Vulnerability Detection by Learning From Syntax-Based Execution Paths of Code

Junwei Zhang, Zhongxin Liu, Xing Hu, Xin Xia, and Shanping Li

***Abstract*—Vulnerability detection is essential to protect software systems. Various approaches based on deep learning have been proposed to learn the pattern of vulnerabilities and identify them. Although these approaches have shown vast potential in this task, they still suffer from the following issues: (1) It is difficult for them to distinguish vulnerability-related information from a large amount of irrelevant information, which hinders their effectiveness in capturing vulnerability features. (2) They are less effective in handling long code because many neural models would limit the input length, which hinders their ability to represent the long vulnerable code snippets. To mitigate these two issues, in this work, we proposed to decompose the syntax-based Control Flow Graph (CFG) of the code snippet into multiple execution paths to detect the vulnerability. Specifically, given a code snippet, we first build its CFG based on its Abstract Syntax Tree (AST), refer to such CFG as syntax-based CFG, and decompose the CFG into multiple paths from an entry node to its exit node. Next, we adopt a pre-trained code model and a convolutional neural network to learn the path representations with intra- and inter-path attention. The feature vectors of the paths are combined as the representation of the code snippet and fed into the classifier to detect the vulnerability. Decomposing the code snippet into multiple paths can filter out some redundant information unrelated to the vulnerability and help the model focus on the vulnerability features. Besides, since the decomposed paths are usually shorter than the code snippet, the information located in the tail of the long code is more likely to be processed and learned. To evaluate the effectiveness of our model, we build a dataset with over 231 k code snippets, in which there are 24 k vulnerabilities. Experimental results demonstrate that the proposed approach outperforms state-of-the-art baselines by at least 22.30%, 42.92%, and 32.58% in terms of Precision, Recall, and F1-Score, respectively. Our further analysis investigates the reason for the proposed approach's superiority.**

***Index Terms*—Vulnerability detection, deep learning, control flow graph, pre-trained model.**

I. Introduction

THE vulnerability detection system plays a vital role in software security [1], [2], [3], which can prevent a series of security incidents [4], [5], [6].

Manuscript received 17 October 2022; revised 9 June 2023; accepted 10 June 2023. Date of publication 15 June 2023; date of current version 15 August 2023. This work was supported by the National Natural Science Foundation of China under Grants 62202420 and 62141222. Zhongxin Liu gratefully acknowledges the support of the Zhejiang University Education Foundation Qizhen Scholar Foundation. Recommended for acceptance by J. Sun.*(Corresponding author: Zhongxin Liu.)*

Junwei Zhang, Zhongxin Liu, Xin Xia, and Shanping Li are with the College of Computer Science and Technology, Zhejiang University, Hangzhou, Zhejiang 310027, China (e-mail: jw.zhang@zju.edu.cn; liu_zx@zju.edu.cn; xin.xia@acm.org; shan@zju.edu.cn).

Xing Hu is with the School of Software Technology, Zhejiang University, Ningbo, Zhejiang 315103, China (e-mail: xinghu@zju.edu.cn).

Digital Object Identifier 10.1109/TSE.2023.3286586

Consequently, interest in more accurate and efficient automated software vulnerability detection methods has increased [7], [8], [9], [10].

In general, existing detection models can be broadly divided into two categories: (1) pattern-based vulnerability detection models [11], [12], [13], [14], [15] and (2) code similarity-based detection methods [9], [10], [16], [17], [18], [19]. Pattern-based vulnerability detection methods [11], [12], [13] rely on experts to manually define vulnerability rules or characteristics to detect vulnerabilities. These approaches require tedious manual efforts and are challenging to simultaneously achieve a low false positive rate and a low false negative rate [18], [20]. Code similarity-based methods adopt data mining and machine learning techniques to predict the presence of software vulnerabilities [16], [17], [18], [21]. This kind of method does not require experts to manually-crafted heuristics, can automatically capture vulnerability features, and has become a promising alternative.

Recently, benefiting from the powerful performance of deep learning (DL) techniques, a number of methods [16], [17], [18], [20], [21] have been proposed to leverage DL models to automatically learn vulnerability features from known vulnerabilities and identify unseen vulnerabilities in projects. For instance, Li et al. [18] proposed a program-slice-based approach named VulDeePecker, which slices source code based on library/API function calls and feds the sliced code into RNN [22] to detect vulnerabilities. Zhou et al. [10] proposed to convert a code snippet into a graph based on its Abstract Syntax Tree (AST), Control Flow Graph (CFG), Data Flow Graph (DFG), and natural code sequence and utilized Graph Neural Network (GNN) [23] to learn the code representation from the graph for identifying vulnerabilities.

Despite the promising performance, existing DL-based vulnerability detection methods are limited by two problems. (1) It is hard for them to identify and focus on vulnerability-related information from a large amount of irrelevant information. For instance, a buffer overflow vulnerability will be triggered when the amount of data in a memory buffer exceeds its storage capacity. This type of vulnerability is only related to the code that uses buffers. In other words, if many statements are unrelated to the buffer in the code snippet, these statements are not helpful for the detection of buffer overflow vulnerabilities. (2) Existing DL-based methods are less effective in handling the long code snippet. Due to limited GPU memory and computation resources, existing neural models often limit their input lengths, e.g., CodeBERT [24] only retains the first 400 tokens and directly truncates others in the input. Therefore, if the information

related to vulnerabilities locates in the truncated part of a code snippet, it would be very hard for existing methods to detect the vulnerability.

To alleviate the two problems, in this work, we propose a novel approach that can better capture vulnerability features from the code snippet, especially the code with much information unrelated to vulnerabilities and the long code snippet. We observe that a code snippet usually contains multiple execution paths to handle different situations, but it is often the case that only a few execution paths are vulnerable. Based on this observation, we propose to decouple a code snippet into multiple execution paths, use a neural model to learn the representation of each path, and combine the representations of multiple paths to obtain the final code representation.

For the first problem of existing methods mentioned above, since each execution path is a cohesive unit with a simple and linear structure, decoupling the code snippet into execution paths can help the model focus on coherent and highly correlated contextual information and better capture vulnerability features. For example, a use-after-free vulnerability is usually related to a single execution path. Compared to directly encoding the whole code snippet, separately encoding its execution paths can filter out some information unrelated to vulnerabilities and ease the extraction of vulnerability features. Meanwhile, because the execution order of each statement in an execution path is linear, neural models do not need to understand complex code structures. They can focus on capturing more accurate semantic information in the code snippet. *As for the second problem*, for the code with multiple execution paths, each of its execution paths is shorter than the whole code snippet and less likely to exceed the length limit of neural models. Hence, the tail of the long code has a greater chance of being processed by the neural network rather than truncated.

Based on the idea mentioned above, we propose a novel approach named **EPVD**, which decomposes a code snippet into several **e**xecution **p**aths for **v**ulnerability **d**etection. Specifically, given a code snippet, EPVD first parses it into an Abstract Syntax Tree (AST) and constructs its CFG based on the AST. We refer to such CFG as syntax-based CFG. Each node in the CFG refers to a statement, and each edge represents a control dependency between two statements. Then, EPVD selects a fixed number of paths that start from the entry node and end at one of the exit nodes from the CFG using a greedy-based path selection algorithm. We refer to such CFG paths as execution paths in this paper. We only select a fixed number of execution paths because loops in code may introduce infinite paths, and we find a few paths that can cover the information related to vulnerabilities in a code snippet. Each selected execution path is fed into a pre-trained code model to capture the intra-attention of the path and learn its feature vector. Further, we adopt a convolutional neural network (CNN) to capture the inter-path attention and fuse the representations of the selected paths to produce the code representation. Finally, a multi-layer perceptron (MLP) classifier is leveraged to detect the vulnerability based on the code representation.

To evaluate the effectiveness of EPVD, we curate a large C/C++ vulnerability dataset by merging three existing high-quality vulnerability datasets, i.e., the ReVeal dataset [25], the Big-Vul dataset [26], and the Devign dataset [10]. The new dataset contains more than 231 k code snippets with 24 k vulnerabilities. We evaluate our approach and compare it with five state-of-the-art DL-based methods. Experiments show that our model outperforms all the baselines by large margins. Specifically, EPVD improves the best-performing baseline by 22.30%, 42.92%, and 32.58% in terms of Precision, Recall, and F1-score, respectively. Further analysis demonstrates the effectiveness of our model in capturing vulnerability features from the vulnerability code snippets with much unrelated information and handling the long code.

In summary, the main contributions of this paper are as follows:

1) We implement a method for building the CFG of a code snippet based on its AST and propose a greedy-based algorithm to select representative execution paths from the CFG.
2) We propose a neural vulnerability detection model based on execution paths, named EPVD, which can better capture vulnerability features and handle long code.
3) We conduct comprehensive experiments to evaluate EPVD and justify the technical decisions in EPVD. Evaluation results show that EPVD outperforms all baseline models and the technical decisions in EPVD are reasonable and beneficial.
4) We release our replication package [27], which includes the source code, the dataset, and our evaluation results.

The remainder of this paper is organized as follows. We first introduce the related work in Section II. Then, we describe the motivation of our method and provide preliminaries of the syntax-based CFG in Section III. Section IV presents our method. The experimental results are presented in Section V. Section VI shows the limitations of our approach and some threats to validity. Finally, we conclude our work and discuss future work in Section VII.

II. Related Work

A. Software Vulnerability Detection With Neural Network

Various techniques have been developed to detect vulnerabilities. In the literature, early works mainly detected vulnerability by manually-designed vulnerability patterns [11], [12], [13]. However, these works require tedious manual efforts to analyze and craft vulnerability patterns. On the other hand, since some rules often contain the same syntax elements appearing in different code snippets, these syntax elements may lead to high false-positive and false-negative rates [14], [15], [20].

To reduce human efforts, recently, some studies leveraged neural network-based models to automatically learn the semantic features of the code snippet [17], [19]. Existing vulnerability detection models based on deep learning can be divided into two main categories: token-based and graph-based models. Token-based models regard the code as a flat sequence and utilize the

neural network to capture the vulnerability features from known vulnerabilities and detect unseen vulnerabilities [17], [18], [28]. For example, Russell et al. [17] leveraged the recurrent neural network (RNN) and convolutional neural network (CNN) to extract code features from code token sequences for vulnerability detection. Li et al. [18] used BiLSTM [29] to encode the sliced version of the input code snippet, namely the code gadget, to detect vulnerabilities. The authors slice each code snippet based on the "key points", such as library/API function calls. However, these token-based approaches do not consider the structure information of the source code, which leads to inaccurate detection.

Graph-based detection models represent the code using various graph representations and adopt the neural network to learn the structure properties of the code snippets for vulnerability detection [7], [8], [9], [10], [25], [30], [31]. For instance, Zhou et al. [10] adopted the gated graph recurrent network [23] to capture the structure information of the code snippets from three types of graph representations of the source code (i.e., AST, CFG, and DFG). Chakraborty et al. [25] proposed ReVeal, which learns the structure properties of the code snippet using the gated graph neural network [23], resampling techniques [32], and the triplet loss [33]. Li et al. [9] proposed IVDetect, which represents the code as the Program Dependency Graph (PDG) and treats vulnerability detection as the graph-based classification task via the graph convolution network. IVDetect first utilizes the GloVe model [34] to learn the representations of nodes in the PDG and then adopts the graph convolutional neural network [35] to optimize the code representation for detection. Since the two stages do not interact with each other during training, the optimal solution in the first stage may not lead to the optimal code representation in the second stage, resulting in the trained model failing to detect vulnerabilities effectively. We have tried our best to replicate and re-trained the IVDetect model (e.g., adopting multiple smaller learning rates and using the adaptive learning rate), but it cannot converge on our merged dataset. Hence, we do not compare it with the proposed model. Recently, Wu et al. [7] converted the PDG of the code snippets into an image to preserve the structure details. Further, they adopted three centrality indicators of the graph (i.e., degree centrality, katz centrality, and closeness centrality) to highlight the attention of vulnerable statements. Cao et al. [8] proposed a statement-level memory-related vulnerability detection approach based on the flow-sensitive graph neural network to jointly learn semantic and structure information.

Chen et al. [36] conducted a comprehensive empirical study to explore the gap in pinpointing bug-triggering paths between the traditional static bug detection approaches and existing neural network-based approaches. The authors first formalized the general vulnerability detection process and divided eleven approaches into three categories: method-level, slice-level, and statement-level. Method and slice-level approaches report the method or slice of the code snippet as vulnerable. Statement-level approaches report the vulnerable statements. Then, they proposed a new fine-grained metric called BTP to calculate the degree of overlap between the statements detected by the existing models and the statements on the bug-triggering paths. By conducting experiments on eleven approaches on the D2A [37] dataset, the authors found that existing vulnerability detection approaches are insufficient in pinpointing bug-triggering paths. Our proposed approach can be regarded as a method-level vulnerability detection method. Because it directly predicts whether a method is vulnerable or not based on multiple syntax-based execution paths instead of explicitly pinpointing vulnerable statements or slices.

Our approach is technically different from existing methods: First, our approach decomposes the input code snippet into several execution paths in its syntax-based CFG, which can simplify code structure and help neural models capture vulnerability features. Second, we propose a new greedy-based path selection method, which can cover as many code statements as possible. In other words, the decomposed syntax-based execution paths can indicate how the vulnerability originates and is triggered. Third, different from the existing method-level detection approaches, our approach applies the CNN to fuse multiple path representations to represent code snippets, which can help capture the bug-triggering information and inter-path attention. Fourth, our approach utilizes the pre-trained code model (i.e., CodeBERT) to learn code representations based on execution paths, which is not investigated by existing work.

B. Code Representation Models Based on Pre-Trained Models

Inspired by the excellent performance of pre-trained models in natural language processing (NLP), some researchers applied the pre-trained models to boost code-related tasks [24], [38], [39], [40], [41], [42]. Most work is dedicated to pre-training models on a massive corpus of source code and fine-tuning for a series of downstream tasks [24], [38]. For example, Feng et al. [24] presented CodeBERT that incorporates masked language modeling and replaced token detection as the pre-training objective to support code search and summarization tasks. CuBERT adopts masked language modeling and next sentence prediction as the pre-training objective to learn the code representation [38]. Besides, some pre-trained models consider the structure information of the code snippet at the pre-training stage [39], [40], [41], [43]. An example is that Guo et al. [43] proposed GraphCodeBERT, which learns data-flow information of the code snippet with edge prediction and node alignment tasks.

Due to the excellent performance of these pre-trained models on multiple code-related tasks to code representation, some studies have tried to adopt the pre-trained model to detect the vulnerability code [1], [2], [44]. However, all these methods directly utilize the pre-trained code model to perform prediction and face challenges in capturing vulnerability features from long code and code with complex structures. Instead, our model extracts multiple execution paths, which are more likely to learn vulnerability features from the tail statements in the long code. Besides, the order of each statement in the execution path is linear, which can ease the capture of the semantic information in the code.

III. Motivation and Preliminary

In this section, we first present the motivation of our approach through two real-world vulnerability code snippets. Then we introduce the definition of the syntax-based CFG.

```
1.  static long vbg_misc_device_ioctl(struct file *filp, unsigned int req, unsigned long arg)
2.  {
3.      struct vbg_session *session = filp->private_data;
4.      size_t returned_size, size;
5.      struct vbg_ioctl_hdr hdr;
6.      bool is_vmmdev_req;
7.      int ret = 0;
8.      void *buf;
9.      if (copy_from_user(&hdr, (void *)arg, sizeof(hdr)))
10.         return -EFAULT;
11.     if (hdr.version != VBG_IOCTL_HDR_VERSION)
12.         return -EINVAL;
13.     if (hdr.size_in < sizeof(hdr) || (hdr.size_out && hdr.size_out < sizeof(hdr)))
14.         return -EINVAL;
15.     size = max(hdr.size_in, hdr.size_out);
16.     if (_IOC_SIZE(req) && _IOC_SIZE(req) != size)
17.         return -EINVAL;
18.     if (size > SZ_16M)
19.         return -E2BIG;
20.     is_vmmdev_req = (req & ~IOCSIZE_MASK) == VBG_IOCTL_VMMDEV_REQUEST(0) ||
21.                     req == VBG_IOCTL_VMMDEV_REQUEST_BIG;
22.     if (is_vmmdev_req)
23.         buf = vbg_req_alloc(size, VBG_IOCTL_HDR_TYPE_DEFAULT);
24.     else
25.         buf = kmalloc(size, GFP_KERNEL);
26.     if (!buf)
27.         return -ENOMEM;
28.     *((struct vbg_ioctl_hdr *)buf) = hdr;
29.     if (copy_from_user(buf, (void *)arg, hdr.size_in)) {
30.         ret = -EFAULT;
31.         goto out;
32.     }
...  ......
67.     return ret;
```

(a)

```
1.  static void handle_data_packet(struct mt_connection *curconn, struct mt_mactelnet_hdr *pkthdr,
2.                  int data_len) {
3.      struct mt_mactelnet_control_hdr cpkt;
4.      struct mt_packet pdata;
5.      unsigned char *data = pkthdr->data;
6.      unsigned int act_size = 0;
...
14.     success = parse_control_packet(data, data_len - MT_HEADER_LEN, &cpkt);
15.     while (success) {
16.         if (cpkt.cptype == MT_CPTYPE_BEGINAUTH) {
17.             int plen,i;
...  ......
58.         else if (cpkt.cptype == MT_CPTYPE_PASSWORD) {
59.             #if defined(__linux__) && defined(_POSIX_MEMLOCK_RANGE)
60.             mlock(curconn->trypassword, 17);
61.             #endif
...  ......
93.         }
94.     }
```

(b)

Fig. 1. Motivation example. (The green shaded statement is the vulnerability.).

A. Motivation

To better demonstrate the two limitations of existing vulnerability detection models, we study some real-world vulnerabilities and obtain two important observations.

Observation 1: A vulnerable function may contain a vast of statements unrelated to vulnerabilities. For example, Fig. 1(a) presents a function from the Linux Kernel project [45], which contains an information leakage vulnerability published by CVE-2018-12633 [46]. Specifically, the line 9 and line 29 are the statements where the vulnerability is located. The function *vbg_misc_device_ioctl()* in this code snippet reads the same user data twice with the *copy_from_user()* function. The header of the user data is double-fetched, and a malicious user can tamper with the critical variables in the header between the two fetches, leading to severe kernel errors. As we can see, there are 67 lines in the function, but only a few of them, i.e., line 9 and line 29, are related to the vulnerability. If we directly input the whole function into a neural model, it would be difficult for the model to accurately capture the characteristics of the vulnerability. On the other hand, we notice that although this function contains a lot of execution paths, this vulnerability can only be taken advantage of by some of them. Therefore, we propose to reduce the information irrelevant to the vulnerability by extracting and sampling execution paths from the syntax-based control flow graph of the code, making it less difficult for neural models to capture vulnerability features.

Observation 2: Truncating the statements in the tail of a long code snippet may negatively affect the effectiveness of vulnerability detection. To reduce the GPU memory and computation resources, most neural vulnerability detection models limit the input size, which may cause the loss of critical information related to vulnerability. The function presented in Fig. 1(b) contains a buffer overflow vulnerability published in the MACTelnet project [47] exposed by CVE-2016-7115 [48]. This vulnerability locates in line 58 and allows remote servers to execute arbitrary code via a long string in an *MT_CPTYPE_PASSWORD* control packet. If we tokenize this function using a subtoken-based tokenizer, such as the tokenizer used by CodeBERT [24], the vulnerable line will be tokenized into the 698-708 tokens. Since the input limits of most detection models are less than 512, the vulnerable line will be truncated by such models, and it would be difficult for them to capture the features of this vulnerability from this function. Intuitively, we can alleviate this problem by splitting this function into several parts and separately encoding these parts using neural models. However, it is challenging to keep vulnerability-related structural and semantic information in a code snippet after splitting it. In this work, we tackle this challenge by decomposing a code snippet into multiple execution paths extracted and selected from its syntax-based CFG. Such execution paths are usually shorter than the whole code snippet, hence are less likely to be truncated. In addition, a code snippet can be regarded as the combination of all its execution paths, and the structure of each path is simple and linear. Therefore, decomposing a code snippet into multiple execution paths can also simplify the capture of the structural information in a code snippet to some extent.

B. Syntax-Based Control Flow Graph

As described above, our approach aims to decompose a code snippet into multiple execution paths and learn its representation from such paths. In practice, it would be expensive to dynamically extract the execution paths for vulnerability detection. It is also expensive to compile large projects, such as the Linux Kernel. Besides, the vulnerability data provided in existing datasets are usually uncompilable and sometimes even incomplete code snippets. So the techniques that require compilation, such as symbolic execution, are not applicable to our scenarios. Therefore, we choose to extract execution paths in a static way. Specifically, we choose to construct the CFG of a code snippet based on its AST and extract its execution paths from the CFG. We refer to such CFG as the syntax-based CFG, where each node represents an individual statement in the code snippet, and each directed edge represents a possible execution order between statements. The syntax-based CFG is similar to the CFG built by Joern [49]. Joern is a widely used static analysis tool and can also construct CFGs based on ASTs. However, Joern builds CFGs by analyzing finer-grained AST nodes, such as operators, which is not very efficient if we only

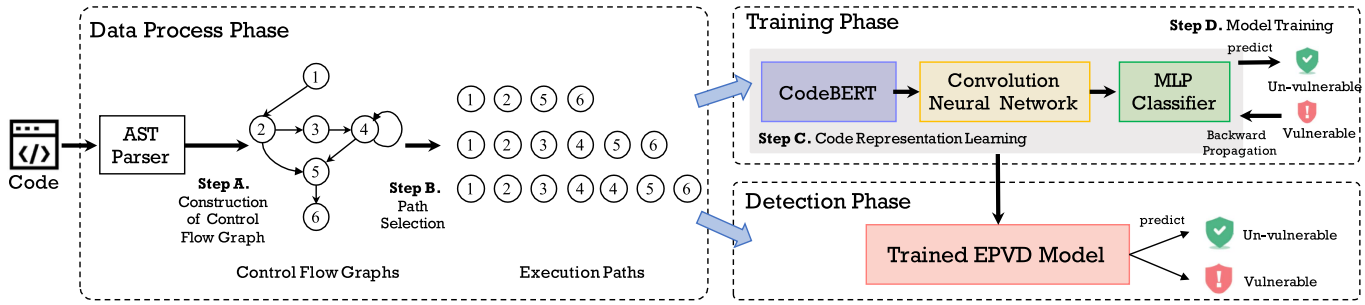


Fig. 2. The framework of the proposed approach.

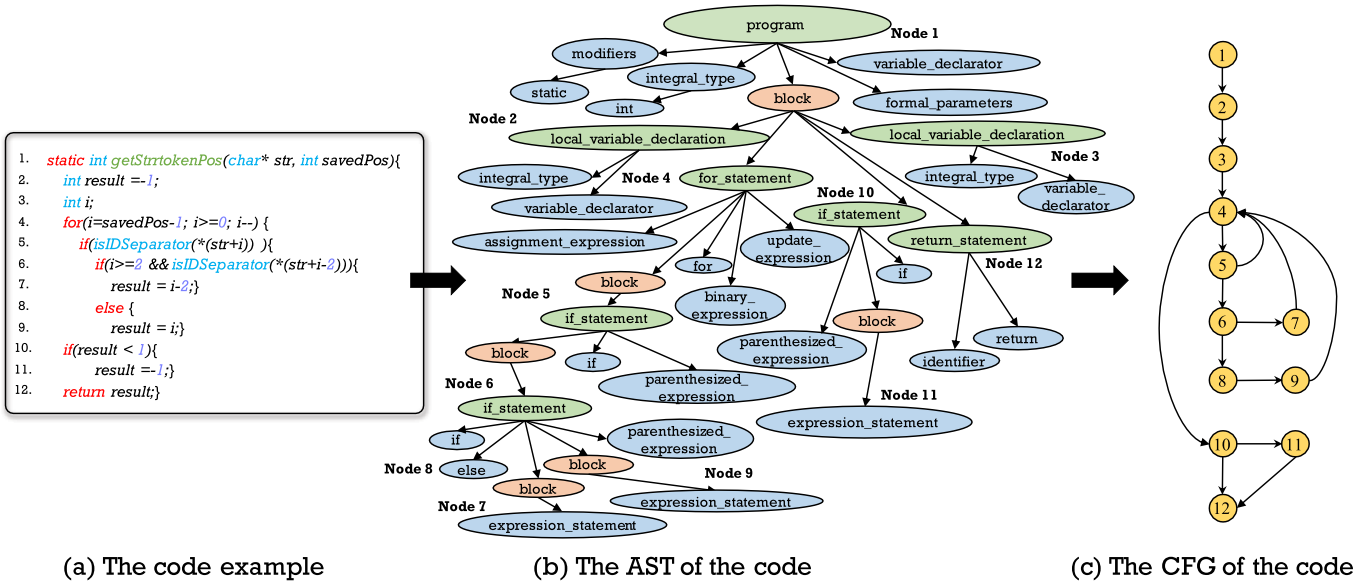


Fig. 3. The construction of the syntax-based CFG.

need to construct CFGs. In our experimental environment, it takes about 4,560 seconds to construct CFGs for 1,000 code snippets using Joern, while the time cost is only 49 seconds for our syntax-based CFG construction method. Therefore, we use the syntax-based CFG construction method instead of Joern.

Formally, we adopt $G_i = (\mathcal{V}_i, \mathcal{E}_i)$ to denote the syntax-based CFG of the code snippet c_i, where $\mathcal{V}_i = (v_i^1, v_i^2, \ldots v_i^{|\mathcal{V}_i|})$ is the node set containing $|\mathcal{V}_i|$ statements. $\mathcal{E}_i$ is the edge set representing the control flows between statements. Each edge is a relation between two statements in c_i, such that one statement could be executed after the other. A path in G_i is a node sequence $p = (n_1, n_2, \ldots, n_k)$, where the node n_k is the statement k of c_i. For any pair of adjacent nodes n_p and n_q, there exists an edge from n_p to n_q. If the start node equals the end node, the path will be a cycle.

IV. The Proposed Approach

We elaborate on the proposed approach in this section. The overall workflow of our approach is illustrated in Fig. 2. We first parse each code snippet into an AST and construct the syntax-based CFG according to the AST. Next, we propose a greedy-based path selection algorithm to select multiple execution paths from the syntax-based CFG, i.e., decompose the code snippet into several execution paths. Then, the selected paths are encoded by CodeBERT [24] into vectors with the intra-path attention, which are then fed into the CNN to capture the inter-path attention. Finally, we leverage an MLP classifier to perform the detection.

A. Construction of Control Flow Graph From AST

This phase takes a code snippet as input, parses it into an AST using the tree-sitter [50], and constructs its syntax-based CFG from the AST. In this section, we use the example presented in Fig. 3 to illustrate how we construct the syntax-based CFG from the AST of a code snippet.

Before constructing the CFG, we remove blank lines and comments in the code snippet using regular expressions. We also mark the line number of each statement in the code. Since each node in the syntax-based CFG represents an individual statement, we only consider statement nodes in the AST when constructing the CFG. To ease the presentation, we make the following definitions:

- *Simple Statement*: the statements that do not contain other statements in their AST.
- *Next Statement*: a Next Statement of a node refers to a statement that is possible to be executed after executing the node. One node may have multiple Next Statements.
- *Non-Child Next Statement*: a Non-Child Next Statement of a node is a statement that is a Next Statement of this node and is not included in the sub-tree with this node as the root.
- *Normal Statement*: a statement that are not *break_statement*, *continue_statement*, *return_statement*, and *throw_statement*.

To construct the syntax-based CFG of a code snippet, we first add all the statements of the code snippet into the CFG as its nodes and regard the first statement as the entry node and all *return_statement*, *assert_statement* and *throw_statement* as exit nodes. If the last statement of the code snippet is not an exit node, we add a dummy exit node at the end of the code. Then, we traverse the AST in a breadth-first way and design rules for each statement type to build the edges in the CFG.

1) For each statement that is both a Simple Statement and a Normal Statement, if its next sibling statement exists in AST, we connect it to this sibling. For example, in Fig. 3, we connect node 2 to node 3 and node 3 to node 4.
2) For each loop statement, i.e., *for_statement* and *while_statement*, we connect it with its first child statement and its next sibling statement if such a statement exists. If its last child statement is a Normal Statement, we connect this statement to the loop statement. For example, in Fig. 3, we connect node 4 to node 5, node 4 to node 10, and node 5 to node 4.
3) For each *break_statement*, we first find its first ancestor that is a loop statement or a *switch_statement* along the AST. Then, we connect it to the Non-Child Next Statement of the ancestor.
4) For each *continue_statement*, we first find its first ancestor that is a loop statement along the AST. Then, we connect it to this ancestor.
5) For each *if_statement*, we first connect it to its next sibling statement if such statement exists and connect the last child statement of its *then_block* to its current Next Statements if the last child statement is a Normal Statement. For example, in Fig. 3, we connect node 6 to node 4, node 7 to node 4, node 10 to node 12, and node 11 to node 12. Then, if its children contain an *else_statement*, we connect the *else_statement* to each of its Next Statements, remove the edges from the *if_statement* to its Next Statements, and add the edge from the *if_statement* to the *else_statement*. For Fig. 3, we connect node 8 to node 4, remove the edge from node 6 to node 4 and connect node 6 to node 8. Next, we traverse the *else_statement*. We connect its last child statement to its current Next Statements. If its last child statement is a Normal Statement, we remove the edges from it to its Next Statement and connect it to its first child statement. For Fig. 3, we connect node 9 to node 4, remove node 8 to node 4 and connect node 8 to node 9. Finally, we connect the *if_statement* to the first child statement of its *then_block*. For Fig. 3, we connect node 5 to node 6, node 6 to node 7 and node 10 to node 11.
6) For each *switch_statement*, we connect it to its first *case_statement*. For each *case_statement*, we first connect it to the next *case_statement* or *default_statement*. Then, if the last child statement of this *case_statement* is a Normal Statement, we connect it to the current Next Statements of this *case_statement*. Finally, we connect the *case_statement* to its first child statement. For each *default_statement*, we connect it to its first child statement and connect its last child statement to the Non-Child Next Statements of the *switch_statement* if its last child statement is not a Normal Statement.
7) For each *try_statement*, we treat its *catch_clauses* as statements. We cannot construct a "sound" CFG for a *try_statement* since we cannot know which exceptions each function call may throw only from the caller's AST. In addition, building a "complete" CFG requires connecting each statement in the *try_block* to each *catch_clause*, which may introduce too many dead paths and negatively affect the following phases. Therefore, we choose to only connect the last Normal Statement in the *try_block* to *catch_clauses*. Specifically, we connect the *try_statement* to the first statement in its *try_block*, connect the last Normal Statement in its *try_block* to its first *catch_clause*. For each *catch_clause*, we connect it to its first statement and the next *catch_clause*. In addition, for the last statement which is a Normal Statement in the *try_block* and each *catch_clause*, we connect it to the Non-Child Next Statement of the *try_statement*.

B. Path Selection

A code snippet can be regarded as the combination of all its execution paths. However, a code snippet may have infinite execution paths if it contains loops. It may require many computation resources to encode all the execution paths of a code snippet. Therefore, we argue that it is impractical to encode all the execution paths in a CFG for learning the representation of the corresponding code snippet. We define an execution path in a syntax-based CFG as a path from the entry node to an exit node of the CFG and refer to it as an execution path from hereon. In addition, it is non-trivial to accurately locate the vulnerable statements in an unseen code snippet. If we only extract one execution path to represent the code snippet, it is very likely to miss vulnerable statements and negatively affect the performance of vulnerability detection. Fortunately, based on our observations of real-world vulnerabilities and the motivating examples presented in Section III, we find that a few execution paths can often cover the root cause of a vulnerability. Therefore, instead of encoding all or only one of the execution paths in a CFG, we select and encode several representative execution paths to represent the corresponding code snippet. A similar strategy is also used by Alon et al. [51] when representing a code snippet as AST paths. This phase is responsible for selecting a few representative execution paths from the CFG constructed by the previous phases.

There are two requirements for selecting execution paths: First, to avoid losing important information in the code snippet, the selected paths should cover as many lines of code as possible. Second, to reduce the burden of model training, we expect the selected paths to be as short as possible. Unfortunately, the two requirements are conflicting to some extent. To make a trade-off between them, we propose a greedy-based path selection algorithm.

Algorithm 1 presents how we select execution paths. To ease the presentation, we make the following definitions:

- *Branch Node*: a node of which the out degree is larger than 1 in the CFG.
- *Branch Edge*: an edge of which the source node is a branch node in the CFG.
- *Path Weight*: the weight of a path is the sum of the weights of all the edges in this path.

Suppose that we can select at most m paths from the CFG. The key idea behind our algorithm is that for the first $m-1$ paths, we try to select diverse and short paths, while for the last path, we try to maximize the total node coverage of all paths. Specifically, first, we set the initial weight of all edges in the CFG to one and mark all nodes in the CFG as uncovered. Then, we select the first $m-1$ execution paths one by one. For each of these paths: (1) For each exit node, we select a candidate execution path that contains at least one uncovered node and has the least path weight from the CFG. (2) We pick out the path with the most uncovered nodes from the selected candidates. For example, in Fig. 3, the first selected execution path is (1, 2, 3, 4, 10, 12). (3) We mark each node in this path as covered and increase the weights of the Branch Edges in this path by 100 to reward the exploration of new edges. (4) We continue to select the next shortest execution path with uncovered nodes from the CFG with the updated weights. For example, in Fig. 3, the second selected execution path is (1, 2, 3, 4, 10, 11, 12) since the weight of the edge from node 10 to node 12 has been increased. Finally, we select the last path to cover as many nodes as possible. In detail, we iterate the execution paths in the CFG, ignore the paths with cycles, find the paths that cover the most uncovered nodes, and choose the shortest one. We ignore the execution path with cycles because, for each such path, there must exist an execution path that covers the same set of nodes but does not contain any cycle and is shorter.

In Section V, we provide more detail to evaluate the influence of different path selection algorithms and the number of paths on model performance.

Algorithm 1: The Path Selection Algorithm.

Input : G: a CFG with edges $\mathcal{E}$, nodes $\mathcal{V}$, entry node v_0 and exit node set V_e,
m: the number of paths to select,
$\mathcal{V}_b$: Branch Nodes in G_i,
$\mathcal{E}_b$: Branch Edges in G_i,
w: edge weights,
r: coverage rate, initialized as 0,
$\mathcal{V}_c$: covered node, initialized as $\emptyset$,
$\mathcal{V}_{uc}$: uncovered node, initialized as $\mathcal{V}$.

Output: P: the selected paths.

$\tilde{P} \leftarrow \emptyset$;
for $i \leftarrow 1$ **to** $m-1$ **do**
  for $j \leftarrow 0$ **to** $|V_e|-1$ **do**
    // find the shortest path with uncovered nodes
    $\hat{P} \leftarrow$ findShortestPath(G_i, v_0, $V_e[j]$, V_{uc});
    $\mathcal{V}_c \leftarrow \hat{P} \cap V_{uc}$;
    if $|\mathcal{V}_c| \geq r$ **then**
      $\tilde{P} \leftarrow \hat{P}$;
      $r \leftarrow |\mathcal{V}_c|$;
    end
  end
  $P[i] \leftarrow \tilde{P}$;
  $\mathcal{V}_c \leftarrow \tilde{P} \cup \mathcal{V}_c$;
  $\mathcal{V}_{uc} \leftarrow \mathcal{V}_{uc} - \tilde{P}$;
  for e *in* $(\mathcal{E}_b \cap \tilde{P})$ **do**
    $w[e] \leftarrow w[e]$ + 100;
  end
  r $\leftarrow$ 0;
end
for $j \leftarrow 0$ **to** $|V_e|-1$ **do**
  // find all paths from the entry node to exit nodes
  $\hat{P} \leftarrow$ `Allpath(`G_i, v_0, $V_e[j]$`)`;
  for $\acute{P} \leftarrow \hat{P}$ **do**
    $\acute{\mathcal{V}}_c \leftarrow \mathcal{V}_{uc} \cap \acute{P}$;
    if $|\acute{\mathcal{V}}_c| \geq r$ **then**
      $\tilde{P} \leftarrow \acute{P}$;
      $r \leftarrow |\acute{\mathcal{V}}_c|$;
    end
  end
end
$P[m] \leftarrow \tilde{P}$;

C. Code Representation Learning

This phase aims to learn the representation of the target code snippet from the selected m execution paths. Intuitively, to obtain a good representation, we need to capture both the features of each path and the relationships among paths.

This phase first leverages the pre-trained CodeBERT model [52] to learn the intra-path attentions and encode each path into a feature vector. We use CodeBERT because it is a transformer-based model, which can better capture long-term dependencies within a long sequence compared to RNN-based models, and it is pre-trained on massive code data and has shown to be effective in code understanding and code generation tasks [24]. Specifically, given a selected path P, we first concatenate all of its nodes, i.e., its statements, in order into a code snippet S. Then, CodeBERT tokenizes S into a token sequence of length n using a subword tokenizer [53], maps the token sequence into a sequence of embeddings $\tilde{\mathbf{X}} = \{\tilde{\mathbf{x}}_1, \tilde{\mathbf{x}}_2, \ldots, \tilde{\mathbf{x}}_n\}$ based on an embedding layer and learn the contextual embedding of each code token using multiple Transformer layers. A Transformer layer takes as input a sequence of vectors, e.g.,

$\tilde{\mathbf{X}}$, adopts a multi-head self-attention layer [54], a feed-forward layer and the layer normalization operation [55] to capture the intra-path attentions and refine the input vectors, as follows:

$$\hat{\mathbf{X}} = \text{MultiHead}(\tilde{\mathbf{X}}, \tilde{\mathbf{X}}, \tilde{\mathbf{X}}), \tag{1}$$

$$\mathbf{X}^i = \text{LayerNorm}(\hat{\mathbf{X}} + \text{FFN}(\hat{\mathbf{X}})), \tag{2}$$

where MultiHead$(\cdot)$, FFN$(\cdot)$ and LayerNorm$(\cdot)$ denotes the multi-head self-attention layer [54], the feed-forward layer and the layer normalization operation [55], respectively. i in $\mathbf{X}^i$ means $\mathbf{X}^i$ is the output of the i_{th} Transformer layer. After being processed by l Transformer layers, path P is encoded as a sequence of contextual embeddings $\mathbf{X}^l = \{\mathbf{x}_1^l, \mathbf{x}_2^l, \ldots, \mathbf{x}_n^l\}$. We regard the embedding $\mathbf{x}_1^l$ of the special token "[CLS]" inserted at the beginning of S as the representation of P, which is a common practice when using CodeBERT for classification tasks [24]. Since we decompose a code snippet into multiple paths, we use CodeBERT to encode each path, respectively, and vertically concatenated their representations into a matrix $\mathbf{E} \in \mathbb{R}^{m \times d}$, as follows:

$$\mathbf{E} = \begin{bmatrix} \mathbf{e}_1 \\ \ldots \\ \mathbf{e}_m \end{bmatrix}, \tag{3}$$

where $\mathbf{e}_i$ refers to the representation of the i_{th} path.

Then, this phase adopts a CNN to learn the inter-path attentions and fuse the feature vectors of the selected paths, i.e., $\mathbf{E}$. Specifically, following the idea of using CNN in text classification [56], we first use a CNN with k convolution filters to capture inter-path features from $\mathbf{E}$, as follows:

$$\mathbf{c}_j = \phi_c\left(\mathbf{E} \odot \mathbf{F}^j\right), \tag{4}$$

where $\odot$ denotes the inner product operator, $\phi_c(\cdot)$ is the convolution operation, $\mathbf{F}^j \in \mathbb{R}^{q \times d}$ is the j_{th} filter, q is the window size of the filter. d denotes the dimensions of the representation. To capture the most significant features, we then use max pooling to combine the convolution results of all filters, as follows:

$$\mathbf{C} = MaxPooling(\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_k). \tag{5}$$

Finally, we combine $\mathbf{C}$ and the representations of all the selected paths to obtain the final code representation, and leverage an MLP classifier to perform vulnerability detection, as follows:

$$\mathbf{Z} = [\mathbf{C} \parallel \mathbf{e}_1 \parallel \cdots \parallel \mathbf{e}_m], \tag{6}$$

$$\hat{y} = \text{Softmax}(\text{MLP}(\tanh(\mathbf{Z}))), \tag{7}$$

where $\parallel$ represents the horizontal concatenation operation.

D. Detection Model Training

We train the classifier by minimizing the following loss:

$$\mathcal{L}(\hat{y}, y) = -y \cdot log\left(\hat{y}\right) + (1 - y) \cdot log\left(1 - \hat{y}\right), \tag{8}$$

where y is the true value. $\hat{y}$ is the output value.

TABLE I
THE STATISTICS OF DATASETS

Dataset	# Non-vul	# Vul	# Ratio	# Total
REVEAL	20,494	2,240	9.15: 1	22,734
Big-Vul	177,736	10,900	16.31: 1	188,636
Devign	14,858	12,460	1.19: 1	27,318
Merged	207,059	24,241	8.54: 1	231,300
Redis	1,233	82	15.04: 1	1,315
Lua	750	32	23.44: 1	782

V. EXPERIMENTS

In this section, we conduct extensive experiments to justify our model's superiority and analyze the reasons for its effectiveness. Specifically, we aim to answer the following research questions:

RQ1: How effective is EPVD compared with the state-of-the-art baselines on vulnerability detection?

RQ2: What are the effects of different design choices for the proposed model?

RQ3: Can EPVD outperform existing vulnerability detection approaches across various types of vulnerabilities?

RQ4: How does the size of training data affect the performance of EPVD?

RQ5: How effective is EPVD in real-world applications?

RQ6: How efficient is EPVD in detecting vulnerabilities?

A. Datasets

To evaluate the effectiveness of EPVD, we established two datasets: (1) The dataset merged from multiple existing vulnerability datasets, (2) The datasets collected from additional real-world open-source projects. The statistics of the two datasets are shown in Table I. Column 2 and Column 3 are the numbers of non-vulnerability and vulnerability functions, respectively. Column 4 denotes the ratio between the number of non-vulnerability functions and the vulnerability functions in each dataset. Column 5 is the total number of the source code in different datasets.

For the merged dataset, we merge three existing high-quality datasets, i.e., the REVEAL dataset [25], the Big-Vul dataset [26], and the dataset constructed by the authors of Devign [10] (from hereon, the Devign dataset), into one large-scale dataset. The REVEAL dataset [25] contains more than 22 K functions from two large-scale open-source projects, i.e., Linux Debian Kernel and Chromium, and 9.15% of the functions are vulnerable. The Big-Vul dataset [26] is built from the CVE entries from 2002 to 2019, covering 348 different projects and 91 different vulnerability types. The Devign dataset is collected from two large C projects, i.e., QEMU [57] and FFmpeg [58]. The three datasets are all built from real-world C/C++ vulnerabilities in open-source projects and have been widely used by prior work on vulnerability detection [3], [9], [10], [25], [26]. We randomly split the merged dataset into disjoint training, validation, and test sets with 80%, 10%, and 10% of the dataset, respectively. To avoid data duplication, we remove duplicated test samples, each of which is the same as one or more training samples, after converting each sample as tokens using a C/C++ lexer and removing its comments. Finally, there are more than 231 k

TABLE II
THE STATISTICS OF DIFFERENT TYPES OF CWE

CWE Types	# Non-vul	# Vul
CWE119	16,964	1,986
CWE20	9,293	1,088
CWE399	6,056	709
CWE125	4,732	554
CWE264	4,109	481
CWE189	2,793	327
CWE416	2,674	313
CWE190	2,451	287
CWE362	2,246	263
CWE476	1,691	198

functions left in the merged dataset, and 10.5% of them are vulnerable.

As for the second dataset, we follow previous work [28] and construct the dataset by collecting security-related commits from Redis [59] and Lua [60]. Redis is a well-known database system server. Lua is a widely-used script language. Following Chen et al. [28], we first extract bug-fixing commits by selecting the commits whose messages contain bug-related keywords, such as "bug", "crash", "memory error", "vulnerability", and "fix". Then three experienced software engineers carefully examine the modifications of each bug-related commit, understand how the bug is fixed, and label safe functions from the fixed commit as negative and vulnerable functions from the corresponding vulnerable commit as positive. It took 120 hours to manually examine all the bug-related commits. Due to the small number of vulnerabilities in the Lua project, we also merge the datasets of the Redis and Lua projects and named it as "mixed dataset".

In addition, we seek to study EPVD's performance for different types of vulnerabilities. Since the Big-Vul dataset [26] is crawled from real-world CVE information and includes multiple vulnerability types, we identify the top 10 vulnerability types with the highest number of vulnerabilities from the Big-Vul dataset. For each vulnerability type, we first pick out the vulnerable code snippets of this type from the Big-Vul dataset and then randomly select a certain proportion of non-vulnerable code snippets from all the non-vulnerable code snippets in the merged dataset so that the ratio of the selected vulnerable code snippets to the selected non-vulnerable snippets is the same as that in the merged dataset. We put 80% of the code snippets into the training set and split the remaining 20% of code snippets into the validation and test set, respectively. Table II summarizes the detailed information of different types of vulnerabilities.

CWE119. Improper Restriction of Operations within the Bounds of a Memory Buffer: This type of vulnerabilities includes the operation on a memory buffer, but it can read from or write to a memory location that is outside of the boundary of the buffer. This vulnerability category would cause the system to crash or leak sensitive information.

CWE20. Improper Input Validation: This type of vulnerabilities includes the functions that receive input or data but do not validate or incorrectly validate. If the input or data is not validated, the software would receive unintended input, which leads to altered control flow or arbitrary code execution.

CWE399. Resource Management Errors: This type of vulnerabilities would result from improper management of system resources, such as memory and files, during software execution.

CWE264. Permissions, Privileges, and Access Controls: Vulnerabilities of this type are related to improper access control, such as the management of permissions, privileges, and other security features.

CWE416. Use After Free: The vulnerabilities belonging to this type would reference previously-freed memory, which can cause the corruption of valid data or the execution of arbitrary code.

CWE125. Out-of-bounds Read: This type of vulnerabilities would cause excessive data to be read, leading to a segmentation fault or a buffer overflow.

CWE189. Numeric Errors: Vulnerabilities of this type are related to improper calculation or conversion of numbers, such as incorrect conversion between numeric types and floating-point comparison with an incorrect operator.

CWE362. Concurrent Execution using Shared Resources with Improper Synchronization: This type of vulnerabilities contains the code snippet that can run concurrently with other code, and the code requires temporary, exclusive access to a shared resource. The shared resource can be modified by another code that is operating concurrently.

CWE476. NULL Pointer Dereference: This type of vulnerabilities would cause a crash or exit when the application dereferences a pointer that it expects to be valid but is NULL.

CWE190. Integer Overflow or Wraparound: This type of vulnerabilities would produce an integer overflow or wraparound when an integer value is incremented to a value larger than the original value. The source code becomes security-critical when the value is used to make a security decision or control loop.

B. Experimental Methodology

Baselines: To evaluate the effectiveness of our approach, we compare it with seven state-of-the-art techniques, i.e., VulDeePecker [18], SySeVR [20], DeepWukong [28], VGDetector [61], Devign [10], REVEAL [25], and CodeBERT [24], belonging to three types, i.e., token-based models, graph-based models, and pre-trained models.

- *VulDeePecker* is a token-based model, which first slices the input code snippet based on "key points" (e.g., library/API function calls) in the code snippet and then uses Long Short-Term Memory network (LSTM) [22] to encode the sliced code snippet and perform prediction.
- *SySeVR* is a graph-based model which represents the code snippet as the PDG and uses vulnerability syntax characteristics, such as Library/API function call, array usage, pointer usage, and the arithmetic expression, to slice the PDG. It adopts LSTM [22] to learn the representation of the sliced code and detect the vulnerability.
- *DeepWukong* is a graph-based model. It first constructs the PDG of the code snippet based on both the control flow and data flow information, and slices the code snippet with the statements containing system API calls and arithmetic operators as slicing criteria. Then, it adopts the graph neural

network [62] and Doc2Vec [63] to encode the code snippet and perform vulnerability prediction.

- *VGDetector* utilizes the control flow graph to represent the code snippet. It uses the graph convolutional network [62] and Doc2vec [63] to learn the representation of the control flow graph for vulnerability detection.
- *Devign* is a graph-based model which utilizes the Gated Graph Recurrent network (GGN) [23] to represent the graph that combines the AST, CFG, DFG, and code sequence of the input code snippet for vulnerability detection.
- *REVEAL* is a graph-based model, which combines Devign [10] with resampling techniques [32] and the triplet loss [33] to detect vulnerabilities.
- *CodeBERT* is a pre-trained-based model, which refers to the approach that directly uses CodeBERT [24] to predict whether the input code snippet is vulnerable or not. This model has achieved good performance on multiple software engineering tasks.

Evaluation Metrics: Referring to prior works [9], [10], [24], we adopt three widely-used classification metrics, i.e., Precision, Recall, and F1-Score, for evaluation. $Precision = \frac{\#TP}{\#TP+\#FP}$, $Recall = \frac{\#TP}{\#TP+\#FN}$, and $F1 = \frac{Precision*Recall}{Precision+Recall}$. $\#TP$ represents the number of vulnerable code snippets correctly detected as vulnerabilities. $\#FP$ denotes the number of non-vulnerable code snippets incorrectly detected as vulnerabilities. $\#FN$ is the number of vulnerable code snippets incorrectly predicted as non-vulnerable.

Experiment Environment: We implemented EPVD in Python using PyTorch [64]. The experiments are performed on a machine with 4 NVIDIA GeForce GTX 3090 GPUs and two Intel Xeon Gold 6226R CPUs of 2.90 GHz.

Experiment settings: For the path selection phase, we set the number of the selected paths, i.e., m, as three by default. In this section, we investigate the impacts of different m on the effectiveness of our approach. For the code representation learning phase, we use three filters with a shape of 128 in the CNN. We set the hidden size of the fully connected layer in (7) as 768. During training, AdamW [65] is used as the optimizer, and the best model is selected based on the optimal F1-Score on the validation set.

C. Overall Performance Analysis (RQ1)

We train and evaluate our approach and all the baselines on the merged dataset, and compare their performance in terms of Precision, Recall, and F1-Score. The results are shown in Table III, with the best results marked in bold.

We can see that EPVD outperforms all the baselines in terms of all three metrics by large margins and achieves an F1-Score of 66.69%, which indicates the effectiveness of EPVD. Specifically, it improves over the best-performing baseline, i.e., CodeBERT, by 22.30%, 42.92%, and 32.58% in Precision, Recall, and F1-Score, respectively. Besides, we also have the following findings:

(1) SySeVR is the worst-performing baseline. One possible reason for this is the limited ability of LSTM in modeling program semantics. Another possible reason is that SySeVR adopts specific pre-defined criteria (i.e., library/API function calls, array usage, pointer usage, and arithmetic expressions) to slice the source code, which may result in the loss of vulnerability-related information during slicing. Besides, VulDeePecker also uses similar slicing criteria (i.e., library/API function calls) and also performs poorly. Compared to VulDeePecker and SySeVR, our approach not only adopts a strong code representation model but also guides neural models to focus on small and cohesive code pieces, i.e., the execution paths, which can help neural models better capture vulnerability patterns from code.

TABLE III
PERFORMANCE COMPARISONS OF OUR APPROACH WITH OTHER BASELINES

Approach	Evaluation Metrics		
	Precision	Recall	F1-Score
VulDeePecker	9.16%	6.65%	7.70%
SySeVR	17.26%	2.80%	4.81%
DeepWukong	40.99%	27.96%	33.24%
VGDetector	47.98%	28.58%	35.82%
Devign	33.10%	15.29%	20.92%
REVEAL	24.09%	43.47%	31.00%
CodeBERT	54.40%	46.68%	50.30%
EPVD	**66.53%**	**66.86%**	**66.69%**
Improvement	22.30%	42.92%	32.58%

(2) We can see that CodeBERT performs better than all other baselines, although it neither slices the input code like VulDeePecker nor explicitly extracts and makes use of the structural information of code like other graph-based baselines. This indicates that CodeBERT can effectively capture the syntactic and semantic information related to vulnerabilities from code, justifying our choice of using CodeBERT to encode execution paths. Apart from CodeBERT and EPVD, VGDetector achieves the best performance in terms of precision and F1-Score. One possible reason is that by encoding the control flow graph, VGDetector can effectively capture the execution logic of code, boosting vulnerability detection. Our approach can be regarded as combining the advantages of both CodeBERT and VGDetector because it extracts the execution paths from the control flow graph and encodes the execution paths using CodeBERT. The simple and linear structure and the shorter lengths of the execution paths make it easier for neural networks to capture vulnerability-related information.

(3) For other baselines, although DeepWukong considers both control flow and data flow, its performance is worse than that of VGDetector and EPVD. After analyzing the results, we think the reasons may be that: 1) The slicing criteria used by DeepWukong, i.e., the statements containing system API calls and arithmetic operators, sometimes are not related to specific vulnerabilities. Thus, the slices generated by DeepWukong may lose vulnerability-related information. 2) The neural model in DeepWukong still needs to understand complex code structures, which is not easy. In contrast, first, EPVD tries its best to keep all information in code by using our path selection algorithm to cover as many statements as possible in the decomposed execution paths. In addition, the structure of execution paths is simple and linear, making it easier for neural networks to capture vulnerability-related information from them.

TABLE IV
PERFORMANCE COMPARISON BETWEEN CODEBERT AND EPVD ON THE SHORT CODE TEST SET

Approach	Evaluation Metrics		
	Precision	Recall	F1-Score
CodeBERT	63.06%	32.26%	42.68%
EPVD	**72.96%**	**65.52%**	**69.04%**
Improvement	**15.70%**	**103.10%**	**61.76%**

TABLE V
PERFORMANCE COMPARISON BETWEEN CODEBERT AND EPVD ON THE LONG CODE TEST SET

Approach	Evaluation Metrics		
	Precision	Recall	F1-Score
CodeBERT	51.65%	54.81%	53.18%
EPVD	**63.30%**	**60.32%**	**61.78%**
improvement	**22.56%**	**10.05%**	**16.17%**

To further understand EPVD's better performance, we compare the prediction results of EPVD and the best-performing baseline, i.e., CodeBERT. First, we select the code snippets with less than 400 tokens, which is the input length limit of CodeBERT, from the test set and compare the performance of EPVD and CodeBERT on these code snippets. Table IV shows the results. We can see that EPVD performs better than CodeBERT regarding all metrics. The Recall of EPVD even increases by 103.10%. One possible reason is that EPVD can simplify code structure and help existing neural networks capture vulnerability features. To support this claim, we use the number of *if_statement* to measure the structural complexity of a code snippet and find that EPVD detects 544 more vulnerabilities than CodeBERT, with 302 of them containing three or more *if_statement*. Besides, we also select code snippets with more than 400 tokens in the test set and treat them as long code snippets. We compare the detection performance with CodeBERT. Table V presents the results, where EPVD achieves a better F1-Score than CodeBERT from 53.18% to 61.78%. In detail, we count the vulnerabilities with more than 400 tokens and find that EPVD correctly identifies 265 of them, while CodeBERT only identifies 87 of them. This confirms our claim. In addition, Fig. 4(a) presents a sample in our test set, which is a function from the php/php-src project [66] containing a vulnerability exposed by CVE-2014-3515 [67]. The root cause of this vulnerability is line 47. Since the function is quite long, line 47 will be truncated before the function is processed by CodeBERT to meet the input length limit of CodeBERT. The lack of line 47 would make it difficult for neural models to capture vulnerability features for this function, so CodeBERT fails to detect this vulnerability. For EPVD, it extracts multiple execution paths from the function and regards them as input. One extracted path is presented in Fig. 4(b). We can see that the extracted path is much shorter than the function, and the root cause of the vulnerability locates in line 29 in this path and will not be truncated before being input into neural models. This increases the probability of neural models capturing vulnerability features.

According to our manual inspection, we also find that EPVD can better distinguish vulnerability-related information from

```
1.  SPL_METHOD(Array, unserialize)
2.  {
3.      spl_array_object *intern = (spl_array_object*)zend_object_store_get_object(getThis() TSRMLS_CC);
4.      char *buf;
5.      int buf_len;
6.      const unsigned char *p, *s;
7.      php_unserialize_data_t var_hash;
8.      zval *pmembers, *pflags = NULL;
9.      long flags;
10.     if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, ""s"", &buf, &buf_len) == FAILURE) {
11.         return;}
12.     if (buf_len == 0) {
13.         zend_throw_exception_ex(spl_ce_UnexpectedValueException, 0 TSRMLS_CC,
14.                                 ""Empty serialized string cannot be empty"");
15.         return;}
16.     s = p = (const unsigned char*)buf;
17.     PHP_VAR_UNSERIALIZE_INIT(var_hash);
18.     if (*p!= 'x' || *++p != ':') {
19.         goto outexcept;}
20.     ++p;
21.     ALLOC_INIT_ZVAL(pflags);
22.     if (!php_var_unserialize(&pflags, &p, s + buf_len, &var_hash TSRMLS_CC) || Z_TYPE_P(pflags) != IS_LONG) {
23.         zval_ptr_dtor(&pflags);
24.         goto outexcept;}
25.     --p;
26.     flags = Z_LVAL_P(pflags);
27.     zval_ptr_dtor(&pflags);
28.     if (*p != ';') {
29.         goto outexcept;}
30.     ++p;
31.     if (*p!='m') {
32.     if (*p!='a' && *p!='O' && *p!='C' && *p!='r') {
33.         goto outexcept;}
34.     intern->ar_flags &= ~SPL_ARRAY_CLONE_MASK;
35.     intern->ar_flags |= flags & SPL_ARRAY_CLONE_MASK;
36.     zval_ptr_dtor(&intern->array);
37.     ALLOC_INIT_ZVAL(intern->array);
38.     if (!php_var_unserialize(&intern->array, &p, s + buf_len, &var_hash TSRMLS_CC)) {
39.         goto outexcept;}}
40.     if (*p != ';') {
41.         goto outexcept;}
42.     ++p;
43.     if (*p!= 'm' || *++p != ':') {
44.         goto outexcept;}
45.     ++p;
46.     ALLOC_INIT_ZVAL(pmembers);
47.     if (!php_var_unserialize(&pmembers, &p, s + buf_len, &var_hash TSRMLS_CC)) {
48.         zval_ptr_dtor(&pmembers);
49.         goto outexcept; }
50.     if (!intern->std.properties) {
51.     rebuild_object_properties(&intern->std); }
52.     zend_hash_copy(intern->std.properties, Z_ARRVAL_P(pmembers), (copy_ctor_func_t) zval_add_ref, (void *) NULL,
53.                    sizeof(zval *));
54.     zval_ptr_dtor(&pmembers);
55.     PHP_VAR_UNSERIALIZE_DESTROY(var_hash);
56.         return;
... ......
61. }
```

(a)

```
1.  SPL_METHOD(Array, unserialize)
2.  {
3.      spl_array_object *intern = (spl_array_object*)zend_object_store_get_object(getThis() TSRMLS_CC);
4.      char *buf;
5.      int buf_len;
6.      const unsigned char *p, *s;
7.      php_unserialize_data_t var_hash;
8.      zval *pmembers, *pflags = NULL;
9.      long flags;
10.     if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, ""s"", &buf, &buf_len) == FAILURE) {}
11.     if (buf_len == 0) {}
12.     s = p = (const unsigned char*)buf;
13.     PHP_VAR_UNSERIALIZE_INIT(var_hash);
14.     if (*p!= 'x' || *++p != ':') {}
15.     ++p;
16.     ALLOC_INIT_ZVAL(pflags);
17.     if (!php_var_unserialize(&pflags, &p, s + buf_len, &var_hash TSRMLS_CC) || Z_TYPE_P(pflags) != IS_LONG) {}
18.     --p;
19.     flags = Z_LVAL_P(pflags);
20.     zval_ptr_dtor(&pflags);
21.     if (*p != ';') {}
22.     ++p;
23.     if (*p!='m') {}
24.     if (*p != ';') {}
25.     ++p;
26.     if (*p!= 'm' || *++p != ':') {}
27.     ++p;
28.     ALLOC_INIT_ZVAL(pmembers);
29.     if (!php_var_unserialize(&pmembers, &p, s + buf_len, &var_hash TSRMLS_CC)) {}
30.     if (!intern->std.properties) {}
31.     zend_hash_copy(intern->std.properties, Z_ARRVAL_P(pmembers), (copy_ctor_func_t) zval_add_ref, (void *) NULL,
32.                    sizeof(zval *));
33.     zval_ptr_dtor(&pmembers);
34.     PHP_VAR_UNSERIALIZE_DESTROY(var_hash);
35.     return;
36. }
```

(b)

Fig. 4. A long function from the test set. (The green shaded statement contains a vulnerability.).

information unrelated to vulnerabilities than the baselines. Fig. 5 presents such an example, where the three functions are collected from the php-src project [66]. The function in Fig. 5(a) is a test sample, which contains an integer overflows vulnerability (line 41 and line 42) and is exposed by CVE-2016-3078 [68]. The functions in Fig. 5(b) and (c) are training samples. We can see that: (1) there are many similar statements between the functions in Fig. 5(a) and (b), highlighted by the gray rectangle. Although our approach has been on the function in Fig. 5(b),

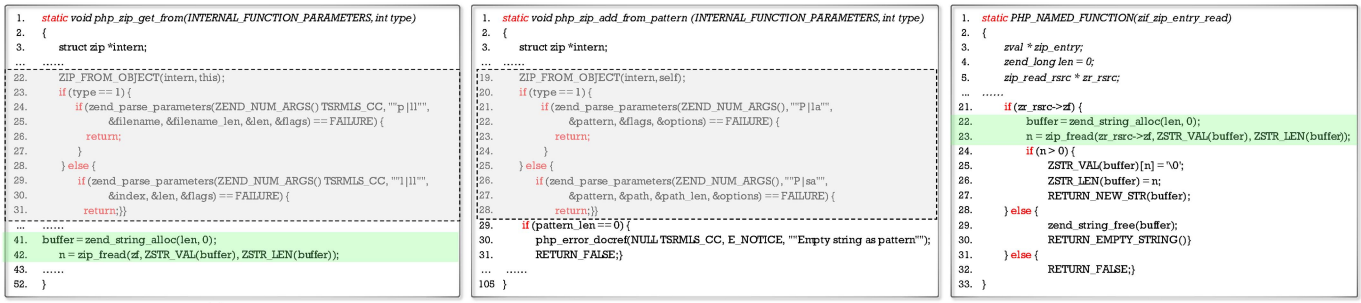


Fig. 5. The test sample. (The green shaded statement contains a vulnerability.).

TABLE VI
PERFORMANCE COMPARISONS OF DIFFERENT PATH SELECTION ALGORITHMS

Path Selection Methods	Coverage Rate	Evaluation Metrics		
		Precision	Recall	F1-Score
EPVD	0.9281	66.53%	66.86%	**66.69%**
EPVD-2	0.9414	65.20%	**67.11%**	66.14%
EPVD-3	0.9423	**66.65%**	64.48%	65.55%

TABLE VII
PERFORMANCE COMPARISONS OF DIFFERENT NUMBER OF PATHS

The number of paths	Evaluation Metrics		
	Precision	Recall	F1-Score
EPVD-p2	63.14%	**70.59%**	66.65%
EPVD	**66.53%**	66.86%	66.69%
EPVD-p4	66.05%	62.85%	64.41%
EPVD-p5	63.67%	70.04%	**66.71%**

which is non-vulnerable, it still correctly predicts the function in Fig. 5(a) as vulnerable. It means that our approach successfully ignores the information unrelated to vulnerabilities. (2) Fig. 5(c) is also an integer overflows vulnerability exposed by CVE-2016-3078 [68], which is similar to the one in Fig. 5(a). Although the two functions are different except for the vulnerability-related statements, our model still regards them are similar in terms of vulnerabilities without being disturbed by other irrelevant information.

D. Detail Analysis (RQ2)

In this subsection, we would like to explore the impact of different model choices on the effectiveness of our approach, including different path selection methods, different numbers of selected paths, and different path fusion methods.

1) The Effect of Different Path Selection Methods: The key idea of EPVD is to decompose the long code into multiple execution paths and learn to capture vulnerability patterns based on the representations of such paths. In order to make the trade-off between code coverage and path lengths, we propose a path selection method that first selects two diverse and short paths and then chooses the path that can cover the most uncovered statements as the last path.

To explore the impact of different path selection methods on EPVD's performance, we compare EPVD with its two variants: (1) *EPVD-2*, which first selects one shortest execution path from the CFG and then selects two paths that can cover the most uncovered statements in turn. (2) *EPVD-3*, which chooses three paths that can cover the most uncovered statements in turn. EPVD and the two variants are identical except for the used path selection methods. Comparing EPVD with the two variants can help us understand the importance of the path selection method and the feasibility of path decomposition.

The results of our comparisons are presented in Table VI. The average coverage rate column is the ratio of code lines in the source code that can be covered in the three paths. We can see that the more long paths a model uses, the higher the average coverage rate it can achieve, which is intuitive. Nevertheless, the differences among the three path selection methods in terms of average coverage rate are small ($<= 1.53\%$). EPVD outperforms EPVD-2 and EPVD-3 in terms of F1-Score by 0.55 and 1.14 points, respectively, although its coverage rate is slightly lower than EPVD-2 and EPVD-3. Moreover, it takes 2.11% and 2.26% more time to train EPVD-2 and EPVD-3 than EPVD, respectively. Therefore, we believe that choosing two short and diverse paths and one long path that maximizes the code coverage is a beneficial decision.

2) The Effect of Different Numbers of Selected Paths: In EPVD, we decompose a code snippet into three execution paths. Although it is impractical to select all execution paths, as discussed in Section IV-B, there are still many options for the number of selected paths. Therefore, we would like to evaluate the effect of different numbers of selected paths on EPVD's performance. We build three variants of our approach, i.e., EPVD-p2, EPVD-p4, and EPVD-p5, which are the same as EPVD except for selecting 2, 4, and 5 executions from the CFG, respectively. For each variant with m paths, we select $m-1$ short and diverse paths with one long and high-coverage path using our path selection method. We compare their performance with EPVD.

Table VII presents the results. We can see that EPVD achieves the second-best performance in terms of F1-Score, and the performance difference between EPVD and EPVD-p5 is very small (only 0.0.3%). However, compared to EPVD, the training time of EPVD-p5 increases by 37.57%. Also, EPVD is the one with the best Precision. Considering the trade-off between

TABLE VIII
PERFORMANCE COMPARISONS OF DIFFERENT PATH FUSION METHODS

The path fusion methods	Evaluation Metrics		
	Precision	Recall	F1-Score
EPVD-mlp	64.14%	**68.33%**	66.17%
EPVD-blstm	**66.95%**	65.61%	66.27%
EPVD	66.53%	66.86%	**66.69%**

TABLE IX
EFFECTS OF THE GREEDY-BASED PATH SELECTION STRATEGY

Approach	Evaluation Metrics		
	Precision	Recall	F1-Score
EPVD-multi-encoder	**67.77%**	62.72%	65.15%
CodeBERT-slicing	52.25%	31.59%	39.37%
EPVD	66.53%	**66.86%**	**66.69%**

effectiveness and efficiency, we believe it is reasonable to set the number of selected paths as three.

3) The Effect of Different Path Fusion Methods: In EPVD, we adopt a CNN to learn inter-path attentions and fuse three execution paths. To explore the impact of different path fusion methods on the performance of our approach, we compare EPVD with two of its variants, i.e., *EPVD-mlp* and *EPVD-blstm*. EPVD-mlp first concatenates the representations of the selected paths horizontally and then inputs the concatenated vector into an MLP to obtain the final code representation. EPVD-blstm adopts a BiLSTM [22] to fuse the path representations. Following previous work [18], we use a three-layer BiLSTM. Other hyper-parameters of the two variants are the same as EPVD. To check whether the performance differences between EPVD and the two variants are statistically significant, we refer to previous work [69] and use the out-of-sample bootstrap validation [70], [71] to generate a bootstrap sample. We repeat this process 50 times and apply the Wilcoxon signed-rank tests [72] at the 95% confidence level for the statistical analysis.

The results are presented in Table VIII. We can see that our model performs better than the two variants by more than 0.42 points in terms of the F1-Score. The improvements achieved by EPVD over the two variants, i.e., EPVD-mlp and EPVD-blstm, are statistically significant. The p-values are both less than 0.001. The results confirm that CNN can better capture inter-path attention among multiple execution paths.

4) The Effect of Greedy-Based Path Selection Strategy: In EPVD, we adopt the greedy-based strategy to decompose the syntax-based CFG of the code snippet. To understand the effect of the greedy-based path selection strategy, we construct two variants of EPVD, namely *EPVD-multi-encoder* and *CodeBERT-slicing*. EPVD-multi-encoder adopts three distinct CodeBERT models to encode the three extracted execution paths, respectively. Then, we utilize three MLP classifiers to calculate three probabilities of these extracted execution paths. Since a code snippet is vulnerable if any of its execution paths is vulnerable, we choose the highest of the three probabilities as the final output probability and calculate the evaluation metrics only based on this probability. Specifically, if the final probability is greater than 0.5, the input code snippet is predicted as vulnerable. Otherwise, it is considered as non-vulnerable. CodeBERT-slicing uses the slicing method of DeepWukong but adopts CodeBERT to encode the code slices.

The results of our comparisons are presented in Table IX. We can see that EPVD performs better than EPVD-multi-encoder in terms of Recall and F1-Score, which indicates that fusing the selected paths for vulnerability detection is more effective than detecting vulnerability from each of them, respectively. One possible reason is that combing multiple paths could provide additional vulnerability-related information. EPVD outperforms CodeBERT-slicing in terms of all metrics by large margins, which means slicing with specific criteria is less effective than our greedy-based path selection strategy. This may be because it is hard, if not impossible, to cover all vulnerability-related slicing criteria, and slicing with pre-defined criteria may cause the loss of important information for some vulnerabilities.

E. Detection Results Across Different Types of Vulnerabilities (RQ3)

As shown in Table X, the performance of EPVD varies when detecting different types of vulnerabilities because different kinds of vulnerabilities exhibit different behaviors. We can also see that: (1) EPVD performs best on 8 out of 10 types, and nearly all of them are both related to the control flow and data flow information, such as CWE20 and CWE264. This is because EPVD not only explicitly decomposes control flow information but also implicitly includes data flow information in each path. As every decomposed execution path has simpler logic, linear structures, and shorter length, CodeBERT can better capture the data flow information of the source code, which help EPVD detect vulnerabilities related to the data flow. (2) EPVD does not achieve the best performance in detecting CWE119 and CWE476 vulnerabilities. By contrast, VGDetector and DeepWukong achieve the best F1-scores, 60.69% and 59.45%, respectively. We manually examine the results and find that the lengths of CWE119 vulnerabilities are usually within the input limit of neural models, which may limit the advantages of the proposed model. Besides, CWE476 vulnerabilities are highly related to data flow information because they usually occur when pointers point to invalid memory addresses and refer to them, leading to unforeseeable errors and software system crashes. EPVD does not explicitly consider data flow information, which may make it harder for EPVD than DeepWukong to capture features related to CWE476. (3) SySeVR cannot detect the vulnerabilities belonging to CWE264. The corresponding entry in Table X is marked as "N/A". The reason may be that SySeVR adopts BiLSTM to encode the sliced source code, leading to the limited ability to capture semantic information of code. Besides, VGDetector and CodeBERT do not detect any CWE190 vulnerabilities. The main reason for the result is that the number of datasets in CWE190 is small, which may only contain limited vulnerability patterns. Moreover, VGDetector does not detect any vulnerabilities of CWE476. One possible reason is that VGDetector only considers control flow information, while CWE476 is tightly related to data flow information.

TABLE X
PERFORMANCE COMPARISONS OF DIFFERENT EPVD WITH OTHER BASELINES ACROSS DIFFERENT TYPES OF VULNERABILITIES

The type of vulnerability	Approach							
	VulDeePecker	Devign	SySeVR	REVEAL	DeepWukong	VGDetector	CodeBERT	EPVD
CWE119	4.47%	6.99%	18.69%	8.63%	11.59%	**60.69%**	12.01%	18.45%
CWE20	19.69%	15.48%	31.26%	12.05%	21.38%	69.99%	57.74%	**80.33%**
CWE399	23.59%	62.22%	7.19%	64.25%	24.97%	61.56%	97.26%	**97.87%**
CWE264	19.29%	28.24%	N/A	28.13%	35.19%	51.57%	46.87%	**79.21%**
CWE416	19.28%	62.50%	9.44%	60.61%	25.24%	30.04%	73.17%	**77.50%**
CWE125	18.18%	27.85%	2.86%	35.00%	30.22%	45.08%	29.33%	**79.57%**
CWE189	10.31%	22.43%	56.85%	28.00%	21.32%	6.97%	46.91%	**79.31%**
CWE362	18.98%	55.32%	2.75%	53.57%	26.99%	23.04%	96.15%	**98.18%**
CWE476	19.05%	25.81%	6.39%	29.03%	**59.45%**	N/A	25.00%	28.57%
CWE190	5.13%	17.65%	17.96%	6.67%	25.98%	N/A	N/A	**33.33%**

TABLE XI
PERFORMANCE COMPARISONS OF DATA SENSITIVITY ANALYSIS

The ratio of the training dataset	Evaluation Metrics		
	Precision	Recall	F1-Score
80%	66.53%	**66.86%**	**66.69%**
70%	65.42%	65.94%	65.68%
60%	**68.11%**	61.13%	64.43%

TABLE XII
PERFORMANCE COMPARISONS OF EPVD WITH OTHER BASELINES ON REAL-WORLD APPLICATIONS IN TERMS OF F1-SCORE

Approach\Dataset	Lua	Redis	Mixed Dataset
VulDeePecker	N/A	19.35%	12.03%
SySeVR	N/A	N/A	3.00%
DeepWukong	N/A	2.93%	1.52%
VGDetector	N/A	5.71%	10.81%
Devign	5.56%	5.13%	8.11%
REVEAl	5.77%	14.29%	12.90%
CodeBERT	N/A	7.69%	12.50%
EPVD	**6.90%**	**23.26%**	**15.58%**

TABLE XIII
TIME COSTS COMPARING EPVD AND CODEBERT ON THE MERGED DATASET

Model	Training	Detection
CodeBERT	0.53 hours/epoch	0.02 hours
EPVD	1.25 hours/epoch	0.12 hours

F. Sensitivity Analysis on Training Data (RQ4)

In this part, we aim to understand the effects of different amounts of training data on the effectiveness of our proposed model. In other research questions, we use 80%, 10%, and 10% of the merged dataset for training, validation, and testing, respectively. In this RQ, we use the same validation and test sets but train our approach with 70% and 60% of the merged dataset as the training sets, respectively. Please note that the two smaller training sets are randomly sampled from the original training set.

As shown in Table XI, the performance of our approach in terms of F1-Score declines when the training set size decreases, which is intuitive. Specifically, the F1-Score drops from 66.69% to 65.68% and 64.43% when using 70% and 60% of the merged dataset for training, respectively. However, even using less training data, our approach still outperforms all the baselines by at least 28.08% (64.43% v.s. 50.30%). This indicates that our approach can consistently perform well with different amounts of training data.

G. Effectiveness of EPVD in Real-World Applications (RQ5)

To investigate the effectiveness of EPVD in real-world applications, we follow previous work [28] and conduct experiments using the datasets collected from two open-source projects, i.e., the Redis dataset, the Lua dataset, and the mixed dataset. For each dataset, we sort the samples in it according to the ascending order of the commit submission time and use the top 80% of the samples for training, 10% for evaluation, and 10% for detection. We compare EPVD with the baselines introduced in Section V-C.

Table XII shows the performance of EPVD on real-world projects compared with the baselines. EPVD outperforms all the baseline models by at least 19.58%, 20.21%, and 20.78% in terms of F1-Score on the three datasets, respectively. According to our experiments, we find some models cannot detect any vulnerability in the Lua and/or Redis datasets. We mark the corresponding entries in Table XII as "N/A". On the Lua dataset, the F1-Scores of most baselines are "N/A". The possible reason is that the small number of vulnerabilities and the data imbalance in the Lua dataset hinder the baseline models from learning vulnerability patterns. These results suggest that EPVD is also more effective than the baselines on real-world applications.

H. Efficiency of EPVD (RQ6)

Table XIII lists the time costs of the EPVD and CodeBERT in two phases, i.e., model training and vulnerability detection. The results show that EPVD requires more time than CodeBERT in the two phases. This is expected because EPVD uses CodeBERT to encode three sequences for a code snippet and contains an additional CNN for path fusion. Specifically, it takes about 1.25 hours to train EPVD for one epoch on the merged dataset. Considering that we only need to train EPVD once, and the training is conducted offline in practice, we argue that the training cost is affordable. It takes 0.02 seconds on average for EPVD to predict whether a code snippet is vulnerable, which means the efficiency of EPVD is sufficient for practical use. Considering that EPVD improves CodeBERT by large margins in terms of the detection performance, as shown in Table III, we argue that the additional time cost of EPVD is worthwhile. We will further investigate how to improve the efficiency of EPVD in future work.

VI. DISCUSSIONS

In this section, we discuss the situations where EPVD may fail, the quality and code coverage of the selected paths, and the threats to the validity of this work.

A. Where Does Our Approach Fail

By randomly selecting and manually analyzing a number of samples where EPVD fails to detect vulnerabilities, we summarize several situations where it is difficult for EPVD to handle. First, it is often difficult for EPVD to detect new vulnerabilities which are not similar to any vulnerable sample in the training set. Almost all learning-based vulnerability detection methods suffer from this situation.

Second, EPVD cannot detect cross-function vulnerabilities. For example, if the detected function contains a function call to a vulnerable function, it is difficult for our model to detect such a vulnerability. It may be challenging for EPVD to detect this kind of vulnerability. We would try to address this limitation in our future work.

Third, since EPVD does not explicitly extract data flow information, it may not achieve superior performance in detecting the vulnerabilities tightly related to data flows. However, as shown in Section V-E, EPVD also outperforms all the baselines on some vulnerability types that require considering data flow information. We think this is because the decomposed execution paths implicitly contain data flow information, which can be captured and encoded by CodeBERT. Therefore, we argue that for the vulnerabilities related to data flows, EPVD can also be helpful. We leave improving EPVD to explicitly consider the data flow information as future work.

B. The Quality and Code Coverage of the Selected Paths

Since we build the syntax-based CFG only based on AST and select execution paths from the syntax-based CFG, some execution paths selected by our approach may be dead paths, i.e., paths that can never be executed during runtime. To obtain accurate execution paths of a code snippet, we need to compile the code and obtain enough runtime information, which is expensive and not very practical. Chen et al. [3] proposed a novel vulnerability detection model, named ContraFlow, which adopts the static analyzer SVF [73] to extract a set of value-flow paths of the source code and utilizes the self-supervised contrastive learning [74] to encode value flow paths. Then, ContraFlow used active learning [75] to select the most representative paths and adopted soft attention [76] to perform path-sensitive analysis and select the most representative paths. Different from ContraFlow, EPVD leverages a greedy-based path selection method to select representative execution paths from the syntax-based CFG and utilizes CNN to fuse the selected paths.

Another thing worth mentioning is that the selected execution paths of a code snippet may not cover all of its statements. This is expected since we can only select finite and limited execution paths to make the trade-off between performance and efficiency. Although we can iteratively select execution paths from the syntax-based CFG until we cover all the statements in the code snippet, we argue that this may not significantly improve the performance but would greatly increase the training and inference time of our approach. For example, as shown in Section V-D, by increasing the number of the select paths, we only slightly improve the F1-Score of our approach by 0.02% but increase the training time by 37.47%.

C. Threats to Validity

One threat to the validity of this work is that our dataset is built only from C/C++ projects, which may not represent all programming languages. However, a majority of CVEs are found in C/C++ projects, and our approach is generic and can be extended to other programming languages.

Another threat to validity is that we are unable to perform extensive hyper-parameter optimizations on our model due to hardware limitations. It could be the case that some of the proposed model or baselines are outlined in Table III, VI, VII, and IX could be heavily impacted by certain hyper-parameters, e.g., learning rate and batch size. This is a common issue of the work using deep learning models and affects nearly all similar types of experiments. We mitigate the impact of this issue by being consistent with the hyper-parameter values in CodeBERT. We also take great care when reproducing our baselines to ensure the experimental setups are reasonable and match as closely to their descriptions in the corresponding papers and replication packages as possible.

VII. Conclusion and Future Work

In this work, we present a novel approach that learns to detect vulnerabilities by selecting, encoding, and fusing multiple syntax-based execution paths of the input code snippet. Compared to existing learning-based vulnerability detection approaches, by decomposing code into multiple execution paths, our approach can better distinguish vulnerability-related information from the information unrelated to vulnerabilities in the code snippet, better handle the long vulnerable code of which the vulnerability-related information locates at the tail, and effective capture vulnerability features from code. We compare our approach with the state-of-the-art approaches on over 231 K functions collected from three high-quality datasets. Experimental results show that our approach significantly outperforms all the baselines by large margins. To facilitate future research, we also make our code and artifacts publicly available at Zenodo [27].

In future work, we plan to adopt other pre-trained code models, such as SPT-Code [39] and GraphCodeBERT [43], in our approach. We also plan to apply our approach to other downstream tasks requiring encoding source code, such as code clone detection and code summarization. Besides, we plan to improve EPVD's effectiveness by explicitly considering other structural information in code, such as data flow and value flow information.

References

[1] M. Fu and C. Tantithamthavorn, "LineVul: A transformer-based line-level vulnerability prediction," in *Proc. IEEE/ACM 19th Int. Conf. Mining Softw. Repositories*, Pittsburgh, PA, USA, May 23/24, 2022, pp. 608–620.
[2] C. Thapa, S. I. Jang, M. E. Ahmed, S. Camtepe, J. Pieprzyk, and S. Nepal, "Transformer-based language models for software vulnerability detection: Performance, model's security and platforms," 2022, *arXiv:2204.03214*.
[3] X. Cheng, G. Zhang, H. Wang, and Y. Sui, "Path-sensitive code embedding via contrastive learning for software vulnerability detection," in *Proc. 31st ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, S. Ryu and Y. Smaragdakis, Eds., 2022, pp. 519–531.
[4] Proxylogon flaw, 2022. [Online]. Available: https://proxylogon.com/

[5] J. Jang-Jaccard and S. Nepal, "A survey of emerging threats in cybersecurity," *J. Comput. Syst. Sci.*, vol. 80, no. 5, pp. 973–993, 2014.

[6] A. Johnson, K. Dempsey, R. Ross, S. Gupta, and D. Bailey, "Guide for security-focused configuration management of information systems," *J. Dairy Sci.*, vol. 800, no. 128, 2011, Art. no. 16.

[7] Y. Wu, D. Zou, S. Dou, W. Yang, D. Xu, and H. Jin, "VulCNN: An image-inspired scalable vulnerability detection system," in *Proc. IEEE/ACM 44th Int. Conf. Softw. Eng.*, Pittsburgh, PA, USA, May 25–27, 2022, pp. 2365–2376.

[8] S. Cao, X. Sun, L. Bo, R. Wu, B. Li, and C. Tao, "MVD: Memory-related vulnerability detection based on flow-sensitive graph neural networks," in *Proc. IEEE/ACM 44th Int. Conf. Softw. Eng.*, Pittsburgh, PA, USA, May 25–27, 2022, pp. 1456–1468.

[9] Y. Li, S. Wang, and T. N. Nguyen, "Vulnerability detection with fine-grained interpretations," in *Proc. 29th ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, Athens, Greece, Aug. 23–28, 2021, pp. 292–303.

[10] Y. Zhou, S. Liu, J. K. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, Vancouver, BC, Canada, Dec. 08–14, 2019, pp. 10 197–10 207.

[11] Checkmarx, 2022. [Online]. Available: https://www.checkmarx.com/

[12] Flawfinder, 2022. [Online]. Available: http://www.dwheeler.com/flawfinde/r

[13] Rough audit tool for security, 2022. [Online]. Available: https://code.google.com/archive/p/rough-auditing-tool-for-security/

[14] F. Yamaguchi, "Pattern-based vulnerability discovery," PhD dissertation, Univ. Göttingen, Göttingen, Germany, 2015.

[15] F. Yamaguchi, "Pattern-based methods for vulnerability discovery," *Inf. Technol.*, vol. 59, no. 2, 2017, Art. no. 101.

[16] X. Duan et al., "VulSniper: Focus your attention to shoot fine-grained vulnerabilities," in *Proc. 28th Int. Joint Conf. Artif. Intell.*, Macao, China, Aug. 10–16, 2019, pp. 4665–4671. [Online]. Available: ijcai.org

[17] R. L. Russell et al., "Automated vulnerability detection in source code using deep representation learning," in *Proc. IEEE 17th Int. Conf. Mach. Learn. Appl.*, Orlando, FL, USA, Dec. 17–20, 2018, pp. 757–762.

[18] Z. Li et al., "VulDeePecker: A deep learning-based system for vulnerability detection," in *Proc. 25th Annu. Netw. Distrib. Syst. Secur. Symp.*, San Diego, CA, USA, Feb. 18–21, 2018, pp. 1–15.

[19] H. K. Dam, T. Tran, T. Pham, S. W. Ng, J. Grundy, and A. Ghose, "Automatic feature learning for vulnerability prediction," 2017, *arXiv: 1708.02368*.

[20] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "SySeVR: A framework for using deep learning to detect software vulnerabilities," *IEEE Trans. Dependable Secure Comput.*, vol. 19, no. 4, pp. 2244–2258, Jul./Aug. 2022.

[21] G. Lin, J. Zhang, W. Luo, L. Pan, and Y. Xiang, "POSTER: Vulnerability discovery with function representation learning from unlabeled projects," in *Proc. SIGSAC Conf. Comput. Commun. Secur.*, Dallas, TX, USA, 2017, pp. 2539–2541.

[22] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.

[23] Y. Li, D. Tarlow, M. Brockschmidt, and R. S. Zemel, "Gated graph sequence neural networks," in *Proc. 4th Int. Conf. Learn. Representations*, San Juan, Puerto Rico, May 02–04, 2016, pp. 1–20.

[24] Z. Feng et al., "CodeBERT: A pre-trained model for programming and natural languages," in *Proc. Findings Assoc. Comput. Linguistics: EMNLP*, Nov. 16–20, 2020, pp. 1536–1547.

[25] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet?," *IEEE Trans. Softw. Eng.*, vol. 48, no. 9, pp. 3280–3296, Sep. 2022.

[26] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "A C/C++ code vulnerability dataset with code changes and CVE summaries," in *Proc. 17th Int. Conf. Mining Softw. Repositories*, Seoul, Republic of Korea, 2020, pp. 508–512.

[27] Zenodo, "Online replication package," 2022. [Online]. Available: https://doi.org/10.5281/zenodo.7123322

[28] X. Cheng, H. Wang, J. Hua, G. Xu, and Y. Sui, "DeepWukong: Statically detecting software vulnerabilities using deep graph neural network," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 3, pp. 1–33, 2021.

[29] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE Trans. Signal Process.*, vol. 45, no. 11, pp. 2673–2681, Nov. 1997.

[30] W. Zheng, Y. Jiang, and X. Su, "Vu1SPG: Vulnerability detection based on slice property graph representation learning," in *Proc. IEEE 32nd Int. Symp. Softw. Rel. Eng.*, 2021, pp. 457–467.

[31] V. Nguyen, D. Q. Nguyen, V. Nguyen, T. Le, Q. H. Tran, and D. Phung, "ReGVD: Revisiting graph neural networks for vulnerability detection," in *Proc. IEEE/ACM 44th Int. Conf. Softw. Eng.*, Pittsburgh, PA, USA, May 22–24, 2022, pp. 178–182.

[32] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Synthetic minority over-sampling technique," *J. Artif. Intell. Res.*, vol. 16, pp. 321–357, 2002.

[33] C. Mao, Z. Zhong, J. Yang, C. Vondrick, and B. Ray, "Metric learning for adversarial robustness," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, Vancouver, BC, Canada, Dec. 08–14, 2019, pp. 478–489.

[34] J. Pennington, R. Socher, and C. D. Manning, "GloVe: Global vectors for word representation," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, 2014, pp. 1532–1543.

[35] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," 2016, *arXiv:1609.02907*.

[36] X. Cheng, X. Nie, N. Li, H. Wang, Z. Zheng, and Y. Sui, "How about bug-triggering paths?-Understanding and characterizing learning-based vulnerability detectors," *IEEE Trans. Dependable Secure Comput.*, early access, Jul. 19, 2022, doi: 10.1109/TDSC.2022.3192419.

[37] Y. Zheng et al., "D2A: A dataset built for AI-based vulnerability detection methods using differential analysis," in *Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng.: Softw. Eng. Pract.*, 2021, pp. 111–120.

[38] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, "Learning and evaluating contextual embedding of source code," in *Proc. 37th Int. Conf. Mach. Learn.*, 2020, pp. 5110–5121.

[39] C. Niu, C. Li, V. Ng, J. Ge, L. Huang, and B. Luo, "SPT-code: Sequence-to-sequence pre-training for learning source code representations," 2022, *arXiv:2201.01549*.

[40] J. Lin, Y. Liu, Q. Zeng, M. Jiang, and J. Cleland-Huang, "Traceability transformed: Generating more accurate links with pre-trained BERT models," in *Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng.*, Madrid, Spain, May 22–30, 2021, pp. 324–335.

[41] J. Bai et al., "Syntax-BERT: Improving pre-trained transformers with syntax trees," in *Proc. 16th Conf. Eur. Chapter Assoc. Comput. Linguistics*, 2021, pp. 3011–3020.

[42] H. Hanif and S. Maffeis, "VulBERTa: Simplified source code pre-training for vulnerability detection," 2022, *arXiv:2205.12424*.

[43] D. Guo et al., "GraphCodeBERT: Pre-training code representations with data flow," in *Proc. 9th Int. Conf. Learn. Representations*, Austria, May 03–07, 2021. [Online]. Available: OpenReview.net

[44] D. Hin, A. Kan, H. Chen, and M. A. Babar, "LineVD: Statement-level vulnerability detection using graph neural networks," in *Proc. IEEE/ACM 19th Int. Conf. Mining Softw. Repositories*, Pittsburgh, PA, USA, May 23–24, 2022, pp. 596–607.

[45] Torvalds, 2023. [Online]. Available: https://github.com/torvalds/linux

[46] Cve-2018–12633. [Online]. Available: https://www.cvedetails.com/cve/CVE-2018--12633/

[47] Mac-telnet, 2022. [Online]. Available: https://github.com/haakonnessjoen/MAC-Telnet/

[48] Cve-2016–7115, 2016. [Online]. Available: https://www.cvedetails.com/cve/CVE-2016--7115/

[49] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Proc. IEEE Symp. Secur. Privacy*, Berkeley, CA, USA, May 18–21, 2014, pp. 590–604.

[50] Tree-sitter, 2023. [Online]. Available: https://tree-sitter.github.io/tree-sitter/

[51] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," in *Proc. ACM Prog. Lang.*, vol. 3, no. POPL, pp. 40:1–40:29, 2019.

[52] A. Vaswani et al., "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst. 30: Annu. Conf. Neural Inf. Process. Syst.*, Long Beach, CA, USA, Dec. 4–9, 2017, pp. 5998–6008.

[53] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," in *Proc. 54th Annu. Meeting Assoc. Comput. Linguistics*, Berlin, Germany, Aug. 7–12, 2016, pp. 1715–1725.

[54] A. P. Parikh, O. Täckström, D. Das, and J. Uszkoreit, "A decomposable attention model for natural language inference," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, Austin, Texas, USA, Nov. 1–4, 2016, pp. 2249–2255.

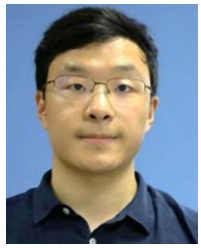
[55] L. J. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," 2016, *arXiv:1607.06450*.

[56] Y. Kim, "Convolutional neural networks for sentence classification," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, Doha, Qatar, Oct. 25–29, 2014, pp. 1746–1751.

[57] Qemu, 2023. [Online]. Available: https://www.qemu.org/

[58] Ffmpeg, 2023. [Online]. Available: https://ffmpeg.org/
[59] "Redis project," 2023. [Online]. Available: https://github.com/redis/redis
[60] "Lua project," 2023. [Online]. Available: https://github.com/lua/lua
[61] X. Cheng et al., "Static detection of control-flow-related vulnerabilities using graph embedding," in *Proc. IEEE 24th Int. Conf. Eng. Complex Comput. Syst.*, Guangzhou, China, Nov. 10–13, 2019, pp. 41–50.
[62] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *Proc. 5th Int. Conf. Learn. Representations*, Toulon, France, Apr. 24–26, 2017, pp. 1–14.
[63] Q. V. Le and T. Mikolov, "Distributed representations of sentences and documents," in *Proc. 31th Int. Conf. Mach. Learn.*, ser. JMLR Workshop and Conference Proceedings, Beijing, China, Jun. 21–26 2014, pp. 1188–1196.
[64] "Pytorch," 2021. [Online]. Available: https://pytorch.org/
[65] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," in *Proc. 7th Int. Conf. Learn. Representations*, New Orleans, LA, USA, May 6–9, 2019.
[66] Php-src, 2023. [Online]. Available: https://github.com/php/php-src
[67] Cve-2014–3515, 2014. [Online]. Available: https://www.cvedetails.com/cve/CVE-2014--3515/
[68] Cve-2016–3078, 2016. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2016--3078, [n.d.]
[69] J. Jiarpakdee, C. Tantithamthavorn, and A. E. Hassan, "The impact of correlated metrics on the interpretation of defect models," *IEEE Trans. Softw. Eng.*, vol. 47, no. 2, pp. 320–331, Feb. 2021.
[70] B. Efron and R. J. Tibshirani, *An Introduction to the Bootstrap*. New York, NY, USA: CRC Press, 1993.
[71] T. Hastie, R. Tibshirani, and J. Friedman, "The elements of statistical learning. 2001," *J. Roy. Statist. Soc.*, vol. 167, no. 1, pp. 192–192, 2004.
[72] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics*, vol. 1, no. 6, 1992, pp. 196–202.
[73] Y. Sui and J. Xue, "SVF: Interprocedural static value-flow analysis in LLVM," in *Proc. 25th Int. Conf. Compiler Construction*, A. Zaks and M. V. Hermenegildo, Eds., Barcelona, Spain, Mar. 12–18, 2016, pp. 265–266.
[74] T. Gao, X. Yao, and D. Chen, "SimCSE: Simple contrastive learning of sentence embeddings," 2021, *arXiv:2104.08821*.
[75] T. Chen, S. Kornblith, M. Norouzi, and G. Hinton, "A simple framework for contrastive learning of visual representations," in *Proc. Int. Conf. Mach. Learn.*, 2020, pp. 1597–1607.
[76] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," in *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 1–29, 2019.

Junwei Zhang received the BS and MS degrees in software engineering from Chongqing University, Chongqing, China, in 2018 and 2021, respectively. He is currently working toward the PhD degree with the College of Computer Science and Technology, Zhejiang University, China. His research interests include deep learning for software engineering and mining software repositories.



Zhongxin Liu received the PhD degree in computer science and technology from Zhejiang University, Zhejiang, China, in 2021. He is currently a distinguished research fellow with the College of Computer Science and Technology, Zhejiang University, China. His research interests include AI for software engineering and mining software repositories, especially helping developers understand, write and test source code and make informed development decisions by learning from software "Big Data". For more information, please visit: https://zhongxin-liu.github.io/

Xing Hu received the PhD degree from Peking University. She is an associate professor in the School of Software Technology with Zhejiang University, China. Her research interests include software engineering, such as AI for SE and mining software repositories. Her work has been published in flagship conferences and journals such as ICSE, ASE, TOSEM, and EMSE. She serves regularly as a program committee member of international conferences in the field of software engineering, such as ASE and SANER, and she is a regular reviewer for software engineering journals such as EMSE and TSE. She is one of two recipients of the distinguished program committee member award for the SANER 2023. For more information, please visit: https://xing-hu.github.io/

Xin Xia received the PhD degree in computer science from Zhejiang University, in 2014. He is the director of the software engineering application technology lab, Huawei, China. Prior to joining Huawei, he was an ARC DECRA Fellow and a Lecturer with Monash University, Australia. To help developers and testers improve their productivity, his current research focuses on mining and analyzing rich data in software repositories to uncover interesting and actionable information. For more information, please visit: https://xinxia.github.io/.

Shanping Li received the PhD degree from the College of Computer Science and Technology, Zhejiang University, Zhejiang, China, in 1993. He is currently a professor with the College of Computer Science and Technology, Zhejiang University, China. His research interests include software engineering, distributed computing, Linux operating system, parallel programming paradigms including Open MPI, PMIx and PRRTE, failure detection and notification, long vector extension analysis and usage of Arm SVE, and Intel AVXs.