Learning to Locate and Describe Vulnerabilities

Jian Zhang[†], Shangqing Liu[†*], Xu Wang[‡§], Tianlin Li[†], and Yang Liu[†]
[†]Nanyang Technological University, Singapore
[‡]SKLSDE Lab, Beihang University, Beijing, China
{jian_zhang, liu.shangqing, tianlin001, yangliu}@ntu.edu.sg, xuwang@buaa.edu.cn

***Abstract*—Automatically discovering software vulnerabilities is a long-standing pursuit for software developers and security analysts. Since detection tools usually provide limited information for vulnerability inspection, recent work turns the attention to identify fine-grained vulnerabilities, i.e., vulnerable statements. However, existing work for vulnerability localization struggles to capture long-range and integral dependency information due to the bottleneck of Graph Neural Networks (GNNs). Moreover, little research has been done to help developers understand detected vulnerabilities, leaving vulnerability diagnosis a challenging task. In this paper, we propose VulTeller, a deep learning-based approach that can automatically locate vulnerable statements in a function and more importantly, can describe the vulnerability. Our approach focuses on extracting precise control and data dependencies in the code, achieved through modeling control flow paths and employing taint analysis. We design a novel neural model that encodes the control flows and taint flows which reside in the control flow paths, and decodes them via node classification and an attentional decoder for the two tasks respectively. We conduct extensive experiments with real-world vulnerabilities to evaluate the proposed approach. The evaluation results, including quantitative measurement and human evaluation, demonstrate that our approach is highly effective and outperforms state-of-the-art approaches. Our work for the first time formulates the problem of vulnerability description generation, and makes one step further towards automated vulnerability diagnosis.**

***Index Terms*—Vulnerability Diagnosis, Vulnerability Localization, Description Generation, Deep Learning**

I. Introduction

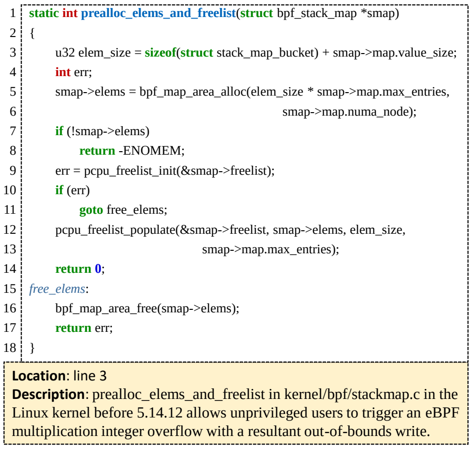
Vulnerabilities are typically security weaknesses in software or systems that can be exploited by attackers to perform malicious actions [1]. Therefore, it is crucial to rapidly discover and eliminate vulnerabilities to prevent security risks. However, in practice, developers often spend considerable time for fixing a vulnerability, especially when they do not know the weakness (e.g., potential attacks) [2], [3]. Hence, in order to facilitate vulnerability repair, it is also vital to explore automated detection and diagnosis techniques that cater to the vulnerability information needs of developers [4].

Prior studies have devoted a substantial effort towards automated vulnerability detection and they can be roughly categorized into static analysis-based and deep learning-based (DL-based) approaches. In general, static analysis tools tend to report an excess of false positives since they usually rely on a set of manually pre-defined rules [4], [5]. It is difficult and time-consuming for developers to find out the real vulnerabilities in redundant and trivial reported issues because most of the information is meaningless [6]. On the contrary, DL-based models have demonstrated their superiority in vulnerability detection by capturing complex code semantics [7], [8], [9]. However, most of these DL-based approaches only report vulnerabilities at the function level (i.e., predicting vulnerable or not). As a result, verifying the reported vulnerability becomes another burden for developers or even security experts due to no enough information [10].

To alleviate the problem, recent work aims to locate vulnerable statements with different DL-based approaches [11], [12], [13], [14]. Existing work can be divided into two categories according to their methodology: feature learning and end-to-end learning. Feature learning means that the vulnerable statements are predicted via inner layer features of function-level detection models. For example, IVDetect employs Graph Convolution Network (GCN) over program dependence graph (PDG) of source code for function-level vulnerability detection, and derives the vulnerable statements of subgraphs as the interpretation using the learned features [12]. However, such features are not necessarily consistent with the vulnerable statements, since a vulnerability usually includes many fewer locations (e.g., single lines) compared to that of the subgraph. Consequently, the performance is found to be not satisfactory when used for locating vulnerable statements [14]. For this reason, LineVD learns to locate vulnerable statements in a fully supervised manner based on PDG and Graph Attention Network (GAT), which achieves a significant improvement [14]. Such approaches are beneficial for reducing the time cost of debugging a discovered vulnerability, yet they still have two main limitations. First, existing vulnerability localization approaches simply borrow off-the-shelf Graph Neural Networks (GNNs). As GNNs are usually designed to capture neighborhood information, whereas the size of program graphs such as PDG can be very large (e.g., an average of 158 nodes and 487 edges in our dataset). Hence, applying them to large program graphs weakens the capability for obtaining precise dependencies including long-range and integral dependency information [15], [16], [17]. Second, in practice, misunderstanding of security concepts is quite common for developers due to lack of security knowledge [18], [19]. Therefore, it remains challenging to figure out the behavior and reason for the vulnerability.

In this paper, we propose a novel neural approach namely VulTeller, which learns to locate vulnerable statements and meanwhile generate natural language descriptions, so as to provide additional clues for diagnosing the reported vulnera-

[*]Corresponding author.
[§]Also with Zhongguancun Laboratory, Beijing, P.R.China.



```
1  static int prealloc_elems_and_freelist(struct bpf_stack_map *smap)
2  {
3      u32 elem_size = sizeof(struct stack_map_bucket) + smap->map.value_size;
4      int err;
5      smap->elems = bpf_map_area_alloc(elem_size * smap->map.max_entries,
6                                       smap->map.numa_node);
7      if (!smap->elems)
8          return -ENOMEM;
9      err = pcpu_freelist_init(&smap->freelist);
10     if (err)
11         goto free_elems;
12     pcpu_freelist_populate(&smap->freelist, smap->elems, elem_size,
13                            smap->map.max_entries);
14     return 0;
15 free_elems:
16     bpf_map_area_free(smap->elems);
17     return err;
18 }
```

Location: line 3
Description: prealloc_elems_and_freelist in kernel/bpf/stackmap.c in the Linux kernel before 5.14.12 allows unprivileged users to trigger an eBPF multiplication integer overflow with a resultant out-of-bounds write.

Fig. 1. A motivating example of one vulnerable C function from CVE-2021-41864. The message below the function is the location and description of the vulnerability that can be automatically reported by our approach. The fix for the integer overflow of line 3 is to define a variable with greater size by "`u64 elem_size = sizeof(struct stack_map_bucket) + (u64)smap->map.value_size;`" to avoid the out-of-bounds write.

bility. That is, we separate it into two sub-tasks: vulnerability localization and description generation. Suppose the function *prealloc_elems_and_freelist* is detected as vulnerable by existing detectors in Figure 1, and a developer tries to inspect and confirm it. In most cases, it is difficult to find out the reason for the vulnerability without additional information. Fortunately, we can first locate the vulnerable statements to narrow down the scope of inspection (e.g., line 3), and then generate descriptive text in the form of natural language to explain what the vulnerability is and why it occurred. Note that there can be multiple vulnerable statements in a function. In this example, an integer overflow can be caused by the variable declaration statement if an attacker writes a large value (i.e., out-of-bounds write) to it. By contrast, providing such messages can substantially reduce the time required for a developer to identify and remediate vulnerabilities.

The challenge for tackling the two tasks is how to obtain precise and sufficient vulnerable code semantics. Different from existing work mentioned earlier, we design a novel neural model based on control flow graph (CFG) and taint analysis [20] to better capture control and data flow dependencies in the functions. Specifically, to analyze the source code of one suspicious function, we begin by converting it into a CFG and extracting all simple control flow paths consisting of multiple nodes. To depict control dependency, we could use one Recurrent Neural Network (RNN) [21] to encode all these node sequences. However, the number of paths is usually large, and most of them would be noise. To obtain precise control dependency and data dependency, we conduct taint analysis on the source code, resulting in potentially tainted data flows from sources to sinks. We rank the control flow paths by their overlapping frequency with the data flows and select the top-k paths, which can filter out noise and refine data dependencies. Next, we use a Bidirectional Gated Recurrent Unit (Bi-GRU) [22] to encode the top-k paths along with their ranks, and apply node-wise max pooling to get the vector representation of each node. To locate vulnerabilities, we perform node classification to classify each node as vulnerable or not. For description generation, we use a GRU-based decoder with an attention mechanism over nodes to generate the word one by one and finally form the sentence.

We collect real-world C/C++ vulnerabilities for evaluation, each of which contains the vulnerable function as well as its vulnerable locations and description. We compare VulTeller with a variety of baselines including static analysis tools and DL-based approaches on the dataset. The extensive experimental results demonstrate that VulTeller outperforms these baselines significantly. We also conduct a human evaluation to assess the quality of generated descriptions and the results further confirm the effectiveness of VulTeller. In summary, the main contributions of this work are as follows:

- We formulate a new problem of vulnerability description generation. To the best of our knowledge, this is the first work that aims to generate descriptions of the vulnerabilities, which can further assist developers in vulnerability diagnosis.
- We propose a novel neural model for vulnerability localization and description generation. It leverages the knowledge from control flows of CFG and data flows of taint analysis to precisely capture control and data dependency information.
- We conduct extensive experiments on a large-scale dataset consisting of vulnerabilities from real-world C/C++ projects, which demonstrates the effectiveness of our approach in terms of automatic metrics and human evaluation.

II. Problem Definition

Our goal is to facilitate developers in diagnosing newly discovered vulnerabilities, especially when they lack security knowledge. To this end, we formulate the problem of vulnerability diagnosis as two conjunctive tasks. One is *vulnerability localization* that locates vulnerable statements within a function. Another is *description generation* that describes symptoms and causes of the vulnerability. We formally define them as follows.

Let $F = (t_1, t_2, \dots, t_C) \equiv (s_1, s_2, \dots, s_N)$ demotes the source code of a function , where $t_i, i \in [1, C]$ and $s_j, j \in [1, N]$ are the tokens and statements respectively in the function.

Vulnerability Localization. With the statements $S = (s_1,\dots,s_N)$ as input, the target is to find a subset $S' \subset S$, each element of which is a vulnerable statement. Alternatively, a learning-based model may learn the mapping $\phi : F \to Y$ to give all the labels of the statements as $Y = (y_1, y_2, \dots, y_N)$, where $y_j \in \{0, 1\}$ indicates if the statement s_j is vulnerable or not.

Description Generation. The input of this task could be either tokens t_1 to t_C or statements s_1 to s_N. The target is to learn a text generator $\psi : F \to D$ that generates natural language description $D = (d_1, d_2, \dots, d_M)$ to describe the vulnerability in F where d_i is the i-th token in the generated sequence of length M.

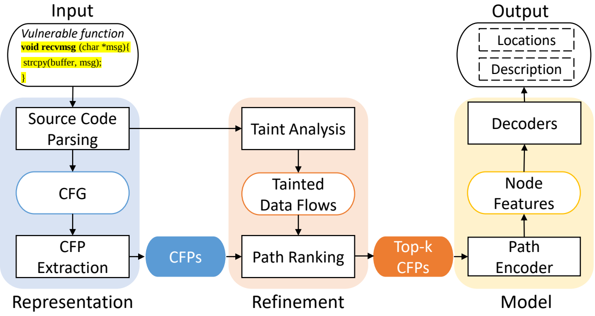
In our work, we assume that the vulnerable function has been reported by a static detector because vulnerability detection is a comparatively well-studied problem, and there remains extra manual effort for debugging. However, our approach can also easily generalize to other scenarios by adding non-vulnerable functions into the training set. For example, when no statements are located by our model, it means that the function is predicted to be not vulnerable, and we do not generate the description for it. We provide the details of our approach in the next section.

III. Approach

A. Overview

Existing studies [8], [12], [14], [23] have confirmed that utilizing the dependency information from graph structures is able to capture complex program semantics. However, it is still under-explored regarding the feasibility of directly borrowing GNNs on vulnerability discovery, including Graph Convolutional Network (GCN) [24], Gated Graph Neural Network (GGNN) [25] and Graph Attention Network (GAT) [26] that are designed to analyze spectral graphs instead of program graphs. GNNs are capable of capturing the local information from the neighborhood nodes, while the long-range dependencies from the remote nodes are prone to be missed [15], [16]. Unfortunately, program graphs especially program dependency graph (PDG) are usually large (e.g., up to 10k nodes in Sec. 4.1). As a result, existing GNNs may not precisely capture the node dependencies and thus weakens the performance. Therefore, we propose a novel approach to overcome the drawback. On the one hand, we use the control flow graph (CFG) instead of PDG as a basis because of its moderate size while carrying essential semantics in the form of control flows. On the other hand, in order to consider data dependencies, we adopt taint analysis to identify tainted data flows in it. Based on them, we devise a novel neural model to capture precise control and data dependencies for the two tasks.

The overall workflow of our approach is shown in Figure 2. In the representation phase, we parse a code fragment into CFG and simplify it by node reduction to remove redundant nodes. We extract the control flow paths (CFPs) from the simplified CFG. Due to the path explosion problem and noises in them, taint analysis is adopted to identify potentially tainted data flows and CFPs are ranked based on their overlap with tainted flows, allowing us to keep the top-k CFPs with ranks that indicate priorities. Next, we use a bidirectional GRU module to encode selected CFPs into node vectors. Note that each node may have multiple vectors, we consider the ranks of their corresponding CFPs as an additional feature to form new vectors. Afterwards, we perform the max pooling over those new vectors and get the node representations. For vulnerability localization, we transform it into a node classification problem, where each node (i.e., statement) is predicted as vulnerable or not; For description generation, we adopt a GRU decoder with node attention to generate sentences that describe the vulnerability.



```
void recvmsg (char *msg){
 strcpy(buffer, msg);
}
```

Fig. 2. The overall workflow of our approach.

B. Representation

Given the source code of one function, we use an off-the-shelf parser (e.g., Joern [27]) to get the CFG representation. Nevertheless, the representations of CFG vary on the granularity of basic blocks. For instance, Joern treats the basic block as one basic operation (e.g., arithmetic operation) and separates one statement into several iterative operations (nodes). Such a practice is useful to know the micro semantics, but they are not suitable for node classification that determines vulnerable statements. To simplify it, we cut off nodes that are elements of other nodes while keep the original control flows. For example, if the control flow is "1 $\rightarrow$ 2 $\rightarrow$ 3 $\rightarrow$ 5" and node 2 belong to the statement of node 3, we convert it to "1 $\rightarrow$ 3 $\rightarrow$ 5". Formally, let $G_o = (\mathcal{V}_o, \mathcal{E}_o)$ denote the original CFG, where $\mathcal{V}_o$ and $\mathcal{E}_o$ are the sets of nodes and edges respectively. We can simplify G_o by node reduction, that is, $v_i = reduce(v'_j, \dots, v'_k), v' \in \mathcal{V}_o \wedge v_i \in \mathcal{V}$ if $v'_j, \dots, v'_k$ come from the statement s_i. Likewise, the edges are simplified accordingly. Through simplification, we obtain the simplified CFG as $G = (\mathcal{V}, \mathcal{E})$.

After that, we extract control flow paths from it. In this paper, we define a control flow path as one simple path from the entry to the exit of a CFG. Here simple path means that a path may have repeat nodes but not edges. This is to consider the semantics of loops that may incur a vulnerability. The algorithm for extracting simple paths is performing a depth-first-search over the nodes of CFG and recording visited edges, which yields the edges in order. We sort out the edges and get the sequences of nodes as full CFPs. For convenience, we denote them as $\mathcal{P} = \{p_1, p_2, \dots, p_T\}$ and each p has multiple nodes from $\mathcal{V}$.

C. Refinement

When we have the full CFPs, a straightforward practice is to consider all of them as the input of neural models. However, the path explosion problem [28] may occur since there could be numerous paths in one CFG, particularly in real-world programs where the nested branches or loops are common. Besides, not all paths are helpful for understanding the semantics of vulnerability, and many of them could be noises [29].

1) Taint Analysis: To overcome it, we employ static taint analysis [30] for reducing the noisy information and meanwhile extracting data dependency information. Taint analysis or taint tracking is a technique of information flow analysis by tracking from a tainted source (e.g., untrusted data) through the system to a specified sink (e.g., sensitive operators) [20], [31], [32]. It consists of three important parts, namely source, sink and sanitizer. Generally, a vulnerability may arise from the source and occur at the sink. However, if there is a sanitizer in the program that checks or cleanses the source before it flows to the sink, then the vulnerability would not occur any more.

Therefore, we adopt the methodology introduced by Yamaguchi et al. [33], [30] for conducting taint analysis, which is applicable to the intra-procedural analysis at the source code level. That is, we mark the parameters, global and local variables as sources, and specify function calls as sinks. The rationale behind regarding local variables as taint sources is that these variables frequently serve as origins of system inputs. For example, a local variable defined to read the untruasted file. Then we traverse the dependency graph of the source code, which starts from the sources and propagates to the sinks to identify cases where attacker-controlled data is used by a sensitive operation. In order to exclude properly sanitized cases, we utilize the symbol-specific syntax-only sanitizer descriptions for UNSANITIZED traversal [30]. The traversal exclusively identifies attacker-controlled sources under the fulfillment of the following conditions:

- There exists a path from the source statement to the sink statement in the control flow graph, such that no node on the path matches any of the sanitizer descriptions.
- A variable initially defined by the source and subsequently utilized by the sink successfully reaches the sink through the control flow path. This signifies that the variable remains unaltered by any node along the path.

Taking buffer overflows as an illustrative example, the motivation of adopting sanitizer descriptions is that buffer overflow instances often result from insufficiently validated length fields supplied to copy operations. To elaborate, consider the context of the Linux kernel code, where numerous buffer overflows occur when size fields, retrieved by the function get_user, are directly passed as third arguments to memcpy without a proper validation. This vulnerability pattern can be captured by a traversal as follows:

$$\text{ARG}^{1}_{\text{get\_user}} \circ \text{UNSANITIZED}\{\mathcal{T}_s\} \circ \text{ARG}^{3}_{\text{memcpy}}$$

This traversal targets third arguments for memcpy that are directly influenced by the first arguments to get_user and lack proper validation, as indicated by nodes satisfying the traversal $\mathcal{T}_s$. A match traversal could involve relational expressions containing the tracked variable s, e.g., $x <$ buffer_size or within a call to the `MIN` macro. In addition, sanitizer descriptions are also well suited for a wide range of vulnerabilities, such as injection vulnerabilities, integer overflows, and missing permission checks. This ensures that not all the data flows are treated as taint flows, which helps us to extract more useful dependency information. Formally, the process of taint analysis yields multiple taint flows $\mathcal{F}$, and each flow $f \in \mathcal{F}$ consists of the tainted variables.

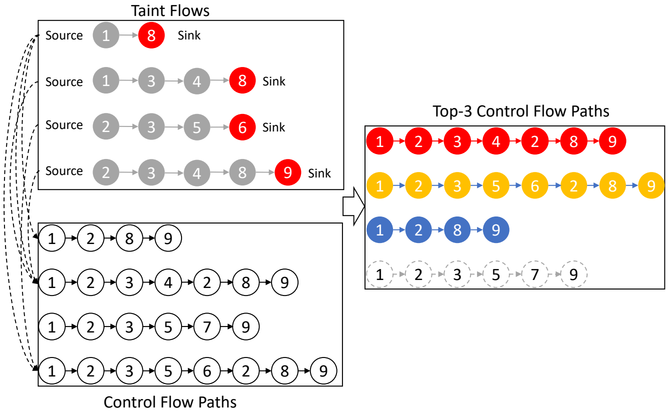


Fig. 3. An example for the process of path ranking.

2) Path Ranking: As it should be, there could still exist false taint flows since a static tool can not perfectly identify all sanitizers. Nevertheless, we can utilize the results of taint analysis to select control flow paths that have more taint flows. The intuition is that one path could more likely contain vulnerable nodes (i.e., statements) if it is prone to be tainted. One possible solution is to consider the overlaps of nodes regardless of their inclusion relations or orders. However, it may wrongly place one CFP to the high rank only because it contains more nodes. Therefore, we propose a flow-aware matching algorithm to rank the CFPs by their overlaps between taint flows. Specifically, for each CFP p_i, if all tainted identifiers of one taint flow f_j can be found in p_i and they follow the order of nodes of p_i, then f_j is a valid overlap and the number increments. Note that we only count the situation of $f \subseteq p$ because it implies that the taint flow f comes from the CFP p, and thus p is more vulnerable. We sort the T CFPs according to their overlapping numbers in an descending order and keep top-k CFPs, denoted as $\mathcal{P}_k \subset \mathcal{P}, k \leq T$. Here k is a hyper-parameter, and can be optimized based on the validation set during the learning stage.

To better illustrate this process, we present an example in Figure 3. In this example, we assume that the CFG has 4 simple paths and there are 4 taint flows obtained from the step of taint analysis. Then, we count the numbers of taint flows that are subsequences for the 4 CFPs p_1 to p_4. In this example, we can observe that p_1 has one taint flow (i.e., the dotted arrow), p_2 has three and so on. Finally, we select the top-3 CFPs which are sorted by their numbers and discard the 4th one.

D. Model

We introduce the neural network for vulnerability localization and description generation, which is presented in

Figure 4. The model consists of three components: the path encoder, locator, and generator. We elaborate on each of these components as follows.

1) Path Encoder: The path encoder is used to encode the top-k control flow paths mentioned above. Specifically, suppose we have the top-k CFPs $\mathcal{P}_k = \{p_1, p_2, \dots, p_k\}$ and $p_i = (v_1, \dots, v_{n_i})$, we encode them individually in a sequential manner. Here v_1 and v_n are the entry and exit node of CFG respectively, and there are n nodes in total. Given one node v_j that corresponds to one statement $s_j = (t_1, \dots, t_b)$ and t is one of the tokens in it, we first embed them into token vectors via an embedding layer W_e, that is, $x^i_j = \bigcup_{t \in s_j} W_e(t)$. For simplicity, we use the notation of e^i_j to represent the initial vector of node v_j in path p_i, and calculate its value by max pooling, that is, $e^i_j = max(x^i_j)$.

Since the dependency information (i.e., control/data dependency) has been integrated into $\mathcal{P}_k$, we can automatically capture it via sequential neural networks such as GRU [34] or LSTM [35], which have been demonstrated effective in learning semantics of dependencies [36], [7]. In this work, we choose to use GRU because it is more efficient than LSTM yet has comparable performance. The GRU works in a forward way to encode the node embeddings by treating one path as a node sequence. Furthermore, we adopt a backward GRU to enhance the captured dependency information and concatenate its hidden states with the forward ones, known as bidirectional GRU (Bi-GRU). Let $Q = [e^i_1, \dots, e^i_{n_i}]$ denote the vectors of embedded nodes in the sequence, we encode the sequence into hidden states $H(Q) = [h^1_1, \dots, h^i_{n_i}]$. For each $h^i_t, t \in [1, n_i]$, we simply abbreviate the formula as:

$$h^i_t = Bi\text{-}GRU(e^i_t, h^i_{t-1}), \tag{1}$$

and t corresponds to the time step during a single path encoding.

After we obtain all the hidden states of the k CFPs, we transit the nodes into vectors in the light of their respective locations as well as path ranks, which we call it *Rank-aware Transition*. In specific, assume that node v_j exists in multiple paths of $\mathcal{P'}_k = \bigcup_{i=i_0}^{i_k} p_i, 1 \leq i_0 \leq i_k \leq k$, for each path p_i, the node vector h'^i_j is calculated by the following equations:

$$\begin{aligned} h^i_j &= map(j, h^i_t), \\ h'^i_j &= \sigma(W_k([h^i_j; 1/r^i_j]) + b_k), \end{aligned} \tag{2}$$

where the function map is used to look up the hidden state of the node in this path, and r^i_j represents the path rank. W_k, b_k are the weight matrix and bias term respectively. We concatenate the inverse rank as an additional feature to the acquired hidden state, and project them to a new vector space. This indicates that the importance of node v_j from p_i will be largely weakened if it ranks poorly. Consequently, the influence of noisy paths will be eliminated or reduced. Then we perform max pooling on all the path-dependent node vectors to get the final vector representation of this node as:

$$h'_j = max(\bigcup_{i \in [i_0, i_k]} h'^i_j). \tag{3}$$

These node vectors encode vulnerability-related dependency information from both control flows and tainted data flows, so as to be exploited for locating vulnerable statements and generating vulnerability descriptions.

2) Vulnerability Locator: Now that we get the node vectors, and recall that we have ensured all the nodes are distinct in terms of statements, we can easily transform the problem of vulnerability localization into node classification. In detail, each node is classified into vulnerable or not via a multi-layer perceptron (MLP) classifier to calculate the logits, namely $l_j = MLP(h'_j)$. Then the process of making predictions can be expressed as:

$$\hat{y_j} = \begin{cases} 1, & sigmoid(l_j) > \delta \\ 0, & sigmoid(l_j) \leq \delta \end{cases} \tag{4}$$

where δ is a threshold.

For training the locator, we use the binary cross-entropy which is defined as:

$$\mathcal{L}(\Theta, \hat{y}, y) = -\sum_{i=1}^{X} \sum_{j=1}^{N} (y^{(i)}_j \cdot log(\hat{y}^{(i)}_j) + (1 - y^{(i)}_j) \cdot log(1 - \hat{y}^{(i)}_j))$$

Here X is the number of samples in the training set.

3) Description Generator: Similarly, we can generate descriptive words that may include the symptom or reason for the vulnerability by decoding the node vectors. Inspired by code comment generation [37], our generator is based on GRU because of its effectiveness in understanding code semantics. When decoding the vulnerable code to generate the m-th word at time m, the hidden state g_m of the decoder is updated by:

$$g_m = GRU(g_{m-1}, d_{m-1}), \tag{5}$$

where d_{m-1} is the previous input vector. Next, an attention module [38] is adopted, which can distinguish the nodes for providing more useful information of a vulnerability. It computes the context vector over the node vectors $H(Q)$, denoted as:

$$c_m = Attention(g_m, H(Q)) \tag{6}$$

To jointly take into account past alignment information, we apply the input-feeding strategy [39] which concatenates c_{m-1} with inputs d_{m-1}. Then the probability for the next word d_m is

$$P(d_m | d_1, \dots, d_{m-1}, F) = gen(d_{m-1}, g_m, c_{m-1}), \tag{7}$$

where gen is the generator function implemented by an MLP layer along with $softmax$.

Training such a generator is to minimize the loss function:

$$\mathcal{L}(\theta) = -\sum_{i=1}^{X} \sum_{j=1}^{M} \log P(d^{(i)}_j | d^{(i)}_{<j}, F^{(i)}),$$

where θ is the trainable parameters.

After training, we choose the word with the highest probability as the generated word at each step. The process iterates until the end of sentence word `</s>` and results in a complete description for the vulnerability.

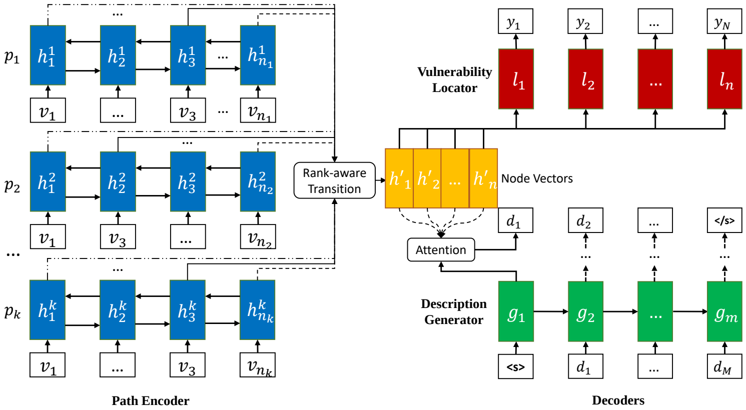


Fig. 4. The architecture of our model. Here the locator and generator only share the structure of the encoder, and they are trained independently to produce two separate models.

IV. Evaluation

A. Dataset

We build our dataset based on the widely-used BigVul [40], which provides a large number of vulnerabilities (CVEs) reported and/or validated by security experts from more than 300 different open-source C/C++ GitHub projects, along with their vulnerability patches and descriptions. Notably, the dataset contains a total of 11,823 vulnerable functions. However, the original dataset has its flaws, such as meaningless code changes (e.g., formatting changes) and incomplete functions. Following LineVD [14], we exclude such instances to keep the dataset consistent with it. We only kept the first two sentences of the CVE descriptions in consideration of covering more vulnerability-related information while removing lengthy and irrelevant descriptions at the same time. Moreover, we normalized the version numbers and file paths to “VERSION” and “FILE” respectively, to further streamline the descriptions and retain more informative words. As a result, our dataset consists of 10,811 distinct vulnerable C/C++ functions along with their patches and descriptions. These vulnerabilities originate from a range of 293 projects, spanning across 71 vulnerability types, thereby contributing to a diverse and comprehensive dataset for our task.

We create ground-truth labels of vulnerable statements based on the patches using heuristics following existing works [12], [14]. Specifically, we treat removed lines or all lines that are control or data dependent on the added lines as vulnerable. We then randomly split the dataset into training, validation, and test sets with a ratio of 8:1:1. Table I provides some statistics about our dataset. #FuncToken and #DescWord represent the number of tokens in the functions and words in the descriptions respectively. #FuncLine and #FuncVline are the numbers of lines and vulnerable lines in the functions. #FuncUnqT. and #DescUnqW. are the respective vocab sizes. We also compare the graph size between PDG and CFG for example #PDGNode and #CFGNode represent the number of nodes in the PDGs and original CFGs. On average, the size of PDG in terms of edges is much larger than that of CFG, which indicates that it is hard to capture precise dependency information if we directly apply existing GNNs to it (as mentioned in Section III-A).

TABLE I
The statistics of our dataset.

Source	Avg.	Max	Graph	Avg	Max
#FuncToken	606	33,939	#PDGNode	158	11,859
#FuncLine	107	6,341	#PDGEdge	487	251,733
#FuncVLine	7	498	#PDGPath	>1MM	>40MM
#FuncUnqT.	191,692		#CFGNode	154	10,561
#DescWord	43	148	#CFGEdge	170	11,439
#DescUnqW.	10,986		#CFGPath	1602	66,593

B. Experimental Settings

We implement VulTeller with Pytorch and Joern, where Joern is used to parse the C/C++ functions to construct CFGs, as well as perform taint analysis. We set the vocabulary sizes for functions and descriptions to 50K and 10K, respectively, and used an embedding size and hidden state dimension of 256. To handle the oversized CFGs, we limited the number of nodes in the CFPs to 200 and kept the top-10 CFPs, since we noticed marginal improvements as the limit was increased. For vulnerability localization, we set the threshold δ to 0.1 according to the best F1-score on the validation set, as is commonly done in existing work [14]. For description generation, we employed a vanilla greedy search approach and established a maximum description length of 50 to mitigate lengthy and repetitive outcomes. We used a batch size of 32 and the Adam optimizer with a learning rate of 0.001 for training the vulnerability locator and description generator, with maximum epochs of 10 and 50 respectively, as they have different convergence rates.

All experiments were conducted on an Ubuntu 18.04 server with 48 cores of Intel(R) Xeon(R) Silver 4214 CPU @ 2.20GHz, 256GB RAM, and a GTX3090 GPU with 24GB memory.

C. Baselines

For vulnerability localization and description generation, we respectively select the following baselines for comparison.

1) Vulnerability Localization: There are several DL-based approaches that directly or indirectly (i.e., supervised / unsupervised learning) predict the locations of vulnerabilities. To investigate the effect of different learning manners, we choose two typical models i.e., **IVDetect** [12] and **LineVD** [14] respectively. We reproduce them from their publicly available repositories with default settings and got similar yet slightly lower results, thus we cite the stated ones to avoid the bias. Additionally, since static analysis tools are commonly used in practice [41], [42], we also include two open-source tools, **Cppcheck** and **Flawfinder**. They can identify line-level vulnerabilities in C/C++ code and have been evaluated for vulnerability detection in previous studies. We acknowledge the presence of more advanced tools such as Infer [43] and CodeQL [44]. However, these tools necessitate access to compiled code. This poses a challenge for our dataset, which encompasses a multitude of projects and over 3,000 commits. Consequently, the application of these tools to our extensive dataset is not straightforward. We use default configurations to scan the vulnerabilities in the functions.

2) Description Generation: After a thorough search of the literature on vulnerability analysis, we did not find any existing work that focuses on generating vulnerability descriptions. Instead, we select some typical models such as Seq2Seq, Transformer, CodeBert, G2S, and GraphCodeBert for comparison. We use them to generate descriptions based on the source code, with the same hyperparameters as our model unless otherwise specified.

- **Seq2Seq**. Seq2Seq is a classic neural encoder-decoder model that has been used to generate code documents including code comments [37], commit messages [45], etc. It encodes the token sequence of one function into vectors via Bi-LSTM and decodes it to text with an attentional LSTM. We limit the max length of source code to 400 for a high coverage.
- **Transformer**. Transformer follows an encoder-decoder structure but uses self-attention mechanisms to process sequences of data (i.e., token sequence) and has also been widely adopted in code or its comment generation [46], [47]. Following existing work, we use 6 layers and 8 heads in the Transformer.
- **G2S**. G2S employs an encoder-based on GGNN and a RNN decoder, which shows strong performance in NMT. Similarly, there exist GNN-based models for code comment generation such as GCN and GAT [48], [49]. In order to apply G2S in vulnerability description generation, we parse the source code into PDG and use LSTM as the decoder. We set the propagation step as 3 which yields a fairly better result.
- **CodeBert**. CodeBert is a bimodal pre-trained model based on Transformer that captures the semantic connection between natural language and programming language, and produces general-purpose representations to support generation tasks [50]. We keep the dimensions of embeddings and hidden states as default since there is no way to easily change the hyper-parameters of pre-trained model.
- **GraphCodeBert**. GraphCodeBert is a more recent pre-trained model for programming language based on Transformer that introduces a data flow graph-guided masked attention function to incorporate the code structure [51]. We keep the settings as same as CodeBert.

TABLE II
THE EFFECTIVENESS OF DIFFERENT APPROACHES FOR VULNERABILITY LOCALIZATION.

Methods	**Precision**	**Recall**	**F1-score**
Cppcheck	11.9	2.3	3.6
FlawFinder	3.9	2.8	2.2
IVDetect	23.8	14.0	17.6
LineVD	27.1	53.3	36.0
VulTeller	**38.9**	**55.6**	**45.8**

D. Evaluation Metrics

We evaluate the performance of different approaches on the two tasks with different metrics. For vulnerability localization, since it can be treated as a binary classification problem, we adopt Precision (abbr. P), Recall (abbr. R) and F1-score (abbr. F1) to measure the accuracy of each approach in locating the vulnerable statements. For description generation, we use smoothed BLEU-4 [52] and ROUGE-L [53] to assess the textual similarity between the reference descriptions and the generated ones, which are also commonly used in code document generation tasks. For these metrics, we present their values in percentage and a higher value means better performance.

E. Results

We investigate the effectiveness of all aforementioned approaches by answering the following three research questions.

RQ1. How does our approach perform on vulnerability localization compared to the state-of-the-art?

Table II presents the experimental results of all approaches. First, we can see that static analysis-based tools perform badly in terms of F1-score. Not only do they report a large fraction of false positives, but also they miss many true positives. For example, the precision of Cppcheck is 11.9%, which means that nearly 90% of the reported statements are not vulnerable. And it does not report any vulnerabilities in some cases, leading to a low recall value. FlawFinder achieves a slightly higher recall, yet the precision is even lower. Intuitively, this is not surprising in that static tools solely depend on predefined rules/patterns, whereas vulnerabilities are naturally complex and varied. To further figure out the reason for the high false positive/negative rate, we manually inspect a subset of the warnings. We find that they tend to report potential vulnerabilities based on the occurrence of specific

TABLE III
THE EFFECTIVENESS OF DIFFERENT APPROACHES FOR DESCRIPTION GENERATION

Groups	Models	BLEU	ROUGE-L
Sequence-based	Seq2seq	42.1	54.2
	Transformer	41.8	54.1
	CodeBert	43.4	57.0
Graph-based	G2S	37.2	50.8
	GraphCodeBert	43.9	57.6
	VulTeller	**46.3**	**58.6**

API calls. For instance, FlawFinder reports a buffer overflow vulnerability where the "fgetc()" is called, even though the boundary has been checked with another API. Conversely, in most cases when it misses a real vulnerability, there are no such APIs and it simply reports the result of no hits found.

It is obvious that neural network-based approaches achieve much better performance. In particular, the precision of IVDetect is 23.8%, nearly twice the precision of Cppcheck. Moreover, the recall values of both neural models far outweigh those of static tools. The significant improvement may largely attribute to the capability of neural models for learning semantic patterns across the structure of PDGs. By comparing IVDetect and LineVD, we can observe that their recall values also vary a lot. Since they have the same source code representation (i.e., PDG) and similar GNNs, the most possible reason is that an end-to-end model captures more statement-level vulnerability information as LineVD does. While IVDetect relies on the inner features of the learned model and thus may not explicitly contribute to locating the vulnerable statements.

Our approach improves the precision of all baselines by a large margin and meanwhile obtains an even higher recall. This demonstrates that more precise dependency information is really beneficial to exclude false positives by capturing control/data dependencies from control flow paths and reducing noises with taint analysis. Nonetheless, the overall performance of all approaches is not high enough, indicating that automated vulnerability localization is challenging. In practice, one can raise the threshold δ to further reduce false positives, and refer to the generated description to more easily confirm a vulnerability.

RQ2. How effective is our proposed approach for vulnerability description generation?

Initially, we could include the generated descriptions of static tools for comparison. But given the inconsistent specifications between the tools and CVEs, and the high false positive rate, the performance can be quite poor (e.g., FlawFinder got BLEU=5.2% and ROUGE-L=29.9%). Therefore, we focus on learning-based approaches here.

As shown in Table III, most of the models yield good results in terms of both metrics. For instance, Seq2seq model obtains a BLEU score of 42.1%, which is competitive with the standard of machine translation. This means that such neural networks are indeed feasible for generating vulnerability descriptions. However, due to the diversity of vulnerabilities and details, improving the quality of generated descriptions is not trivial. For example, Transformer performs worse than Seq2seq (e.g., 0.3% lower in terms of BLEU) despite its self-attention mechanism. In fact, not all the tokens necessarily have dependencies inside a program, hence considering all the relations of the tokens may cause confusion. By contrast, CodeBert improves them significantly since it utilizes the prior knowledge learned from a large code corpus, which helps to focus on real dependencies that are related to vulnerability semantics.

G2S is expected to capture useful dependencies over the PDGs of a given vulnerable function. However, the BLEU score as well as the ROUGE-L score is substantially lower than sequence-based ones, for example, Transformer outperforms it by 4.6% in terms of BLEU. This may blame to the large size of PDG and the bottleneck of existing GNNs in capturing long-range dependency information as previously discussed. GraphCodeBert shows a marked advantage over G2S and achieves the best performance among all the baselines, even though it is only based on BERT and data flow graph without GNNs. In specific, the score of ROUGE-L is 57.6%, which improves G2S and CodeBert by 6.8% and 0.6% respectively. We can deduce that such a model focuses more on the relation of "where-the-value-comes-from" between variables and thus excels in capturing data dependencies compared to others.

Recently, ChatGPT has showed a great capability of text generation, and one would like to know how it performs when generating vulnerability descriptions. Through few-shot learning [54], we got an BLEU score of 28.4% and ROUGE-L score of 48.6%. Even if there are ways to improve, we do not recommend relying on the online ChatGPT to analysis vulnerabilities (especially in industrial projects) due to the potential privacy and security issues.

VulTeller significantly outperforms all the baselines. Specifically, the BLEU and ROUGE-L scores exceed the best one (i.e., GraphCodeBert) by 2.4% and 1.0% respectively. It can be concluded that with precise control and data dependencies from the control flows along with tainted data flows, our model learns more accurate semantics of vulnerabilities, which actually helps in generating vulnerability descriptions. Besides, these results also pose a further need for more precise representations to understand the vulnerability.

RQ3. To what extent do the components of our approach contribute to the effectiveness of both tasks?

In this RQ, we want to know how the CFG simplification, taint analysis and path ranking influence the overall performance. We conduct the ablation study by removing or replacing each component once at a time, leaving everything else the same. The results are presented in Table IV.

First, we extract CFPs from the original CFG generated by Joern, and feed them into our two models (i.e., w/o CFG Simp.). We can observe that the performance on both tasks drops by 0.9% to 1.5%. This is because there are many redundant nodes and edges in the original CFG, where several nodes

TABLE IV
THE EFFECTS OF DIFFERENT COMPONENTS IN VULTELLER FOR THE TWO TASKS.

	Localization			Generation	
Strategy	**P**	**R**	**F1**	**BLEU**	**ROUGE-L**
VulTeller	**38.9**	**55.6**	**45.8**	**46.3**	**58.6**
w/o CFG Simp.	37.6	53.9	44.3	45.2	57.7
w/o taints	36.1	50.5	42.2	42.3	54.4
w/o ranks	38.7	53.2	44.8	45.6	58.1

belong to one single statement. Such redundancy information has a negative impact on capturing the natural dependency. After simplification, the average numbers of nodes and edges are 52 and 69, which are approximately twice smaller as the original ones. Hence, the size of the graph can indeed affect the performance of our model.

Next, we investigate the effect of CFP selection from two perspectives. One is totally abandoning taint analysis and using random K paths instead (i.e., w/o taints). As the table shows, the performance in terms of all metrics decreases significantly. In particular, the F1-score for vulnerability localization is 3.6% lower than VulTeller, and similarly the BLEU or ROUGE-L score. Although the control and data dependency can also be captured through our model, it misses vulnerable paths to a large extent without taint analysis. Moreover, such a random selection can bring much noise that weakens the capability of our model to focus on more important ones. Another is to keep the filtered CFPs but ignore their ranks when encoding these paths (i.e., w/o ranks). Compared to the former, it achieves a notable improvement, especially in locating vulnerable statements. Nonetheless, in comparison to the utilization of path ranking (i.e., VulTeller), the improvement achieved in description generation is relatively modest. We conjecture that the attention mechanism can partially counteract the weakness of unordered paths.

In summary, the size of CFG, taint analysis and path ranking all have a positive impact on the performance of our approach. When combining them together, we obtain the best results on two tasks.

F. Human Evaluation

Previously, we use the automatic metrics to calculate the similarity between generated descriptions and references. However, such metrics are not perfect for measuring semantic similarity because they are based on statistical analysis. Therefore, we perform a human evaluation for an in-depth measurement.

We invite 6 human evaluators with 2-5 years of software security experience to assess the quality of vulnerability descriptions generated by different approaches. Specifically, we compare our approach VulTeller and two representative baselines including Seq2seq and GraphCodeBert from each group. We choose Seq2seq instead of CodeBert because CodeBert exhibits close performance with GraphCodeBert, which makes it not distinguishable. We randomly select 100 function-description pairs from the testing set for evaluation, where the function contains the vulnerability and the description serves as a reference. Then we fetch generated ones of the three approaches and evenly divide them into two groups. Each group is assigned to three different evaluators since the redundancy can help obtain more consistent results. The generated descriptions are randomly presented, thus an evaluator will not be aware of which approach the description comes from. The evaluators can give a score between 1 to 7 to measure the semantic similarity between the generated description and the reference, where 1 means "Not Similar At All" and 7 means "Highly Similar/Identical". The higher score means closer quality of generated description to the reference. For each generated description, we get three scores from evaluators and choose the median value as the final score.

TABLE V
THE SCORE DISTRIBUTION OF GENERATED DESCRIPTIONS

Score	1	2	3	4	5	6	7	Avg
Seq2seq	16	18	17	16	11	10	12	3.66
GraphCodeBert	12	14	12	22	10	19	11	4.05
VulTeller	12	11	10	15	16	14	22	4.42

The detailed scores of generated descriptions are shown in Table V. We can see that our approach achieves the best results and improves the average (Avg) score from 3.66 (Seq2seq), 4.05 (GraphCodeBert) to 4.42. Specifically, among the randomly selected 100 samples, our approach can generate 22 highly similar or even identical descriptions with the reference ones (score = 7), 36 good descriptions (score $\geq$ 6). Our approach also receives the smallest number of negative results (score $\leq$ 3). Based on the 100 final scores for each approach of Seq2Seq, GraphCodeBert and VulTeller, we conduct Wilcoxon signed-rank tests [55]. Comparing our approach with two baselines, the p-values of Wilcoxon signed-rank tests at 95% confidence level are 2.5e-05 and 0.028 respectively, showing that the improvements achieved by our approach are statistically significant. To sum up, the results of human evaluation confirm the effectiveness of the proposed approach.

G. Case Study

We present two cases of vulnerable functions with their references and generated descriptions to discuss the superiority and limitation of our model for description generation, which is shown in Table VI. We add line comments "/*Vulnerable statement*/" to point out the location. In the first case, the function from MongoDB client aims to save history files on the local storage. However, it simply opens a specified file (i.e., "fopen()") and writes data to it without checking the permissions (i.e., "fprintf()"). This may lead to information leakage when the data has sensitive content. Therefore, the developer fixes it by adding conditional compilation statements for example "#if" and "#else" around the vulnerable statement. Among the generated descriptions of baselines, the most similar key phrase is "writing data to a temporary file". But it neither presents what the vulnerability is nor explains

TABLE VI
TWO CASES OF VULNERABLE C/C++ FUNCTIONS WITH THE DESCRIPTIONS. THE YELLOW HIGHLIGHTS ARE THE IMPORTANT INFORMATION FOR THE VULNERABILITY, WHILE PINK AND RED HIGHLIGHTS INDICATE THE CLOSE OR IDENTICAL MEANING TO THE REFERENCE RESPECTIVELY.

CASE 1: CVE-2016-6494	CASE 2: CVE-2015-6817

`int linenoiseHistorySave(const char* filename)`
`{`
`    /*Vunerable statement*/`
`    FILE* fp = fopen(filename, "wt");`
`    if (fp == NULL) {`
`        return -1;`
`    }`
`    for (int j = 0; j < historyLen; ++j) {`
`        if (history[j][0] != '\0') {`
`            fprintf(fp, "%s\n", history[j]);`
`        }`
`    }`
`    fclose(fp);`
`    return 0;`
`}`
	`static void start_auth_request(PgSocket *client`
	`    , const char *username)`
	`{`
	`    int res;`
	`    PktBuf *buf;`
	`    /*Vunerable statement*/`
	`    client->auth_user = client->db->auth_user;`
	`    client->pool = get_pool(client->db, client`
	`        ->db->auth_user);`
	`    if (!find_server(client)) {`
	`        client->wait_for_user_conn = true;`
	`        return;`
	`    }`
	`    ...(18 lines more)`
	` }`

Reference: The client in MongoDB uses world-readable permissions on .dbshell history files, which might allow local users to obtain sensitive information by reading these files. **Seq2seq**: Keepalived VERSION, did n't check for pathnames with symlinks when writing data to a temporary file upon a call to Printdata. **Transformer**: FILE in python before VERSION, has a length value that only might allow an attacker to dereference during the client 's unsigned long key. **CodeBert**: Keepalived VERSION, did n't check for pathnames with symlinks when writing data to a temporary file upon a call to PrintData or PrintStats. **G2S**: An issue was discovered in php before VERSION. There is an uninitialized read in files. **GraphCodeBert**: Integer overflow in FILE in Git, allows remote attackers to cause a denial of service (application crash) via a crafted file. **VulTeller**: The FILE before VERSION does not verify that use of the permissions on history files, allows local guest to read sensitive information by reading these files.	**Reference**: PgBouncer before VERIOSN, when configured with auth_user, allows remote attackers to gain login access as auth_user via an unknown username. **Seq2seq**: The filesystem implementation in the FileInfo component does not ensure that pix_fmt is used for a remote advertisement's application. **Transformer**: The in the linux the block devices in the linux kernel before VERSION, allows remote attackers to cause a denial of service. **CodeBert**: Unspecified vulnerability in libpng before VERSION, as used in Android VERSION, before VERSION, before VERSION...(twice more) **G2S**: An issue was discovered in libvncserver before VERSION, FILE has a null pointer dereference. **GraphCodeBert**: PgBouncer before VERSION, allows remote attackers to cause a denial of service (NULL pointer dereference and crash) via unspecified vectors. **VulTeller**: A crafted authentication, which can cause a denial of service (connection to crash) by sending a connection to client.

how it can be exploited. By contrast, VulTeller successfully generates the symptom and cause, which goes a long way towards understanding and fixing it. The reason behind this is that our approach learns precise dependencies and treats the missing control dependency (i.e., the condition evaluation) along with data dependency (i.e., from source “FILE *p” to sink “fprintf(fp, ...)”) as vulnerable.

Despite some progress, generating high-quality descriptions automatically remains a great challenge, and our approach has its own limitations. For example, in Case 2, almost all the methods failed to generate important information about the vulnerability related to malicious access control. GraphCodeBert correctly identified that the vulnerability originated from PgBouncer, but reported an unrelated symptom (i.e., “denial of service”). Similarly, our approach predicted the same symptom, though it did indicate the presence of a crafted authentication. This bias may have stemmed from the absence of external dependency information, such as the definition of the PgSocket class and its members. Additionally, our approach is insensitive to the value of a variable, which could lead to incorrect predictions. These findings motivate us to consider global dependency and dynamic information in our future work and also highlight the need for more research to address this challenging problem.

V. THREATS TO VALIDITY

Internal Validity. The internal threat is that the quality of the functions along with the ground-truths (i.e., locations and descriptions) may affect the effectiveness of our approach. In particular, we treat lines that have control/data dependencies on the added lines as vulnerable, which may not always be the truth. As introduced, this practice is borrowed from the baselines and thus the threat is minimized. Moreover, the original description of a vulnerability may contain meaningless words such as version numbers, and miss some details like the cause and impact. To mitigate it, we have applied some rules to normalize the versions and file paths. We will consider more accurate heuristics in the future.

External Validity. We only collected vulnerabilities from C/C++ projects to construct the dataset. It remains unknown whether our approach will perform well in other programming languages such as PHP and Python. However, it is widely known that C/C++ projects produce more vulnerabilities because of their freedom on accessing system memory and foundation for other programming languages. We will extend the diversity of our dataset in future. Besides, some of the baselines may not fit well on the tasks. Static analysis tools may not achieve optimal performance in vulnerability localization since they are limited to identifying vulnerable lines within individual functions. We will explore automatic compilation techniques to support the evaluation of more advanced tools. For description generation, G2S is implemented based on the parsed PDG instead of AST-based graph, which differs from it in code summarization. But since it is important to capture dependency information as prior work demonstrated, we believe such a practice is reasonable. CodeBert and GraphCodeBert are pre-trained on programming languages excluding C/C++, which is not perfect for our dataset. To reduce the threat, we fine-tune them on the dataset for better performance. We will investigate other large pre-trained models to improve them in the future.

VI. Related Work

A. DL-based Vulnerability Discovery

Many studies have proposed deep learning-based approaches for discovering vulnerabilities from the source code, including the granularity of function level and statement level. At the function level (aka vulnerability detection), various types of neural networks have been adopted to detect if a given function is vulnerable by automatically learning vulnerability features [7], [8], [9]. Though effective, these work is limited to only notify developers the existence of vulnerability, without the locations that are crucial for fixing it.

Therefore, recent work focuses more on the statement-level vulnerability detection (i.e., vulnerability localization), most of them fall into unsupervised learning. For example, VulDeeLocator [11] extends a Bi-LSTM detector with an inner multiply layer and predicts the vulnerable statements with the output of the inner layer. Similarly, IVDetect [12] and LineVul [56] train a vulnerability detector and then extract either subgraphs or attention weights from the trained model for locating vulnerable statements. Due to the inconsistency between inner features and the locations, the subsequent studies propose to pinpoint vulnerable statements in a supervised manner. VELVET [13] learns to rank vulnerable statements by combining graph-based and sequence-based neural networks. LineVD [14] formulates statement-level vulnerability detection as a node classification task and leverages graph neural networks with a transformer-based model over PDG for end-to-end learning.

Compared to the above work, our VulTeller is a fully supervised neural model for vulnerability localization and focuses on control flows and taint flows to capture more precise dependencies.

B. Code Document Generation

Inspired by the success of neural machine translation, researchers also widely explore encoder-decoder neural networks to generate documents for code snippets, including but not limited to source code summarization, [37], [47], [57], code commit generation [45], [58]. For example, Ahmad et al. [47] explores the Transformer model that uses a self-attention mechanism and is attached with relative position representations and copy attention to generate code comments. Jiang et al. [45] adopt the standard attentional sequence-to-sequence model that encodes the code changes and generates their commit messages. Apart from these tasks, Liu et al. [59] propose a sequence-to-sequence model with copy mechanism and reinforcement learning to automatically generate pull request descriptions based on the commit messages and the added source code comments in the pull-requests. Hu et al. [60] exploit the Transformer to generate user notice for smart contract functions, so as to help end-users better understand the smart contract and be aware of the financial risks.

Different from the ones outline above, we make the first step towards automated vulnerability description generation, which provides additional assistance for developers in understanding the facts and potential hazards of vulnerabilities.

VII. Conclusion

This paper introduces a novel approach to automatically localize and generate descriptions for vulnerabilities using deep learning. To address the challenge of capturing vulnerability semantics, we propose VulTeller that converts the control flow graph into control flow paths and incorporating taint analysis to refine control dependencies and integrate data dependencies, thereby avoiding the bottleneck of GNNs. We design neural models based on these techniques to automatically learn precise semantics for both tasks. We conduct extensive experiments to evaluate the effectiveness of our approach, and the results demonstrate the superior performance of our models in both tasks compared to baseline models. Despite these promising results, there is still space for improvement before the approach can be applied in real-world scenarios.

Our work can be considered as an initial step towards automated vulnerability diagnosis. In future, we expect researchers will explore more advanced strategies to improve the accuracy of both tasks. And we may also conduct studies to investigate more information need of developers on diagnosing vulnerabilities. The source code and experimental data are available at https://github.com/zhangj111/VulTeller.

Acknowledgment

This work is supported by Nanyang Technological University (NTU)-DESAY SV Research Program under Grant 2018-0980. It is also supported by the National Research Foundation, Singapore, and the Cyber Security Agency under its National Cybersecurity R&D Programme (NCRP25-P04-TAICeN) and the NRF Investigatorship NRF-NRFI06-2020-0001. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore and Cyber Security Agency of Singapore. This work has also received partial support from the National Key Research and Development Program of China (No. 2022YFB4502003) and the National Natural Science Foundation of China (No. 62072017).

References

[1] M. Shahzad, M. Z. Shafiq, and A. X. Liu, “A large scale exploratory analysis of software vulnerability life cycles,” in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 771–781.

[2] D. Mu, A. Cuevas, L. Yang, H. Hu, X. Xing, B. Mao, and G. Wang, “Understanding the reproducibility of crowd-reported security vulnerabilities,” in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 919–936.

[3] M. Fu, C. Tantithamthavorn, T. Le, V. Nguyen, and D. Phung, “Vulrepair: a t5-based automated software vulnerability repair,” in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 935–947.

[4] J. Smith, B. Johnson, E. Murphy-Hill, B. Chu, and H. R. Lipford, “Questions developers ask while diagnosing potential security vulnerabilities with static analysis,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 248–259.

[5] R. Croft, D. Newlands, Z. Chen, and M. A. Babar, “An empirical study of rule-based and learning-based approaches for static application security testing,” in *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2021, pp. 1–12.

[6] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, “Why don’t software developers use static analysis tools to find bugs?” in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 672–681.

[7] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, “Vuldeepecker: A deep learning-based system for vulnerability detection,” in *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, 2018.

[8] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, “Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks,” *Advances in neural information processing systems*, vol. 32, 2019.

[9] Y. Wu, D. Zou, S. Dou, W. Yang, D. Xu, and H. Jin, “Vulcnn: An image-inspired scalable vulnerability detection system,” in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 2365–2376.

[10] J. Smith, B. Johnson, E. Murphy-Hill, B. Chu, and H. R. Lipford, “How developers diagnose potential security vulnerabilities with a static analysis tool,” *IEEE Transactions on Software Engineering*, vol. 45, no. 9, pp. 877–897, 2018.

[11] Z. Li, D. Zou, S. Xu, Z. Chen, Y. Zhu, and H. Jin, “Vuldeelocator: a deep learning-based fine-grained vulnerability detector,” *IEEE Transactions on Dependable and Secure Computing*, 2021.

[12] Y. Li, S. Wang, and T. N. Nguyen, “Vulnerability detection with fine-grained interpretations,” in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 292–303.

[13] Y. Ding, S. Suneja, Y. Zheng, J. Laredo, A. Morari, G. E. Kaiser, and B. Ray, “VELVET: a novel ensemble learning approach to automatically locate vulnerable statements,” in *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022, Honolulu, HI, USA, March 15-18, 2022*. IEEE, 2022, pp. 959–970.

[14] D. Hin, A. Kan, H. Chen, and M. A. Babar, “Linevd: Statement-level vulnerability detection using graph neural networks,” in *19th IEEE/ACM International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022*. ACM, 2022, pp. 596–607.

[15] U. Alon and E. Yahav, “On the bottleneck of graph neural networks and its practical implications,” in *International Conference on Learning Representations*, 2020.

[16] S. Liu, X. Xie, J. Siow, L. Ma, G. Meng, and Y. Liu, “Graphsearchnet: Enhancing gnns via capturing global dependencies for semantic code search,” *IEEE Transactions on Software Engineering*, 2023.

[17] J. Zhang, X. Wang, H. Zhang, H. Sun, X. Liu, C. Hu, and Y. Liu, “Detecting condition-related bugs with control flow graph neural network,” in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 1370–1382.

[18] H. Assal and S. Chiasson, “’think secure from the beginning’ a survey with software developers,” in *Proceedings of the 2019 CHI conference on human factors in computing systems*, 2019, pp. 1–13.

[19] D. Votipka, K. R. Fulton, J. Parker, M. Hou, M. L. Mazurek, and M. Hicks, “Understanding security mistakes developers make: Qualitative analysis from build it, break it, fix it,” in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 109–126.

[20] J. Newsome and D. X. Song, “Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software.” in *NDSS*, vol. 5. Citeseer, 2005, pp. 3–4.

[21] M. Schuster and K. K. Paliwal, “Bidirectional recurrent neural networks,” *IEEE transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, 1997.

[22] K. Cho, B. van Merrienboer, C. Gulcehre, F. Bougares, H. Schwenk, and Y. Bengio, “Learning phrase representations using rnn encoder-decoder for statistical machine translation,” in *Conference on Empirical Methods in Natural Language Processing (EMNLP 2014)*, 2014.

[23] J. K. Siow, S. Liu, X. Xie, G. Meng, and Y. Liu, “Learning program semantics with code representations: An empirical study,” in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 554–565.

[24] T. N. Kipf and M. Welling, “Semi-supervised classification with graph convolutional networks,” in *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017.

[25] Y. Li, D. Tarlow, M. Brockschmidt, and R. S. Zemel, “Gated graph sequence neural networks,” in *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2016.

[26] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, “Graph attention networks,” in *International Conference on Learning Representations*, 2018.

[27] Joern, “Joern: The bug hunter’s workbench,” https://joern.io/.

[28] C. Flanagan and J. B. Saxe, “Avoiding exponential explosion: Generating compact verification conditions,” in *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, 2001, pp. 193–205.

[29] J. Zhang, X. Wang, H. Zhang, H. Sun, Y. Pu, and X. Liu, “Learning to handle exceptions,” in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 29–41.

[30] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, “Modeling and discovering vulnerabilities with code property graphs,” in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 590–604.

[31] E. J. Schwartz, T. Avgerinos, and D. Brumley, “All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask),” in *2010 IEEE symposium on Security and privacy*. IEEE, 2010, pp. 317–331.

[32] Y. W. Chow, M. Schäfer, and M. Pradel, “Beware of the unexpected: Bimodal taint analysis,” *arXiv preprint arXiv:2301.10545*, 2023.

[33] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck, “Chucky: Exposing missing checks in source code for vulnerability discovery,” in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 499–510.

[34] K. Cho, B. van Merriënboer, D. Bahdanau, and Y. Bengio, “On the properties of neural machine translation: Encoder–decoder approaches,” in *8th Workshop on Syntax, Semantics and Structure in Statistical Translation, SSST 2014*. Association for Computational Linguistics (ACL), 2014, pp. 103–111.

[35] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[36] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, “A novel neural source code representation based on abstract syntax tree,” in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 783–794.

[37] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, “Deep code comment generation,” in *Proceedings of the 26th conference on program comprehension*, 2018, pp. 200–210.

[38] D. Bahdanau, K. H. Cho, and Y. Bengio, “Neural machine translation by jointly learning to align and translate,” in *3rd International Conference on Learning Representations, ICLR 2015*, 2015.

[39] M.-T. Luong, H. Pham, and C. D. Manning, “Effective approaches to attention-based neural machine translation,” in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, 2015, pp. 1412–1421.

[40] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, “Ac/c++ code vulnerability dataset with code changes and cve summaries,” in *Proceedings of the*

17th International Conference on Mining Software Repositories, 2020, pp. 508–512.

[41] S. Lipp, S. Banescu, and A. Pretschner, “An empirical study on the effectiveness of static c code analyzers for vulnerability detection,” in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 544–555.

[42] H. Liu, S. Chen, R. Feng, C. Liu, K. Li, Z. Xu, L. Nie, Y. Liu, and Y. Chen, “A comprehensive study on quality assurance tools for java,” in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2023, 2023, p. 285–297.

[43] Facebook, “Infer: A tool to detect bugs in java and c/c++/objective-c code,” https://fbinfer.com/.

[44] G. S. Lab, “Codeql,” https://securitylab.github.com/tools/codeql.

[45] S. Jiang, A. Armaly, and C. McMillan, “Automatically generating commit messages from diffs using neural machine translation,” in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 135–146.

[46] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, “Intellicode compose: Code generation using transformer,” in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1433–1443.

[47] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, “A transformer-based approach for source code summarization,” in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 2020, pp. 4998–5007.

[48] A. LeClair, S. Haque, L. Wu, and C. McMillan, “Improved code summarization via a graph neural network,” in *Proceedings of the 28th international conference on program comprehension*, 2020, pp. 184–195.

[49] Y. Wang, Y. Dong, X. Lu, and A. Zhou, “Gypsum: learning hybrid representations for code summarization,” in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, 2022, pp. 12–23.

[50] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, “Codebert: A pre-trained model for programming and natural languages,” in *Findings of the Association for Computational Linguistics: EMNLP 2020*, 2020, pp. 1536–1547.

[51] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. B. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, “Graphcodebert: Pre-training code representations with data flow,” in *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*, 2021.

[52] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, “Bleu: a method for automatic evaluation of machine translation,” in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

[53] C.-Y. Lin, “Rouge: A package for automatic evaluation of summaries,” in *Text summarization branches out*, 2004, pp. 74–81.

[54] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, “Language models are few-shot learners,” *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[55] F. Wilcoxon, S. Katti, and R. A. Wilcox, “Critical values and probability levels for the wilcoxon rank sum test and the wilcoxon signed rank test,” *Selected tables in mathematical statistics*, vol. 1, pp. 171–259, 1970.

[56] M. Fu and C. Tantithamthavorn, “Linevul: a transformer-based line-level vulnerability prediction,” in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 608–620.

[57] J. Zhang, X. Wang, H. Zhang, H. Sun, and X. Liu, “Retrieval-based neural source code summarization,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1385–1397.

[58] Z. Liu, X. Xia, A. E. Hassan, D. Lo, Z. Xing, and X. Wang, “Neural-machine-translation-based commit message generation: how far are we?” in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 373–384.

[59] Z. Liu, X. Xia, C. Treude, D. Lo, and S. Li, “Automatic generation of pull request descriptions,” in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 176–188.

[60] X. Hu, Z. Gao, X. Xia, D. Lo, and X. Yang, “Automating user notice generation for smart contract functions,” in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 5–17.