DeepVD: Toward Class-Separation Features for Neural Network Vulnerability Detection

Wenbo Wang
Department of Informatics
New Jersey Institute of Technology
New Jersey, USA
ww6@njit.edu

Tien N. Nguyen
Computer Science Department
The University of Texas at Dallas
Texas, USA
tien.n.nguyen@utdallas.edu

Shaohua Wang*
Department of Informatics
New Jersey Institute of Technology
New Jersey, USA
davidsw@njit.edu

Yi Li
Department of Informatics
New Jersey Institute of Technology
New Jersey, USA
yl622@njit.edu

Jiyuan Zhang
Computer Science Department
University of Illinois Urbana-Champaign
Illinois, USA
jiyuanz3@illinois.edu

Aashish Yadavally
Computer Science Department
The University of Texas at Dallas
Texas, USA
Aashish.Yadavally@utdallas.edu

***Abstract*—The advances of machine learning (ML) including deep learning (DL) have enabled several approaches to implicitly learn vulnerable code patterns to automatically detect software vulnerabilities. A recent study showed that despite successes, the existing ML/DL-based vulnerability detection (VD) models are limited in the ability to distinguish between the two classes of vulnerability and benign code. We propose DEEPVD, a graph-based neural network VD model that emphasizes on class-separation features between vulnerability and benign code. DEEPVD leverages three types of class-separation features at different levels of abstraction: statement types (similar to Part-of-Speech tagging), Post-Dominator Tree (covering regular flows of execution), and Exception Flow Graph (covering the exception and error-handling flows). We conducted several experiments to evaluate DEEPVD in a real-world vulnerability dataset of 303 projects with 13,130 vulnerable methods. Our results show that DEEPVD relatively improves over the state-of-the-art ML/DL-based VD approaches 13%–29.6% in precision, 15.6%–28.9% in recall, and 16.4%–25.8% in F-score. Our ablation study confirms that our designed features and components help DEEPVD achieve high class-separability for vulnerability and benign code.**

***Index Terms*—neural vulnerability detection, graph neural network, class separation**

I. INTRODUCTION

Software vulnerability is a defect in software that can be exploited by attackers. Several automated approaches have been proposed to detect vulnerable code. The existing vulnerability detection (VD) approaches can broadly classified into three categories: *program analysis* (PA), *software mining* (SM), and *machine-learning* (ML) (including *deep learning* (DL)). First, the PA-based VD tools [1]–[10] put the emphasis on specific types of vulnerabilities, e.g., buffer overflow [7], SQL injection [8], cross-site scripting [9], authentication bypass [10], *etc*. More general types of software vulnerabilities could manifest in several forms, e.g., misuses in software libraries and frameworks, incorrect exception handling, etc. For those types, *software mining* approaches [11], [12] leverage the history of prior vulnerabilities to detect the current ones. The advances in ML/DL [13]–[18] have enabled the implicit learning of the patterns of vulnerable code for such detection.

Despite their successes, the ML/DL-based approaches still have limitations. In their study in the real-world vulnerability scenarios, Chakraborty *et al.* [17] reported four key issues with the state-of-the-art ML/DL-based VD approaches: 1) learning irrelevant features, 2) inadequate model, 3) data duplication in training and testing, and 4) data imbalance. While the last three issues are about the model and training/testing data, the first issue is about the features relevant to code representation learning, which is a crucial step as feature engineering in data science. Using explainable artificial intelligence technique [19], the authors [17] found that the state-of-the-art ML/DL-based vulnerability detection models are essentially learning up *the features that are too general and not directly relevant to the actual cause of the vulnerabilities* [17].

Let us explain the limitations of the state-of-the-art DL-based VD approaches with regard to those points. Russell *et al.* [15] treats source code as a sequence of tokens, and does not consider the program dependencies among statements, which plays a vital role in VD. In VulDeePecker [13], the instances of vulnerable and benign code are represented via the *code gadgets*, which are built from the program slices starting at the function calls. SySeVR [14] includes the program slices from more syntactic units: array usages, pointer usages, and arithmetic expressions. Moreover, VulDeePecker and SySeVR use sequence-based, bidirectional LSTM, which treats source code as sequences, thus do not capture other crucial program dependencies. To address that, Devign [16], Reveal [17], and IVDetect [18] leveraged *graph neural network models*, e.g., Gated Graph Recurrent Layers [16], Gated Graph Neural Network (GGNN) [17], Graph Convolution Network [18] to learn from Data Flow Graph (DFG), Control Flow Graph (CFG), Program Dependence Graph (PDG), and Code Property Graph (CPG), a union of code sequences, AST edges, and PDG.

* Corresponding Author

Despite the successes of those graph-based VD approaches, as reported in [17], they do not focus on the class-separation features between the vulnerability and benign classes. Those graph-based representations (CFG, DFG, PDG, CPG) are important in representing program dependencies and semantics. However, they often contain many program entities and their dependencies that are irrelevant to the vulnerability (noises) and do not help a model better recognize vulnerabilities.

We propose DEEPVD, a graph-based neural network VD model with the goal of leveraging the features that emphasize on the *class separation between vulnerability and benign categories*. We design DEEPVD with the following insights. First, as a program is executed through a method, the execution can flow in two ways: 1) a regular flow from the start of the method m reaching an exit point of m, and 2) the exception/error handling flow(s) from the start of m to the exception/error handling point(s). One of the key reasons for a vulnerability is the mishandling of exceptions and error cases. For example, a program could miss a case in the data validation of an input, leading to an injection attack via a crafted input (Section II). Thus, for a method m, we capture the program slices from the input of m to the exception/error handling point(s). Those slices are combined into a data structure, called *Exception Flow Graph* (EFG) [20]. EFG is expected to consist of the key program elements and their dependencies pertaining the mishandling of exceptions/errors, leading to vulnerability.

Second, while using EFG to address the exception flows in a program, we also consider the *Post-Dominator Tree* (PDT) [21] of each method for the regular flows. A PDT is a tree in which each node represents a statement and each edge represents a post-dominance relation. A statement d is considered as a *post-dominator* of another statement s if all the paths to the exit point of the method starting at s must go through d. While PDT is simpler than CFG, it could help a model learn the associations between the executions of s and d in a regular flow leading to the exit point. *If d crashes, the execution path of s will never reach the exit point*.

Third, according to the vulnerability analysis from Checkmarx [22], vulnerable code is often involved to the specific *syntactic types*. Thus, we enhance the EFG with a technique equivalent to the Part-of-Speech (POS) tagging in natural language processing (NLP). POS tagging has been shown to improve the performance of the downstream NLP tasks (text-to-speech conversion [23], name entity recognition [24], etc.). Such tagging was also applied in code completion to achieve high accuracy [25]. For each statement node in the graph representation, we associate it with a statement type since a vulnerability is often relevant to specific statement types, e.g., array declarations/references, pointer declarations/references, assignments, and expressions [14], [22]. The statement types complement to the semantic dependencies captured by the EFG and PDT, and improve class-separation. EFG and PDT are encoded and fed into a Label Graph Convolutional Network (Label-GCN) [26] and Tree-LSTM [27] for VD.

We conducted several experiments to evaluate DEEPVD in a real-world vulnerability dataset of 303 open-source projects with 13,130 vulnerabilities. Our empirical results show that DEEPVD relatively improves over the state-of-the-art ML/DL-based VD approaches from 13%–29.6% in precision, 15.6%–28.9% in recall, and 16.4%–25.8% in F-score. The high performance is also achieved across different vulnerability types, e.g., Denial of Service, Overflow, Execute Code, etc. Our ablation study confirms that all designed features/components positively contribute to DEEPVD's accuracy. With the t-SNE technique, we show that our proposed features help DEEPVD achieve better class separation, leading to better distinguish the vulnerable and benign code. Our sensitivity analysis shows that the proposed features (e.g., PDT, EFG) are better than the traditional program graph representations (PDG, CFG, DFG, CPG) in helping DEEPVD in VD. We also use Lemna [19], an explainable AI model, to show that DEEPVD indeed uses vulnerability-relevant statements in its correct prediction. In brief, the key contributions of this paper include:

1) **DEEPVD, a graph-based neural network VD model** that is the first to emphasize on the class-separation features (EFG, PDT, calling relations, and statement types) to help a model better distinguish the vulnerable code and benign code.
2) **Extensive empirical evaluation.** Our empirical results show that DEEPVD achieves better performance than ML/DL-based VD models due to the better class-separation features.

Our data and code is available in our website [28].

```
1 void Jp2Image::printStructure(...) {
2  ...
3  subBox.length=getLong((byte*)&subBox.length,bigEndian);
4  subBox.type=getLong((byte*)&subBox.type,bigEndian);
5  - // subBox.length makes no sense if it is larger than
       the rest of the file
6  - if (subBox.length > io_->size() - io_->tell()) {
7  + // subBox.length makes no sense if it is larger than
       the rest of the file || 0
8  + if (subBox.length == 0 || subBox.length > io_->size()
       - io_->tell()) {
9      throw Error(kerCorruptedMetadata);
10  }
11  DataBuf data(subBox.length - sizeof(box));
12  io_->read(data.pData_,data.size_);
13 }
```

Fig. 1: CVE-2020-18899: a Denial of Service (DoS) from an Uncontrolled Memory Allocation in `Exiv2` 0.27

II. MOTIVATING EXAMPLES

A. Examples and Motivation

Fig. 1 shows the vulnerable code and the fix to a C++ project named `Exiv2` 0.27 with the Common Vulnerabilities and Exposures (CVE) CVE-2020-18899: *"An uncontrolled memory allocation in `Exiv2 0.27` allowing attackers to cause a denial of service (DOS) via a crafted input"*. The issue was an integer overflow with the unexpected value of zero for `subBox.length`. This made the function `data()` at line 11 attempt to consume too much memory, leading to a crash. Thus, to fix this vulnerability, a developer added an extra condition `subBox.length == 0` (line 8).

```
1 public JSONObject rename() {
2   String oldFile = this.get.get("old");
3   String newFile = this.get.get("new");
4   oldFile = getFilePath(oldFile);
5   ...
6   String path = oldFile.substring(0, pos + 1);
7   File fileFrom = null;
8   File fileTo = null;
9   try {
10    fileFrom = new File(this.fileRoot + oldFile);
11    fileTo = new File(this.fileRoot + path + newFile);
12    if (fileTo.exists()) {
13      if (fileTo.isDirectory()) {
14       this.error(sprintf(lang("DIRECTORY_ALREADY_EXISTS");
15       error = true;
16      } else { // fileTo.isFile
17        this.error(sprintf(lang("FILE_ALREADY_EXISTS").));
18        error = true;
19      }
20    } else if (!fileFrom.renameTo(fileTo)) {
21     this.error(sprintf(lang("ERROR_RENAMING_DIRECTORY"));
22     error = true;
23    }
24  } catch (Exception e) {
25    if (fileFrom.isDirectory()) {
26     this.error(sprintf(lang("ERROR_RENAMING_DIRECTORY").;
27    } else {
28      this.error(sprintf(lang("ERROR_RENAMING_FILE"),..));
29    }
30    error = true;
31  }...
```

Fig. 2: CVE-2020-19155: Improper Access Control in `Jfinal`

The input value `subBox.length` was not validated, and the *exception/error-throwing statement* at line 9 was supposed to catch the case with the unexpected zero value of `subBox.length`. This leaded to the uncontrolled large memory allocation (line 11), i.e., the execution flow to the exception handling-point was improperly constructed. The correct flow goes to line 9 if `subBox.length=0` as in the code after fix.

To be able to detect such an improper exception/error handling cases, one needs to examine both data and control dependencies among the program elements that are involved in the exception flow. For example, we can observe that there exist a data dependency between the assignment statement to `subBox.length` at line 3 and the `if` statement referring to `subBox.length` at line 6, and the control dependency between that `if` statement and the `throw` statement at line 9.

Observation 1. *A model could investigate the* data and control flows toward the exception/error-handling points to detect a potential vulnerability.

Fig. 2 shows the vulnerable code in the `JFinal` project for the CVE record CVE-2020-19155, reporting on *"Improper Access Control in Jfinal CMS v4.7.1 and earlier allows remote attackers to obtain sensitive information and/or execute arbitrary code via the 'FileManager.rename()"*. In this vulnerability, despite of multiple validations for the renaming operation, there is no restriction or filtering on *the new file name* passed by the user, which allows the malicious attacker to rename the file to a `jsp` file. For example, an attacker could login to the background, upload a file named `shell.jpg` with a malicious content, rename the file from `shell.jpg` to `shell.jsp`, and access the shell and execute the malicious code.

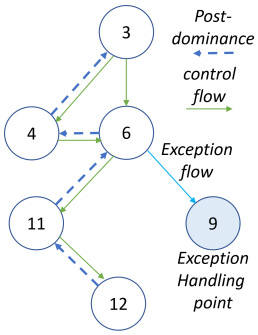

Fig. 3: EFG and PDT for the Example in Fig. 1

The exit points for exception handling were defined with `this.error(...)` at lines 14, 17, 21, 26, and 28. However, none of them verifies the validity of the new file name `newFile`. That is, there is a missing flow from the variable `newFile` to `fileTo`, to a new exception-handling point, for a needed validity checking on the file name. This example also suggests that a model can examine *the flows from the variables to the exception-handling points* for vulnerability detection.

While several state-of-the-art ML-based vulnerability detection (VD) approaches have explored various program dependencies with different graph-based program representations such as PDG [13], [18], CFG [16], DFG [16], and CPG [17], none of them has explored the flows to the exception-handling points. In addition, too much information from those graphs might add noises or redundant knowledge. An exception flow is defined as an inter-procedural program slice from the entry point to the exception-handling points. All of those slices connecting together form an *Exception Flow Graph* (EFG) [20].

B. Key Ideas

To improve accuracy in VD, we aim to extract the features of source code that improve *class separability*, i.e., the separation between the vulnerability and benign categories.

1) **Exception Flow Graph (EFG)***:* As shown in Section II-A, EFG is expected to consist the key program elements and their dependencies pertaining to the vulnerability. EFG is expected to help a model distinguish the key characteristics in the improper and proper handling of exceptions and error cases, which is one of the key aspects of vulnerabilities.

2) **Post Dominator Tree (PDT)***:* While EFG is for exception flows, to accommodate the regular flows, we also build the *Post-Dominator Tree* (PDT) [21] in which each node represents a statement and each edge represents a post-dominance relation. A statement d is considered as a *post-dominator* of another statement s if all the paths to the exit point of the method starting at s must go through d.

In Fig. 1 and Fig. 3, the statement at line 11, s_{11}, is a post-dominator of the assignment statement at line 6, s_6, because all the paths to the exit point starting at that assignment statement must go through line 11. If s_{11} crashes, all the execution paths from the line 6 will never reach the exit point. In other words, if the condition at s_6 is incorrect, the execution might crash at the statement s_{11}, which is the case of this vulnerability.

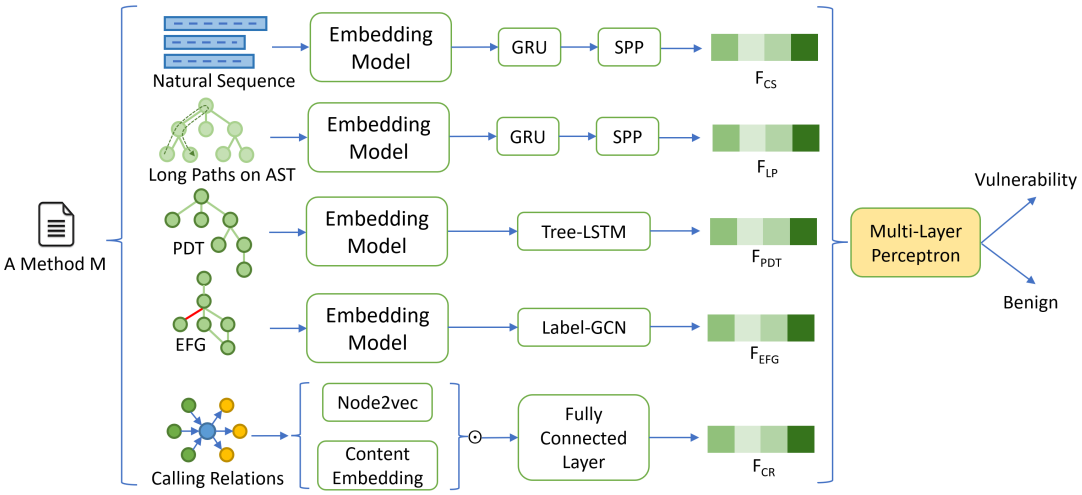

Fig. 4: DEEPVD: Architecture Overview

While being simpler than CFG and complementary to EFG, PDT is expected to help a model learn the dependencies between the executions of the statements and their post-dominators in the regular flows leading to the normal exit points in a program.

Fig. 3 displays the graph combined from the EFG and PDT for the source code in Fig. 1. The node label is the line number in the source code. The exception flow graph is from $s_3 \rightarrow s_6 \rightarrow s_9$ (solid arrows). The PDT from $s_{12} \rightarrow s_{11} \rightarrow s_6 \rightarrow s_4 \rightarrow s_3$ (dash arrows). First, the `if` statement s_6 did not have the correct condition (missing the checking on `subBox.length`). Thus, in the case of `subBox.length` being assigned with 0 at line 3, we can see that via EFG, the execution will not end at the node s_9. Via the PDT, all the paths from s_6 to the exit point of the method must go through the statements s_{11} and s_{12}. Thus, the consequence of incorrect condition at s_6 could lead to the incorrect result or crash at either s_{11} or s_{12}. That is, the crash at s_{11} could be an indication of incorrect condition at s_6. Second, following the edges in the PDT, a normal execution from s_3 to the end of the method must go through s_4, s_6, s_{11}, and s_{12}. Thus, we expect that *an ML/DL model can learn to distinguish the vulnerable code and the benign one through the conditions leading to the exception-handling points and normal exit points*. That is, the model could learn from EFG and PDT the *class-separation features* for vulnerability detection.

3) **Statement Types as Class-Separation Feature***:* We also associate with each node a label indicating the statement type. The rationale has two folds. First, from Checkmarx's vulnerability analysis [22], vulnerable code occurs at the statements with specific syntactic types including array declarations/references, pointer declarations/references, assignments, and expressions [14], [22]. Thus, tagging the nodes with the statement types would help a model distinguish better the vulnerable and benign code. Second, Part-of-Speech tagging has been shown to be effective in several downstream tasks in both NLP [23], [24] and software engineering [25].

III. APPROACH OVERVIEW

Fig. 4 displays DEEPVD's architecture. The model receives as input the source code of all methods in training. For prediction, it determines if a method is vulnerable or not. The training and predicting processes share all the components except that during training, the labels, V (vulnerability) and B (benign), are available for the methods in the training dataset. For each method, we extract the following features.

1) **Code Sequence***:* The sequence of code tokens for a method M is important in VD because it contains concrete lexical values. We use a lexer to parse and collect all the lexical code tokens in the given method. We consider a statement as a sentence and a code token as a word, and use an embedding model to produce the vector representations for all the code tokens. After obtaining the embeddings for all the tokens, we use Gated Recurrent Unit (GRU) [29] to produce the vector for the entire sequence. Then, we apply Spatial Pyramid Pooling (SPP) [30] to progressively reduce the spatial size of the vector produced by the GRU. Finally, we obtain the feature vector F_{CS} representing the code sequence in a given method M.

2) **Long Paths on AST***:* As an important part of source code, Abstract Syntax Tree (AST) carries the structural and syntactic information. Directly using the AST structure with tree-based embedding model could incur the high computation cost. Instead, we choose the long paths over the AST built from the method's body. A long path is a path that starts from a leaf node and ends at another leaf node and passes through the root node of the AST. As shown in previous works [31], [32], the AST structure can be captured and represented via the paths with certain lengths across the AST nodes. Taking the nodes in the long paths, we use an embedding model, an attention-based GRU layer [33] (for the AST structure), and then an SPP [30] to build the vector F_{LP} representing the long paths for the given method (Section IV).

3) **Post-Dominator Tree (PDT)***:* We first build the PDT according to the algorithm in Ferrante *et al.* [21]. Because

PDT is tree-structured, we choose to perform representation learning for PDT using Tree-LSTM [27], a tree-based neural network model that has been shown to perform well in source code. An alternative design would be adding the post-dominator relations into the EFG and use a graph-based neural network model to learn the representations. We do not choose that alternative because a graph-based model must learn to distinguish both types of relations in PDT and EFG.

Each node in the PDT is a statement. We consider tokens as words and statements as sentences, and use an embedding model to build the vectors for all the tokens. The embeddings will go through SPP, and then a Tree-LSTM model is used to produce the feature vector F_{PDT} for the method (Section V).

4) **Exception Flow Graph (EFG)***:* We follow the algorithm in Allen and Horwitz [20] to build the EFG for the given method. As in the PDT, each EFG node represents a statement, thus, we perform the same procedure to produce the vector for each statement using a word embedding model and a SPP layer. After this step, we obtain a graph structure in which each node (statement) is represented by a vector. Finally, we feed that graph structure as the input of a Label-GCN model to produce the feature vector F_{EFG} (Section VI).

5) **Calling Relations***:* For a method M, we consider the calling/called statements in the caller/callee methods. Together with the calling/called statements in M and the calling/called statements of M, we build a star graph. Each node in that graph represents a statement, thus, for the node content, we use the same above procedure to build the vector for the statements. We also apply network embedding Node2Vec [34] to encode the nodes. The vectors representing the node content and the calling structures are combined to produce the feature vector F_{CR} (Section VII). Finally, a multi-layer perceptron is used on all the feature vectors to classify the method.

IV. Learning Feature Embeddings for Long Paths

Because learning the embeddings for the code sequence feature is straightforward as explained in Section III, we will explain our procedure to learn the feature embeddings for the remaining feature types.

We adopted the long-path technique from Li *et al.* [35] to approximate the AST structure via the paths on the AST.

Definition 1 (Long Path)**.** *A long path in AST is a path that starts from a leaf node, ends at another leaf node, and passes through the root node of the AST.*

The reason for a path to start and end at leaf nodes is that the leaf nodes are terminal nodes with concrete lexical values. The long path from two leaf nodes captures their *structural relations* via the intermediate nodes. Thus, with a large number of long paths, we expect to approximate the AST structure. Fig. 5 shows a few examples of long paths on the AST of the vulnerable code in Fig. 1. An example long path (green color) starts from the leaf node `io_→size`, passing through the nodes `<operator>.subtraction`, `<operator>.greaterThan`, `CONTROL_STRUCTURE`, `BLOCK`, to the root node `Jp2Image :: printStructure`, and finally going down to `io_→tell`.

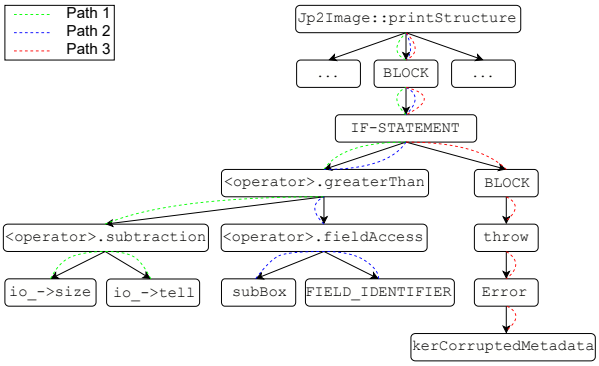

Fig. 5: Examples of Long Paths on the AST for Fig. 1

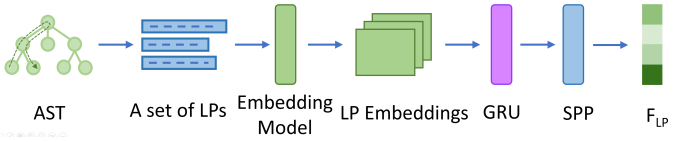

Fig. 6: Learning Long-Path Feature Embeddings

Let us explain how we build the feature embeddings for the long paths (Fig. 6). First, the set of long paths in a given method is extracted: $P = (T, D)$, where $T = \{e_1, e_2, ..., e_M\}$ is the set of tokens/nodes from AST, and $D = \{S_1, S_2, ..., S_N\}$ is a set of long paths (i.e., an ordered list of tokens/nodes). The token at the position j in S_i is denoted as $S_i[j]$ and $S_i[j] \in T$. We then use an embedding model to build the vectors for each token in T as we treat a long path as a sentence. A token $S_i[j]$ is represented as $x_j \in \mathbb{R}^d$ and a long path S_i as $H_i \in \mathbb{R}^{m \times d}$.

After obtaining the embeddings for all the tokens, we use GRU [33] to produce the vector for the entire sequence by summarizing the sequence of vectors H_i into one feature vector. We choose GRU because it can adaptively capture the dependencies of different sequence length scales. GRU also has gating units that modulate the flow of information inside the unit, however, without having a separate memory cells compared to LSTM [36]. For each GRU unit, we define a hidden gate h_t, an update gate u_t and a reset gate r_t. The transition equations of GRU are as follows:

$$h_j = (1 - u_j)h_{j-1} + u_j\tilde{h}_j, \tag{1}$$

$$u_j = \sigma(W_u x_j + U_u h_{j-1}), \tag{2}$$

$$\tilde{h}_j = \tanh(W x_j + U(r_j \odot h_{j-1})) \tag{3}$$

$$r_j = \sigma(W_r x_j + U_r h_{j-1}), \tag{4}$$

where σ is a logistic sigmoid function, $\odot$ is an element-wise multiplication. Then the last hidden state vector $h_{last} \in \mathbb{R}^d$ can be treated as the representation of S_i. Stacking all the last hidden state vectors of S_i in D, then we get the representations of D, namely Q ($Q \in \mathbb{R}^{N \times d}$).

Next, we use a Spatial Pyramid Pooling (SPP) layer to normalize it into a d-dimensional vector: $F_{LP} = SPP(Q, L)$, where L is the set of pooling sizes of SSP layer.

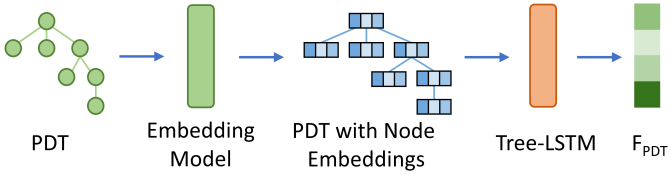

Fig. 7: Learning PDT Feature Embeddings

V. Learning Feature Embeddings for PDT

Let us explain how DeepVD learns the feature embeddings for the Post-Dominator Tree (PDT).

Definition 2 (Post-Dominator Tree)**.** *Given a Control Flow Graph G, a node p is said to post-dominate (`p-dom`) a node v if every path from v to the exit point in a method contains p. p is called a post-dominator of v.*

A node m is called the immediate post-dominator *of v if $m \neq v$; m `p-dom` v; $\forall d \neq v, d$ `p-dom` $v \Rightarrow d$ `p-dom` m. A PDT is a tree where each node's children are those nodes that it immediately post-dominates (`im-pdom`).*

PDT describes the post-domination relations among the statements in a method. PDT can be constructed by Control Flow Graph (CFG) [37]. In a CFG $G = (V, E')$, where V is a set of nodes with each node representing a statement of control predicate, and E' is a set of direct edges with each edge representing the possible flow of control between a pair of nodes. E is the subset of E' in which the edges satisfy the immediate post-domination condition in Definition 2.

Part of Fig. 3 shows the PDT: n_{12} `im-pdom` n_{11}, n_{11} `im-pdom` n_6, n_6 `im-pdom` n_4, and n_4 `im-pdom` n_3. n_4 `im-pdom` n_3 because all the execution paths from the statement s_3 to the exit point of the method must go through s_4. Node n_9 is not in the PDT because no execution path to the exit point must go through it. To build the PDT of a method, we use Joern [38] to generate CFG, then build the immediate post-dominators of each node to form the PDT.

Let us explain how we build the feature embeddings (Fig. 7). Each node in the PDT represents a statement in the given method. We first tokenize each statement into the code tokens. We then consider each statement as a sentence and each token as a word, and use an embedding technique [39] to build the embeddings for the tokens. Each node/statement now is represented by a 2D matrix $H_v \in \mathbb{R}^{m \times d}$, where m is the number of tokens in the statement, d is the dimension of token embeddings. Because the number of tokens m in a statement varies, we use a Spatial Pyramid Pooling (SPP) layer to normalize the statement representation matrix H_v into a uniform size, and reduce its total size: $x_v = SPP(H_v, L)$, where L is a set of pooling layer's sizes, $x_v \in \mathbb{R}^{d'} (d' = \sum_{l \in L} l^2)$, in which x_v is the pooled representation of the node v with the fixed length d'. For example, we use a 4-level spatial pyramid, the pooling sizes of each level are $L = \{8 \times 8, 4 \times 4, 2 \times 2, 1 \times 1\}$ (85 total), then x_v will be a vector with length 85.

After this step, we obtain the PDT $T(V, E)$ in which each statement/node is represented by a vector produced from the SPP layer. We then use the Child-Sum Tree-LSTM [27] to capture the structure of T into the final representation F_{PDT}. Let $C(v)$ denote the set of children of the node v. For each Tree-LSTM unit, we define an input gate i_v, a forget gate f_v, an output gate o_v, a memory cell c_v and a hidden state h_v. Then, the transition equations are as follows:

$$\tilde{h}_v = \sum_{k \in C(v)} h_k, \tag{5}$$

$$i_v = \sigma(W_i x_v + U_i \tilde{h}_v + b_i), \tag{6}$$

$$f_{vk} = \sigma(W_f x_v + U_f h_k + b_f), k \in C(v), \tag{7}$$

$$o_v = \sigma(W_o x_v + U_o \tilde{h}_v + b_o), \tag{8}$$

$$u_v = \tanh(W_u x_v + U_u \tilde{h}_v + b_u), \tag{9}$$

$$c_v = i_v \odot u_v + \sum_{k \in C(v)} f_{vk} \odot c_k, \tag{10}$$

$$h_v = o_v \odot \tanh(c_v) \tag{11}$$

where σ denotes the logistic sigmoid function, $\odot$ denotes element-wise multiplication; W, U are the weight matrices; bs are the bias vector parameters. Finally, the hidden state vector h_{root} of the root can be used as the feature vector F_{PDT}.

VI. Learning Feature Embeddings For EFG

For the given source code, we follow Allen and Horwitz's algorithm [20] to extract the EFG. Compared to CFG, EFG adds *exception flows*, which are the execution paths under the exception-handling conditions to the exception-handling exit points [40]. The EFG is defined as $G = (V, E)$, where V is a set of nodes with each node representing a statement, E is a set of direct edges with each edge representing the control flow edge or exception flow edge. In Fig. 3, each post-dominance edge is a reverse of a control flow edge. Among all the control flow edges, $n_6 \rightarrow n_9$ is an exception flow, which is executed only if the condition of the statement at line 6 (n_6) holds.

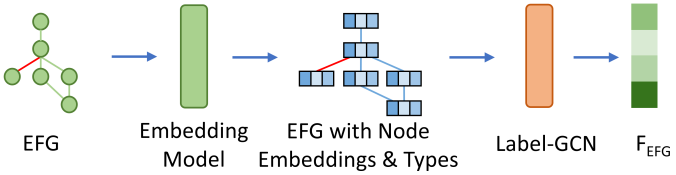

Fig. 8: Learning EFG Feature Embeddings

Fig.8 illustrates how we build the feature embeddings for EFG $G = (V, E)$. First, we use the same embedding model as building the feature embeddings for PDT in Section V. That is, we process the statements, use an embedding technique, and then use a SPP layer to obtain the vector for each statement. The difference is that instead of obtaining the tree structure of PDT whose nodes are the statement embeddings, we obtain

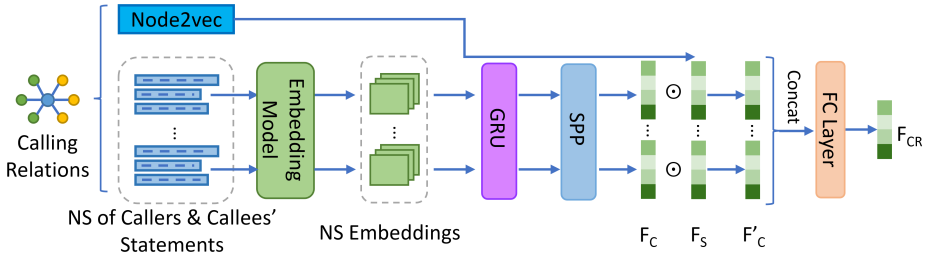

Fig. 9: Learning Feature Embeddings for Calling Relations

now the graph structure of EFG in which each node in V is replaced by an embedding $x_v \in \mathbb{R}^d$ with the fixed length d.

Moreover, we also consider the type of a statement by using the type of the AST node of the statement. For example, in Fig. 5, the type of the statement at line 6 is `IF_STATEMENT`, while the type of its condition expression is `<operator>.greaterThan`.

Next, we use Label-GCN [26] to capture the structure of G with the feature vector F_{EFG}. The idea of Label-GCN is that the knowledge of the labels surrounding a center node can help learn the feature of the center node [26]. Compared to the traditional GCN [41], Label-GCN adds a label for each node, then modify the first layer of a GCN to include the labels of the neighbors. Finally, we aggregate the features of nodes at the last layer of the GCN model using a mean operator to obtain the feature embedding F_{EFG} for the EFG.

VII. Learning Feature Embeddings for Calling Relations

For a method M under study, we capture the calling relations between M and its callers and callees. Since we consider all the methods in a project, we keep only one hop of calling relations from and to the method M. To obtain the callers/callees of M, we analyze the source code of the entire project. We represent M and its callers/callees as a star graph $G = (V, E)$, where a node $v \in V$ is a method and a directed edge $(v_1, v_2) \in E$ represents the method v_1 calling method v_2.

Note that not all the statements in a caller or callee of M are directly relevant to the potentially vulnerable statements in M. Thus, to focus on the class-separation features, instead of taking all the statements in a caller/callee, we extract only the statements in a caller/callee that are directly relevant to the method M. Specifically, we first generate the PDG for the method and its callers/callees. For a caller method of M, there is a call to $M(...)$ inside its body. We collect only the statements that have data and control dependencies via the arguments or the return value of the call to M. For a callee method N of M (i.e., M calls N), we collect in the method N all the statements having the data/control dependencies with the formal arguments of N. For all of those statements in the callers/callees, we collect the code sequences.

Fig. 9 shows the process of learning feature embeddings for the calling relations for M. We collect the code tokens of the relevant statements as explained earlier. Then, we make DeepVD learn the structure and content separately and combines the embeddings. We use Node2Vec [34] to encode the graph to get the node embedding F_S of the method M. For each caller or callee M_C, we have its code sequence representation: $P_C = (T, D)$, where $T = \{e_1, e_2, ..., e_M\}$ is the set of tokens, $D = \{S_1, S_2, ..., S_N\}$ is a set of sequences (for statements). We use an embedding technique to build the embedding of each token as a statement is considered as a sentence. Each statement now can be represented as H_i. We stack all statement embeddings to get the representation $Q \in \mathbb{R}^{n \times m \times d}$ of the entire caller or callee, where n is the number of code sequence in M_C, m is the number of tokens in a statement and d is the dimension of each token embedding. Similar to feature learning for long paths, we use GRU and SPP layer to summarize it into a 1-D representation vector F_C. Next, we multiply F_C by F_S to merge the structure and content embeddings, then concatenate them and use a fully connected layer to reduce its size. Finally, we obtain the feature vector F_{CR} for the calling relations of the callers/callees of M.

VIII. Empirical Evaluation

A. Research Questions

For evaluation, we seek to answer the following questions:

RQ1. **Comparison with state-of-the-art DL-based Vulnerability Detection Approaches.** How well does DeepVD perform compared with the DL-based VD approaches?

RQ2. **Detection on Different Types of Vulnerabilities.** How does DeepVD perform on different vulnerability types?

RQ3. **Sensitivity Analysis.** How do different components in DeepVD affect its overall performance?

RQ4. **Class Separability.** How does DeepVD provide the separability of the feature vectors of two classes (vulnerable and benign code) compared with the baselines?

An approach extracts the features from source code and converts them into a numeric feature vector/embedding to be used to train a vulnerability detection (VD) model. The performance of the approach/model depends on how separable the feature vectors of the two classes of vulnerable and benign code. This characteristic is referred to as the *separability of the classes*. The greater the separability, the easier it is for a model to detect vulnerabilities. Thus, the answer to RQ4 helps show how well DeepVD provides the separability of the classes, which has impacts on its performance. The answer to RQ4 would show that the high accuracy in VD can be attributed to DeepVD's design for feature embeddings.

RQ5. **Learning Relevant Features.** How well DEEPVD assign the importance to the vulnerability-related code features to make its prediction?

It is important to understand what features a model uses to make its prediction. The answer to RQ5 helps understand whether a model uses the vulnerability-relevant features by assigning a greater importance level, thus showing that DEEPVD learns well vulnerability-relevant features for VD.

B. Experimental Methodology

1) Data Collection: We collected a dataset of vulnerabilities from the CVE database [42]. Each CVE entry provides the bug-fixing commit links to the code patches for each vulnerability. Our mining tool first verified the link from a CVE entry to make sure that it contains the correct project name. If the commit number is available in the link, our tool verifies that the link refers to the correct commit. We use Git [43] to clone the source code repository to a local machine. For each bug-fixing commit, we use `git checkout` command to collect its buggy version V_{bug} and fixed version V_{fix}. Each modified method M in V_{fix} is labeled as benign, and M' in V_{bug} is labeled as vulnerable. Finally, the dataset contains 303 large and popular C/C++ projects covering the CVEs from 2000-2021 with 13,130 vulnerable methods. We use Scitool Understand (version 6.0.1077-Linux-64bit) [44] to analyze each project and collect the callers/callees of each method.

2) Experimental Setup and Procedures:

RQ1. Comparison with State-of-the-Art DL-based Vulnerability Detection Approaches

Baselines. We compare DEEPVD with the following DL-based baselines: **VulDeePecker** [13], **Devign** [16], **SySeVR** [14], **Russell** *et al.* [15], **Reveal** [17], and **IVDetect** [18].

Procedure. We use 13,130 vulnerable methods and randomly select the same number of non-vulnerable methods from the fixed version projects, to build a dataset with the vul:benign ratio of 1:1. We randomly split all the data 80%, 10%, 10% on the project basis without changing the vul:benign ratio for training, tuning, and testing, respectively. We also evaluated DEEPVD in the real-world vulnerability setting as well with a 9:1 benign to vulnerability ratio. In this experiment, we randomly selected 10% of the vulnerable instances in the test set 10 times, and finally took the average of accuracies.

Parameter Tuning. We use AutoML [45] on all models to automatically tune hyper-parameters on the tuning dataset. We tuned DEEPVD with the parameters *batch size, hidden size, learning rate, dropout rate*. The hyper-parameters we tuned for the baselines can be found in their papers. We choose the hyper-parameters with the best performance for a model.

Evaluation Metrics. We use the following metrics to measure the effectiveness of a model: $Recall = \frac{TP}{TP+FN}$, $Precision = \frac{TP}{TP+FP}$, and $Fscore = \frac{2*Recall*Precision}{Recall+Precision}$. where TP = True Positives; FP = False Positives; FN = False Negatives; TN = True Negatives.

RQ2. Detection on Different Types of Vulnerabilities.

We use the same trained model, training and testing datasets, baselines, and evaluation metrics as in RQ1. However, we tabulate the results based on different types of vulnerabilities. The type of each vulnerability is recorded in CVE Details from which we mined our data.

TABLE I: Comparison with DL-based VD Approaches (RQ1)

Approach	Precision	Recall	F-score
VulDeePecker	0.55	0.77	0.64
SySeVR	0.54	0.74	0.63
Russell *et al.*	0.54	0.72	0.62
Devign	0.56	0.73	0.63
Reveal	0.62	0.69	0.65
IVDetect	0.54	0.77	0.67
DEEPVD	**0.70**	**0.89**	**0.78**

RQ3. Sensitivity Analysis. We conduct an ablation study to evaluate the impact of different components in DEEPVD on its performance: PDT, EFG, and Calling Relations. We built a base model with only code sequences (NS) and long paths (LP), incrementally added each of those components to the base model, and compared the results to evaluate the impacts. We also added to the base model (NS+LP) different types of traditional graph representations to compare with DEEPVD to evaluate our proposed features in PDT, EFG, and CR.

RQ4. Class Separability. We use t-SNE plots to investigate the class separability. t-SNE [46], a dimensionality reduction technique, visualizes high-dimensional datasets into a smaller-dimensional feature space. The separation in that space is an indication of whether the vulnerability and benign classes are distinguishable. We also compute the Euclidean distance between the centroid of each of the two classes. The larger the distance the greater class separation a model exhibits [47].

RQ5. Learning Relevant Features. To find whether a model uses vulnerability-relevant features in its prediction, we use Lemna [19], an explainable AI technique, to identify the importance of features. Lemna assigns an entity in the input a score indicating a contribution of that entity toward the decision of a model. Similar to Reveal [17], for the graph-based models, we use the activation value of each vertex in the graph to obtain the feature importance.

IX. EMPIRICAL RESULTS

A. Comparison with State-of-the-Art, DL-based, Vulnerability Detection Approaches (RQ1)

As seen in Table I, DEEPVD improves over all the baselines in all the metrics. Specifically, DEEPVD relatively improves over the baseline models from 13%– 29.6% in Precision, from 15.6%–28.9% in Recall, and from 16.4%–25.8% in F-score.

To understand how DEEPVD is complementary to the baselines, we compute how the top-100 result from DEEPVD overlap with that from the top-performing baseline, IVDetect [18]. As seen in Fig. 10, DEEPVD can detect 19 vulnerable methods that IVDetect missed, while IVDetect can detect only 10 vulnerable methods that DEEPVD missed. Both detected the same 25 vulnerabilities. This result shows that DEEPVD not only detects more vulnerabilities than the baseline approaches, but also is complementary to them well.

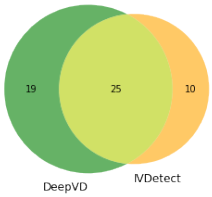

Fig. 10: Overlapping Results between DEEPVD and IVDetect

```
1 BIO *PKCS7_dataDecode(PKCS7 *p7, EVP_PKEY *pkey, BIO
      *in_bio, X509 *pcert) {
2   ...
3   if (evp_cipher != NULL) {
4     ...
5     if (pcert == NULL) {
6       for (i = 0; i < sk_PKCS7_RECIP_INFO_num(rsk); i++) {
7         ri = sk_PKCS7_RECIP_INFO_value(rsk, i);
8         if (pkcs7_decrypt_rinfo(&ek, &eklen, ri, pkey) < 0)
9           goto err;
10        ERR_clear_error();
11      }
12    } else {...}
13 }
```

Fig. 11: CVE-2019-1563: A vulnerable code in OpenSSL with 186 lines of code (after removed comments and empty lines). ReVeal and IVDetect failed to detect it but DEEPVD detected.

Examining the results, we found that DEEPVD often performed better than the best baselines, Reveal [17] and IVDetect [18], in the code with complex PDGs (which IVDetect uses as a key feature for detection) and CPGs (which Reveal uses for detection). Moreover, DEEPVD detects vulnerabilities well in the code with improper exception/error-handling.

Fig. 11 shows the vulnerable code from OpenSSL that was reported in CVE-2019-1563. The code (186 lines of code) has the PDG with 145 nodes and 477 edges, and the CPG with 622 nodes and 1,393 edges. In contrast, PDT+EFG has 145 nodes and 295 edges (including the error-handling flow from line 8 to lines 9–10). Thus, complex PDG or CPG produce much noise for IVDetect and Reveal, which missed this vulnerability. DEEPVD with its PDT+EFG features detects well this type of vulnerability with improper error-handling (e.g., the incorrect code at line 8, which was fixed with an additional condition).

In the setting of 9:1 benign-to-vulnerability ratio, DEEPVD achieves F-score of 45.3% while the best baseline, IVDetect, achieves F-score of 25.4%. Thus, DEEPVD exhibits a consistent improvement trend over IVDetect while outperforming it by 78.3% in F-score. This is lower than that in Table I due to the imbalance data between benign and vulnerability classes.

B. Detection on Different Types of Vulnerabilities (RQ2)

Table II shows DEEPVD's result for the most five popular vulnerability types in the dataset. For *Denial of Service* (DoS), with the highest number of vulnerabilities (1,636) in our dataset, it detects well with the F-score of 70% (second highest F-score value among the top-5 vulnerability types behind *Obtain Information* with 75%). For *Execute Code*, it achieves only 58% in F-score with high recall (95%) but lower precision

TABLE II: DEEPVD's Result on Vulnerability Types (RQ2)

	Vulnerability Type	TN	FP	FN	TP	Total	**Precision**	**Recall**	**F-score**
1	Denial Of Service	424	490	64	658	1,636	0.57	0.91	0.70
2	Overflow	225	371	28	340	964	0.48	0.92	0.63
3	Execute Code	129	279	11	202	621	0.42	0.95	0.58
4	Memory corruption	102	190	9	162	463	0.46	0.95	0.62
5	Obtain information	63	45	7	76	191	0.63	0.92	0.75

TABLE III: Contributions of Different Features (RQ3)

Variant	**Precision**	**Recall**	**F-score**
1. NS + LP	0.55	0.76	0.63
2. NS + LP + PDT	0.64	0.80	0.71
3. NS + LP + EFG	0.64	0.83	0.72
4. NS + LP + PDT + EFG	0.66	0.85	0.74
5. NS + LP + PDT + EFG + CR	0.69	0.87	0.77
6. NS + LP + PDT + EFG + CR + Stmt_Types	0.70	0.89	0.78

NS: Natural code Sequence; LP: Long Path; PDT: Post-Dominator Tree; EFG: Exception Flow Graph; CR: Calling Relations

(42%). Examining this, we found that DEEPVD did not handle well the cases with the manipulation of string literals/values.

For the other types of vulnerabilities (not shown), the F-score values are also high. For example, DEEPVD detected with F-score of 83%, 73%, and 71% for *Cross-site scripting*, *Restriction bypass*, and *Privilege Gaining*, respectively.

Examining the results, we found that *many Denial Of Service (DOS) vulnerabilities are involved with the improper exception/error-handling*. For example in Fig. 11, the DoS vulnerability involves the mis-handling of the error in line 8.

C. Sensitivity Analysis (RQ3)

1) **Contributions of Different Features:** Table III shows the performance of different variant models, as we incrementally added one feature at a time to the base model with natural code sequences and long paths (`NS+LP`).

Comparing the rows (1) and (2), adding the PDT as features, there are the relative improvements of 16.4%, 5.2%, and 11.4% in Precision, Recall, and F-score, respectively. This result shows that PDT helps detect more precisely and more completely the vulnerable code than `NS+LP`. This is reasonable because `NS+LP` does not consider any execution flow at all, while PDT considers the post-dominator relations.

Comparing the rows (1) and (3), adding the EFG as features creates the relative improvements of 16.4%, 6.3%, and 13.2% in Precision, Recall, and F-score, respectively. This result shows that EFG helps more than PDT in covering more vulnerabilities and EFGs are more relevant to the vulnerabilities.

Comparing the rows (4) and (2), adding EFG to `NS+LP+PDT` creates the improvements from 64% to 66% in Precision, from 80% to 85% in Recall, from 71% to 74% in F-score. Comparing (4) and (3), the improvement in Recall is from 83% to 85%. Thus, EFG contributes to better recall more than PDT.

Comparing the rows (5) and (4), we see that adding calling relations has the improvements from 66% to 69% in Precision, from 85% to 87% in Recall, and from 74% to 76% in F-score. Calling relations among the methods provide the global

TABLE IV: Impacts of PDT+EFG and other graphs (RQ3)

Variant	Precision	Recall	F-score
1. NS + LP	0.55	0.76	0.63
2. NS + LP + CFG	0.63	0.81	0.71
3. NS + LP + DFG	0.63	0.79	0.70
4. NS + LP + PDG	0.68	0.76	0.72
5. NS + LP + CPG	0.58	0.83	0.68
6. NS + LP + PDT + EFG	0.66	0.85	0.74
7. NS + LP + PDT + EFG + CR	0.69	0.87	0.77
8. NS + LP + PDT + EFG + CR + Stmt_Types	0.70	0.89	0.78

context among the methods while PDT+EFG provides the internal context within individual methods. This result shows that *global context complements to the internal context* for VD.

Finally, comparing the row (6) (DEEPVD) and the row (5), the statement types help further improve both Precision and Recall, as well as F-score as a result.

2) **Comparison of the Impacts of PDT+EFG and Traditional Graphs:** Table IV shows the performance of different variants when we added different traditional program graphs to the base model NS+LP. We compare them with the variants of DEEPVD using PDT+EFG and PDT+EFG+CR.

First, comparing the row (7) and each row (2)–(5), we can see that PDT+EFG provides better features than any of the traditional program graphs (CFG, DFG, PDG, and CPG).

Second, comparing the rows (2)–(4) with (1), we can see that the impacts of CFG, DFG, and PDG on improving the base NS+LP are similar (0.71, 0.70, 0.72 in F-score), while the variant with PDG achieves highest precision with lowest recall among them. That could be explained by the fact that PDG provides more details on dependencies than CFG and DFG, thus, the corresponding variant model is more precise but would miss more vulnerabilities. In contrast to the case with PDG, the variant model NS+LP+CPG produces lower precision, but higher recall, leading to lower F-score than the variant models with CFG, DFG, and CPG. That could be due to that CPG consists of more program dependencies and properties, leading to covering more vulnerabilities. However, CPG could add more noises, leading to imprecision in VD.

Third, comparing the variant model at row (6) and the previous ones, the combination of PDT and EFG contributes more than each of the traditional graphs. Compared with the variant with PDG, despite that having slightly lower precision (0.66 vs 0.68), the variant with PDT+EFG has higher recall (0.85 vs 0.76), leading to higher F-score (0.74). This is due to the fact that (1) PDT+EFG is less complex than PDG, and (2) PDT+EFG has more class-separation features, leading much higher recall and higher F-score. Finally, DEEPVD with PDT+EFG, calling relations, and statement types, achieves the highest in all metrics. The calling relations (CR) add the global context among the methods, while together with PDT+EFG, the statement types help with the internal context for VD.

D. Class Separability (RQ4)

1) **t-SNE Plots:** Fig. 12 illustrates the t-SNE plots for DEEPVD and the baseline models when we projected the

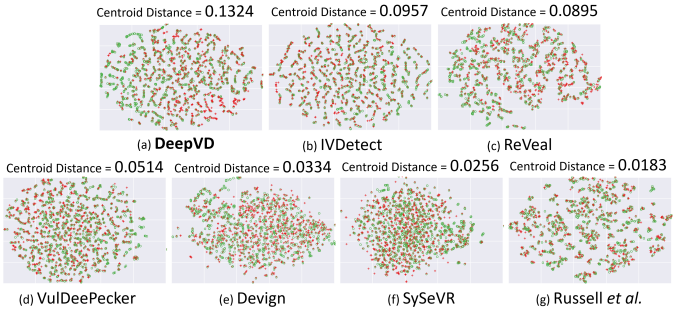

Fig. 12: t-SNE Plots Illustrating the Separation between Vulnerability (denoted by +) and Benign (denoted by o) Classes

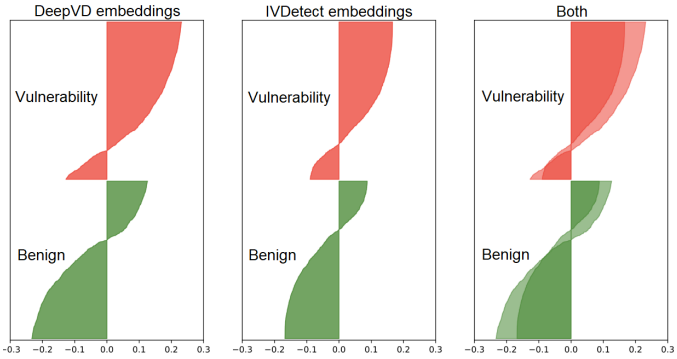

Fig. 13: Comparison of Silhouette Plots for Embeddings

vectors for all 4,606 instances (2,303 vulnerable and 2,303 benign ones). As seen in Fig. 12b–g, the baseline models exhibit a significant degree of overlap in the feature space between the two classes of vulnerability and benign code. This is also reflected by the relatively low distances between the centroids for the baseline models. The centroid distances for the baseline models range from 0.01–0.09, with the lowest being from Russell *et al.* (0.01). All three graph-based models (IVDetect, ReVeal, and Devign) have higher centroid distances of 0.09, 0.08 and 0.03. As seen in Fig. 12b–g, the vulnerability class is almost inseparable from the benign class in the feature space. The lack of class-separation explains why the baseline models (despite that some of them using graph neural networks) did not perform well in vulnerability detection.

With the designed features emphasizing on class separation, DEEPVD has improvements over the baseline models. Compare the centroid distances with the other graph-based neural network approaches (Fig. 12), DEEPVD exhibits the higher separation between the vulnerable and benign classes: 38% and 48% relatively higher than the top two baseline graph-based models in IVDetect and ReVeal, respectively. Compared with VulDeePecker, Devign, SySeVR, and Russell *et al.*, DEEPVD has 2.57x, 3.96x, 5.17x, and 7.23x, respectively, higher separation between the vulnerability and benign classes. As seen in Fig. 12a, for DEEPVD, the benign class (green) tends to be on the left side, while the vulnerability class (red) is on the right side. Thus, DEEPVD has better class separability than the baselines, leading to better classification.

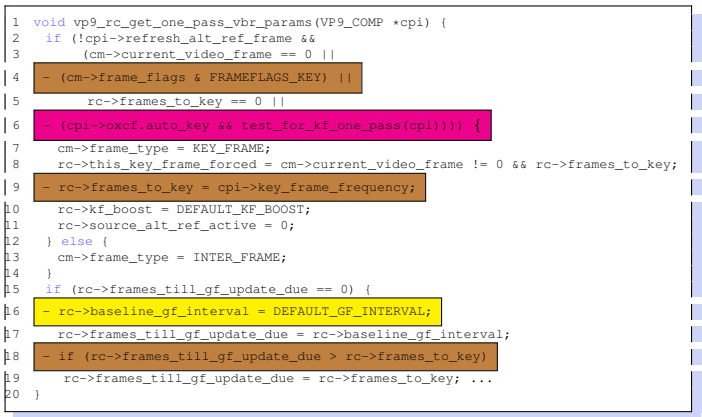

Fig. 14: Contributions of Different Code Components in Correct Classification of Vulnerability by DEEPVD (RQ5)

2) **Silhouette Plots:** We also used the silhouette plot [48] to present the data points for those above embeddings for DEEPVD and the best baseline, IVDetect [18]. The silhouette coefficient value (X-axis) is a measure of how similar an object is to its own class compared to other classes. The silhouette coefficient value is in [-1,1], where a high value indicates that an object is well matched to its own class and poorly matched to neighboring classes. The lines are sorted from largest to smallest and drawn from top to bottom, creating a knife shape. If most objects have high values, the class configuration is appropriate. For VD, that corresponds to *better class-separation*, facilitating VD. If many points have low or negative values, the class separation is poor.

Considering the overlap between two plots in Fig. 13, the knife shapes from DEEPVD for both classes (vulnerability and benign) are wider and have less negative values than those from IVDetect. Specifically, the *average silhouette score in* DEEPVD *is 0.025, while that of IVDetect is 0.0076*. Thus, this result shows that DEEPVD has better class-separation, leading to better performance than the baseline models.

E. Learning Relevant Features (RQ5)

To visualize the feature importance, we use a heatmap to highlight the most to least important statements decided by Lemna. Fig. 14 shows a method in the `libvpx` project reported in CVE-2016-1621. All five vulnerable lines of code were ranked in the top-5 statements contributing to the correct VD by DEEPVD: the line 6 (in magenta) has highest score; the lines 4, 9, and 18 have the same range of scores (in brown), which is higher than the score for line 16 (in yellow). The non-color lines are given the lowest scores by Lemna (i.e., contributing the least to the model's decision). Thus, DEEPVD *uses vulnerability-relevant features in its prediction*.

We also computed the list L of the top-5 statements with the highest Lemna scores for each method M that our model correctly detected as vulnerabilities. If L overlaps with the set of the fixed/vulnerable statements in M, we count it as a hit; otherwise, it is a miss. Accuracy is defined as the ratio between the hits and the total number of correctly detected methods. We report that DEEPVD was able to use the vulnerability-relevant statements in 84% of the cases of correct predictions.

Threats to Validity: We only tested on the vulnerabilities in C and C++ code. DEEPVD can apply to other programming languages. We tried our best to tune the baselines on same dataset for fair comparisons. DEEPVD does not handle well the vulnerabilities involving string literals, overflow/underflow, and memory corruption because it does not consider values.

F. Complexity

Despite the different moving parts from the third-party analysis tools, DEEPVD only has about 875K parameters. It took 23m13s per epoch for training on an Nvidia Titan RTX GPU, and 0.0165 seconds for the detection of an instance.

X. Related Work

The traditional **program analysis** (PA)-based VD tools [1]–[10] leverage static and dynamic analysis techniques to provide the specific rules for each vulnerability type. Second, **software mining** approaches leverage known vulnerability patterns to discover possible vulnerable code (FlawFinder [1], RATS [2], ITS4 [3], Checkmarx [4], Fortify [5] and Coverity [6]).

Machine Learning (ML) including Deep learning (DL) has been applied to detect vulnerabilities [49]–[54]. Harer *et al.* [55] train an RNN to detect vulnerabilities. Lin *et al.* [56] automatically learns high-level representations of functions based on AST for VD. Russell *et al.* [15] combine the neural feature representations of functions with random forest as a classifier. Harer *et al.* [57] compared the effectiveness in VD of using source code and the compiled code. VulDeePecker [13] uses a RNN trained on program slices from API calls for VD. SySeVR [14] expands VulDeePecker by including the program slices from more syntactic units: arrays, pointers, and arithmetic expressions. Both of them do not consider exception flows. Devign [16] uses Gated Graph Recurrent Layers on CPG, PDG, CFG, AST and code sequences. As shown, DEEPVD performs better than Devign and our class-separation graph features is more useful than CPG/PDG/CFG. Reveal [17] uses CPG with GGNN, which is not as effective as EFG+PDT. IVDetect [18] directly uses PDG with GCN. LineVul [58], a transformer-based vulnerability prediction, works at the line level and improves significantly over IVDetect.

XI. Conclusion

We propose DEEPVD, a graph-based neural network VD model that emphasizes on class-separation features between vulnerability and benign code. DEEPVD leverages three types of class-separation features at different levels of abstraction: statement types, Post-Dominator Tree (covering regular flows), and EFG (covering exception/error-handling flows). Empirical results show that our proposed class-separation features contribute significantly in improving the accuracy in VD over the existing ML/DL models from 16.4% to 25.8% in F-score.

Acknowledgments

This work was supported in part by the US National Science Foundation (NSF) grants CNS-2120386.

REFERENCES

[1] Flawfinder. [Online]. Available: http://www.dwheeler.com/FlawFinder

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

[3] J. Viega, J.-T. Bloch, Y. Kohno, and G. McGraw, "Its4: A static vulnerability scanner for c and c++ code," in *Proceedings 16th Annual Computer Security Applications Conference (ACSAC'00)*. IEEE, 2000, pp. 257–267.

[4] Checkmarx. [Online]. Available: https://www.checkmarx.com/

[5] Hp fortify. [Online]. Available: https://www.hpfod.com/

[6] Coverity. [Online]. Available: https://scan.coverity.com/

[7] Cwe-120: Buffer overflow. [Online]. Available: https://cwe.mitre.org/data/definitions/120.html

[8] Cwe-89: Sql injection. [Online]. Available: https://cwe.mitre.org/data/definitions/89.html

[9] Cwe-79: Cross-site scripting. [Online]. Available: http://cwe.mitre.org/data/definitions/79.html

[10] Cwe-290: Authentication bypass by spoofing. [Online]. Available: https://cwe.mitre.org/data/definitions/290.html

[11] N. H. Pham, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Detection of recurring software vulnerabilities," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 447–456. [Online]. Available: https://doi.org/10.1145/1858996.1859089

[12] B. Bowman and H. H. Huang, "VGRAPH: A Robust Vulnerable Code Clone Detection System Using Code Property Triplets," in *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2020, pp. 53–69.

[13] 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.

[14] 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.

[15] 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.

[16] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Advances in Neural Information Processing Systems*, 2019, pp. 10 197–10 207.

[17] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet?" *IEEE Transactions on Software Engineering*, vol. 48, no. 09, pp. 3280–3296, sep 2022.

[18] 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*. New York, NY, USA: Association for Computing Machinery, 2021, p. 292–303. [Online]. Available: https://doi.org/10.1145/3468264.3468597

[19] W. Guo, D. Mu, J. Xu, P. Su, G. Wang, and X. Xing, "Lemna: Explaining deep learning based security applications," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 364–379. [Online]. Available: https://doi.org/10.1145/3243734.3243792

[20] M. Allen and S. Horwitz, "Slicing java programs that throw and catch exceptions," in *Proceedings of the 2003 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, ser. PEPM '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 44–54. [Online]. Available: https://doi.org/10.1145/777388.777394

[21] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, p. 319–349, jul 1987. [Online]. Available: https://doi.org/10.1145/24039.24041

[22] Checkmarx. [Online]. Available: https://checkmarx.com/

[23] M. Sun and J. R. Bellegarda, "Improved pos tagging for text-to-speech synthesis," in *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2011, pp. 5384–5387.

[24] Ankita and K. A. Abdul Nazeer, "Part-of-speech tagging and named entity recognition using improved hidden markov model and bloom filter," in *2018 International Conference on Computing, Power and Communication Technologies (GUCON)*, 2018, pp. 1072–1077.

[25] M. Izadi, R. Gismondi, and G. Gousios, "CodeFill: Multi-Token Code Completion by Jointly Learning from Structure and Naming Sequences," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 401–412. [Online]. Available: https://doi.org/10.1145/3510003.3510172

[26] C. Bellei, H. Alattas, and N. Kaaniche, "Label-GCN: An effective method for adding label propagation to graph convolutional networks," *arXiv preprint arXiv:2104.02153*, 2021.

[27] K. S. Tai, R. Socher, and C. D. Manning, "Improved semantic representations from tree-structured long short-term memory networks," *arXiv preprint arXiv:1503.00075*, 2015.

[28] (2022) DeepVD. [Online]. Available: https://github.com/deepvd2022/deepvd2022

[29] D. Tang, B. Qin, and T. Liu, "Document Modeling with Gated Recurrent Neural Network for Sentiment Classification," in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Sep. 2015, pp. 1422–1432. [Online]. Available: https://www.aclweb.org/anthology/D15-1167

[30] K. He, X. Zhang, S. Ren, and J. Sun, "Spatial pyramid pooling in deep convolutional networks for visual recognition," *CoRR*, vol. abs/1406.4729, 2014. [Online]. Available: http://arxiv.org/abs/1406.4729

[31] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *CoRR*, vol. abs/1803.09473, 2018. [Online]. Available: http://arxiv.org/abs/1803.09473

[32] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Accurate and efficient structural characteristic feature extraction for clone detection," in *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, ser. FASE. Springer-Verlag, 2009, pp. 440–455.

[33] K. Cho, B. van Merrienboer, Ç. Gülçehre, F. Bougares, H. Schwenk, and Y. Bengio, "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation," *CoRR*, vol. abs/1406.1078, 2014. [Online]. Available: http://arxiv.org/abs/1406.1078

[34] A. Grover and J. Leskovec, "Node2vec: scalable feature learning for networks," in *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, 2016, pp. 855–864.

[35] Y. Li, S. Wang, T. N. Nguyen, and S. Nguyen, "Improving bug detection via context-based code representation learning and attention-based neural networks," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, oct 2019. [Online]. Available: https://doi.org/10.1145/3360588

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

[37] F. E. Allen, "Control flow analysis," *ACM Sigplan Notices*, vol. 5, no. 7, pp. 1–19, 1970.

[38] (2022) Joern. [Online]. Available: https://joern.io/

[39] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Empirical Methods in Natural Language Processing (EMNLP)*, 2014, pp. 1532–1543. [Online]. Available: http://www.aclweb.org/anthology/D14-1162

[40] S. Thummalapenta and T. Xie, "Mining exception-handling rules as sequence association rules," in *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009, pp. 496–506.

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

[42] "CVE security vulnerability database," https://www.cvedetails.com/, accessed: 2022-08-27.

[43] "Git," https://git-scm.com/, accessed: 2022-08-27.

[44] "Understand: An IDE and Static Code Analysis Tool by SciTools," https://www.scitools.com/, accessed: 2022-08-27.

[45] Microsoft, "Neural network intelligence." https://github.com/microsoft/nni, last Accessed August 28th, 2020.

[46] L. van der Maaten and G. Hinton, "Visualizing Data using t-SNE," *Journal of Machine Learning Research*, vol. 9, no. 86, pp. 2579–2605, 2008. [Online]. Available: http://jmlr.org/papers/v9/vandermaaten08a.html

[47] C. Mao, Z. Zhong, J. Yang, C. Vondrick, and B. Ray, "Metric learning for adversarial robustness," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019.

[48] "Silhouette (clustering)," https://en.wikipedia.org/wiki/Silhouette_(clustering), last Accessed March 15, 2022.

[49] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Predicting vulnerable software components via text mining," *IEEE Transactions on Software Engineering*, vol. 40, no. 10, pp. 993–1006, 2014.

[50] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *Proceedings of the 14th ACM conference on Computer and communications security*, 2007, pp. 529–540.

[51] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE transactions on software engineering*, vol. 37, no. 6, pp. 772–787, 2010.

[52] S. Neuhaus and T. Zimmermann, "The Beauty and the Beast: Vulnerabilities in Red Hat's Packages." in *USENIX Annual Technical Conference*, 2009.

[53] F. Yamaguchi, M. Lottmann, and K. Rieck, "Generalized vulnerability extrapolation using Abstract Syntax Trees," in *Proceedings of the 28th Annual Computer Security Applications Conference*, 2012, pp. 359–368.

[54] F. Yamaguchi, F. Lindner, and K. Rieck, "Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning," in *Proceedings of the 5th USENIX conference on Offensive technologies*, 2011, pp. 13–13.

[55] J. Harer, O. Ozdemir, T. Lazovich, C. Reale, R. Russell, and L. Kim, "Learning to repair software vulnerabilities with generative adversarial networks," in *Advances in Neural Information Processing Systems*, 2018, pp. 7933–7943.

[56] G. Lin, J. Zhang, W. Luo, L. Pan, and Y. Xiang, "Poster: Vulnerability discovery with function representation learning from unlabeled projects," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2539–2541.

[57] 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, and P. M. Ellingwood, "Automated software vulnerability detection with machine learning," *arXiv preprint arXiv:1803.04497*, 2018.

[58] M. Fu and C. Tantithamthavorn, "LineVul: A Transformer-Based Line-Level Vulnerability Prediction," in *Proceedings of the 19th International Conference on Mining Software Repositories*, ser. MSR '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 608–620. [Online]. Available: https://doi.org/10.1145/3524842.3528452