MVD: Memory-Related Vulnerability Detection Based on Flow-Sensitive Graph Neural Networks

Sicong Cao
Yangzhou University
Yangzhou, China
MX120190439@yzu.edu.cn

Xiaobing Sun*
Yangzhou University
Yangzhou, China
xbsun@yzu.edu.cn

Lili Bo*
Yangzhou University
Yangzhou, China
lilibo@yzu.edu.cn

Rongxin Wu
Xiamen University
Xiamen, China
wurongxin@xmu.edu.cn

Bin Li
Yangzhou University
Yangzhou, China
lb@yzu.edu.cn

Chuanqi Tao
Nanjing University of Aeronautics
and Astronautics
Nanjing, China
taochuanqi@nuaa.edu.cn

ABSTRACT

Memory-related vulnerabilities constitute severe threats to the security of modern software. Despite the success of deep learning-based approaches to generic vulnerability detection, they are still limited by the underutilization of flow information when applied for detecting memory-related vulnerabilities, leading to high false positives.

In this paper, we propose *MVD*, a *statement-level* **M**emory-related **V**ulnerability **D**etection approach based on flow-sensitive graph neural networks (FS-GNN). FS-GNN is employed to jointly embed both unstructured information (i.e., source code) and structured information (i.e., control- and data-flow) to capture implicit memory-related vulnerability patterns. We evaluate *MVD* on the dataset which contains 4,353 real-world memory-related vulnerabilities, and compare our approach with three state-of-the-art deep learning-based approaches as well as five popular static analysis-based memory detectors. The experiment results show that *MVD* achieves better detection accuracy, outperforming both state-of-the-art DL-based and static analysis-based approaches. Furthermore, *MVD* makes a great trade-off between accuracy and efficiency.

CCS CONCEPTS

• **Security and privacy** → **Software security engineering**.

KEYWORDS

Memory-Related Vulnerability, Vulnerability Detection, Graph Neural Networks, Flow Analysis

*Xiaobing Sun and Lili Bo are the corresponding authors.

1 INTRODUCTION

As one of the most representative vulnerabilities, *memory-related vulnerabilities* can result in performance degradation and program crash, severely threatening the security of modern software [19, 66]. According to the data released by CVE (Common Vulnerabilities and Exposures [2]), nearly a third of the vulnerabilities (32.6%) in Linux Kernel [10] are related to improper memory operations [33].

Many static analysis approaches [21, 24, 31, 34, 38, 40, 52, 56, 58, 60, 61] have been proposed to detect memory-related vulnerabilities and shown their effectiveness. They use some pre-defined vulnerability rules or patterns to search for improper memory operations [41, 42]. However, well-defined vulnerability rules or patterns are highly dependent on expert knowledge, and thus it is difficult to cover all the cases. What's worse, the sophisticated programming logic in real-world software projects gets in the way of the manual identification of the rules, and thus greatly compromises the performance of the traditional static analysis-based approaches [51]. Recently, benefiting from the powerful performance of deep learning (DL), a number of approaches [16, 17, 20, 23, 44–46, 64, 70–72] have been proposed to leverage DL models to capture program semantics to identify potential software vulnerabilities. Compared with traditional static analysis-based approaches, they can automatically extract implicit vulnerability patterns from prior vulnerable code instead of requiring expert involvement. However, the existing DL-based approaches suffer from two limitations when applied to memory-related vulnerability detection, as described below.

Flow Information Underutilization: Due to the underutilization of flow information, existing DL-based approaches failed to detect complicated memory-related vulnerabilities in real-world projects [20] for the following two aspects: (1) lack of interprocedural analysis, and (2) partial flow information loss in model training. For the former one, most of DL-based approaches [17, 23, 44, 64, 71]

take the *function-level* vulnerable code as input to conduct intraprocedural analysis for feature extraction, ignoring call relations between functions. However, in real-world programs, operations like calling a user-defined function which realizes memory allocation or free are widespread. Missing interprocedural analysis may cause incomplete semantic modeling, resulting in lower Recall and Precision. For the latter one, limited by the capability of popular DL models (e.g., BiLSTM [30, 45, 46, 62], GGNN [64, 71], and GCN [20]) in handling multiple relations, partial flow information is lost during the process of model training. For example, *Devign* [71] uses GGNN [43] as the basic model to propagate and aggregate information across multi-relational graphs. Since GGNN treats the relational graph as multiple directed graphs without attributes (i.e., feature information is only passed between nodes connected by edges of the same type), its effectiveness is often compromised by the tremendous increase in the number of data-flow edges with different attributes. Thus, *Devign* has to substitute them with three other *token-level* relations (i.e., *LastRead*, *LastWrite*, and *ComputedFrom* [1]) to make it more adaptive for the graph embedding, sacrificing partial precise data-flow information well preserved in graphs. A simple 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).

Coarse Granularity: The detection granularity of the existing DL-based approaches is mostly at the *function-level* [17, 23, 44, 64, 71] or *slice-level* [20, 45, 46, 72]. However, developers still need to spend a deal of time in manually narrowing down the range of suspicious statements (or operations). Achieving fine-grained detection results is non-trivial. Due to the huge differences between various vulnerabilities, existing DL-based approaches for generic vulnerability detection have to sacrifice unique semantic features specific to certain vulnerabilities to ensure that the trained model can cover the general characteristics of the majority of vulnerabilities. In comparison with other vulnerabilities, memory-related vulnerabilities are usually fixed with one or several lines of code, which makes fine-grained detection possible. For example, *memory leak* can be located to the statement which allocates memory, while *use-after-free* can be located to the statement which frees memory.

In this paper, we propose a novel approach (*MVD*) based on flow-sensitive graph neural networks to alleviate the above limitations.

Fully Utilizing Flow Information: To capture more comprehensive and precise program semantics, *MVD* combines Program Dependence Graph (PDG) with Call Graph (CG) [53] to capture interprocedural control- and data-flow information. First, we conduct interprocedural analysis by extending PDG with additional semantic information (including call relations and return values between functions) using CG. In our approach, code snippets and relations (i.e., edges) are embedded in compact low-dimensional representations to preserve both the unstructured (i.e., source code) and structured (i.e., control- and data-flow) information. Furthermore, in order to make the detection model learn effective memory-related vulnerability patterns from comprehensive and precise flow information, *MVD* constructs a novel Flow-Sensitive Graph Neural Networks (FS-GNN) to jointly embed statements and flow information to capture program semantics from vulnerable code.

Fine Granularity: We formalize the detection of vulnerable statements as a node classification problem, i.e., identifying which statement(s) in the program is vulnerable. Specifically, *MVD* receives the graph representation of a program (in which graph nodes represent statements and edges indicate their relations) and outputs node labels (i.e., vulnerable or not).

Since there is currently no dataset that can be directly used for training a *statement-level* memory-related vulnerability detection model, we construct a dataset which contains 4,353 real-world memory-related vulnerabilities. The dataset as well as the empirical data are available online[1].

In summary, this paper makes the following contributions:

- We propose a novel Flow-Sensitive Graph Neural Networks (FS-GNN) to support effective detection of memory-related vulnerabilities.
- We formalize vulnerability detection as a fine-grained node classification problem to identify suspicious vulnerable statements.
- We evaluate *MVD* on our constructed dataset, and the results show that *MVD* can effectively detect memory-related vulnerabilities over state-of-the-art vulnerability detection approaches (including three DL-based and five static analysis-based approaches).

2 BASICS AND MOTIVATION

2.1 Definitions

Program Dependence Graph. Given a program, all the program statements and dependencies among statements constitute a *Program Dependence Graph (PDG)* [26]. *PDG* includes two types of edges: data dependency edges which reflect the influence of one variable on another and control dependency edges which reflect the influence of predicates on the values of variables.

Call Graph. Given a program, its *Call Graph (CG)* [53] indicates a series of function calls from call sites (caller) to the callee.

Graph Neural Networks. Due to the outstanding ability in processing graph data structures, *Graph Neural Networks (GNNs)* have been used in a variety of data-driven software engineering (SE) tasks (e.g., code representation [1], clone detection [65], and bug localization [48]) and have achieved great breakthroughs. The goal of GNNs is to train a parametric function via message passing between the nodes of graphs for downstream tasks, i.e., graph classification, node classification, and link prediction.

2.2 Motivating Examples

Figure 1 shows a typical *use-after-free* vulnerability CVE-2019-15920 [5] in *Linux Kernel*. The vulnerable function `SMB2_read` has been simplified for a clear illustration. We can observe that the memory space pointed by the pointer `req` is released in advance by the memory release statement `cifs_small_buf_release(req)` at line 5, while it is still used at lines 8-13. This operation may allow attackers to write malicious data. To fix this vulnerability, the pointer `req` should be released after it is used for the last time (e.g., line 14). Despite the support of precise data dependence analysis, this vulnerability cannot be easily detected by some static analysis-based approaches because they may not know `mempool_free()` at line 24 is a user-defined memory deallocation function.

```
1  int SMB2_read(const unsigned int xid, struct cifs_io_parms
   ↪ *io_parms, unsigned int *nbytes, char **buf, int *buf_type)
2  {
3      struct smb2_read_plain_req *req = NULL;
4      ...
5  -   cifs_small_buf_release(req);
6      if (rc) {
7          if (rc != -ENODATA) {
8              trace_smb3_read_err(xid, req->PersistentFileId,
                 ↪ io_parms->tcon->tid, ses->Suid, io_parms->offset,
                 ↪ io_parms->length, rc);
9          } else {
10             trace_smb3_read_done(xid, req->PersistentFileId,
                 ↪ io_parms->tcon->tid, ses->Suid, io_parms->offset,
                 ↪ 0);
11         return rc == -ENODATA ? 0 : rc;
12     } else {
13         trace_smb3_read_done(xid, req->PersistentFileId,
             ↪ io_parms->tcon->tid, ses->Suid, io_parms->offset,
             ↪ io_parms->length);
14  +   cifs_small_buf_release(req);
15      ...
16      return rc;
17 }
18 void cifs_small_buf_release(void *buf_to_free)
19 {
20     if (buf_to_free == NULL) {
21         cifs_dbg(FYI, "Null buffer passed to
             ↪ cifs_small_buf_release\n");
22         return;
23     }
24     mempool_free(buf_to_free, cifs_sm_req_poolp);
25     atomic_dec(&smBufAllocCount);
26     return;
27 }
```

Figure 1: A *Use-After-Free* Vulnerability (CVE-2019-15920) in *Linux Kernel*

From this example, we can draw the following observations:

Observation 1. Comprehensive and precise interprocedural flow analysis is necessary. As shown in Figure 1, we can find that the program semantics of vulnerable code and non-vulnerable code are different. In the vulnerable code, vulnerable statement at line 5 releases req when req is still being used after that, while in the non-vulnerable code, req is released by the patched statement at line 14 only when req is no longer used. However, due to the lack of interprocedural analysis, critical program semantics (i.e., memory deallocation via mempool_free, which is involved in the function call to cifs_small_buf_release(req) at line 5) are ignored by a number of deep learning-based approaches [17, 44, 64, 71], resulting in incomplete program semantic modeling towards vulnerable statement at line 5.

In our approach, we extend basic Program Dependence Graph (PDG) with additional semantic information like call relations and return values obtained from Call Graph (CG) [53] to capture comprehensive and precise interprocedural program semantics. With such rich information, features of memory-related vulnerabilities can be extracted for more effective detection.

Observation 2. Sensitive contextual information within flows helps to refine detection granularity. As shown in the motivating example, the premise of identifying the statement at line 5 as vulnerable is that we should know in advance that req released by this statement will be used later. Thus, to distinguish vulnerable statements from others, the neural networks used as detection models should be able to capture sensitive contextual information within flows of vulnerable statements for inference. However, due to the limitations of popular DL models[45, 46, 64, 71] in handling multiple relations, rich contextual information are

lost during the process of model training. For example, *FUNDED* [64] only considers one-directional transmission of multiple relations such as control- and data-flows, and adopts GGNN [43] to learn vulnerability patterns for detection. While it is successful in *function-level* vulnerability detection, it loses some important contextual information from output flows, e.g., the data-flow information within Edge 5->8 will not be used for feature update of vulnerable statement at line 5. Thus, it is hard for *FUNDED* to identify the statement line 5 as vulnerable because critical output flow information (i.e., using req after freeing it) is not preserved in cifs_small_buf_release(req).

Based on the above observations, we propose a novel model, FS-GNN, for effectively detecting memory-related vulnerabilities. FS-GNN is a novel flow-sensitive graph neural network to jointly embed both statements and flow information for better information propagation between statements. With FS-GNN, rich contextual semantics of neighbors are aggregated through multiple relations to update the embedding of the central node.

3 OUR APPROACH: *MVD*

Figure 2 shows the overview of *MVD*. It consists of two phases: training phase and detection phase.

The training phase includes three steps. In step 1, *MVD* constructs the Program Dependence Graph (PDG) based on the control- and data-flow of the program. To capture comprehensive and precise program semantics, *MVD* extends the PDG with additional semantic information like call relations and return values obtained from Call Graph (CG) [53] to conduct interprocedural analysis. Furthermore, to reduce irrelevant semantic noise, *MVD* conduct program slicing [59, 67] from the program points of interest. In step 2, *MVD* uses Doc2Vec [39] to transform the statements of each slice into low-dimensional vector representations. In step 3, *MVD* uses Flow-Sensitive Graph Neural Networks (FS-GNN) to jointly embed nodes and relations 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*.

For the detection phase, with the interprocedural analysis, the control- and data-dependence of a target program is first extracted for program slicing to capture precise program semantics related to memory usage. Then, for each slice, both its unstructured (i.e., statement embedding by Doc2Vec) 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.

3.1 Feature Extraction

First, we use the static analysis tool, *Joern* [68], to parse source code and construct the Program Dependence Graph (PDG). Then, we extend the PDG with additional semantic information like call relations and return values obtained from Call Graph (CG) to conduct interprocedural analysis, which preserves comprehensive control- and data-flow information. However, since a 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 reduce the performance of identifying key features. Thus, *MVD* adopts program slicing [67] to perform
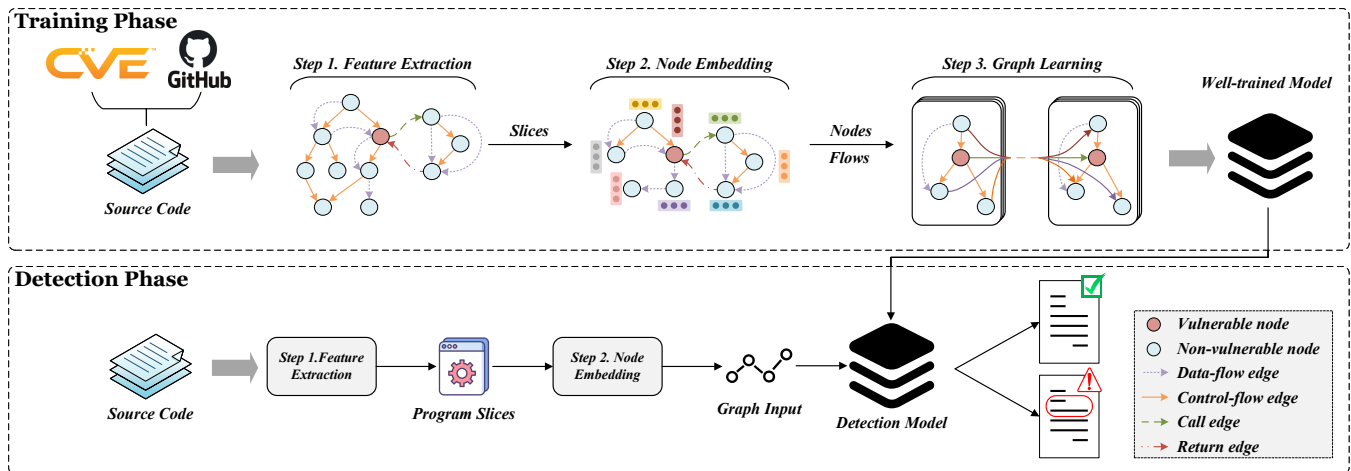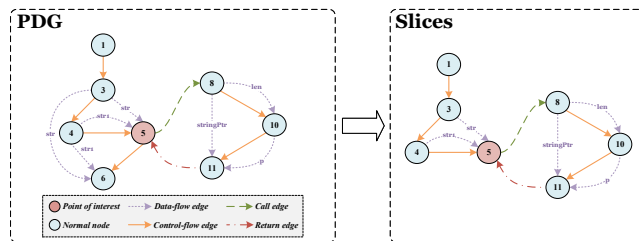
Figure 2: Overview of *MVD*

```
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

Figure 3: Details of Feature Extraction

backward and forward slicing from the program point of interest to avoid noise induced by irrelevant statements.

To ensure that the generated 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 [20, 45, 46, 72], 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 conclude 217 *system API calls* from several static memory detectors [24, 56] as slicing criteria for extracting vulnerable code snippets. For *pointer variable*, it has been widely adopted by traditional static analysis-based approaches [24, 40, 60].

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, and usually forward control-dependence will induce a great deal of irrelevant statements.

Figure 3 provides an example to show the process of feature extraction. As shown in Figure 3a, 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, the control- and data-flow information of the vulnerable program is first extracted to construct the PDG of the program, which is shown in Figure 3b. Then, based on interprocedural analysis, the call relation (i.e., Edge 5->8) and return value (i.e., Edge 11->5) information is added. 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

After feature extraction, we transform all statement nodes in the graph into low-dimensional vectors as input to the graph neural network models.

We use Doc2Vec [39] to represent code statements as vectors since it is a widely-used technique [20, 28] to encode the documents (i.e., code statements) instead of an individual word (e.g., a variable) into a fixed-length vector. Then, the vectors of the statement and context words are input to the hidden layer to obtain the intermediate vector as the input of *softmax* layer. Finally, in the inference stage, the vector of a given statement is achieved through Stochastic Gradient Descent (SGD) [57]. In this way, the Doc2Vec can provide a more precise embedding of the code statement that will preserve the semantic information.

3.3 Graph Learning

To train a model which can learn implicit vulnerability patterns from source code and identify suspicious vulnerable statements, we construct a novel graph learning framework, Flow-Sensitive Graph Neural Network (FS-GNN) for graph learning. The details of our approach are shown in Figure 4. 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 [14] 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:

$$h_v = f\left(\sum_{(u,r)\in\mathcal{N}(v)} W_{\lambda(r)}\phi(x_u, z_r)\right) \tag{1}$$

where h_v denotes the updated representation of node v. $\mathcal{N}(v)$ is a set of immediate neighbors of v for its outgoing edges. $\phi(\cdot)$ is a composition operator, including subtraction, multiplication, and circular-correlation. x_u and z_r denotes initial features for node u (encoded by Doc2Vec) and edge r, respectively. Similar to traditional Relational Graph Neural Networks (RGCN) [54], initial edge representation for edge r can be encoded by basis decomposition [54] as $z_r = \sum_{b=1}^{\mathcal{B}} \alpha_{br} v_b$, where $v_{b\in\mathcal{B}}$ 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. $W_{\lambda(r)}$ 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., $W_{\lambda(r)} = W_O$ when r is an initial edge, and $W_{\lambda(r)} = W_I$ when r is an inverse edge).

Similarly, the edge embedding h_r of edge r can be updated by $h_r = W_{rel}z_r$, where W_{rel} is a learnable transformation matrix which projects all the relations to the same embedding space as nodes.

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

$$h_v^{l+1} = f\left(\sum_{(u,r)\in\mathcal{N}(v)} W_{\lambda(r)}^l \phi(h_u^l, h_r^l)\right) \tag{2}$$

$$h_r^{l+1} = W_{rel}^l z_r^l \tag{3}$$

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 4, 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, 5 are embedded jointly with their incoming edges (i.e., Edges 3->1, 3->4, 3->5) by Eq. 2 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 3, although we have filtered out some irrelevant nodes by program slicing, the number of non-vulnerable nodes (i.e., Node 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* [69], a *graph-level* oversampling framework, as the basic component for our resampling.

Concretely, it contains two steps: (1) node generation, and (2) edge generation. Firstly, to generate high-quality synthetic nodes, we utilize the widely-used *SMOTE* [18] 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 [69] 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, 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)[2].

Owing to the contribution of resampling, the proportion of memory-related vulnerable statements increases, avoiding the well-trained detection model biased caused by imbalanced distribution of vulnerable nodes and non vulnerable nodes. For example, in Figure 4, 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 Eq. 2. 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):

$$\min_\theta \mathcal{L} = -\sum_{G\in\mathcal{G}} \frac{1}{|\mathcal{V}|} \sum_{i\in\mathcal{V}} \sum_{k=1}^{K} t_{ik} \ln h_{ik}^{(L)} \tag{4}$$

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

---

[2]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.
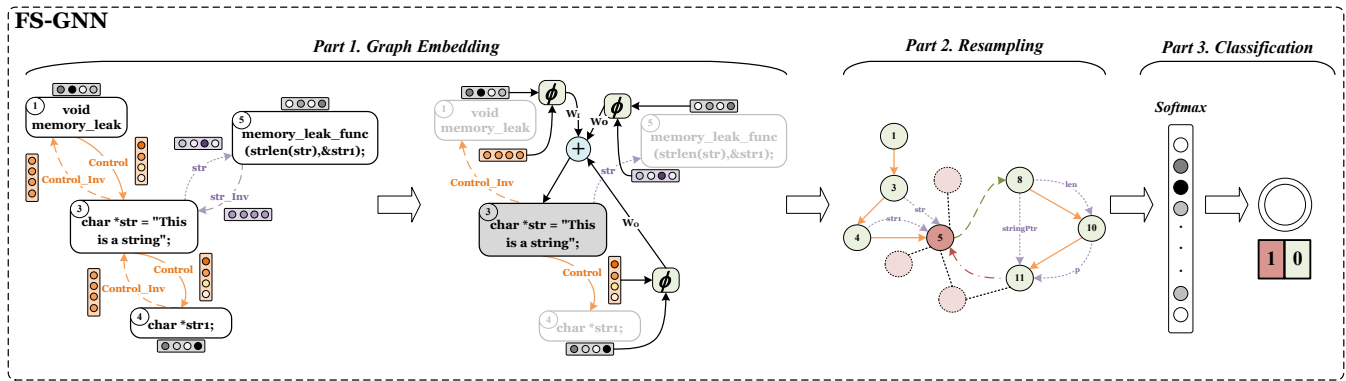
Figure 4: Graph Learning with FS-GNN

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 identify suspicious statements.

Specifically, similar to training phase, program semantics reflected in the graph representations of source code are captured through interprocedural analysis. In order 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 Doc2Vec (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 EXPERIMENTS

4.1 Research Questions

RQ1. How effective is *MVD* in detecting memory-related vulnerabilities compared to existing deep learning-based vulnerability detection approaches?

Works most relevant to *MVD* are deep learning-based vulnerability detection approaches. By investigating this RQ, we aim to answer how well does *MVD* perform in comparison with the state-of-the-art deep learning-based approaches in memory-related vulnerability detection.

RQ2. How effective is *MVD* in detecting memory-related vulnerabilities compared to 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 *MVD* perform compared with existing static analysis-based detectors.

RQ3. How effective is FS-GNN for memory-related vulnerability detection?

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 and precise program semantics. Different from RQ1, we aim to show whether sensitive contextual information captured by FS-GNN contributes to memory-related vulnerability detection in comparison with other popular GNNs (i.e., evaluating the effectiveness of fully utilizing flow information).

RQ4. How efficient are *MVD* and baselines in terms of their time cost for detecting memory-related vulnerabilities?

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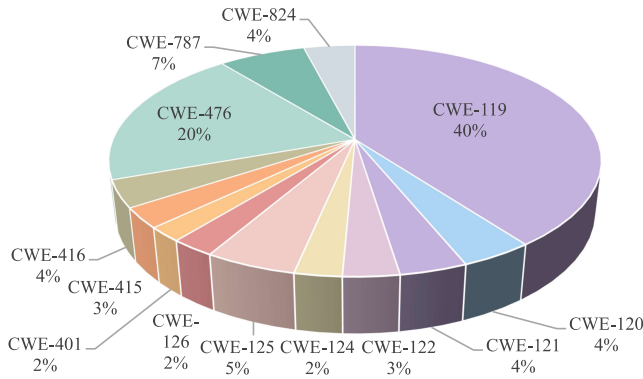
4.2 Experiment Setup

4.2.1 Dataset. Since existing vulnerability datasets are either not tailored for memory-related vulnerabilities [17, 20, 25, 29, 71], or not sufficient for training deep learning models (e.g., SPEC CINT2000 [32]), we *manually* constructed a new vulnerability dataset which covers 13 common memory-related vulnerabilities (including CWE-119, -120, -121, -122, -124, -125, -126, -401, -415, -416, -476, -787, and -824) for model training and evaluation. Our dataset is based on two widely-adopted sources: (1) SARD [13], a well-known sample vulnerability data set, and (2) CVE [2], a famous vulnerability database. In this work, we focused on C/C++ programs due to their frequent memory problems caused by low-level control of memory [63] and adopted vulnerability types (i.e., CWE-IDs [3]) as our search criteria to collect memory-related vulnerabilities from SARD and CVE. For real-world vulnerabilities, we only considered CVEs which contain source code and from which we collect both vulnerable functions and corresponding patched functions. For SARD, we collected all test cases labeled as "bad".

The statistics of the vulnerable programs in our dataset are shown in Table 1. It includes 1,208 real-world vulnerabilities in CVE, covering 10 open-source C/C++ projects which are widely adopted as target projects by prior works [27, 45, 46, 71], and 3,145 vulnerable samples (i.e., test cases) in SARD. Column 2 represents the scope of project versions affected by vulnerabilities in our dataset. Column

Table 1: Details of vulnerability dataset.

Project	#Version	#Samples	#Vertices	#Edges
Linux Kernel	2.6-4.20	868	26,917	29,512
FFmpeg	0.5-4.1	73	1,971	2,168
Asterisk	1.4-16.14	18	468	502
Libarchive	2.2-3.4	11	235	269
Libming	0.4.7	7	119	141
LibTIFF	3.8-4.0	24	584	639
Libav	12.3	16	526	573
LibPNG	1.0.x-1.6.x	13	392	447
QEMU	0.9-4.3	121	4,711	5,308
Wireshark	1.2-3.2	57	2,056	2,190
SARD	-	3,145	11,237	13,049
Total	-	4,353	49,216	54,798

Figure 5: Distribution of Vulnerability Types

3 denotes the number of vulnerable samples. A vulnerable sample may contain one or more vulnerable functions. Column 4 and Column 5 are the number of nodes and edges in slices, respectively. Furthermore, the distribution of different types of memory-related vulnerabilities in our dataset is shown in Figure 5, with CWE-119 (Improper Restriction of Operations within the Bounds of a Memory Buffer) accounting for the highest percentage at 40% (including 1731 vulnerable samples).

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. 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 (i.e., starting with "-" in *diff* files), 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 [6], memory leaks because allocated memory can not 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 [49] (e.g., memory allocation or deallocation statements). In order to avoid introducing artificial deviation, two postgraduates and one Ph.D participated in this labeling process. If two postgraduates disagreed on the label of the same sample, the sample would be forwarded to the Ph.D evaluator for further investigation.

4.2.3 Baseline Methods. To answer the first research question, we selected three state-of-the-art DL-based vulnerability detection techniques, i.e., *VulDeePecker* [46], *SySeVR* [45], and *Devign* [71]. *VulDeePecker* and *SySeVR* represented source code as sequences and used BiLSTM model for vulnerability detection at the *slice-level*. *Devign* constructed a joint graph structure (including AST, CFG, DFG, and code sequences) and used GGNN model to detect vulnerabilities at the *function-level*. They are widely adopted as baselines in recent works [20, 44, 64] and have been shown to be effective in detecting memory-related vulnerabilities [20] even though they are designed for generic vulnerability detection.

To investigate RQ2, we selected five popular static analysis-based vulnerability detectors, i.e., *PCA* [40], *Saber* [60], *Flawfinder* [8], *RATS* [12], and *Infer* [9]. They have shown relatively good performance on memory-related vulnerabilities and are widely adopted as baselines in prior works [20, 24, 45, 46].

4.2.4 Implementation. We implemented *MVD* in Python using PyTorch [11]. Our experiments were performed with the Nvidia Graphics Tesla T4 GPU, installed with Ubuntu 18.04, CUDA 10.1.

The neural networks are trained in a batch-wise fashion until converging and the batch size is set to 32. The dimension of the vector representation of each node is set to 100 and the dropout is set to 0.1. ADAM [36] 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.

For RQ1, it is unfair to compare *MVD* with other DL-based approaches because of our finer granularity. Thus, we used the *function-level* as a compromise formula, i.e., if a vulnerable statement was identified correctly by *MVD*, we would consider the function it belonged to was also detected correctly. For DL-based baselines, we respectively used the vulnerable and non-vulnerable functions as positive and negative samples to train the detection models. For *MVD*, we only trained the detection model based on vulnerable functions (i.e., vulnerable statements are deemed as positive samples, while non-vulnerable statements are deemed as negative samples). We randomly chose 80% of the programs for training and the remaining 20% for testing. To make sure that our model was fine-tuned, we used *ten-fold cross-validation* to evaluate the generalization ability of each approach. In RQ2, we evaluated *MVD* and static analysis-based approaches at the *function-level* and *statement-level*, respectively. At the *function-level*, we adopted the same experimental setup as RQ1. At the *statement-level*, we evaluated each approach by randomly selecting 30 latest real-world vulnerabilities (reported in 2021) from our dataset, covering five

Table 2: Comparison with DL-based approaches

Approach	A (%)	P (%)	R (%)	F1 (%)
VulDeePecker	60.9	51.4	35.1	41.7
SySeVR	63.4	53.3	62.9	57.7
Devign	68.3	54.8	66.1	59.9
MVD	**74.1**	**61.5**	**69.4**	**65.2**

common memory-related vulnerabilities (including *Memory Leak* (ML), *Double Free* (DF), *Buffer Overflow* (BO), *Use-After-Free* (UAF), and *Out-of-bounds Read/Write* (OR/W)). These vulnerabilities are representative because they cover the vast majority of the vulnerability types in our dataset. For example, *Buffer Overflow* (BO) corresponds to multiple CWEs, including CWE-119, -120, -121, and -122. To ensure that the trained model is tested on "unseen" programs, we excluded these samples from the training set of *MVD*. For answering RQ3, we respectively replaced our FS-GNN model with three famous GNN models, including GCN [37], GGNN [43], and RGCN [54], to evaluate the contribution of each model to memory-related vulnerability detection. For answering RQ4, we recorded the average training and detection time of each approach in RQ1 and RQ2 to evaluate time cost of *MVD* and baselines.

4.3 Evaluation Metrics

We used the following evaluation metrics to measure the effectiveness of our model:

Accuracy (A) evaluates the performance that how many instances can be correctly labeled. It is calculated as:

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{5}$$

Precision (P) is the fraction of true vulnerabilities among the detected ones. It is defined as:

$$Precision = \frac{TP}{TP + FP} \tag{6}$$

Recall (R) measures how many vulnerabilities can be correctly detected. It is calculated as:

$$Recall = \frac{TP}{TP + FN} \tag{7}$$

F1 score (F1) is the harmonic mean of *Recall* and *Precision*, and can be calculated as:

$$F1\ score = 2\frac{Recall \cdot Precision}{Recall + Precision} \tag{8}$$

5 EXPERIMENTAL RESULTS

5.1 RQ1: *MVD* VS. DL-Based Approaches

Table 2 shows the overall results of each deep learning-based approach in terms of the aforementioned evaluation metrics. Overall, *MVD* achieves better results and outperforms all of the three referred deep learning-based approaches. On average, for *MVD*, the Accuracy is 74.1%, the Precision is 61.5%, the Recall is 69.4%, and the F1 score is 65.2%. Moreover, in terms of all metrics, *MVD* can improve the best performed baseline *Devign* by 5.0%-12.2%.

<u>*MVD* vs. *VulDeePecker*.</u> As shown in Table 2, our approach improves Accuracy, Precision, and F1 score over *VulDeePecker* by

```c
void invalid_memory_access (){
    int i=1;
    char *buf,*c;
    while(i>0) {
        buf = (char *) malloc (25 * sizeof(char));
        if(buf!=NULL)
            strcpy(buf,"This is String");
            free(buf);
        c = buf;
        i++;
        if(i>=10)
            break;}
    psink = c;
}
```

(a) A Vulnerability Missed by *VulDeePecker*

```c
static int __init init_msp_flash(void){
    ...
    msp_parts[i] = kcalloc(...), GFP_KERNEL);
    ...
    if (msp_maps[i].virt == NULL) {
        kfree(msp_parts[i]);
        goto cleanup_loop;}
    if (!msp_maps[i].name) {
        kfree(msp_parts[i]);
        goto cleanup_loop;}
    ...
}
```

(b) A Non-vulnerable Code Sample Misidentified by *SySeVR*

```c
static bool try_merge_free_space(...){
    ...
    right_info = tree_search_offset(ctl, offset + bytes, 0, 0);
    if (right_info && rb_prev(&right_info->offset_index))
        left_info = rb_entry(rb_prev(&right_info->offset_index),
        ↪ struct btrfs_free_space, offset_index);
    else
        left_info = tree_search_offset(ctl, offset - 1, 0, 0);
    if (...) { ...
        kmem_cache_free(btrfs_free_space_cachep, right_info);
        merged = true;}
    if (...) { ...
        info->offset = left_info->offse;
        info->bytes += left_info->bytes}
    return merged;
}
```

(c) CVE-2019-19448 [7] Missed by *Devign*

Figure 6: A Case study of RQ1

21.7%, 19.6%, and 56.4%. Specifically, *VulDeePecker* achieves the Recall of only 35.1%. By contrast, the Recall of *MVD* is as high as 69.4%, nearly double (1.98x). Poor Recall indicates a great deal of vulnerabilities can not be detected by *VulDeePecker*. A main reason is that *VulDeePecker* only takes data-flow into account without regard to control-flow information. For example, as shown in Figure 6a, it contains a *invalid memory access* vulnerability at <u>line 9</u> because *buf* has been freed at <u>line 8</u> in an infinite *while* loop. However, it is missed by *VulDeePecker* in our experiment because without control-flow information, semantics of different branch structures will be ignored.

<u>*MVD* vs. *SySeVR*.</u> We can observe from Table 2 that, in spite of significant improvement (1.79x) in Recall (62.9%) in comparison with *VulDeePecker*, *SySeVR* still behaves worse than *MVD* in terms of each metric. Particularly, *MVD* improves Precision over *SySeVR* by 15.4%. The root cause for this performance gap is that *SySeVR* cannot overcome the main limitation of sequence models (e.g., BiLSTM) in program semantics modeling. For example, in Figure 6b, there is a non-vulnerable code sample from *Linux Kernel*, which is misidentified by *SySeVR*. Although *SySeVR* captures control- and data-dependence relations between statements by constructing

Table 3: Comparison with static analysis-based approaches

Approach	A (%)	P (%)	R (%)	F1 (%)
PCA	65.2	48.9	61.1	54.3
Saber	64.4	47.6	59.2	52.8
Flawfinder	61.1	18.2	23.5	20.5
RATS	56.3	7.9	11.6	9.4
Infer	50.7	33.1	54.8	41.3
MVD	**67.6**	**54.8**	**63.6**	**58.9**

Control Flow Graph (CFG) and Data Flow Graph (DFG), program semantics implied in these information can not be utilized because *SySeVR* treats complex code structures as a sequential sequence of tokens, which omits the control flow divergence. Thus, this sample is misidentified as *double free* because `msp_parts[i]` is considered to be freed twice by *kfree* at line 6 and line 9.
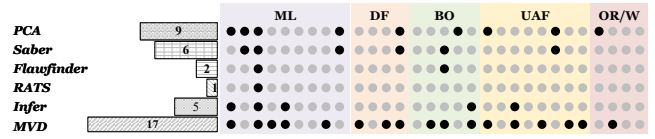
MVD vs. *Devign*. *MVD* also outperforms the best performed baseline *Devign*. It indicates that although powerful performance of GNN in inferring potential vulnerability semantics from graph representation of the program makes *Devign* outstanding, the underutilization of flow information still restricts the performance of *Devign* in detecting more complex memory-related vulnerabilities. For example, as shown in Figure 6c, it is a real-world vulnerability (CVE-2019-19448 [7]) in *Linux Kernel* missed by *Devign*. It may lead to *arbitrary address free* or *double free* vulnerability by attacker because in certain situations, the pointer `left_info` at line 7 can be the same as `right_info` at line 3. However, after `right_info` has been freed at line 9, `left_info` which is the same as `right_info` will be used at line 12-13 again. There are two main reasons why this vulnerability cannot be detected by *Devign*. On the one hand, due to the lack of interprocedural analysis, precise program semantics like memory free (`kmem_cache_free` at line 9) are hard to be captured by *Devign*, causing imprecise semantic modeling. On the other hand, since *Devign* processes multiple flows information by passing information in each individual graph and then aggregates them across graphs, `left_info` and `right_info` are treated as different data-flows, which causes complex semantic relations difficult to be preserved.

> **Answer to RQ1:** *In comparison with the popular DL-based approaches, MVD achieves better detection performance by* fully utilizing flow information via interprocedural analysis and FS-GNN.

5.2 RQ2: *MVD* VS. Static Analysis-Based Approaches

Table 3 shows the experimental results of *MVD* and the static analysis-based techniques. Overall, *MVD* outperforms all baselines with regard to the evaluation metrics.

Among all static analysis-based baselines, *PCA* and *Saber* obtain relatively better detection performance. *PCA* achieves the highest Precision (48.9%) and Recall (61.1%) due to its consideration of global variables and accurate interprocedural flow analysis. Our approach still improves *PCA* by 4.1% in terms of Recall, and by 12.1% in terms

Figure 7: The Number of Memory-Related Vulnerabilities in Real-World Projects Detected by Each Approach (ML: Memory Leak; DF: Double Free; BO: Buffer Overflow; UAF: Use-After-Free; OR/W: Out-of-bounds Read/Write)

```
1   static int l2tp_ip_bind(struct sock *sk, struct sockaddr *uaddr,
    ↪   int addr_len){
2       ...
3   -   if (!sock_flag(sk, SOCK_ZAPPED))
4   -       return -EINVAL;
5       ...
6       read_unlock_bh(&l2tp_ip_lock);
7       lock_sock(sk);
8   +   if (!sock_flag(sk, SOCK_ZAPPED))
9   +       goto out;
10      ...
11  }
```

Figure 8: A *Use-After-Free* Vulnerability (CVE-2016-10200 [4]) Missed by All Static Memory Detectors We Compared

of Precision. The reason is that static analysis-based approaches mainly rely on well-defined vulnerability rules or patterns handcrafted 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, instead of explicit rules or specifications, making it more effective in real-world scenarios.

Figure 7 shows the detection results of each approach for five common memory-related vulnerabilities in real-world projects. These vulnerabilities are randomly selected from our dataset. Each detected individual vulnerability successfully is labeled by a dark circle and the bars on the left-hand side are the total number of successfully detected vulnerabilities. Overall, *MVD* outperforms all baselines by detecting 17 out of 30 vulnerabilities, including nine vulnerabilities cannot be detected by baselines. Especially, in comparison with the best performed baseline *PCA*, our approach can detect eight more vulnerabilities. For example, as shown in Figure 8, it is a *use-after-free* vulnerability because a concurrent call could modify the socket flags between `sock_flag(sk, SOCK_ZAPPED)` at line 4 and `lock_sock()` at line 8, allowing local users to gain privileges or cause a denial of service by making multiple bind system calls without properly ascertaining whether a socket has the `SOCK_ZAPPED` status. Unfortunately, it is missed by all the static memory detectors because they cannot detect *use-after-free* caused by race condition through static analysis only. In our approach, it can be correctly detected because of the advantage of deep learning models in mining implicit vulnerability patterns.

Table 4: Contributions of different graph neural networks

Approach	A (%)	P (%)	R (%)	F1 (%)
GCN	61.2	17.3	8.2	11.1
GGNN	69.4	41.8	52.5	46.5
RGCN	72.7	49.3	58.1	53.3
FS-GNN	**77.5**	**56.4**	**62.9**	**59.5**

> **Answer to RQ2:** *With the advantage of deep learning models in mining implicit vulnerability patterns, MVD performs better in comparison with the popular static analysis-based approaches.*

5.3 RQ3: FS-GNN VS. Other GNNs

Table 4 shows the results of different GNNs. We observe that FS-GNN can improve the best performed baseline RGCN by 6.7%-14.4%. There are mainly two reasons for this. First, 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. In addition, 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.

Furthermore, we observe that the performance of GCN is poor. The main reason is that the neglection of edge types leads to the missing of structured code features (e.g., control- and data-flow). Without accurate control- and data-flow information, the performance of memory-related vulnerability detection drops sharply.

> **Answer to RQ3:** *FS-GNN can effectively contribute to the performance of MVD, as it can better capture the structured information of vulnerable code.*

5.4 RQ4: Efficiency

Table 5 lists the time cost in seconds of each approach in training and detecting vulnerabilities. The results show that in comparison with the popular static analysis-based approaches, *MVD* achieves less time cost over other approaches, except *PCA*. This is because *PCA* speeds up data dependence computation through sacrificing partial detection precision.

Among the four deep learning-based approaches, *VulDeePecker* incurs the least training and detection time because it only considers data-flows and uses a simple sequence model, BiLSTM, for model training. However, combining with the results in Table 2, 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. Compared with other learning-based approaches, *MVD* spends relatively longer training and detection time (excluding *Devign*) because learning complicated program semantics in graphs is more time-consuming than in sequence. However, *MVD* yields better detection results. In fact, due to the characteristic that deep learning models can be trained offline, their training cost 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.

> **Answer to RQ4:** *In spite of a great deal of training time, MVD achieves relatively shorter detection time with better detection results, making a trade-off between accuracy and efficiency.*

6 THREATS TO VALIDITY

External validity. The main external threat to our study is the generalizability of our experiment results. We respectively investigated 4,353 vulnerable samples from 10 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, our experiments are limited to memory-related vulnerabilities in C/C++ programs. 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.

Internal validity. Internal validity in our experiment relates to two factors. The first is our imperfect node labeling. In this work, we manually labeled nodes which 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[3] since it is closed-source. we try our best to build and tune the *Devign* parameters on our dataset.

7 RELATED WORKS

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

Static Analysis-Based. Static analysis-based approaches aim to detect vulnerabilities based on specific vulnerability patterns

---
[3]https://github.com/epicosy/devign

Table 5: Time cost in <u>seconds</u> of different approaches. N/A: Not Applicable

Method	MVD	VulDeePecker	SySeVR	Devign	PCA	Saber	Infer	Flawfinder	RATS
Training Time(s)	2386.2	1019.5	1833.9	2583.7	N/A	N/A	N/A	N/A	N/A
Detection Time(s)	10.4	8.1	9.7	11.9	**9.2**	11.8	145.8	17.4	20.6

or memory state model. Cherem et al. [21] proposed a solution named *FastCheck*, which reduces the memory leak analysis to a reachability problem over the guarded value-flow graph. Sui et al. [60, 61] proposed *Saber*, a full-sparse value-flow graph (SVFG) based approach, to achieve the def-use chains and value-flow of the memory for pointer analysis. Shi et al. [56] proposed *Pinpoint* to optimize widely-used sparse value-flow analysis through decomposing the cost of high-precision points-to analysis. Fan et al. [24] presented *SMOKE*, a staged approach for memory leak detection, to solve the scalability problem at industrial scale. Li et al. [40] proposed *PCA*, a static interprocedural data dependency analyzer, to speed up data dependency computation through partial call-path analysis. Differently, our approach can learn vulnerability features from large amounts of vulnerability data without requiring any prior knowledge of vulnerabilities.

Dynamic Analysis-Based. Dynamic detection methods run the source code and dynamically track the allocation, use and release of memory at the run-time. *LEAKPOINT* [22] monitored the state of memory objects based on stain analysis and tracked the last used location of memory and the location where references were lost. *DoubleTake* [47] 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* [35] used the processor's monitoring unit (PMU) to track the access instructions to heap memory. Then, it calculated the staleness of heap objects and executed relevant instructions again to capture memory leakage during program execution. At the binary level, some dynamic analysis-based tools such as *Valgrind* [50], *Dr.Memory* [15], *AddressSanitizer* [55] also perform well. They track memory allocation and deallocation during a program's execution, and detect leaks by scanning the program's heap for memory blocks that no pointer points to. Unlike dynamic analysis-based approaches, our approach does not require the execution of programs.

Learning-Based. With the advance of machine learning (ML) and especially deep learning (DL) models, some approaches are proposed to automatically learn explicit or implicit vulnerability features from known vulnerabilities to identify unseen vulnerabilities in projects. Li et al. [45, 46] proposed two slice-based vulnerability detection approaches, *VulDeePecker* and *SySeVR*, to learn syntax and semantic information of vulnerable code. They represented source code as sequences at the *slice-level* and used RNN (e.g., LSTM and BGRU) to train their detection models. Zou et al. [72] proposed an attention-based multi-class vulnerability detection approach, *μVulDeePecker*, to pinpoint types of vulnerabilities. It introduced *code attention* to accommodate information useful for learning local features and used a building-block BiLSTM to fuse different code features. Zhou et al. [71] proposed a graph neural network-based

vulnerability detection model through learning on a rich set of code semantic representations. Cheng et al. [20] embedded both textual and structured information of code into a comprehensive code representation and leveraged a GCN to perform the graph classification. Wang et at. [64] proposed *FUNDED*, a GNN-based vulnerability detection approaches. They combined nine mainstream graphs to extract finer program semantics and extended GGNN to model multiple code relationships. Different from existing learning-based vulnerability detection approaches, our approach aims to leverage rich flow information to support fine-grained memory-related vulnerability detection via the novel flow-sensitive graph neural networks.

8 CONCLUSION

In this paper, we propose *MVD* to detect memory-related vulnerability statements that are related to sensitive operations. *MVD* employs a new graph neural network-based approach that leverages the flow-sensitive graph neural network (FS-GNN) to jointly embed both unstructured information and structured information for preserving high-level program semantics to learn implicit vulnerability patterns. The experimental results show the effectiveness of our approach by comparing our approach with three state-of-the-art deep learning-based techniques and five popular static analysis-based memory detectors.

In the near future, we plan to compare our approach with more DL-based approaches (e.g., DeepWukong) and static memory detectors on a larger dataset to gain more insights. In addition, we aim to investigate other code representation techniques to efficiently model flow information specific to memory-related vulnerabilities.

REFERENCES

[1] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net.
[2] August. 2021 (last accessed). Common Vulnerabilities and Exposures. https://cve.mitre.org/.
[3] August. 2021 (last accessed). Common Weakness Enumeration. https://cwe.mitre.org/.

[4] August. 2021 (last accessed). CVE-2016-10200. https://github.com/torvalds/linux/commit/32c231164b762dddefa13af5a0101032c70b50ef.
[5] August. 2021 (last accessed). CVE-2019-15920. https://github.com/torvalds/linux/commit/088aaf17aa79300cab14dbee2569c58cfafd7d6e.
[6] August. 2021 (last accessed). CVE-2019-19083. https://github.com/torvalds/linux/commit/055e547478a11a6360c7ce05e2afc3e366968a12.
[7] August. 2021 (last accessed). CVE-2019-19448. https://github.com/bobfuzzer/CVE/tree/master/CVE-2019-19448.
[8] August. 2021 (last accessed). Flawfinder. http://www.dwheeler.com/flawfinder.
[9] August. 2021 (last accessed). Infer. https://fbinfer.com.
[10] August. 2021 (last accessed). Linux Kernel. https://www.kernel.org/.
[11] August. 2021 (last accessed). PyTorch. https://pytorch.org/.
[12] August. 2021 (last accessed). Rough Audit Tool for Security. https://code.google.com/archive/p/rough-auditing-tool-for-security.
[13] August. 2021 (last accessed). Software Assurance Reference Dataset. https://samate.nist.gov/SARD/index.php.
[14] Antoine Bordes, Nicolas Usunier, Alberto García-Durán, Jason Weston, and Oksana Yakhnenko. 2013. Translating Embeddings for Modeling Multi-relational Data. In *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States*. 2787–2795.
[15] Derek Bruening and Qin Zhao. 2011. Practical memory checking with Dr. Memory. In *Proceedings of the CGO 2011, The 9th International Symposium on Code Generation and Optimization, Chamonix, France, April 2-6, 2011*. IEEE Computer Society, 213–223.
[16] 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.
[17] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2020. Deep Learning based Vulnerability Detection: Are We There Yet? *arXiv preprint arXiv:2009.07235* (2020).
[18] 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.
[19] Zhe Chen, Chong Wang, Junqi Yan, Yulei Sui, and Jingling Xue. 2021. Runtime detection of memory errors with smart status. In *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*. ACM, 296–308.
[20] 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.
[21] Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. 2007. Practical memory leak detection using guarded value-flow analysis. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*. ACM, 480–491.
[22] James A. Clause and Alessandro Orso. 2010. LEAKPOINT: pinpointing the causes of memory leaks. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*. ACM, 515–524.
[23] 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. Software Eng.* 47, 1 (2021), 67–85.
[24] 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 2019, Montreal, QC, Canada, May 25-31, 2019*. IEEE / ACM, 72–82.
[25] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. In *MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020*. ACM, 508–512.
[26] 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.
[27] Aayush Garg, Renzo Degiovanni, Matthieu Jimenez, Maxime Cordy, Mike Papadakis, and Yves Le Traon. 2020. Learning To Predict Vulnerabilities From Vulnerability-Fixes: A Machine Translation Approach. *arXiv preprint arXiv:2012.11701* (2020).
[28] Seyed Mohammad Ghaffarian and Hamid Reza Shahriari. 2021. Neural software vulnerability analysis using rich intermediate graph representations of programs. *Inf. Sci.* 553 (2021), 189–207.
[29] Antonios Gkortzis, Dimitris Mitropoulos, and Diomidis Spinellis. 2018. VulinOSS: a dataset of security vulnerabilities in open-source systems. In *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*. ACM, 18–21.
[30] Xi Gong, Zhenchang Xing, Xiaohong Li, Zhiyong Feng, and Zhuobing Han. 2019. Joint Prediction of Multiple Vulnerability Characteristics Through Multi-Task Learning. In *24th International Conference on Engineering of Complex Computer Systems, ICECCS 2019, Guangzhou, China, November 10-13, 2019*. IEEE, 31–40.
[31] David L. Heine and Monica S. Lam. 2006. Static detection of leaks in polymorphic containers. In *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*. ACM, 252–261.
[32] John L. Henning. 2000. SPEC CPU2000: Measuring CPU Performance in the New Millennium. *Computer* 33, 7 (2000), 28–35.
[33] Nasif Imtiaz and Laurie A. Williams. 2021. Memory Error Detection in Security Testing. *arXiv preprint arXiv:2104.04385* (2021).
[34] Xiujuan Ji, Jufeng Yang, Jing Xu, Lei Feng, and Xiaohong Li. 2012. Interprocedural path-sensitive resource leaks detection for C programs. In *Proceedings of the Fourth Asia-Pacific Symposium on Internetware, Internetware 2012, QingDao, China, October 30-31, 2012*. ACM, 19:1–19:9.
[35] Changhee Jung, Sangho Lee, Easwaran Raman, and Santosh Pande. 2014. Automated memory leak detection for production use. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*. ACM, 825–836.
[36] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.
[37] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net.
[38] Daniel Kroening and Michael Tautschnig. 2014. CBMC - C Bounded Model Checker - (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8413)*. Springer, 389–391.
[39] Quoc V. Le and Tomás Mikolov. 2014. Distributed Representations of Sentences and Documents. In *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014 (JMLR Workshop and Conference Proceedings, Vol. 32)*. JMLR.org, 1188–1196.
[40] Wen Li, Haipeng Cai, Yulei Sui, and David Manz. 2020. PCA: memory leak detection using partial call-path analysis. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*. ACM, 1621–1625.
[41] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Precision-guided context sensitivity for pointer analysis. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 141:1–141:29.
[42] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2020. A Principled Approach to Selective Context Sensitivity for Pointer Analysis. *ACM Trans. Program. Lang. Syst.* 42, 2 (2020), 10:1–10:40.
[43] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. 2016. Gated Graph Sequence Neural Networks. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*.
[44] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. Vulnerability detection with fine-grained interpretations. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*. ACM, 292–303.
[45] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, Zhaoxuan Chen, Sujuan Wang, and Jialai Wang. 2018. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. *arXiv preprint arXiv:1807.06756* (2018).
[46] 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 *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society.
[47] 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 2016, Austin, TX, USA, May 14-22, 2016*. ACM, 911–922.
[48] 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 *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*. ACM, 664–676.
[49] Kangjie Lu, Aditya Pakki, and Qiushi Wu. 2019. Detecting Missing-Check Bugs via Semantic- and Context-Aware Criticalness and Constraints Inferences. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*. USENIX Association, 1769–1786.
[50] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*. ACM, 89–100.
[51] 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.

[52] Maksim Orlovich and Radu Rugina. 2006. Memory Leak Analysis by Contradiction. In *Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4134)*. Springer, 405–424.

[53] Barbara G. Ryder. 1979. Constructing the Call Graph of a Program. *IEEE Trans. Software Eng.* 5, 3 (1979), 216–226.

[54] Michael Sejr Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. 2018. Modeling Relational Data with Graph Convolutional Networks. In *The Semantic Web - 15th International Conference, ESWC 2018, Heraklion, Crete, Greece, June 3-7, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10843)*. Springer, 593–607.

[55] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*. USENIX Association, 309–318.

[56] 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 2018, Philadelphia, PA, USA, June 18-22, 2018*. ACM, 693–706.

[57] Naresh K. Sinha and Michael P. Griscik. 1971. A Stochastic Approximation Method. *IEEE Trans. Syst. Man Cybern.* 1, 4 (1971), 338–344.

[58] Justin Smith, Brittany Johnson, Emerson R. Murphy-Hill, Bill Chu, and Heather Richter Lipford. 2019. How Developers Diagnose Potential Security Vulnerabilities with a Static Analysis Tool. *IEEE Trans. Software Eng.* 45, 9 (2019), 877–897.

[59] 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.

[60] Yulei Sui, Ding Ye, and Jingling Xue. 2012. Static memory leak detection using full-sparse value-flow analysis. In *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*. ACM, 254–264.

[61] Yulei Sui, Ding Ye, and Jingling Xue. 2014. Detecting Memory Leaks Statically with Full-Sparse Value-Flow Analysis. *IEEE Trans. Software Eng.* 40, 2 (2014), 107–122.

[62] Xiaobing Sun, Xin Peng, Kai Zhang, Yang Liu, and Yuanfang Cai. 2019. How security bugs are fixed and what can be improved: an empirical study with

Mozilla. *Sci. China Inf. Sci.* 62, 1 (2019), 19102:1–19102:3.

[63] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*. IEEE Computer Society, 48–62.

[64] 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.

[65] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. 2020. Detecting Code Clones with Graph Neural Network and Flow-Augmented Abstract Syntax Tree. In *27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2020, London, ON, Canada, February 18-21, 2020*. IEEE, 261–271.

[66] 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).

[67] Mark Weiser. 1984. Program Slicing. *IEEE Trans. Software Eng.* 10, 4 (1984), 352–357.

[68] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 21-23, 2014*. IEEE Computer Society, 590–604.

[69] Tianxiang Zhao, Xiang Zhang, and Suhang Wang. 2021. GraphSMOTE: Imbalanced Node Classification on Graphs with Graph Neural Networks. In *WSDM '21, The Fourteenth ACM International Conference on Web Search and Data Mining, Virtual Event, Israel, March 8-12, 2021*. ACM, 833–841.

[70] 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.

[71] 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 *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*. 10197–10207.

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