Enhancing Deep Learning-based Vulnerability Detection by Building Behavior Graph Model

Bin Yuan[1,4,†], Yifan Lu[1,†], Yilin Fang[1,†], Yueming Wu[2,*], Deqing Zou[1,†], Zhen Li[1,†], Zhi Li[1,†], Hai Jin[3,†]
[1] School of Cyber Science and Engineering, Huazhong University of Science and Technology, China
[2] Nanyang Technological University, Singapore
[3] School of Computer Science and Technology, Huazhong University of Science and Technology, China
[4] Shenzhen Huazhong University of Science and Technology Research Institute, China

***Abstract*—Software vulnerabilities have posed huge threats to the cyberspace security, and there is an increasing demand for automated *vulnerability detection* (VD). In recent years, *deep learning-based* (DL-based) vulnerability detection systems have been proposed for the purpose of automatic feature extraction from source code. Although these methods can achieve ideal performance on synthetic datasets, the accuracy drops a lot when detecting real-world vulnerability datasets. Moreover, these approaches limit their scopes within a single function, being not able to leverage the information between functions. In this paper, we attempt to extract the function's abstract behaviors, figure out the relationships between functions, and use this global information to assist DL-based VD to achieve higher performance. To this end, we build a Behavior Graph Model and use it to design a novel framework, namely *VulBG*. To examine the ability of our constructed Behavior Graph Model, we choose several existing DL-based VD models (*e.g., TextCNN*, *ASTGRU*, *CodeBERT*, *Devign*, and *VulCNN*) as our baseline models and conduct evaluations on two real-world datasets: the balanced FFMpeg+Qemu dataset and the unbalanced Chrome+Debian dataset. Experimental results indicate that *VulBG* enables all baseline models to detect more real vulnerabilities, thus improving the overall detection performance.**

***Index Terms*—Vulnerability Detection, Behavior Graph, Deep Learning**

I. Introduction

With the rapid development of modern software, there are increasingly security incidents [1], [2] caused by software vulnerabilities, such as hacking, botnet attacks, and user information leakage, which have brought a huge serious threat to software systems. It was reported that 75% of the open-source codebases — a critical component in modern software supply chain — contained vulnerabilities, with half of the codebases containing high-risk vulnerability [3]. Therefore, it is urgent to carry out large-scale and accurate software vulnerability detection methods to better protect software security, as well as the software supply chain.

Essentially, there are two categories of automated *vulnerability detection* (VD) techniques: dynamic analysis and static analysis. It is well-known that the dynamic-analysis-based VD methods suffer from the problem of time overheads and limited path coverage. Moreover, dynamic methods usually require running the system under inspection. Such limitations make it not suitable for VD on a large scale. By contrast, static methods can scale to large-scale software analysis, yet it usually suffers from high false negatives and false positives. Specifically, the similarity-based static analysis performs well at detecting vulnerabilities caused by code cloning but failed in detecting other bugs. The pattern-based static analysis depends on the rules predefined by experts for identifying the vulnerabilities. However, the rules need to be constantly updated to detect new bugs. At last, the dataflow-based static method generally uses approximations to ensure convergence, which guarantees no false negatives at the expense of high false positives.

In recent years, since *deep learning* (DL) can automatically extract features from code, different DL-based VD approaches [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18] have been proposed and proved to be effective. To achieve scalable vulnerability analysis, some approaches treat source code as text and turn the vulnerability detection task into a *natural language processing* (NLP) problem [4], [5], [8], [9], [16], [17], [18]. For example, *VulDeePekcer* [4] applies static analysis to extract the program slices and trains a *bidirectional long short term memory* (Bi-LSTM) model to detect slice-level vulnerabilities. In practice, although text-based VD methods can achieve high scalability, their detection performance is not ideal since they ignore the program semantics. To mitigate the issue, researchers [10], [11], [12], [13], [14], [15], [19] intend to extract the intermediate representations such as *abstract syntax tree* (AST) and *program dependency graph* (PDG) to retain the program details. For instance, *Funded* [19] first conducts a complex program analysis to extract augmented AST of source code, and then uses a *graph neural network* (GNN) to train a vulnerability detector. Due to the consideration of program semantics, these semantics-based methods perform better than the text-based tools on synthetic vulnerability datasets. However, according to a recent study [20], not only text-based methods but also semantics-based approaches suffer from poor performance when detecting real-world vulnerabilities.

Meanwhile, we also observe that almost all existing DL-based VD methods leave the semantics extraction work to the *neural network* (NN) with little in-depth investigation of the

† Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab.
* Yueming Wu is the corresponding author.

code's feature. This is inappropriate as vulnerabilities tend to be in a small part of the function. Passing the whole function to a neural network model introduces huge information irrelevant to the vulnerability. Besides, existing approaches limit their focus on one function, ignoring underlying connections between functions. A function is a collection of code that implements certain functionalities. Even if two functions implement totally different functionalities, there might be common logic, algorithms, and programming patterns within their subtasks. Further, we define behavior as such logic, algorithm, or programming pattern in the code that implements a certain functionality and treat each function as a set of behaviors. Hence, the connections between the functions can be addressed by the behaviors they share. A vulnerability could exist in one behavior or a set of behaviors. By leveraging connections between functions, functions with similar vulnerabilities can be more easily detected.

In this paper, we build a Behavior Graph Model to connect the behaviors of different functions and use it to enhance the detection ability of existing DL-based methods. In specific, we first perform program slicing to split a function into a set of slices and regard each slice as a kind of behavior of the function. Then the graph is constructed based on the similarity computation of different program slices (*i.e.,* behaviors) in different functions. After obtaining the global Behavior Graph, we apply a node embedding technique to convert each node (*i.e.,* function) into a vector representation as the Behavior Feature of the function. These vectors can be used to boost the capability of existing DL-based VD models.

To this end, we implement a novel framework named *VulBG*. We evaluate *VulBG* with different real-world datasets: FFMpeg+Qemu dataset proposed by [15] with vulnerable rate of 45.6% and Chrome+Debian dataset proposed by [20] with vulnerable rate of 9.7%. Experimental results show that our proposed Behavior Graph Model enables existing methods to detect more vulnerabilities on both balanced and unbalanced dataset. Moreover, we also make comparison to five state-of-the-art DL-based VDs of different methods (a.k.a, the baseline models): Text-based model *TextCNN* [21], AST-based model *ASTGRU* [22], pretrained-based model *CodeBERT* [23], and graph-based models *Devign* [15] and *VulCNN* [24], with evaluations on FFMpeg+Qemu dataset and Chrome+Debian dataset. Experimental results show that the performance of the Behavior Graph Model itself is good enough to defeat other models. Moreover, *VulBG* can further improve the performance of all the baseline models after applying our Behavior Graph Model to them.

In summary, this paper makes the following contributions:

- We propose a novel idea that can extract abstract behaviors from the source code of a function, and construct a Behavior Graph that correlates the behavior of different functions to assist vulnerability detection.
- We implement *VulBG* framework to improve the performance of vulnerability detection by combining Behavior Graph with other DL-based VD models.
- We evaluate *VulBG* and select five state-of-the-art VD models as our baseline models. Experimental results show that *VulBG* enables all the baseline models to detect more real-world vulnerabilities.

Paper organization. The remainder of the paper is organized as follows. Section II presents the motivation of our paper. Section III introduces our system. Section IV reports the experimental results. Section V discusses the future work. Section VI describes the related work. Section VII concludes the present paper.

II. Motivation

Vulnerabilities are flaws in code that may cause security problems. Though there are infinite codes and different types of vulnerabilities, there are implicit patterns among vulnerabilities, and that is the reason why VD could make detections. State-of-the-art DL-based VD approaches usually follow this procedure: select a method to represent the function, and then use NN to learn patterns of vulnerabilities. In real-world situations, functions tend to have complex logic and vulnerable semantics usually take up only a little part of the function. Given a function as input, there are a bunch of semantics for DL-based VD models to learn, and finding the vulnerable one is like finding a needle in the haystack. In addition, there are also connections between functions, such as code reuse and similar implementation. The same vulnerability may exist in different functions but VD may be confused by the differences of functions.

We take CVE-2018-17958 [25] and CVE-2018-17962 [26] as examples to show the problem. These two vulnerabilities lie in function $rtl8139\_do\_receive$ and $pcnet\_receive$, which implement the receiving functionality for different network cards in Qemu. For ease of expression, we simplify function's logic related to vulnerabilities and show the problem in Figure 1. The cause of the two vulnerabilities is both functions use an implicit type conversion to cast a $size\_t$ variable to int type which may result in an integer sign overflow, and then the variable is used in the following $memcpy$ without proper bounds check.

Though these two vulnerabilities share the exactly same logic, the scales of the functions introduce interference for VD model to filter out vulnerable semantics. The size of the two functions is 284 lines and 160 lines respectively, and the scales of slices of the two vulnerabilities are 8 lines and 10 lines. In other words, only less than 5% of the code of the functions contributes to the vulnerability and all other code is redundant. Therefore, extracting vulnerable semantics and removing unrelated code will be useful for VD.

A good way to extract vulnerable semantics is program slicing, but there are still problems in slice-based vulnerability detection. Firstly, information for generating precise slices is not always available. Slicing requires a point of interest as its entry. To generate precise slices, fine-grained information such as the line of the vulnerability, and variables related to the vulnerability is needed but hard to obtain. Limitations of static analysis tools also make it difficult to obtain precise slices.

```
ssize_t pcnet_receive(NetClientState *nc, const
uint8_t *buf, size_t size_){
    ...
    uint8_t buf1[60];
    int size = size_;
    ...
    if (size < MIN_BUF_SIZE) {
        memcpy(buf1, buf, size);
        memset(buf1 + size, 0, MIN_BUF_SIZE - size);
        size = MIN_BUF_SIZE;
    }
    ...
}
```

```
ssize_t rtl8139_do_receive(VLANClientState *nc, const
uint8_t *buf, size_t size_, int do_interrupt){
    ...
    int size = size_;
    uint8_t buf1[MIN_BUF_SIZE + VLAN_HLEN];
    ...
    if (size < MIN_BUF_SIZE + VLAN_HLEN) {
        memcpy(buf1, buf, size);
        memset(buf1 + size, 0, MIN_BUF_SIZE +
VLAN_HLEN - size);
    }
    ...
}
```

Fig. 1: Simplified code of CVE-2018-17958 and CVE-2018-17962

According to *Scvd* [27], the state-of-the-art AST-based static analysis tool *Joern* [10] fails in parsing some statements and cannot handle macros correctly. Compile-based static analysis tools have an unacceptable time overhead to generate slices for a dataset with thousands of functions. Secondly, though the NN model learns patterns from slices of vulnerabilities, it still needs to read in the whole function for detection, in which the vulnerable semantics are discrete. Therefore, we need another way to extract vulnerable semantics.

Besides, there are underlying connections between the two functions, since they have similar functionality, and the cause of the two vulnerabilities is the same. However, the similarity of the two functions is low. The ratio of function sizes is 1.8 to 1 and the cosine similarity between the two functions is 21% (functions are vectorized by *tf-idf* [28]). In this condition, similarity-based VD approaches may fail to identify these vulnerabilities. Therefore, there is a need for representing the connections between functions other than similarity.

To extract vulnerable semantics and address the connections between functions, we propose the Behavior Graph to assist DL-based VD. Instead of trying to fetch slices containing complete vulnerable semantics, we first attempt to split the function by semantics and extract abstract behaviors of the function. By slicing on potentially vulnerable operations, we can break down the function into parts. Each slice contains part of the function's semantics, so we consider it as one behavior

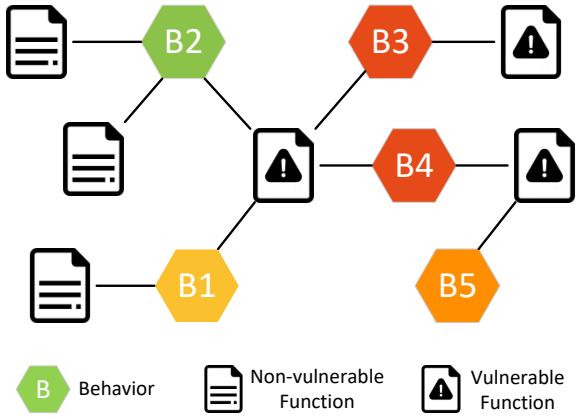


Fig. 2: Identify vulnerable semantics with Behavior Graph

of the function, and then the function could be represented by a set of behaviors. In this case, vulnerabilities could exist in a subset of the function's behaviors. Though we cannot yet determine which set contains vulnerability semantics, statements that are unlikely to contain vulnerabilities are filtered out.

To further identify vulnerability semantics and address connections between functions, we construct a graph based on the behaviors of functions. By calculating similarities of behaviors, we could figure out connections between behaviors, and by connecting behaviors with their corresponding functions, we could find out functions that share similar behaviors. In this way, connections between functions are established. Besides, for "innocent" behaviors that do not contribute to a vulnerability, there would be edges from non-vulnerable functions connecting to them to clarify their innocence. For behaviors that may contribute to a vulnerability, there would be more edges from vulnerable functions. We take Figure 2 as an example. The more connections to the vulnerable functions, the more likely the behavior contains vulnerable semantics (e.g, behavior 3 is more likely to contain vulnerable semantic than behavior 5). Specifically, behavior 1 could be considered "innocent" as many non-vulnerable functions also have this behavior. For behavior 3 and 4, they could be regarded as containing vulnerable semantics because there are vulnerable functions that share these behaviors. For behavior 5, it is a rare behavior since no other function possesses it. We cannot identify whether it contributes to vulnerabilities but this limitation can be eliminated by adding more functions to the Behavior Graph to reduce outliers. With the help of the Behavior Graph, we can automatically identify which set of behaviors may be vulnerable, and similar vulnerabilities can be addressed by the connections between functions.

In general, to help filter vulnerable semantics and address the connections between functions, we propose the concept of Behavior Graph, and based on the Behavior Graph we design *VulBG* to enhance the ability of vulnerability detection.

III. System Architecture

In this section, we introduce how to construct the Behavior Graph and use it to design *VulBG* for enhancing DL-based VD.

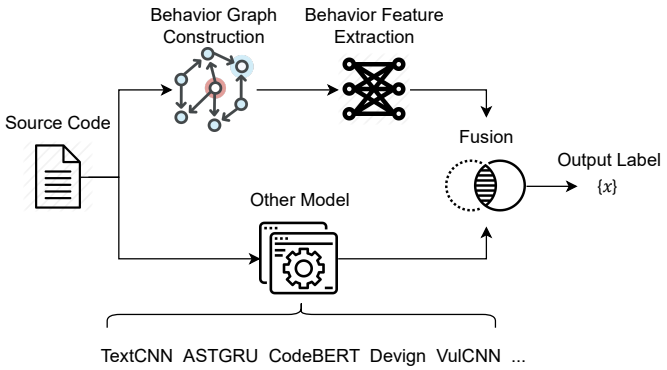


Fig. 3: System overview of *VulBG*

A. Overview

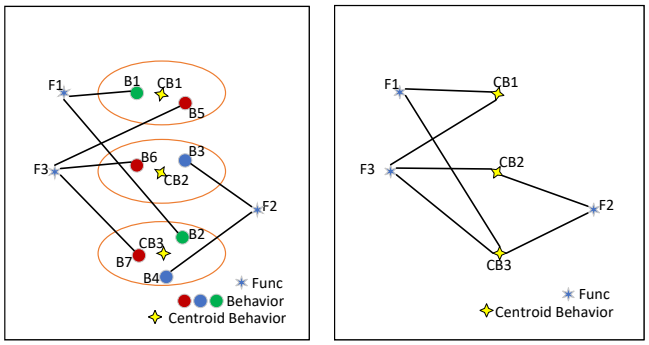
As shown in Figure 3, *VulBG* consists of three phases: *Behavior Graph Construction*, *Behavior Feature Extraction*, and *Model Fusion*.

- *Behavior Graph Construction:* Given the source code of functions, we first use slicing and code embedding to obtain behaviors of functions. By clustering the behaviors, we get a set of centroid behaviors and then construct the Behavior Graph based on centroid behaviors and similarities of behaviors.
- *Behavior Feature Extraction:* To utilize the structure information of the Behavior Graph, we apply graph embedding on each function node to transform it into a vector, and then use a *multilayer perceptron* (MLP) to further process the Behavior Feature of functions.
- *Model Fusion:* In order to use the Behavior Feature to enhance existing DL-based VD models (*e.g., TextCNN*, *ASTGRU*, *CodeBERT*, *Devign*, and *VulCNN*), we use model fusion to combine Behavior Feature with the features extracted by the above models to make classifications together.

B. Behavior Graph Construction

VulBG aims to utilize the connections between functions to achieve higher accuracy in vulnerability detection. We propose the concept of the Behavior Graph to represent the Behavior Features of the functions. Specifically, Behavior Graph consists of function nodes and behavior nodes. Function nodes are functions from training data, and behavior nodes are cluster centers of slices. For each slice owned by one function, there is an edge connecting its corresponding cluster center (behavior node) to the function node.

Figure 4 shows a simple example of a Behavior Graph that consists of three functions and three centroid behaviors. There are two types of nodes in the Behavior Graph: centroid behavior nodes (yellow stars) and function nodes (blue stars). We take Function 1 (F1) as an example to introduce the composition of the Behavior Graph. In Figure 4 (a), F1 has two behaviors (B1, B2) which respectively belong to centroid behavior 1 (CB1) and centroid behavior 3 (CB3), so there are two edges in the Behavior Graph, one connects F1 and CB1, and the other connects F1 and CB3. F3 in the example also has edges to CB1 and CB3, thus the connections between F1 and F3 are established through these two shared centroid behaviors. Besides, the weights of edges are distances between behaviors and their corresponding centroid behaviors.



(a) Behaviors (b) Behavior Graph

Fig. 4: A case of Behavior Graph

Figure 5 and Algorithm 1 describe the detailed procedures on how to construct the Behavior Graph. It mainly consists of four phases: *Code Slicing*, *Embedding*, *Clustering*, and *Graph Construction*. To construct the graph, we first extract behaviors of a large number of functions and then cluster them to obtain a set of centroid behaviors to represent all behaviors. By representing functions with behaviors they possess, we can connect functions to centroid behaviors, and then the Behavior Graph is built and connections between functions are clear.

Algorithm 1 Constructing a Behavior Graph from the source code of the functions

Input: F: Functions' source code.
Output: G: A Behavior Graph.

```
Slices ← CodeSlicing(F)
Behaviors ← CodeEmbedding(Slices)
CentroidBehaviors ← Clustering(Behaviors)
G ← EmptyBehaiviorGrpah()
for each f ∈ F do
    behaviors ← GetFunctionBehaviors(Behaviors, f)
    for each b ∈ behaviors do
        c ← GetCentroidBehavior(CentroidBehaviors, b)
        d ← SimilarityCalculation(b, c)
        AddEdge(G, f, c, d)
    end for
end for
return G
```

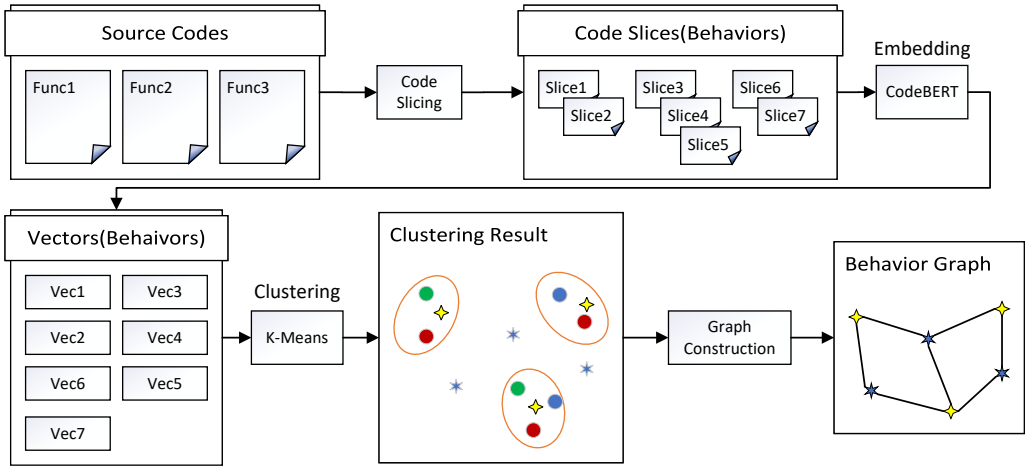


Fig. 5: An example to illustrate how to convert the source code of three functions into one Behavior Graph

1) Code Slicing

To figure out functions' behaviors, we utilize program slicing to break down functions into parts. Program slicing is a widely used static analysis technique that only extracts statements relevant to some interests, thus eliminating the interference of statements we do not care about. Since vulnerable semantics often exist in specific operations (*e.g.,* memory accessing and dangerous API call), slicing on these operations can weed out code unlikely to be vulnerable and keep VD focused on potentially vulnerable code. Therefore, we believe that in the context of vulnerability detection, slicing is a good way to describe behaviors of functions.

SySeVR [8] proposes a set of vulnerability syntax characteristics that includes four kinds of operations (*i.e.,* API call, pointer operation, array operation, and arithmetic operation) that are more prone to bugs. Based on these types of operations, we develop the following slicing rules:

- *API call:* function calls to specific libraries' APIs, including memory allocator-related APIs, string operation, and memory operation APIs. Misuses of these APIs could result in different kinds of vulnerabilities such as buffer overflow, use-after-free, and information leakage. We conduct backward slicing on all parameters of function call and forward slicing on its return value.
- *Memory operation:* operations to pointer type variables and array-like variables, including Pointer operation and Array operation of *SySeVR*'s syntax characteristics. This kind of operation is the main cause of memory corruptions like null pointer dereference and out-of-bound access. For this kind of operation, we conduct backward slicing on the pointer-like variable. If the memory accessing is indexed by a variable, we also make backward slicing on this variable.

For arithmetic operations, we do not explicitly slice operations of this type. For vulnerability detection, checking arithmetic operations is to find out operations that may have integer overflow or division by zero. Integer overflow not always results in a vulnerability, but it will lead to out-of-bound access when the overflowed variable is used in some memory accesses, and logical errors may occur when overflowed variable is used in sensitive APIs. Since we have sliced the memory operations and API calls, most of the buggy arithmetic operations are already contained in the slice, so there is no need to slice arithmetic operations separately. Besides, almost every line of code in a program contains arithmetic operations, so slicing arithmetic operations will introduce noise that is irrelevant to vulnerabilities.

We implement program slicing based on *Joern* [29]. The dataflow dependency and control flow dependency required for slicing are obtained through the PDG and CFG provided by *Joern*, and the variables involved in the operations in the rules are obtained through *Joern*'s query result. For each variable to be sliced, we traverse the CFG forwards or backwards to collect statements and variables according to the dataflow dependency, and variables involved in the dataflow dependency will also be sliced later. Branch statements like 'if', 'for', and 'while' are included if they are post-dominators/dominators of sliced code for forward/backward slicing. Variables involved in these branch statements will not be sliced if they have no dataflow dependency on sliced variables.

We represent slices in text form instead of using subgraphs of PDG like *SySeVR* [8]. Experiments show that 95.8% of slices have less than 64 words, so there is no need to represent a slice with a graph because of the high complexity of graph.

2) Embedding

After code slicing, we can obtain the functions' behaviors. Since the behaviors of the function are code slices, they are in text forms. We can use the pattern matching algorithms (*e.g.,* K.M.P [30]) to describe the similarity, but it is difficult to implement it in a clustering algorithm and it ignores the syntax and semantics features in code slices. To make subsequent processing more flexible and convenient, we choose to use a text-embedding algorithm to transform the behaviors into vectors. In our paper, we use *CodeBERT* [23] to embed the behaviours. Firstly, *CodeBERT* is constructed on a bidirectional trans-

former that can capture long-distance dependencies of code sequences. It can maintain the relationship between contexts, collect potentially vulnerable code patterns, and minimize information loss. Secondly, *CodeBERT* inherits the structure of multi-head attention, which makes the model focus on multiple key points of a code sequence. Therefore, *CodeBERT* performs better than other embedding algorithms (*e.g., Word2Vec* [31]) when processing the loop condition. Besides, [32] shows that *CodeBERT* does well in code classification even without fine-tuning, so *CodeBERT* is a good fit for our scenario. *CodeBERT* embeds each word into a vector with 768 dimensions, which is too heavyweight for subsequent processing. We use the pooler result to transform each behavior (code slice) into a vector with 768 dimensions to reduce embedding size.

3) Clustering

After embedding the behaviors into vectors, we are able to construct a Behavior Graph. However, the graph can be huge and the information it carries is coarse, making it difficult to store and process. To reduce the number of nodes and make the information more focused, we use clustering to extract centroid behaviors. In our paper, we use *MiniBatchKMeans* [33] to cluster the vectors. *MiniBatchKMeans* is an optimization variant of *K-Means* [34] for clustering a large amount of data, which is an unsupervised clustering algorithm that attempts to partition samples to K clusters by minimum within-cluster variances.

4) Graph Construction

The Behavior Graph is obtained by connecting the functions to its behaviors' corresponding centroid behaviors. Since there are differences between behaviors, we set the weights of the edges to the similarity of behaviors to preserve this information. The more similar the two behaviors are, the closer their embedding results are. We can use the distance between a behavior and its corresponding centroid behavior to express their similarity. In our paper, we use Euclidean Distance [35] to describe the similarity of behaviors and centroid behaviors.

C. Behavior Feature Extraction

To efficiently take the advantage of function's behavior information and the connection information between functions in the Behavior Graph, we adopt graph embedding to transform nodes into vectors. Using graph embeddings to represent Behavior Graph has several benefits:

- It is simpler and faster to calculate on vectors than directly operating on the graph.
- The information of nodes and edges in the graph can only be represented and calculated through mathematics and statistics. After embedding, the information of the graph can be processed more flexibly in the vector space.

Specially, we leverage a widely used tool (*e.g., Node2Vec* [36]) for embedding. *Node2Vec* is a graph embedding method that uses a biased random walk procedure that integrates *depth-first-search* (DFS) and *breadth-first-search* (BFS) to explore neighborhoods. The BFS part can accurately obtain a microscopic view of the network by exploring around the source node, and the DFS part can move further to obtain a macro-view of the network. *Node2Vec* treats results of random walks as words and utilizes *Word2Vec* to do the embedding. It is a simple but efficient unsupervised objective to train distributed representations of graphs. In our paper, we use *Node2Vec* to transform a function node in the Behavior Graph into a vector whose dimension is 128 for further processing.

To further process the graph vector, we construct a four-layer MLP classifier with graph vector as input, and use the output of its last hidden layer as Behavior Feature for fusion step. This Behavior Graph Model consists of four linear layers, using ReLU as the activation function with a dropout rate of 0.5 and a learning rate of 0.0001. This model will also be used to evaluate the vulnerability detection capability of Behavior Graph.

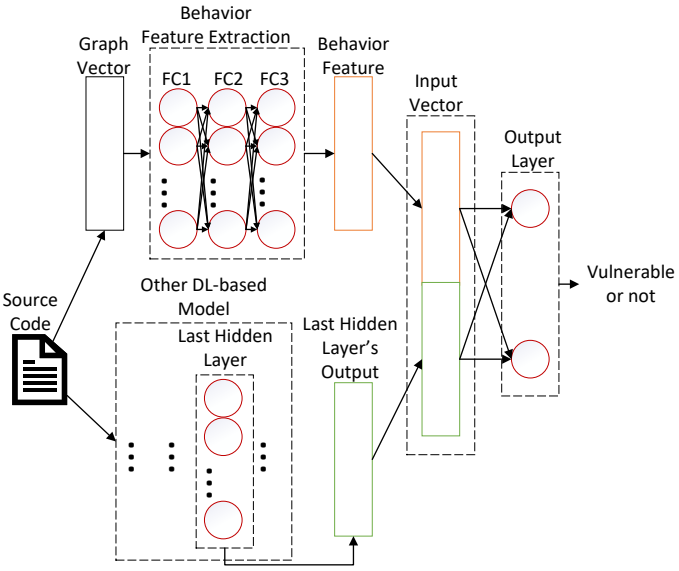


Fig. 6: Model fusion of *VulBG*

D. Predict Behavior Feature For New Samples

To put the Behavior Graph Model into practice, the model should be able to predict Behavior Feature for new functions. As Figure 7 shows, the overall predict process is the same as that of training as described in Figure 5, except that a modified *Node2Vec* is used in the graph embedding step to obtain vectors for new nodes.

We modify *Node2Vec* to enable it to represent new nodes when making predictions. For each new function node that needs to be embedded, we temporarily add it to the graph and apply a random walk with the new node as the entry. Then we pass the result of the walk to *Word2Vec* and apply its online updating ability to learn the representation of the new node. After that, the new node is removed from the graph. Compared with the large number of old nodes in the graph, for each prediction only one new node needs to be added and its impact on the graph structure is little. Moreover, the random walk could be parallelized and the online updating of

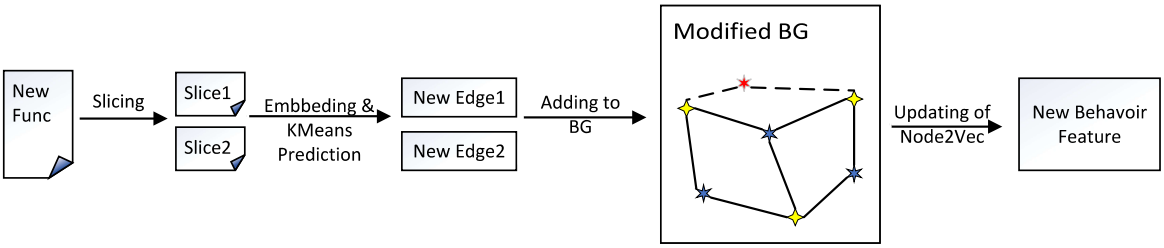


Fig. 7: The process of predicting Behavior Feature of new samples by Behavior Graph Model

Word2Vec could be batched, which will not be time-consuming when making large amounts of predictions.

E. Fusion

Since existing DL-based VDs ignore the connection between functions, we aim to apply Behavior Features to them to improve vulnerability detection ability. For model fusion, we concatenate the output of the Behavior Graph Model's last hidden layer with the output of the existing DL-based VD's last hidden layer, and finally use a linear layer for output. Figure 6 shows the process of model fusion. Although such a fusion method is simple, it modifies other models as little as possible, making *VulBG* highly scalable.

To prove the effect of *VulBG*, we choose five state-of-the-art DL-based VDs in four classes as baselines: text-based *TextCNN* [21], AST-based *ASTGRU* [22], pretrained-based *CodeBERT* [23], graph-based *Devign* [15], and *VulCNN* [24].

For *TextCNN*, *CodeBERT*, and *ASTGRU*, we re-implement the baseline models and report the best performance in our paper. All re-implemented baselines use 0.0001 as learning rate, 32 as batch size, Cross Entropy Loss as loss function, and Adam as optimizer. For *Devign* and *VulCNN*, we directly use their open-source code to commence our experiments.

1) TextCNN

TextCNN is a simple *convolutional neural network* (CNN) that utilizes the 1d-convolution layer to extract features from embeddings of the source code. This model treats vulnerability detection as a pure NLP classification task, so its structure is simple and can be trained fast. In our paper's implementation of *TextCNN*, *Word2Vec* is used for text embedding. For the 1d-convolution layer, we select three filter sizes (3, 4, 5), and each size has 100 filters to extract features of different parts of the source code. Then we use four linear layers to process the features to output the probability of vulnerability. The *TextCNN* baseline uses tanh as activation function and a dropout rate of 0.3.

2) ASTGRU

ASTGRU [22] uses AST as the model's input rather than source code. It is also a tree-based model that extracts AST from the source code and traverses the tree to get an input sequence. Then it uses *Word2Vec* [31] to embed the sequence and uses the *bidirectional gate recurrent unit* (Bi-GRU) to learn features of code. The *ASTGRU* baseline consists of two Bi-GRU layers and one linear layer, with ReLU as activation function and a dropout rate of 0.2.

3) CodeBERT

As described in Section III-B1, *CodeBERT* learns representations of programming languages and supports different downstream tasks. Here we adopt the method of using the *BERT* [37] model for sequence classification and add linear layers after *CodeBERT*'s pooled result to make classification. The pre-trained model we use is *codebert-base* [38] and it is not fine-tuned during training due to its huge memory and time consumption. The *CodeBERT* baseline uses two linear layers to further process the embedding of code, and uses ReLU as activation function and 0.3 as dropout rate.

4) Devign

Devign is a graph-based VD, which includes three sequential components: 1) *Graph Embedding Layer*, which encodes source code of a function into a joint graph structure with comprehensive program semantics; 2) *Gated Graph Recurrent Layer*, which leverages a *Gated Recurrent Unit* (GRU) layer to learn features of nodes in the graph through aggregating and passing information on neighboring nodes; 3) *Conv Module*, which has a convolutional layer and a softmax layer to extract node representation for graph-level prediction.

5) VulCNN

VulCNN uses *Joern* [29] and *Sent2Vec* [39] to generate PDGs from source code. It computes degree centrality, katz centrality, and closeness centrality on PDG. Based on these three types of centrality, *VulCNN* generates three vectors and treats them as the three channels of the image. Then it uses a CNN to complete the classification task.

IV. Experimental Evaluation

In this section, we aim to answer the following research questions:

- *RQ1: What is the performance of Behavior Graph on vulnerability detection?*
- *RQ2: How effective is VulBG in improving state-of-the-art models' performance on vulnerability detection?*
- *RQ3: What is the performance of VulBG with different code embedding methods and graph embedding techniques on vulnerability detection?*

A. Experiment Settings

1) Dataset

We use FFMpeg+Qemu and Chrome+Debian datasets to evaluate our work. Statistics of datasets are shown in Table I. The FFMpeg+Qemu dataset is a balanced dataset that has

been used in many previous studies [15], [20], [40]. It consists of two popular real-world software written in C language, providing +25K functions with 45.56% of those vulnerable. The Chrome+Debian dataset is an unbalanced dataset proposed by *ReVeal* [20]. It consists of code from Chromium and Debian source code repository with +21K functions and 9.79% of those are vulnerable.

For each dataset, we randomly split it into an 80% training set, a 10% validation set, and a 10% testing set. We use the same dataset split for all following experiments. Due to the severe imbalance in Chrome+Debian dataset, we use oversampling to rebalance the training set of this dataset.

TABLE I: Dataset statistics

Dataset	Samples	Vul Rate	Avg Size
FFMpeg+Qemu	25,524	45.56%	55 lines
Chrome+Debian	21,059	9.79%	32 lines

2) Performance Metrics

We consider Precision (P), Recall (R), and F-measure (F1) as mainstream performance metrics. Formulas of metrics are listed below, with *true positive* (TP) being the number of samples correctly detected as vulnerable, *false positive* (FP) being the number of non-vulnerable samples incorrectly detected as vulnerable, *true negative* (TN) being the number of samples correctly detected as non-vulnerable, and *false negative* (FN) being the number of vulnerable samples incorrectly detected as non-vulnerable.

$$P = \frac{TP}{TP + FP}$$

$$R = \frac{TP}{TP + FN}$$

$$F1 = 2 * \frac{P * R}{P + R}$$

3) Environment

All experiments are run on a server with 32 cores of CPU and an RTX 5000 GPU. For intermediate phases, we implement slicing phase based on *Joern* [29], clustering based on *Scikit-learn* [41], graph embedding based on modified *Node2Vec* [36], word embedding based on *Word2Vec* [31] and *CodeBERT* [23]. We build and train models based on *PyTorch* [42].

B. Parameter Selection for Clustering

During the construction of Behavior Graph, *MiniBatchKMeans* is used for clustering, and its performance largely depends on the input hyperparameter K, which defines the number of classes. We apply elbow method [43] to figure out the most suitable K. Elbow method is a heuristic commonly used to determine the number of clusters by finding the "elbow" of the variation curve, and in this paper we use distortion as the variation metric.

Figure 8 shows the curve of the distortion of *MiniBatchKMeans* as a function of K, and the "elbows" appear where K is 1140 and 1150 for FFMpeg+Qemu dataset and Chrome+Debian dataset, respectively. To keep generality, we select 1140 as the optimal value of K for the following experiments.

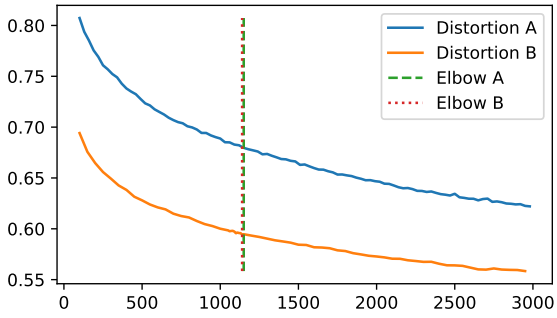


Fig. 8: Distortion curve of MiniBatchKMeans on FFMpeg+Qemu dataset (A) and Chrome+Debian (B) datasets

C. Performance of Behavior Graph

In this section, we aim to answer *RQ1: What is the performance of the Behavior Graph on vulnerability detection?*

To prove the performance of Behavior Graph, we first evaluate the Behavior Graph Model described in Section III-B4 which takes the Behavior Feature as input. We also compare our Behavior Graph Model to five state-of-the-art approaches of text-based, AST-based, pretrained-based, and graph-based models. Experimental results are shown in Table II.

TABLE II: Performance of Behavior Graph Model vs. baseline models

Technique	FFMpeg+Qemu			Chrome+Debian		
	F1	P	R	F1	P	R
Behavior Graph	56.1	51.8	61.2	36.5	26.4	59.3
TextCNN	53.7	57.9	50.0	40.2	30.2	60.6
ASTGRU	51.8	53.8	49.9	22.6	33.5	15.4
CodeBERT	53.2	56.1	50.6	24.7	14.6	80.2
Devign	55.3	53.4	57.2	29.2	26.4	32.8
VulCNN	54.9	51.8	58.4	31.5	22.8	51.0

Overall, our Behavior Graph Model achieves high F1 (56.1%) and Recall (61.2%). In terms of F1 and Recall, among the six approaches, it ranks first on the FFMpeg+Qemu dataset and second on the Chrome+Debian dataset, which could already prove that Behavior Graph works well in vulnerability detection.

Except for our Behavior Graph Model, graph-based models generally perform better than other approaches because graph-based models take account of syntax, data-flow, and

control-flow characteristics of the program. The AST-based approach *ASTGRU* processes code's syntax and uses trees as model input, but experimental results show that it is not as effective as the text-based model *TextCNN*. To our surprise, *TextCNN* performs particularly well on the unbalanced dataset, achieving an F1 of 40.2%. Compared with the Behavior Graph Model, *TextCNN* has a 3.8% higher Precision and a similar Recall, which leads to its overall performance advantage in F1.

Since we use *CodeBERT* to embed slices during the construction of Behavior Graph, we also compare the *CodeBERT* classifier with Behavior Graph Model to demonstrate the effectiveness of *CodeBERT*. As shown in Table II, the F1 of *CodeBERT* classifier is 53.2%, which is similar to *TextCNN*. The performance of the *CodeBERT* classifier is not outstanding, we believe the main reason for its limited performance is *CodeBERT*'s inability to process large functions: *CodeBERT* can only handle code snippets up to 512 words in length, and for larger functions, vulnerable semantics may be lost due to truncation, so it is inappropriate to use *CodeBERT* directly for vulnerability detection on entire functions.

We observe that Behavior Graph Model gets high Recall (61.2% and 59.3%) but its Precision is not high (51.8% and 26.4%), which means that it performs well at finding out vulnerabilities but mistakenly judges some non-vulnerable functions as vulnerable. The reasons for this result are as follows:

- The relationship between behaviors addressed by Behavior Graph can indicate similarly vulnerable semantics, resulting in the MLP being good at detecting vulnerable functions and the Recall being high.
- Some non-vulnerable functions share similar behaviors with vulnerable ones, resulting in false positives, so the Precision is not so good.

In one word, the Behavior Graph Model performs well on both balanced and unbalanced datasets, proving that Behavior Graph does carry useful information for vulnerability detection.

D. Improvements of Behavior Graph

In this section, we aim to answer *RQ2: How effective is VulBG in improving state-of-the-art models' performance on vulnerability detection?*

To answer the question, we apply *VulBG* to five state-of-the-art approaches according to Section III-D, and then compare these models' performance respectively with and without Behavior Graphs.

Experimental results are shown in Table III, and changes in each metric are listed under scores. According to Table III, all fusion model has a higher F1 and a higher Recall. On FFMpeg+Qemu dataset, aside from Precision of *BG+TextCNN*, all metrics of each model reach higher scores, and on Chrome+Debian dataset, except for the drop of Recall of *BG+CodeBERT*, all metrics of each model also improve.

For *BG+TextCNN*, on the FFMpeg+Qemu dataset, its F1 improves by 6% and its Recall has a significant improvement

TABLE III: Performance of fusion models

Technique	FFMpeg+QEMU			Chrome+Debian		
	F1	P	R	F1	P	R
TextCNN	59.7 +6.0	54.8 -3.1	65.7 +15.7	44.4 +4.2	33.7 +3.5	65.2 +4.6
ASTGRU	54.4 +2.6	53.9 +0.1	54.9 +5.0	30.3 +7.7	39.8 +6.3	24.5 +9.1
CodeBERT	58.5 +5.3	56.2 +0.1	60.9 +10.3	32.5 +7.8	22.7 +8.1	57.2 -23.0
Devign	60.2 +4.9	55.9 +2.5	65.2 +8.0	37.1 +7.9	30.7 +4.3	47.0 +14.2
VulCNN	57.2 +2.3	52.5 +0.7	62.7 +4.3	36.3 +4.8	27.2 +4.4	54.6 +3.6

of 15.7%. Although there is a drop in Precision by 3.1%, *BG+TextCNN* fusion model achieves better results overall. The reason for the drop in Precision is that Behavior Graph's Precision is not high while the *TextCNN* baseline reaches the highest Precision (57.9%) among all the baselines, so the fusion of models may result in Precision being averaged. On the Chrome+Debian dataset, all metrics of *BG+TextCNN* have improvements ranging from 3.5% to 4.6%, and due to the high performance of *TextCNN* baseline, the fusion model ranks first in F1 for 44.4%. For *BG+ASTGRU*, the fusion model improves all the metrics on the two datasets, and the improvement is mainly reflected in Recall for 5.0% and 9.1%.

For *BG+CodeBERT*, the fusion model has a notable improvement in higher F1 for 5.3% and 7.8% on the two datasets. On Chrome+Debian dataset, there is a huge decrease in Recall for 23%. The Recall of *CodeBERT* baseline is 80.2% while its Precision is only 14.6%, which means the *CodeBERT* model judges almost all samples vulnerable. In this case, the Recall of 80.2% is abnormal and the fusion model's decrease in Recall is reasonable. For graph-based models, the fusion models also have higher performance on all metrics on both datasets. *BG+Devign*'s F1, Precision, and Recall on the FFMpeg+Qemu dataset is 60.2%, 55.9%, and 65.2%, respectively, which is the best overall performance on the balanced dataset. On the Chrome+Debian dataset, *BG+Devign* also gets a significant improvement in Recall for 14.2%, and its Precision also increases by 4.3%, which results in a great improvement in F1 for 7.9%.

In general, *VulBG* has a significant effect on improving the performance of DL-based VDs. On average, F1, Precision, and Recall are improved by 4.2%, 0.1%, and 8.7% on the FFMpeg+Debian dataset, and 6.5%, 5.3%, 1.7% on the Chrome+Debian dataset. The high Recall of *VulBG* enables DL-based VDs to find more vulnerabilities, and the higher F1 also proves that *VulBG* could improve the overall performance of different models.

E. Ablation Study

In this section, we aim answer *RQ3*, to study the effectiveness of the code embedding and graph embedding techniques that are required in *VulBG*, and to figure out the most suitable

components of *VulBG*. *VulBG* requires code embedding to transform slices to vectors for better similarity calculation, and node embedding to extract Behavior Feature from the Behavior Graph. We select different code embedding and graph embedding methods and retrain the model described in Section III-B4 on the FFMpeg+Qemu dataset.

1) Code Embedding

Word2Vec [31], *CodeBERT* [23], and *Sent2Vec* [44] are evaluated for the code embedding step. *Sent2Vec* is a widely used sentence embedding method which adopts a simple but efficient unsupervised objective to train distributed representations of sentence. *Word2Vec* is a classic unsupervised word embedding model based on *continuous bag-of-words* (CBOW) Model or Skip Gram Model.

The default output dimension of embedding is used for all three methods and the average pooled result is used to represent the slice. We retrain and tune the models respectively, and as the experimental results shown in Table IV, BG using *CodeBERT* for code embedding performs better than others. Thus, *CodeBERT* is adopted as the code embedding method used in *VulBG*.

TABLE IV: Performance of BG using different techniques

Phase	Technique	F1	P	R
Code Embedding	Word2Vec	54.7	52.0	56.7
	CodeBERT	56.1	51.8	61.2
	Sent2Vec	54.0	50.7	57.3
Graph Embedding	Node2Vec	56.1	51.8	61.2
	ProNE	53.8	50.1	58.2

2) Graph Embedding

We select *Node2Vec* [36] and *ProNE* [45], two widely used graph embedding techniques, to study their performance on Behavior Graph. *Node2Vec* has been mentioned in Section III-B4, and *ProNE* is a fast and scalable network embedding approach that formulates network embedding as sparse matrix factorization. The same output dimension is set for both of the methods, and *CodeBERT* is used for the code slice embedding step. It should be noted that the implementation of *ProNE* does not support weighted edges, so the weight on the edge of the Behavior Graph is removed when using *ProNE*. After retraining and tuning the models respectively, Table IV shows that *Node2Vec* outperforms *ProNE* in the Behavior Graph model. Therefore, we make use of *Node2Vec* in *VulBG*.

V. Discussion

A. Threats to Validity

Datasets. Each sample of the two real-world datasets we use contains only one function, without structure definitions and macro definitions. During our experiments, we find that *Joern* may generate error PDGs with only one node. This happens when the function definition is wrapped in a macro, resulting in the return type and parameter type missing or "##" in the function name. In this case, *Joern* fails to parse the function and cannot do further slicing. To keep as many samples as possible, we try to replace these function names with a fixed string and remove all its parameters. This solution will cause the slicing to fail to continue when it encounters function parameters. There are also malformed functions in datasets, such as functions with mismatching brackets and not closed "$\#ifdef$" macros. We remove functions that cannot be processed by *Joern* and conduct experiments on the rest of the datasets. The origin size of the dataset is 27,318 for FFMpeg+Qemu dataset and 22,734 for Chrome+Devign dataset, and after sanitizing the size is 25,524 and 21,059 as described in section IV-A1.

B. Limitations and Potential Solutions

PDG Generation. The program slicing phase in our work is based on PDGs generated by *Joern*. We find a missing edge issue in PDGs, which would lead to inaccurate slice results. To generate a complete PDG, pointer analysis, and other dataflow analyses are necessary, but these advanced static analysis techniques work on *intermediate representation* (IR), which is generated during compilation. Since code in datasets is not compilable, these techniques are unavailable. Other DL-based approaches based on PDGs also face this problem but there is no solution yet. The best solution is to construct a compilable dataset so that the following research could use IR-based analysis techniques. Moreover, IR is more suitable for analysis because it has a neat form with simplified semantic information, while AST focuses mostly on syntax. *LLVM* [46] provides precise IR-based PDG generation and it will be helpful if datasets meet its requirements.

Behavior Graph. We use slices of potentially vulnerable operations to represent the function's behavior, but there are also other approaches to break down the function, such as using paths of the function. Besides, we set the weights of edges in the Behavior Graph to the similarity between behaviors and use Euclidean Distance as the similarity metric, which is a fairly simple method. By changing weights' measurements, the Behavior Graph could address other information.

C. The Benefits of Behavior Graph

The benefits of Behavior Graph are mainly two-fold. First, functions with similar slices can be easily grouped. Second, the importance of a slice can be obtained by the number of edges connected to the corresponding cluster centers in the Behavior Graph. During training, it is possible to distinguish classes of slices that may lead to vulnerabilities.

D. Interpretability

As a function-level model, *VulBG* outputs whether there are vulnerabilities in the function. Line-level approaches have also been proposed recently, which first use a detection model to detect vulnerabilities, and then use an additional interpretation model to locate the vulnerability, such as *IVDetect* [47], *LineVul* [48], and *LineVD* [49]. *VulBG* can also be applied to the detection model of these interpretable approaches to detect

more vulnerabilities, and then use the interpretation model to obtain explainable results.

We will do research on the interpretability of the Behavior Graph in future work, and we hope to locate the vulnerability in slice-level by investigating Behavior Feature of functions, which will be more helpful to practitioners in finding bugs. In Behavior Graph there are two indicators that can be used to interpret the output:

- The weights of the edges in Behavior Graph represent the similarity of the behavior of the function to the centroid behavior. When a set of behavior has a high correlation with a vulnerability, the similarity to centroid behavior could indicate the potential vulnerable slices.
- After node embedding, we can calculate the distance from new function nodes to known vulnerable function nodes in Behavior Graph, which helps to point out what pattern the bugs follow in these new functions.

E. Extending to Other Dataset

Generally, *VulBG* is not designed for a specific programming language or source code dataset. It can be easily extended to any other C/C++ dataset (*e.g.,* NVD datasets [50]) with minor modification. For the other languages, users may need to adapt the slicing rules to language characteristics and change to a slicing tool that supports the programming language.

However, it is always tough to find a large number of vulnerabilities in real-world scenarios. So here comes a valuable topic how to keep high detection performance with only a few training data. In future work, we will try to apply few-shot learning on *VulBG* to make it more effective for real-world scenarios.

VI. Related Work

Currently, available vulnerability detection systems can be divided into the following three categories: static vulnerability detection, similarity-based vulnerability detection, and machine-learning-based vulnerability detection.

As for static vulnerability detection, many traditional program analysis tools or vulnerability detection systems were developed based on this way (*e.g., Checkmarx* [51], *RATS* [52], and *FlawFinder* [53]). These works firstly need the human experts to determine the detecting rules, then analyze programs based on conventional static analysis theories (*e.g.,* , data-flow, abstract interpretation, and taint analysis). These works have been widely used and proved that they can find vulnerabilities in all kinds of the software system. However, these works have the following shortage: they heavily rely on the detecting rules, therefore, they can not detect the vulnerabilities not covered in the detecting rules.

To get rid of detecting rules, similarity-based vulnerability detection emerged. This type of work detects vulnerabilities by calculating the similarity between the samples to be tested and samples with vulnerabilities. When the similarity exceeds a threshold, the sample is identified as vulnerable. Since the source code of function can be treated as a piece of text [54], [55], a sequences of tokens [56], [57], a tree [58], [59], or even a graph [60], many various aspects [61] can measure the similarity of functions. These works may not need detecting rules anymore, but still need human experts to determine the vulnerable samples. Therefore, they can only find the cloned vulnerabilities but can not find the new vulnerabilities [4].

The machine-learning-based vulnerability detection approaches [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18] have been proved that it performs much better in detecting a new pattern of vulnerabilities. These works can be divided into two subcategories. 1) These works treat the source code as a piece of text, turn the vulnerability detection into a text classification problem, and use NLP solutions to solve the problem [4], [5], [6], [7], [8], [9], [16], [17], [18]. 2) These works build PDGs from the source code's AST through static analysis and then transform the vulnerability detection mission into a graph or node classification mission, and use a GNN to achieve the goal [10], [11], [12], [13], [14], [15]. In general, *VulBG* belongs to the second category of solutions. The difference with other schemes is that *VulBG* puts all samples into one graph, instead of converting each sample into one separate graph. By doing so, *VulBG* can find connections between samples. To summarize, *VulBG* proposes a novel concept of Behavior Graph to describe the connection between functions and enhance the other DL-based VD approaches by building the Behavior Graph Model.

VII. Conclusion

In this paper, we propose a novel approach that can extract functions' behaviors and then construct a Behavior Graph to represent the connections of different functions. We design and implement *VulBG*, a framework for higher performance in vulnerability detection by combining the Behavior Graph with other DL-based VD approaches. The evaluation results on two real-world dataset report that Behavior Graph itself is good enough for vulnerability detection, and *VulBG* can further effectively improve the overall performance of different kinds of DL-based VD approaches(*i.e., TextCNN*, *ASTGRU*, *CodeBERT*, *Devign*, and *VulCNN*).

VIII. Data Availability

The data sets used in our evaluations can be publicly accessed at *https://github.com/CGCL-codes/VulBG*. The source code of our tool has also been published on the above website.

Acknowledgement

Thanks to the anonymous reviewers for their insightful comments. This work was supported by the National Natural Science Foundation of China (No. 62172168), the National Key R&D Plan of China (No. 2022YFB3103403), the Hubei Province Key R&D Technology Special Innovation Project (No. 2021BAA032), the Wuhan Applied Foundational Frontier Project (No. 2020010601012188), and the Guangdong Provincial Key R&D Plan Project (No. 2019B010139001).

REFERENCES

[1] "Wannacry ransomware attack," https://en.wikipedia.org/wiki/WannaCry_ransomware_attack, 2020, accessed: 2022-08.

[2] "The exactis breach: 5 things you need to know," https://blog.infoarmor.com/individuals-and-families/the-exactis-breach-5-things-you-need-to-know, 2020, accessed: 2022-08.

[3] "5 key takeaways from the 2020 open source security and risk analysis report," https://securityboulevard.com/2020/05/5-key-takeaways-from-the-2020-open-source-security-and-risk-analysis-report, 2020, accessed: 2022-08.

[4] Z. Li, D. Zou, S. Xu, X. Ou, and Y. Zhong, "VulDeePecker: A deep learning-based system for vulnerability detection," in *Proceedings of the 25th Network and Distributed System Security Symposium (NDSS'18)*, 2018, pp. 1–15.

[5] 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*, vol. 18, no. 5, pp. 2224–2236, 2019.

[6] 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 24th ACM SIGSAC Conference on Computer and Communications Security (CCS'17)*, 2017, pp. 2539–2541.

[7] X. Duan, J. Wu, S. Ji, Z. Rui, T. Luo, M. Yang, and Y. Wu, "VulSniper: Focus your attention to shoot fine-grained vulnerabilities," in *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI'19)*, 2019, pp. 4665–4671.

[8] 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*, vol. 19, no. 4, pp. 2244–2258, 2021.

[9] 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 *Proceedings of the 17th IEEE International Conference on Machine Learning and Applications (ICMLA'18)*, 2018, pp. 757–762.

[10] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P'14)*, 2014, pp. 590–604.

[11] L. Cui, Z. Hao, Y. Jiao, H. Fei, and X. Yun, "Vuldetector: Detecting vulnerabilities using weighted feature graph comparison," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 2004–2017, 2020.

[12] S. Cao, X. Sun, L. Bo, Y. Wei, and B. Li, "Bgnn4vd: Constructing bidirectional graph neural-network for vulnerability detection," *Information and Software Technology*, vol. 136, p. 106576, 2021.

[13] X. Cheng, H. Wang, J. Hua, G. Xu, and Y. Sui, "Deepwukong: Statically detecting software vulnerabilities using deep graph neural network," *ACM Transactions on Software Engineering and Methodology*, vol. 30, no. 3, pp. 1–33, 2021.

[14] G. Lin, W. Xiao, L. Y. Zhang, S. Gao, Y. Tai, and J. Zhang, "Deep neural-based vulnerability discovery demystified: data, model and performance," *Neural Computing and Applications*, vol. 33, no. 20, pp. 13 287–13 300, 2021.

[15] 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 *Proceedings of the 33rd Advances in Neural Information Processing Systems (NIPS'19)*, 2019, pp. 10 197–10 207.

[16] G. Qiang, "Research on software vulnerability detection method based on improved CNN model," *Scientific Programming*, vol. 2022, 2022.

[17] C. B. Şahin, "DCW-RNN: Improving class level metrics for software vulnerability detection using artificial immune system with clock-work recurrent neural network," in *Proceedings of the 15th International Conference on INnovations in Intelligent SysTems and Applications (INISTA'21)*, 2021, pp. 1–8.

[18] W. Lin and S. Cai, "An empirical study on vulnerability detection for source code software based on deep learning," in *Proceedings of the 21st IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C'21)*, 2021, pp. 1159–1160.

[19] 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.

[20] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet?" *arXiv preprint arXiv:2009.07235*, 2020.

[21] Y. Zhang and B. Wallace, "A sensitivity analysis of (and practitioners' guide to) convolutional neural networks for sentence classification," *arXiv preprint arXiv:1510.03820*, 2015.

[22] H. Feng, X. Fu, H. Sun, H. Wang, and Y. Zhang, "Efficient vulnerability detection based on abstract syntax tree and deep learning," in *Proceedings of the 39th IEEE Conference on Computer Communications Workshops (INFOCOM'20 Workshop)*, 2020, pp. 722–727.

[23] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, and D. Jiang, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[24] Y. Wu, D. Zou, S. Dou, W. Yang, D. Xu, and H. Jin, "VulCNN: an image-inspired scalable vulnerability detection system," in *Proceedings of the 44th International Conference on Software Engineering (ICSE'22)*, 2022, pp. 2365–2376.

[25] "CVE-2018-17958," https://nvd.nist.gov/vuln/detail/CVE-2018-17958, 2022, accessed: 2022-08.

[26] "CVE-2018-17962," https://nvd.nist.gov/vuln/detail/CVE-2018-17962, 2022, accessed: 2022-08.

[27] D. Zou, H. Qi, Z. Li, S. Wu, H. Jin, G. Sun, S. Wang, and Y. Zhong, "SCVD: A new semantics-based approach for cloned vulnerable code detection," in *Proceedings of the 14th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'2017)*, 2017, pp. 325–344.

[28] "tf-idf," https://en.wikipedia.org/wiki/Tf-idf, 2022, accessed: 2022-08.

[29] "Open-source code analysis platform for C/C++ based on code property graphs," https://joern.io/, 2022, accessed: 2022-08.

[30] D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt, "Fast pattern matching in strings," *SIAM Journal on Computing*, vol. 6, no. 2, pp. 323–350, 1977.

[31] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *Proceedings of the 31st International Conference on Machine Learning (ICML'14)*, 2014, pp. 1188–1196.

[32] X. Yuan, G. Lin, Y. Tai, and J. Zhang, "Deep neural embedding for software vulnerability discovery: Comparison and optimization," *Security and Communication Networks*, vol. 2022, pp. 1–12, 2022.

[33] D. Sculley, "Web-scale k-means clustering," in *Proceedings of the 19th International Conference on World Wide Web (WWW'10)*, 2010, pp. 1177–1178.

[34] S. Lloyd, "Least squares quantization in PCM," *IEEE Transactions on Information Theory*, vol. 28, no. 2, pp. 129–137, 1982.

[35] "Euclidean distance," https://en.wikipedia.org/wiki/Euclidean_distance, 2022, accessed: 2022-08.

[36] 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 (KDD'16)*, 2016, pp. 855–864.

[37] 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.

[38] "codebert-base," https://huggingface.co/microsoft/codebert-base, 2020.

[39] M. Pagliardini, P. Gupta, and M. Jaggi, "Unsupervised learning of sentence embeddings using compositional n-gram features," *arXiv preprint arXiv:1703.02507*, 2017.

[40] 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 (ESEC/FSE'21)*, 2021, pp. 292–303.

[41] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[42] "Tensors and dynamic neural networks in python with strong GPU acceleration (PyTorch)," https://pytorch.org, 2022, accessed: 2022-08.

[43] R. Thorndike, "Who belongs in the family?" *Psychometrika*, vol. 18, no. 4, pp. 267–276, 1953.

[44] M. Pagliardini, P. Gupta, and M. Jaggi, "Unsupervised learning of sentence embeddings using compositional n-gram features," *arXiv preprint arXiv:1703.02507*, 2017.

[45] J. Zhang, Y. Dong, Y. Wang, J. Tang, and M. Ding, "ProNE: Fast and scalable network representation learning." in *Proceedings of the*

28th International Joint Conference on Artificial Intelligence (IJCAI'19), vol. 19, 2019, pp. 4278–4284.

[46] "The LLVM compiler infrastructure," https://llvm.org/, 2022, accessed: 2022-08.

[47] 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 (ESEC/FSE'21)*, 2021, pp. 292–303.

[48] M. Fu and C. Tantithamthavorn, "LineVul: A transformer-based line-level vulnerability prediction," in *Proceedings of the 19th International Conference on Mining Software Repositories (MSR'22)*, 2022, pp. 608–620.

[49] D. Hin, A. Kan, H. Chen, and M. A. Babar, "LineVD: Statement-level vulnerability detection using graph neural networks," in *Proceedings of the 19th International Conference on Mining Software Repositories (MSR'22)*, 2022, pp. 596–607.

[50] "National vulnerability database (NVD)," https://www.nist.gov/programs-projects/national-vulnerability-database-nvd, 2022, accessed: 2022-08.

[51] "Checkmarx," https://www.checkmarx.com/, 2022, accessed: 2022-08.

[52] "Rough audit tool for security," https://code.google.com/archive/p/rough-auditing-tool-for-security/, 2022, accessed: 2022-08.

[53] "Flawfinder," http://www.dwheeler.com/flawfinde/r, 2022, accessed: 2022-08.

[54] J. Jang, A. Agrawal, and D. Brumley, "ReDeBug: Finding unpatched code clones in entire os distributions," in *Proceedings of 33rd IEEE Symposium on Security and Privacy (S&P'12)*, 2012, pp. 48–62.

[55] S. Kim, S. Woo, H. Lee, and H. Oh, "VUDDY: A scalable approach for vulnerable code clone discovery," in *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P'17)*, 2017, pp. 595–614.

[56] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "SourcererCC: Scaling code clone detection to big-code," in *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*, 2016, pp. 1157–1168.

[57] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[58] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, 2007, pp. 96–105.

[59] N. H. Pham, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Detection of recurring software vulnerabilities," in *Proceedings of the 35th International Conference on Automated Software Engineering (ASE'10)*, 2010, pp. 447–456.

[60] J. Li and M. D. Ernst, "CBCD: Cloned buggy code detector," in *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*, 2012, pp. 310–320.

[61] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, "VulPecker: An automated vulnerability detection system based on code similarity analysis," in *Proceedings of the 32nd Annual Conference on Computer Security Applications (ACSAC'16)*, 2016, pp. 201–213.