable functions). The Devign dataset [12] contains a set of security issue-related commits from Linux Kernel, QEMU, Wireshark, and FFmpeg projects. It includes 26 037 functions, in which 45% of them are vulnerable (11 888 vulnerable functions). The Big-Vul [37] dataset is collected from over 300 C/C++ GitHub projects. It covers approximately 10k vulnerable functions and 177k nonvulnerable functions involved in vulnerability reports from 2002 to 2019. In total, there are 1,471 vulnerable functions along with 59 279 nonvulnerable ones in IoT software.

C. Baselines

To demonstrate the effectiveness of EXVUL on vulnerability detection, we adopt four state-of-the-art DL-based binary vulnerability detectors:

1) *VulDeePecker* [4] 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.
3) *Devign* [12] combines multiple code representations (e.g., AST, CFG) to model programs at the function-level, and adopts GGNN [38] to learn comprehensive vulnerability semantics for classification.
4) *ReVeal* [13] proposes to leverage CPG and GGNN to automatically learn the graph properties of source code.

Compared to traditional rule-based analysis tools [8], [9], [10], these approaches have shown promising results in scalability and effectiveness because they can automatically learn implicit vulnerability features without any prior knowledge [39].

To investigate the effectiveness of EXVUL on vulnerability explanation, we employ three recent GNN-specific explanation approaches as baselines:

1) *GNNExplainer* [19] is one of the most popular explanation approaches and has been integrated into the state-of-the-art vulnerability explainer IVDetect to simplify the detected vulnerable code to a *minimal* program dependence subgraph composed of a set of crucial statements along with program dependencies while retaining the initial model prediction as explanations. Its key idea lies in accomplishing a maximum MI optimization task, which leverages edge and feature masks to select important structures and features.
2) *PGExplainer* [28] enables simultaneous explanation of multiple instances, whereas GNNExplainer is developed for individual graph instance. It trains a parameterized

mask predictor (so-called explanation network) on a universal embedding of graph edges to predict edge masks.

3) *GNN-LRP* [40] leverages a higher-order Taylor decomposition of model prediction to decompose the scores into the importance of different walks. The relevance per walk is computed using a back-propagation similar to LRP for each node in the walk. The final output of GNN-LRP is the set of walks associated with the prediction.

D. Evaluation Metrics

We used the following evaluation metrics to measure the performance of EXVUL on vulnerability detection.

1) *Accuracy (Acc)* evaluates the performance that how many instances can be correctly labeled. It is calculated as: $\text{Acc} = (\text{TP} + \text{TN})/(\text{TP} + \text{FP} + \text{TN} + \text{FN})$.
2) *Precision (Pre)* is the fraction of true vulnerabilities among the detected ones. It is defined as: $\text{Pre} = \text{TP}/(\text{TP} + \text{FP})$.
3) *Recall (Rec)* measures how many vulnerabilities can be correctly detected. It is calculated as: $\text{Rec} = \text{TP}/(\text{TP} + \text{FN})$.
4) *F1-score (F1)* is the harmonic mean of *Recall* and *Precision*, and can be calculated as: $F1 = 2 * (\text{Rec} * \text{Pre})/(\text{Rec} + \text{Pre})$.

To quantify the quality of generated explanations, we use three fine-grained vulnerability-triggering paths (VTPs) metrics [41] to evaluate the faithfulness of explanations, and the *Stability* metrics [42] to evaluate the consistency of explanations, respectively. They are formally defined as follows.

1) *MSP:* $MSP = (1/N)\sum_{i=1}^{N} SP_i$, where $SP_i = |S_e \cap S_p|/|S_e|$ represents the proportion of contextual statements truly related to the detected vulnerability sample $i \in N$ in the explanations. Here, S_e denotes the set of explanatory statements provided by explainers, while S_p denotes the set of labeled vulnerability-contexts (ground truth) in the dataset. $|\cdot|$ represents the size of a set.
2) *MSR:* $MSR = (1/N)\sum_{i=1}^{N} SR_i$, where $SR_i = |S_e \cap S_p|/|S_p|$ denotes that how many contextual statements in the triggering path of the detected vulnerability sample i can be covered in explanations.
3) *MIoU:* $MIoU = (1/N)\sum_{i=1}^{N} IoU_i$, where $IoU_i = |S_e \cap S_p|/|S_e \cup S_p|$ reflects the degree of overlap between the explanatory statements and the contextual statements on the VTP.
4) *Stability (Stb):* $Stb_i = (1/C_M^2)\sum_{j=1}^{C_M^2} IoU_j$, where $IoU_j = |S_m \cap S_n|/|S_m \cup S_n|$ reflects the degree of overlap between sample i's explanatory statements in the run m and $n \in [1, M]$. C_M^2 is the combination operation, i.e., the total combinations when comparing the results of any two runs. We calculated the arithmetic mean for evaluation in our experiments.

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. We implemented our approach in Python using PyTorch.[2] We generated CPGs and SE variants of the code snippets based on the ASTs parsed by `tree-sitter`.[3] The dimension of the vector representation of each node/token in CPG is set to 128 and the dropout is set to 0.1. The other hyper-parameters of our approach are tuned through grid search. For model training, we employed CodeSearchNet [43], a large-scale unlabeled code corpus which contains 2.1M bimodal comment-function pairs and 6.4M unimodal functions across six programming languages, to perform self-supervised CL, and conducted supervised CL on our IoT vulnerability dataset, in which vulnerable samples are regarded as positives, and patches (benign/nonvulnerable samples) are negatives (the patched version of the anchor sample is *hard negative*).

F. RQ1: Effectiveness on Vulnerability Detection

Objective: Benefiting from the powerful representation capability of deep NNs, many DL-based vulnerability detection approaches have been proposed. However, as manually constructing such a large-scale dataset with human annotations for IoT vulnerabilities is nontrivial and time-consuming, it's unrealistic to train a well-performing IoT vulnerability detection model. In this article, we propose a novel approach EXVUL, which combines the strengths of large-scale unlabeled code corpus and limited labeled data to train an effective IoT vulnerability detection model. The experiments are conducted to investigate whether EXVUL outperforms state-of-the-art DL-based approaches, which are pretrained on general software vulnerability dataset, on the IoT vulnerability detection task.

Experimental Design: We considered four state-of-the-art baselines: VulDeePecker, SySeVR, Devign, and ReVeal. As mentioned earlier, since the above four DL-based binary vulnerability detectors are not designed for IoT vulnerabilities, we, respectively, trained them based on three general software vulnerability datasets (randomly split into the ratio of 8:2 for training and validation), and applied transfer learning to fine-tune these pretrained models on part of our IoT vulnerability samples to port it for IoT vulnerability detection. For EXVUL, 80% of IoT code samples are treated as training data in supervised CL, 10% of samples are treated as validation data (also used for fine-tuning pretrained baseline models), and the left 10% of samples are treated as testing data. We also keep the distribution as same as the original ones in training, validating, and testing data. Besides, in order to comprehensively compare the performance among baselines and EXVUL, we considered four widely-used binary classification metrics (i.e., Accuracy, Precision, Recall, and F1-score) and conducted experiments on the IoT dataset.

Results: Fig. 7 shows the performance comparison of EXVUL with respect to four state-of-the-art DL-based

[2]https://pytorch.org/
[3]https://tree-sitter.github.io/tree-sitter/

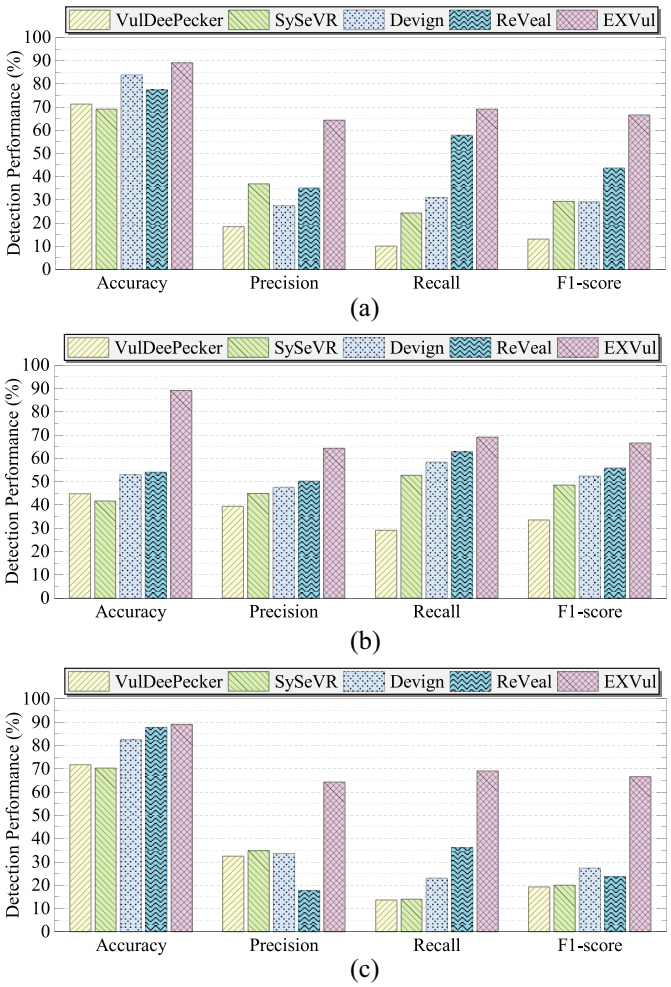


Fig. 7. Performance of IoT vulnerability detection regarding EXVUL and baselines. (a) Comparison with baselines pretrained on the Devign dataset. (b) Comparison with baselines pretrained on the ReVeal dataset. (c) Comparison with baselines pretrained on the Big-Vul dataset.

approaches (pretrained on three datasets) in terms of the aforementioned evaluation metrics. Overall, EXVUL generally outperforms all of the baselines, achieving 0.89 on Accuracy, 0.64 on Precision, 0.69 on Recall, and 0.67 on F1.

In particular, we find that the average improvements of EXVUL over each metric are significant, ranging from 33.44% to 72.91% on Accuracy, from 28.29% to 63.12% on Precision, from 10.09% to 137.06% on Recall, and from 19.52% to 98.78% on F1, respectively. These results verify the effectiveness of our combinatorial CL strategy in detecting IoT vulnerabilities without sufficient labeled data for model training. The root cause for this performance gap is that directly migrating a DL model pretrained on a general vulnerability dataset to IoT vulnerability detection via transfer learning ignores the distribution disparity between the source domain (i.e., general vulnerabilities) and target (IoT vulnerabilities) in feature space, confusing the classifier seriously. By contrast, benefiting from the combination of large-scale unlabeled code corpus and limited labeled vulnerability data, EXVUL learns to capture discriminative semantic features of source code and leverages these features to distinguish vulnerable code from benign ones.

Answer to RQ1: EXVUL outperforms the state-of-the-art baselines on IoT vulnerability detection. Particularly, it achieves overwhelming results at both Accuracy and F1-score, which indicate that EXVUL equipped with self-supervised CL as well as supervised CL has a stronger ability to learn the semantics of IoT vulnerabilities.

G. RQ2: Explainability on Vulnerability Detection

Objective: Though many novel approaches have been proposed and indeed achieved remarkable performance, they fall short in the capability to explain why a given code is predicted as vulnerable. The form of an explanation can be diverse, such as vulnerability types [44], root cause [45], similar vulnerability reports [46], and so on. In this article, we follow the related work IVDetect to formalize the vulnerability explanation as a fine-grained classification task, i.e., locating vulnerability-related code snippets. The experiments are conducted to investigate whether EXVUL outperforms state-of-the-art vulnerability explanation approaches in terms of faithfulness and stability.

Experimental Design: We considered the three state-of-the-art baselines: IVDetect, P2IM, and mVulPreter. To gain the ground truths of the vulnerability samples in the testing set, we adopted a simple yet effective labeling strategy [47], i.e., comparing changed statements between each vulnerable function and its corresponding fixed version in the corresponding vulnerable function according to *diff* files. If a statement was deleted or altered (i.e., starting with "-" in *diff* files), it would be labeled as *vulnerable*, and *nonvulnerable* otherwise. In order to avoid introducing artificial deviation, two postgraduates and one Ph.D participated in this labeling process. If two postgraduates disagreed on the label of the same sample, the sample would be forwarded to the Ph.D evaluator for further investigation. In order to comprehensively compare the performance among baselines and EXVUL, we considered three faithfulness-specific metrics (i.e., MSP, MSR, and MIoU). We reported results averaged across 100 IoT vulnerabilities, which were randomly sampled from our IoT dataset (independent from samples used for training EXVUL), correctly detected by baselines and EXVUL. Due to the randomness in initialization, the explanation for the same instance given by an explainer could be different for different runs. Thus, in addition to comparing to ground truths, we also evaluate the obtained explanations in terms of stability. For the same vulnerability sample, we ran each explainer five times and reported the average values of the stability metric. A higher stability score indicates more consistent explanations at different runs.

Results: Table III shows the performance comparison of our approach with respect to state-of-the-art vulnerability explainers. As can be seen, EXVUL substantially outperforms the best-performing approach PGExplainer by 22.97% in MSP, 49.55% in MSR, and 48.40% in MIoU, respectively. The main reasons leading to this result are two folds. On the one hand, Owing to our combinatorial CL, potential vulnerable behavior of programs are captured by EXVUL, leading to more reliable prediction labels (as discussed in RQ1) for

TABLE III
EVALUATION RESULTS ON VULNERABILITY EXPLANATION IN PERCENTAGE COMPARED WITH BASELINES

Approach	Faithfulness			Consistency
	MSP	MSR	MIoU	Stb
GNNExplainer	25.91	27.14	23.77	8.07
PGExplainer	36.48	33.56	28.12	16.22
Graph-LRP	33.61	38.02	26.94	13.67
EXVUL	44.86	50.19	41.73	35.64

TABLE IV
EVALUATION RESULTS ON VULNERABILITY DETECTION IN PERCENTAGE COMPARED WITH VARIANTS

Approach	Accuracy	Precision	Recall	F1-score
EXVUL_{self}	0.82	0.59	0.47	0.52
EXVUL_{sup}	0.75	0.57	0.61	0.59
EXVUL_{CE}	0.86	0.33	0.26	0.29
EXVUL	0.89	0.64	0.69	0.67

explanation generation. On the other hand, by incorporating our deviation-aware strategy into GNNExplainer, EXVUL can help to identify more important substructures used for predictions, hence yielding the best explanation performances on all metrics (especially MSR).

The consistency of the explanations generated by each explanation approach is also shown in Table III. Unfortunately, the stability scores of all explanation approaches are below 20%. Among them, the consistency of the best-performing explainer PGExplainer is only 16.22%, indicating that for the same input, the explanations generated by each explainer vary greatly at different runs and fail to meet the requirement of trustworthiness. The above results show that the existing explanation approaches suffer from significant variability in the explanation of the same vulnerability. By contrast, we can find that our approach significantly outperforms the state-of-the-art explainers from 119.73% to 341.64% in terms of stability, demonstrating the introduction of our deviation-aware alignment strategy can significantly improve the consistency.

Answer to RQ2: The faithfulness and consistency of existing explanation approaches are not satisfactory, making it difficult for security analysts to establish trust on the model decision. By contrast, our proposed EXVUL significantly outperforms the state-of-the-art explainers in terms of MSP, MSR, MIoU, and Stb, demonstrating the practical value of our approach in providing faithful and stable explanations.

H. RQ3: Impacts of Combinatorial Contrastive Learning

Objective: Different from traditional supervised learning-based vulnerability detection framework, which trains a well-performing model on the large-scale labeled dataset, we propose a new combinatorial CL-based training strategy to combine the strengths of large-scale unlabeled code corpus and limited IoT vulnerability dataset. Therefore, it is important to conduct a study on how the combinatorial CL affect the learning of IoT vulnerability semantics.

Experimental Design: We compared the performance of four versions of EXVUL: with only self-supervised CL (denoted as $\text{EXVUL}_{\text{self}}$), with only supervised CL (denoted as $\text{EXVUL}_{\text{sup}}$), without CL (equivalent to tradition supervised learning with cross-entropy, and denoted as EXVUL_{CE}, and with combinatorial CL (the default EXVUL). The experimental dataset is set the same as the experiment of RQ1 (i. e., 80% for training, 10% for validation, and 10% for testing). We also consider the four performance measures(i. e., Accuracy, Precision, Recall, and F1-score) for comprehensively studying the impact of different training strategies.

Results: As shown in Table IV, compared with $\text{EXVUL}_{\text{self}}$ and $\text{EXVUL}_{\text{sup}}$, EXVUL improves the Accuracy by 8.53% and 18.67%, respectively, and improves the F1-score by 28.85% and 13.56%, respectively. The reason for the improvements is that, benefiting from the combination of large-scale unlabeled code corpus and limited labeled vulnerability data, EXVUL learns to capture discriminative semantic features of source code and leverages these features to distinguish vulnerable code from benign ones. In addition, we can find that both $\text{EXVUL}_{\text{self}}$ and $\text{EXVUL}_{\text{sup}}$ outperform EXVUL_{CE} in terms of all metrics. The potential cause for this performance gap may be that suffering from the severe imbalanced sample distribution (only 2.48% samples in our IoT dataset are vulnerable), the classifier fails to identify vulnerable samples from plenty of nonvulnerable samples. Furthermore, $\text{EXVUL}_{\text{sup}}$ slightly outperforms $\text{EXVUL}_{\text{self}}$ by 29.79% in terms of Recall. The results indicate that by performing supervised CL on the labeled vulnerability dataset at the same time as model training, the detection model can effectively capture the vulnerability semantics for classification.

Answer to RQ3: Self-supervised and supervised CL present their own advantages in learning program semantics, and combining them together can produce the best improvements on IoT vulnerability detection.

I. RQ4: Impacts of Deviation-Aware Alignment

Objective: As mentioned in RQ2, the reason why existing explanation approaches are unreliable is that, due to the randomness in initialization of the explainer, the explanation for the same instance given by an explainer could be different for different runs, which violates the stability of explanations. Therefore, we want to conduct a deeper experiment on how our proposed deviation-aware alignment strategy impacts the performance of EXVUL on vulnerability explanation.

Experimental Design: Similar to RQ2, we still adopted three aforementioned GNN-based explanation approaches (GNNExplainer, PGExplainer, and Graph-LRP) as baselines. For each approach, we created a variant by incorporate our deviation-aware alignment loss term into its optimization [as we did in (13)]. We empirically ran each approach ten times on 100 sampled IoT vulnerabilities used in RQ2, and calculated the *Stb* metric after each run. Although the value of *Stb* may

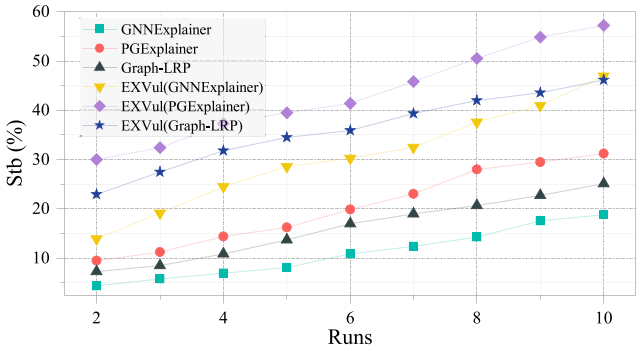


Fig. 8. Varying performance of each approach (with and without our deviation-aware alignment strategy) in terms of Stability.

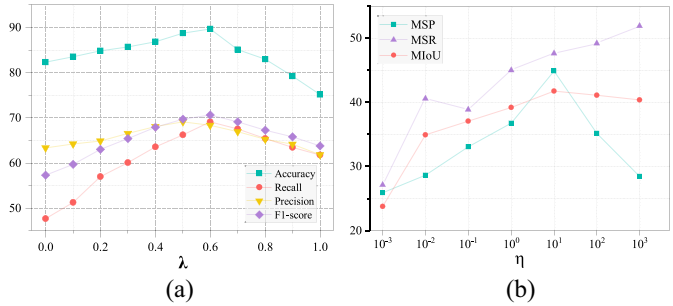


Fig. 9. Sensitivity of EXVUL toward different hyper-parameters. (a) Detection performance under different weights of supervised loss. (b) Explanation performance under different weights of alignment loss.

benefit from more runs or randomness of graph sampling, comparison between different execution rounds is beyond the scope of this research and we leave that for future research.

Results: The evaluation results of each approach (with and without our deviation-aware alignment strategy) are illustrated in Fig. 8. According to the results, we find that it can be seen that the overall stability increases slowly with increasing runs. It is reasonable since the more rounds the explainer runs, the more likely the explanations will overlap. In addition, the stability of all explanation approaches go up with the incorporation with our deviation-aware alignment, which validates effectiveness of our proposal in obtaining consistent explanations.

Answer to RQ4: The overall stability increases slowly with increasing runs. With the incorporation with our deviation-aware alignment, the stability of each approach can be significantly improved.

J. RQ5: Influences of Hyper-Parameters on EXVUL

Objective: In our approach, two key hyper-parameters λ and η affects the effectiveness of vulnerability detection and explanation, respectively. The former balances the weights between feature representations learned in self-supervised and supervised manner, the latter makes a tradeoff between faithfulness and stability of generated explanations. Therefore, we vary the hyper-parameters λ and η to explore EXVUL's sensitivity toward both two values.

Experimental Design: To keep simplicity, all other configurations were kept consistent with RQ1 and RQ2, including data split and metric computation. λ was varied from 0 to 1 with an internal of 0.1, while η was varied at scale [1e-3, 1e-2, 1e-1, 1, 10, 1e2, 1e3].

Results: The evaluation results are shown in Fig. 9. In particular, for vulnerability detection, we can observe from Fig. 9(a) that, different weights of self-supervised and supervised loss in our combinatorial CL has varying impart on EXVUL's performance. all the metrics of EXVUL go up with the increasing of the weight of SupCon part, and reach the optimal performance (except Precision) when λ equals 0.6. After that, each metric drop to different degrees. The results indicate that the use of (partial) labeled samples is benefit to the performance of IoT vulnerability detection.

In addition, for vulnerability explanation, we can find from Fig. 9(b) that, higher weights of embedding alignment loss bring a noticeable improvement in MSP, MSR, and MIoU, which shows that our deviation-aware alignment strategy is helpful for generating both faithful and stable explanations. However, we also observe that blindly increasing the weight (larger than 10) is not always beneficial, and even result in a performance drop (e.g., MSP and MIoU).

Answer to RQ5: Different settings of hyper-parameters can influence the performance of EXVUL in vulnerability detection. Our default hyper-parameter settings achieve optimal results.

V. THREATS AND LIMITATIONS

The first threat to validity comes from the application scenario of our approach. Since our approach is designed for code-centric vulnerability detection and evaluated on a C/C++ dataset, it cannot be used to detected vulnerabilities in IoT applications with only binaries or written in other programming languages. However, benefiting from the language-agnostic nature of CPG, EXVUL can be easily extended to these scenarios. In addition, different from dynamic approaches (e.g., Fuzzing [48], [49]) which are able to detect vulnerabilities in real-time, our approach is statically constructed (i.e., off-line training) and work during code review phase (i.e., on-line detection). Thus, dynamic approaches can serve as a supplement to our approach to construct a more effective detection system throughout the application's life cycle.

The second threat to validity is the computational efficiency of our proposed approach. We employed CPG as a model-understandable intermediate representation to extract vulnerability features at the function-level. Given that the function in a real-world project is commonly large (maybe over 100 lines), graph construction is more complex and time-consuming compared to other code representations such as sequence and syntax tree. In the future, we try to explore a more effective and simpler code representation to further improve our approach.

VI. Related Work

A. DL-Based 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 without the need of hand-crafted vulnerability patterns/rules. Prior works focus on representing source code as sequences and use LSTM-like models to learn the syntactic and semantic information of vulnerabilities [4], [5]. Li et al. [4] proposed VulDeePecker, a *slice-level* vulnerability detection approach which represents source code as sequences and uses RNN (e.g., LSTM and BGRU) to learn the syntactic and semantic information of vulnerabilities. Recently, a large number of works [11], [12], [13], [50] turn to leveraging GNNs to extract rich and well-defined semantics of the program structure from graph representations for downstream vulnerability detection tasks. For example, Zhou et al. [12] proposed Devign, which combines multiple code representations to model vulnerability features and adopts GGNN to learn rich code semantics from structured graph representations, to detect vulnerable functions.

Despite their effectiveness, the *function-level* or *slice-level* detection results are still coarse-grained. To alleviate heavy manual review, MVD [47], [51] and LineVD [52] formalizes vulnerability detection as a fine-grained graph node classification problem to identify suspicious vulnerable statements.

In contrast to these studies constructing an effective detection model based on large-scale vulnerability data with human annotations, we explore the potential of training a well-performing DL model with limited labeled data for IoT vulnerability detection.

B. Explainability on DL-Based Vulnerability Detection

While DL-based code models are remarkably effective in a variety of tasks, one growing concern about their adoption is explainability. The requirement for explainability is more urgent in vulnerability detection because it is hard to establish trust on the system decision from simple binary (vulnerable or benign) results without credible evidence. IVDetect [14] built an additional model based on binary detection results to derive crucial statements that are most relevant to the detected vulnerability as explanations. LineVul [53] leveraged the self-attention mechanism inside the Transformer architecture to locate vulnerable statements for explanation. Chakraborty et al. [13] computed the contribution of each code token toward the prediction. mVulPreter [54] combined the attention weight with the vulnerability probability outputted by the multigranularity detector to compute the importance score for each code slice. In addition, a few efforts simplified the instance to be explained to a minimal set of statements that still preserves the initial model prediction. For example, P2IM [55] borrows *Delta Debugging* [56] to reduce a program sample to a minimal snippet which a model needs to arrive at and stick to its original vulnerable prediction to uncover the model's detection logic.

The main difference between our approach and the above vulnerability explanation approaches is that existing approaches focus only on how to improve the explainability of DL-based vulnerability detection models, ignoring special concerns in security domains. By contrast, EXVul proposes a deviation-aware strategy, which aligns the original code graph with its explanatory substructure in the latent space, and incorporates it into existing explanation framework to obtain more faithful and stable explanations.

VII. Conclusion and Future Work

In this article, we propose EXVul, a novel DL-based approach for effective and explainable IoT vulnerability classification. The key insight of EXVul is that combining the strengths of large-scale unlabeled code corpus and limited labeled data can facilitate training an effective IoT vulnerability detection model, and both faithful and stable explanations can help security practitioners understand the detected vulnerabilities. The experimental results show that EXVul significantly outperforms the state-of-the-art baselines in terms of all metrics. In the near future, we plan to automatically construct a large-scale IoT vulnerability dataset to explore the generalizability of our approach. In addition, we aim to work with our industry partners to deploy EXVul in their proprietary security systems to test its effectiveness in practice.

References

[1] "State of IoT—Spring 2023." 2023. [Online]. Available: https://iot-analytics.com/product/state-of-iot-spring-2023/

[2] "Internet of Things (IoT) security: Challenges and best practices." 2023. [Online]. Available: https://www.apriorit.com/white-papers/513-iot-security

[3] L. Aversano, M. L. Bernardi, M. Cimitile, and R. Pecori, "A systematic review on deep learning approaches for iot security," *Comput. Sci. Rev.*, vol. 40, May 2021, Art. no. 100389.

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

[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] H. K. Dam, T. Tran, T. Pham, S. W. Ng, J. Grundy, and A. Ghose, "Automatic feature learning for predicting vulnerable software components," *IEEE Trans. Softw. Eng.*, vol. 47, no. 1, pp. 67–85, Jan. 2021.

[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, Aug. 2021, Art. no. 106576.

[8] "Flawfinder." 2023, http://www.dwheeler.com/FlawFinder

[9] "Checkmarx." 2023. [Online]. Available: https://www.checkmarx.com/

[10] "Infer." 2023. [Online]. Available: https://fbinfer.com/

[11] H. Wang et al., "Combining graph-based learning with automated data collection for code vulnerability detection," *IEEE Trans. Inf. Forensics Security*, vol. 16, pp. 1943–1958, 2021.

[12] 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. 33rd Annu. Conf. Neural Inf. Process. Syst. (NeurIPS)*, 2019, pp. 10197–10207.

[13] 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.

[14] 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)*, 2021, pp. 292–303.

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

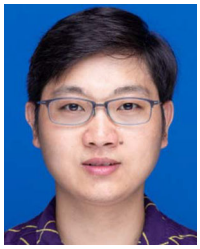
[16] B. Steenhoek, M. M. Rahman, R. Jiles, and W. Le, "An empirical study of deep learning models for vulnerability detection," in *Proc. 45th Int. Conf. Softw. Eng. (ICSE)*, 2018, pp. 2237–2248.

[17] H. K. Dam, T. Tran, and A. Ghose, "Explainable software analytics," in *Proc. 40th Int. Conf. Softw. Eng., New Ideas Emerg. Results*, 2018, pp. 53–56.

[18] S. Cao et al., "A systematic literature review on explainability for machine/deep learning-based software engineering research," 2024, *arXiv:2401.14617*.
[19] Z. Ying, D. Bourgeois, J. You, M. Zitnik, and J. Leskovec, "GNNexplainer: Generating explanations for graph neural networks," in *Proc. 33rd Annu. Conf. Neural Inf. Process. Syst.*, 2019, pp. 9240–9251.
[20] Y. Hu et al., "Interpreters for GNN-based vulnerability detection: Are we there yet?" in *Proc. 32nd ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, 2023, pp. 1407–1419.
[21] T. Chen, S. Kornblith, M. Norouzi, and G. E. Hinton, "A simple framework for contrastive learning of visual representations," in *Proc. 37th Int. Conf. Mach. Learn. (ICML)*, vol. 119, 2020, pp. 1597–1607.
[22] N. Lin, Y. Fu, X. Lin, D. Zhou, A. Yang, and S. Jiang, "CL-XABSA: contrastive learning for cross-lingual aspect-based sentiment analysis," *IEEE ACM Trans. Audio Speech Lang. Process.*, vol. 31, pp. 2935–2946, Jul. 2023.
[23] P. Jain, A. Jain, T. Zhang, P. Abbeel, J. Gonzalez, and I. Stoica, "Contrastive code representation learning," in *Proc. 26th Conf. Empir. Methods Nat. Lang. Process. (EMNLP)*, 2021, pp. 5954–5971.
[24] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 32, no. 1, pp. 4–24, Jan. 2021.
[25] A. Fout, J. Byrd, B. Shariat, and A. Ben-Hur, "Protein interface prediction using graph convolutional networks," in *Proc. 31st Annu. Conf. Neural Inf. Process. Syst.*, 2017, pp. 6533–6542.
[26] X. Shi, B. Li, L. Chen, and C. Yang, "Bi-neighborhood graph neural network for cross-lingual entity alignment," *Knowl. Based Syst.*, vol. 277, Oct. 2023, Art. no. 110841.
[27] J. Cai, B. Li, J. Zhang, and X. Sun, "Ponzi scheme detection in smart contract via transaction semantic representation learning," *IEEE Trans. Reliab.*, early access, Oct. 9, 2023.
[28] D. Luo, W. Cheng, D. Xu, W. Yu, B. Zong, H. Chen, and X. Zhang, "Parameterized explainer for graph neural network," in *Proc. 34th Annu. Conf. Neural Inf. Process. Syst.*, 2020, pp. 19620–19631.
[29] S. Chakraborty, T. Ahmed, Y. Ding, P. T. Devanbu, and B. Ray, "NatGen: generative pre-training by 'naturalizing' source code," in *Proc. 30th ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2022, pp. 18–30.
[30] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Proc. 35th IEEE Symp. Security Privacy (SP)*, 2014, pp. 590–604.
[31] C. Zhang, B. Liu, Y. Xin, and L. Yao, "CPVD: Cross project vulnerability detection based on graph attention network and domain adaptation," *IEEE Trans. Softw. Eng.*, vol. 49, no. 8, pp. 4152–4168, Aug. 2023.
[32] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proc. 27th Annu. Conf. Neural Inf. Process. Syst.*, 2013, pp. 3111–3119.
[33] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discov. Data Min. (KDD)*, 2016, pp. 855–864.
[34] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," in *Proc. 6th Int. Conf. Learn. Represent. (ICLR)*, 2018, pp. 1–12.
[35] P. Khosla et al., "Supervised contrastive learning," in *Proc. 34th Annu. Conf. Neural Inf. Process. Syst.*, 2020, pp. 1–23.
[36] H. Mei, G. Lin, D. Fang, and J. Zhang, "Detecting vulnerabilities in IoT software: New hybrid model and comprehensive data analysis," *J. Inf. Secur. Appl.*, vol. 74, May 2023, Art. no. 103467.
[37] 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. Min. Softw. Repos. (MSR)*, 2020, pp. 508–512.
[38] Y. Li, D. Tarlow, M. Brockschmidt, and R. S. Zemel, "Gated graph sequence neural networks," in *Proc. 4th Int. Conf. Learn. Represent. (ICLR)*, 2016, pp. 1–20
[39] Y. Li, Y. Zuo, H. Song, and Z. Lv, "Deep learning in security of Internet of Things," *IEEE Internet Things J.*, vol. 9, no. 22, pp. 22133–22146, Nov. 2022.
[40] T. Schnake et al., "Higher-order explanations of graph neural networks via relevant walks," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 44, no. 11, pp. 7581–7596, Nov. 2022.
[41] X. Cheng, X. Nie, L. Ningke, H. Wang, Z. Zheng, and Y. Sui, "How about bug-triggering paths?—Understanding and characterizing learning-based vulnerability detectors," *IEEE Trans. Dependable Secur. Comput.*, vol. 21, no. 2, pp. 542–558, Mar./Apr. 2024.
[42] A. Warnecke, D. Arp, C. Wressnegger, and K. Rieck, "Evaluating explanation methods for deep learning in security," in *Proc. 5th IEEE Eur. Symp. Security Privacy*, 2020, pp. 158–174.
[43] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "CodeSearchNet challenge: Evaluating the state of semantic code search," 2019, *arXiv:1909.09436*.
[44] M. Fu, V. Nguyen, C. Tantithamthavorn, T. Le, and D. Phung, "VuLExplainer: A transformer-based hierarchical distillation for explaining vulnerability types," *IEEE Trans. Softw. Eng.*, vol. 49, no. 10, pp. 4550–4565, Oct. 2023.
[45] J. Sun et al., "Silent vulnerable dependency alert prediction with vulnerability key aspect explanation," in *Proc. 45th IEEE/ACM Int. Conf. Softw. Eng.*, 2023, pp. 970–982.
[46] C. Ni, X. Yin, K. Yang, D. Zhao, Z. Xing, and X. Xia, "Distinguishing look-alike innocent and vulnerable code by subtle semantic representation learning and explanation," in *Proc. 31th ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2023, pp. 1611–1622.
[47] 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. 44th IEEE/ACM Int. Conf. Softw. Eng. (ICSE)*, 2022, pp. 1456–1468.
[48] S. Cao et al., "ODDFUZZ: Discovering java deserialization vulnerabilities via structure-aware directed greybox fuzzing," in *Proc. 44th IEEE Symp. Security Privacy (SP)*, 2023, pp. 2726–2743.
[49] S. Cao et al., "Improving java deserialization gadget chain mining via overriding-guided object generation," in *Proc. 45th IEEE/ACM Int. Conf. Softw. Eng. (ICSE)*, 2023, pp. 397–409.
[50] X. Cheng, H. Wang, J. Hua, G. Xu, and Y. Sui, "DeepWukong: Statically detecting software vulnerabilities using deep graph neural network," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 3, pp. 38, pp. 1–33, 2021.
[51] S. Cao et al., "Learning to detect memory-related vulnerabilities," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 2, pp. 43, pp. 1–35, 2024.
[52] D. Hin, A. Kan, H. Chen, and M. A. Babar, "LineVD: Statement-level vulnerability detection using graph neural networks," in *Proc. 19th IEEE/ACM Int. Conf. Min. Softw. Repos. (MSR)*, 2022, pp. 596–607.
[53] M. Fu and C. Tantithamthavorn, "LineVul: A transformer-based line-level vulnerability prediction," in *Proc. 19th IEEE/ACM Int. Conf. Min. Softw. Repos. (MSR)*, 2022, pp. 608–620.
[54] D. Zou, Y. Hu, W. Li, Y. Wu, H. Zhao, and H. Jin, "mVulPreter: A multi-granularity vulnerability detection system with interpretations," *IEEE Trans. Dependable Secure Comput.*, early access, Aug. 22, 2022.
[55] S. Suneja, Y. Zheng, Y. Zhuang, J. A. Laredo, and A. Morari, "Probing model signal-awareness via prediction-preserving input minimization," in *Proc. 29th ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2021, pp. 945–955.
[56] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Trans. Softw. Eng.*, vol. 28, no. 2, pp. 183–200, Feb. 2002.

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

His research interests include software security and deep learning. Some of his publications have been published in the top-tier conferences (e.g., ICSE, S&P) and journals (e.g.,*ACM Transactions on Software Engineering and Methodology*).



Xiaobing Sun (Member, IEEE) received the B.S. degree in computer science and technology from 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 a Professor with the School of Information Engineering, Yangzhou University, Yangzhou, China. He has been authorized more than 20 patents, and authored and coauthored more than 80 papers in referred international journals and conferences. His research interests include software maintenance and evolution, software repository mining, and intelligence analysis.

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

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

Di Wu (Member, IEEE) received the Ph.D. degree from University of Technology Sydney, Ultimo, NSW, Australia.

He is a Lecturer with the School of Mathematics, Physics, and Computing, University of Southern Queensland, Toowoomba, QLD, Australia, and a Visiting Fellow with the School of Computer Science, University of Technology Sydney (UTS), Sydney, NSW, Australia. Prior to that, he was a Researcher with the Australian Institute for Machine Learning and School of Computer Science, University of Adelaide, Adelaide, SA, Australia. He has more than ten years of experience in Research and Development and Academia. He has substantial industry experience in large project management, software development, and large system maintenance experience while working on various projects with China Telecom (Global 500), Shanghai, China. He has published over 20 papers in refereed books, conferences, and journals. Previous to this, he was an Associate Research Fellow, Artificial Intelligence with Deakin Blockchain Innovation Lab, School of Information Technology, Deakin University, Melbourne, VIC, Australia, and worked as a Postdoctoral Fellow with the School of Computer Science, UTS. His research area focuses on applying AI on edge devices and AI applications.

Dr. Wu has served as a Special Session Chair for IJCNN. He also serves as a reviewer for many high-quality academic conferences and journals, such as CoRL, PR, and IEEE TRANSACTIONS ON EMERGING TOPICS IN COMPUTATIONAL INTELLIGENCE.

Jiale Zhang (Member, IEEE) received the Ph.D. degree in computer science and technology from the College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China, in 2021.

He is currently an Associate Professor with the School of Information Engineering, Yangzhou University, Yangzhou, China. He has published over 40 research papers in refereed international conferences and journals, such as IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING, IEEE TRANSACTIONS ON INFORMATION FORENSICS AND SECURITY, IEEE TRANSACTIONS ON INDUSTRIAL INFORMATICS, IEEE INTERNET OF THINGS JOURNAL, *Computers and Security*, *Journal of Systems and Software*, IEEE ICC, and IEEE Globecom. His research interests are mainly federated learning, AI security, and blockchain security.

Yan Li (Senior Member, IEEE) received the Ph.D. degree from Flinders University, Adelaide, Australia.

She is currently a Professor of Computer Science with the Faculty of Health, Engineering and Sciences, University of Southern Queensland, Toowoomba, QLD, Australia. Her research interests are in the areas of signal and image processing, biomedical engineering, artificial intelligence, big data analytics, and computer networking technologies.

Tom H. Luan (Senior Member, IEEE) received the B.E. degree in electrical and computer engineering from Xi'an Jiaotong University, Xi'an, Shaanxi, China, in 2004, the M.Phil. degree in electrical and computer engineering from The Hong Kong University of Science and Technology, Hong Kong, in 2007, and the Ph.D. degree in electrical and computer engineering from the University of Waterloo, Waterloo, ON, Canada, in 2012.

He is with the School of Cyber Science and Engineering, Xi'an Jiaotong University. He has authored or coauthored more than 150 peer-reviewed papers in journal and conferences. His research interests include content distribution and media streaming in vehicular ad hoc networks, peer-to-peer networking, protocol design, performance evaluation of digital network, and edge computing.

Dr. Luan was the recipient of the 2017 IEEE VTS Best Land Transportation Best Paper Award and the IEEE ICCS 2018 Best Paper Award.

Longxiang Gao (Senior Member, IEEE) received the Ph.D. in computer science from Deakin University, Melbourne, VIC, Australia.

He is currently a Professor with Shandong Computer Science Center, Qilu University of Technology (Shandong Academy of Sciences), Jinan, China. He is also an Adjunct Professor with the University of Southern Queensland, Toowoomba, QLD, Australia. He was a Senior Lecturer with the School of Information Technology, Deakin University and a Postdoctoral Research Fellow with IBM Research and Development, Melbourne. He has over 130 publications, including patent, monograph, book chapter, journal and conference papers. Some of his publications have been published in the top venue, such as IEEE TRANSACTIONS ON MOBILE COMPUTING, IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, IEEE INTERNET OF THINGS JOURNAL, IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING, IEEE TRANSACTIONS ON VEHICULAR TECHNOLOGY, IEEE TRANSACTIONS ON COMPUTATIONAL SOCIAL SYSTEMS, IEEE TRANSACTIONS ON INDUSTRIAL INFORMATICS, IEEE TRANSACTIONS ON NETWORK SCIENCE AND ENGINEERING, IEEE TRANSACTIONS ON WIRELESS COMMUNICATIONS, and *ACM Computing Surveys*. He has been chief investigator for more than 20 research projects (the total awarded amount is over $5 million), from pure research project to contracted industry research. His research interests include fog/edge computing, blockchain, data analysis, and privacy protection.