SEVulDet: A Semantics-Enhanced Learnable Vulnerability Detector

Zhiquan Tang*, Qiao Hu*(✉), Yupeng Hu*(✉), Wenxin Kuang*, Jiongyi Chen†
*College of Computer Science and Electronic Engineering, Hunan University, Changsha, China
Email:{zqtang, huqiao, yphu, wenxinkuang}@hnu.edu.cn
†College of Electronic Science and Engineering, National University of Defense Technology, Changsha, China
Email: chenjiongyi@nudt.edu.cn

***Abstract*— Recent years have seen increased attention to deep learning-based vulnerability detection frameworks that leverage neural networks to identify vulnerability patterns. Considerable efforts have been made; still, existing approaches are less accurate in practice. Prior works fail to comprehensively capture semantics from source code or adopt the appropriate design of neural networks. This paper presents SEVulDet, a Semantics-Enhanced learnable Vulnerability Detector that can accurately pinpoint vulnerability patterns by preserving path semantics into gadgets and learning from flexible-length codes. SEVulDet has two main characteristics: (i) SEVulDet employs a path-sensitive code slicing approach to extract sufficient path semantics and control flow logic into code gadgets. (ii) by inserting a spatial pyramidal pooling layer into the Convolutional Neural Network (CNN) with a well-designed multilayer attention mechanism, SEVulDet can handle gadgets of flexible-length semantics to avoid semantics loss incurred by traditional truncating or padding operations, and thus learn more potential vulnerability patterns. Comprehensive experimental results show that SEVulDet significantly outperforms classical static approaches and excels with state-of-the-art deep learning-based solutions by improving F1-measure to roughly 94.5%. Particularly, the elaborate design of the SEVulDet architecture helps us identify more real-world vulnerabilities than existing technologies.**

***Index Terms*—vulnerability detection, deep learning, semantics, program analysis, program representation.**

I. Introduction

Software vulnerabilities are prevailing in cyberspace and prompt various system attacks [1], [2] and data breaches [3]. The open-source code makes the vulnerabilities even more widespread. Classic static detection techniques [4]–[7] which are less dependent on the running environment of the target have been widely adopted in practice. However, they also have the primary drawback: relying on experts to define vulnerability characteristics, which requires intense manual labor but has unstable effectiveness.

Hereby deep learning-based detection frameworks have attracted a wealth of research efforts, which reduce both time cost and required expert knowledge by recognizing vulnerability patterns [8]–[12]. Nevertheless, with recent progress made in this line of research, learning more code semantics from source codes remains a major obstacle to making deep learning-based frameworks more practical.

(✉): Corresponding authors

One reason is the challenges in code preprocessing algorithms connecting code fragments to path semantics. Recently, researchers have preferred the code gadget slicing approach, which divides the program into groups of interrelated statements to extract as much semantics as possible (*not only the correct code structures but the thorough logic of statements [13]*). Considering that vulnerabilities are not limited to a single function or file, slicing at the fine-grained program level is a preferred solution for preprocessing [9]–[12]. However, we notice that existing code gadget fragments are simply stacks of dependent statements that lack proper contextual path relationships between statements and more details on dependencies, which causes semantics loss. Divergent source codes can even yield the same code gadget fragments because of incomplete path information. Whatever the detection result, the detector will always misjudge some source codes corresponding to semantically insufficient fragments.

The other reason for the difficulty in deeply learning semantics is the lack of proper network architecture to learning semantic features from code fragments. Recurrent Neural Networks (RNNs), which excel at learning contextual information, are widely used for vulnerabilities detection. It can help find some vulnerabilities, but its predefined time steps that fix the length of the input sequence make it not a satisfactory solution. For over-length codes, fixed-length may result in discarding a portion of the code gadget with critical semantics; for under-length codes, fixed-length may introduce irrelevant semantics because of the padding bits. Both cases can affect how the network operates in terms of performance and accuracy [14]. Thus, network structures that drop and scramble semantics are very tricky to confront with real-world software.

To overcome these obstacles, we propose a path-sensitive gadget generating algorithm and a CNN with a spatial pyramid pooling layer which can preserve entire semantic information in the gadgets. Then we make a further step on utilizing a multilayer attention mechanism to help the CNN analyze and learn semantic information better.

The main contributions are as follows.

First, we propose a code semantic enhancement algorithm based on path-sensitive program slicing method. Concretely, an extra step is added to the code gadget slicing method to ensure that more path semantics can be retained during

2158-3927/22/$31.00 ©2022 IEEE
DOI 10.1109/DSN53405.2022.00026

the slicing process. The algorithm has been implemented and evaluated in the Software Assurance Reference Dataset (SARD) project [15], the National Vulnerability Dataset (NVD) project [16], and real-world software.

Second, a high-precision network architecture is proposed to eliminate the limitation of RNNs on code length. Specifically, CNN with a spatial pyramid pooling layer is tailored for code vulnerability detection of flexible length, which addresses the semantic loss problem on the network. In addition, a multilayer attention mechanism is carefully designed to capture the bounded hierarchical structure of source codes, so as to identify relatively interesting tokens and crucial semantics for high-precision vulnerability detection.

Finally, we implement the detection framework named SEVULDET[1] (Semantics-Enhanced learnable Vulnerability Detector) based on the above designs and conduct extensive experiments to demonstrate effectiveness of the framework. Compared with state-of-the-art solutions, SEVULDET improves F1-measure up to 94.5%. Furthermore, SEVULDET successfully identified one previously unreported vulnerability on Xen software products.

The rest of the paper is organized as follows. Section II presents the background and motivation. Section III discusses the design of SEVULDET. Section IV shows our experimental evaluation of SEVULDET and the results. Section V describes the related work. Finally, Section VI discusses the conclusions of SEVULDET.

II. Background and Motivation

In this section, we first provide some preliminary guiding principles for deep learning networks based vulnerability detection frameworks. Then we illustrate an example showing the drawbacks of existing frameworks, inspiring us to redesign the solution. Finally, we conclude the main drawback, semantic loss, of existing vulnerability detection frameworks that negatively impact on vulnerability detection.

A. Definitions

We declare the following definitions for precise description.

Definition 1 (program, statement, token): A program P consists of a series of ordered statements $s_1, ..., s_\epsilon$, which denoted by $P = \{s_1, ..., s_\epsilon\}$. A statement s_w ($1 \leqslant w \leqslant \epsilon$) consists of a series of ordered tokens $t_{w_1}, ..., t_{w_m}$, which denoted by $s_w = \{t_{w_1}, ..., t_{w_m}\}$. A token t_{w_η} ($1 \leqslant \eta \leqslant m$) is a particular word in s_w which could be a function identifier, variable identifier, constant, operator, keyword, etc.

Definition 2 (data-dependence [17]): For a program $P = \{s_1, ..., s_\epsilon\}$, two statements s_α,s_β ($\alpha \neq \beta$) in P, and a variable identifier token t_i in s_α, statement s_β is said to be data-dependence on s_α when t_i is also used in s_β.

Definition 3 (control-dependence [17]): For two statements s_α,s_β ($\alpha \neq \beta$) in a program $P = \{s_1, ..., s_\epsilon\}$, statement s_β is said to has control dependence with s_α when the execution outcome of s_α affects whether s_β will be executed or not.

[1]https://github.com/SEVulDet/SEVulDet

Definition 4 (special token): Given a program $P = \{s_1, ..., s_\epsilon\}$, and a statement s_w ($1 \leqslant w \leqslant \epsilon$) composed of m special tokens $\{t_{w_1}, ..., t_{w_m}\}$, the special token t_{w_n} ($1 \leqslant n \leqslant m$) is a token that matches the syntactic characteristics of library functions calls, arrays usage, pointers usage, and expression in statement s_w.

Definition 5 (code gadget [9]): Given a program $P = \{s_1, ..., s_\epsilon\}$, and a statement s_w ($1 \leqslant w \leqslant \epsilon$) composed of m special tokens $\{t_{w_1}, ..., t_{w_m}\}$, a code gadget C_{w_η} ($1 \leqslant \eta \leqslant m$) generated from the token t_{w_η} consists of multiple ordered statements that have recursive data-dependence or control-dependence on s_w.

Definition 6 (PDG [17]): Given a program $P = \{s_1, ..., s_\epsilon\}$, and a statement s_w ($1 \leqslant w \leqslant \epsilon$) containing m library/API function calls $\{f_{w_1}, ..., f_{w_m}\}$, a Program Dependency Graphs (PDG) corresponding to function call f_{w_η} is a directed graph $G_{w_\eta} = (V_{w_\eta}, E_{w_\eta})$, where V_{w_η} is a set of statements and control predicates and E_{w_η} is a set of direct edges that represent the dependency.

B. Preliminaries

The source code contains rich semantics. But parts of the vulnerability semantics will be lost if we translate the source code into LLVM IR [18]. We therefore choose optimization algorithms in neural networks (e.g. gradient descent) to adapt the network parameters and learn potentially vulnerable features from the source code dataset. Recent efforts [13] have shown that this technique can effectively recognize vulnerability patterns from source code with the help of two critical procedures: code preprocessing and network constructing.

The purpose of code preprocessing is to comprehensively extract the semantics of vulnerability patterns, which can also help the network learn critical code. The commonly employed method is the program analysis technique, which abstracts the necessary syntaxes and semantics from the program. Previous methods processed source code into several fragments by preprocessing, such as files [19], functions [8], git commits [20], and code gadgets [9] assembled from interdependent statements. The minimum granularity for detecting vulnerabilities is the corresponding fragment.

An underlying principle for constructing a deep learning network suitable for code vulnerability detection is the ability of the network to learn contextual semantic features from code fragments. RNNs can remember the information they have processed and use it to influence future decisions [21], becoming a popular frameworks for vulnerability detection.

In addition, the attention mechanism enables the neural networks to pay more attention to the input subsets (or features) to reduce the reliance on external information, helping capture the internal correlation of inputs [22]. Recent works [23], [24] have theoretically demonstrated the advantages of attention networks over RNNs on bounded hierarchical language processing. Hereby we intend to design attention mechanisms to capture essential tokens and their combinatorial patterns in the source codes, and further visualize the weights to analyze the rationale for the inference of the network [25].

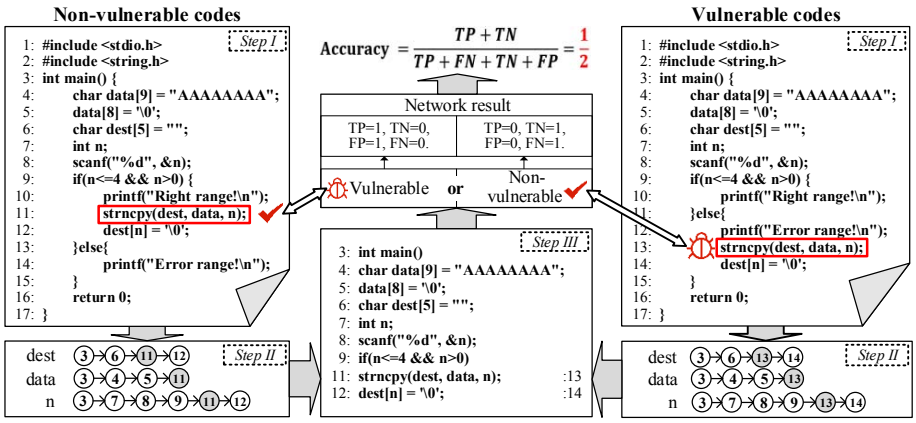

Fig. 1. Using the existing code gadget generating method, correct codes at left and vulnerable codes at right will extract identical code gadgets (Step III). Whether the detection result for these two cases is vulnerable or non-vulnerable, the accuracy keeps equal to 0.5, which is useless for vulnerability detection.

C. Motivating Example

Although code gadget has been traditionally used in learnable vulnerability detection detectors [9], [10], it has a critical problem - *path insensitive*. An example in Fig. 1 depicts the drawback of program processing in previous works. The entire detection framework input is a series of source codes with and without vulnerability labels. The highlighted function $strncpy$ in Fig. 1 is one of the library functions in the C standard library that copies a string of a specified length to the character array.

All three dependents tokens of $strncpy$ (i.e., $dest$, $data$ and n) are extracted with their forward and backward data-dependence statements (Step II of Fig. 1), forming multiple forward and backward slices [9], e.g., $dest$ 3→6→11→12. Subsequently, all statements are broken up and reorganized according to both the function where they are located and the calling relationship between the functions (Step III of Fig. 1).

Notably, the statements in the source code on the left of Fig. 1 are not vulnerable. It is obvious that the size of n is judged before the function $strncpy$ is called, and the $strncpy$ is dependent on the valid (i.e., not any) size of n, which would prevent array index out of bounds error. But another code in the right of Fig. 1, where the n used to address items in $dest$ exceeds the allowed value, is obviously vulnerable. As shown in Fig.1, the slicing methods in previous works generate identical code gadgets at Step III from both non-vulnerable and vulnerable codes. Identical code gadgets mean identical network classification results. In the case where the model classification result is vulnerable, $TP = 1$, $TN = 0$, $FP = 1$, $FN = 0$. In the case where the model classification result is non-vulnerable, $TP = 0$, $TN = 1$, $FP = 0$, $FN = 1$. So when the network classify codes similar to the example, the accuracy would always be 50% which has no contribute to vulnerability detection.

D. Semantic Loss

Path-insensitive code gadget. A path transformation of a statement causes changes in the control range in which the statement is located, but code gadgets in existing frameworks do not track them. The reasons are in the following aspects: (i) control-dependence is a rough description of the relationship between two statements (i.e., whether there is a dependency) and does not indicate the path to the statement (i.e., dependent on legal or illegal values); (ii) Step III is a process of reorganizing the order of statements, where brute stacking may cause statements that are not in the same control range to be directly adjacent to each other, thus creating path insensitive. One solution is to identify the control range of each control statement, match and record all the ranges that can be passed to the statements in slices, and thus save the positive or negative dependencies between statements into the path-sensitive code gadgets.

Definition 7 (path-sensitive code gadget): Given a program $P = \{s_1, ..., s_\epsilon\}$, and a statements s_w $(1 \leqslant w \leqslant \epsilon)$ composed of m special tokens $\{t_{w_1}, ..., t_{w_m}\}$, a path-sensitive code gadget C_{w_η} $(1 \leqslant \eta \leqslant m)$ generated from token t_{w_η} consists of multiple statements along with call sequence. Each of these statements either recursively depends on s_w or is a control scope statement on the path to t_{w_η}.

Fixed token length. RNNs are the most popular vulnerability detection models because they can handle context and text classification. However, they process the input brute force: deletion or padding. The time steps of RNNs are predefined, and only one token can be fed to each time step. Hence the length of token contained in a code gadget must be predefined and fixed. The vector is padded with zeros during the embedding phase if tokens are not long enough or truncated if tokens are longer than the predetermined length. The cropped region may contain part of a bug semantics, and the padded region may import distortion [14]. A big enough predefined length may solve the interference of short sequence padding, but not the truncation of ultra-long real software codes as overlong input brings significant time and energy costs. Avoiding dropping extra-long data with a sizeable predefined length would make the standard length data incur extra overhead and interference.

Definition 8 (fixed-length code gadget): For an RNN-

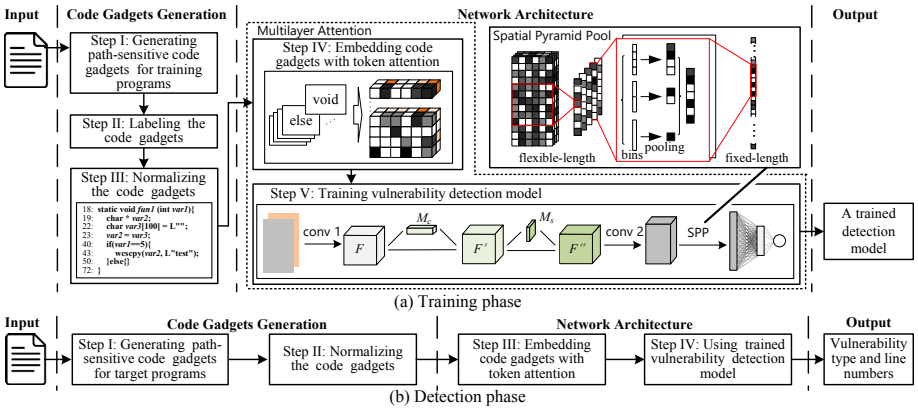

Fig. 2. Overview of SEVULDET. (a) Five steps to generate a trained model in the training phase. The details of slicing algorithm in Step I are shown in Fig. 3. The improved spatial pyramid pooling (SPP) designed to handle flexible-length codes. The colored parts in Steps IV and V represent the weight assignment of the multilayer attention mechanism. (b) Four steps to output vulnerability type and line number (if exists) in the detection phase.

based detection network R_τ with time step τ, given a code gadget C containing υ tokens, which denoted by $C = \{t^1, t^2, ..., t^\upsilon\}$, a fixed-length code gadget C_f generated by C can be denoted by

$$C_f = \begin{cases} \{t^1, t^2, ..., t^\tau\} & \tau <= \upsilon; \\ \{t^1, t^2, ..., t^\upsilon, 0^{\upsilon+1}, ..., 0^\tau\} & \tau > \upsilon. \end{cases}$$

III. DESIGN OF SEVULDET

This section describes the design of SEVULDET, a deep learning-based detector with a path-sensitive code gadget generating algorithm and a neural network carefully designed for high-accuracy detection of codes of flexible length. As shown in Fig. 2, SEVULDET can be divided into two phases: the training phase and the detection phase. At the training phase, steps I through III save the necessary semantics and syntax from the source code into path-sensitive code gadgets, while steps IV and V learn potential vulnerability patterns. Similarly, the detection phase is the same as the training phase, except there is no step to label the code gadgets.

A. Technical Challenges

SEVULDET can semantically enhance the detection framework from both code gadget processing and network design perspectives to eliminate the semantic loss problem. However, SEVULDET needs to address the following three thorny technical challenges.

The first challenge is encapsulating the correct path information into the code gadget. The codes are divided into blocks of statements by line breaks and curly brackets, and the execution path is a long chain of successively executed statements across these blocks. Therefore, SEVULDET needs to collect all possible path semantics and select the necessary paths to merge into the code gadgets at the right places to preserve the control dependencies of the scope and statements (i.e., statements in or out of range) into the code gadgets further.

The other two challenges come from designing network structures that preserve and learn semantics in path-sensitive code gadgets as much as possible. Firstly, unlike the recent advances in natural language processing [26] and computer vision [27] that limit the input scale, we dedicate to replacing the ordinary pooling layer in CNNs with the spatial pyramid pooling to meet the requirements of flexible length for semantics preserving. Secondly, concerning the limits in size and depth of convolutional kernels, we need to improve CNNs to enhance the semantic learning capabilities of code gadgets.

The implementation also faces some obstacles to be overcome. First, we must address the semantics loss of previous code gadgets-based works [9]–[11] by integrating the path-sensitive algorithm and customized network structure. Meanwhile, the prototype system needs to be low-coupling to ensure detection performance.

B. Path-sensitive Code Gadgets Generation

Existing gadgets generation methods are prone to semantic loss, the procedure below extracts fine-grained path-sensitive code gadgets involving rich semantic for a special token.

Step I: Generating path-sensitive code gadgets for the corresponding function calls, expressions, arrays, and pointers. Vulnerabilities often exhibit some syntactic features in common, and SEVULDET focuses on vulnerabilities arising from function calls, expressions, arrays, and pointers. For more clarity, we have split Step I into four sub-steps in Fig. 3.

a) Step I.1: Generating Program Dependence Graphs: We generate PDGs based on data-dependence and control-dependence, facilitating the generation of path-sensitive code gadgets with dependencies. Existing work [28] has given the standard algorithm for generating PDGs, and we use the open-source C/C++ code analysis tool Joern [29] to implement it. The second column in Fig. 3 is the PDG derived from the function in the sample program. We display the statement corresponding to each node with the line number.

b) Step I.2: Identify four types of special tokens: Based on manual inspection rules in Checkmark [4], SySeVR [11] has designed four common types of vulnerabilities: library/API function call, array usage, and pointer usage. We identify four

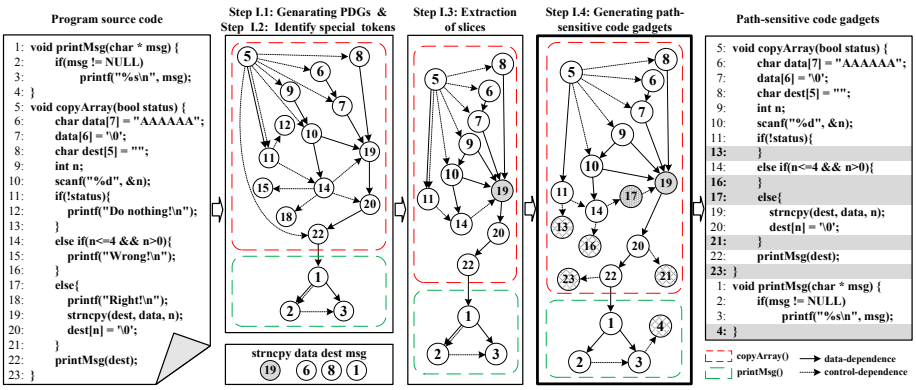

Fig. 3. A path-sensitive code gadget generation process for strncpy (Node 19). Step I.4 is the critical step to abstract the path semantics. Node 17 is inserted between 14 and 19, capturing a more detailed dependence. Nodes 4, 13, 16, 21, and 23 are saved as the endpoints of the path to the block for logical integrity.

special tokens (library/API function call (FC), array usage (AU), pointer usage (PU), and arithmetic expression (AE)) listed in [11].

c) Step I.3: Extraction of the forward and backward slices [10]: Since the vulnerability is a combination of several dependent statements in the context, We consider extracting the corresponding forward and backward slices for each special token of PDG. One-way slices may lead to semantic ambiguity or loss. The forward and backward slices are obtained by abstracting the successors and precursors of the special tokens in PDGs, respectively. It is worth pointing out that data and control-dependence are used to generate slices for two reasons: i) to find statements that are vulnerable to attacks via data-dependence; and ii) to enrich the semantic information via control-dependence, which in most cases can mitigate the accuracy degradation caused by missing semantic. The formal description of this step can be found at [10], and we open-sourced the implementation.

d) Step I.4: Generating path-sensitive code gadgets: Code gadget is the miniature set of semantics in which each statement has dependencies. As indicated in the motivating example, reconstructing statements in a stacked manner in the absence of path information may result in two control scopes not semantically linked overlapping. Furthermore, The control-dependence between two statements is only a rough description since the details and scope of the dependence are not captured. Therefore, Step I.4 relies on a novel Algorithm 1 for path semantics extraction, resolving the problem of semantic loss and accuracy reduction.

To identify all the path information, there are eight types of control statements whose control range can be clearly identified by algorithm 1: if, else if, else, for, while, do while, switch and case. We define the nodes that match the eight syntactic features as key nodes since there is a precise control range starting here. We observe that the paths of jump statements (e.g., "goto" or "setjmp/longjmp") are path-sensitive and can be represented by the forward and backward slices, so these statements are not included in the key nodes. The control range through which all dependencies in the forward and backward slices pass is selected and saved so that the path information in the code gadget becomes clear and there is no overlap between the two control blocks due to stacking. For example, lines 14 to 16 of the program in Fig. 2 are the control ranges corresponding to $elseif$, and lines 17 to 21 are the control ranges corresponding to $else$. Since the latter contains the 19th statement, if lines 17 and 21 are not kept in the code gadget, the execution path of $strncpy$ will be unclear, and the scope of $elseif$ and $else$ will overlap vaguely.

Here we list the implementation details of Algorithm 1 for path-sensitive code gadgets generation: a) generating the corresponding abstract syntax tree for the program of interest (Line 4) and identifying the key nodes that match the eight syntax characteristics (Line 8); b) calculating the maximum and minimum of row numbers in the subtree rooted by a key node (Line 7-8); c) binding adjacent control ranges with semantic relevance (Line 9-11) in special cases (e.g., if $elseif$ and $else$, or $switch$ and $case$); d) fixing the wrong correspondence between the start and end nodes of the control range with the help of the stack (Line 15-18); e) fine-tuning the dependencies in the slices by inserting the control ranges crossed by nodes in the slices and the bound control ranges into the slices (Line 19-23); f) adjusting statement relationships within functions according to line numbers and between functions based on call relationships (Line 25-36).

Fig. 3 showcases how algorithm 1 works in Step I.4. The if, $elseif$, and $else$ in the code are captured as key nodes and their control ranges are identified correspondingly. Since node 19 is within the range corresponding to $else$, the ranges of all three nodes are preserved. Specifically, two cases arise: nodes such as 4, 13, 16, 21, and 23 are inserted into the graph as leaf nodes to complete the logic of the block; nodes such as 17 need to be inserted in the middle of two nodes to act as control dependency details. These improvements can clearly identify the path to node 19 to resolve the problem of statements depending on legal or illegal data in Fig. 1. Finally, all nodes need to be adjusted at the function level to generate the path-sensitive code gadget. It is worth note that the key nodes here can be generalized to the eight control statements.

Algorithm 1 Generating path-sensitive code gadgets (Step 1.4)

Input: A program $P = \{s_1, ..., s_\epsilon\}$; A syntax characteristics set for eight control statements, $Z = \{z_{if}, z_{elseif}, z_{else}, z_{for}, z_{while}, z_{dowhile}, z_{switch}, z_{case}\}$; A statement in program P, s_w; A token generated by s_w, t_{w_η}; The slices corresponding to t_{w_η} , L_{w_η};
Output: The code gadget corresponding to t_{w_η}, C_{w_η};

$F \leftarrow$ dividing P into a set of functions
for each function $f_i \in F$ **do**
  $M_{w_\eta} \leftarrow \emptyset$
  $A_i \leftarrow AbstractSyntaxTree(f_i)$
  for each token $t_j \in A_i$ **do**
    if t_j matches z_k in Z **then**
      $a_{ij} \leftarrow$ subtree of A_i with t_j as root node
      $m_{cur} \leftarrow$ {minimum in a_{ij}, maximum in a_{ij}}
      if t_j matches z_{elseif} or z_{else} or z_{case} **then**
        Binding m_{cur} and m_{pre}
      end if
      $M_{w_\eta} \leftarrow M_{w_\eta} \cup m_{cur}$
    end if
  end for
  $M_{st} \leftarrow$ symbolic match via Stack
  for control ranges $m_a \in M_{w_\eta}$, $m_b \in M_{st}$ and $m_a[0] = m_b[0]$ **do**
    $m_a[1] \leftarrow Max(m_a[1], m_b[1])$
  end for
  for control range $m_q \in M_{w_\eta}$ **do**
    if there is a statement in slices L_{w_η} in m_q **then**
      $L_{w_\eta} \leftarrow L_{w_\eta} \cup m_q$
    end if
  end for
  $\zeta_{w_\eta,i} \leftarrow \emptyset$
  for statements s_λ, s_μ appearing in L_{w_η} **do**
    if s_μ is a successor node to s_λ or line number of s_μ is less than s_λ **then**
      $\zeta_{w_\eta,i} \leftarrow \zeta_{w_\eta,i} \cup \{s_\lambda, s_\mu\}$
    end if
  end for
end for
$C_{w_\eta} \leftarrow \emptyset$
for functions f_υ, f_ω appearing in L_{w_η} **do**
  if f_υ calls f_ω **then**
    $C_{w_\eta} \leftarrow C_{w_\eta} \cup \{\zeta_{w_\eta,\upsilon}, \zeta_{w_\eta,\omega}\}$
  end if
end for
return C_{w_η}

Step II: Labeling the code gadgets. SEVULDET is a kind of supervised learning, we need to obtain the source codes with tags from the open-source datasets (SARD [15] and NVD [16]) and label the corresponding code gadgets. Specifically, a code gadget, heuristically generated from a vulnerable code, is automatically marked as 1, potentially tagging some code gadgets incorrectly. This is caused by the invulnerable statements being the same as the vulnerable statements. We use the k-fold cross-validation [30] to narrow down the check range and then relabel it after manual judgment. Specifically, the code gadget set is randomly divided into k groups. Training and detection are repeated k times. During each time, one group is selected as validation data for testing while the other $k-1$ groups are used for training. Samples with multiple false-positive classifications need to be tested artificially for the correctness of labels.

Step III: Normalizing the code gadgets. Custom function and variable names do not affect the determination and genesis of vulnerabilities and increase the burden of model learning. Therefore, tokens that contain contextual semantic features are embedded as vectors, and tokens that are susceptible to programming conventions and other factors, such as function and variable names, are normalized. Step III in Fig. 2(a) shows that we rename a variable name or function name in the program in a mapping style to a name in an ordered set (i.e., {"$var1$", "$var2$", ...} and {"$fun1$", "$fun2$", ...}). Subsequently, we remove the non-ASCII characters and leave the macros, library/API function names, and constants intact.

C. Network Architecture Design

Latent vulnerabilities are generally associated with essential tokens and their specific combinatorial patterns, e.g., the case of Use-After-Free (UAF) vulnerability is induced by some free pointers followed by assignments. Moreover, the predefined time steps in the popular RNNs limit the length of semantics that the network intends to learn and detect. Therefore, we design a multilayer attention mechanism and a spatial pyramid pooling (SPP) layer in the convolutional neural network structure. The CNN with the SPP eliminates predefined length restrictions of the code gadget in RNN-based frameworks. *Thanks to the attention network's advantage of dealing boundary layered languages [24], we carefully design the multilayer attention mechanism for in-depth code learning via capturing essential tokens and their combinatorial patterns in the source codes.* Note that the attention mentioned in this paper is an adaptive deep neural network structure, which is different from code attention [10], which is a collection of statements that match expert-defined syntax characteristics.

Generally, the construction of the vulnerability detection network consists of steps IV and V. In step IV, we use the token attention to obtain the importance of tokens and assign weights. In step V, we employ the attention mechanism on both spatial and channel to capture the combined patterns of tokens. The design of SPP successfully lifted the limitation of the fixed network structure on the length of the code gadget.

Step IV: Embedding code gadgets with token attention. Each path-sensitive code gadget needs to be encoded into a vector via its symbolic representations. The primary challenge the token embedding has to overcome is to identify the tokens that contribute most to vulnerability detection since not every token is equally essential to a vulnerability.

Hereby we use the pre-trained word2vec [31] model to embed tokens as vectors and design token attention to learn which tokens are helpful for detection. We illustrate the

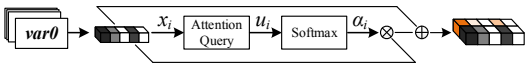

Fig. 4. Embedding code gadgets with attention. The colored part indicates the feature map after being influenced by attention weights.

process in Fig. 4. Given code gadget C contains tokens T, and t^i represents the i-th token in T. Firstly, we feed the embedded token vector x_i into a single-layer MLP to obtain u_i as a hidden representation of x_i. Secondly, we train a token-level context vector u_w, which can be regarded as a fixed attention query for context information. u_w can learn important contextual semantic features during training. We calculate the similarity between u_i and u_w via dot product to measure the importance of x_i, and obtain a normalized importance weight α_i by the softmax function for weight normalization. Finally, we weighted the tokens to get $\hat{x_i}$, which is the embedding vector after being affected by the embedding importance weight. The equations listed below are to get their values for these symbols.

$$x_i = word2vec(t^i) \tag{1}$$

$$u_i = \tanh\left(W_w x_i + b_w\right) \tag{2}$$

$$\alpha_i = \frac{\exp\left(u_i^\top u_w\right)}{\sum_i \exp\left(u_i^\top u_w\right)} \tag{3}$$

$$\hat{x_i} = \alpha_i x_i \tag{4}$$

Step V: Training vulnerability detection model. In step V, we carefully designed the second part of the multilayer attention mechanism to learn interesting token combinatorial patterns, which is composed of channel attention and spatial attention. We construct a CNN with a built-in spatial pyramid pooling layer, to alleviate the loss and distortion of vulnerability features in path-sensitive code gadgets.

a) Channel Attention and Spatial Attention: In the model training stage, as shown in Fig. 2(a), we also introduced the attention mechanism. Specifically, we add channel attention and spatial attention from the CBAM [32] to our network to improve the performance of convolution for potential vulnerability pattern recognition. The channel attention focuses on "*what*" is a meaningful input tensor, while the spatial attention is focused on "*where*" as the most informative part, which complements channel attention with the compressed channel dimension of the input. The channel attention map $M_c(F) \in \mathbb{R}^{C\times1\times1}$ and the spatial attention map $M_s(F) \in \mathbb{R}^{1\times H\times W}$ can be formally expressed as following:

$$\begin{aligned} M_c(F) &= \sigma(MLP(\text{AvgPool}(F)) + MLP(\text{Max Pool}(F))) \\ &= \sigma\left(W_1\left(W_0\left(F^c_{avg}\right)\right) + W_1\left(W_0\left(F^c_{\max}\right)\right)\right) \end{aligned} \tag{5}$$

$$\begin{aligned} M_s(F) &= \sigma\left(f^{7\times7}([\text{AvgPool}(F); \text{Max Pool}(F)])\right) \\ &= \sigma\left(f^{7\times7}\left(\left[F^s_{avg}; F^s_{max}\right]\right)\right) \end{aligned} \tag{6}$$

where $F \in \mathbb{R}^{C\times H\times W}$ is an intermediate feature map.

Spatial attention map $M_s(F)$ is created by connecting these two feature maps and performing standard convolution with a 7×7 convolution kernel. The Channel attention map $M_c(F)$ is generated by putting these two into the shared multilayer perceptrons (MLPs). Next, we aggregate the spatial information of the feature maps by exploiting average pooling and maximum pooling operations to generate average pooling feature F^c_{avg} and maximum pooling feature F^c_{max} respectively. We notice that the sequential alignment of the two modules gives better results than parallel alignment. After obtaining these two maps, according to Step V in Figure 2, for an intermediate feature map F, we first compute the channel attention to get F' and then calculate the spatial attention to get F'' for F'. The formal description of this process is:

$$F' = M_c(F) \otimes F \tag{7}$$

$$F'' = M_s(F') \otimes F' \tag{8}$$

where $\otimes$ denotes element-wise multiplication, σ denotes sigmoid function, $W_0 \in \mathbb{R}^{C/r\times C}$ and $W_1 \in \mathbb{R}^{C\times C/r}$ are shared MLP weights, and $f^{7\times7}$ represents a convolution operation with the filter size of 7×7.

b) Spatial Pyramid Pooling: A CNN consists of a convolutional layer, pooling layer, and fully connected layer in turn. The convolution operation calculates the result for any feature map size, while the pooling and fully connected layers are predefined fixed-size network structures. We carefully design the SPP to pool the flexible-length convolutional output into a fixed-length fully-connected layer input, thus the CNN can adaptively process vectors of flexible length.

The SPP is performed on the feature map after a convolution on the attention computation result. The structure of SPP for flexible-length codes is shown in Fig. 2. We note that the code is a sequence, with mutual information only in the dimension of neighboring tokens. Therefore, we designed a convolution kernel with the same width as the feature map for convolution and output a sequence. We divide the one-dimensional feature map into 4, 2, and 1 *spatial bins*. In each spatial bin, we perform spatial pyramid pooling on the response of each filter and then splice them into a fixed-length vector, which is the input to the dense layer. More specifically, if the number of channels in the last convolutional layer is k, the length of the one-dimensional vector output by SPP is fixed as $(4+2+1)\times k$, which obviously has nothing to do with the size of the input vector. The code gadgets can be of any length with SPP.

The last part of the model is the dense layer, where we designed three rows of neurons. Each neuron is connected to all neurons in the previous row. The number of neurons in the first two rows is 256 and 64. The last layer has only one neuron, and it is a float number.If this number is greater than 0.8, the output is flawed. Otherwise, it is normal. The closer the output is to "1", the more likely there is a vulnerability.

In summary, we present SEVULDET, the first semantically enhanced deep learning vulnerability detection framework, which employs the path-sensitive code gadgets generation algorithm to derive more details of path semantics and control-dependence. Furthermore, it carefully constructs a CNN with the built-in spatial pyramid pooling and multilayer attention mechanism to analyze and thus learn as much critical semantics as possible in variable-length codes.

IV. EVALUATIONS

Our evaluations focus on answering the following four Research Questions (RQs):

- RQ1: Would the additional semantics extracted and preserved by flexible-length path-sensitive code gadgets be helpful for detecting vulnerability, and to what extent?
- RQ2: Would a multilayer attention mechanism enable SEVULDET to be more effective, and to what extent?
- RQ3: Compared with the state-of-the-art frameworks, how effective is the SEVULDET?
- RQ4: Why do path-sensitive code gadgets and multilayer attention mechanisms can help SEVULDET discover more vulnerabilities?

A. Evaluation Metrics

To compare with the previous work, we employ five widely used indicators: False-positive rate (FPR), False-negative rate (FNR), Accuracy (A), Precision (P), and F1-measure (F1). Let TP, FP, FN, and TN be the number of true positives, false positives, false negatives, and true negatives, respectively. The false-positive rate denotes the proportion of negative instances that were reported as being positive. The false-negative rate represents the proportion of positive instances that were reported as being negative. Accuracy shows the correctness of all detected instances while noting that $A = (TP + TN)/(TP + FP + TN + FN)$. Precision, $P = TP/(TP+FP)$, gives the correctness of all detected vulnerable instances. F1-measure is the harmonic average of Precision and Recall, $F1 = 2 \cdot P \cdot (1 - FNR))/(P + (1 - FNR))$.

B. Dataset preparation

We adopt two widely used datasets: Software Assurance Reference Dataset (SARD) and Nation Vulnerability Dataset (NVD). SARD organizes and labels a large number of production synthetic and academic programs, which can be broadly classified into three categories: "Good" (i.e., no vulnerabilities), "Flaw" (i.e., with vulnerabilities), and "Mixed" (i.e., with vulnerabilities and their corresponding patched versions). The manifest.xml file in SARD details the file path, line number, type, and language of the vulnerability via XML format. NVD is a dataset containing open source software vulnerabilities where the software versions are affected by the vulnerabilities and the corresponding patches. The diff file in NVD indicates the location of the vulnerabilities, and each vulnerability corresponds to a CVE ID and CWE ID.

For SARD, we finally pick 126,231 C/C++ test cases involving 711 CWE IDs that can be recursively mapped by the tree-structured CWE-1000 [33] onto 126 CWE IDs and eventually divided into four classes as our sample. For NVD, we extracted 1,590 open source C/C++ test cases, of which 54.9% have vulnerabilities and 45.1% do not. The test cases synthesized in SARD are rich in diversity and quantity. Although the amount of cases in NVD is not large, but these cases contain complex semantics in real software, facilitating transfer learning between domains. After merging and de-duplication, we collected 549,555 API/library function-generated gadgets, of which 8.1% were vulnerable; 439,447 array-generated gadgets, of which 10.2% were vulnerable; 42,551 expression-generated gadgets, of which 8.7% were vulnerable; 54,300 pointer-generated gadget, of which 5.5% are vulnerable. Details given in Table I. For each category in our prepared dataset, we randomly select 30,000 path-sensitive code gadgets and divide them into five equal parts for five-fold cross-validation [30]. The experimental data are indeed unbalanced, but we note that the implementation of imbalanced data processing approaches does not help to enhance the detection [34]. So we still conduct our experiments with the unbalanced data.

TABLE I
THE NUMBER OF FOUR TYPES OF PATH-SENSITIVE CODE GADGETS FROM 127,821 PROGRAMS (126,231 IN SARD AND 1,590 IN NVD).

Categories	Vulnerable	Non-vulnerable	Total
Library/API function call	44,683	504,872	549,555
Array usage	44,996	394,451	439,447
Pointer usage	29,424	512,876	542,300
Arithmetic expression	3,696	38,855	42,551
All	122,799	1,451,054	1,573,853

TABLE II
EFFECTIVE OF THE ADDITIONAL SEMANTICS EXTRACTED BY PATH-SENSITIVE CODE GADGETS AND FLEXIBLE INPUT LENGTH.

Neural network	Flexible-length	Kind	A(%)	P(%)	F1(%)
BLSTM	✗	CG	94.9	82.5	85.2
		PS-CG	**95.1**	**87.8**	**88.8**
BGRU	✗	CG	96.0	84.1	85.9
		PS-CG	**97.0**	**88.6**	**90.7**
SEVULDET	✓	CG	95.4	91.0	89.6
		PS-CG	**97.3**	**96.2**	**94.2**

C. Experiments for Answering RQ1

To determine whether additional semantics would be useful for vulnerability detection, we select three networks for comparisons, i.e., BLSTM [35], BGRU [36], and network of SEVULDET. BLSTM and BGRU are representative networks in bidirectional RNN-based frameworks, and their input lengths must be fixed, while the input length of SEVULDET is flexible. We measure their effectiveness under the following two conditions: a) using the code gadget extracted purely from data and control dependence as a dataset ("CG" for short); b) using the code gadget extracted from data-dependence, control-dependence, and path semantics as a dataset ("PS-CG" for short). We randomly select 37,500 gadgets from each category, of which 30,000 are used as a training set and 7,500 as a test set. For BRNN, we predefine the time step as 500 (i.e., 500 tokens per code gadget).

The results are shown in Table II. Firstly, we compare CG sets with PS-CG sets. As we can see, the accuracy and precision of the selected networks improve by an average of 1.1% and 5.0% respectively when more path semantics are obtained. This indicates that the lost path semantics in previous works can help the network detect more vulnerabilities, and SEVULDET compensates for these semantic losses.

TABLE III
ABLATION EXPERIMENTS ON THE EFFECTIVENESS OF THE MULTILAYER ATTENTION MECHANISM.

Neural network	A(%)	P(%)	F1(%)
CNN	95.4	88.4	89.1
CNN-TokenATT	95.5	90.1	91.0
CNN-MultiATT	**97.3**	**96.2**	**94.2**

Moverover, we compare the results of three kinds of networks. It is clear that the accuracy and precision of our network structure are higher than that of BLSTM and BGRU. That means the semantics preserved by the flexible input length realized by the spatial pyramid pooling layer does help detect vulnerabilities.

RQ 1 Answer: SEVULDET is more efficient than bidirectional RNN-based frameworks owing to the semantics enhanced by flexible-length path-sensitive code gadgets. It has improved F1-measure to 94.2%.

D. Experiments for Answering RQ2

To measure the effectiveness of the multilayer attention mechanism, we conduct ablation experiments utilizing the dataset mentioned in Section IV-B for training and detection. As shown in Fig. 2, our framework applies the attention mechanism in two steps: token embedding and model training, which together are referred to as the multilayer attention mechanism. For the ablation experiments, three network structures are adopted, i.e., a CNN without attention (CNN for short), a CNN with token attention (CNN-TokenATT for short), and a CNN with a multilayer attention mechanism (CNN-MultiATT for short). We apply the same hyperparameters and datasets to the above networks.

Table III summarizes the comparison. The attention mechanism effectively improve the accuracy and precision of the detection at both steps. The embedding step increased the F1-measure by 1.9%, while the model training step with the channel and spatial attention increased the F1-measure by 3.2%, the latter being slightly larger. Combining the above observations, it is clear that the attention mechanism enhances the vulnerability detection framework for deep learning.

It is still an open problem to explain the effectiveness of deep neural networks. Literature [11] demonstrates that CNN-based vulnerability detection frameworks are only slightly less effective than RNN-based frameworks. Table II and table III have demonstrated the effect of semantic information and the multilayer attention mechanism on vulnerability detection.

We provide theoretical explanations with the knowledge of deep learning to some extent. The receptive field [37] is defined as the region in the input space that a particular CNN's feature is looking at (i.e., be affected by), and size of it gradually expands with the stack of convolutional layers. The calculations in the multilayer attention mechanism allow features of the same dimension to interact, expanding the perceptual field without increasing the depth. Moreover, attention networks have been shown to be helpful in processing languages with hierarchical boundaries [24]. Therefore, the feature maps of different locations and channels can be perceived from each other and adaptively skewed toward primary potential features.

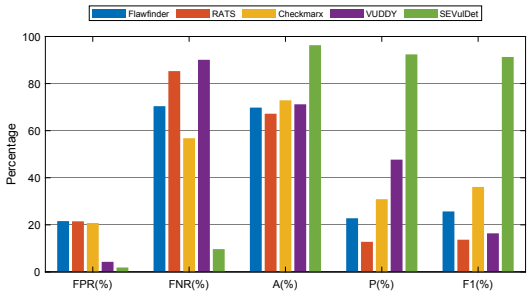

Fig. 5. The results of comparative experiment with advanced classical static vulnerability detection frameworks.

RQ 2 Answer: The multilayer attention mechanism enables the code vulnerability detection framework to capture primary potential vulnerability features.

E. Experiments for Answering RQ3

To fully demonstrate the effectiveness of SEVULDET, we make a comprehensive comparison with classical widely used static vulnerability detection frameworks firstly. Specifically, we choose to compare with the open-source analysis tool Flawfinder [5], the Rough Auditing Tool for Security [38] (RATS), the commercial detection tool Checkmarx [4], and the similarity-based framework VUDDY [39], in view of their popularity in C/C++ codes vulnerabilities detection.

Fig. 5 summarizes the experimental results. Open source Flawfinder and RATS have both high FPR and FNR. Checkmarx is better than Flawfinder and RATS but has high FPR and FNR. In addition, VUDDY can only detect vulnerabilities almost identical to those in the training program, so it trades a high FNR for a low FPR. We observe that our framework SEVULDET vastly outperforms the widely used classical static vulnerability detection methods.

In addition to the comparison with classical static methods, we compare the effectiveness of SEVULDET with other deep learning-based works VulDeePecker [9] and SySeVR [11]. We compare the effectiveness of three schemes on five categories of code gadgets in our dataset (i.e., FC, AU, PU, AE, and All). Only the FC category is used to evaluate VulDeePecker, as it simply detects vulnerabilities generated by library function calls. The main hyper-parameters of previous studies and SEVULDET are summarized in Table IV.

Table V summarizes the results of the comparison experiments. Because VulDeePecker not only cannot accommodate control-dependent semantic information but also extracts much less fine-grained path-sensitive semantic information than SEVULDET, we believe that SEVULDET is more effective than VulDeePecker. For datasets with only one type of vulnerabilities, the average F1 value of SEVULDET is 7.94% higher than other methods. For the dataset with all four types of

TABLE IV
THE HYPER-PARAMETERS OF VULDEEPECKER, SYSEVR, AND SEVULDET.

Parameters	VulDeePecker	SySeVR	SEVulDet
Dimension	50	30	30
Flexible-length	✗	✗	✓
Batch size	64	16	16
Learning rate	0.001	0.002	0.0001
Dropout	0.5	0.2	0.2
Epochs	4	20	20

TABLE V
COMPARISON RESULTS OF THREE VULNERABILITY FRAMEWORKS BASED ON DEEP LEARNING AND CODE GADGETS (I.E., VULDEEPECKER, SYSEVR, AND SEVULDET).

Work - Kind	FPR(%)	FNR(%)	A(%)	P(%)	F1(%)
VulDeePecker-FC	4.1	21.7	92.0	84.0	81.0
SySeVR-FC	3.1	7.6	95.9	89.5	90.9
SEVulDet-FC	**1.9**	**5.0**	**97.3**	**94.9**	**94.9**
SySeVR-AU	**3.0**	10.2	95.2	90.6	90.2
SEVulDet-AU	4.9	**3.6**	**96.0**	**93.3**	**94.8**
SySeVR-PU	1.7	22.7	96.2	83.2	80.1
SEVulDet-PU	**1.4**	**9.3**	**97.2**	**93.1**	**91.9**
SySeVR-AE	1.4	3.8	98.2	93.7	94.9
SEVulDet-AE	**0.5**	**3.6**	**99.8**	**96.3**	**96.3**
SySeVR-All	2.7	12.3	96.0	84.1	85.9
SEVulDet-All	**1.9**	**9.7**	**96.3**	**92.4**	**91.3**

TABLE VI
COMPARISON RESULTS OF THREE VULNERABILITY FRAMEWORKS (I.E., VULDEEPECKER, SYSEVR, AND SEVULDET) ON REAL-WORLD SOFTWARE PRODUCTS.

Work	FPR(%)	FNR(%)	A(%)	P(%)	F1(%)
VulDeePecker	4.3	26.7	94.3	51.6	60.6
SySeVR	3.5	19.8	95.5	60.0	67.9
SEVulDet	**3.3**	**11.5**	**96.2**	**62.7**	**73.4**

vulnerabilities, SEVULDET outperformed SySeVR in all metrics, with a 5.4% increase in F1. Further, F1 values obtained by SEVULDET for all single-type vulnerability detection are higher than the F1 values obtained by SEVULDET for four types of vulnerabilities, which means that SEVULDET has a better performance in single-type than multiple types.

Test on Real-world Software Products. We have applied SEVULDET to eight new versions of real-world software products, Xen, to further show its validity in real-world software. We have carefully selected 175 CVEs from these versions that are not affected by upstream code vulnerabilities and are classified under the CWE IDs we concern about. After de-duplication, a total of 126,943 path-sensitive code gadgets are generated, of which 7,558 (i.e., 6.0%) are vulnerable.

We evaluate several pre-trained detection frameworks and summarize the results in Table VI. On real software Xen, the path-sensitive code gadgets and semantic-enhanced network proposed by SEVULDET can help reduce misclassifications, especially false negative. Furthermore, we find three vulnerabilities in our classification results that have not been reported in the NVD for Xen but do exist (i.e., their existence are not known to us until we manually check their patches). Our findings are identified in Table VII, and are similar to the three vulnerabilities (i.e., CVE-2016-4453, CVE-2016-9104, CVE-2016-9776) reported in Qemu, as Qemu and Xen share some codes. As we can see, SEVULDET can detect at least one more vulnerability than VulDeePecker and SySeVR. We also manually run a 24-hour fuzzing test for each version based on the prevalent fuzzing tool named AFL [40] with fuzzing harness. While the vulnerability CVE-2016-9104 can not be identified by AFL due to the special *offset* value and the far apart trigger position to the test input interface.

We verify all these vulnerabilities in the subsequent patches of Xen. In IV-F, we further study how semantic enhancement can help identify vulnerabilities by detailing the finding of the vulnerability CVE-2016-9776 induced by the elaborate payload received by the Ethernet Controller emulator in Xen.

RQ 3 Answer: SEVULDET outperforms classical static approaches and excels with state-of-the-art deep learning-based solutions, with an average F1-measure of up to 94.5%. Moreover, SEVULDET identifies more real-world vulnerabilities than existing technologies.

F. Experiments for Answering RQ4

To showcase the intrinsic interpretability of the path-sensitive code gadgets generation algorithm and multilayer attention mechanism on vulnerabilities detection, below we articulate the discovery of the unreported vulnerability CVE-2016-9776 found above, through hooking and visualizing of the token weight calculated by the attention mechanism.

The left side of Fig. 6 exemplifies part of the path-sensitive code gadget generated by SEVULDET for the source code in Xen corresponding to the infinite loop vulnerability CVE-2016-9776. This vulnerability arises since the user-controllable variable $s{\rightarrow}emrbr$ (Line 465) is assigned a value of 0, leading to $size$ to remain constant (Line 467), creating an infinite loop issue. The path-sensitive code gadget successively stores the data-dependence, the control range corresponding to $while$ (from 463 to 495), and the path information to line 465 (repeated execution of statements between 463 and 495). Thus the abstract logic of lines 463, 465 and 467 that interacts and ultimately affects the loop count can be extracted in its entirety, which was not possible with the previous code gadget generation approach.

After the vulnerability logic is stored intactly in the path-sensitive code gadget, the multilayer attention mechanism will acquire the critical semantics. Specifically, the gadget contains 711 tokens, and the length-adaptive network does not cut or discard any tokens. We feed it into pretrained SEVULDET, and hook token weight calculated by the attention mechanism. The top ten most weighted tokens are visualized in right subplot of Fig. 6, and their percentages are obtained by regularizing based on the maximum weight. Multiple of these tokens appear on lines 463, 465, 466, and 467, which are the locations where the vulnerability was formed. Moreover, the weight of the brackets in line 495 ranks 8th, indicating that the network can

TABLE VII
THREE VULNERABILITIES IN REAL-WORD SOFTWARE, WHICH DO NOT REPORTED BY PUBLIC VULNERABILITY DATASET, IDENTIFIED BY SEVULDET.

Real-world software	The path of vulnerable files	Vulnerability release date	1st patched version of target product	CVE-ID (QEMU)	Systems which can detect the vulnerability
Xen 4.4.2	*/display/vmware vga.c	06/01/2016	Xen 4.8.0	CVE-2016-4453	AFL, SySeVR, **SEVulDet**
Xen 4.6.0	*/9pfs/virtio-9p.c	12/09/2016	Xen 4.9.0	CVE-2016-9104	VulDeePecker, **SEVulDet**
Xen 4.7.4	*/net/mcf_fec.c	12/29/2016	Xen 4.9.0	CVE-2016-9776	AFL, **SEVulDet**

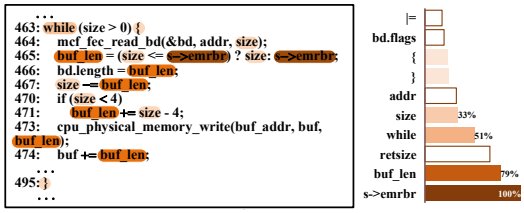

Fig. 6. A path-sensitive code gadget corresponding to CVE-2016-9776 and a visualization of the ten tokens of most interest to attention in SEVULDET.

notice our path semantics [24]. The operators and variables associated with vulnerabilities will be of sufficient concern to SEVULDET.

Therefore, the multilayer attention mechanism in SEVULDET arguably better captures the bounded hierarchical structure of source codes so as to successfully learns more potential vulnerability patterns from path-sensitive code gadgets which contain sufficient vulnerability logic.

RQ 4 Answer: Path-sensitive gadgets with complete vulnerability logic help the multilayer attention mechanism learn new potential vulnerability patterns.

V. RELATED WORK

We broadly categorize state-of-the-art solutions in three categories: classical approaches, conventional machine learning-based approaches and deep learning-based approaches.

A. Classical Approaches

Early efforts in software vulnerability detection [5], [41], [42] relied excessively on complex rules drafted by experts. Solutions such as code similarity detection [6], [39] and symbolic execution [43], [44] that rely primarily on analysis of source code often suffer from high false positives. Fuzzing [7], [45]–[47] and taint analysis [48], [49] usually reduce false positives but suffer from low code coverage [50] and inaccurate localization [51]. Hybrid analysis schemes [52] integrate multiple analysis techniques, combining their advantages and disadvantages, and are inefficient to operate in practice.

B. Conventional Machine Learning-Based Approaches

Conventional ML-Based (non-neural network) technologies for vulnerability analysis and discovery fall into three major categories [53]: 1) software metrics-based; 2) vulnerable code pattern-based; and 3) anomaly-based. The software metrics measure software product quality but vulnerabilities because they do not provide an analysis of code security [54]. Patterns of code are depicted in various ways, such as code tokens [55], ASTs, CFGs, program execution trace and so on. Each form of representation provides a different view of the source code. Methods for abnormal patterns [56] can detect codes that do not conform to program guidelines or conventions for potential flaws/vulnerabilities but may incur false positives or be confined with task-specific applications.

C. Deep Learning-Based Approaches

Our work is part of a recent DL-based software vulnerability detection effort [8], [9], [12], [57], [58]. A comprehensive summary and identifies challenges of the field can be found at [13]. Rebecca et al. [8] labeled vulnerabilities in open-source code functions using a variety of static analysis frameworks and learned the labels using neural networks. Li et al. [9] proposed the code gadgets generated by data-dependence and used BLSTM to learn and detect vulnerabilities, which can only detect vulnerabilities caused by library/API function calls. Subsequent research [59] conducted a comparative study, introducing control dependencies and comparing different network structures, but still analyzing the same types of vulnerabilities. SySeVR [11] prefered the SyVcs than functions to locate 811 library/API functions calls vulnerabilities. μVulDeePecker [10] presents code attention, coupled with code gadgets, which locate 40 CWE IDs caused by library/API functions calls. These code gadget-based works suffer from semantic loss, which SEVULDET compensates for both in terms of preprocessing and networking.

VI. CONCLUSION

We propose SEVULDET, the first semantically enhanced deep learning vulnerability detection framework. SEVULDET employs a path-sensitive code gadgets generation algorithm to derive more details of path semantics and control dependence. In addition, it constructs a CNN with built-in spatial pyramid pooling and multilayer attention mechanism to preserve and further learn as much critical semantics as possible in flexible-length source codes. Extensive evaluations show SEVULDET far outperforms classical detection approaches and excels with state-of-the-art deep learning-based solutions, by improving F1-measure to around 94.5%. We also verify its effect in real-world software products and identify more vulnerabilities than existing technologies. Moreover, we make the first step in analyzing the intrinsic mechanism of attention mechanism in improving the effectiveness of vulnerability detection.

REFERENCES

[1] "Security update for microsoft windows smb server (4013389)," https://docs.microsoft.com/en-us/security-updates/Securitybulletins/2017/ms17-010, 2022.

[2] "Cve-2020-8597: buffer overflow in pppd," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-8597, 2022.

[3] N. Sun, J. Zhang, P. Rimba, S. Gao, L. Y. Zhang, and Y. Xiang, "Data-driven cybersecurity incident prediction: A survey," *IEEE Communications Surveys Tutorials*, vol. 21, no. 2, pp. 1744–1772, 2019.

[4] "Checkmarx - application security testing and static code analysis," https://www.checkmarx.com, 2022.

[5] "Flawfinder home page," https://dwheeler.com/flawfinder, 2022.

[6] H. Sun, L. Cui, L. Li, Z. Ding, Z. Hao, J. Cui, and P. Liu, "Vdsimilar: Vulnerability detection based on code similarity of vulnerabilities and patches," *Computers & Security*, vol. 110, p. 102417, 2021.

[7] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 711–725.

[8] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, "Automated vulnerability detection in source code using deep representation learning," in *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 2018, pp. 757–762.

[9] 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," *arXiv preprint arXiv:1801.01681*, 2018.

[10] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, "μvuldeepecker: A deep learning-based system for multiclass vulnerability detection," *IEEE Transactions on Dependable and Secure Computing*, p. 1–1, 2019. [Online]. Available: http://dx.doi.org/10.1109/TDSC.2019.2942930

[11] 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 Transactions on Dependable and Secure Computing*, 2021.

[12] H. Wang, G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, Y. Feng, L. Bian, and Z. Wang, "Combining graph-based learning with automated data collection for code vulnerability detection," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 1943–1958, 2020.

[13] G. Lin, S. Wen, Q.-L. Han, J. Zhang, and Y. Xiang, "Software vulnerability detection using deep neural networks: A survey," *Proceedings of the IEEE*, vol. 108, no. 10, pp. 1825–1848, 2020.

[14] M. Dwarampudi and N. Reddy, "Effects of padding on lstms and cnns," *arXiv preprint arXiv:1903.07288*, 2019.

[15] "Software assurance reference dataset," https://samate.nist.gov/SARD/, 2022.

[16] "Nvd - home," https://nvd.nist.gov/, 2022.

[17] F. Tip, *A survey of program slicing techniques*. Centrum voor Wiskunde en Informatica Amsterdam, 1994.

[18] "Using svace static analysis tool in samsung environments (youil kim, isprasopen-2019)," https://0x1.tv/Using_Svace_static_analysis_tool_in_Samsung_environments_(Youil_Kim,_ISPRASOPEN-2019), 2022.

[19] J. A. Harer, L. Y. Kim, R. L. Russell, O. Ozdemir, L. R. Kosta, A. Rangamani, L. H. Hamilton, G. I. Centeno, J. R. Key, P. M. Ellingwood *et al.*, "Automated software vulnerability detection with machine learning," *arXiv preprint arXiv:1803.04497*, 2018.

[20] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," *arXiv preprint arXiv:1909.03496*, 2019.

[21] Y. Hu, W. Kuang, Z. Qin, K. Li, J. Zhang, Y. Gao, W. Li, and K. Li, "Artificial intelligence security: Threats and countermeasures," *ACM Comput. Surv.*, vol. 55, no. 1, nov 2021.

[22] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[23] J. Li, W. Monroe, and D. Jurafsky, "Understanding neural networks through representation erasure," *arXiv preprint arXiv:1612.08220*, 2016.

[24] S. Yao, B. Peng, C. Papadimitriou, and K. Narasimhan, "Self-attention networks can process bounded hierarchical languages," 2021.

[25] S. Liu, T. Li, Z. Li, V. Srikumar, V. Pascucci, and P.-T. Bremer, "Visual interrogation of attention-based models for natural language inference and machine comprehension," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2018.

[26] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[27] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, "Yolov4: Optimal speed and accuracy of object detection," 2020.

[28] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.

[29] "Joern," https://github.com/ShiftLeftSecurity/joern, 2022.

[30] T. Fushiki, "Estimation of prediction error by using k-fold cross-validation," *Statistics and Computing*, vol. 21, no. 2, pp. 137–146, 2011.

[31] "models.word2vec - word2vec embeddings," https://radimrehurek.com/gensim/models/word2vec.html, 2022.

[32] S. Woo, J. Park, J.-Y. Lee, and I. S. Kweon, "Cbam: Convolutional block attention module," 2018.

[33] "Cwe-1000: Research concepts," http://cwe.mitre.org/data/definitions/1000.html, 2022.

[34] Z. Li, D. Zou, J. Tang, Z. Zhang, M. Sun, and H. Jin, "A comparative study of deep learning-based vulnerability detection system," *IEEE Access*, vol. 7, pp. 103 184–103 197, 2019.

[35] A. Graves and J. Schmidhuber, "Framewise phoneme classification with bidirectional lstm and other neural network architectures," *Neural networks*, vol. 18, no. 5-6, pp. 602–610, 2005.

[36] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[37] A. Araujo, W. Norris, and J. Sim, "Computing receptive fields of convolutional neural networks," *Distill*, vol. 4, no. 11, p. e21, 2019.

[38] "Rough auditing tool for security (rats)," https://code.google.com/p/rough-auditing-tool-for-security/, 2021.

[39] S. Kim, S. Woo, H. Lee, and H. Oh, "Vuddy: A scalable approach for vulnerable code clone discovery," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 595–614.

[40] "american fuzzy lop (2.52b)," https://lcamtuf.coredump.cx/afl/, 2022.

[41] "Cppcheck - a tool for static c/c++ code analysis," http://cppcheck.net, 2022.

[42] "Coverity scan - static analysis," https://scan.coverity.com, 2022.

[43] D. A. Ramos and D. Engler, "Under-constrained symbolic execution: Correctness checking for real code," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 49–64.

[44] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–39, 2018.

[45] "Peach fuzzing platform," http://peachfuzzer.com, 2022.

[46] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2329–2344.

[47] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 679–696.

[48] J. Kim, T. Kim, and E. G. Im, "Survey of dynamic taint analysis," in *2014 4th IEEE International Conference on Network Infrastructure and Digital Content*. IEEE, 2014, pp. 269–272.

[49] J. Ming, D. Wu, G. Xiao, J. Wang, and P. Liu, "Taintpipe: Pipelined symbolic taint analysis," in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015, pp. 65–80.

[50] V. J. Manes, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "Fuzzing: Art, science, and engineering," *arXiv preprint arXiv:1812.00140*, 2018.

[51] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting

fuzzing through selective symbolic execution," in *NDSS*, vol. 16, no. 2016, 2016, pp. 1–16.

[52] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee, "Hfl: Hybrid fuzzing on the linux kernel." in *NDSS*, 2020.

[53] S. M. Ghaffarian and H. R. Shahriari, "Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey," *ACM Comput. Surv.*, vol. 50, no. 4, aug 2017. [Online]. Available: https://doi.org/10.1145/3092566

[54] P. Morrison, K. Herzig, B. Murphy, and L. Williams, "Challenges with applying vulnerability prediction models," in *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security*, ser. HotSoS '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2746194.2746198

[55] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar, "Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 426–437.

[56] H. Xue, Y. Chen, F. Yao, Y. Li, T. Lan, and G. Venkataramani, "Simber: Eliminating redundant memory bound checks via statistical inference," in *IFIP International Conference on ICT Systems Security and Privacy Protection*. Springer, 2017, pp. 413–426.

[57] G. Lin, J. Zhang, W. Luo, L. Pan, O. De Vel, P. Montague, and Y. Xiang, "Software vulnerability discovery via learning multi-domain knowledge bases," *IEEE Transactions on Dependable and Secure Computing*, 2019.

[58] Y. Li, S. Wang, T. N. Nguyen, and S. Van Nguyen, "Improving bug detection via context-based code representation learning and attention-based neural networks," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–30, 2019.

[59] Z. Li, D. Zou, J. Tang, Z. Zhang, M. Sun, and H. Jin, "A comparative study of deep learning-based vulnerability detection system," *IEEE Access*, vol. 7, pp. 103 184–103 197, 2019.