

# The Vulnerability Is in the Details: Locating Fine-grained Information of Vulnerable Code Identified by Graph-based Detectors

Baijun Cheng, Kailong Wang, Cuiyun Gao, Xiapu Luo, Yulei Sui, Li Li, Yao Guo, Xiangqun Chen, Haoyu Wang

**Abstract**—Vulnerability detection is a crucial component in the software development lifecycle. Existing vulnerability detectors, especially those based on deep learning (DL) models, have achieved high effectiveness. Despite their capability of detecting vulnerable code snippets from given code fragments, the detectors are typically unable to further locate the fine-grained information pertaining to the vulnerability, such as the precise vulnerability triggering locations. In this paper, we propose VULEXPLAINER, a tool for automatically locating vulnerability-critical code lines from coarse-level vulnerable code snippets reported by DL-based detectors. Our approach takes advantage of the code structure and the semantics of the vulnerabilities. Specifically, we leverage program slicing to get a set of critical program paths containing vulnerability-triggering and vulnerability-dependent statements and rank them to pinpoint the most important one (i.e., sub-graph) as the data flow associated with the vulnerability. We demonstrate that VULEXPLAINER performs consistently well on four state-of-the-art graph-representation (GP)-based vulnerability detectors, i.e., it can flag the vulnerability-triggering code statements with an accuracy of around 90% against eight common C/C++ vulnerabilities, outperforming five widely used GNN-based explanation approaches. The experimental results demonstrate the effectiveness of VULEXPLAINER, which provides insights into a promising research line: integrating program slicing and deep learning for the interpretation of vulnerable code fragments.

## I. INTRODUCTION

The proliferation of modern software programs developed for diverse purposes and usage scenarios is inevitably and persistently coupled with intensified security threats from vulnerabilities, evidenced by the substantial surge in the volume of reported vulnerabilities via the Common Vulnerabilities and Exposures (CVE) [1]. To counteract the potential exploitation, both academia and industrial communities have proposed numerous techniques for identifying and locating those vulnerabilities.

Traditional approaches, such as the rule-based analysis techniques (e.g., SVF [2], CHECKMARX [3], INFER [4], and CLANG STATIC ANALYZER [5]), leverage predefined signatures or rules to identify vulnerabilities. Unfortunately, similar to other static analysis techniques, they typically suffer from high false positive and negative rates. More recently, DL-based detection techniques [6], [7], [8], [9], which generally operate on extracted code feature representations, have shown great effectiveness in flagging vulnerability-containing code fragments (i.e., functions or slices). However, the coarse granularity and the black-box nature of the analysis renders poor interpretability in the detection results. For example, a

function or a code snippet could contain over a dozen of code lines, which remains challenging for the developers to understand the root cause of the vulnerabilities and further take action to fix them.

One promising way to tackle this problem is leveraging explanation approaches to select important features for the DL-based detectors, and then mapping them to the corresponding code lines. Recent rapid advances in the graph-based explainability technology shows great potential towards this solution. In particular, the existing explanation methods commonly facilitate model interpretability from three angles: assigning numeric values to graph edges [10], [11], computing importance scores for nodes [12], and calculating scores for graph walks while traversing through GNNs [13]. Despite their success in tasks such as molecular graph classifications, the existing GNN-based explanation techniques still contain intrinsic insufficiencies that hinder the direct application to derive fine-grained information regarding the vulnerability, such as the triggering code lines.

The first insufficiency lies in the limited capability for capturing the subtle but rich semantics encompassed in benign and vulnerable code bases. Similar to the fact that chemical molecule function is defined by its chemical groups (i.e., the important structures including nodes and edges), the function of a program is defined by statements (i.e., nodes) and their information flows (i.e., edges) in the extracted code graph as well. Therefore, program-specific semantics is crucial for explaining approaches. Nonetheless, existing explaining methods are unable to locate such fine-grained information as they generally neglect the rich semantic information in program graphs. This is plausibly attributed to the complexity of program vulnerability detection in comparison to the existing tasks (i.e., simpler topological structure). For example, the control-flow or program-dependence relationship between two statements represented by an edge is hardly reflected. In addition, the semantic information contained in a node is difficult to be encoded into the latent space.

The second insufficiency arises from inadequate consideration of statements that are critical for vulnerability detection. Most vulnerable programs and their patched versions typically have similar topological structures, as they all contain the statements which trigger vulnerabilities, as shown in Figure 1. The only difference could reside in a few statements where the vulnerabilities are patched, entailing control-flow and program-dependent information related to the vulnerability triggers. Unfortunately, none of the existing methods target

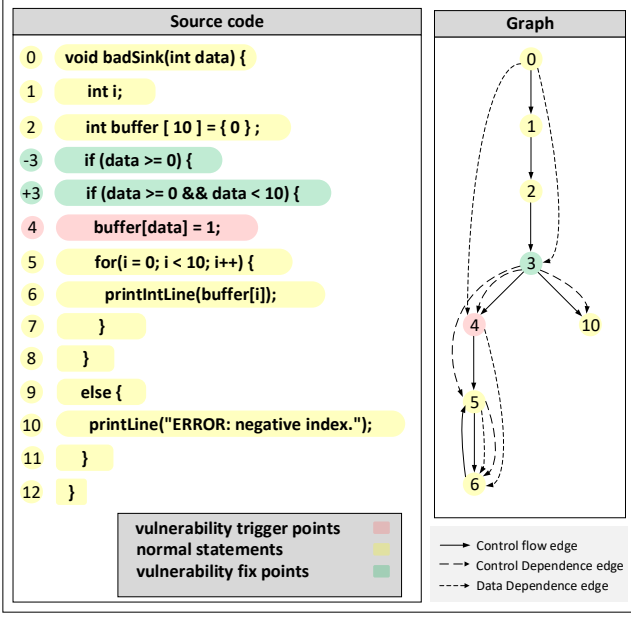


Fig. 1: A vulnerable code from SARD where in its fixed version Line 3 has been modified.

or capture such relationships between vulnerability-triggering and vulnerability-fixing statements.

**Our solution.** In this work, we propose VULEXPLAINER, a novel approach to identify fine-grained information from vulnerable code reported by GNN-based vulnerability detectors. Given a detected vulnerable code fragment, VULEXPLAINER first extracts program slices from it and then constructs the control- and data-dependence information. Compared with prior works (e.g., DEEPWUKONG [6]), VULEXPLAINER only preserves vulnerability-triggering and vulnerability-dependent program path-level information, rather than that of the full program. This significantly improves the analysis efficiency as program paths contain a smaller number of code lines. Leveraging the program slicing method, VULEXPLAINER captures more semantic information encompassed in code lines. As a result, it can provide more accurate explanation results than the approaches only focusing on topological features.

The goal of VULEXPLAINER is to identify the root cause of the vulnerabilities. A recent work [14] suggests that the bug trigger path is the key to locating and fixing a vulnerability. Thus, to evaluate the effectiveness of our approach, we propose a new evaluation metric, *vulnerability-triggering code line coverage* (or referred to as *LC* hereafter, to be detailed in Section V-B). We conduct multi-dimensional evaluations on the effectiveness of VULEXPLAINER. In the first comparison dimension, we apply VULEXPLAINER to interpret the outputs of four state-of-the-art graph code representation-based vulnerability detectors, including DEEPWUKONG [6], REVEAL [7], IVDETECT [8], and DEVIGN [9]. All of these four detectors use program dependence graphs (PDGs, DEVIGN uses only data dependence graph, not control dependence graph) as the code graph representation. We observe that VULEX-

PLAINER can better locate vulnerability-triggering code lines on slice-level detector DEEPWUKONG than the other three function-level detectors. In the second dimension, we compare VULEXPLAINER with other GNN explainers, including PG-EXPLAINER, GNNEXPLAINER, GRAD, DEEPLIFT [15] and GNN-LRP. VULEXPLAINER achieves an average *LC* of 90% which is significantly higher than that of other explanation approaches. More specifically, the explanation results given by the existing approaches are not strongly related to the dependence paths of vulnerable statements, which could be the cause of their lower effectiveness. In addition, we show that the performance of VULEXPLAINER is independent of the front-end detectors, which further confirms that VULEXPLAINER is a general-purpose approach.

In summary, we make the following main contributions:

- **A novel vulnerability fine-grained information locating technique for GNN-based vulnerability detectors.** Given the inadequate explainability of the existing GNN-based vulnerability detectors, we propose the framework VULEXPLAINER as a solution. It can identify important flow paths in a program that contain vulnerability-triggering statements, providing finer-grained semantics contexts for the identified vulnerabilities. We release our source code and dataset used in this work on our anonymous repository [16].
- **Approach effectiveness.** Through the multi-dimensional evaluation of the comprehensive benchmark dataset, we show that VULEXPLAINER outperforms the existing explanation methods in terms of *LC*, which is the key factor that impacts the localization and fix of a vulnerability. On average, VULEXPLAINER achieves *LC* higher than 85% for all vulnerability detectors used in this study, showing good generalization for different GNN-based vulnerability detectors.

## II. BACKGROUND

### A. GNN-based Vulnerability Detectors

Recently, GNNs have been utilized by security analysts and researchers in vulnerability detection tasks [9], [7], [8], [6], [17], [18]. They presume the graph representation of codes could better preserve critical semantic information of vulnerability-related programs, compared with traditional sequence-based representation. Typically, the most frequently used graph representation is code property graph (CPG), which is combined with abstract syntax tree (AST), control flow graph (CFG), control dependence graph (CDG), and data dependence graph (DDG). In addition, another graph representation program dependence graph (PDG) is composed of CDG and DDG, which could be deemed as a substructure of CPG and is widely used in program slicing [19]. In this study, we mainly utilize PDG for slicing. Generally, the detection phase of a GNN-based detector usually consists of three steps, as shown in Figure 2:

(a) **Parsing source code into a graph representation.** A target source code fragment is typically a function or a slice. Here we utilize Joern [20] to dump the graph representations

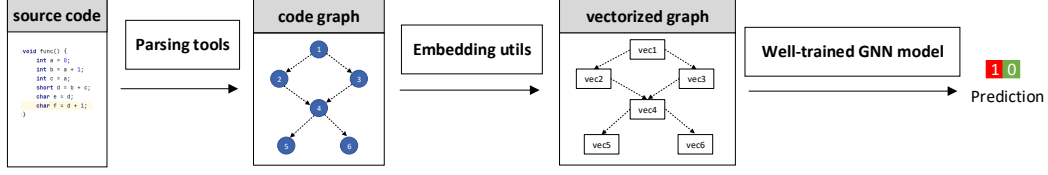


Fig. 2: General detection phase of deep-learning-based vulnerability detectors with graph representations

of the code fragment to support those GNN-based detectors (DEEPWUKONG [6], REVEAL [7], IVDetect).

**(b) Embedding code graph into vectorized representation.**

In a code graph, a node usually stands for a program statement while an edge denotes a relationship (execution order or def-use) between two statements. Here, each node could be vectorized by DOC2VEC [21] or WORD2VEC [22]. Then the vectorized graph data is generated by sequentially vectorizing all the contained nodes.

**(c) Using a well-trained GNN model to classify vectorized code graph.** With vectorized graphs of code fragments and their labels, a GNN model, such as Graph Convolutional Networks (GCN) and Gated Graph Neural Networks (GGNN), could be trained to detect vectorized graph data from target programs.

### B. Control and Data dependence relation

In a PDG, a control-dependence edge  $S_i \rightarrow S_j$  denotes whether statement  $S_j$  will be executed depending on constraint conditions in  $S_i$ . A data-dependence edge  $S'_i \rightarrow S'_j$  means that a value defined at  $S'_i$  is used in  $S'_j$ . And there are no other statements redefining corresponding value among the path from  $S'_i$  to  $S'_j$ .

The control-dependence graph of a program can be determined through the algorithm presented by Cytron R et al [23]. While data dependence relations can be calculated through reaching definition analysis.

## III. AN EXAMPLE OF LOCATING VULNERABILITY

To illustrate the essential idea of our methodology, we use a code fragment from SARD [24] as a toy example, as shown in Figure 3. It contains a buffer overflow vulnerability that is triggered by copying more data (i.e., 100 bytes defined by line 11 of the code fragment) than the maximum capacity of an array (i.e., 50 bytes defined by line 2). A GNN-based vulnerability detector only outputs the detection results as 1, indicating the code fragment is vulnerable (or 0 vice versa). The target of the vulnerability locating task is to construct a control- and data-dependence path, or *flow path* hereafter, that contains the vulnerability triggering code line and the critical assignment of vulnerability-related variables. To this end, we first convert the source code into a graph representation by mapping the statements into nodes and then constructing the flow paths according to the dependence information among the nodes. From the paths, we select ones that satisfy our vulnerability locating target. Specifically, in our example shown in Figure 3, there are multiple flow paths extracted from the original code fragment, such as “8 – 11”, “2 – 6 – 7 – 13”,

etc. Among them, “2 – 6 – 7 – 11” is deemed the most critical one as both lines 2 (critical variable assignment) and line 11 (vulnerability trigger) are included.

**Technical Challenges.** According to this vulnerability locating sample, the technical challenges for a general and automatic location of the detected vulnerability code are at least two-fold:

- **Challenge#1** Given a correctly detected vulnerable code by a GNN-based detector, there is a lack of an efficient vulnerability locating approach that generates flow paths covering both vulnerability triggers and related critical variable assignments.
- **Challenge#2** Given the generated flow paths, there is a lack of an efficient path selection mechanism that identifies the most suitable path as the most reasonable final data flow for a detected vulnerability.

To address the two challenges, we propose VULEXPLAINER that automatically generates feasible flow paths from the PDG derived from a code fragment, and rank them to select the most reasonable path. The technical details of this framework are to be presented in Section IV.

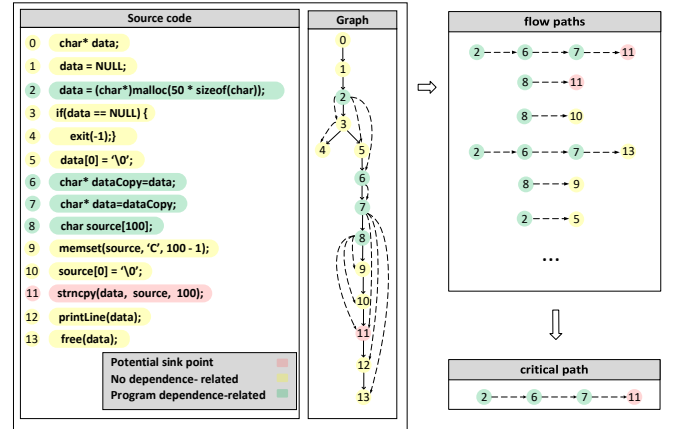


Fig. 3: A example extracted from SARD.

## IV. LOCATING VULNERABILITY STATEMENTS USING GNN-BASED DETECTORS

By manually investigating the vulnerable and non-vulnerable code fragment samples in the existing public datasets respectively, we can always find code fragment pairs (i.e., one labeled as vulnerable and the other non-vulnerable) that are highly similar except for the vulnerability-triggering and the vulnerability-fixing code lines. They are crucial features utilized by the current GNN-based vulnerability

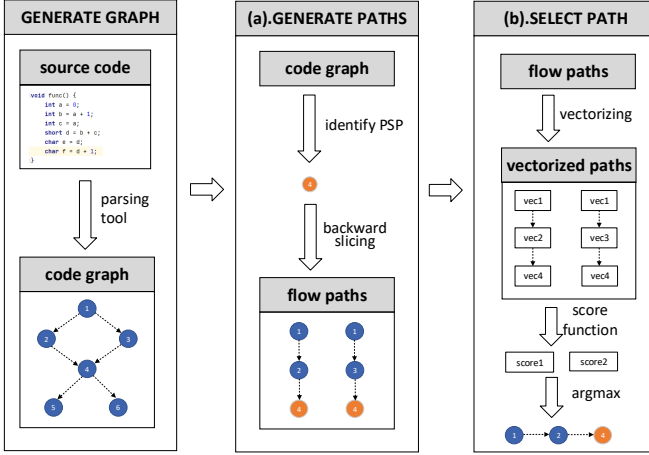


Fig. 4: Overview of VULEXPLAINER.

detectors. Based on the insights drawn from the control- and data-dependence between those vulnerability-triggering and vulnerability-fixing code lines, we propose VULEXPLAINER to facilitate the vulnerability code line locating of the GNN-based detection results.

#### A. VULEXPLAINER Overview

As shown in Figure 4, the overall framework of VULEXPLAINER consists of two modules: flow path generation from the original code graph and critical path selection.

**Flow Path Generation.** Given a vulnerable code fragment represented as a graph with its control- and data-dependence computed (in Figure 4(a)), VULEXPLAINER first identifies statements (i.e., nodes) in the program that might trigger the vulnerability, denoted as potential sink points (PSPs). Next, VULEXPLAINER iteratively traverses backward from a PSP along a flow path in the program graph, until the source of the PSP (e.g., node representing critical variable assignment) is reached. Similarly, VULEXPLAINER generates all the qualified flow paths from the graph, each ending with a PSP.

**Flow Path Selection.** VULEXPLAINER first vectorizes each flow path and computes an importance score correlated to the vulnerability probability (in Figure 4(b)). Next, VULEXPLAINER selects the flow path with the highest importance score as the vulnerability data flow. Note that we do not directly train a classifier for the path selection as each path is regarded as a data flow rather than a code fragment.

#### B. Flow Path Generation

To generate flow paths from the original code graph (i.e., PDG), we utilize program slicing based on DLVD methods, which have been widely adopted by previous works such as DEEPWUKONG, REVEAL, IVDetect, DEVIGN. The slicing principle is based on both control and data dependence of the PDG. More specifically, the detailed flow path generation approach is described by the “GENERATESLICE” function in Algorithm 1. It takes a code graph  $G$  and a path length limit  $k$  (i.e., to effectively remove lengthy paths for subsequent searching) as input. We detail the algorithm as follows.

**Algorithm 1 Details.** In line 2, the path set  $S$  for the current program is initialized as an empty set. In line 3, the algorithm extracts PSPs with the given code graph (Section IV-B1). Then the algorithm generates flow paths for each PSP with the following steps. In line 5, we initialize the current traversed path  $p$  with the corresponding PSP. Then in line 6, the current PSP’s flow-path set is initialized as an empty set. In line 7, flow paths are generated with a DFS algorithm (to be explained next). In line 8, we include all flow paths of the current PSP to the path set  $S$ .

The function DFS describes the process of the backward traversing algorithm when generating flow paths. In lines 14-16, if the length of the current flow path  $p$  reaches the upper limit, then  $p$  will be appended to the path set  $S$  and the function will return. In lines 18-19, the algorithm extracts nodes on which the last node  $n$  of  $p$  is dependent. In lines 20-22, if  $p$  cannot continue to extend, then  $p$  will be appended to  $S$ . Otherwise, in lines 24-27, we repeat this DFS process for each node that  $n$  is dependent on.

---

#### Algorithm 1 Slice Generation Algorithm.

---

**Input:** code graph  $G$ , max length of path  $k$

**Output:** path set  $S$

```

1: function GENERATESLICE( $G, k$ )
2:    $S \leftarrow \emptyset$ 
3:    $\text{sink\_nodes} \leftarrow \text{ExtractSinkNodes}(G)$ 
4:   for  $\text{sink\_node} \in \text{sink\_nodes}$  do
5:      $p \leftarrow \{\text{sink\_node}\}$ 
6:      $S' \leftarrow \emptyset$ 
7:     DFS( $p, G, 1, k, S'$ )
8:     Append all slice in  $S'$  to  $S$ 
9:   end for
10:  return  $S$ 
11: end function

12:
13: function DFS( $p, G, l, k, S$ )
14:  if  $l = k$  then
15:    Append  $p$  to  $S$ 
16:    return
17:  end if
18:   $n \leftarrow \text{last node in } p$ 
19:   $\text{prec\_nodes} \leftarrow \text{ExtractPrecNodes}(n, G)$ 
20:  if  $\text{prec\_nodes}$  is  $\emptyset$  then
21:    Append  $p$  to  $S$ 
22:    return
23:  end if
24:  for  $\text{prec\_node} \in \text{prec\_nodes}$  do
25:    Append  $\text{prec\_node}$  to  $p$ 
26:    DFS( $p, G, l + 1, k, S$ )
27:    pop the last node in  $p$ 
28:  end for
29: end function

```

---

1) *Potential Sink Points (PSPs)*: PSPs are statements that are critically related to vulnerabilities. In Algorithm 1, they are extracted by the function “ExtractSinkNode” (line 3) which considers the following four types of PSPs in our program slicing. We adopt the same definition proposed by Li et al [25].

- **Library/API Function Call (FC).** This kind of PSP covers almost all vulnerability types except for integer overflow. Different types of vulnerabilities are triggered by various types of API calls. For example, OS command injection is usually triggered by APIs such as *system* and *execl*, while buffer overflow is normally triggered by data copy functions like *memcpy*.
- **Array Usage (AU).** This kind of PSP usually appears in memory errors. In this study, AU only covers the buffer overflow vulnerability. For example, “data[i] = 1;” might cause a buffer overflow. Note that we do not consider trivial cases such as array accesses with constant indexes in this work.
- **Pointer Usage (PU).** Similar to AU, PU usually appears in memory errors. This study only covers buffer overflow vulnerability.
- **Arithmetic Expression (AE).** This type of PSP is usually an arithmetic expression like “a + 1” or “a++”. AE is usually related to integer overflow and division-by-zero vulnerabilities. Here we mainly focus on the former. Note that we do not consider trivial cases such as self-increment and self-decrement operations with conditional checks in this work.

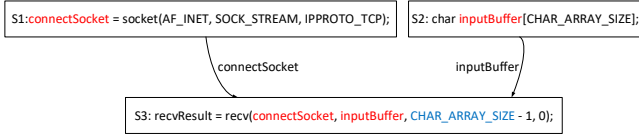


Fig. 5: An example of ignored data-dependence edges.

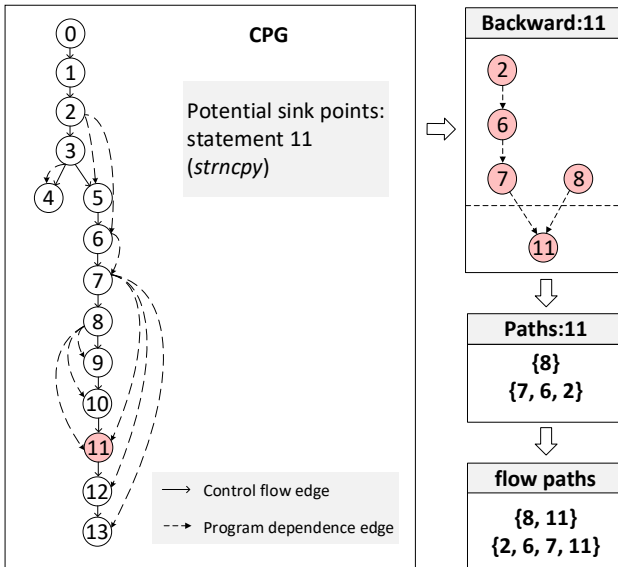


Fig. 6: An example to demonstrate how slicing works by revisiting the code in Figure 3.

2) *Dependent Statements*: The function “ExtractPrecNodes” (line 19) in Algorithm 1 establishes the dependence relation for the node “n” (i.e., identify nodes that the node “n” is dependent on). We find that not every dependence

relation for node “n” is related to the vulnerability, as a source code statement might contain multiple expressions among which only one could trigger the vulnerability. Therefore, we only focus on the control and data dependence involving key variables related to each PSP when extracting dependent nodes. For illustration in Figure 5, our tool identified arithmetic operation “CHAR\_ARRAY\_SIZE - 1” which might trigger integer underflow in statement S3. Although S3 is data-dependent with S1 via the variable “connectSocket”, they do not appear in arithmetic operations. We thus do not consider the data-dependence edge “S1 - S3” when performing slicing. For other nodes, we consider all dependent statements of the current node.

### C. Flow Path Selection

Among the flow paths, we aim to select one that can best locate the vulnerability-triggering statements based on the prediction results. The key intuition is that if a path contains both the PSP and its source node, the path should be selected. For example, the path “2 - 6 - 7 - 11” in the example in Section III. If there is more than one qualified path, we further rank them based on the path importance (to be detailed below) and select the one with the highest importance score.

More formally, given a code graph  $G$ , we extract flow paths from it and vectorize each flow path. The process of vectorizing one flow path is the same as that of detectors vectorizing the corresponding code graph. Then we compute the importance score for each flow path by treating each vectorized flow path as a subgraph of the original code graph and feeding it into the well-trained GNN-based vulnerability detector. This process could be formally described as:

$$p_g = \Phi(\text{vec}(g)) \quad (1)$$

where  $g$  is a flow path extracted from  $G$ ,  $\Phi$  is one of the GNN-based vulnerability detectors.

Finally, we compute the importance score  $IS_g$  for each path, measuring their contribution to the detector predicting the corresponding code fragment.

$$IS_g = 1 - (\Phi(\text{vec}(G)) - p_g) \quad (2)$$

Suppose there are  $n$  flow paths after slicing  $G$  and denoted as  $\{g_1, \dots, g_i, \dots, g_n\}$ . The vulnerability data flow  $g_*$  is denoted as:

$$g^* = \text{argmax } IS_{g_i} \quad (3)$$

## V. EXPERIMENTAL SETTINGS

We evaluate the effectiveness of VULEXPLAINER for locating vulnerability statements from the prediction results of DEEPWUKONG, REVEAL, IVDetect, and DEVIGN. The evaluation is conducted for detecting 8 of the top 30 vulnerabilities on CWE, comparing with five state-of-the-art explainers for GNN. To this end, we outline the dataset used in this study and the process involved in labeling them (Section V-A). Next, we provide a detailed exposition of the experimental setup (Section V-B).



```

stringprep (char *in,
            size_t maxlen,
            Stringprep_profile_flags flags,
            const Stringprep_profile * profile)
{
    int rc;
    char *utf8 = NULL;
    uint32_t *ucs4 = NULL;
    size_t ucs4len, maxucs4len, adducs4len = 50;

    do
    {
        uint32_t *newp;

        free (ucs4);
        ucs4 = stringprep_utf8_to_ucs4 (in, -1, &ucs4len);
        if (ucs4 == NULL)
            return STRINGPREP_ICONV_ERROR;
        maxucs4len = ucs4len + adducs4len;
        newp = realloc (ucs4, maxucs4len * sizeof (uint32_t));
        if (!newp)
            return STRINGPREP_MALLOC_ERROR;

        ucs4 = newp;

        rc = stringprep_4i (ucs4, &ucs4len, maxucs4len, flags, profile);
        adducs4len += 50;
    }
    ...

```

Fig. 7: An example function where lines marked green are added to fix the vulnerability.

#### A. Dataset

1) *Target Vulnerabilities.*: The dataset used here must support fine-grained detection, which requires explicit information on vulnerable code lines. Many flaw lines in real-world datasets like DEVIGN [9], REVEAL, Fan [26] are labeled with code change information extracted from committed version patches. As shown in Figure 7, a sample code with vulnerability ID CVE-2015-2029 includes the vulnerability-fixing code lines that are marked as green. However, such a labeling method only enables the detection of vulnerability-fixing lines, while leaving the vulnerability-triggering lines undetected. In the same example in Figure 7, the statements in the tainted flow are marked as pink, which does not cover the vulnerability-triggering code lines. In addition, a vulnerability fixed at function  $f_A$  might be triggered at function  $f_B$ . In such cases,  $f_A$  would be labeled as vulnerable while  $f_B$  as non-vulnerable. Even worse, Roland Croft et al [27] have report the existence of approximately 20-71% false positive vulnerability samples in the real-world dataset. From the above, accurately flagging vulnerable code lines in real-world datasets can be challenging.

Since noised datasets could affect the performance of deep learning models [28], we assemble our dataset from SARD [24], a synthetic vulnerability database. In the SARD dataset, each program (i.e., test case) could be related to one or more CWE IDs, as a program could contain different types of vulnerability. More importantly, the vulnerability-triggering statements of each vulnerable program have been properly marked. Our objective is to examine eight out of the 30 most hazardous software flaws in C/C++ for the year 2021, specifically focusing on CWE20, CWE22, CWE78, CWE119, CWE125, CWE190, CWE400, and CWE787. We employ the same web crawler as DEEPWUKONG [6] to collect all available programs.

TABLE I: Distribution of labeled samples from SARD.

Vulnerability Category	code graph	# vulnerable samples	# safe samples	# total
CWE20	XFG	58,350	174,250	232,600
	CPG	25,829	54,842	80,671
CWE22	XFG	8,771	11,500	20,271
	CPG	4,063	3,809	7,872
CWE78	XFG	4,643	8,385	13,028
	CPG	4,250	8,156	12,406
CWE119	XFG	34,901	80,155	115,056
	CPG	21,662	40,466	62,128
CWE125	XFG	6,147	12,469	18,616
	CPG	4,315	7,907	12,222
CWE190	XFG	4,173	10,168	14,341
	CPG	3,948	11,347	15,295
CWE400	XFG	11,296	37,417	48,713
	CPG	2,199	10,831	13,030
CWE787	XFG	23,493	49,718	73,211
	CPG	15,977	27,378	43,355
<b>TOTAL</b>	XFG	151,774	384,062	535,836
	CPG	82,243	164,736	246,979

2) *Benchmark Dataset Processing.*: The data collected from SARD is processed in the following steps. First, we parse the functions of SARD programs into CPGs for REVEAL and IVDetect. We directly utilize the slice level XFGs (a sub-graph of the PDG) generated by DEEPWUKONG as they are available on their repository [6]. Then we label and deduplicate those CPGs and XFGs following previous works [8], [6]. Any CPG or XFG containing one or more vulnerable statements will be labeled vulnerable and vice versa. Apart from that, we label the key statements in the vulnerable samples as node indexes.

3) *Benchmark Dataset Distribution.*: After the processing stage, we collect 82,243 vulnerable CPGs and 164,736 non-vulnerable CPGs from the SARD dataset, as listed in Table I. We downloaded the XFGs datasets from DEEPWUKONG at <sup>1</sup>. After relabeling, we assemble 151,774 vulnerable XFGs and 384,062 non-vulnerable XFGs in total.

#### B. Experimental Setup

1) *Experimental Configuration.*: The experiments are performed on a machine with 2 NVIDIA GeForce GTX TitanX GPUs and Intel Xeon E5-2603 CPU operating at 1.70GHz. We implement the graph neural networks based on PYTORCH GEOMETRIC [29].

Initially, we conduct individual detection tasks for each vulnerability type, whereby we train a model for each of the eight vulnerability categories. For dataset splitting, we randomly choose 80% of the programs for training, 10% for validating, and the remaining 10% for detecting. For model implementation we follow DEEPWUKONG [30], IVDetect [31], DEVIGN [32] and REVEAL [33]. We utilize the same hyper-parameters as detailed in the works. Note that DEEPWUKONG utilizes three different types of GNN models including GCN, GAT, and k-GNNs. Considering their similar effectiveness, we focus on the widely used GCN for simplicity in this work. The neural networks are trained in batches, and we set the batch size to 64. In the training stage, ADAM [34] is used for training and the learning rate is set to 0.001. All

<sup>1</sup>[https://bupteducn-my.sharepoint.com/:u:/g/personal/jackiecheng\\_bupt\\_edu\\_cn/EalnVAYC8zZDgwhPmGJ034cBYNZ8zB7-mNSNm-a7oYXkcw?e=eRUc50](https://bupteducn-my.sharepoint.com/:u:/g/personal/jackiecheng_bupt_edu_cn/EalnVAYC8zZDgwhPmGJ034cBYNZ8zB7-mNSNm-a7oYXkcw?e=eRUc50)

TABLE II: Detection performance of four detectors.

Detector	DeepWuKong	IVDetect	Reveal	Devign
Mean $F1$	0.95	0.97	0.95	0.93
Mean $ACC$	0.97	0.98	0.97	0.95

involved neural networks are randomly initialized by Torch initialization.

For detection result explanation tasks, we follow DIG [35] to implement 5 state-of-art explaining methods: PGEXPLAINER, GNNEXPLAINER, GRAD, DEEPLIFT, and GNN-LRP.

2) *Evaluation Metrics*: We first assess the effectiveness of four vulnerability detection tools with six commonly utilized metrics, including accuracy ( $ACC$ ), false positive rate ( $FPR$ ), false negative rate ( $FNR$ ), recall rate ( $R$ ), Precision ( $P$ ), F1 Score ( $F1$ ). The simplified results are summarized in Table II, and the fully-detailed results are available in our repository [16].

To assess the efficacy of explanation methods as well as VULEXPLAINER, we propose metric *Line Coverage*, or  $LC$ . Note the evaluation metric only applies to the true positive samples which are both labeled and detected as vulnerable.  $LC$  is defined as follows: given a flow path  $g$  of code fragment  $C$  containing  $n$  vulnerable code lines, then  $LC = \frac{n}{m}$  where  $m(m \geq n)$  represents the total number of code lines that are labeled as vulnerable in the dataset.  $LC$  is 1 if  $g$  contains all vulnerable statements labeled in the dataset,  $LC$  is 0 if  $g$  contains no vulnerable statements labeled.

Note that we have considered using fidelity [36] to measure the performance of explainers and VULEXPLAINER. However, there is a lack of a generalized and standard method for calculating fidelity, causing huge discrepancies among the derived result derived from different methods. This renders it unreliable to utilize fidelity as one of the evaluation metrics. In addition, our goal is to locate and interpret the causes for the detected vulnerabilities, which does not necessarily require constructing a subgraph that maximally retains the properties from the original graph.

## VI. VULEXPLAINER EVALUATION

Our evaluation aims to answer the following two research questions:

- RQ1 Can VULEXPLAINER accurately localize vulnerability-triggering code lines?** In particular, we would like to further individually investigate **RQ1.1**: can VULEXPLAINER perform consistently across various types of vulnerabilities? **RQ1.2**: can VULEXPLAINER perform consistently on different graph-based vulnerability detectors? In other words, is the performance of VULEXPLAINER impacted by the choice of detectors?
- RQ2 Can VULEXPLAINER outperform existing explaining methods for GNN-based vulnerability detection?**

### A. RQ1: The performance of VULEXPLAINER

**Overall Results.** For the computation efficiency, we set two parameters,  $sparsity$  and  $k$ , to control the number of nodes

TABLE III: Overall performance of VULEXPLAINER over eight types of vulnerability.

CWE-ID	20	22	78	119	125	190	400	787
Mean $LC$	0.84	0.92	0.98	0.87	0.97	0.90	0.89	0.88

TABLE IV: Performance of VulExplainer on four detectors.

method	$LC(k=3)$	$LC(k=5)$	$LC(k=7)$
DeepWuKong	0.93	0.94	0.94
Reveal	0.86	0.88	0.89
IVDetect	0.92	0.91	0.91
Devign	0.87	0.90	0.90

(i.e., program statements) in each flow path in our experiments. The parameter  $sparsity$  is calculated by  $1 - \frac{n}{m}$  where  $m$  and  $n$  are the total numbers of nodes in a graph and a path respectively. Intuitively,  $sparsity$  controls how evenly distributed are the nodes within the graph. More nodes concentrated in fewer paths (i.e., relatively longer paths) results in lower  $sparsity$ . The parameter  $k$  specifies the maximum number of nodes in a path, as mentioned in Algorithm 1. Taking both parameters into consideration, the max number of nodes  $MaxN$  in a flow path is given by  $\min(k, (1 - sparsity) * m)$ .

From our manual analysis, we observe that the value of  $sparsity$  has limited practical influence on the actual path length. Therefore, we adopt the fixed value of 0.5 for simplicity in our large-scale experiment. Meanwhile, we also observe that most vulnerable paths contain less than 7 nodes, we thus evaluate the explanation performance against  $k = 3, 5, 7$  for simplicity.

The overall explanation performance of VULEXPLAINER for each vulnerability type at  $k = 5$  is averaged over four vulnerability detectors, as listed in Table III. The overall explanation performance of VULEXPLAINER for each detector is average over the eight types of vulnerabilities, as listed in Table IV. In general, VULEXPLAINER attains encouraging outcomes. On average, the  $LC$  is above 90% for DEEP-WUKONG and IVDetect, above 85% for REVEAL and DEVIGN.

1) *RQ1.1: Detection Results across Vulnerabilities*: As shown in Table III, VULEXPLAINER demonstrates varying performance when locating different types of vulnerabilities. It achieves the highest  $LC$  score (98%) for CWE-78 vulnerability, and relatively lower  $LC$  scores for CWE-20 (84%) and CWE-119 (87%). To better understand the reasons behind such variations, we conduct a manual review of several exceptional cases and identified the following three possible causes.

- First, the selected PSPs might be incomplete. Vulnerabilities could be triggered by various types of statements. Certain vulnerabilities, such as CWE-78, can only be triggered by APIs related to system commands, making the vulnerability pattern comparatively straightforward. However, other kinds of vulnerabilities (e.g., buffer overflows) may be triggered by a variety of statements, including memory-related APIs, array operations, and pointer operations. This results in more PSPs in the code

graph, and subsequently more interference during path generation and selection, which poses a challenge to our method. Additionally, our target PSP patterns might not be complete, potentially excluding some vulnerability types during the explanation.

- We leverage program slicing to generate flow paths with the aim of preserving the vulnerability semantics. During this process, we filter out some control- & data-dependence relations without variables that are potentially relevant to vulnerabilities. Nonetheless, analyzing a large number of paths using our approach could become challenging, considering the exponentially expanding possibilities to be explored and parsed.
- The deep learning-based detectors are not as reliable as we expect. In some cases, the output score from the detector is significantly lower than expected, even when a path is strongly related to vulnerability. Such unreliability could consequently impact the measured effectiveness of VULEXPLAINER. This also indicates that traditional evaluation metrics may not fully capture the effectiveness of these detectors.

Nevertheless, VULEXPLAINER still exhibits high explanation effectiveness across all types of vulnerabilities covered in our evaluation.

**ANSWER: VULEXPLAINER demonstrates excellent performance in automatically determining crucial statements in a code graph when a vulnerability is detected, demonstrated by its ability to effectively explain the predictions of detectors across eight different types of vulnerabilities. A few exceptional cases could be introduced by the selection of PSPs.**

2) *RQ1.2: Comparison of Explaining Results on Different Detectors.*: The average detection performance of four detectors over eight types of vulnerabilities is summarized in Table II.

According to the metrics  $F1$  and  $ACC$ , the detection performance is ranked as follows:  $IVDETECT > DEEPWUKONG = REVEAL > DEVIGN$ . In comparison, the average vulnerability locating performance of VULEXPLAINER on the detection results of the four detectors is summarized in Table IV. In terms of the evaluation metric  $LC$ , the explanation performance ranking of VULEXPLAINER on the four detectors is  $DEEPWUKONG > IVDETECT > DEVIGN > REVEAL$ .

Comparing the results, we find that the performance of vulnerability detectors and that of the explainers are not necessarily aligned. For example, DEEPWUKONG performs relatively better for explanation than detection, as the adopted slice-level detection technique fits better with VULEXPLAINER. In particular, an XFG (slice) used in DEEPWUKONG comprises multiple statements related to a specific point of interest in the program, which is more fine-grained than approaches operating at the function level. Additionally, the XFG utilized in DEEPWUKONG typically incorporates interprocedural information. Thus, accurately retain code characteristics, including control and data flows, exclusively for code statements that are pertinent to the identified vulnerabilities. There-

fore although DEEPWUKONG might not surpass IVDETECT, VULEXPLAINER can still achieve better results on DEEPWUKONG.

**ANSWER: The performance of VULEXPLAINER is not strongly tied to the performance of the detector, as it achieves promising results across all four detectors. VULEXPLAINER performs better on DEEPWUKONG than other detectors, as it can perform detection on a more fine-grained level.**

#### B. RQ2: VULEXPLAINER VS. Existing Deep-learning-based Approaches

To answer RQ2, we compare VULEXPLAINER with five representative node-based and edge-based explaining methods, including GNNEXPLAINER [10], PGEXPLAINER [11], GRAD [12], GNN-LRP and DEEPLIFT [15]. Here we denote them as GE, PE, GR, GL, and DL respectively, and VULEXPLAINER as VE. We further summarize the technical details for each method in our repository [16] for a complete reference.

1) *Results*: As shown in Figure 8, when evaluated against the eight types of vulnerabilities using the evaluation metrics  $LC$ , VE surpasses all five explanation approaches in comparison.

Since the same trend in explainers' performance can be observed when the value of  $k$  is varied, we present and analyze the final results using  $k = 7$  due to page constraints. Taking the CWE-20 as an example, when locating vulnerable lines based on the predictions of DEEPWUKONG, VE is roughly 30% higher than GL in terms of  $LC$ . As for CWE-125, when locating vulnerable lines for REVEAL, GR only obtains 51%  $LC$ , while our approach achieves 96%. For IVDETECT, DL obtains only 4%  $LC$  while VE achieves 97%. Compared with GE, when locating vulnerable lines for DEVIGN on CWE-787, VE achieves 91%  $LC$ , nearly 33% higher than that of GE. A similar pattern is observed for PE, with only 33%  $LC$  achieved.

2) *Analysis*: The experimental results suggest that solely relying on node embeddings and topological structures of code graphs is insufficient to locate the root cause of the vulnerable code fragment.

We conduct a case study on the example we use in Section III, and the result is shown in Figure 9. We can observe that PE, GR, and GL do not succeed in identifying the statement that triggers the vulnerability. And GE, DL are unable to capture the connections between the vulnerability-triggering statement and other vulnerability-relevant statements, despite recognizing that statement "11" is vulnerability-relevant. VE determines whether a code fragment is vulnerable by selecting a flow path including both potential vulnerability-triggering statements and those passing along tainted data to them.

**ANSWER: VULEXPLAINER surpasses other explanation methods as it examines the semantic information of statements and the relationships among them in a program-dependent context.**



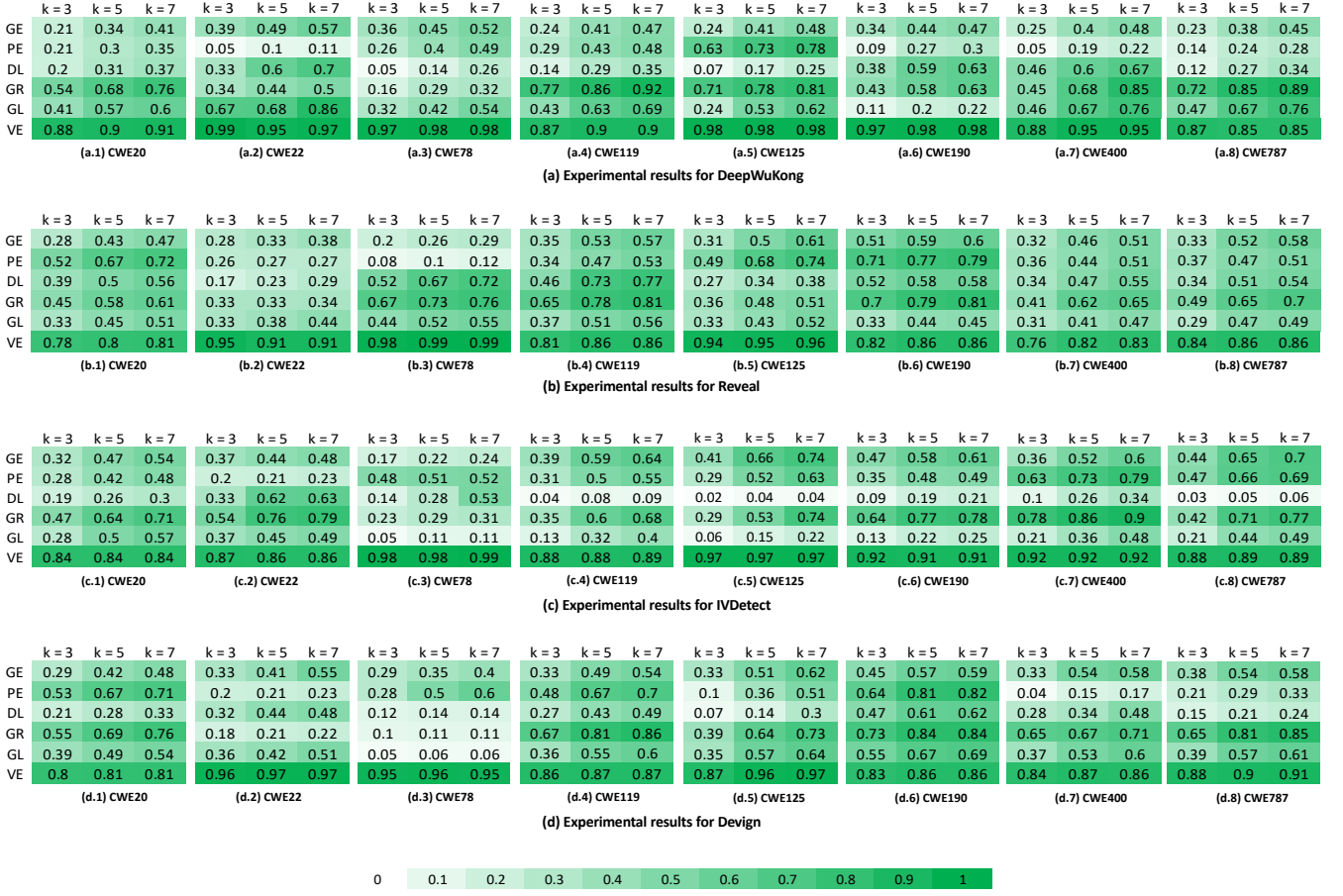


Fig. 8: Detailed experimental results on four detectors.

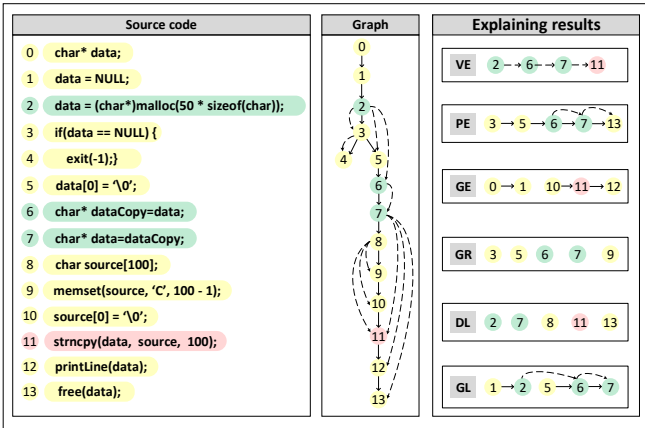


Fig. 9: Vulnerability locating results by different explainers for the prediction of REVEAL in the motivating example.

## VII. THREATS TO VALIDITY

**First**, we only conduct experiments on the SARD dataset, which contains synthetic and academic programs, but it may not be representative of real-world software products. We have discussed the problems in existing real-world datasets in section V-A. It remains an open problem to generate reliable datasets on a fine-grained granularity.

**Second**, our framework performs program slicing using key library API calls, array or pointer operations, and operator statements as PSPs. As mentioned in Section VI-A1, implying that certain corner cases might be neglected. In addition, it could also introduce irrelevant statements as sink points. One way to enhance our methodology is to detect additional supplementary types of PSP patterns, and subsequently screen out extraneous sink points to diminish potential inaccuracies. This requires extra insights into how real-world vulnerabilities are triggered and how they are fixed. We only consider analyzing PSPs here, although it is a promising diagonal research direction to further analyze potential source points where the data related to vulnerabilities are input to the programs.

**Third**, our experiments are limited to eight vulnerability types in C/C++ programs. Nonetheless, our methodology can be effortlessly expanded to encompass additional source-sink vulnerabilities and other programming languages.

**Fourth**, our approach only considers locating vulnerable statements based on four graph-based vulnerability detectors. However, our approach is easily applicable to other detectors, and potentially to other program analysis tasks.

## VIII. RELATED WORK

**Conventional static analysis tools.** Several conventional static program analysis frameworks (e.g. CLANG STATIC AN-

ALYZER [5], INFER [4], SVF [2], MALWUKONG [37]) have been designed to detect vulnerabilities or identify malicious behaviors in software systems. CLANG STATIC ANALYZER [5] is a constraint-based static analysis tool that performs symbolic execution to explore paths in the program’s control-flow graph and detect potential bugs. While INFER [4] is a static program analysis tool for detecting security issues such as null-pointer dereference and memory leaks based on abstract interpretation. SVF [2] first parses a program into a sparse value-flow graph(SVFG) then conduct path-sensitive source-sink analysis by traversing SVFG. The effect of conventional approaches depends on two factors: static analysis theories and security rules. static analysis theories include but are not limited to, parsing code into abstract structures (such as SVFG), where a better abstract structure facilitates the development of more sophisticated rules for detecting vulnerabilities. The effectiveness of detection rules depends on the expertise of the person who writes the rules. The quantity of rules is restricted, and it is impossible to encompass all of the vulnerability patterns, which frequently results in high rates of false positives and false negatives when analyzing intricate programs.

**Deep learning based vulnerability detection.** Compared to conventional static analysis, another field is machine/deep-learning-based analysis [38], [39], [40]. DeepBugs [41] represents code via text vector for detecting name-based bugs. VGDETECTOR [42] uses control flow graph and graph convolutional network to detect control-flow-related vulnerabilities. In this field, Devign [9] and REVEAL [7] utilize graph representations to represent source code to detect vulnerabilities. They aim to pinpoint bugs at the function level. VULDEEPECKER [43] applies code embedding using data-flow information of a program for detecting resource management errors and buffer overflows. SYSEVR [25] and *mu*VULDEEPECKER [44] extend VULDEEPECKER by combining both control and data flow and different Recurrent neural networks(RNN) to detect various types of vulnerability. DEEPWUKONG [6] utilizes program slicing methods to generate code fragments that are vectorized to apply the GNN model for classification. Hao et al. [45] extend CFG in the domain of exception handling, subsequently leveraging this extension to enhance the detection capability of existing DL-based detectors for exception-handling bugs. W Zheng et al. [17] combine DDG, CDG, and function call dependency graph (FCDG) into slice property graph (SPG), which is materialized into the implementation of the detection tool vulspg. Bin Yuan et al. [46] construct a behavior graph for each function and implement VulBG to enhance the performance of DL-based detectors by behavior graphs. All these solutions can only detect vulnerabilities on coarse granularity, and they could only tell whether a given code fragment is vulnerable.

**Fine-grained vulnerability detection.** On the basis of deep learning vulnerability detection, fine-grained vulnerability detection has received increasing attention in recent years. More recently, Zou et al. [47] proposes an explanation framework to select key tokens in code gadgets generated by VULDEEPECKER and SeVCs generated by SYSEVR to locate the vulnerable lines. VULDEELOCATOR [48] compiles source codes into LLVM IRs, performs program slicing, and uses

a customized neural network to predict relevance to vulnerabilities. LineVul [49] analyses each function with fine-tuned CodeBert and ranks each statement based on attention scores, a higher attention score implies stronger relation with vulnerability. While IVDetect [8] attain this goal by first identifying vulnerabilities at the source code level and utilizing existing explanation approach GNNEXPLAINER to generate a subgraph of the PDG to locate vulnerabilities in the function subsequently. However, several recent studies [50], [51], [52] have substantiated the inefficiency of current explanation approaches in vulnerability detection. proved the inefficiency of current explanation approaches in vulnerability detection.

**Machine-learning for software engineering.** In addition to vulnerability detection, deep learning has made significant progress in recent years in software engineering tasks such as code clone detection and code understanding, The main difference between these methods lies in the different vectorization processes proposed for their specific tasks. The vectorizing pipelines can be categorized into tokens-based [53], [54], [55], [56], ASTs-based [57], [58], [59], [60] and graphs-based [61], [62], [63], [64], [65], [66]. Complex vectorizing pipelines often yield better results on specific tasks, but also rely on more precise program analysis theories.

## IX. CONCLUSION

In this paper, we present VULEXPLAINER, a new method to locate vulnerable statements based on the predictions of GP-based vulnerability detectors by analyzing semantic information in statements of input programs and the program-dependence relationship between those statements. VULEXPLAINER first performs program slicing to extract the flow paths of a code fragment. Each flow path ends with a potential sink point (PSP) in the code. After program slicing, VULEXPLAINER utilizes a score function to compute an importance score for each path and select the path with the highest score as the explanation of the vulnerability data flow. We have applied VULEXPLAINER for 8 of the 30 most dangerous C/C++ vulnerabilities, and demonstrate that VULEXPLAINER outperforms several state-of-the-art explainers for GNN in the vulnerability detection task.

## REFERENCES

- [1] American Information Technology Laboratory. NATIONAL VULNERABILITY DATABASE, 2020. <https://nvd.nist.gov/>.
- [2] Yulei Sui and Jingling Xue. SVF: Interprocedural static value-flow analysis in LLVM. In *CC '16*, pages 265–266, 2016.
- [3] Checkmarx, 2020. <https://www.checkmarx.com/>.
- [4] Infer, 2020. <https://fbinfer.com/>.
- [5] Clang static analyzer, 2020. <https://clang-analyzer.llvm.org/scan-build.html>.
- [6] Xiao Cheng, Haoyu Wang, Jiayi Hua, Guoai Xu, and Yulei Sui. Deepwukong: Statically detecting software vulnerabilities using deep graph neural network. *ACM Trans. Softw. Eng. Methodol.*, 30(3), 2021.
- [7] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. Deep learning based vulnerability detection: Are we there yet? *arXiv preprint arXiv:2009.07235*, 2020.
- [8] Y. Li, S. Wang, and T. N. Nguyen. Vulnerability detection with fine-grained interpretations. 2021.
- [9] Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d’Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 10197–10207, 2019.
- [10] R. Ying, D. Bourgeois, J. You, M. Zitnik, and J. Leskovec. Gnnexplainer: Generating explanations for graph neural networks. *Advances in neural information processing systems*, 32:9240–9251, 2019.
- [11] D. Luo, W. Cheng, D. Xu, W. Yu, and .X. Zhang. Parameterized explainer for graph neural network. 2020.
- [12] P. E. Pope, S. Kolouri, M. Rostami, C. E. Martin, and H. Hoffmann. Explainability methods for graph convolutional neural networks. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.
- [13] T. Schnake, O. Eberle, J. Lederer, S. Nakajima, KT Schütt, KR Müller, and G. Montavon. Higher-order explanations of graph neural networks via relevant walks. 2020.
- [14] Xiao Cheng, Xu Nie, Ningke Li, Haoyu Wang, Zheng Zheng, and Yulei Sui. How about bug-triggering paths? - understanding and characterizing learning-based vulnerability detectors. *IEEE Transactions on Dependable and Secure Computing*, pages 1–18, 2022.
- [15] Avanti Shrikumar, Peyton Greenside, and Anshul Kundaje. Learning important features through propagating activation differences. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 3145–3153. PMLR, 06–11 Aug 2017.
- [16] Dataset Repository. <https://anonymous.4open.science/r/VulExplainerExp-84ED>, 2023.
- [17] Weining Zheng, Yuan Jiang, and Xiaohong Su. Vulspg: Vulnerability detection based on slice property graph representation learning. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, pages 457–467. IEEE, 2021.
- [18] X. Cheng, H. Wang, J. Hua, M. Zhang, and Y. Sui. Static detection of control-flow-related vulnerabilities using graph embedding. In *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2019.
- [19] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE ’81, page 439–449. IEEE Press, 1981.
- [20] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy (SP)*, pages 590–604, Los Alamitos, CA, USA, may 2014. IEEE Computer Society.
- [21] Quoc V. Le and Tomas Mikolov. Distributed representations of sentences and documents. In *ICML*, volume 32 of *JMLR Workshop and Conference Proceedings*, pages 1188–1196. JMLR.org, 2014.
- [22] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2, NIPS’13*, pages 3111–3119, USA, 2013. Curran Associates Inc.
- [23] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.
- [24] Software Assurance Reference Dataset, 2017. <https://samate.nist.gov/SARD/index.php>.
- [25] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, pages 1–1, 2021.
- [26] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. A c/c++ code vulnerability dataset with code changes and cve summaries. In *MSR ’20: 17th International Conference on Mining Software Repositories*, 2020.
- [27] Roland Croft, M. Ali Babar, and Mehdi Kholoosi. Data quality for software vulnerability datasets, 2023.
- [28] Xu Nie, Ningke Li, Kailong Wang, Shangguang Wang, Xiapu Luo, and Haoyu Wang. Understanding and tackling label errors in deep learning-based vulnerability detection (experience paper). In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 52–63, 2023.
- [29] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.
- [30] Cheng, Xiao and Wang, Haoyu and Hua, Jiayi and Xu, Guoai and Sui, Yulei. <https://github.com/jumormt/DeepWukong>, 2021.
- [31] Yi Li, Shaohua Wang, Tien N. Nguyen. <https://github.com/vulnerabilitydetection/VulnerabilityDetectionResearch>, 2021.
- [32] Yaqin Zhou and Shangqing Liu and Jing Kai Siow and Xiaoning Du and Yang Liu. <https://github.com/vulnerabilitydetection/VulnerabilityDetectionResearch>, 2019.
- [33] Chakraborty, Saikat and Krishna, Rahul and Ding, Yangruibo and Ray, Baishakhi. <https://github.com/VulDetProject/ReVeal>, 2020.
- [34] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.
- [35] Meng Liu, Youzhi Luo, Limei Wang, Yaochen Xie, Hao Yuan, Shurui Gui, Haiyang Yu, Zhao Xu, Jingtun Zhang, Yi Liu, Keqiang Yan, Haoran Liu, Cong Fu, Bora M Oztekin, Xuan Zhang, and Shuiwang Ji. DIG: A turnkey library for diving into graph deep learning research. *Journal of Machine Learning Research*, 22(240):1–9, 2021.
- [36] H. Yuan, H. Yu, S. Gui, and S. Ji. Explainability in graph neural networks: A taxonomic survey. 2020.
- [37] Ningke Li, Shenao Wang, Mingxi Feng, Kailong Wang, Meizhen Wang, and Haoyu Wang. Malwukong: Towards fast, accurate, and multilingual detection of malicious code poisoning in oss supply chains. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1993–2005. IEEE, 2023.
- [38] Stephan Neuhaus and Thomas Zimmermann. The beauty and the beast: Vulnerabilities in red hat’s packages. In *In Proceedings of the 2009 USENIX Annual Technical Conference (USENIX ATC)*, 2009.
- [39] Gustavo Grieco, Guillermo Luis Grinblat, Lucas Uzal, Sanjay Rawat, Josselin Feist, and Laurent Mounier. Toward large-scale vulnerability discovery using machine learning. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy, CODASPY ’16*, pages 85–96, New York, NY, USA, 2016. ACM.
- [40] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. Machine-learning-guided tpestate analysis for static use-after-free detection. In *Proceedings of the 33rd Annual Computer Security Applications Conference, ACSAC 2017*, page 42–54, New York, NY, USA, 2017. Association for Computing Machinery.
- [41] Michael Pradel and Koushik Sen. Deepbugs: A learning approach to name-based bug detection. *Proc. ACM Program. Lang.*, 2(OOPSLA):147:1–147:25, October 2018.
- [42] X. Cheng, H. Wang, J. Hua, M. Zhang, G. Xu, L. Yi, and Y. Sui. Static detection of control-flow-related vulnerabilities using graph embedding. In *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 41–50, Nov 2019.
- [43] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *The Network and Distributed System Security Symposium (NDSS)*, 2018.
- [44] Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li, and Hai Jin. mvuldeepecker: A deep learning-based system for multiclass vulnerability detection. *IEEE Transactions on Dependable and Secure Computing*, 18(5):2224–2236, 2021.
- [45] Hao Zhang, Ji Luo, Mengze Hu, Jun Yan, Jian Zhang, and Zongyan Qiu. Detecting exception handling bugs in c++ programs. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1084–1095. IEEE, 2023.

- [46] Bin Yuan, Yifan Lu, Yilin Fang, Yueming Wu, Deqing Zou, Zhen Li, Zhi Li, and Hai Jin. Enhancing deep learning-based vulnerability detection by building behavior graph model. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2262–2274. IEEE, 2023.
- [47] Deqing Zou, Yawei Zhu, Hai Jin, Hengkai Ye, Y Zhu, H Ye, Shouhuai Xu, and Zhen Li. Interpreting deep learning-based vulnerability detector predictions based on heuristic searching. *ACM Transactions on Software Engineering and Methodology*, 30, 08 2020.
- [48] Zhen Li, Deqing Zou, Shouhuai Xu, Zhaoxuan Chen, Yawei Zhu, and Hai Jin. Vuldelocator: A deep learning-based fine-grained vulnerability detector. *IEEE Transactions on Dependable and Secure Computing*, PP:1–1, 04 2021.
- [49] Michael Fu and Chakkrit Tantithamthavorn. Linevul: a transformer-based line-level vulnerability prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 608–620, 2022.
- [50] Tom Ganz, Martin Härterich, Alexander Warnecke, and Konrad Rieck. Explaining graph neural networks for vulnerability discovery. In *Proceedings of the 14th ACM Workshop on Artificial Intelligence and Security*, pages 145–156, 2021.
- [51] Yutao Hu, Suyuan Wang, Wenke Li, Junru Peng, Yueming Wu, Deqing Zou, and Hai Jin. Interpreters for gnn-based vulnerability detection: Are we there yet? In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1407–1419, 2023.
- [52] Baijun Cheng, Shengming Zhao, Kailong Wang, Meizhen Wang, Guangdong Bai, Ruitao Feng, Yao Guo, Lei Ma, and Haoyu Wang. Beyond fidelity: Explaining vulnerability localization of learning-based detectors, 2023.
- [53] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilingualistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.
- [54] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3):176–192, March 2006.
- [55] H. Sajnani and C. Lopes. A parallel and efficient approach to large scale clone detection. In *2013 7th International Workshop on Software Clones (IWSC)*, pages 46–52, May 2013.
- [56] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. Sourcerccc: Scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE ’16, pages 1157–1168, New York, NY, USA, 2016. ACM.
- [57] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. A novel neural source code representation based on abstract syntax tree. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE ’19, pages 783–794, Piscataway, NJ, USA, 2019. IEEE Press.
- [58] Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE ’16, pages 297–308, New York, NY, USA, 2016. ACM.
- [59] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL):40:1–40:29, January 2019.
- [60] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. *CoRR*, abs/1711.00740, 2017.
- [61] Kai Chen, Peng Liu, and Yingjun Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 175–186, New York, NY, USA, 2014. ACM.
- [62] Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE ’08, pages 321–330, New York, NY, USA, 2008. ACM.
- [63] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In Patrick Cousot, editor, *Static Analysis*, pages 40–56, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.
- [64] Jens Krinke. Identifying similar code with program dependence graphs. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE’01)*, WCRE ’01, pages 301–, Washington, DC, USA, 2001. IEEE Computer Society.
- [65] Chao Liu, Fen Chen, Jiawei Han, and Philip Yu. Gplag: Detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, volume 2006, pages 872–881, 01 2006.
- [66] Yulei Sui, Xiao Cheng, Guanqin Zhang, and Haoyu Wang. Flow2vec: Value-flow-based precise code embedding. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020.