Learning to Detect Memory-related Vulnerabilities

SICONG CAO, XIAOBING SUN, and LILI BO, School of Information Engineering, Yangzhou University, China
RONGXIN WU, School of Informatics, Xiamen University, China
BIN LI and XIAOXUE WU, School of Information Engineering, Yangzhou University, China
CHUANQI TAO, College of Computer Science and Technology/College of Artificial Intelligence, Nanjing University of Aeronautics and Astronautics, China
TAO ZHANG, School of Computer Science and Engineering, Macau University of Science and Technology, China
WEI LIU, School of Information Engineering, Yangzhou University, China

Memory-related vulnerabilities can result in performance degradation or even program crashes, constituting severe threats to the security of modern software. Despite the promising results of deep learning (DL)-based vulnerability detectors, there exist three main limitations: (1) rich contextual program semantics related to vulnerabilities have not yet been fully modeled; (2) multi-granularity vulnerability features in hierarchical code structure are still hard to be captured; and (3) heterogeneous flow information is not well utilized. To address these limitations, in this article, we propose a novel DL-based approach, called *MVD+*, to detect memory-related vulnerabilities at the statement-level. Specifically, it conducts both intraprocedural and interprocedural analysis to model vulnerability features, and adopts a hierarchical representation learning strategy, which performs syntax-aware neural embedding within statements and captures structured context information across statements based on a novel Flow-Sensitive Graph Neural Networks, to learn both syntactic and semantic features of vulnerable code. To demonstrate the performance, we conducted extensive experiments against eight state-of-the-art DL-based approaches as well as five well-known static analyzers on our constructed dataset with 6,879 vulnerabilities in 12 popular C/C++ applications. The experimental results confirmed that *MVD+* can significantly outperform current state-of-the-art baselines and make a great trade-off between effectiveness and efficiency.

This work is supported by the National Natural Science Foundation of China (Grants No. 62202414, No. 61972335, No. 62002309, and No. 62272400); the Six Talent Peaks Project in Jiangsu Province (No. RJFW-053); the Jiangsu "333" Project and Yangzhou University Top-level Talents Support Program (2019); Postgraduate Research & Practice Innovation Program of Jiangsu Province (KYCX22_3502); the Macao Science and Technology Development Fund (Grants No. 0047/2020/A1 and No. 0014/2022/A); the Open Funds of State Key Laboratory for Novel Software Technology of Nanjing University (Grant No. KFKT2022B17); the China Scholarship Council Foundation (Grants No. 202209300005 and No. 202308320436); and the grant from Huawei.

Authors' addresses: S. Cao, X. Sun (Corresponding author), L. Bo (Corresponding author), B. Li, X. Wu, and W. Liu, School of Information Engineering, Yangzhou University, 196 Huayang West Road, Yangzhou 225009, China; e-mails: {DX120210088, xbsun, lilibo, lb, xiaoxuewu, weiliu}@yzu.edu.cn; R. Wu, School of Informatics, Xiamen University, 422 Siming South Road, Xiamen 361005, China; e-mail: wurongxin@xmu.edu.cn; C. Tao, College of Computer Science and Technology/College of Artificial Intelligence, Nanjing University of Aeronautics and Astronautics, 29 Jiangjun Road, Nanjing 211106, China; e-mail: taochuanqi@nuaa.edu.cn; T. Zhang, School of Computer Science and Engineering, Macau University of Science and Technology, Avenida Wai Long, Macao 999078, China; e-mail: tazhang@must.edu.mo.

1049-331X/2023/12-ART43 $15.00
https://doi.org/10.1145/3624744

CCS Concepts: • **Software and its engineering → Software reliability**; • **Security and privacy → Software security engineering;**

Additional Key Words and Phrases: Statement-level vulnerability detection, abstract syntax tree, graph neural networks, flow analysis

ACM Reference format:
Sicong Cao, Xiaobing Sun, Lili Bo, Rongxin Wu, Bin Li, Xiaoxue Wu, Chuanqi Tao, Tao Zhang, and Wei Liu. 2023. Learning to Detect Memory-related Vulnerabilities. *ACM Trans. Softw. Eng. Methodol.* 33, 2, Article 43 (December 2023), 35 pages.
https://doi.org/10.1145/3624744

1 INTRODUCTION

Memory-related vulnerabilities, such as buffer overflows, use-after-free, and memory leaks, and so on, are prevalent in software systems written in programming languages with manual memory management mechanism like C/C++ [79]. Such security-critical weaknesses can result in performance degradation or even program crashes, severely threatening the security of modern software [17, 87]. A recent study found that around 40% of vulnerabilities reported in **Common Vulnerabilities and Exposures (CVE)** [21] are related to memory [39].

Existing Efforts. Many static analysis-based approaches [20, 27, 34, 45, 71, 78] have been proposed to detect memory-related vulnerabilities and shown their effectiveness. They use predefined vulnerability patterns to search for improper memory usage [63]. However, manually writing vulnerability patterns is highly dependent on expert knowledge, which is a labor-intensive process and subject to error-prone as well as time-consuming tasks. Recently, benefiting from the powerful performance of **Deep Learning (DL)**, a number of approaches [12, 15, 25, 47, 49, 50, 77, 84, 98, 99] have been proposed to leverage DL models to learn program semantics to identify potential software vulnerabilities. Figure 1 presents the typical DL-based vulnerability detection pipeline. It operates in three phases, namely, representation learning phase, training phase, and detection phase. In the representation learning phase, training samples (both vulnerable and non-vulnerable code) are converted into abstract code representations (e.g., trees [64] and graphs [93]) containing rich syntactic and semantic information, and embedded as compact and uniform length feature vectors through DL models, such as **Recurrent Neural Network (RNN)** [38] and **Graph Neural Network (GNN)** [92]. Then, these high-dimensional feature representations are used to train a detection model that can demarcate the vulnerable examples from non-vulnerable examples. Code snippets to be analyzed will be fed into the well-trained detection model to predict whether they are prone to vulnerabilities or not. Compared with traditional static analysis-based approaches, DL-based approaches can *automatically* extract implicit vulnerability patterns from prior vulnerable code instead of requiring expert involvement [94].

Limitations. Despite their tremendous progresses, the performance of existing DL-based general vulnerability detectors is still far away from the satisfactory when applied to detecting memory-related vulnerabilities due to the following three main limitations. (1) The first is that contextual program semantics related to vulnerabilities are not well captured. Recent works [19, 75] have reported that interprocedural program contexts (e.g., control- and data-flows) cover comprehensive code features and are beneficial to memory-related vulnerability detection. Unfortunately, in the representation learning phase (as shown in Figure 1), almost all DL-based vulnerability detection approaches perform intraprocedural analysis at the *function-level* [12, 15, 84, 99] or *slice-level* [19, 49, 50, 101] to extract syntactic and semantic information of vulnerable code. Due to the complicated program logic in real-world scenarios, interprocedural

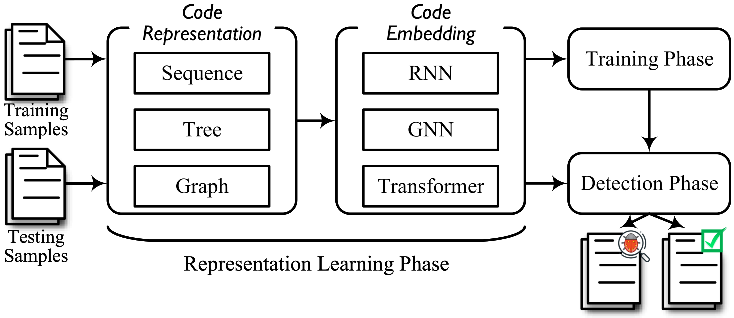


Fig. 1. DL-based vulnerability detection pipeline.

operations (e.g., releasing the heap memory allocated by another function) are common and inevitable. As a result, some critical vulnerability features may be absent in a vulnerable snippet, resulting in *unsound* (i.e., high *false-negative rates*) and *imprecise* (i.e., high *false-positive rates*) detection results. (2) The second limitation lies in that multi-granularity vulnerability features are treated equally although they should not be. State-of-the-art DL-based vulnerability detection approaches usually adopt single neural network architecture to extract rich vulnerability features from hybrid abstract representations (e.g., **Code Property Graph (CPG)** [93]) of vulnerable code. Nonetheless, given the inherent hierarchical structure of source code (i.e., tokens forming statements, and statements forming snippets), such single-architecture neural models may be ill-suited for capturing different levels (e.g., token-level buffer size useful for detecting overflow-like vulnerabilities and statement-level execution order effective in control flow-related memory vulnerabilities) of program semantics. For example, limited to the long-term dependency problem [2], popular GNN-based vulnerability detection models fail to propagate message between distant nodes in normally large AST-based representations, leading to the information loss in inter-level communication. (3) The third concerns the underutilization of flow information. For simplicity, previous studies either regard hybrid code representations as homogeneous structures, which neglect the semantic type information of different edges [19], or categorize them in a coarse-grained manner (e.g., control- and data-flow edges) [99], obstructing the detection model from sufficiently capturing the vulnerability features to a certain degree. A straightforward instance is that receiving a normal pointer variable (non-vulnerable) is obviously not the same as receiving a pointer variable, which points to the memory just released (vulnerable).

Our Solution. In this article, we propose a novel DL-based vulnerability detection approach, named *MVD+*, targeted at memory-related vulnerabilities, that can learn comprehensive program semantics and locate suspicious statements related to vulnerabilities. First, instead of simply performing intraprocedural analysis to model local contextual features of vulnerable code, *MVD+* conducts interprocedural analysis to build **Abstract Syntax Tree (AST)** [64] and **System Dependence Graph (SDG)** [72], which explicitly maintain global syntactic and semantic contexts across functions, and performs forward and backward slicing from the program point of interest (i.e., slicing criteria) based on interprocedural control- and data-flows in SDG to extract vulnerability-related code snippets. Second, to preserve the syntactic and semantic information of source code for training an accurate detection model, *MVD+* adopts a hierarchical representation learning strategy, which performs syntax-aware neural embeddings for each statement of source code with a tree-based neural network and feeds these unstructured (statement embeddings) and structured information (control- and data-flows between statements) into a novel ***Flow-sensitive Graph Neural Networks*** **(FS-GNN)**, to learn the distributed representation of statements. By hierarchically processing code components at different levels (i.e., tokens and statements) with a combination of tree-based and graph-based neural network architectures, *MVD+* can effectively learn multi-granularity features (i.e., lexical syntactic information within statements and different flow semantics between statements) of vulnerable code. Third, in the detection phase, *MVD+* formalizes the detection of vulnerable statements as a node classification problem, which receives the graph representation of a code snippet (in which nodes represent statements and edges indicate their relations) as input and outputs corresponding node labels (i.e., vulnerable or not).

Evaluation. To evaluate the effectiveness of our proposed *MVD+*, we crawl and build a new benchmark, since existing benchmarks [27, 63] targeting at memory-related vulnerabilities are not sufficient for training DL models. The experimental results on both function-level (i.e., predict whether a function is vulnerable) and statement-level (i.e., locate which statements in a function are vulnerable) vulnerability detection tasks demonstrate that *MVD+* can significantly outperform state-of-the-art static analysis-based and DL-based detectors. Specifically, *MVD+* outperforms current DL-based and static analysis-based approaches significantly with 9.47–22.46% higher Precision and 12.64–28.02% higher Recall.

In summary, our main contributions are as follows:

- We propose *MVD+*, a novel DL-based approach that hierarchically captures syntactic and semantic information of source code by leveraging tree-based and graph-based neural networks to effectively support memory-related vulnerability detection at the statement-level.
- We design a novel Flow-Sensitive Graph Neural Network, called FS-GNN, to jointly consider heterogeneous flow information for capturing the patterns of vulnerable code.
- We construct a large-scale benchmark repository targeted at memory-related vulnerabilities. The vulnerability repository, called *MemoryVul* [59], is composed of 37,735 function-level code snippets (8,163 of which are vulnerable) with statement-level ground-truths (i.e., which statements in a function are vulnerable) from 12 well-known C/C++ projects.
- We compare *MVD+* with five static memory analyzers and eight DL-based vulnerability detectors (five for function-level detection and three for statement-level detection). Experimental results show that *MVD+* outperforms the existing approaches on both function-level and statement-level vulnerability detection.

Article Organization. The remainder of this article is organized as follows. Section 2 introduces the background knowledge related to the proposed approach. Then, we describe the details about *MVD+* in Section 3. Section 4 presents the experimental setup, followed by the evaluation results in Section 5. Section 6 gives a case study about *MVD+* and threats to the validity. Section 7 surveys the related work. Finally, Section 8 summarizes this article. This article extends our prior publication [13] presented at the 44th International Conference on Software Engineering (ICSE'22), which presented our *MVD* approach to detect memory-related vulnerabilities at the statement-level based on FS-GNN. In this follow-up work, we focus on further improving model performance and experiment solidity. New materials with respect to the conference version include:

- **We further improve the approach by proposing a hierarchical representation learning strategy to capture multi-granularity semantic information of vulnerable code**. In the conference version, the proposed approach adopts Doc2Vec [44] to transform statements into low-dimensional feature vectors as initial representations for graph learning, which has a negative influence on capturing fine-grained token-level semantics of statements. Learning program semantics with the strategies proposed in this article improves the performance of *MVD* [13] on both function-level and statement-level vulnerability detection tasks. The details are presented in Section 3.2.

- **We build a larger and more diverse benchmark repository, called *MemoryVul*, to improve the quality of training data.** In the conference version, the dataset we constructed is small-scale (only containing 4,353 vulnerable samples) and imbalanced (CWE-119 and CWE-476 account for 60%), which may threaten the effectiveness of the pre-trained detection model in practice. In this extended version, we re-train *MVD+* and baselines on the enlarged *MemoryVul* benchmark for evaluation. It now contains 5,266 synthetic and 1,613 real-world memory-related vulnerabilities (58% larger than the previous one). For each vulnerability, we provided the vulnerable code version, the fixed code version, the vulnerable code line, and the vulnerability type. We plan to open-source our dataset and approach [59] to facilitate other research works on memory-related vulnerabilities.
- **Additional experimental comparison with recently proposed statement-level DL-based detectors are provided.** In the conference version, we only compare our approach with three DL-based approaches. As of this extended version, multiple statement-level approaches have been proposed. Thus, we adopt two fine-grained metrics, **Mean First Ranking (MFR)** and **Mean Average Ranking (MAR)**, to measure and demonstrate the statement-level prediction performance of the proposed approach compared with these newly presented baselines. The details are provided in Section 4.5.1.

2 PRELIMINARIES

In this section, we first introduce several basic concepts relevant to DL-based vulnerability detection, the motivation of our approach, and the DL models used in this article.

2.1 Definitions

Definition 1 (Code Snippet, Statement, and Token). Following the definition of Reference [101], a *code snippet* C is an ordered set of statements $\{s_1, s_2, \ldots, s_m\}$, where s_i $(1 \leq i \leq m)$ denotes the ith statement in the code snippet C and m is the total number of code lines. A *statement* s_i is an ordered set of tokens $\{t_{i,1}, t_{i,2}, \ldots, t_{i,n}\}$, where $t_{i,j}$ $(1 \leq j \leq n)$ denotes the jth token in the statement s_i and n is the total number of code tokens. A *token* $t_{i,j}$ is a piece of code text that can be an identifier, operator, constant, keyword, and anything else that can be extracted by a lexical parser [37].

Definition 2 (Abstract Syntax Tree). AST is a kind of data structure that organizes a code snippet C as a tree to represent the syntactic features of source code [64]. Formally, an AST for a code snippet C is a tuple $(V_A, E_A, \lambda_A, \mu_A)$, where V_A denotes a set of tree nodes corresponding to constructs or symbols of source code, E_A are the corresponding tree edges labeled as AST edges by the labeling function λ_A, and μ_A is a function that maps a node to an associated value (the operator or operand the node represents).

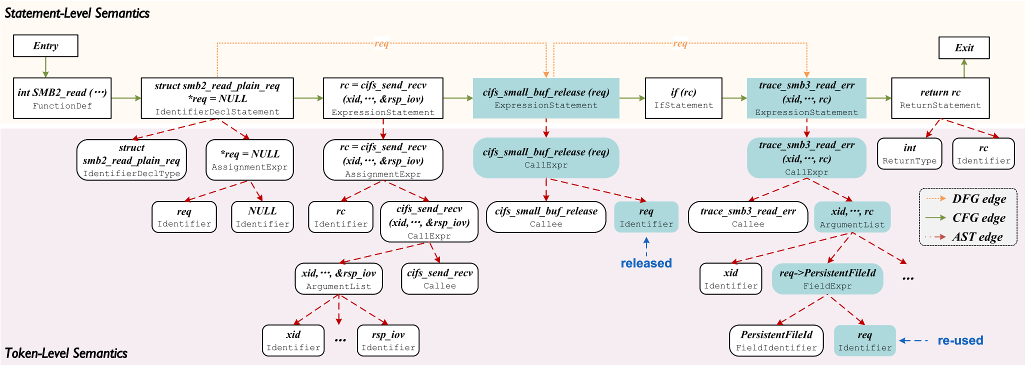
Definition 3 (System Dependence Graph). SDG [72] is derived from a set of ***Program Dependence Graph*** **(PDG)** [30] connected by the caller-callee relations. Given a code snippet $C = \{s_1, s_2, \ldots, s_m\}$, a PDG $\mathcal{G} = (V_P, E_P, \lambda_P, \mu_P)$ represents data and control dependencies among statements, where $V_P \subseteq V_A$ corresponds to statement nodes in AST, E_P is a set of directed edges labeled by the function $\lambda_P : E_P \rightarrow \Sigma_P$ where Σ_P corresponds to semantic associations (control and data dependencies) between statement s_u and s_v $(1 \leq u < v \leq m)$.

Definition 4 (Code Slice). Consider a code snippet $C = \{s_1, s_2, \ldots, s_m\}$, a PDG $\mathcal{G}$, and a statement s_i $(1 \leq i \leq m)$ containing the program point of interest p_i, a *code slice* $\mathcal{S}' = \mathcal{S}_f \cup \mathcal{S}_b$ is an ordered set of statements in C, where $\mathcal{S}_f$ is a set of statements obtained by traversing the PDG $\mathcal{G}$ forward from the program point p_i, and $\mathcal{S}_b$ is collected by traversing the PDG backward from p_i to all reachable statements [88].

```
1  /* fs/cifs/smb2pdu.c */
2  int
3  SMB2_read (const unsigned int xid, struct cifs_io_parms *io_parms,
4             unsigned int *nbytes, char **buf, int *buf_type)
5  {
6      struct smb2_read_plain_req *req = NULL;
7      rc = cifs_send_recv(xid, ses, &rqst, &resp_buftype, flags, &rsp_iov);
8  -   cifs_small_buf_release(req);
9      if (rc) {
10             trace_smb3_read_err(xid, req->PersistentFileId,
11                                 io_parms->tcon->tid, ses->Suid,
12                                 io_parms->offset, io_parms->length,
13                                 rc);
14 +   cifs_small_buf_release(req);
15     return rc;
16 }

17 /* fs/cifs/misc.c */
18 void
19 cifs_small_buf_release(void *buf_to_free)
20 {
21     if (buf_to_free == NULL) {
22             cifs_dbg(FYI, "Null buffer passed to cifs_small_buf_release\n");
23             return;
24     }
25     mempool_free(buf_to_free, cifs_sm_req_poolp);
26     return;
27 }
```

(a) A UAF vulnerability (CVE-2019-15920) in *Linux Kernel*.



(b) The simplified CPG of the vulnerable function `SMB2_read()` in Figure 2a.

Fig. 2. Motivating example.

Definition 5 (Statement-level Vulnerability Detection). Statement-level vulnerability detection can be formalized as a binary classification task, i.e., learning to determine whether a statement s_i in a given code snippet C is vulnerable or not. Let a sample of data can be defined as $((s_i, l_i | s_i \in \mathcal{S}, l_i \in \mathcal{L}), i \in \{1, 2, \ldots, m\}$, where $\mathcal{S}$ denotes the set of statements in the code snippet C, $\mathcal{L} = \{0, 1\}^m$ represents the label set with 1 for vulnerable and 0 otherwise, and m is the number of statements in C. The goal of neural networks in vulnerability detection is to fit a function $f : \mathcal{S} \rightarrow \mathcal{L}$ to predict whether a statement is vulnerable or not.

2.2 Motivation

Figure 2(a) shows a typical ***Use-after-free*** **(UAF)** vulnerability CVE-2019-15920 [23] in *Linux Kernel* [51]. The vulnerable function `SMB2_read()` (lines 2-16) is simplified for a clear illustration. We can observe that the memory space pointed by the pointer `req` (line 6) is released in advance by the function `mempool_free()` (line 25) in the statement `cifs_small_buf_release(req)` (line 8), while it is still used at line 10. This problem may result in a system crash or other security impacts. Despite the support of precise interprocedural analysis, this vulnerability may not be detected by static memory detectors if they do not know that the function `cifs_small_buf_release()` behaves like a `free()` function [4, 33].

Inspired by the good performance of DL in learning program semantics, it is natural to train a classification model to detect whether a given code snippet is vulnerable or not. These approaches generally combine multiple abstract representations (e.g., CPG [93], which is a widely used code representation composed of AST, **Control Flow Graph (CFG)** and PDG) to represent source code and design deep neural networks to learn the syntactic and semantic features of vulnerabilities. However, such a multi-granularity (covering token-level tree nodes and statement-level graph nodes) and heterogeneous (nodes are connected by different types of relations, such as control- and data-dependencies) code representation will complicate embedded graphs and learn irrelevant features [99].

Taking the simplified CPG of the vulnerable function `SMB2_read` illustrated in Figure 2(a) as an example. The propagation path of the tainted variable `req` is highlighted in blue. To learn the vulnerability logic that the released variable `req` is re-used again, the neural network should be able to capture the long-range information passing (involving seven-hop neighbors) from the released `req` at line 8 to the re-used one at line 10. Unfortunately, GNN models are prone to the long-term dependency problem [2], resulting that they fail to propagate message between distant nodes in normally large AST-based representations. As observed by the recent work [90], the overall performance of the popular CPG-based vulnerability detection approaches degrades sharply as the number of nodes increases. To alleviate the inherent limitation, some works [19, 36] adopt coarse-grained abstract representations such as PDG, which models the semantic information of source code at the statement-level, as basic components to extract vulnerability features. However, such global statement embeddings may hard to capture local vulnerability semantics (e.g., `req`) within fine-grained code tokens.

Inspired by the hierarchical structure of source code (i.e., coarse-grained code components are represented by combining fine-grained ones together), we adopt a hierarchical representation learning strategy, which combines tree-based and graph-based neural networks to, respectively, capture the syntactic and semantic information of source code, for multi-granularity vulnerability feature extraction. In addition, to learn precise code semantics from heterogeneous representations, we propose FS-GNN, a novel graph neural network that jointly embeds both unstructured node embeddings and structured flow relations of neighbors to update the embedding of the central node, to better preserve context information of statements.

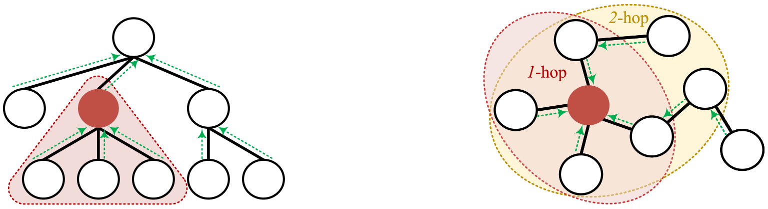
2.3 Tree-based Neural Networks

Tree-based Neural Networks (TNNs) have been proposed to learn code embedding from the AST of a program to support downstream **software engineering (SE)** tasks, such as code clone detection [86, 91, 95, 96] and method name prediction [3, 9, 61]. As illustrated in Figure 3(a), TNNs update node representations by receiving information from their children.

TBCNN [61] is the first deep neural network for program language processing. It uses a tree-based convolution kernel, which applies a set of fixed-depth feature detectors by sliding over the entire AST, to learn node embeddings in a bottom-up way, and introduces *continuous binary trees* to cope with ASTs of different sizes and shapes by dynamic pooling. The core convolution layer of TBCNN can be formalized as follows:

$$\boldsymbol{y} = tanh\left(\sum_{i=1}^{n} \boldsymbol{W}_{conv,i} \cdot \boldsymbol{x_i} + \boldsymbol{b}_{conv}\right), \tag{1}$$

where $\boldsymbol{x}_i$ $(1 \leq i \leq n)$ is the vector representation of node i within each sliding window, $\boldsymbol{W}_{conv,i}$ are the weight matrices, and $\boldsymbol{b}_{conv}$ is the bias term.



(a) Tree-based Neural Network

(b) Graph-based Neural Network

Fig. 3. The comparison between tree-based and graph-based neural networks, where the green dotted arrow represents the direction of node information propagation and the dash area is the context of target node shaded in red.

ASTNN [96] is a novel AST-based neural network for source code representation. To solve the the gradient vanishing and long-term dependency problems experienced by previous approaches, ASTNN splits large ASTs into a sequence of small statement trees, and computes the vector representations of statements by recursively encoding multi-way statement trees. The vector representation of the non-leaf node n in a statement tree is computed by the following equation:

$$\boldsymbol{h} = \sigma\left(\boldsymbol{W}_n^{\top}\boldsymbol{v}_n + \sum_{i\in[1,C]} \boldsymbol{h}_i + \boldsymbol{b}_n\right), \tag{2}$$

where $\boldsymbol{W}_n$ is the weight matrix, $\boldsymbol{v}_n$ is the lexical vector of node n, $\boldsymbol{h}_i$ is the hidden state for each children node $i \in [1, C]$ of the non-leaf node n, $\boldsymbol{b}_n$ is the bias term, σ is the activation function such as $tanh$, and $\boldsymbol{h}$ is the updated hidden state.

Finally, the final representation of a statement tree is obtained by a max pooling layer.

Tree-LSTM [80] is another popular tree-based code embedding model. Different from standard LSTM, the gating vectors and memory cell updates of Tree-LSTM are dependent on the states of possibly many child units for state updating across the tree structure. Additionally, instead of a single forget gate, Tree-LSTM contains one forget gate f_{jk} for each child k (as calculated in Equation (3)), which allows it to selectively incorporate information from each child:

$$f_{jk} = \sigma\left(\boldsymbol{W}_{x_j}^{(f)} + \boldsymbol{U}_{h_k}^{(f)} + \boldsymbol{b}^{(f)}\right), \tag{3}$$

where $k \in C(j)$, $C(j)$ denotes the set of children of node j.

Inspired by the promising results on multiple code-related tasks, we propose that the ability to capture the lexical and syntactic information of programs makes TNN a suitable neural architecture to encode code statements. These feature representations can be used as initial node embeddings of GNNs to learn code semantics (e.g., control- and data-dependence) across statements. In this work, we make use of ASTNN due to its effectiveness in learning statement-level vector representations and alleviating long-term dependency problem.

2.4 Graph-based Neural Networks

Due to the outstanding ability in learning program semantics, GNNs have been applied to a variety of SE tasks (e.g., variable misuse prediction [1], vulnerability detection [99], and bug localization [55]) and have achieved great breakthroughs. As shown in Figure 3(b), modern GNNs follow a neighborhood aggregation scheme, where the representation of a node is updated by iteratively aggregating representations of its k-hop neighbors, to capture the structural information of graphs.

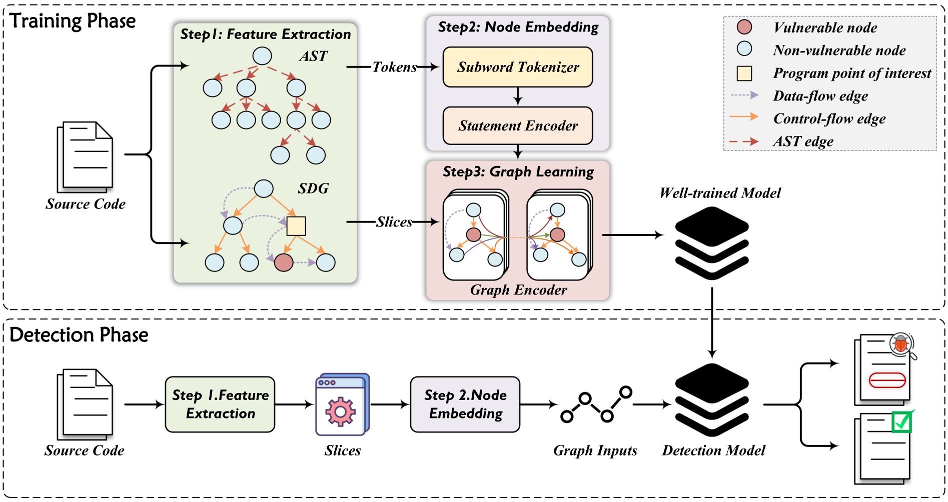


Fig. 4. Overview of *MVD+*.

This procedure can be formulated by

$$\boldsymbol{h}_v^{(t)} = \sigma\left(\boldsymbol{h}_v^{(t-1)}, AGG^{(t)}\left(\left\{\boldsymbol{h}_u^{(t-1)} : u \in \mathcal{N}(v)\right\}\right)\right), \tag{4}$$

where $\boldsymbol{h}_v^{(t)}$ is the feature representation of node v at the tth iteration, $u \in \mathcal{N}(v)$ is the neighbors of v, and $AGG(\cdot)$ and $\sigma(\cdot)$ denote aggregation (e.g., *MEAN*) and activation (e.g., *ReLU*) functions for node feature computation.

According to different goals, the final node representation $h_v^{(T)}$ can be used for graph classification, node classification, and link prediction [92].

Graph Classification. Given a graph $G_i = (V, E, X) \in \mathcal{G}$ and a set of graph labels $\mathcal{L} = \{l_1, \ldots, l_m\}$, where each node $v \in V$ is represented by a real-valued feature vector $\boldsymbol{x}_v \in X$ and m denotes the number of graph labels, graph classification aims to learn a mapping function $f : \mathcal{G} \rightarrow \mathcal{L}$ to predict the label of the ith graph G_i.

Node Classification. Given a graph $G_j = (V, E, X) \in \mathcal{G}$ and its node label set $\mathcal{L} = \{l_1, \ldots, l_n\}$, node classification aims to learn a mapping function $g : \mathcal{V} \rightarrow \mathcal{L}$ to predict the label of node v.

Link Prediction. Given node u and node v, link prediction aims to predict the probability of connection between node u and node v by $y_{u,v} = \phi(\boldsymbol{h}_u^{(k)}, \boldsymbol{h}_v^{(k)})$, where $\boldsymbol{h}_u^{(k)}$ and $\boldsymbol{h}_v^{(k)}$ are the node representations after k iterations of aggregation and $\phi(\cdot)$ refers to the composition operator such as *Inner Production*.

Existing DL-based approaches parse a program into a structured graph with multiple types of edges and view vulnerability detection as a graph classification task. However, the rich contextual program semantics related to vulnerabilities have not yet been fully utilized, and the detection granularity is still coarse-grained. Hence, to overcome the limitations of existing approaches, we have proposed a novel FS-GNN. FS-GNN jointly embeds both statement embedding and flows information to capture sensitive contextual information for semantic learning. Section 3.3 explains the detailed architecture of FS-GNN.

3 OUR APPROACH: *MVD+*

Figure 4 shows the overview of *MVD+*. Overall, it consists of two phases: training phase and detection phase.

The training phase includes three steps. In **step 1**, *MVD+*, respectively, performs intraprocedural and interprocedural analysis to build AST and SDG to capture rich syntactic and semantic information of source code. To reduce irrelevant semantic noise introduced by the distribution of vulnerable and non-vulnerable statements in the training data, *MVD+* conducts program slicing [76, 88] from the program points of interest. To preserve rich syntactic and semantic information of source code, *MVD+* adopts a hierarchical representation learning strategy (**step 2** and **step 3**) for code embedding. In **step 2**, *MVD+* leverages **Byte Pair Encoding (BPE)** [69] to build a subword tokenizer for token embedding, and constructs a statement encoder to transform each statement into low-dimensional vector representations to capture both lexical and syntactic information of source code. In **step 3**, we designs a novel FS-GNN to jointly embed nodes and edges in code slices to learn implicit vulnerability patterns and re-balance node labels distribution. Finally, a well-trained model is produced for memory-related vulnerability detection at the *statement-level*.

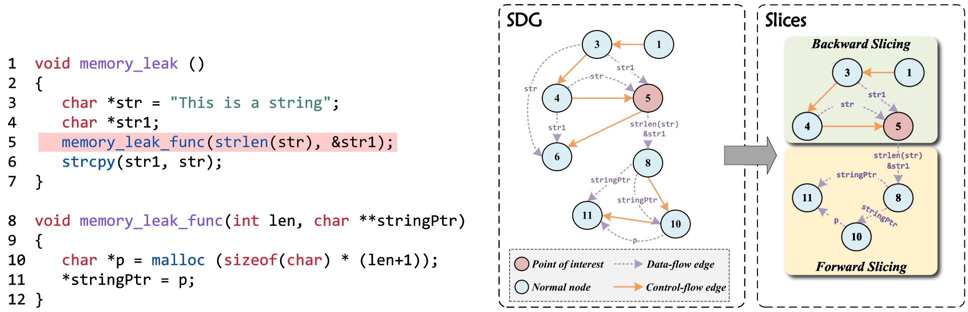
In the detection phase, *MVD+* first splits the target program into functions and repeats feature extraction (**step 1**) and node embedding (**step 2**) to obtain code slices and their corresponding vector representations. Then, for each slice, both its unstructured (i.e., statement embedding by statement encoder) and structured (i.e., control- and data-flow) information are fed into the well-trained detection model as graph inputs for vulnerability detection.

3.1 Feature Extraction

Previous studies [58, 73] have shown that combining diverse dimensional code representations, such as CPG, is beneficial for DL models to capture the program semantics. To this end, conduct intraprocedural and interprocedural analysis to build AST and SDG, respectively. AST reflects the syntax structure of a function and SDG provides the control- and data-flow information between statements within- and cross-functions. These two code representations preserve rich syntactic and semantic information of programs beneficial to feature representation learning. However, a single function usually contains dozens or even hundreds of code lines while the vulnerability exists only in a few lines of code, simply taking the whole program to train a detection model will degrade the capability to identify key features relevant to vulnerabilities. Thus, *MVD+* adopts program slicing [88] to perform backward and forward slicing based on SDG from a program point of interest to filter noise induced by irrelevant statements.

To ensure that the extracted program slices contain memory-related vulnerabilities, we mainly focus on two types of program points of interest: (1) *system API calls* and (2) *pointer variable*. As mentioned in previous works [19, 49, 50, 101], the misuse of *system API calls* is one of the major causes of vulnerabilities, including memory-related vulnerabilities. For example, *syscall_buf* is a typical *system API call* related to buffer operations in *Linux Kernel*. It often occurs in *Out-of-bounds Read/Write* and other similar buffer-related vulnerabilities. In total, we collect 537 *system API calls* from popular static memory detectors [27, 71] and security operation checkers [4, 53] as slicing criteria for extracting vulnerable code snippets. For *pointer variable*, it has been widely adopted by traditional static analysis-based approaches [27, 45, 78]. It should be noted that starting from the program point of interest, we perform backward slicing according to both control- and data-dependence, but forward slicing based on only data-dependence, because improper memory operations (e.g., allocating memory but not freeing it) have been involved in the forward data-dependence, while usually forward control-dependence will cover a great deal of irrelevant statements that would not be vulnerable in most cases [49].

Figure 5 provides an example to show the process of our feature extraction. As shown in Figure 5(a), it is a *memory leak* vulnerability. At line 5, it allocates memory through `malloc()` in function `memory_leak_func()` without freeing even to the end of the program. In our approach,



```
1  void memory_leak ()
2  {
3     char *str = "This is a string";
4     char *str1;
5     memory_leak_func(strlen(str), &str1);
6     strcpy(str1, str);
7  }

8  void memory_leak_func(int len, char **stringPtr)
9  {
10    char *p = malloc (sizeof(char) * (len+1));
11    *stringPtr = p;
12 }
```

(a) Exemplary Code Sample

(b) Program Slicing

Fig. 5. Details of our program slicing based on interprocedural control- and data-flow analysis.

the interprocedural control- and data-flow information of the vulnerable program is first extracted to construct the SDG of the program, which is shown in Figure 5(b). To reduce irrelevant nodes, we adopt the sensitive function call at line 5 (i.e., Node 5 highlighted in red) as the program point to perform backward and forward slicing. Node 6 is control-dependent on Node 5 with an Edge 5 → 6, and data-dependent on Node 4 with an Edge 4 → 6. After slicing, Node 6 is removed, because it is not data-dependent on Node 5.

3.2 Node Embedding

Existing DL-based vulnerability detectors [15, 84, 99] view the AST as a special graph structure and leverage GNNs to learn program semantics from diverse code representations (e.g., AST, CFG, and PDG) for model training. However, the neighborhood aggregation scheme (as described in Section 2.4) adopted by GNNs is ill-suited for modeling tree structures, because the hierarchical syntactic information between parent and children nodes is hard to capture effectively.

To this end, we adopt a hierarchical representation learning strategy, which performs syntax-aware neural embedding for each statement of source code and feeds these unstructured (code embeddings of statements) and structured information (control- and data-flows between statements) into the GNN model to learn the distributed representation of statements. Specifically, we first apply the BPE algorithm to capture the lexical information of code tokens. Code tokens (i.e., the leaf nodes of ASTs) will be split and merged into meaningful subwords, and embedded into the fixed-size vector representations through the pre-trained CodeBERT model [29]. The use of BPE subword tokenization will help reduce the vocabulary size, alleviating the problem of ***Out-of-vocabulary*** **(OOV)** in code embedding.

Then, these token embeddings are served as initial feature representations to train a statement encoder. We adopt ASTNN [96] as the basic component of our statement encoder (as shown in Figure 6), because it can effectively alleviate the gradient vanishing and long-term dependency problems in TNNs. In particular, an AST $\mathcal{T}$ of a function is decomposed to a sequence of multi-way (a tree with more than two children nodes) **statement trees (ST-tree)** by traversing each node of $\mathcal{T}$ in a depth-first walk. Given a ST-tree $t \in \mathcal{T}$, the vector representations of its non-leaf node are updated by Equation (2). Next, all nodes along with their feature vectors in the ST-tree t are fed into a *max pooling* layer to calculate the final representation $\boldsymbol{x}_u$ of its corresponding statement u:

$$\boldsymbol{x}_u = [max(\boldsymbol{h}_{i1}), \ldots, max(\boldsymbol{h}_{ik})], i = 1, \ldots, N, \tag{5}$$

where N is the number of nodes in a ST-tree t.

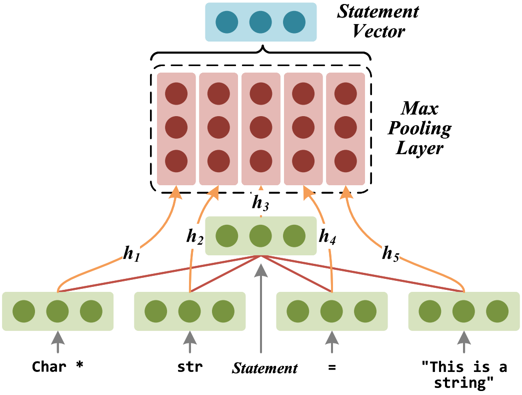


Fig. 6. The statement encoder used in our node embedding module.

Based on our syntax-aware neural embedding strategy, the feature vector of each statement node in SDG can preserve both lexical and syntactic information of source code.

3.3 Graph Learning

To train a model that can learn implicit vulnerability patterns from source code and locate suspicious vulnerable statements, we construct a novel graph learning framework, FS-GNN for graph learning. The details of our approach are shown in Figure 7. The key insight of FS-GNN is to jointly embed both statement embedding and flows information to capture sensitive contextual information for semantic learning. FS-GNN is composed of three parts: graph embedding, resampling, and classification.

Graph Embedding. Different from most of the existing graph embedding approaches that embed only nodes in the graph, we leverage the entity-relation composition operations $\phi(\cdot)$ used in Knowledge Graph embedding approaches [7, 18] to jointly embed statement nodes and multiple flow edges to incorporate edge embedding into the update of node information. To be specific, during the process of graph embedding in FS-GNN, the node embedding h_v of statement node v can be updated by

$$\boldsymbol{h}_v = f\left(\sum_{(u,r)\in\mathcal{N}(v)} \boldsymbol{W}_{\lambda(r)}\phi(\boldsymbol{x}_u, \boldsymbol{z}_r)\right), \tag{6}$$

where $\boldsymbol{h}_v$ denotes the updated representation of node v. $\boldsymbol{\mathcal{N}}(v)$ is a set of immediate neighbors of v for its outgoing edges. $\phi(\cdot)$ is a composition operator, including subtraction, multiplication, and circular-correlation. $\boldsymbol{x}_u$ and $\boldsymbol{z}_r$ denotes initial features for node u (encoded by our syntax-aware statement encoder in Equation (5)) and edge r, respectively. Similar to traditional **Relational Graph Convolutional Networks (RGCN)** [68], initial edge representation for edge r can be encoded by basis decomposition [68] as $\boldsymbol{z}_r = \sum_{b=1}^{\mathcal{B}} \alpha_{br}\boldsymbol{v}_b$, where $\boldsymbol{v}_{b\in\mathcal{B}}$ is a set of learnable basis vectors and $\alpha_{br} \in \mathbb{R}$ is also the learnable scalar weight specific to edge type and basis. $\boldsymbol{W}_{\lambda(r)}$ represents a edge type specific parameter. To make FS-GNN context-aware and capture important information from outgoing edges, we double edges by adding inverse edges and assign different weight parameters according to edge types (i.e., $\boldsymbol{W}_{\lambda(r)} = \boldsymbol{W}_O$ when r is an initial edge, and $\boldsymbol{W}_{\lambda(r)} = \boldsymbol{W}_I$ when r is an inverse edge).

Similarly, the edge embedding h_r of edge r can be updated by $\boldsymbol{h}_r = \boldsymbol{W}_{rel}\boldsymbol{z}_r$, where $\boldsymbol{W}_{rel}$ is a learnable transformation matrix that projects all the relations to the same embedding space as nodes.

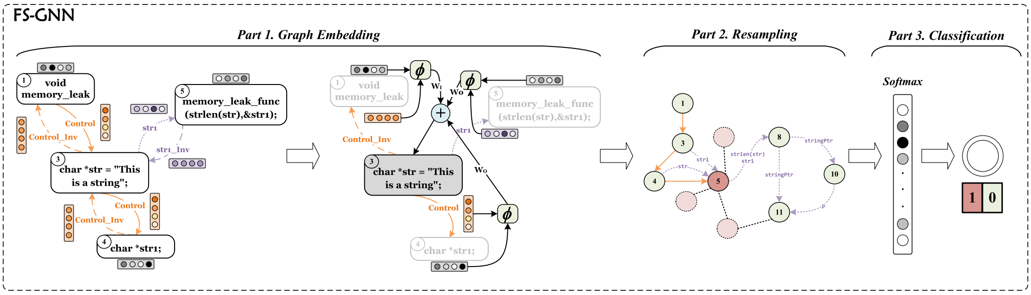


Fig. 7. Graph learning with FS-GNN.

Finally, the representation of a node v and edge r updated after l layers are shown as

$$\boldsymbol{h}_v^{l+1} = f\left(\sum_{(u,r)\in\mathcal{N}(v)} \boldsymbol{W}_{\lambda(r)}^{l} \phi(\boldsymbol{h}_u^l, \boldsymbol{h}_r^l)\right), \tag{7}$$

$$\boldsymbol{h}_r^{l+1} = \boldsymbol{W}_{rel}^{l} \boldsymbol{z}_r^l. \tag{8}$$

Note that $h_v^0 = x_u$ and $h_r^0 = z_r$ (i.e., initial representation of node v and edge r).

With the help of our flow-sensitive graph learning, contextual information can be captured and sensitive flow information is given more attention. For example, in Figure 7, initial node representation is encoded by Doc2Vec and edge representation is calculated by basis decomposition. Edge matrix is inversed first to capture contextual feature information. Then, to aggregate information from neighbor nodes to update the representation of Node 3, initial representations of Nodes 1, 4, and 5 are embedded jointly with their incoming edges (i.e., Edges 3 → 1, 3 → 4, 3 → 5) by Equation (7) to preserve some important features from outgoing nodes.

Resampling. After l layers graph learning, directly training the classifiers on all statement nodes is biased, because the distribution of non-vulnerable nodes and vulnerable nodes is extremely imbalanced. For example, in Figure 5(b), although we have filtered out some irrelevant nodes by program slicing, the number of non-vulnerable nodes (i.e., Nodes 1–4, 6, 8–11) is still larger than that of vulnerable nodes (i.e., Node 5). To generate some synthetic vulnerable nodes to re-balance the distribution, we adopt *GraphSMOTE* [97], a *graph-level* oversampling framework, as the basic component for our resampling.

Concretely, it contains two steps: (1) node generation and (2) edge generation. First, to generate high-quality synthetic nodes, we utilize the widely used *SMOTE* [16] algorithm to perform interpolation on vulnerable nodes. It searches for the closest neighbour node around each minority node (i.e., vulnerable node) in the embedding space and generates synthetic nodes between them. Then, edge generator adopts weighted inner production [97] to generate edges and gives link predictions for synthetic nodes by setting a threshold η to keep the connectivity of the graph. If the predicted probability of connection between synthetic node v' and its closest neighbor node u is greater than η, then both the synthetic node v' and edge $[v', u]$ will be put into the augmented adjacency matrix of original graphs. To make the analysis easier, the type of all synthetic edges is set as *"Control"* (i.e., synthetic nodes are control-dependent on their neighbor nodes).[1]

Owing to the contribution of resampling, the proportion of memory-related vulnerable statements increases, avoiding the well-trained detection model biased caused by imbalanced

[1] We omit data-dependency flow, because during the empirical study, we find that a large number of irrelevant synthetic data-dependency edges can introduce biases and make the performance of the detection model deteriorate.

distribution of vulnerable nodes and non vulnerable nodes. For example, in Figure 7, three synthetic nodes (Pink-shaded) are connected with one vulnerable node (i.e., Node 5) and one non-vulnerable node (i.e., Node 11).

Classification. Before training the classification model, FS-GNN adopts one-layer *flow-sensitive graph learning* block in Section 3.1 again to update node information by Equation (7). By learning both the unstructured (i.e., statement embedding) and structured (i.e., various flows) features from nodes and edges, the classification model are employed to distinguish vulnerable and non-vulnerable statements.

To train the model, we use the *softmax* activation function as the last linear layer for node classification and minimize the following cross-entropy loss on all labeled nodes (i.e., vulnerable or non-vulnerable):

$$\underset{\theta}{min}\,\mathcal{L} = -\sum_{G\in\mathcal{G}} \frac{1}{|\mathcal{V}|} \sum_{i\in\mathcal{V}} \sum_{k=1}^{K} t_{ik} \ln h_{ik}^{(L)}, \tag{9}$$

where G is a code slice graph in the training set $\mathcal{G}$, $\mathcal{V}$ is the set of nodes in our training set. $h_{ik}^{(L)}$ represents the probability of node i belonging to class k, where $k = \{0, 1\}$ for the binary node classification task. t_{ik} denotes respective ground truth label for node i.

3.4 Vulnerability Detection

In the detection phase, we apply the well-trained model to detect potential memory-related vulnerabilities in programs and locate suspicious statements.

Specifically, similar to training phase, program semantics reflected in the graph representations of source code are captured through interprocedural analysis. To reduce the number of memory operations-irrelevant statements, programs are sliced according to points of interest (*system API calls* and *pointer variable*) to obtain a batch of program slices (Section 3.1). Next, statement nodes in program slices are embedded into low-dimensional vectors through syntax-aware statement encoder (Section 3.2). Finally, both unstructured (i.e., statement embedding) and structured (i.e., control- and data-flow) information are used as graph input to feed into the well-trained detection model for vulnerability detection.

4 EXPERIMENTAL SETUP

4.1 Research Questions

In this section, our experiments focus on answering the following **Research Questions (RQs)**:
RQ1: Effectiveness. In particular, we would like to investigate:

- **RQ1a:** *Can MVD+ outperform state-of-the-art DL-based vulnerability detection approaches?*

The studies that are most relevant to *MVD+* are DL-based vulnerability detection approaches. By investigating this RQ, we aim to answer how effective *MVD+* performs in memory-related vulnerability detection comparing with the state-of-the-art DL-based approaches.

- **RQ1b:** *Can MVD+ outperform traditional static analysis-based vulnerability detectors?*

Static analysis-based vulnerability detection tools are widely used and perform well on memory-related vulnerabilities. In addition, static analysis-based approaches can identify the statement-level results for vulnerability detection (i.e., fine-grained detection results). Therefore, the purpose of this RQ is to analyze how effective *MVD+* performs compared with existing static analysis-based detectors.

RQ2: Efficiency. *How efficient is MVD+ compared with baselines in terms of time cost of detecting memory-related vulnerabilities?*

Efficiency is important for evaluating the performance of memory-related vulnerability detection approaches. An approach that costs too much time for detecting vulnerabilities may encounter adoption barriers in practice. This RQ is to investigate whether *MVD+* can make a reasonable trade-off between accuracy and efficiency.

RQ3: Ablation Study. In this work, we perform both intraprocedural and interprocedural analysis to model vulnerability features, and adopt a novel hierarchical representation learning strategy with syntax-aware node embedding and flow-sensitive graph learning. Thus, we, respectively, study the contribution of each component:

- **RQ3a:** *How does interprocedural contexts contribute to the performance of vulnerability feature extraction?*

Almost all DL-based approaches extract vulnerability features from vulnerable functions or slices, which reduce the difficulty and overhead of static analysis while also lose a part of contextual program semantics. Thus, in this RQ, we focus on investigating that whether the interprocedural contextual semantics are beneficial to improving the quality of vulnerability features?

- **RQ3b:** *To what extent does the syntax-aware node embedding strategy influence the performance of vulnerability feature learning?*

Due to the gap between tree-based and graph-based code representations, simply leveraging a unified neural network model (e.g., GNN) to learn program semantics from mixed code representations may confuse different feature information of vulnerabilities. To this end, we aim to study whether learning code syntactic features first by adopting a tree-based node embedding strategy can improve the performance of subsequent vulnerability feature learning?

- **RQ3c:** *Is the flow-sensitive graph learning powerful in capturing precise program semantics?*

One of the key contributions of our approach is flow-sensitive graph neural network, which jointly embeds both unstructured (i.e., code snippets) and structured (i.e., control- and data-flows) information to learn comprehensive program semantics. We aim to show whether sensitive contextual information captured by FS-GNN contributes to memory-related vulnerability detection in comparison with other popular GNNs.

4.2 Dataset

4.2.1 Data Collection. Existing vulnerability datasets are either not tailored for memory-related vulnerabilities [15, 28, 99], or not sufficient for training DL models (e.g., SPEC CINT2000 [35]). To this end, we constructed *MemoryVul* [59], a large-scale memory-related vulnerability dataset, which covers 23 common memory-related vulnerabilities (including CWE-119, -120, -121, -122, -124, -125, -126, -127, -131, -188, -244, -401, -415, -416, -476, -590, -761, -763, -787, -789, -805, -806, and -824 [22]). *MemoryVul* is based on two data sources: (1) SARD [74], a collection of test cases (varying from synthetic programs to real-world applications) with known vulnerabilities; and (2) CVE [21], a well-known vulnerability database. In this work, we focused on applications written in C/C++ due to their frequent memory problems caused by low-level control of memory [79]. Finally, 11 popular open-source applications are selected as target projects.

Table 1 reports the statistics of *MemoryVul*. In total, there are 1,613 real-world vulnerabilities in CVE and 5,266 test cases in SARD. Columns 3–5 denote the function-level statistics of each project, including the number of vulnerable functions (Column 3), non-vulnerable functions (Column

Table 1. Statistics of *MemoryVul*

Project	**# Samples**	**Function-Level**			**Slice-Level**		
		# Vul	**# Non-vul**	**Total**	**# Vul**	**# Non-vul**	**Total**
Linux Kernel	934	1,176	9,063	10,239	2,561	16,823	19,384
FFmpeg	84	92	717	809	164	1,527	1,691
Asterisk	18	19	184	203	43	415	s458
Libarchive	24	29	276	305	64	503	567
LibTIFF	24	26	193	219	57	368	425
Libav	16	16	102	118	27	184	211
LibPNG	13	13	107	120	35	191	226
QEMU	121	146	1,535	1,681	383	3,705	4088
Wireshark	57	64	443	507	158	725	883
OpenSSL	46	52	397	449	142	646	788
Chromium	276	338	2,869	3,207	670	4,862	5,532
SARD	5,266	6,192	13,686	19,878	10,269	22,417	32,686
Total	**6,879**	**8,163**	**29,572**	**37,735**	**14,753**	**52,366**	**67,119**

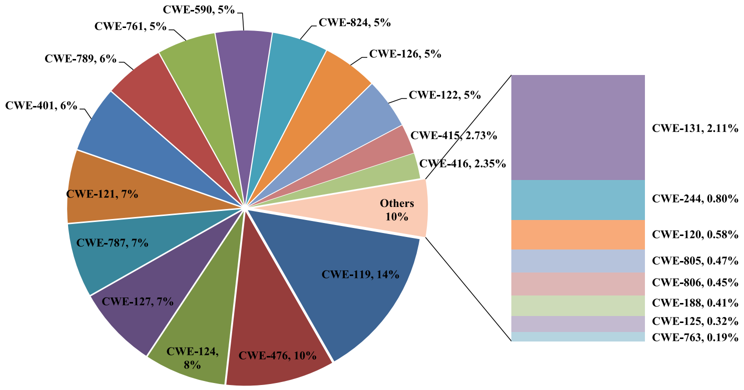


Fig. 8. Distribution of Vulnerability Types.

4), and total functions (Column 5). Similar, Columns 6–8, respectively, represent the number of vulnerable slices (Column 6), non-vulnerable slices (Column 7), and total slices (Column 8). Furthermore, the distribution of different types of memory-related vulnerabilities in our dataset is shown in Figure 8, with CWE-119 (Improper Restriction of Operations within the Bounds of a Memory Buffer, 14%), CWE-476 (NULL Pointer Dereference, 10%), and CWE-124 (Buffer Underflow, 8%) accounting for the top three vulnerability types.

4.2.2 Data Labeling. To train a detection model, we first need to conduct data labeling. There are two types of labels for statement nodes in the graph representation of a program: (1) *vulnerable* represents that the node is related to an improper operation in the vulnerable programs; (2) *non-vulnerable* represents that the node is related to the normal operation. To make this process automatic, we adopted a simple labeling strategy with *diff* files [6, 36]. We first conduct program slicing for each vulnerable sample to generate a number of slices. Then, for each slice of the samples in SARD, we labeled the statement nodes annotated with “errors” as *vulnerable*. For each slice of the real-world vulnerabilities in CVE, we compared statements in each slice and that in the corresponding vulnerable function according to *diff* files. If a statement was deleted or altered

Table 2. Details of DL-based Baselines

Approach	**Granularity**	**Feature Extraction**		**Training and Detecting**	
		Code Representation	**Node Embedding**	**Model Type**	**Model**
VulDeePecker	Slice	DDG	Word2Vec	RNN	BLSTM
SySeVR	Slice	PDG	Word2Vec	RNN	BGRU
DeepWuKong	Slice	PDG	Doc2Vec	GNN	GCN
Devign	Function	AST+CFG+DDG+NCS	Word2Vec+Node Type	GNN	GGNN
ReVeal	Function	CPG	Word2Vec+Node Type	GNN	GGNN
IVDetect	Statement	AST+PDG	Glove+Tree-LSTM+BGRU	GNN	GCN+GNNExplainer
LineVD	Statement	PDG	CodeBERT	GNN	GAT
LineVul	Statement	Sequence	Word&Positional Encoding	Transformer	CodeBERT
MVD	Statement	PDG+CG	Doc2Vec	GNN	FS-GNN

(i.e., starting with "-" in *diff* files), then it would be labeled as *vulnerable*, and *non-vulnerable* otherwise. However, in practice, part of memory-related vulnerabilities did not contain "-" in their patches. For example, in CVE-2019-19083 [24], memory leaks because allocated memory cannot be released when memory allocation fails. This vulnerability can be fixed by adding a memory release statement. Thus, for these vulnerabilities can not be directly labeled, we manually labeled vulnerable nodes through identifying improper operations [56] (e.g., memory allocation or deallocation statements). 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, then the sample would be forwarded to the Ph.D. evaluator for further investigation.

4.3 Baselines

To evaluate our approach, we compared *MVD+* with both DL-based and static analysis-based vulnerability detection approaches.

4.3.1 DL-based Approaches. We selected eight state-of-the-art DL-based vulnerability detection approaches and our previous work [13] as baselines. The details are shown in Table 2.

- **VulDeePecker** [50] 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).
- **SySeVR** [49] 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.).
- **DeepWuKong** [19] generates program slices based on PDG and leverages **Graph Convolutional Network (GCN)** to learn both unstructured and structured code information of a program to support detection of a wide range of vulnerabilities.
- **Devign** [99] combines multiple code representations (e.g., AST, CFG, and DDG) to model programs at the function-level, and adopts GGNN to learn the implicit vulnerability semantics for classification.
- **ReVeal** [15] proposes to leverage CPG and GGNN to automatically learn the graph properties of source code.
- **IVDetect** [47] combines multiple embedding techniques (Glove [65], Tree-LSTM [80], etc.) to extract the contextual information of source code for function-level vulnerability detection, and leverages GNNExplainer to interpret the detection results by providing crucial statements (the sub-graph in PDG) relevant to vulnerabilities.
- **LineVD** [36] leverages GAT [82] and CodeBERT [29] to capture control- and data-dependencies between statements, and formulates statement-level vulnerability detection as a node classification task.

- **LineVul** [32] uses a transformer-based model (BERT architecture) to generate representation of source code and leverages the attention mechanism to locate vulnerable statements.

4.3.2 Static Analysis-based Approaches. We also selected five state-of-the-art static analysis-based vulnerability detection approaches as baselines. The details of these five approaches are shown as follows:

- **PCA** [45] is a static interprocedural data-flow analysis tool that scales to industry-scale software systems with a practical cost-effectiveness trade-off. It features a partial call-path analysis to speed up data dependence computation.
- **Saber** [78] tracks the flow of values from allocation to free sites using a sparse value-flow graph (SVFG) and detects memory leaks by solving a graph reachability problem on SVFG.
- **Flawfinder** [31] scans source code against its built-in vulnerability pattern database, and produces a list of potential vulnerabilities sorted by risk.
- **RATS** [67] performs a rough analysis of source code and flags common security related programming errors such as buffer overflows.
- **Infer** [40] statically checks a series of bugs, such as null pointer access, resource and memory leaks, which lead to the crash of applications or serious performance degradation.

4.4 Implementation

We implemented *MVD+* in Python using PyTorch [66]. Our experiments were performed with the Nvidia Graphics Tesla T4 GPU, installed with Ubuntu 18.04, CUDA 10.1. We used Joern [93] to construct AST and PDG for each function, and then generated SDGs based on the caller-callee relation between functions. The hidden dimension of the statement encoder is 100. FS-GNN is trained in a batch-wise fashion until converging and the batch size is set to 64. The dimension of the vector representation of each node is set to 128 and the dropout is set to 0.1. ADAM [42] optimization algorithm is used to train the model with the learning rate of 0.001. Weight decay is set to 5e-1 and over-sampling scale is set as 1.0. The other hyper-parameters of our neural network are tuned through grid search.

4.5 Experimental Methodology

4.5.1 RQ1: Effectiveness.

RQ1a: Comparison with DL-based Vulnerability Detection Approaches.

We compared *MVD+* with the state-of-the-art DL-based vulnerability detection approaches on both function-level and statement-level vulnerability detection tasks.

Function-Level Vulnerability Detection. We selected five function-level DL-based approaches (VulDeePecker, SySeVR, DeepWuKong, Devign, and ReVeal) as baselines. Since MVD and *MVD+* were not designed for function-level vulnerability detection, we adopted a compromise strategy, i.e., if a vulnerable statement was identified correctly by *MVD+*, we would consider the function it belonged to was also detected correctly. We used ***Accuracy (Acc)***, ***Precision (Pre)***, ***Recall (Rec)***, and ***F1*** as our function-level evaluation metrics.

- ***True Positive*** **(TP)** is the number of functions correctly predicted as vulnerable.
- ***True Negative*** **(TN)** is the number of functions correctly predicted as non-vulnerable.
- ***False Positive*** **(FP)** is the number of functions incorrectly classified as vulnerable.
- ***False Negative*** **(FN)** is the number of functions incorrectly classified as non-vulnerable.
- ***Accuracy*** **(Acc)** evaluates the performance that how many instances can be correctly labeled. It is calculated as: $Acc = \frac{TP+TN}{TP+FP+TN+FN}$.

- ***Precision*** **(Pre)** is the fraction of true vulnerabilities among the detected ones. It is defined as: $Pre = \frac{TP}{TP+FP}$.
- ***Recall*** **(Rec)** measures how many vulnerabilities can be correctly detected. It is calculated as: $Rec = \frac{TP}{TP+FN}$.
- ***F1-score*** **(F1)** is the harmonic mean of *Recall* and *Precision*, and can be calculated as: $F1 = 2 * \frac{Rec*Pre}{Rec+Pre}$.

The whole dataset was randomly split into 80%, 10%, and 10% for training, validation, and testing. To ensure that different types of vulnerabilities are evenly distributed in each subset, we shuffle the dataset based on vulnerability types in parallel. For example, 80% vulnerable samples belonging to CWE-119 (Improper Restriction of Operations within the Bounds of a Memory Buffer) are demarcated as training set, and the remaining 20% samples are divided into two halves for validation (10%) and testing (10%). To make sure that our model was fine-tuned, we used *ten-fold cross-validation* to evaluate the generalization ability of each approach.

Statement-Level Vulnerability Detection. We selected three statement-level DL-based approaches (IVDetect, LineVul, and LineVD) and MVD as baselines. IVDetect outputs a sequence of control- and data-dependent statements (the sub-graph in PDG) as fine-grained detection results, while LineVul and LineVD directly pinpoint the actual vulnerable statements. To evaluate the fine-grained detection performance of each statement-level approach, apart from three previously used binary classification metrics (including *Pre*, *Rec*, and *F1*), we also considered two imbalance-aware metrics, ***Matthews Correlation Coefficient (MCC)*** [81] and ***Area Under the Precision-Recall Curve (PR-AUC)*** [36], and two ranked metrics, ***Mean First Rank (MFR)*** and ***Mean Average Rank (MAR)*** [47], to evaluate the confidence of each statement-level vulnerability detection approach.

- ***Mean First Rank*** **(MFR)** is the mean of the rankings for the first true-positive vulnerable statement located by the model.
- ***Mean Average Rank*** **(MAR)** is the mean of the rankings for all true-positive vulnerable statements located by the model.
- ***Matthews Correlation Coefficient*** **(MCC)** measures the performance of the model on imbalanced dataset. It is defined as: $\frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$.
- ***Area Under the Precision-Recall Curve*** **(PR-AUC)** is a threshold-independent performance metric that measures a classifier's ability to discriminate between vulnerable and non-vulnerable statements. PR-AUC is computed by measuring the area under the curve that plots the *Precision* against *Recall*.

We also randomly split the dataset into the ratio of 8:1:1 for training, validation, and testing. The performance results were obtained from the average of 10 independent runs.

RQ1b: Comparison with Static Analysis-based Vulnerability Detection Approaches.

Following Lipp et al. [52], we compared *MVD+* with five static analysis-based approaches (PCA, Saber, Flawfinder, RATS, and Infer) at the *function-level*, because the marked code location where a vulnerability potentially manifests as a security-critical program state varied widely in different static analyzers. We adopted the same experimental setup (i.e., a vulnerable function will be considered to be correctly identified if any statement in it is reported as vulnerable) and metrics (*Acc*, *Pre*, *Rec*, and *F1*) as RQ1a for evaluation.

In addition, we also evaluated the practical performance of *MVD+* in detecting different types of memory-related vulnerabilities. However, it is challenging to compare different static analyzers fairly because of their inconsistent classification standard of the vulnerability types they support. For example, Flawfinder adopts CWE-120: Buffer Copy without Checking Size of

Table 3. Mapping Relationship between CWE Categories and Analyzer-specific Vulnerability Identifiers

Vulnerability Type	CWE-ID
Buffer Overflow	119, 120, 121, 122, 124, 125, 126, 127, 131, 805, 806, 787, 824
Memory Leak	401, 590, 761, 763, 789
Double Free	415
Use After Free	416
Null Pointer Dereference	476

Input (“`Classic Buffer Overflow`”) to describe buffer copies without length checks, while Infer uses `Buffer_Overrun`. To this end, we leveraged the CWE hierarchy [22] to map each analyzer-agnostic CWE-ID to the corresponding analyzer-specific vulnerability identifier. The mapping relationship is listed in Table 3. We excluded CWE-188 and CWE-244 from the evaluation, because they (1) could not be categorized into analyzer-specific vulnerability types and (2) occupied a relatively low proportion in the dataset (accounting for 1.21%, covering 67 vulnerable samples). Overall, 21 types of memory-related vulnerabilities in our dataset are grouped into five common analyzer-specific vulnerability identifiers, including ***Buffer Overflow*** **(BO)**, ***Memory Leak*** **(MemL)**, ***Double Free*** **(DF)**, ***Use After Free*** **(UAF)**, and ***Null Pointer Dereference*** **(NPD)**.

4.5.2 RQ2: Efficiency. For answering RQ2, we re-conducted the function-level vulnerability detection experiments to compare *MVD+* with both DL-based and static analysis-based approaches under the same experimental setup as RQ1, and recorded the average training (only for DL-based approaches) and detection time of each approach.

4.5.3 RQ3: Ablation Study.

RQ3a: Contribution of Interprocedural Contexts.

We created a variant without interprocedural contexts by directly constructing the joint graph representation (AST and PDG), a CPG-like code representation [93] that does not contain the CFG edge, for each input function snippet. We fairly evaluated them under two configurations: our previous proposed MVD and *MVD+*, and adopted the same experimental setup and evaluation metrics as RQ1.

RQ3b: Effect of Syntax-aware Statement Embedding.

We built multiple variants of our model by replacing the syntax-aware statement encoder (ASTNN) with other feature embedding approaches usually seen in vulnerability detection models [13, 47, 99], including token-based (Word2Vec [60], Doc2Vec [44], and Glove [65]) and tree-based techniques (TBCNN [61], Tree-LSTM [80], and ASTNN (equipped with statement encoder initialized by Word2Vec) [96]). We adopted the same experimental setup and evaluation metrics as RQ1.

RQ3c: Impact of Flow-sensitive Graph Learning.

We respectively replaced our graph embedding layer with three famous GNN models, including **GCN** [43], **GGNN** [46], and **RGCN** [68], to evaluate the contribution of flow-sensitive graph learning to memory-related vulnerability detection. (1) **GCN** scales linearly in the number of graph edges and learns hidden layer representations that encode both local graph structure and features of nodes. (2) **GGNN** uses Gated Recurrent Units and unrolls the recurrence for a fixed number of steps and use backpropagation through time to compute gradients. It can incorporate higher degree neighborhoods across relation graphs. (3) **RGCN** introduces relation-specific transformations to aggregate node information across relation graphs, and addresses over-parameterization

Table 4. Evaluation Results on Function-level Vulnerability Detection in Percentage Compared with DL-based Baselines

Metric	RNN-based		GNN-based				
	VulDeePecker	SySeVR	Devign	ReVeal	DeepWuKong	MVD	*MVD+*
Accuracy	74.26	70.05	82.57	83.91	81.59	88.52	**90.11**
Precision	39.85	46.31	59.19	62.60	56.04	64.19	**68.53**
Recall	26.11	53.58	67.22	66.46	63.27	71.35	**74.86**
F1-score	31.55	49.68	62.95	64.47	59.43	67.58	**71.55**

by proposing basis and block-diagonal decomposition. In addition, to understand the contribution of our resampling layer in training an unbiased classifier on such a highly imbalanced node classification task, we also deploy two variants of FS-GNN, one with resampling layer and the other without. We adopted the same experimental setup and evaluation metrics as RQ1.

5 EXPERIMENTAL RESULTS

5.1 RQ1: Effectiveness of *MVD+*

5.1.1 RQ1a: Comparison with DL-based Approaches.

Performance of Vulnerable Function Identification. Table 4 shows the overall results of each DL-based approach on function-level vulnerability detection in terms of the aforementioned evaluation metrics. Overall, *MVD+* outperforms all of the six referred DL-based approaches, achieving 90.11% in Accuracy and 71.55% in F1, exceeding the best baseline (except our previous work MVD) ReVeal by 9.47% in Precision and by 12.64% in Recall.

MVD+ vs. RNN-based Approaches. We can find that our approach relatively improves two representative sequence-based approaches, VulDeePecker and SySeVR, by 71.96% and 47.98%, respectively, in terms of Precision, and by 186.71% and 39.72%, respectively, in terms of Recall. Poor Precision and Recall indicates that existing sequence-based approaches either give a large number of false-positive results or miss potential vulnerabilities. The root cause for this performance gap is that sequence-based models, such as BLSTM and BGRU, are ill-suited for modeling the well-structured control- and data-flows of programs.

MVD+ vs. GNN-based Approaches. Table 4 shows that our approach also outperforms three state-of-the-art GNN-based vulnerability detection baselines and MVD in every measurement metric. Specifically, *MVD+* improves F1 over the baselines Devign, ReVeal, DeepWuKong, and MVD by 1.13×, 1.10×, 1.20×, and 1.06×, respectively. The key reason for *MVD+* to detect more true-positive vulnerabilities than existing GNN-based approaches is that program semantics (control- and data-flows) between statements are well-captured by our flow-sensitive graph learning. By contrast, The existing methods either treat different code representations equally (e.g., GCN adopted in DeepWuKong), ignoring rich program semantics, or are difficult to expand to large program graphs with complex semantic relations (e.g., GGNN used by Devign and ReVeal). As a result, the under-utilization of flow information severely restricts the performance of existing GNN-based baselines in detecting complex memory-related vulnerabilities.

Performance of Vulnerable Statement Localization. Table 5 summarizes the performance comparison of our approach with respect to four DL-based baselines on statement-level vulnerability detection. As can be seen, we find that *MVD+* is more precise than other existing DL-based vulnerability locators, achieving significant improvements ranging from 10.29% (compared to LineVul) to 58.61% (compared to LineVD). In particular, compared to the popular baselines, the F1-score metric is improved by 146.07%, 56.85%, 53.85%, and 21.81%, respectively.

Table 5. Evaluation Results on Statement-level Vulnerability Detection Compared with DL-based Baselines

Metric		IVDetect	LineVD	LineVul	MVD	*MVD+*
Classification	**Precision**	32.64	28.39	40.83	39.44	**45.03**
	Recall	15.78	40.46	29.16	47.19	**62.48**
	F1-score	21.27	33.37	34.02	42.97	**52.34**
	MCC	17.92	26.81	29.35	25.07	**34.72**
	PR-AUC	54.99	61.43	53.01	66.74	**72.39**
Ranked	**MFR**	7.35	4.87	5.51	3.94	**3.06**
	MAR	8.64	6.31	6.82	5.52	**4.88**

In terms of two imbalance-aware metrics, MCC and PR-AUC, *MVD+* achieves 5.37% (compared to LineVul) and 5.65% (compared to MVD) absolute improvement over the best baseline, respectively. The corresponding related improvements are 18.30% and 8.47%. The results demonstrate that *MVD+* can effectively alleviate the negative influence of class imbalance issue on fine-grained vulnerable statement localization. In addition, for ranking vulnerable statements, our approach improves MFR by 4.29, 1.81, 2.45, and 0.88 ranks, and improves MAR by 3.76, 1.43, 1.94, and 0.64 ranks over IVDetect, LineVD, LineVul, and MVD.

MVD+ vs. IVDetect. IVDetect leverages GNNExplainer, a model-agnostic explanation technique for GNNs, to locate a group of statements that are control- or data-dependent on the actual vulnerable statements. Such a sample-specific explanation suffers from efficiency (having to be retrained for each detected vulnerable function) and precision (the located vulnerable sub-graph may not be a substructure of the original PDG) problems. As a result, IVDetect has much higher MFR (7.35) and MAR (8.64) that is 1.40× and 0.77× higher than our MFR and MAR, respectively.

MVD+ vs. LineVD. Similar to our approach, LineVD also formulates vulnerable statement localization as a node classification task. It incorporates an element-wise multiplication between the prediction of statement- and function-level embeddings to balance Information conflicts of different granularity. However, as shown in Table 5, due to the insensitivity of flow information and imbalanced distribution of vulnerable statements, fine-grained localization performance is still not satisfactory, which has a high MFR and MAR, i.e., 4.87 and 6.31, that is approximately 59.15% and 29.30% higher than ours.

MVD+ vs. LineVul. Despite the powerful performance of Transformer-like architecture, LineVul performs the worst among these approaches, we infer that it is caused by the limited amount of vulnerable samples used for training. Specifically, the total number of samples in the dataset is only 37,735 (21.63% of them are vulnerable), which makes transformer overfitting to the training data. In addition, LineVul only considers sequence features alone while ignoring structured information like control- and data-flows that hard to be modeled by sequence-based models.

Answer to RQ1a: *MVD+* significantly outperforms five state-of-the-art function-level baselines in Accuracy and F1, and achieves better localization precision compared with three recent statement-level baselines. We attribute the improvements to the crafted hierarchical representation learning strategy and fine-grained graph learning model.

5.1.2 RQ1b: Comparison with Static Analysis-based Approaches.

Performance of Vulnerable Function Identification. Table 6 shows that our model is also more effective than other static analysis-based approaches and MVD in detecting memory-related vulnerabilities at the function-level. Overall, *MVD+* outperforms all of the baselines in every measurement metric.

Table 6. Evaluation Results on Function-level Vulnerability Detection in Percentage Compared with Static Analysis-based Approaches

Metric	PCA	Saber	Flawfinder	RATS	Infer	MVD	*MVD+*
Accuracy	76.57	69.12	52.31	60.14	63.84	88.52	**90.11**
Precision	54.82	49.05	13.86	11.09	32.97	64.19	**68.53**
Recall	58.03	52.44	16.97	18.25	26.72	71.35	**74.86**
F1-score	56.38	50.69	15.26	13.80	29.52	67.58	**71.55**

Table 7. Evaluation Results on Detecting Different Types of Memory-related Vulnerabilities in Percentage Compared with Static Analysis-based Approaches

Vul Type	PCA		Saber		Flawfinder		RATS		Infer		MVD		*MVD+*	
	Acc	F1	Acc	F1	Acc	F1	Acc	F1	Acc	F1	Acc	F1	Acc	F1
BO	78.46	55.41	73.88	49.41	47.68	18.53	75.21	8.89	51.34	35.92	90.74	72.82	**91.94**	**76.15**
MemL	**89.11**	67.83	84.90	55.07	65.67	10.21	62.46	5.54	67.53	22.46	83.56	71.83	85.08	**72.64**
DF	73.75	55.23	70.11	54.92	56.86	11.05	61.34	4.05	58.17	14.24	75.62	66.42	**79.13**	**70.04**
UAF	66.59	51.95	58.31	45.86	52.83	7.31	55.31	4.19	63.87	20.59	89.44	70.91	**82.66**	**73.08**
NPD	66.04	27.52	69.34	34.87	52.76	0	63.89	0	52.29	11.36	74.25	60.97	**75.99**	**65.82**

Among the static memory detectors we chose, PCA and Saber obtain relatively better detection performance. PCA achieves the highest Precision (54.82%) and Recall (58.03%) due to its consideration of global variables and precise interprocedural data flow analysis. Our approach still improves PCA by 22.46% in terms of Precision, and by 28.02% in terms of Recall. The reason is that static analysis-based approaches mainly rely on well-defined vulnerability rules or patterns hand-crafted by human experts. They are effective in simple memory-related vulnerabilities (e.g., SARD dataset). However, real-world vulnerabilities are more complicated, restricting the effectiveness of these static analysis-based detectors. Similar to these detectors, *MVD+* also analyzes interprocedural control- and data-flow information. Owing to the powerful performance of deep learning models, *MVD+* can learn implicit vulnerability patterns from vulnerable code, making it more effective in real-world scenarios.

Performance of Vulnerability Type Prediction. We further conduct experiments to compare the performance of different static memory detectors in detecting common memory-related vulnerabilities to confirm the effectiveness of our approach. The experimental results are presented in Table 7. We can observe that *MVD+* outperforms all of the static analysis-based baselines and MVD by a significant margin in terms of Acc and F1, except that Acc of PCA is 4.74% higher than ours in memory leak detection. However, our approach improves PCA by 7.09% in terms of F1, which means that *MVD+* can detect more true positives. In addition, we find that each approach (including *MVD+*) performs worse in detecting ***Null Pointer Dereference*** **(NPD)** than other types of vulnerabilities. On average, the performance of PCA, Saber, and Infer drops by 109.32%, 47.15%, and 105.13%, respectively, in terms of F1. Particularly, Flawfinder and RATS can not detect any true-positive NPD vulnerability. We attribute to the highly reliance on well-designed vulnerability patterns or rules and imprecision of pure static analysis. For example, the best performed baseline PCA uses context-insensitivity points-to analysis to compute interprocedural data-flows to trade for scalability and efficiency. As a result, long-term data-dependence information will be missed, making the data dependence computed by PCA suffers imprecision.

Answer to RQ1b: *MVD+* produces improvements of up to 39.70% in terms of Accuracy and 195.28% in terms of F1-score on average when comparing with the state-of-the-art static vulnerability detectors. In addition, our approach also achieves higher performance as compared to the baselines in detecting five common memory-related vulnerabilities.

Table 8. Time Cost in Minutes of Different Approaches on Training and Detecting Vulnerable Functions

Approach	Training	Detection	Total
VulDeePecker	72.39	2.18	74.57
SySeVR	92.46	3.03	95.49
Devign	141.13	4.94	146.07
ReVeal	129.76	3.20	132.96
DeepWuKong	183.77	5.66	189.43
IVDetect	163.48	4.72	168.20
LineVD	104.85	3.74	108.59
LineVul	377.91	8.37	386.28
MVD	194.62	4.11	198.73
PCA	N/A	2.58	2.58
Saber	N/A	3.07	3.07
Flawfinder	N/A	10.66	10.66
RATS	N/A	12.09	12.09
Infer	N/A	155.99	155.99
MVD+	229.36	4.43	233.79

N/A: Not Applicable.

5.2 RQ2: Efficiency of *MVD+*

Table 8 lists the time cost in minutes of each approach in training and detecting vulnerable functions. On average, our approach costs 229.36 min for model training and uses 4.43 min to finish detecting memory-related vulnerabilities in the validation set.

In comparison with existing DL-based baselines, we can see that our approach spends more time on training detection model except for the transformer-based LineVul. We attribute to the relatively complicated data pre-processing operations, such as graph construction and program slicing, which occupies most of the time in our approach and other graph-based baselines like DeepWuKong. In fact, due to the characteristic that DL models can be trained off-line, their training costs may not be that important. Based on private vulnerability datasets, the users can train their own detection models offline and make a prediction within seconds. Furthermore, among all of the DL-based approaches, VulDeePecker incurs the least training and detection time, because it only considers data-flows and uses a simple sequence model, BLSTM, for model training. However, combining with the results in Table 4, we can find that it generates the lowest detection results, because the lack of control-flows and the limitations of sequence model make it fail to capture the structured information. By contrast, graph-based modeling of source code makes the neural models more sensitive to vulnerability-related features, while it also increases the model complexity compared with sequences. In general, we can still conclude that the time cost of our approach is acceptable, because (1) the detection time (4.43) of *MVD+* is a bit lower than the average (4.44) of all nine DL-based baselines; and (2) our approach performs much better than the baselines on detecting memory-related vulnerabilities as reported in RQ1.

Last, we find that more than half of (three of five) static analyzers cost more than 10 min in detecting vulnerabilities except for PCA and Saber. We believe it is reasonable, because precise static analyzers are heavyweight and suffer from scalable problems. To solve this limitation, PCA speeds up data dependence computation through sacrificing partial detection precision. In spite of its competitive detection efficiency, PCA gives a large number of false positives as our evaluation results confirmed in RQ2.

Table 9. Effectiveness of Interprocedural Contexts

Granularity	Metric	MVD$_{Intra}$	MVD	*MVD+*$_{Intra}$	*MVD+*
Function	**Accuracy**	83.99	88.52	84.16	**90.11**
	Precision	57.81	64.19	61.42	**68.53**
	Recall	68.34	71.35	66.29	**74.86**
	F1-score	62.63	67.58	63.76	**71.55**
Statement	**Precision**	33.75	39.44	37.64	**45.03**
	Recall	31.44	47.19	38.52	**62.48**
	F1-score	32.55	42.97	30.07	**52.34**
	MCC	20.34	25.07	22.15	**34.72**
	PR-AUC	58.41	66.74	61.08	**72.39**
	MFR	6.99	3.94	6.13	**3.06**
	MAR	8.81	5.52	7.48	**4.88**

"MVD$_{Intra}$" and "*MVD+*$_{Intra}$" represent the performance of MVD (or *MVD+*) on the intraprocedural static analysis strategy, respectively.

Answer to RQ2: Although the training time of *MVD+* is longer than most of the DL-based baselines, our approach obtains better detection results with relatively shorter detection time, making a trade-off between effectiveness and efficiency.

5.3 RQ3: Ablation Study

5.3.1 RQ3a: Contribution of Interprocedural Contexts. Table 9 illustrates both the function- and statement-level vulnerability detection performance of MVD and *MVD+* with their variants, which only conduct intraprocedural analysis (MVD$_{Intra}$ and *MVD+*$_{Intra}$) to construct graph representations of input programs for feature extraction. We can see that interprocedural contextual information can indeed enrich the vulnerability-related program semantics and are beneficial to improving the quality of extracted vulnerability features. In particular, compared with their corresponding variants, the F1-score of MVD and *MVD+* are improved by 7.9% and 12.22% on function-level vulnerability detection, and 32.01% and 74.06% on statement-level vulnerability detection, respectively. Furthermore, we observe that compared to the other binary classification metrics, Recall is more sensitive to the addition of interprocedural contextual semantics, ranging from 4.4% to 62.2%. This is likely, because interprocedural function calls (e.g., releasing a pointer spanning multiple functions) are abused in C/C++ applications and it's difficult for existing DL-based code models to infer such a complex program logic, thus resulting in high false negatives.

As for two imbalance-aware metrics (MCC and PR-AUC) and two ranked metrics (MFR and MAR), they can consistently achieve higher values with the addition of interprocedural contexts. Specially, compared with the variant model (*MVD+*$_{Intra}$), which is only trained on the intraprocedural contexts, the MFR of *MVD+* is improved by 100.33%. *MVD+* also increases the MAR by 53.28%. In other words, with the addition of interprocedural contexts, truly vulnerable statements can be located by *MVD+* in nearly top-3 alarms, which save developers half the time for double-review compared to the previous one.

Answer to RQ3a: The combination of intraprocedural and interprocedural analysis contributes significantly to the performance of *MVD+*, with an F1-score improvement of 12.22% and 74.06% on function- and statement-level vulnerability detection, respectively. The results indicate that the addition of interprocedural contextual semantics can promote the quality of vulnerability feature extraction and benefit the performance of vulnerability detection.

Table 10. Evaluation Results on Different Statement Embedding Techniques

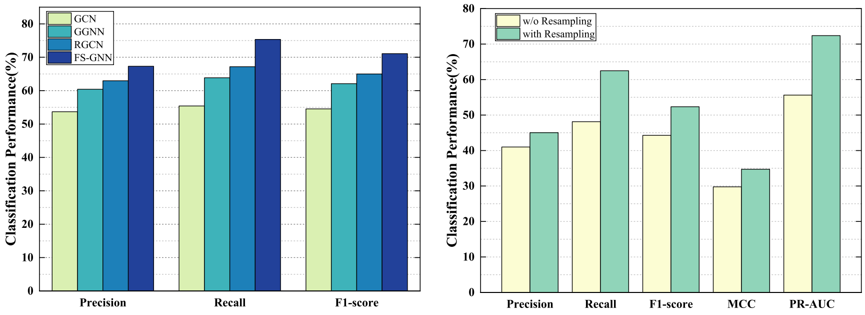
Approach	Acc	Pre	Rec	F1
Word2Vec	78.32	52.96	57.72	55.24
Doc2Vec (MVD)	79.84	54.80	58.39	56.54
Glove	81.71	56.01	58.24	57.10
TBCNN	81.29	56.88	61.32	59.01
Tree-LSTM	87.41	59.13	65.98	62.37
ASTNN	85.25	63.77	68.43	63.61
ASTNN+CodeBERT (*MVD+*)	**89.41**	**66.84**	**72.50**	**69.56**

5.3.2 RQ3b: Effect of Syntax-aware Statement Embedding. Table 10 presents the experimental results of our *MVD+* and the six baseline statement embedding techniques according to our four evaluation metrics (i.e., Accuracy, Precision, Recall, and F1-score). We can observe that our statement encoders with ASTNN and CodeBERT is more suitable than other embedding techniques in memory-related vulnerability detection. Specifically, our *MVD+* improves Word2Vec, Doc2Vec, Glove, TBCNN, Tree-LSTM, and ASTNN by 14.16%, 11.99%, 9.42%, 9.99%, and 4.88%, respectively, in terms of Accuracy, and by 25.92%, 23.03%, 21.82%, 17.88%, 11.53%, and 9.35%, respectively, in terms of F1.

An intuitive finding is that the tree-based embedding techniques (including TBCNN, Tree-LSTM, and ASTNN) outperform the token-based embedding techniques (i.e., Word2Vec, Doc2Vec, and Glove) in terms of Acc and F1. We believe it is reasonable, because the syntactic information contained in AST of source code makes the detection model focus on abstract structure features instead of learning irrelevant lexical features. Furthermore, we can see that due to the more powerful feature encoding ability of pre-trained CodeBERT model, *MVD+* has a higher performance than existing tree-based embedding techniques. Compared with the best performed baseline ASTNN, which is equipped with a statement encoder initialized by Word2Vec, our approach improves the detection performance significantly, achieving 89.41% in Accuracy and 69.57% in F1.

Answer to RQ3b: Our syntax-aware statement embedding strategy performs better as compared to existing embedding techniques, we attribute to the powerful ability of pre-trained models and tree-based neural networks in capturing lexical and syntactic information of source code.

5.3.3 RQ3c: Impact of Flow-sensitive Graph Learning. Table 9(a) shows the results of different graph learning models. We observe that our FS-GNN can improve the best performed baseline RGCN by 6.89% in terms of Precision and 12.16% in terms of Recall. There are mainly two reasons for this. On the one hand, FS-GNN adds edge types into the process of representation learning. It can be regarded as the joint learning of edge embedding and node embedding. Thus, FS-GNN can preserve the comprehensive program semantics based on interprocedural control- and data-flow, improving the flow-sensitivity for memory-related vulnerabilities. On the other hand, RGCN aggregates node and edge information through directed edge, while FS-GNN boosts the effect of edge types on context by adding corresponding inverse edges. Still taking the *double free* vulnerability as an example, information of memory free in different branch statements will affect their condition nodes jointly. Therefore, important features of output nodes are also preserved by FS-GNN for node update and information propagation. In addition, we can find that although GGNN can process multiple relations across graphs, it is still limited by the increasing number of relations, resulting in lower performance in comparison with RGCN and FS-GNN. We can also observe that the performance of GCN is poor. The main reason is that neglecting edge types leads to the missing



(a) Comparison Results between Different GNN Models

(b) Effect of Node-level Data Balancing

Fig. 9. Contributions of flow-sensitive graph learning.

of structured code features (e.g., control- and data-flows). Without accurate control- and data-flow information, the performance of memory-related vulnerability detection drops sharply.

Figure 9(b) illustrates the effect of adding resampling layer in FS-GNN's pipeline. We can find that node-level data re-balancing positively contributes to locating fine-grained vulnerable statements. In particular, *MVD+* with the resampling layer achieves 4.04%, 14.35%, and 8.07% absolute improvement in Precision, Recall, and F1-score over the variant without re-balancing, respectively. The corresponding relative improvements are 9.86%, 29.82%, and 18.23%. Furthermore, two imbalance-aware metrics, MCC and PR-AUC, consistently achieve higher value with the addition of the resampling layer. The results indicate that synthesized new vulnerable samples with natural semantic relation information can effectively suppress the bias towards the majority non-vulnerable samples of trained GNN models, which facilitates the node classification performance of *MVD+* when only a small portion of statements are vulnerable.

Answer to RQ3c: Our flow-sensitive graph learning is effective in comprehensively learning fine-grained semantic features of source code, which significantly improves the performance of memory-related vulnerability detection, as it can better capture and distinguish structured semantic interactions between vulnerable and non-vulnerable statements.

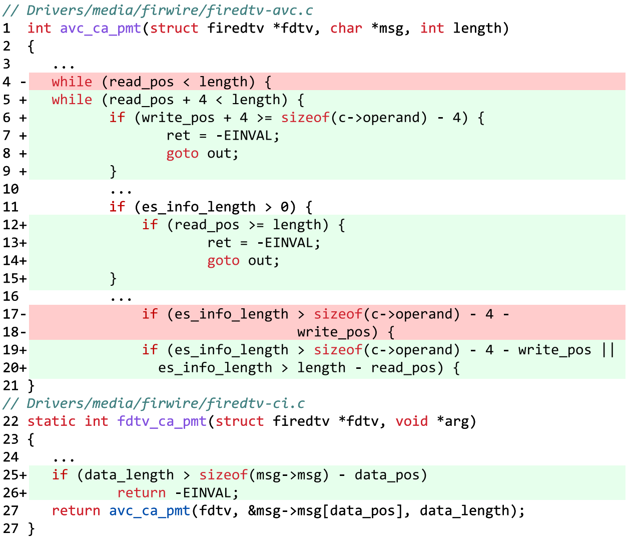
6 DISCUSSION

In this section, we present a case study to understand why our approach achieves better results than others and then disclose the threats to the validity.

6.1 Case Study

This case shows a heap buffer overflow (CWE-122) vulnerability in Linux Kernel. As shown in Figure 10, the root cause of this vulnerability is that the bounds checking in function `avcavc_ca_pmt ()` is not strict enough. If the variable `es_info_length` at line 17 is non-zero, then it reads a sixth byte, which allows a local user of the host machine to crash the system or escalate privileges on the system. Furthermore, the variable `data_length` from function `fdtv_ca_pmt()` can be invalid, which will lead to shift wrapping once more than four bytes in the loop are read.

The capability to detect this type of popular memory-related vulnerabilities involving multiple functions is due to the way we perform interprocedural analysis to capture comprehensive control- and data-dependencies across functions. In the process of code representation learning,



```
// Drivers/media/firwire/firedtv-avc.c
1  int avc_ca_pmt(struct firedtv *fdtv, char *msg, int length)
2  {
3     ...
4 -   while (read_pos < length) {
5 +   while (read_pos + 4 < length) {
6 +          if (write_pos + 4 >= sizeof(c->operand) - 4) {
7 +                 ret = -EINVAL;
8 +                 goto out;
9 +          }
10           ...
11           if (es_info_length > 0) {
12+              if (read_pos >= length) {
13+                     ret = -EINVAL;
14+                     goto out;
15+          }
16           ...
17-              if (es_info_length > sizeof(c->operand) - 4 -
18-                                     write_pos) {
19+              if (es_info_length > sizeof(c->operand) - 4 - write_pos ||
20+                es_info_length > length - read_pos) {
21 }
// Drivers/media/firwire/firedtv-ci.c
22 static int fdtv_ca_pmt(struct firedtv *fdtv, void *arg)
23 {
24    ...
25+   if (data_length > sizeof(msg->msg) - data_pos)
26+           return -EINVAL;
27    return avc_ca_pmt(fdtv, &msg->msg[data_pos], data_length);
27 }
```

Fig. 10. Case study: A buffer overflow vulnerability (CVE-2021-42739) in Linux Kernel. The vulnerable and patched statements are shaded in red and green, respectively. For simplicity purpose and page limitation, we only show the key lines of fixes.

these rich program semantics will be cooperated with code embeddings to learn vulnerability patterns. Furthermore, benefiting from our hierarchical representation learning strategy, token-level syntax information (e.g., numerical values) and statement-level semantic information (e.g., interprocedural data-flow `data_length` from `fdtv_ca_pmt` at line 22 to `avc_ca_pmt` at line 1) are well preserved by our approach for information interaction at different levels. Other DL-based baselines, such as VulDeePecker, Devign, and ReVeal, do not support interprocedural analysis and ignore the relationships among functions, which provides a global perspective of program semantics. By contrast, although static analyzers like PCA conduct interprocedural data dependence computation to support vulnerability detection, their imprecision gives a huge number of false positives, resulting in limited effectiveness.

6.2 Threats to Validity

Threats to external validity concern the generalizability of our experiment results. We respectively investigated 6,879 vulnerable samples from 11 distinct C/C++ open-source projects and SARD, and used the mixed dataset for model evaluation like prior works. However, due to the huge gap in code complexity, detection results in practical scenarios may not be so satisfactory. Furthermore, *MVD+* is specifically designed for detecting memory-related vulnerabilities in C/C++ programs. Thus, our experimental results may not be reproducible when applied to more complex vulnerabilities or languages (e.g., Java). Nevertheless, our approach is generic and can be extended for other vulnerabilities and languages.

Threats to internal validity relate to two factors. The first is our imperfect node labeling. In this work, we manually labeled nodes that did not contain any "*delete*" statement as vulnerable through identifying related sensitive operations. Thus, it is possible that some samples are mislabeled. To avoid harmful influence caused by incorrect node labels, we tried our best to conduct the node labeling for the vulnerable samples in our dataset by three experienced researchers. In addition, the implementation of baselines also threats the results of our experiments. To compare with existing deep learning-based vulnerability detection approaches, we have re-implemented *Devign* based on a popular repository,[2] since it is closed-source. We try our best to build and tune the *Devign* parameters on our dataset.

Threats to construct validity relate to the suitability of our evaluation metrics. We use only four traditional measurements designed for classification models (i.e., Accuracy, Precision, Recall, and F1) to evaluate the performance of function-level vulnerability detection results and use two statement-level measurements that are preferred by previous works [36, 47]. However, other metrics can be used for evaluation as well. Thus, future work may conduct an observational study with industrial practitioners to better understand how do security experts perform code inspection in real-world practices.

7 RELATED WORKS

Existing vulnerability detection approaches can be divided into three main categories: static analysis-based, dynamic analysis-based, and learning-based approaches.

7.1 Static Memory-related Vulnerability Detection

Static analysis-based approaches aim to detect vulnerabilities based on specific vulnerability patterns or memory state model. Emamdoost et al. [26] proposed K-MeLD, which combines multiple techniques to automatically identify specialized allocation/deallocation functions and then reasons on the ownership of the allocated memory object to infer the location of expected dealocation call, to detect kernel memory leaks. Lyu et al. [57] presented Goshawk, which leverages the structure-aware and object-centric memory operation synopsis to summarize the **memory management (MM)** behaviors, and combines natural language processing and data flow analysis to identify complex and custom MM functions in source code, for memory corruption bug detection. Shi et al. [71] proposed Pinpoint to optimize widely used sparse value-flow analysis through decomposing the cost of high-precision points-to analysis. Fan et al. [27] presented Smoke, a staged approach that computes a succinct set of candidate memory leak paths based on use-flow graph and leverages a dedicated constraint solver to verify the feasibility of those candidates, to solve the scalability problem of memory leak detection at industrial scale. Wang et al. [85] proposed MLEE, which cross-checks the presence of memory deallocations on different early-exit paths and normal paths, to intelligently detect memory leaks in operating system kernels. Differently, our approach learns vulnerability patterns from large amounts of vulnerability data without requiring any prior knowledge of vulnerabilities.

7.2 Dynamic Memory-related Vulnerability Detection

Dynamic detection methods run the source code and dynamically track the allocation, use and release of memory at the run-time. DoubleTake [54] split the program execution into multiple blocks and saved the program state before each block started running. The program state would be checked after the execution of the block ended to judge whether there was an error in memory. Sniper [41] used the processor's monitoring unit to track the access instructions to heap memory. It calculated the staleness of heap objects and executed relevant instructions again to capture memory leakage during program execution. Some binary-level dynamic approaches, such as Valgrind [62], Dr.Memory [8], and AddressSanitizer [70], tracked memory allocation and deallocation during a program's execution, and detected leaks by scanning the program's heap for memory blocks that no pointer points to. Recently, a number of works adopt fuzzing [11, 14] to automatically explore different program paths and expose potential memory vulnerabilities. MemLock [89] statically identified memory consumption-related statements and operations, and employed branch

[2] https://github.com/epicosy/devign

coverage as well as memory consumption information to guide the fuzzing process, for memory consumption bug detection. UAFL [83] leveraged operation sequence coverage to progressively cover the operation sequences that are likely to trigger use-after-free vulnerabilities, and adopted information flow analysis to identify the relationship between the input and the program variables in the conditional statement to improve the efficiency of the fuzzing process. Unlike dynamic analysis-based approaches, our approach does not require the execution of programs.

7.3 Learning-based Vulnerability Detection

With the advance of **Machine Learning (ML)**, especially DL, a large number of learning-based approaches are proposed to automatically learn explicit or implicit vulnerability features from known vulnerabilities to identify unseen vulnerabilities in projects [5, 15, 19, 49, 84]. Li et al. [50] proposed VulDeePecker, a *slice-level* vulnerability detection approach that represents source code as sequences and uses RNN (e.g., LSTM and BGRU) to learn the syntactic and semantic information of vulnerabilities. Zhou et al. [99] proposed Devign, which combines multiple code representations (e.g., AST, CFG, and PDG) to model vulnerability features and adopts GGNN [10] to learn rich code semantics from structured graph representations for *function-level* vulnerability detection. To pinpoint concrete vulnerability types, Zou et al. [101] proposed an attention-based multi-class vulnerability detection approach, μVulDeePecker. They introduced *code attention* to accommodate information useful for learning local features and used a building-block BiLSTM to fuse different code features.

Despite their effectiveness, the detection granularity of these approaches are mainly at the function- and slice-level, which is still coarse-grained. Thus, Li et at. [48] proposed VulDeeLocator, which leverages the k-max pooling layer and the average pooling layer to narrow down the scope of vulnerable statements. Li et al. [47] proposed IVDetect, which uses a GNN model to predict vulnerable functions first and then leverages GNNExplainer to output the sub-graph that contributes the most to the predictions. To further locate vulnerable statements, Fu et al. [32] proposed LineVul, which leverages the self-attention mechanism [100] of the BERT architecture to rank statements based on their attention scores.

The main difference between our approach and the above DL-based vulnerability detection approaches is that existing approaches mainly focus on constructing a unified representation learning model to support detection of multiple types of vulnerabilities, while our approach aims to learn fine-grained code semantics from different types of flows (e.g., data-flows with different variables) via a novel flow-sensitive graph neural network to detect memory-related vulnerabilities at the statement-level.

8 CONCLUSION

In this article, we propose *MVD+*, a novel DL-based memory-related vulnerability detection approach, which captures comprehensive program semantics to support suspicious statement localization. By the hierarchical representation learning strategy, which performs a syntax-aware neural embedding within statements and captures structured context information across statements based on a novel Flow-Sensitive Graph Neural Networks (FS-GNN), to learn both the syntactic and semantic features of vulnerable code, *MVD+* outperforms current state-of-the-art DL-based approaches as well as well-known static analyzers with 9.47–22.46% higher Precision and 12.64–28.02% Recall on our constructed dataset for memory-related vulnerabilities.

REFERENCES

[1] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to represent programs with graphs. In *Proceedings of the 6th International Conference on Learning Representations (ICLR'18)*.

[2] Uri Alon and Eran Yahav. 2021. On the bottleneck of graph neural networks and its practical implications. In *Proceedings of the 9th International Conference on Learning Representations (ICLR'21).*

[3] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.* 3, POPL (2019), 40:1–40:29.

[4] Pan Bian, Bin Liang, Jianjun Huang, Wenchang Shi, Xidong Wang, and Jian Zhang. 2020. SinkFinder: Harvesting hundreds of unknown interesting function pairs with just one seed. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'20).* ACM, 1621–1625.

[5] Lili Bo, Yue Li, Xiaobing Sun, Xiaoxue Wu, and Bin Li. 2023. VulLoc: Vulnerability localization based on inducing commits and fixing commits. *Frontiers Comput. Sci.* 17, 3 (2023), 173207.

[6] Lili Bo, Xuanrui Zhu, Xiaobing Sun, Ni Zhen, and Bin Li. 2021. Are similar bugs fixed with similar change operations? An empirical study. *Chinese J. Electr.* 30, 1 (2021), 55–63.

[7] Antoine Bordes, Nicolas Usunier, Alberto García-Durán, Jason Weston, and Oksana Yakhnenko. 2013. Translating embeddings for modeling multi-relational data. In *Proceedings of the 27th Annual Conference on Neural Information Processing Systems (NeurIPS'13).* 2787–2795.

[8] Derek Bruening and Qin Zhao. 2011. Practical memory checking with Dr. Memory. In *Proceedings of the 9th International Symposium on Code Generation and Optimization (CGO'11).* IEEE, 213–223.

[9] Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. 2021. TreeCaps: Tree-based capsule networks for source code processing. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI'21).* AAAI Press, 30–38.

[10] Jie Cai, Bin Li, Jiale Zhang, Xiaobing Sun, and Bing Chen. 2023. Combine sliced joint graph with graph neural networks for smart contract vulnerability detection. *J. Syst. Softw.* 195 (2023), 111550.

[11] Sicong Cao, Biao He, Xiaobing Sun, Yu Ouyang, Chao Zhang, Xiaoxue Wu, Ting Su, Lili Bo, Bin Li, Chuanlei Ma, Jiajia Li, and Tao Wei. 2023. ODDFUZZ: Discovering Java deserialization vulnerabilities via structure-aware directed greybox fuzzing. In *Proceedings of the 44th IEEE Symposium on Security and Privacy (SP'23).* IEEE.

[12] Sicong Cao, Xiaobing Sun, Lili Bo, Ying Wei, and Bin Li. 2021. BGNN4VD: Constructing Bidirectional Graph Neural-network for Vulnerability Detection. *Inf. Softw. Technol.* 136 (2021), 106576.

[13] Sicong Cao, Xiaobing Sun, Lili Bo, Rongxin Wu, Bin Li, and Chuanqi Tao. 2022. MVD: Memory-related vulnerability detection based on flow-sensitive graph neural networks. In *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering (ICSE'22).* ACM, 1456–1468.

[14] Sicong Cao, Xiaobing Sun, Xiaoxue Wu, Lili Bo, Bin Li, Rongxin Wu, Wei Liu, Biao He, Yu Ouyang, and Jiajia Li. 2023. Improving Java deserialization gadget chain mining via overriding-guided object generation. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE'23).* ACM.

[15] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2022. Deep learning based vulnerability detection: Are we there yet? *IEEE Trans. Softw. Eng.* 48, 9 (2022), 3280–3296.

[16] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. 2002. SMOTE: Synthetic minority over-sampling technique. *J. Artif. Intell. Res.* 16 (2002), 321–357.

[17] Zhe Chen, Chong Wang, Junqi Yan, Yulei Sui, and Jingling Xue. 2021. Runtime detection of memory errors with smart status. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'21).* ACM, 296–308.

[18] Xingqi Cheng, Xiaobing Sun, Lili Bo, and Ying Wei. 2022. KVS: A tool for knowledge-driven vulnerability searching. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'22).* ACM, 1731–1735.

[19] Xiao Cheng, Haoyu Wang, Jiayi Hua, Guoai Xu, and Yulei Sui. 2021. DeepWukong: Statically detecting software vulnerabilities using deep graph neural network. *ACM Trans. Softw. Eng. Methodol.* 30, 3 (2021), 38:1–38:33.

[20] Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. 2007. Practical memory leak detection using guarded value-flow analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07).* ACM, 480–491.

[21] Common Vulnerabilities and Exposures. 2022. Retrieved from https://cve.mitre.org/

[22] Common Weakness Enumeration. 2022. Retrieved from https://cwe.mitre.org/

[23] CVE-2019-15920. 2022. Retrieved from https://github.com/torvalds/linux/commit/088aaf17aa79300cab14dbee2569c58cfafd7d6e

[24] CVE-2019-19083. 2022. Retrieved from https://github.com/torvalds/linux/commit/055e547478a11a6360c7ce05e2afc3e366968a12

[25] Hoa Khanh Dam, Truyen Tran, Trang Pham, Shien Wee Ng, John Grundy, and Aditya Ghose. 2021. Automatic feature learning for predicting vulnerable software components. *IEEE Trans. Softw. Eng.* 47, 1 (2021), 67–85.

[26] Navid Emamdoost, Qiushi Wu, Kangjie Lu, and Stephen McCamant. 2021. Detecting kernel memory leaks in specialized modules with ownership reasoning. In *Proceedings of the 28th Annual Network and Distributed System Security Symposium (NDSS'21).* The Internet Society.

[27] Gang Fan, Rongxin Wu, Qingkai Shi, Xiao Xiao, Jinguo Zhou, and Charles Zhang. 2019. Smoke: Scalable path-sensitive memory leak detection for millions of lines of code. In *Proceedings of the 41st International Conference on Software Engineering (ICSE'19)*. IEEE / ACM, 72–82.

[28] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. A C/C++ code vulnerability dataset with code changes and CVE summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR'20)*. ACM, 508–512.

[29] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020 (Findings of ACL, Vol. EMNLP 2020)*. ACL, 1536–1547.

[30] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (1987), 319–349.

[31] Flawfinder. 2022. Retrieved from http://www.dwheeler.com/flawfinder

[32] Michael Fu and Chakkrit Tantithamthavorn. 2022. LineVul: A transformer-based line-level vulnerability prediction. In *Proceedings of the 19th IEEE/ACM International Conference on Mining Software Repositories (MSR'22)*. IEEE, 608–620.

[33] David Gens, Simon Schmitt, Lucas Davi, and Ahmad-Reza Sadeghi. 2018. K-miner: Uncovering memory corruption in linux. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS'18)*. The Internet Society.

[34] David L. Heine and Monica S. Lam. 2006. Static detection of leaks in polymorphic containers. In *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*. ACM, 252–261.

[35] John L. Henning. 2000. SPEC CPU2000: Measuring CPU performance in the new millennium. *Computer* 33, 7 (2000), 28–35.

[36] David Hin, Andrey Kan, Huaming Chen, and Muhammad Ali Babar. 2022. LineVD: Statement-level vulnerability detection using graph neural networks. In *Proceedings of the 19th IEEE/ACM International Conference on Mining Software Repositories (MSR'22)*. IEEE, 596–607.

[37] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar T. Devanbu. 2012. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*. IEEE, 837–847.

[38] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Comput.* 9, 8 (1997), 1735–1780.

[39] Nasif Imtiaz and Laurie A. Williams. 2021. Memory error detection in security testing. Retrieved from https://arXiv:2104.04385

[40] Infer. 2022. Retrieved from https://fbinfer.com

[41] Changhee Jung, Sangho Lee, Easwaran Raman, and Santosh Pande. 2014. Automated memory leak detection for production use. In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*. ACM, 825–836.

[42] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR'15)*.

[43] Thomas N. Kipf and Max Welling. 2017. Semi-supervised classification with graph convolutional networks. In *Proceedings of the 5th International Conference on Learning Representations (ICLR'17)*.

[44] Quoc V. Le and Tomás Mikolov. 2014. Distributed representations of sentences and documents. In *Proceedings of the 31th International Conference on Machine Learning (ICML'14)*, Vol. 32. 1188–1196.

[45] Wen Li, Haipeng Cai, Yulei Sui, and David Manz. 2020. PCA: Memory leak detection using partial call-path analysis. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'20)*. ACM, 1621–1625.

[46] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. 2016. Gated graph sequence neural networks. In *Proceedings of the 4th International Conference on Learning Representations (ICLR'16)*.

[47] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. Vulnerability detection with fine-grained interpretations. In *Proceeding of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'21)*. ACM, 292–303.

[48] Zhen Li, Deqing Zou, Shouhuai Xu, Zhaoxuan Chen, Yawei Zhu, and Hai Jin. 2022. VulDeeLocator: A deep learning-based fine-grained vulnerability detector. *IEEE Trans. Depend. Secur. Comput.* 19, 4 (2022), 2821–2837.

[49] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2022. SySeVR: A framework for using deep learning to detect software vulnerabilities. *IEEE Trans. Dependable Secur. Comput.* 19, 4 (2022), 2244–2258.

[50] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A deep learning-based system for vulnerability detection. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS'18)*. The Internet Society.

[51] Linux Kernel. 2022. Retrieved from https://www.kernel.org/

[52] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. 2022. An empirical study on the effectiveness of static C code analyzers for vulnerability detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on*

Software Testing and Analysis (ISSTA'22). ACM, 544–555.

[53] Dinghao Liu, Qiushi Wu, Shouling Ji, Kangjie Lu, Zhenguang Liu, Jianhai Chen, and Qinming He. 2021. Detecting missed security operations through differential checking of object-based similar paths. In *Proceedings of the 27th ACM SIGSAC Conference on Computer and Communications Security (CCS'21)*. ACM, 1627–1644.

[54] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. 2016. DoubleTake: Fast and precise error detection via evidence-based dynamic analysis. In *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*. ACM, 911–922.

[55] Yiling Lou, Qihao Zhu, Jinhao Dong, Xia Li, Zeyu Sun, Dan Hao, Lu Zhang, and Lingming Zhang. 2021. Boosting coverage-based fault localization via graph-based representation learning. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'21)*. ACM, 664–676.

[56] Kangjie Lu, Aditya Pakki, and Qiushi Wu. 2019. Detecting missing-check bugs via semantic- and context-aware criticalness and constraints inferences. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security'19)*. USENIX Association, 1769–1786.

[57] Yunlong Lyu, Yi Fang, Yiwei Zhang, Qibin Sun, Siqi Ma, Elisa Bertino, Kangjie Lu, and Juanru Li. 2022. Goshawk: Hunting memory corruptions via structure-aware and object-centric memory operation synopsis. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy (SP'22)*. IEEE, 2096–2113.

[58] Alejandro Mazuera-Rozo, Anamaria Mojica-Hanke, Mario Linares-Vásquez, and Gabriele Bavota. 2021. Shallow or deep? An empirical study on detecting vulnerabilities using deep learning. In *Proceedings of the 29th IEEE/ACM International Conference on Program Comprehension (ICPC'21)*. IEEE, 276–287.

[59] MemoryVul. 2022. Retrieved from https://github.com/SicongCao/MemoryVul

[60] Tomás Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 27th Annual Conference on Neural Information Processing Systems (NeurIPS'13)*. 3111–3119.

[61] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence (AAAI'16)*. AAAI Press, 1287–1293.

[62] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*. ACM, 89–100.

[63] Yu Nong, Haipeng Cai, Pengfei Ye, Li Li, and Feng Chen. 2021. Evaluating and comparing memory error vulnerability detectors. *Inf. Softw. Technol.* 137 (2021), 106614.

[64] Robert E. Noonan. 1985. An algorithm for generating abstract syntax trees. *Comput. Lang.* 10, 3/4 (1985), 225–236.

[65] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. Glove: Global vectors for word representation. In *Proceedings of the 26th Conference on Empirical Methods in Natural Language Processing (EMNLP'14)*. ACL, 1532–1543.

[66] PyTorch. 2022. Retrieved from https://pytorch.org/

[67] Rough Audit Tool for Security. 2022. Retrieved from https://code.google.com/archive/p/rough-auditing-tool-for-security

[68] Michael Sejr Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. 2017. Modeling relational data with graph convolutional networks. Retrieved from https://arXiv:1703.06103

[69] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL'16)*.

[70] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 23rd USENIX Annual Technical Conference (USENIX ATC'12)*. USENIX Association, 309–318.

[71] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. Pinpoint: Fast and precise sparse value flow analysis for million lines of code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18)*. ACM, 693–706.

[72] Saurabh Sinha, Mary Jean Harrold, and Gregg Rothermel. 1999. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*. ACM, 432–441.

[73] Jing Kai Siow, Shangqing Liu, Xiaofei Xie, Guozhu Meng, and Yang Liu. 2022. Learning program semantics with code representations: An empirical study. In *Proceedings of the 29th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER'22)*. IEEE, 554–565.

[74] Software Assurance Reference Dataset. 2022. Retrieved from https://samate.nist.gov/SARD/index.php

[75] Zihua Song, Junfeng Wang, Kaiyuan Yang, and Jigang Wang. 2023. HGIVul: Detecting inter-procedural vulnerabilities based on hypergraph convolution. *Inf. Softw. Technol.* 160 (2023), 107219.

[76] Ezekiel O. Soremekun, Lukas Kirschner, Marcel Böhme, and Andreas Zeller. 2021. Locating faults with program slicing: An empirical analysis. *Empir. Softw. Eng.* 26, 3 (2021), 51.

[77] Fazli Subhan, Xiaoxue Wu, Lili Bo, Xiaobing Sun, and Muhammad Rahman. 2022. A deep learning-based approach for software vulnerability detection using code metrics. *IET Softw.* 16, 5 (2022), 516–526.

[78] Yulei Sui, Ding Ye, and Jingling Xue. 2012. Static memory leak detection using full-sparse value-flow analysis. In *Proceedings of the 21st International Symposium on Software Testing and Analysis (ISSTA'12).* ACM, 254–264.

[79] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal war in memory. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (SP'13).* IEEE, 48–62.

[80] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. 2015. Improved semantic representations from tree-structured long short-term memory networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing (ACL'15).* ACL, 1556–1566.

[81] Chakkrit Tantithamthavorn, Ahmed E. Hassan, and Kenichi Matsumoto. 2020. The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. *IEEE Trans. Softw. Eng.* 46, 11 (2020), 1200–1219.

[82] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph attention networks. In *Proceedings of the 6th International Conference on Learning Representations (ICLR'18).*

[83] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. 2020. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE'20).* ACM, 999–1010.

[84] Huanting Wang, Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Yansong Feng, Lizhong Bian, and Zheng Wang. 2021. Combining graph-based learning with automated data collection for code vulnerability detection. *IEEE Trans. Inf. Forensics Secur.* 16 (2021), 1943–1958.

[85] Wenwen Wang. 2021. MLEE: Effective detection of memory leaks on early-exit paths in OS kernels. In *Proceedings of 32nd USENIX Annual Technical Conference (USENIX ATC'21).* USENIX Association, 31–45.

[86] Huihui Wei and Ming Li. 2017. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI'17).* ijcai.org, 3034–3040.

[87] Ying Wei, Xiaobing Sun, Lili Bo, Sicong Cao, Xin Xia, and Bin Li. 2021. A comprehensive study on security bug characteristics. *J. Softw. Evol. Process.* 33, 10 (2021).

[88] Mark Weiser. 1984. Program slicing. *IEEE Trans. Softw. Eng.* 10, 4 (1984), 352–357.

[89] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. 2020. MemLock: Memory usage guided fuzzing. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE'20).* ACM, 765–777.

[90] Xin-Cheng Wen, Yupan Chen, Cuiyun Gao, Hongyu Zhang, Jie M. Zhang, and Qing Liao. 2023. Vulnerability detection with graph simplification and enhanced graph representation learning. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE'23).* ACM.

[91] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE'16).* ACM, 87–98.

[92] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. 2021. A comprehensive survey on graph neural networks. *IEEE Trans. Neural Netw. Learn. Syst.* 32, 1 (2021), 4–24.

[93] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (SP'14).* IEEE Computer Society, 590–604.

[94] Guoqing Yan, Sen Chen, Yude Bai, and Xiaohong Li. 2022. Can deep learning models learn the vulnerable patterns for vulnerability detection? In *Proceedings of the 46th IEEE Annual Computers, Software, and Applications Conference (COMPSAC'22).* IEEE, 904–913.

[95] Hao Yu, Wing Lam, Long Chen, Ge Li, Tao Xie, and Qianxiang Wang. 2019. Neural detection of semantic code clones via tree-based convolution. In *Proceedings of the 27th International Conference on Program Comprehension (ICPC'19).* IEEE/ACM, 70–80.

[96] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *Proceedings of the 41st International Conference on Software Engineering (ICSE'19).* IEEE / ACM, 783–794.

[97] Tianxiang Zhao, Xiang Zhang, and Suhang Wang. 2021. GraphSMOTE: Imbalanced node classification on graphs with graph neural networks. In *Proceedings of the 14th ACM International Conference on Web Search and Data Mining (WSDM'21)*. ACM, 833–841.

[98] Tianchi Zhou, Xiaobing Sun, Xin Xia, Bin Li, and Xiang Chen. 2019. Improving defect prediction with deep forest. *Inf. Softw. Technol.* 114 (2019), 204–216.

[99] Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Proceedings of the 33rd Annual Conference on Neural Information Processing Systems (NeurIPS'19)*. 10197–10207.

[100] Zhou Zhou, Lili Bo, Xiaoxue Wu, Xiaobing Sun, Tao Zhang, Bin Li, Jiale Zhang, and Sicong Cao. 2022. SPVF: Security property assisted vulnerability fixing via attention-based models. *Empir. Softw. Eng.* 27, 7 (2022), 171.

[101] Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li, and Hai Jin. 2021. μVulDeePecker: A deep learning-based system for multiclass vulnerability detection. *IEEE Trans. Depend. Secur. Comput.* 18, 5 (2021), 2224–2236.

Received 20 February 2023; revised 27 August 2023; accepted 1 September 2023