Singapore Management University

Institutional Knowledge at Singapore Management University

Research Collection School Of Computing and Information Systems

School of Computing and Information Systems

12-2023

Learning program semantics for vulnerability detection via vulnerability-specific inter-procedural slicing

Bozhi WU

Shangqing LIU

Xiao YANG

Zhiming LI

Jun SUN
Singapore Management University, junsun@smu.edu.sg

See next page for additional authors

Follow this and additional works at: https://ink.library.smu.edu.sg/sis_research

Part of the Artificial Intelligence and Robotics Commons, Information Security Commons, and the Theory and Algorithms Commons

Citation

WU, Bozhi; LIU, Shangqing; YANG, Xiao; LI, Zhiming; SUN, Jun; and LIN, Shang-Wei. Learning program semantics for vulnerability detection via vulnerability-specific inter-procedural slicing. (2023). *ESEC/FSE '23: Proceedings of ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, San Francisco, December 3-9*. 1371-1383.

Available at: https://ink.library.smu.edu.sg/sis_research/8578

This Conference Proceeding Article is brought to you for free and open access by the School of Computing and Information Systems at Institutional Knowledge at Singapore Management University. It has been accepted for inclusion in Research Collection School Of Computing and Information Systems by an authorized administrator of Institutional Knowledge at Singapore Management University. For more information, please email cherylds@smu.edu.sg.

Author

Bozhi WU, Shangqing LIU, Xiao YANG, Zhiming LI, Jun SUN, and Shang-Wei LIN

This conference proceeding article is available at Institutional Knowledge at Singapore Management University: https://ink.library.smu.edu.sg/sis_research/8578

Learning Program Semantics for Vulnerability Detection via Vulnerability-Specific Inter-procedural Slicing

Bozhi Wu*
Singapore Management University
Singapore
bozhiwu@smu.edu.sg

Shangqing Liu†
Nanyang Technological University
Singapore
shangqingliu666@gmail.com

Yang Xiao
Chinese Academy of Sciences
China
xiaoyang@iie.ac.cn

Zhiming Li
Nanyang Technological University
Singapore
ZHIMING001@e.ntu.edu.sg

Jun Sun
Singapore Management University
Singapore
JunSun@smu.edu.sg

Shang-Wei Lin
Nanyang Technological University
Singapore
shang-wei.lin@ntu.edu.sg

ABSTRACT

Learning-based approaches that learn code representations for software vulnerability detection have been proven to produce inspiring results. However, they still fail to capture complete and precise vulnerability semantics for code representations. To address the limitations, in this work, we propose a learning-based approach namely *SnapVuln*, which first utilizes multiple vulnerability-specific inter-procedural slicing algorithms to capture vulnerability semantics of various types and then employs a Gated Graph Neural Network (GGNN) with an attention mechanism to learn vulnerability semantics. We compare *SnapVuln* with state-of-the-art learning-based approaches on two public datasets, and confirm that *SnapVuln* outperforms them. We further perform an ablation study and demonstrate that the completeness and precision of vulnerability semantics captured by *SnapVuln* contribute to the performance improvement.

CCS CONCEPTS

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

KEYWORDS

Vulnerability detection, program semantics, code representations.

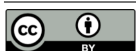
ACM Reference Format:
Bozhi Wu, Shangqing Liu, Yang Xiao, Zhiming Li, Jun Sun, and Shang-Wei Lin. 2023. Learning Program Semantics for Vulnerability Detection via Vulnerability-Specific Inter-procedural Slicing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616351

1 INTRODUCTION

Software security is crucial in practice and thus has attracted widespread attention from academia and industry. Although various techniques including symbolic execution [7, 14, 53], data flow analysis [5, 16, 23] and fuzz testing [29, 49, 57] have been proposed to improve software security, it is still a far from being solved. Specifically, symbolic execution aims at traversing all execution paths to find the vulnerabilities but suffers from path explosion problems. Static analysis such as data flow analysis may result in excessive false positives while fuzz testing will result in high false negatives. Inspired by the great success of deep learning techniques, many learning-based approaches [15, 17, 18, 37–40, 51, 68, 70] have been proposed to build an automated vulnerability detection system and achieve encouraging results. The main idea of learning-based approaches is to learn effective code representations from programs that can reveal vulnerability patterns for vulnerability classification. For example, Russell et al. [51] utilize Convolutional Neural Network (CNN) and Gated Recurrent Unit (GRU) to learn vector representations from lexical tokens of source code for vulnerability detection. To obtain comprehensive code representations, Devign [68] first extracts four kinds of graphs from the source code, including Abstract Syntax Tree (AST), Control Flow Graph (CFG), Data Flow Graph (DFG) and Natural Code Sequence (NCS), and learn code representations from these graphs using graph neural network (GNN) for vulnerability detection. But the extracted program semantics are not all about vulnerabilities. In order to capture the precise vulnerability patterns, VulDeePecker [40] applies slicing techniques on Data Dependency Graph (DDG) to extract program semantics of vulnerable parts, and leverage BLSTM to generate the final code representations for vulnerability classification. However, existing learning-based approaches still suffer from the following two limitations, which prevent them from obtaining precise program semantics of vulnerable parts for code representation generation. In this paper, we refer to the program semantics of vulnerable parts as **vulnerability semantics**.

Limitation1: The extracted vulnerability semantics are incomplete. Some works [15, 51, 68] utilize neural networks to detect vulnerabilities for a single function. One well-received work Devign [68] constructed a code property graph to extract program structures and utilized graph neural networks (GNNs) to learn program semantics for vulnerability detection. Although it produced inspiring results, it only focuses on single-function detection, while the semantics of callee functions are missed because the implementations of these callee functions are not considered in the proposed approach. In practice, vulnerabilities may span across

*Also with Nanyang Technological University.
†Corresponding author.



This work is licensed under a Creative Commons Attribution 4.0 International License.

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

© 2023 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-0327-0/23/12
https://doi.org/10.1145/3611643.3616351

multiple functions (i.e., between a function and its callee functions). Hence, these techniques targeting single-function detection cannot capture complete vulnerability semantics.

Limitation2: The extracted vulnerability semantics are imprecise. Other works [17, 38–40] leverage program slicing algorithms to extract vulnerability semantics starting from various vulnerable program points (e.g., dangerous API calls or variables) for code representation generation. However, these slicing algorithms can not discriminate the sink or source, and every statement that depends on the vulnerable program points will be included. This problem has also been stated in previous research work [28]. Therefore, statements unrelated to vulnerabilities will be introduced into the slice as noise, making the extracted vulnerability semantics imprecise. Besides, these slicing algorithms perform slicing on DDG or Program Dependence Graph (i.e., PDG), without considering CFG that reveals the program execution order. This would lead to failure to catch the semantics of vulnerabilities caused by incorrect execution order, such as "Use After Free", which is triggered when the operation "use" occurs after the operation "free". In summary, the program slicing algorithms used in previous learning-based approaches are insufficient to capture precise vulnerability semantics.

To address the aforementioned limitations, in this work, we propose *SnapVuln*, a learning-based approach that applies multiple vulnerability-specific inter-procedural algorithms that identify the source and sink to capture precise program semantics of various vulnerability types for vulnerability detection. Specifically, we extract the Inter-procedural Graph (IG) from the source code, which can be regarded as a combination of PDG, CFG and Call Graphs (CG). Then, we design multiple vulnerability-specific slicing algorithms for different vulnerability types to capture precise vulnerability semantics, which identify source and sink and further operate on the inter-procedural graph. In this work, we implement vulnerability-specific slicing algorithms for six common vulnerability types in C/C++. To learn better code representation for different vulnerability types, we employ a submodel for each vulnerability type, which utilizes a Gated Graph Neural Network (GGNN) with an attention mechanism, and a GNN model for vulnerabilities not covered by the slicing algorithms. After that, we ensemble the submodels and GNN model to predict vulnerabilities. For instance, given an example, *SnapVuln* extracts six types of subgraphs from IG as potential vulnerability semantics according to the program slicing algorithms. Each type of subgraph is then fed into the corresponding submodel, which capture the structural semantics in the subgraphs and dynamically learn different weights for different subgraphs to generate better code representations. The IG is fed into the GNN model to capture comprehensive program semantics. Finally, we ensemble the output of the six submodels and the GNN model to obtain a comprehensive prediction.

To demonstrate the effectiveness of *SnapVuln*, we conduct extensive experiments on two public datasets and compare *SnapVuln* with seven state-of-the-art baselines. The experimental results show that *SnapVuln* outperforms these baselines. We further perform an ablation study to demonstrate that the completeness and precision of our learned vulnerability semantics contribute to the improvement of vulnerability detection. In summary, our main contributions are as follows:

```
1  int SMB2_write(...){
2          ...
3          req->sync_hdr.ProcessId = cpu_to_le32(io_parms->pid);
4          ...
5  -        cifs_small_buf_release(req);
6          ...
7          if (rc) {
8                  trace_smb3_write_err(xid, req->PersistentFileId,
9                  ...
10         }
11 +        cifs_small_buf_release(req);
12         ...
13 }
14 void cifs_small_buf_release(void *buf_to_free){
15         ...
16         mempool_free(buf_to_free, cifs_sm_req_poolp);
17         ...
18 }
19 void mempool_free(void *element, mempool_t *pool){
20         ...
21         pool->free(element, pool->pool_data);
22 }
```

(a) "Use After Free" from commit *6a3eb33*

```
1  int dtls1_buffer_message(SSL *s, int is_ccs){
2          hm_fragment *frag;
3          ...
4          frag = dtls1_hm_fragment_new(s->init_num, 0);
5  +        if (!frag)
6  +                return 0;
7          memcpy(frag->fragment, s->init_buf->data, s->init_num);
8          ...
9          frag->msg_header.msg_len = s->d1->w_msg_hdr.msg_len;
10         frag->msg_header.seq = s->d1->w_msg_hdr.seq;
11         frag->msg_header.type = s->d1->w_msg_hdr.type;
12         ...
13         item = pitem_new(seq64be, frag);
14         if ( item == NULL){
15                 dtls1_hm_fragment_free(frag);
16                 return 0;
17         }
18         ...
19 }
```

(b) "Null Pointer Dereference" from commit *7a9d59c*

Figure 1: Two Real-world Examples from GitHub Open-source Projects. The lines starting with "-" indicates that these lines of statements have security risks and they are fixed by the statements marked with "+".

- We propose dedicated slicing algorithms for six common vulnerability types in C/C++ to achieve precise vulnerability semantics. To the best of our knowledge, we are the first to design vulnerability-specific slicing algorithms to capture precise vulnerability semantics based on different vulnerability characteristics.
- We incorporate the attention mechanism into the gated graph neural network (GGNN) to ensure that the model can learn to assign different weights to subgraphs produced by slicing algorithms, so as to help the model learn better representations for vulnerability detection.
- We analyse the limitations of existing deep learning-based works for vulnerability detection and show that the completeness and precision of vulnerability semantics is vital for automated vulnerability detection by extensive experiments.
- We conduct extensive experiments to compare *SnapVuln* with seven state-of-the-art baselines, including five learning-based vulnerability detection baselines and two pre-trained approaches on two public datasets. Experimental results show that *SnapVuln* outperforms these baselines significantly. We have made our code and data public at our website [6] for reproduction.

2 MOTIVATION

The key to learning-based vulnerability detection is to learn better code representations for the detection. To achieve this, program semantics about vulnerable parts (here refer to **vulnerability semantics**) should be well captured for representation. Therefore, in order to investigate how well existing learning-based approaches capture vulnerability semantics, we conduct an in-depth analysis and discover two important aspects on which learning-based approaches should improve.

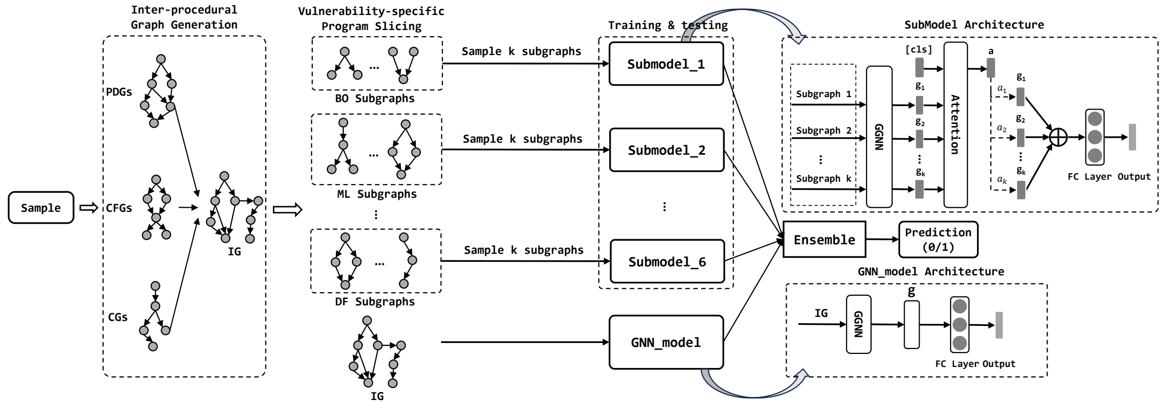


Figure 2: The overview of *SnapVuln*.

Table 1: Line number of statements captured by different slicing approaches for the example in Figure 1(b).

Line number	Ground Truth	VulDeePecker	SySeVR	DeepWukong	*SnapVuln*
2	✓	✓	✓	✓	✗
4	✓	✓	✓	✓	✓
7	✓	✓	✓	✓	✓
9	✗	✓	✓	✓	✗
10	✗	✓	✓	✓	✗
11	✗	✓	✓	✓	✗
13	✗	✓	✓	✓	✗
14	✗	✗	✓	✓	✗
15	✗	✓	✓	✓	✗
16	✗	✗	✓	✓	✗

Finding 1: Inter-procedural analysis should be introduced to capture the complete vulnerability semantics. In the real world, it is common for a vulnerability to span multiple functions. For example, as shown in Figure1(a), the pointer "req" is used again on line 8 after being freed on line 5, which triggers "Use After Free" vulnerability. This vulnerability spans three functions. If only the function "SMB2_write" is considered as input, the function "mempool_free" that contains the important semantics of releasing memory will be lost, and consequently, the model will be unable to learn code representation well for vulnerability detection. However, existing learning-based approaches such as Devign [68], REVEAL [15] and Draper [51] detect vulnerabilities for a single function. Therefore, an inter-procedural analysis should be introduced to capture complete vulnerability semantics across multiple functions for code representation generation.

Finding 2: Slicing algorithms should be designed to capture precise vulnerability semantics. We investigate how existing learning-based approaches obtain vulnerability semantics for code representation generation, and discover that most of them can not capture precise vulnerability semantics. Specifically, VulDeePecker [40] performed program slicing on DDG based on library/API function calls, while SySeVR [39] and DeepWukong [17] performed program slicing on PDGs. On one hand, these slicing algorithms do not capture the control flow information (i.e., CFG) in the code, which makes them impossible to reveal some vulnerabilities caused by wrong execution order, such as "Use After Free". On other hand, these slicing algorithms may include statements that are not related to vulnerabilities, since some statements that are not related to vulnerabilities may also depend on vulnerability-related statements. We take the "Null Pointer Dereference" sample in Figure 1(b) as an example, and follow the slicing algorithms in these approaches to obtain the corresponding results shown in Table 1. We assume that these approaches take line 4 as a slicing criterion, where the function "dtls1_hm_fragment_new" is called. The results show that they obtain 160% more statements than real vulnerability semantics, which are imprecise. We can infer that slicing algorithms should be improved to capture precise vulnerability semantics.

3 APPROACH

Motivated by the aforementioned findings, we propose a learning-based approach namely *SnapVuln*. It applies vulnerability-specific inter-procedural slicing algorithms to capture complete and precise vulnerability semantics of various vulnerability types in C/C++ and leverages a GGNN with an attention mechanism to learn good representations for vulnerability detection. We present the details of *SnapVuln* below.

3.1 Overview

Figure 2 shows the overview of *SnapVuln*, which mainly consists of three components: inter-procedural graph generation, vulnerability-specific program slicing, and the well-designed models. In the first component, we extract CFGs, PDGs and call graphs from each sample, and utilize them to construct an inter-procedural graph to represent the complete program semantics for each sample. The second component utilizes the vulnerability-specific inter-procedural slicing algorithms to extract the precise vulnerability semantics from the complete program semantics. As a result, multiple sets of subgraphs are extracted from the inter-procedural graph, each containing the potential vulnerability semantics corresponding to its specific type. The third component employs multiple submodels and a GNN model to individually capture the distinct types of vulnerability semantics for vulnerability detection. Specifically, the novel submodel that customizes a GGNN with an attention mechanism is proposed to capture the vulnerability semantics of specific type from the corresponding set of subgraphs individually. Moreover, to address vulnerabilities that are not covered by the slicing algorithms, we utilize a GNN model to capture the comprehensive program semantics of the sample from the inter-procedural graph. This GNN model can be considered as a submodel without an attention mechanism, allowing it to identify and predict those unhandled vulnerabilities by the slicing algorithms. Finally, the predictions from the multiple submodels and the GNN model are ensembled to

generate the final predictions. This ensemble approach combines the outputs of each model, resulting in more accurate and reliable predictions for vulnerability detection.

Process: In the training phase, *SnapVuln* constructs the inter-procedural graphs for all training data and extracts six sets of subgraphs for each sample. The six sets of subgraphs contain potential vulnerability semantics of specific types. Since a sample may generate an arbitrary number of subgraphs, we randomly sample k subgraphs from each set to represent the vulnerability semantics corresponding to its specific type, and define them as $\mathcal{G} = \{g_1, ..., g_k\}$. After that, each set of subgraphs $\mathcal{G}$, corresponding to a specific vulnerability type, is individually fed into a submodel for training. These submodels are designed to learn and capture the specific vulnerability semantics associated with each vulnerability type. Furthermore, the inter-procedural graphs are used to train a GNN model, which aims to capture vulnerability semantics that are not handled by the existing slicing algorithms. All these models are trained independently. If a new slicing algorithm is introduced for other vulnerability type, a new submodel can be added, and the subgraphs extracted by the new slicing algorithm can be used to train the submodel. This modular approach provides scalability by allowing the incorporation of new slicing algorithms and the expansion of the range of vulnerability types that can be detected.

In the testing phase, six sets of subgraphs and an inter-procedural graph are extracted from each sample. For each set of subgraphs, k subgraphs are sampled to represent the vulnerability semantics of its specific type. After that, the six sets of subgraphs, along with the inter-procedural graph, are then individually fed into the corresponding well-trained submodels and GNN model. Finally, *SnapVuln* ensembles the outputs of the submodels and GNN model to make a final prediction for each sample.

3.2 Inter-procedural Graph Generation

In order to capture the complete vulnerability semantics, the inter-procedural analysis should be utilized in the data preprocessing stage. Therefore, we utilize the popular static analysis tool Joern [1] to extract code property graphs and further build an inter-procedural graph for each sample.

Definition. In this paper, an inter-procedural graph $\mathcal{IG}$ is a directed graph consisting of a set of nodes and edges where each node represents one statement in the sample and each edge represents the relationship between two statements, including control flow, data/control dependency, and function call. The inter-procedural graph $\mathcal{IG}$ is a combination of CFGs, PDGs and call graphs (i.e., CGs), which exposes a variety of program semantics for vulnerability detection. Specifically, PDGs reveal the data and control dependency of the program, which can be utilized for detecting some specific types of vulnerabilities. For example, with detailed data dependencies on the target pointer and some control dependencies, we can intuitively infer whether the program lacks the operation of releasing memory or not. CFGs further supplement some other types of vulnerabilities such as "Use After Free", since CFGs reveal the explicit execution order of statements, we can detect whether the memory is used after free on CFGs. CGs can be used to enrich inter-procedural graphs to allow for inter-procedural analysis. Therefore, we extract the inter-procedural graph (IG) to ensure that complete vulnerability semantics are captured.

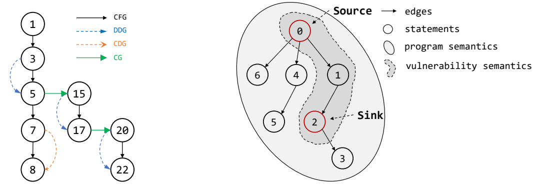


Figure 3: IG of Fig. 1(a). Figure 4: vulnerability semantics.

Generation process. To build inter-procedural graphs, we first employ the popular static tool Joern [1] to extract CFGs, PDGs and CGs from the source code, where PDG consists of data dependencies (DDG) and control dependencies (CDG). Specifically, through the call graphs generated by Joern, we identify the call edge between caller function and callee function, so as to connect CFGs and PDGs of different functions and get the inter-procedural graph for the sample. In this way, The inter-procedural graph contains the complete program semantics and is able to support inter-procedural analysis. We illustrate an inter-procedural graph for the example of Figure 1(a) in Figure 3. Specifically, CFG, DDG and CDG in the functions "SMB_write", "cifs_small_buf_release" and "mempool_free" are extracted first. Then according to the call graph, the function "cifs_small_buf_release" is called in statement 5 of the function "SMB_write". Therefore, we establish a call edge between statement 5 of the function "SMB_write" and the function "cifs_small_buf_release". Similarly, a call edge is established between the function "cifs_small_buf_release" and the statement of line 15 in the function "mempool_free". Finally, we obtain the inter-procedural graph in Figure 3.

3.3 Slicing Algorithms

By constructing the inter-procedural graph for each sample, we capture the complete program semantics. However, program semantics are not equal to vulnerability semantics. As shown in Figure 4, vulnerability semantics are only a part of program semantics. Therefore, we need to further extract precise vulnerability semantics for code representation generation. Inspired by previous work on slicing algorithms named Chopping [28], which identifies the statements that cause the definitions of the source to affect the uses of the sink, we propose vulnerability-specific inter-procedural slicing algorithms for various vulnerability types, which are well-designed by identifying sources and sinks based on the characteristics of each vulnerability type.

Key Idea. The key idea of vulnerability-specific inter-procedural slicing algorithms is to locate the vulnerability-relevant part of the whole inter-procedural graph and slice it out to represent the vulnerability semantics. In detail, *SnapVuln* locates the source (i.e., where it comes from) and sink (i.e., where it ends) [64] of potential vulnerability in the code, and takes the paths from source to sink to construct a subgraph as a representation of vulnerability semantics. Since different vulnerability types have different causes, leading to different sources and sinks, we propose specific slicing algorithms according to the characteristics of different vulnerability types to capture precise vulnerability semantics. In this paper, we implement slicing algorithms for six common vulnerability types in C/C++. We illustrate the corresponding algorithms for each vulnerability type in details.

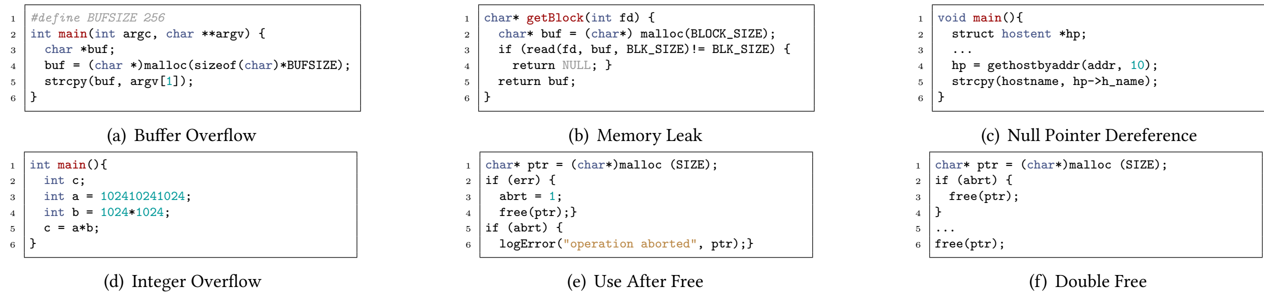


Figure 5: Simple Examples of Six Vulnerability Types.

Type 1: Buffer Overflow (BO). A buffer overflow occurs when the program copies an input buffer to an output buffer without verifying that the size of the input buffer is less than the size of the output buffer. An example is shown in Figure 5(a).

Source: In general, there are two types of buffer overflow, i.e., stack-based and heap-based, where overflow occurs on arrays (stack) and pointers (heap) respectively. Therefore, we take the allocation of arrays and pointers as potential sources. For example, line 4 in Figure 5(a) is the source.

Sink: The sink of buffer overflow is a statement that invokes function calls or assignments to manipulate pointers or arrays. The function call includes "strcpy", "memcpy" and so on, while the assignment is various, where any statement that assigns value to memory that pointer points to or array with index are all included. For example, line 5 in Figure 5(a) is the sink.

Algorithm: *SnapVuln* takes sinks in the target program as the slicing criterion and performs backward slicing on inter-procedural PDG to obtain the statements that affect the sinks until all the sources in the inter-procedural PDG are reached. Note that we do not perform forward slicing by taking sources as a slicing criterion, because it will include those statements that are affected by the source but not related to the vulnerability. We present the details in Algorithm 1.

Algorithm 1: Slicing for "buffer overflow"

Input : Inter-procedural PDG $ipdg$
Output: Vulnerability features vc

1. Retrieve all expression statements that write to arrays or pointers as sinks ks
2. Initialize $vc = \emptyset$
3. **for** $k \in ks$ **do**
4. $\quad s = \text{backwardSlicing}(k, ipdg)$
5. $\quad g = \text{constructGraph}(s, ipdg)$
6. $\quad vc = vc \bigcup g$
7. **return** vc

Type 2: Memory Leak (ML). A memory leak occurs when the program does not sufficiently track and release memory after it has been used. It consumes the remaining memory. An example is shown in Figure 5(b).

Source: The source of memory leak vulnerability is an assignment statement that allocates memory space to a pointer. For example, in the source code of Linux kernel, the program may call standard function (i.e., "malloc", "calloc") directly, or call wrapper functions (i.e., "cifs_buf_get", "AcquireQuantumInfo"). These wrapper functions can be found by analyzing the call graph of the program. For example, "cifs_buf_get" calls "mempool_alloc", and "mempool_alloc" calls "pool->alloc", therefore, "cifs_buf_get" can be taken as source.

Sink: According to the characteristics of the memory leak, it will be triggered if the memory has not been released until the end of the program. Therefore, we take the end of the program as a sink if there are no statements that free the memory.

Algorithm: *SnapVuln* takes the sources as a slicing criterion and performs forward slicing on inter-procedural PDG to obtain the statements that are affected by the sources until the end of the inter-procedural PDG or free functions are found. The slicing process is shown in Algorithm 2.

Algorithm 2: Slicing for "memory leak"

Input : Inter-procedural graph $ipdg$
Output: Vulnerability features vc

1. Retrieve all statements that assign to pointers and invoke function calls as sources es
2. Initialize $vc = \emptyset$
3. **for** $e \in$ es **do**
4. $\quad s = \text{forwardSlicing}(e, ipdg)$
5. $\quad g = \text{constructGraph}(s, ipdg)$
6. $\quad vc = vc \bigcup g$
7. **return** vc

Type 3: Null Pointer Dereference (NP). A null pointer dereference occurs when the program accesses a pointer that expects to be valid but is NULL instead. Figure 5(c) shows an example of a null pointer dereference.

Source: Similar to memory leak, the source of null pointer dereference is also an assignment statement that invokes API function calls such as "malloc" and "calloc" to allocate memory space for a pointer.

Sink: According to the characteristics of null pointer dereference, we conclude that it will be triggered when the null pointer is used for the first time. Therefore, we take the first statement using the pointer as the potential sink. For example, Line 7 in Figure 1(b) is a sink, instead of line 9 to line 11.

Algorithm 3: Slicing for "null pointer"

Input : Inter-procedural graph ig
Output: Vulnerability features vc

1. Let $ipdg$ be the inter-procedural PDG in ig
2. Let $icfg$ be the inter-procedural CFG in ig
3. Retrieve statements that assign the return value of function calls to pointers as sources es
4. Initialize $vc = \emptyset$
5. **for** $e \in es$ **do**
6. $\quad sd = \text{forwardSlicing}(e, ipdg)$
7. $\quad sc = \text{forwardSlicing}(e, icfg)$
8. $\quad fs = \text{computeFirstStatement}(e, sd, sc)$
9. $\quad$ Choose path from e to fs as slice s
10. $\quad g = \text{constructGraph}(s, ipdg)$
11. $\quad vc = vc \bigcup g$
12. **return** vc

Algorithm: For null pointer dereference, *SnapVuln* takes statements that assign the return value of function calls to pointers as

sources (i.e., slicing criterion). To locate the corresponding sinks (i.e., the first statement using the pointers in sources), *SnapVuln* first performs forward slicing starting from slicing criterion on the inter-procedural graph and obtains all statements that are data-dependent on the pointers according to PDG, as well as the execution order according to CFG. With data dependencies between statements and corresponding execution order, we can deduce the first statement using the pointers in sources. After collecting the sources and sinks, *SnapVuln* selects the path from sources to sinks as vulnerability semantics. The process is presented in Algorithm 3.

Type 4: Integer Overflow (IO). An integer overflow occurs when an integer value is incremented after calculation to a value that is too large to store in the associated representation, as depicted in the example of Figure 5(d).

Source: The source of integer overflow is the statement that assigns value to a variable, whose type may be int, long, short, and so on. The type indicates the range of variables.

Sink: Based on the characteristics of integer overflow, those statements that perform arithmetic operations (e.g, +, *, ++) on the variable may trigger integer overflow vulnerability. Therefore, *SnapVuln* takes such statements as sinks.

Algorithm 4: Slicing for "integer overflow"

```
  Input  : Inter-procedural graph ipdg
  Output : Vulnerability features vc
1 Retrieve all statements that contain arithmetic operators as sinks ks
2 Initialize vc = ∅
3 for k ∈ ks do
4 |   s = backwardSlicing(k, ipdg)
5 |   g = constructGraph(s, ipdg)
6 |   vc = vc ⋃ g
7 return vc
```

Algorithm: *SnapVuln* performs backward slicing on inter-procedural PDG by taking sinks as a slicing criterion until the source is reached. Similar to the buffer overflow, *SnapVuln* does not perform forward slicing by taking the sources as a slicing criterion, since it will also include statements that depend on the sources but are not related to the vulnerability. The detailed process is listed in Algorithm 4.

Type 5: Use After Free (UAF). Heap memory is explicitly allocated to pointers through API function calls (such as "malloc"). Once the heap memory is used again after release, a "use after free" vulnerability occurs. Figure 5(e) displays an example of use after free vulnerability.

Algorithm 5: Slicing for "use after free"

```
   Input  : Inter-procedural graph ig
   Output : Vulnerability features vc
 1 Let ipdg be the inter-procedural PDG in ig
 2 Let icfg be the inter-procedural CFG in ig
 3 Retrieve all statements that assign to pointers and invoke function calls as sources es
 4 Initialize vc = ∅
 5 for e ∈ es do
 6 |   sd = forwardSlicing(e, ipdg)
 7 |   sc = forwardSlicing(e, icfg)
 8 |   sl = computeLastStatement(e, sd, sc)
 9 |   Forward slice on ig from e to ls, and obtain slice s
10 |   g = constructGraph(s, ig)
11 |   vc = vc ⋃ g
12 return vc
```

Source: Similar to memory leak vulnerability, the source of use after free vulnerability is also an assignment statement that allocates heap memory space to a pointer, including standard functions and wrapper functions.

Sink: Based on the characteristics, a use after free vulnerability will be triggered when a program uses memory again after it has been freed. Therefore, the sink should be the first statement that utilizes the pointer after the free operation. However, determining which statements perform the free operation can be challenging, since they may be wrapped within user-defined functions. To address this, we propose capturing the sink by considering the last statement that uses the pointer as the end of the slicing, which can ensure that the sink and the free operation are included in the slice.

Algorithm: For use after free, the key is to locate the last statement that uses the pointers in sources, which ensures the slice contains the sink. To achieve this, *SnapVuln* first performs forward slicing starting from the source on the inter-procedural graph and obtains all statements that are data-dependent on the pointers according to PDG, as well as the execution order according to CFG. With data dependencies between statements and corresponding execution order, we can deduce the last statement that uses the pointers. After collecting the source and sink, *SnapVuln* performs forward slicing from source to sink on the inter-procedural graph and gets the slice as vulnerability semantics. The details of the slicing algorithm are shown in Algorithm 5.

Type 6: Double Free (DF). If the same heap pointer is released twice or more in a program, a double free vulnerability occurs, as shown in the example of Figure 5(f).

Source: Similar to use after free vulnerability, the source of double free vulnerability is an assignment statement that allocates heap memory space to a pointer.

Sink: The sinks of double free are two or more free operations on pointers in sources. Similar to use after free, identifying statements used for free operations can be challenging. Therefore, we also take the last statement that uses the pointers as the end of slicing.

Algorithm: Unlike use after free, double free is not related to execution order. The vulnerability semantics of a double free issue are associated with how many times a pointer is freed. Therefore, we perform forward slicing on inter-procedural PDG from the sources to the last statement that uses the pointers of the sources. In this way, we can track all statements that manipulate the pointers to capture the vulnerability semantics. The process is similar to use after free in Algotirhm 5. The only distinction is that in steps 9 and 10, the algorithm replaces *ig* with *ipdg* for double free analysis. Consequently, we do not present the detailed algorithm.

3.4 Model Design

SnapVuln first employs six submodels and a GNN model to learn vulnerability semantics from six sets of subgraphs and inter-procedural graph respectively, and then ensemble the outputs of these models to make a final prediction. In particular, the novel submodel customizes a GGNN with an attention mechanism, which makes it effectively capture the structural information within each subgraph and learn distinct weights for the subgraphs within each set to generate representations. The six submodels are designed to independently capture the vulnerability semantics associated with a specific vulnerability type. For the GNN model, it can be seen as a submodel without attention mechanism, aiming to learn comprehensive program semantics from inter-procedural graph for vulnerability detection. This enables *SnapVuln* to detect vulnerabilities that are not covered by the slicing algorithms.

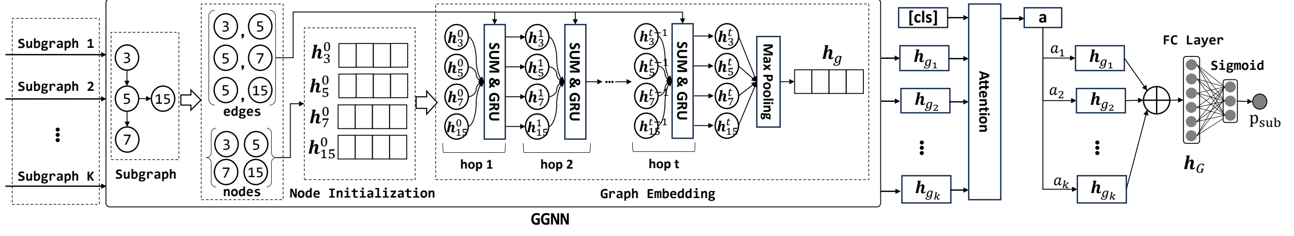


Figure 6: An example of Figure 3 to illustrate the pipeline of the submodel.

3.4.1 Submodel. Each submodel learns vulnerability semantics from a set of subgraphs $\mathcal{G} = \{g_1, ..., g_k\}$ per sample. The submodel first transforms nodes and edges in each subgraph g_i into vector representations, and then utilizes a GGNN to learn graph representation $\boldsymbol{h}_{g_i}$ for each subgraph. After that, the submodel leverages the attention mechanism over the graph representations of a set of subgraphs $\mathcal{G}$ so that it can learn to assign different weights for each subgraph and obtain the vector $\boldsymbol{h}_{\mathcal{G}}$ by the weighted summation of each graph representation $\boldsymbol{h}_{g_i}$. Finally, we take $\boldsymbol{h}_{\mathcal{G}}$ with a fully connected layer to generate the output of the submodel. We describe each component in the submodel in details below, and use an example in Figure 3 to illustrate the pipeline of the submodel, as shown in Figure 6.

Node Initialization. For each subgraph $g_i \in \mathcal{G}$, it can be represented as $g_i = (\mathcal{V}, \mathcal{E})$ where the node $v \in \mathcal{V}$ in g_i consists of a code statement and the edge $e \in \mathcal{E}$ represents different relations between nodes including data/control dependencies, control flow and function calls. Since the node v consists of a code statement (i.e., a sequence of multiple tokens), to obtain the node initial vector representation, we sum up each token vector initialized by a learnable embedding matrix $\boldsymbol{E} \in \mathbb{R}^{m \times d}$ in the code statement, where m is the length of the vocabulary set and d is the dimensional length. The node initial vector representation can be expressed as $\boldsymbol{h}_v^0 \in \mathbb{R}^d$.

Graph Embedding. For each subgraph $g_i \in \mathcal{G}$, we utilize GGNN as the encoder to learn the graph representation $\boldsymbol{h}_{g_i}$. Specifically, for node v in g_i, we utilize a fixed number of hops (i.e., T) to propagate the information along the edges. At each computation hop t, where $1 \leq t \leq T$, we utilize the summation function to aggregate the neighboring node features computed from the previous hop and this process can be expressed as follows:

$$\boldsymbol{h}_{N(v)}^t = \mathrm{SUM}(\{\boldsymbol{h}_u^{t-1} | \forall u \in N_{(v)}\}) \tag{1}$$

where $N_{(v)}$ is a set of neighborhood nodes that are connected to v. Then a gated recurrent unit [19] is used to update the feature of v, which can be formulated as follows:

$$\boldsymbol{h}_v^t = \mathrm{GRU}(\boldsymbol{h}_v^{t-1}, \boldsymbol{h}_{N_{(v)}}^t) \tag{2}$$

After a fixed number of hops (i.e., T), we obtain the final node representation $\boldsymbol{h}_v^T$ for the node v. We further apply the max-pooling over all nodes (i.e., $\{\boldsymbol{h}_v^T | \forall v \in \mathcal{V}\}$) of subgraph g_i to obtain the graph representation $\boldsymbol{h}_{g_i}$ as follows:

$$\boldsymbol{h}_{g_i} = \mathrm{maxpool}(\mathrm{FC}(\{\boldsymbol{h}_v^T | \forall v \in \mathcal{V}\})) \tag{3}$$

where $\mathrm{FC}(\cdot)$ is a fully connected layer. For each subgraph $g_i \in \mathcal{G}$, we obtain its corresponding graph representation denoted as $\boldsymbol{G} = \{\boldsymbol{h}_{g_1}, ..., \boldsymbol{h}_{g_k}\}$ where $\boldsymbol{G} \in \mathbb{R}^{k \times d}$.

Attention. We further add an attention module to ensure that the model learns to assign different weights of subgraphs $\mathcal{G}$ for the prediction. Specifically, we add an extra token "[CLS]" with its initial vector $\boldsymbol{h}_{\mathrm{cls}}$ and concatenate it with $\boldsymbol{G}$ to obtain a new representation $\boldsymbol{G'} = \{\boldsymbol{h}_{\mathrm{cls}}, \boldsymbol{h}_{g_1}, ..., \boldsymbol{h}_{g_k}\}$ where $\boldsymbol{G'} \in \mathbb{R}^{(k+1) \times d}$. Inspired by self attention [54], we use scaled dot-product attention to obtain different weights for subgraphs $\mathcal{G}$ and it can be expressed as follows:

$$\boldsymbol{A} = \mathrm{softmax}(\frac{\boldsymbol{Q}\boldsymbol{K}^T}{\sqrt{d}}) \qquad \boldsymbol{h}_{\mathcal{G}} = \sum_{j=1}^{k} a_j \boldsymbol{h}_j \tag{4}$$

where $\boldsymbol{G'} = \boldsymbol{Q} = \boldsymbol{K}$, $\boldsymbol{a} \in \mathbb{R}^{k+1}$ is the vector of the index token "[CLS]" in the matrix $\boldsymbol{A} \in \mathbb{R}^{(k+1) \times (k+1)}$, a_j is the value of the j-th index for the subgraph g_j in the vector $\boldsymbol{a}$, $\boldsymbol{h}_j \in \mathbb{R}^d$ is the vector of the index j in $\boldsymbol{G'} \in \mathbb{R}^{(k+1) \times d}$ and d is the dimensional length. The vector $\boldsymbol{a}$ is considered as the weighted vector learnt by the model for different subgraphs.

Prediction. We take the weighted representation $\boldsymbol{h}_{\mathcal{G}}$ followed by a fully connected layer for the prediction:

$$\mathrm{p_{sub}} = \mathrm{Sigmoid}(\mathrm{FC}(\boldsymbol{h}_{\mathcal{G}})) \tag{5}$$

where $\mathrm{p_{sub}}$ denotes the probability computed by the activation function Sigmoid.

3.4.2 GNN Model. The GNN model can be considered as a submodel without an attention mechanism, sharing the same GGNN module as the submodel in Figure 6. By utilizing the GGNN module, the GNN model converts the inter-procedural graph into a graph representation denoted as $\boldsymbol{h}_{\mathcal{I}G}$. Subsequently, we map the graph representation $\boldsymbol{h}_{\mathcal{I}G}$ to the predicted probabilities in the prediction module, described as follows:

$$\mathrm{p}_{\mathcal{I}\mathrm{G}} = \mathrm{Sigmoid}(\mathrm{FC}(\boldsymbol{h}_{\mathcal{I}\mathrm{G}})) \tag{6}$$

3.4.3 Ensemble Model. Through the above feature learning process, we obtain predicated probabilities from the six submodels and GNN model. The final step is to ensemble the learned results from these models for the final prediction. There are multiple ways to achieve this goal. In this work, we refer to a similar paper [69] which ensembles two models to identify security-related patches, and use a weighted combination of these models output as ensemble model, which can be described as follows:

$$\mathrm{prob} = \alpha \cdot (\mathrm{p_{sub_1}} + \mathrm{p_{sub_2}} + ... + \mathrm{p_{sub_i}} + \mathrm{p}_{\mathcal{I}\mathrm{G}}) \tag{7}$$

where $i = 6$ and $\alpha = 1/(i+1)$. The classification result is 0 for prob less than 0.5 and 1 for the others, as below:

$$\text{prediction} = \begin{cases} 1, & \text{if prob} \geq 0.5 \\ 0, & \text{if prob} < 0.5 \end{cases} \tag{8}$$

4 EVALUATION

To demonstrate the effectiveness of *SnapVuln*, we conduct extensive experiments to investigate the following four research questions.

- RQ1: Can *SnapVuln* outperform state-of-the-art baselines?
- RQ2: Can the completeness of the extracted vulnerability semantics affect the performance in vulnerability detection?
- RQ3: Can the precision of the extracted vulnerability semantics affect the performance of vulnerability detection?
- RQ4: What is the memory usage and execution time of *SnapVuln* in detecting vulnerabilities compared to the baselines?

4.1 Evaluation Setup

4.1.1 Dataset. To study the above research questions, we should choose a dataset containing multi-function samples for evaluation, since the impact of vulnerability semantic completeness can be revealed by comparing the vulnerability detection performance on samples of the single-function version and the corresponding multi-function version. We search the papers from 2018 to 2022 for vulnerability datasets in C/C++, and find several public datasets including Devign [68], Draper [51], Fan [20], D2A [66] and Juliet [2]. In the end, we choose a real-world vulnerability dataset D2A and a widely used synthetic dataset Juliet as evaluation datasets since they are datasets containing multi-function samples.

Preprocessing: We conduct the following steps to preprocess the datasets for the evaluation:

1) We collect the samples of six common vulnerability types in C/C++ from D2A and Juliet, and filter out those samples that cannot be parsed into code property graphs by Joern. Finally, we get the final dataset for evaluation, as shown in Table 2. Since samples in Juliet contains tokens with obvious vulnerability meanings such as "sink", "source", "good" and "bad", We normalize them to "norm", and map all user-defined function names to symbolic names (e.g., func0).

2) To support the study, we need to obtain the evaluation datasets of the single-function version and the corresponding multi-function version. Since the samples in the original datasets are multi-function, we only need to obtain the corresponding single-function version. Specifically, given a multi-function vulnerability, we select the function that triggers the vulnerability as the corresponding single-function version. For the example in Figure 1(a), "Use After Free" is triggered at line 8 of function "SMB2_write". Therefore, we choose the function "SMB2_write" as the single-function version.

3) For model training and testing, we split the data of each vulnerability type with the ratio of 80%:10%:10%, and then fuse the divided data of each vulnerability type as the train/validation/test set for evaluation.

4.1.2 Baseline. To evaluate *SnapVuln*, we select five state-of-the-art learning-based vulnerability detection approaches and two popular pre-trained approaches as baselines. Specifically, the existing five vulnerability detection approaches can be divided into two

Table 2: The statistics of the used datasets.

Dataset	Juliet							D2A						
	# Samples	LoC			# Tokens			# Samples	LoC			# Tokens		
		min	max	avg	min	max	avg		min	max	avg	min	max	avg
BO	10074	8	63	21	53	349	128	2437	8	2156	328	62	9235	2140
IO	3744	8	34	17	34	172	71	10624	4	2303	270	28	8206	1755
NP	316	6	34	13	22	131	55	413	8	1027	125	73	4296	849
ML	1548	9	30	17	33	196	91	-	-	-	-	-	-	-
DF	880	8	25	15	25	154	66	-	-	-	-	-	-	-
UF	458	9	40	19	35	213	89	-	-	-	-	-	-	-
Non-Vuln	17002	6	63	13	23	339	111	13321	5	2807	258	39	9609	2107
Total	34022	6	63	16	22	349	110	26795	4	2807	267	28	9609	1951

categories: (1) function-level detection (i.e., detecting vulnerabilities at function granularity), such as Devign [68] and REVEAL [15]. (2) multi-function-level detection (i.e., detecting vulnerabilities at the granularity of multiple functions), such as VulDeePecker [40], SySeVR [39], and DeepWukong [17]. Since the pre-trained models achieve promising results in many code-related tasks [48, 67], we also compare *SnapVuln* with two popular pre-trained models (i.e., CodeBERT [21] and Graphcodebert [25]). We use their implementation if available (i.e., REVEAL, SySeVR and DeepWukong, CodeBERT, and Graphcodebert) and re-implement the approaches otherwise according to their papers (i.e., Devign, VulDeePecker).

4.1.3 Metrics. Following previous works [17, 39, 68], we select the widely used metrics of accuracy and F1-score to evaluate the vulnerability detection performance of different approaches. Accuracy is the ratio of the number of correct predictions to the total number of input samples, which indicates the correctness of identifying both vulnerable and non-vulnerable samples. F1-score is the harmonic mean between precision and recall and indicates the balance between them, where precision implies the correctness of predicted vulnerable samples and recall implies the effectiveness of vulnerability prediction.

4.1.4 Model Setting. We set the embedding size of tokens to 128 and the batch size to 16 for training. The hop is set to 4. The number of subgraphs k is set to 16 and 4 on D2A and Juliet respectively. Note that we will introduce why and how we choose the number of subgraphs in Section 4.2.4. We adopt the Adam optimizer with a learning rate of 0.001 to train the model on NVIDIA RTX A6000 with 10-epoch patience for early stopping. The model settings are validated to achieve the optimal performance via grid search.

4.1.5 Evaluation Procedure. For RQ1, we compare *SnapVuln* with baselines on D2A and Juliet, respectively. For RQ2, to verify the completeness of vulnerability semantics on the detection performance, we conduct comparative experiments on the datasets of the multi-function version and the corresponding single-function version. For a fair comparison, we disable the slicing component and just utilize interprocedural-graph to conduct experiments (i.e., *SnapVuln*_single/*SnapVuln*_multi). For RQ3, to evaluate the precision of extracted vulnerability semantics on the detection performance, we first perform an ablation study on *SnapVuln* by disabling the slicing component to verify the effectiveness of slicing algorithms (i.e., *SnapVuln*_multi). Then we compare our proposed slicing algorithms with three typical slicing algorithms respectively proposed by VulDeePecker, SySeVR and DeepWukong for further verification (i.e., *SnapVuln*_vuldeepecker, *SnapVuln*_sysevr and *SnapVuln*_deepwukong). Specifically, DeepWukong extracted subgraphs from PDGs by taking API function calls or arithmetic operations as a slicing criterion, while VulDeePecker and SySeVR

Table 3: The evaluation results on two open-source datasets.

Approach	D2A															
	IO				BO				NP				Overall			
	Acc	F1	P	R	Acc	F1	P	R	Acc	F1	P	R	Acc	F1	P	R
Devign	67.2	67.6	66.8	68.4	66.1	65.9	66.2	65.6	61.8	56.7	65.5	50.0	67.0	67.1	66.8	67.4
REVEAL	67.5	68.3	66.8	69.7	63.6	64.5	62.9	66.2	62.5	57.7	66.1	51.2	67.1	69.1	65.1	73.6
VulDeePecker	67.4	66.7	68.1	65.3	67.5	66.7	68.3	65.1	61.0	57.8	63.0	53.4	67.3	66.5	68.2	64.8
SySeVR	68.0	67.1	69.1	65.3	69.2	68.3	70.4	66.4	64.0	62.1	65.6	58.8	68.1	67.2	69.1	65.4
DeepWukong	69.7	70.9	68.2	73.7	66.6	67.2	66.0	68.4	65.0	64.5	65.4	63.6	68.7	69.6	67.8	71.4
CodeBERT	74.8	77.1	70.7	**84.7**	75.2	77.1	71.5	**83.6**	73.2	71.8	74.7	69.1	74.3	76.6	70.2	**84.3**
Graphcodebert	76.1	77.2	73.8	81.0	75.0	75.7	73.6	77.8	70.7	69.2	72.0	66.7	75.6	76.7	73.4	80.2
SnapVuln	**81.0**	**80.7**	**79.2**	82.3	**78.1**	**77.7**	**79.1**	76.4	**74.4**	**73.4**	**76.9**	70.7	**80.6**	**80.7**	**80.1**	81.6
SnapVuln_single	68.7	69.2	68.2	70.2	66.1	66.5	65.6	67.4	64.5	62.0	66.7	57.9	68.6	69.0	68.2	69.8
SnapVuln_multi	71.4	70.9	72.3	69.6	72.0	67.6	78.9	58.5	66.1	68.8	63.8	**74.6**	71.1	70.2	72.5	68.0
SnapVuln_vuldeepecker	74.2	74.4	73.9	74.9	74.1	74.9	72.7	77.2	65.0	64.1	65.8	62.5	75.0	75.1	74.8	75.4
SnapVuln_sysevr	77.4	76.7	79.0	74.6	73.7	72.0	77.0	67.6	73.4	71.2	76.6	65.8	77.1	77.0	77.4	76.6
SnapVuln_deepwukong	76.2	76.2	76.1	76.3	73.7	73.5	74.2	72.8	67.5	68.3	66.7	70.0	76.7	76.5	77.3	75.7

Approach	Juliet																											
	IO				BO				NP				ML				DF				UF				Overall			
	Acc	F1	P	R	Acc	F1	P	R	Acc	F1	P	R	Acc	F1	P	R	Acc	F1	P	R	Acc	F1	P	R	Acc	F1	P	R
Devign	61.7	53.5	68.0	44.1	65.8	65.7	65.8	65.6	58.9	53.1	61.9	46.4	64.2	64.9	63.6	66.2	62.5	62.3	62.6	62.0	75.0	73.2	78.9	68.2	64.6	58.7	70.6	50.2
REVEAL	62.6	61.3	63.4	59.3	67.8	67.9	67.7	68.2	62.5	64.1	61.5	66.8	64.8	66.9	63.2	71.0	65.7	68.1	63.6	73.3	68.7	69.9	67.4	72.6	66.8	68.0	65.5	70.6
VulDeePecker	76.0	78.5	71.1	87.7	66.6	66.8	66.4	67.2	77.8	80.5	71.7	91.7	69.9	71.4	68.0	75.2	53.9	46.1	55.5	39.5	57.9	57.7	57.9	57.4	69.0	71.1	66.6	76.2
SySeVR	84.8	85.0	84.1	85.9	67.7	61.0	76.8	50.6	83.3	83.3	83.3	83.3	75.9	74.6	79.0	70.7	60.1	49.0	67.8	38.3	66.7	62.9	70.9	56.5	71.2	68.6	75.2	63.1
DeepWukong	81.3	82.6	77.2	88.8	68.8	66.8	71.4	62.8	83.3	84.2	80.0	88.9	73.9	75.1	71.8	78.8	58.4	51.9	61.6	44.8	61.1	58.3	62.8	54.4	72.1	72.5	71.4	73.7
CodeBERT	93.3	93.0	**96.6**	89.7	82.2	79.2	95.0	67.9	92.2	92.3	90.8	93.9	81.8	78.6	85.1	73.0	75.6	69.5	92.4	55.7	63.0	45.2	79.8	31.5	83.7	81.4	90.6	73.9
Graphcodebert	**94.8**	**94.9**	92.6	**97.3**	79.6	77.1	88.1	68.5	82.8	83.1	81.8	84.3	77.0	74.4	83.7	66.9	**86.9**	**86.9**	87.5	86.3	71.7	66.7	81.3	56.5	83.1	81.7	89.0	75.5
SnapVuln	88.2	87.3	94.0	81.6	**92.1**	**91.8**	**95.5**	**88.4**	**96.4**	**96.3**	**99.9**	92.9	**95.3**	**95.4**	**93.4**	95.4	79.8	78.4	86.7	71.7	**88.8**	**88.3**	**92.3**	84.5	**93.4**	**93.6**	**91.0**	**96.4**
SnapVuln_single	65.0	66.1	64.1	68.2	66.4	60.8	73.0	52.1	69.6	70.2	69.0	71.5	64.9	59.1	70.8	50.7	64.1	55.4	73.2	44.6	75.3	74.9	76.1	73.8	65.8	66.0	65.6	66.3
SnapVuln_multi	74.2	78.0	68.1	91.2	70.8	73.5	67.2	81.1	76.8	81.2	70.5	**95.0**	69.9	72.5	66.8	79.1	66.9	67.7	66.0	69.6	78.4	81.2	71.9	**93.2**	72.1	74.9	68.1	83.2
SnapVuln_vuldeepecker	81.3	83.3	75.3	93.2	73.7	74.1	72.9	75.3	80.0	82.0	74.6	90.9	87.7	89.0	80.5	**99.4**	75.0	67.9	95.0	52.8	74.4	76.1	71.4	81.4	76.3	77.2	74.2	80.5
SnapVuln_sysevr	82.8	82.6	83.5	81.6	81.3	81.7	80.1	83.4	83.9	84.8	80.6	89.3	89.9	90.7	83.6	99.3	81.5	81.7	80.8	82.6	84.1	84.8	81.3	88.6	84.2	84.9	81.2	88.8
SnapVuln_deepwukong	79.6	72.6	95.8	58.4	81.5	79.6	88.6	72.3	84.0	84.2	83.3	85.1	93.2	93.2	92.3	94.2	80.3	73.2	**96.4**	59.0	81.5	79.6	88.6	72.3	85.0	84.8	86.3	83.3

sliced a sequence of code statements on DDGs and PDGs respectively. Since *SnapVuln* takes graphs as input, we transform the code statements extracted by VulDeePecker and SySeVR into graphs by connecting the statements based on DDGs and PDGs. In this way, we can compare the performance of slicing algorithms of different approaches with a unified model *SnapVuln*.

4.2 Evaluation Results

4.2.1 Comparison with State-of-the-art Approaches (RQ1). The first two rows of Table 3 present the results of baselines on D2A and Juliet. We observe that *SnapVuln* outperforms baselines on the overall testset of D2A and Juliet in terms of accuracy and F1.

As seen in Table 3, *SnapVuln* outperforms vulnerability detection baselines (first row) by at least 11% (absolute differences) and surpasses pre-trained models (second row) by at least 4% in terms of accuracy and F1 on the overall testset, which shows the effectiveness of *SnapVuln*. Furthermore, for most types of vulnerabilities (i.e., "buffer overflow", "memory leak" and "use after free"), *SnapVuln* also has higher accuracy, except for "integer overflow (IO)" and "double free (DF)" on Juliet testset. For these cases, the pre-trained models achieve higher accuracy. Since the samples from Juliet are synthetic and simple, which leads to powerful pre-trained models easily capture the inner features among the samples. In contrast, *SnapVuln* outperforms the pre-trained models on D2A dataset, indicating that *SnapVuln* is more effective for real-world vulnerabilities. We attribute this to the vulnerability-specific inter-procedural slicing algorithms that are designed to capture precise vulnerability semantics.

Analysis. We conclude that in most cases, *SnapVuln* achieves the best performance for vulnerability detection, especially on the real-world D2A testset. It is attributed to the complete and precise vulnerability semantics captured by the slicing algorithms and the attention mechanism used in the model to dynamically assign different weights for subgraphs. The shortcomings of the baselines are twofold. On one hand, the baselines such as Devign capture the vulnerability semantics on the single function, which are incomplete for vulnerability detection. On other hand, some approaches such as VulDeePecker, SySeVR, and DeepWukong propose the general slicing algorithms without considering the characteristics of different vulnerability types for vulnerability detection, which are imprecise. Furthermore, although the pre-trained approaches can achieve better performance compared with the aforementioned vulnerability detection baselines, they are limited by learning precise vulnerability semantics that *SnapVuln* does for vulnerability detection.

Answer to RQ1: The overall performance of *SnapVuln* outperforms existing approaches on the real-world dataset D2A and the synthetic dataset Juliet.

4.2.2 The Completeness of Vulnerability Semantics (RQ2). The results of *SnapVuln*_single and *SnapVuln*_multi are presented in Table 3, we can observe that the overall detection performance on multi-function datasets is better than single-function datasets in terms of accuracy and F1 on D2A and Juliet.

In particular, the accuracy for multi-function version are 2.5% and 6.3% higher than that of single-function version on the overall testsets of D2A and Juliet. Furthermore, we find that the performance of each vulnerability type is improved when using multi-function datasets D2A and Juliet. In addition, the magnitude of the improvement on D2A is smaller than Juliet. For example, the vulnerability type "integer overflow" on D2A increases by 2.7% in accuracy while Juliet increases by 9.2%.

Analysis. We attribute the improvements to the version of the multifunction dataset consisting of the complete vulnerability semantics to help the model learn more vulnerability-related features compared with the single function for vulnerability detection. However, we also need to point out that when introducing multi-function for detection, we also introduce substantial noise (i.e., vulnerability-irrelevant semantic information) for the model, especially in the real-world dataset D2A. In contrast, the Juliet dataset is a synthetic dataset that contains fewer code statements unrelated to vulnerabilities. Hence, the irrelevant information is less than D2A on the version of multi-function datasets, which provides more noticeable improvements. It also indicates that the complete vulnerability semantics are not sufficient for the model to obtain good performance and some slicing algorithms are needed to extract precise vulnerability semantics for the detection.

Answer to RQ2: The completeness of the extracted vulnerability semantics affects the performance of vulnerability detection. Although some vulnerability-irrelevant semantic information is introduced at the same time, it can still improve detection performance. It also indicates that precise slicing criteria are demanded to reduce the noise and further improve the performance.

4.2.3 The Precision of Vulnerability Semantics (RQ3). The experimental results of different slicing algorithms that baselines used are presented in the last row of Table 3. First, comparing the results of *SnapVuln* with *SnapVuln*_multi, we can find that the improvements are significant, which demonstrates that slicing is an effective approach to remove vulnerability-irrelevant information for

vulnerability detection. Furthermore, by comparing the results of different slicing algorithms with *SnapVuln*, we find that *SnapVuln* achieves better performance in terms of vulnerability detection, which demonstrates that our slicing algorithm is effective in extracting relevant vulnerability semantics.

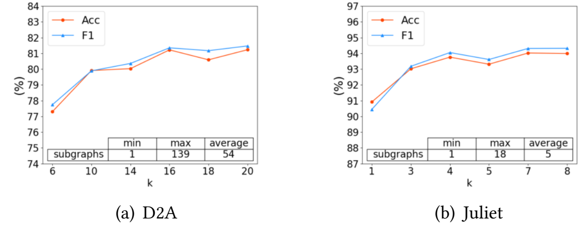
Specifically, when using the slicing algorithms, the accuracy of *SnapVuln* on the overall testset of D2A and Juliet increases by 9.5% and 21.3% respectively. Furthermore, the accuracy of each type of vulnerability on D2A and Juliet has also improved. In addition, compared with other slicing algorithms, *SnapVuln* achieves 5.6%, 3.5%, 3.9% higher accuracy than VulDeePecker, SySeVR and DeepWukong on the overall testset of D2A and 17.1%, 9.2%, and 8.4% on Juliet.

Analysis. The improved performance by *SnapVuln* can be attributed to the precise vulnerability semantics captured by the proposed slicing algorithms for the six most dangerous vulnerability types. Through the vulnerability-specific inter-procedural slicing algorithms, *SnapVuln* is able to extract features containing precise vulnerability semantics as the model input from samples, thus improving the effectiveness and accuracy of *SnapVuln*. On the contrary, the baselines either perform slicing on DDG to obtain all statements that are data-dependent on API function calls (e.g., VulDeePecker) or slice on PDG to get related statements (e.g., SySeVR and DeepWukong) as vulnerability semantics, without considering the characteristics of different vulnerability types. Therefore, they inevitably introduce statements irrelevant to vulnerabilities as model input, resulting in poor performance.

Answer to RQ3: The precision of the extracted vulnerability semantics is vital to the performance of vulnerability detection. Compared with other slicing approaches, we confirm that our slicing algorithms extract more precise vulnerability semantics.

4.2.4 Performance Analysis (RQ4). In this RQ, we first conduct experiments to determine the optimal value of k that achieves the best performance while keeping training feasible. Based on this selected k, we further analyze *SnapVuln* and baselines with respect to the time spent on constructing and analyzing a sample and memory requirements for training and testing.

Choice of k. The value of k (i.e., the number of sampled subgraphs per set in each sample) may affect the memory requirement and vulnerability detection performance, since a larger k requires more memory for model training and testing, while a smaller k value may cause the loss of vulnerability semantics of large samples, resulting in detection failure. We experiment with different k to determine the optimal value. As depicted in Figure 7(a) and Figure 7(b), we observe that values below $k = 16$ and $k = 4$ show worse results in D2A and Juliet, respectively. Additionally, increasing the values of k does not result in consistent improvement. Hence, setting k to 16 for D2A and 4 for Juliet can be considered a reasonable sweet spot between capturing sufficient information for vulnerability detection and keeping training feasible in the GPU's memory. Furthermore, from the table in Figure 7(a) and Figure 7(b), we can find that the average number of subgraphs per set is 54 and 5 in D2A and Juliet, respectively. Setting k above 16 in D2A and above 4 in Juliet may be beneficial for some large samples. However, in most cases, 16 subgraphs in D2A and 4 subgraphs in Juliet contain sufficient



(a) D2A		(b) Juliet

Figure 7: The impact of k on accuracy and F1.

Table 4: Time and memory spent on the two datasets.

Dataset	D2A				Juliet			
	Time(s)		Memory(GB)		Time(s)		Memory(GB)	
	constructing	analyzing	CPU	GPU	constructing	analyzing	CPU	GPU
Devign	2.60	0.04	23.4	41.2	0.93	0.01	8.4	3.0
REVEAL	2.60	0.03	23.7	39.6	0.93	0.01	9.1	3.2
VulDeePecker	3.35	0.03	8.5	5.5	1.05	0.01	6.1	1.4
SySeVR	3.82	0.03	7.6	5.6	1.10	0.01	6.2	1.7
DeepWukong	3.90	0.05	24.2	10.6	1.08	0.01	6.4	1.6
CodeBERT	-	0.03	5.8	14.2	-	0.01	5.6	**14.2**
Graphcodebert	-	0.03	5.7	14.2	-	0.01	5.6	**14.2**
SnapVuln	**4.68**	**0.19**	**29.0**	**43.3**	**1.22**	**0.10**	**9.7**	3.5

information for vulnerability detection, which may be attributed to the overlap between these subgraphs. For example, some small subgraphs may be part of those larger subgraphs, since they may be sliced from branches of those larger subgraphs. This may result in these subgraphs containing duplicate vulnerability semantics.

Time and Memory. All approaches, with the exception of the pre-trained models (i.e., CodeBERT and Graphcodebert), are required to construct graphs for vulnerability detection. Considering the various time required for graph construction and analysis across different samples, we calculate the average time across all samples to represent the time spent by each approach. From the Table 4, we can observe that *SnapVuln* spends more time than the baselines, especially the time spent on constructing the graphs. That can be attributed to the incorporation of inter-procedural graph construction and the execution of multiple slicing algorithms. However, in practice, we reduce the time to less than 0.5 second per sample by multiprocessing, making the time spent by *SnapVuln* acceptable. Furthermore, regarding memory usage, we record the maximum memory consumed by each approach. Since *SnapVuln* employs multiple independently trained submodels, we take the largest memory consumed by these submodels as its final result. As can be seen from Table 4, *SnapVuln* consumes more memory in CPU and GPU compared to the baselines during model training (except for the pre-trained models on Juliet), especially in the real-world dataset D2A. However, compared to Devign using the same GGNN network, the GPU memory usage of *SnapVuln* has only witnessed a minor increase, with an additional 2.1 GB. This can be attributed to the precise slicing algorithms, which greatly reduces the size of each subgraph. As a result, despite the utilization of multiple subgraphs per sample for vulnerability detection, the memory requirement remains relatively stable and does not experience significant increase. Hence, *SnapVuln* proved to be acceptable in terms of time and memory requirements.

Answer to RQ4: *SnapVuln* is optimal with k of 16 on D2A and 4 on Juliet. Although *SnapVuln* takes more time and memory than baselines, it proved to be acceptable.

5 DISCUSSION

When scaling *SnapVuln* to real-world industry databases, it may encounter challenges of the diversity of vulnerability types and sample size. (1) **Vulnerability types**. Vulnerabilities in real-world codebases are diverse, and the slicing algorithms in *SnapVuln* may not cover every single case. In this work, we employ multiple submodels and a GNN model to individually capture the distinct types of vulnerability semantics for vulnerability detection. On one hand, the GNN models can capture comprehensive program semantics to detect those vulnerabilities that are not handled by slicing algorithms. On the other hand, this modular approach enables *SnapVuln* to easily accommodate other vulnerability types by simply adding new slicing algorithms and submodels without the need to retrain existing submodels. Furthermore, it helps prevent the aggregation of all subgraphs into a single model, thereby mitigating the substantial increase in GPU memory requirements. (2) **Sample size**. when dealing with large samples, such as entire programs that can span thousands of lines of code, *SnapVuln* may face challenges in processing and analyzing them effectively. To scale *SnapVuln* to those samples with immense lines of code, we may increase the value of k or employ methods to divide them into smaller pieces for testing. For instance, the samples are constrained to functions at the file level or within the three call levels.

6 THREATS TO VALIDITY

Internal validity: (1) Some samples may be mislabled in the datasets (i.e., D2A [66] and Juliet [2]). On one hand, we trust most of their labeling results, since Juliet is maintained by the domain experts in NIST, and D2A is collected through differential analysis and enhanced with an industrial-strength static analysis tool. On the other hand, we manually validate both datasets and remove those mislabeled samples we found. (2) The tool "Joern" may fail to generate the correct inter-procedural graphs for some samples. To avoid this problem, we delete those samples that are incorrectly parsed by "Joern". (3) *SnapVuln* leverages a GGNN to capture vulnerability semantics for vulnerability detection. However, different models may be suitable for different vulnerability types. For example, GNNs, which can encode code structure information, may be better suited for vulnerability types involving complex logic, while pre-trained models may be better suited for small-sample or simple vulnerability types, since they learn a lot of background knowledge from a large code corpus with huge model parameters. More research should be conducted to study how to choose appropriate models for specific vulnerability types. (4) In RQ3, we re-implement the slicing algorithms from the baselines in *SnapVuln*, by following their provided implementation as outlined. The reproduced algorithms may have some difference from the original ones.

External validity: *SnapVuln* is designed to detect vulnerabilities in C/C++ program. To support other programming languages, it is necessary to design slicing algorithms tailored to the vulnerabilities specific to those languages. Furthermore, vulnerability datasets for those languages are also required to retrain the models.

7 RELATED WORK

Program Slicing Techniques. Program slicing is a useful decomposition technique for extracting program statements relevant to some special computation, starting from a subset of program behaviour and slicing that program into a minimal form that still produces that behavior [63]. It was first proposed by Weiser [61] in 1979, which supports intra-procedural program slicing. After that, many studies related to program slicings are proposed, such as extensions of the algorithm proposed by Horwitz [11, 12, 27, 35], computing efficiency [44, 45], and SDG construction [26, 34]. Program slicing has been applied to many aspects of the software development life cycle, including software maintenance [31–33, 65], software measurement [9, 10, 30] and software debugging [13, 24, 28, 47, 50, 52]. It has also been used to capture program semantics to generate code representations in learning-based vulnerability detection approaches [17, 38–40]. However, these program slicing algorithms include vulnerability-unrelated statements for code representation generation. In this paper, referring to the inter-procedural chopping [50] which identifies the statements that cause the definitions of the source to affect the uses of the sink, we propose vulnerability-specific inter-procedural slicing algorithms to capture precise vulnerability semantics for code representation generation.

GNNs for Software Engineering. Since graph neural networks (GNN) can capture extensive structure information in code, it has become increasingly popular for various tasks [4, 8, 36, 41–43, 46, 58, 59, 62] in software engineering. For example, for type inference task, Allamanis et al. [3] propose modeling the type information for optionally-typed languages by embedding an abstract syntax tree and data flow analysis based graph with a gated graph neural network (GGNN); For code summarization task, attempts has been made to incorporate synactical and sematic code information leveraging graph neural networks [22, 36, 41], which empirically demonstrates superior performance than sequence-based models. For vulnerability detection, graph neural networks are widely used to leverage structure information such as control flow and program dependency in the code to generate representations [15, 17, 37, 56, 68]. Therefore, in this paper we leverage graph neural network to capture vulnerability semantics in the code for vulnerability detection.

8 CONCLUSION

We propose *SnapVuln*, a novel learning-based approach, which employs vulnerability-specific inter-procedural slicing algorithms for different vulnerability types in C/C++ to capture complete and precise vulnerability semantics for code representation generation and incorporates the attention mechanism to the gated graph neural network (GGNN) to ensure the model assign different weights for subgraphs to learn better representations. Extensive experiments on two public datasets demonstrate that *SnapVuln* outperforms seven state-of-the-art baselines. We also conduct an ablation study to demonstrate that the completeness and accuracy of vulnerability semantics contribute to the performance improvement.

ACKNOWLEDGMENTS

This research is partially supported by the Ministry of Education, Singapore under its Academic Research Fund Tier 3 (Award ID: MOET32020-0004), Chinese National Natural Science Foundation under Grant #62202462. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the Ministry of Education, Singapore.

REFERENCES

[1] 2022. *joern.* https://joern.io/

[2] 2022. *Juliet.* https://samate.nist.gov/SARD

[3] Miltiadis Allamanis, Earl T Barr, Soline Ducousso, and Zheng Gao. 2020. Typilus: Neural type hints. In *Proceedings of the 41st acm sigplan conference on programming language design and implementation.* 91–105.

[4] Miltiadis Allamanis, Henry Jackson-Flux, and Marc Brockschmidt. 2021. Self-supervised bug detection and repair. *Advances in Neural Information Processing Systems* 34 (2021), 27865–27876.

[5] Frances E. Allen and John Cocke. 1976. A program data flow analysis procedure. *Commun. ACM* 19, 3 (1976), 137.

[6] Authors. 2023. Learning Precise Program Semantics for Vulnerability Detection via Type-specific Inter-procedural Slicing. https://sites.google.com/view/snapvuln.

[7] Domagoj Babić, Lorenzo Martignoni, Stephen McCamant, and Dawn Song. 2011. Statically-directed dynamic automated test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis.* 12–22.

[8] David Bieber, Charles Sutton, Hugo Larochelle, and Daniel Tarlow. 2020. Learning to execute programs with instruction pointer attention graph neural networks. *Advances in Neural Information Processing Systems* 33 (2020), 8626–8637.

[9] James M Bieman and Byung-Kyoo Kang. 1998. Measuring design-level cohesion. *IEEE Transactions on software engineering* 24, 2 (1998), 111–124.

[10] James M Bieman and Linda M Ott. 1994. Measuring functional cohesion. *IEEE transactions on Software Engineering* 20, 8 (1994), 644–657.

[11] David Binkley. 1993. Precise executable interprocedural slices. *ACM Letters on Programming Languages and Systems (LOPLAS)* 2, 1-4 (1993), 31–45.

[12] David Binkley. 1993. Slicing in the presence of parameter aliasing. In *Software Engineering Research Forum.* 261–268.

[13] Robert S Boyer, Bernard Elspas, and Karl N Levitt. 1975. SELECT—a formal system for testing and debugging programs by symbolic execution. *ACM SigPlan Notices* 10, 6 (1975), 234–245.

[14] Sang Kil Cha, Maverick Woo, and David Brumley. 2015. Program-adaptive mutational fuzzing. In *2015 IEEE Symposium on Security and Privacy.* IEEE, 725–741.

[15] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering* (2021).

[16] Walter Chang, Brandon Streiff, and Calvin Lin. 2008. Efficient and extensible security enforcement using dynamic data flow analysis. In *Proceedings of the 15th ACM conference on Computer and communications security.* 39–50.

[17] Xiao Cheng, Haoyu Wang, Jiayi Hua, Guoai Xu, and Yulei Sui. 2021. Deepwukong: Statically detecting software vulnerabilities using deep graph neural network. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 3 (2021), 1–33.

[18] Xiao Cheng, Guanqin Zhang, Haoyu Wang, and Yulei Sui. 2022. Path-sensitive code embedding via contrastive learning for software vulnerability detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis.* 519–531.

[19] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078* (2014).

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

[21] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[22] Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. 2018. Structured neural summarization. *arXiv preprint arXiv:1811.01824* (2018).

[23] Lloyd D Fosdick and Leon J Osterweil. 1976. Data flow analysis in software reliability. *ACM Computing Surveys (CSUR)* 8, 3 (1976), 305–330.

[24] Peter Fritzson, Nahid Shahmehri, Mariam Kamkar, and Tibor Gyimothy. 1992. Generalized algorithmic debugging and testing. *ACM Letters on Programming Languages and Systems (LOPLAS)* 1, 4 (1992), 303–322.

[25] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366* (2020).

[26] Dixie Hisley, Matt Bridges, and Lori Pollock. 2002. Static interprocedural slicing of shared memory parallel programs. (2002).

[27] Susan Horwitz, Thomas Reps, and David Binkley. 1990. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 1 (1990), 26–60.

[28] Daniel Jackson and Eugene J Rollins. 1994. *Chopping: A generalization of slicing.* Technical Report. CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE.

[29] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. 2019. Razzer: Finding kernel race bugs through fuzzing. In *2019 IEEE Symposium on Security and Privacy (SP).* IEEE, 754–768.

[30] Byung-Kyoo Kang and James M Bieman. 1996. Design-level cohesion measures: Derivation, comparison, and applications. In *Proceedings of 20th International Computer Software and Applications Conference: COMPSAC'96.* IEEE, 92–97.

[31] Taeho Kim, Yeong-Tae Song, Lawrence Chung, and Dung T Huynh. 1999. Software architecture analysis using dynamic slicing. In *AoM/IAoM 17th International Conference on Computer Science.* 242–247.

[32] Taeho Kim, Yeong-Tae Song, Lawrence Chung, and Dung T Huynh. 2000. Software architecture analysis: a dynamic slicing approach. *International Journal of Computer & Information Science* 1, 2 (2000), 91–103.

[33] Taeho Kim, Yeong-Tae Song, Lawrence Chung, and DT Hyunh. 1999. Dynamic software architecture slicing. In *Proceedings. Twenty-Third Annual International Computer Software and Applications Conference (Cat. No. 99CB37032).* IEEE, 61–66.

[34] Akos Kiss, Judit Jász, Gábor Lehotai, and Tibor Gyimóthy. 2003. Interprocedural static slicing of binary executables. In *Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation.* IEEE, 118–127.

[35] Arun Lakhotia. 1992. Improved interprocedural slicing algorithm. *Report CACS TR-92-5-8, University of Southwestern Louisiana* (1992).

[36] Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved code summarization via a graph neural network. In *Proceedings of the 28th international conference on program comprehension.* 184–195.

[37] Yi Li, Shaohua Wang, Tien N Nguyen, and Son Van Nguyen. 2019. Improving bug detection via context-based code representation learning and attention-based neural networks. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–30.

[38] Zhen Li, Deqing Zou, Shouhuai Xu, Zhaoxuan Chen, Yawei Zhu, and Hai Jin. 2021. Vuldeelocator: a deep learning-based fine-grained vulnerability detector. *IEEE Transactions on Dependable and Secure Computing* (2021).

[39] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2021. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* (2021).

[40] 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. *arXiv preprint arXiv:1801.01681* (2018).

[41] Shangqing Liu, Yu Chen, Xiaofei Xie, Jing Kai Siow, and Yang Liu. 2020. Retrieval-Augmented Generation for Code Summarization via Hybrid GNN. In *International Conference on Learning Representations.*

[42] Shuwen Liu, Bernardo Grau, Ian Horrocks, and Egor Kostylev. 2021. Indigo: Gnn-based inductive knowledge graph completion using pair-wise encoding. *Advances in Neural Information Processing Systems* 34 (2021), 2034–2045.

[43] Shangqing Liu, Xiaofei Xie, Lei Ma, Jingkai Siow, and Yang Liu. 2021. Graph-searchnet: Enhancing gnns via capturing global dependency for semantic code search. *arXiv preprint arXiv:2111.02671* (2021).

[44] Panos E Livadas and Stephen Croll. 1993. System dependence graph construction for recursive programs. In *Proceedings of 1993 IEEE 17th International Computer Software and Applications Conference COMPSAC'93.* IEEE, 414–420.

[45] Panos E Livadas and Stephen Croll. 1995. A new algorithm for the calculation of transitive dependences. *Journal of Software Maintenance: Research and Practice* 7, 3 (1995), 151–176.

[46] Wai Weng Lo, Siamak Layeghy, Mohanad Sarhan, Marcus Gallagher, and Marius Portmann. 2022. GNN-based Android Malware Detection with Jumping Knowledge. *arXiv preprint arXiv:2201.07537* (2022).

[47] R Lyle. 1987. Automatic program bug location by program slicing. In *Proceedings 2nd international conference on computers and applications.* 877–883.

[48] Ehsan Mashhadi and Hadi Hemmati. 2021. Applying codebert for automated program repair of java simple bugs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR).* IEEE, 505–509.

[49] Brian S Pak. 2012. Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution. *School of Computer Science Carnegie Mellon University* (2012).

[50] Thomas Reps and Genevieve Rosay. 1995. Precise interprocedural chopping. In *Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering.* 41–52.

[51] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. 2018. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE international conference on machine learning and applications (ICMLA).* IEEE, 757–762.

[52] Ehud Yehuda Shapiro. 1982. *Algorithmic program debugging.* Yale University.

[53] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution.. In *NDSS*, Vol. 16. 1–16.

[54] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all

you need. *Advances in neural information processing systems* 30 (2017).

[55] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *International Conference on Learning Representations*.

[56] Huanting Wang, Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Yansong Feng, Lizhong Bian, and Zheng Wang. 2020. Combining graph-based learning with automated data collection for code vulnerability detection. *IEEE Transactions on Information Forensics and Security* 16 (2020), 1943–1958.

[57] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 724–735.

[58] 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 *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 261–271.

[59] Yu Wang, Ke Wang, Fengjuan Gao, and Linzhang Wang. 2020. Learning semantic program embeddings with graph interval neural network. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–27.

[60] Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. 2020. Lambdanet: Probabilistic type inference using graph neural networks. *arXiv preprint arXiv:2005.02161* (2020).

[61] Mark David Weiser. 1979. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method.* University of Michigan.

[62] Bozhi Wu, Shangqing Liu, Ruitao Feng, Xiaofei Xie, Jingkai Siow, and Shang-Wei Lin. 2022. Enhancing Security Patch Identification by Capturing Structures in Commits. *IEEE Transactions on Dependable and Secure Computing* (2022).

[63] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. 2005. A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes* 30, 2 (2005), 1–36.

[64] 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*. IEEE, 590–604.

[65] Jianjun Zhao. 1998. Applying slicing technique to software architectures. In *Proceedings. Fourth IEEE International Conference on Engineering of Complex Computer Systems (Cat. No. 98EX193)*. IEEE, 87–98.

[66] Yunhui Zheng, Saurabh Pujar, Burn Lewis, Luca Buratti, Edward Epstein, Bo Yang, Jim Laredo, Alessandro Morari, and Zhong Su. 2021. D2A: a dataset built for AI-based vulnerability detection methods using differential analysis. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 111–120.

[67] Jiayuan Zhou, Michael Pacheco, Zhiyuan Wan, Xin Xia, David Lo, Yuan Wang, and Ahmed E Hassan. 2021. Finding A Needle in a Haystack: Automated Mining of Silent Vulnerability Fixes. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 705–716.

[68] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems* 32 (2019).

[69] Yaqin Zhou, Jing Kai Siow, Chenyu Wang, Shangqing Liu, and Yang Liu. 2021. Spi: Automated identification of security patches via commits. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021), 1–27.

[70] Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li, and Hai Jin. 2019. μVulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection. *IEEE Transactions on Dependable and Secure Computing* 18, 5 (2019), 2224–2236.

Received 2023-02-02; accepted 2023-07-27