Function-level Vulnerability Detection Through Fusing Multi-Modal Knowledge

Chao Ni, Xinrong Guo, Yan Zhu*, Xiaodan Xu, and Xiaohu Yang
Zhejiang University, China. Email: {chaoni,22151096,yan.zhu,xiaodanxu,yangxh}@zju.edu.cn

***Abstract*—Software vulnerabilities damage the functionality of software systems. Recently, many deep learning-based approaches have been proposed to detect vulnerabilities at the function level by using one or a few different modalities (e.g., text representation, graph-based representation) of the function and have achieved promising performance. However, some of these existing studies have not completely leveraged these diverse modalities, particularly the underutilized image modality, and the others using images to represent functions for vulnerability detection have not made adequate use of the significant graph structure underlying the images.**

In this paper, we propose MVulD, a multi-modal-based function-level vulnerability detection approach, which utilizes multi-modal features of the function (i.e., text representation, graph representation, and image representation) to detect vulnerabilities. Specifically, MVulD utilizes a pre-trained model (i.e., UniXcoder) to learn the semantic information of the textual source code, employs the graph neural network to distill graph-based representation, and makes use of computer vision techniques to obtain the image representation while retaining the graph structure of the function. We conducted a large-scale experiment on 25,816 functions. The experimental results show that MVulD improves four state-of-the-art baselines by 30.8%-81.3%, 12.8%-27.4%, 48.8%-115%, and 22.9%-141% in terms of F1-score, Accuracy, Precision, and PR-AUC respectively.

***Index Terms*—Vulnerability Detection, Computer Vision, Deep Learning, Multi-Modal Code Representations**

I. Introduction

Software vulnerabilities make software systems unreliable and many approaches have been proposed to detect vulnerabilities. These approaches can be divided into three types: program analysis-based, machine learning-based, and deep learning-based and they have their own advantages in detecting vulnerabilities. For example, program analysis-based approaches [1], [2] aim to find specific types of vulnerabilities, machine learning-based approaches [3]–[5] target learning the characteristics of vulnerabilities with human-crafted metrics, and deep learning based approaches [6]–[9] build deep learning models to automatically learn both the metrics and detection models simultaneously. Among them, deep learning-based approaches perform better in detecting vulnerability at function-level [10], [11].

For a given function, it usually has various presentations: source code tokens, the abstract syntax tree, the control flow graph, the control dependency graph, or the data dependency graph. These different representations have been partially used for vulnerability detection. For example, some works adopt one type of representation for a function (e.g., source code only [7], [12], abstract syntax tree only [13], different graph representations [6], [9], [14]–[16]). Even, some works [17], [18] merely adopted computer vision technology into software vulnerability detection and achieved good performance on VD. Meanwhile, some works [9], [15] try to fuse different representations of a function for better detecting vulnerabilities. For example, simultaneously utilizing several code representations (i.e., AST, CFG, PDF, and code sequences [16]) and embedding them with a deep learning model.

Though previous approaches have achieved promising performance in detecting vulnerability at the function level by using different representations of functions, there are still a few limitations: ① different modalities have varying representation abilities of a function according to previous findings [19]. However, these modalities are not fully utilized by existing works, especially the neglected image modality has also proved its utility for VD tasks but it has not been fused with other representations. ② Computer vision techniques can help to detect vulnerabilities in functions that are represented as images. However, the generated images of existing works [17], [18] only consider the importance of nodes while ignoring the relative positional relationships of nodes. In this way, they destroy the overall structure of the function which is crucially important.

To fully exploit the code semantics from multiple code representations, in this paper, we propose a novel multi-Modal-based function-level Vulnerability Detection approach called MVulD, which leveraging multi-modal content involving textural information, image information, and graph information of the source code as well as the mapping information between code structure and the position inside an image. In particular, we adopt a pre-trained model (i.e., UniXcoder [20]) to extract semantic information for textural representation. For the graph-based representation, we extract the code property graph (CPG) of a function and adopt a graph-based model as an encoder for inferring. As for image representation, the CPG is transformed into an image, and a transformer-based CV model is utilized (i.e., SwinV2 [21]) for the image encoder to obtain the global information. Finally, the different code representations from the three dimensions are fused to produce the multi-modal features of the function, which is used to build the final classification model.

we evaluate the effectiveness of MVulD by conducting a large-scale study (cf. Section V-A) on 25,816 functions and comparing with four state-of-the-art baselines in terms of four

* Yan Zhu is the corresponding author.

2643-1572/23/$31.00 ©2023 IEEE
DOI 10.1109/ASE56229.2023.00084

performance metrics. The experimental results indicate the priority of our proposed approach. More precisely, MVulD obtains 0.272 of F1-score, 0.880 of Accuracy, 0.183 of Precision, and 0.231 of PR-AUC, which improves baselines by 30.8%-81.3%, 12.8%-27.4%, 48.8%-115%, and 22.9%-141%, respectively. We also investigate the practical effectiveness of MVulD on the Top-25 Most Dangerous CWEs (cf. Section V-B) and it can also achieve high accuracy. Eventually, this paper makes the main contributions as below:

- We propose an effective Multi-Modal based function-level Vulnerability Detection approach named MVulD, which fully fuses three important modals (i.e., textual semantic, graph structure, and image information) for vulnerability detection[1].
- We conduct large-scale experiments on 25,816 functions and experimental results demonstrate that MVulD outperforms the state-of-the-art baselines by a significant margin.

II. Related Work

Vulnerability detection (VD) is a very crucial yet challenging task in security, and multiple methods have been proposed to detect vulnerabilities, which can be broadly divided into three categories: Program analysis (PA)-based methods, Machine learning (ML)-based methods, and deep learning (DL)-based methods.

PA-based methods detect vulnerabilities according to predefined vulnerability patterns manually generated by human experts. Since experts can not generate all patterns of different vulnerabilities, PA-based vulnerability detectors (e.g., RATS [22], Checkmarx [1], and FlawFinder [2]) can only detect limited types of vulnerabilities.

ML-based methods train machine learning models on certain features to detect vulnerabilities [3]–[5], [23]. These methods are actually semi-automatic since they require human-crafted metrics as features to characterize vulnerabilities, which consumes a lot of time to collect [24], [25].

DL-based methods [7], [9], [16], [26], [27] can automatically learn features from source code for VD, which can be further classified into three types according to their code representation techniques. (1) Token-based methods [7], [27]. These methods treat source code as text and leverage natural language processing techniques to detect vulnerabilities. (2) Tree-based methods [13]. Instead of treating source code as natural language sentences, tree-based methods leverage the structural information contained in a program. (3) Graph-based methods. To obtain a more comprehensive representation of source code, some approaches [6], [9], [14], [16], [16] distill different types of graph representations (i.e., control flow graph (CFG), control dependency graph (CDG), and data dependency graph (DDG)) and employ graph neural networks (GNN) to make vulnerability detection.

Meanwhile, some works adopt the technology from computer vision to creatively represent a function [18] or a source code file [17]. However, the generated image of these works damages the graph structure of studied functions which is crucially important to understand the semantics.

Apart from utilizing one representation of a function, some works turn to using multiple representations (e.g., AST, CFG, DFG, DDG, and code sequence) for vulnerability detection tasks [15], [16], [20]. However, to the best of our knowledge, no one has attempted to fuse the image feature of source code with features of other modalities (i.e., text or graph) to detect vulnerabilities. Therefore, we want to investigate the potentiality of such a combination of image features, text features, and graph features.

III. Approach

We propose a multi-modal approach MVulD (illustrated in Fig. 1) and it contains four main phases: ❶ graph extraction aims to obtain the structure representation of the function, ❷ image generation helps to transform the structural graph into graphical representation, ❸ multi-modal feature fusion builds the relationship of various modalities to obtain enriched code representations and ❹ classification will be applied to detect whether a function is vulnerable or not. Details of MVulD are presented in the following subsections.

A. Graph Extraction

In our study, we extract the code property graph(CPG) [15] of the function, a multigraph that combines properties of AST, CFG, and PDG in a joint data structure. Each graph representation can provide a unique view of the source code emphasizing different aspects of the underlying program. In particular, AST faithfully encodes how statements are nested to produce programs, CFG provides control flow information and PDG provides data flow information. Following previous works [18], [19], all three types of graphs are extracted with the Joern program [15]. Especially, when parsing the source code of a function into Joern, we can obtain CPG, a multigraph whose nodes represent the set of statements and edges represent the set of flow information among statements.

B. Image Generation

Computer vision (CV) has achieved great success in many applications (e.g., object detection [28]–[31] and image segmentation [31]–[35]) with the rapid development of deep learning techniques [36]. Recently, a few researchers [17], [18] adopt novel techniques from CV to address the vulnerability detection tasks in software engineering and indeed achieve promising performance. Therefore, we want to transform the code into an image and completely leverage the novel technique in CV to better embed a function. Previous works turn function into an image in different ways. For example, Wu et al. [18] use the embedded vectors of the statements as the values in the image, while Chen et al. [17] use the ASCII values of code characters as the values in the image. To some degree, these two types of image generation will damage the information of the function. Besides, it is not easy for a developer to recognize the relevance between the generated

[1]https://github.com/jacknichao/MVulD

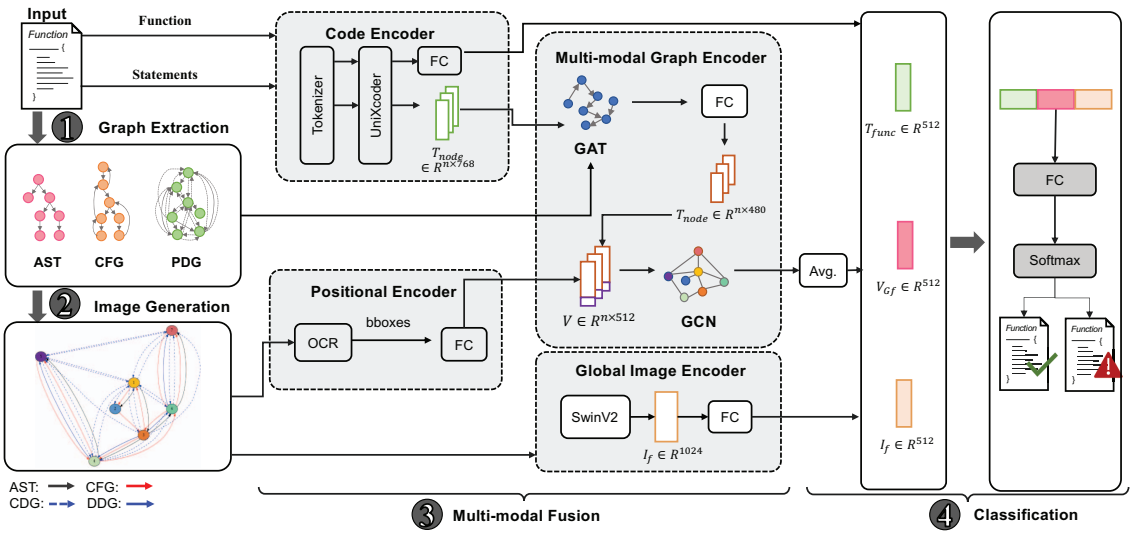

Fig. 1. The framework of MVulD. The proposed model fuses the multiple modalities of a function including text, graph, and image representations to detect function-level vulnerability.

image and the analyzed function when visualizing this image. Different from previous works, we directly draw the image of the studied function with the information of CPG. Therefore, when visualizing this image, the developers can easily see the structure (AST, CFG, and PDG) and the statement nodes inside it.

During the image generation phase, we adopt the tool named *Graphviz* [37] to convert the CPG generated by *Joern* to an image. To represent each statement node of the function, we use numbered circular nodes with different colors, and the number inside each node corresponds to the line number of each statement. Meanwhile, to better distinguish the difference among the three structures' information in the image, we adopt different colors as well as different line types to draw the connection lines between two statement nodes.

C. Multi-modal Feature Fusion

Each modality differs in the ability to represent source code and fusing various modalities can promote the understanding of source code [19]. In our study, we design four encoders: ① global image encoder, ② code encoder, ③ positional encoder, and ④ multi-modal graph encoder to get three types of code features (i.e., global image feature, text feature at the function level, and graph feature) and combine them to obtain the final enriched multi-modal code representations. The details of each encoder are introduced as follows.

Global Image Encoder. To acquire the global image features, MVulD employs a large-scale vision model named SwinV2 [21] as the major part of the global image encoder, which is pre-trained on ImageNet [38] and has a promising performance in various tasks (e.g., image classification [39]–[41], object detection [28]–[31], and semantic segmentation [33], [42]–[44]). More precisely, for a given image I, SwinV2 takes it as input and outputs the vectorized feature representation. The dimension of the output feature vector is 1,024, which is then projected to 512 by a fully-connected layer to obtain the final global image features, which can be denoted as $I_f \in \mathbb{R}^{1\times D}$, where D equals to 512.

Code Encoder. Apart from the structure information of a function, the semantic information of the source code is also significant. Pre-trained models associated with programming languages [20], [45]–[47] are widely used in software engineering and indeed have achieved much success since they have a good understanding of tokens in the programming language domain. UniXcoder, proposed by Guo et al. [20], is a unified pre-trained model incorporating semantic and syntax information from both code comment and AST, and we adopt it as the code extractor in MVulD to embed the code features both at the function level and statement level. More precisely, given the source code of a function, MVulD first splits the function into individual statements. Then, the entire function and the individual statements (code lines) are tokenized by the pre-trained BPE (Byte Pair Encoding) tokenizer of UniXcoder [20]. Following that, UniXcoder transforms the input function to 768-dimensional vectors at both the function level and statement level. That is, the code encoder of MVulD generates $n+1$ code representations in total, i.e., one embedding for the overall function, and n embeddings for all the lines of code.

Similar to obtaining the final image representation of a function, MVulD further projects the obtained function-level code feature through a fully-connected layer to get the final encoded text feature. The final function-level text feature can be denoted as $T_f \in \mathbb{R}^{1\times D}$, while the statement-level text features $T_{node} = \{t_1, t_2, ..., t_n\}$ will serve as the initial node embedding of the graph (i.e., CPG).

Positional Encoder. The image of the graph generated in the second phase illustrates the structure information but ignores the relationship between the nodes in the image (i.e., the statements in the code), which can provide vital relation information and spatial information and consequently help to better understand the semantics of functions. Therefore, the position information of each node inside the generated code image should be obtained. We adopt the widely used open-source character recognition (OCR) techniques to extract the bounding boxes of nodes and design a positional encoder to obtain the positional feature embedding. Specifically, MVulD uses an accurate scene text detector named East [29] to obtain the corresponding bounding boxes of each text instance (i.e., the line number inside each node). When obtaining the bounding boxes, MVulD applies Python-tesseract[2] to further extract the line number of each node (i.e., statement numbers) in the image, which can be used to align the positional feature embedding with its corresponding semantic feature embedding of textual statements. In cases of nodes with no bounding boxes (e.g., the text cannot be recognized by OCR), the zero padding strategy is applied to them. After that, the detected bounding boxes of each node can be described using the bottom left (x_1, y_1) and top right (x_2, y_2) coordinates normalized by the image size.

Consequently, the input of an entire image can be represented by: $bboxes_{input} = \{bbox_1, bbox_2, ..., bbox_n\}$, where $bbox_i = \{x1, y1, x2, y2\}$, and n is the number of nodes in the image. Meanwhile, the positional encoder of MVulD transforms the input representations into the final encoded positional feature embedding: $bboxes_f(bboxes_{output}) = \left\{bbox_1^{'}, bbox_2^{'}, ..., bbox_n^{'}\right\}$.

Multi-modal Graph Encoder. Graph neural networks (GNNs) [6], [9], [16], [19] have shown the capability of modeling relationships between nodes in a given graph to learn both structural information and semantic information. Graph attention network (GAT) [48] and graph convolutional network (GCN) [49] are two mainstream GNNs. To obtain the enriched graph features, MVulD constructs the multi-modal graph encoder mainly by utilizing both GAT and GCN. We begin by applying GAT to the CPG. Since our CPG is a directed graph, GAT is capable of flexibly handling asymmetric relationships and directionalities between nodes through attention mechanisms. After GAT, we put the enriched statement feature embeddings jointly with positional embeddings extracted from the image into the GCN. GCNs have been successfully used in image-related tasks that require reasoning such as image captioning [50], [51] and image-sentence retrieval [52], [53], which can be used to further enhance node representation by performing convolutional operations in the local neighborhood of the image.

In the first step, we take n statement embeddings(i.e.,T_{node}) as input, and the GAT model is adopted to optimize them on the basic CPG. Specifically, GAT utilizes the attention mechanism to aggregate the neighborhood information of each node and embeds edge information between neighboring statements in an incremental manner to propagate the information. The aggregation strategy in GAT can be described as follows.

$$Z_i^{(l)} = W^{(l)} h_i^{(l)} \tag{1}$$

$$e_{i,j}^{(l)} = LeakyReLU\left(\vec{a}^{(l)^T}\left(z_i^{(l)} \parallel z_j^{(l)}\right)\right) \tag{2}$$

$$\alpha_{i,j}^{(l)} = \frac{exp\left(e_{i,j}^{(l)}\right)}{\sum_{k \in N(i)} exp\left(e_{i,k}^{(l)}\right)} \tag{3}$$

$$h_i^{(l+1)} = \sigma\left(\sum_{j \in N(i)} \alpha_{i,j}^{(l)} z_j^{(l)}\right) \tag{4}$$

where l indicates the current state, $h_i^{(l)}$ represents the node embedding vectors at current layer, $W^{(l)}$ is the learnable weight matrix, $\vec{a}$ is a learnable weight vector, and σ defines the activation function. After GAT, a fully-connected layer is leveraged to get the output node representations: $T_{gat} = \left\{t_1^{'}, t_2^{'}, ..., t_n^{'}\right\}$.

In the second step, we construct a new graph $G = (V, R)$ based on the image, where V is the set of nodes detected in the image, and R is obtained by calculating the affinity edge of each pair of nodes. We initialize V with T_{gat} and corresponding positional embeddings $bboxes_f$, which can be described by: $V = \left\{\left(t_1^{'}, bbox_1^{'}\right), \left(t_2^{'}, bbox_2^{'}\right), ..., \left(t_n^{'}, bbox_n^{'}\right)\right\}, V \in \mathbb{R}^{n \times D}$. R can be constructed using the following formulation:

$$R_{i,j} = \phi\left(v_i\right)^T \gamma\left(v_j\right) \tag{5}$$

where $v_i, v_j \in V$, $\phi(.)$ and $\gamma(.)$ are two fully connected layers that can be learned via back-propagation during the training phase. Considering GCNs is good at capturing the local patterns and feature relationships in the spatial domain between pixels in an image, we apply the GCNs incorporating the residual connections [52] following previous work [49] to perform visual reasoning on $G = (V, R)$. A single GCN layer can be described by the following equation:

$$V_g^l = W_r^l\left(R^l V^{l-1} W_g^l\right) + V^{l-1} \tag{6}$$

where V is the node embedding, R is the affinity matrix, W_g is a learnable weights matrix of the GCN, W_r is the residual weights matrix, and l is the number of GCN layer. After GCN, we can obtain the enhanced node embedding: $V_G = \{v_{g1}, v_{g2}, ..., v_{gn}\}, V_G \in \mathbb{R}^{n \times D}$.

After the two steps, global structure information is attached to each line of code(i.e., node). The final graph feature representation V_{Gf} is obtained by averaging the information from all nodes, which can be represented as:

$$V_{Gf} = \frac{1}{n}\sum_{i=1}^{n} V_{gi} \tag{7}$$

Before making classifications, MVulD simply concatenate the three obtained features I_f, V_{Gf}, and T_f from respective encoder together to obtain the multi-modal code features $F = [I_f, V_{Gf}, T_f]$.

[2]https://github.com/madmaze/pytesseract

D. Classification

The target of MVulD is to detect whether a function is vulnerable or not, which is a typical binary classification task. By passing the final representation F through a fully-connected layer (i.e., the final classification layer) and the $softmax$ layer, the probability distribution of a class label given an input function can be obtained. During the training phase, we use the cross-entropy loss to guide the optimization process of MVulD by comparing the difference between the prediction probability of the model and the ground-truth label. The objective is to make the cross entropy as small as possible so that the prediction result is closer to the real one.

IV. Experimental Settings

A. Research Question

We focus on the following two research questions:

- *RQ-1: How does MVulD perform compared to baselines for function-level vulnerability detection?*
- *RQ-2: How does MVulD perform in detecting the Top-25 Most Dangerous CWEs?*

B. Datasets

We choose the Big-Vul dataset [54] since it is one of the largest vulnerability datasets collected from 348 open-source GitHub projects spanning 91 different vulnerability types. We also perform three filtering steps (i.e., (1) remove the improperly truncated functions; (2) remove functions that failed to be parsed by *Joern* [15], and (3) filter functions whose LOCs<=100 (i.e., occupy 95%) to avoid noise) on the original dataset to obtain the valid function and the details are displayed in Table I Finally, we randomly split the final dataset (LOCs<=100) into training (80%), validation (10%), and testing (10%).

TABLE I
THE STATISTIC OF STUDIED DATASET

Datasets	Vul.	Non-vul.	# Total	%Ratio
Original Big-Vul	10,900	177,736	188,636	5.78%
Step-1: Clean Dataset	9,480	174,270	183,750	5.16%
Step-2: Dataset after *Joern*	5,260	96,308	101,568	5.19%
Step-3: Dataset (LOCs<=100)	4,069	92,460	96,529	4.22%

C. Baseline Models

We consider four state-of-the-art approaches to better illustrate the performance difference among SOTAs and MVulD. More precisely, we consider three graph-based approaches (i.e., IVDetect, Devign, and Reveal) and one pre-trained approach (i.e., UniXcoder [20] considering both AST and code comment)). IVDetect [6] considers five types of feature representations (i.e., sequence of sub-tokens, AST, variable names and types, data dependency context and control dependency context), Devign [16] considers four code semantic representations (i.e., AST, CFG, PDG, and code sequences) and Reveal [9] considers the code property graph (CPG) of a function.

V. Experimental Results

A. [RQ-1]: Effectiveness of Vulnerability Detection

TABLE II
VULNERABILITY DETECTION RESULTS OF MVULD COMPARED AGAINST FOUR BASELINES.

Methods	F1-score	Accuracy	Recall	Precision	PR-AUC
UniXcoder	0.208	0.780	**0.686**	0.123	0.188
IVDetect	0.160	0.737	0.590	0.093	0.106
Devign	0.157	0.720	0.617	0.090	0.112
Reveal	0.150	0.691	0.649	0.085	0.096
MVulD	**0.272**	**0.880**	0.531	**0.183**	**0.231**

Many DL-based approaches have been proposed for function-level vulnerability detection by using different code representation techniques, and Siow et al. [19] have proved that different code representations contain varying program semantics. Meanwhile, some works [17], [18] have also tried to visualize source code as images and fed them into a CV model for vulnerability detection. However, these methods have limitations in fully exploiting all different modalities to detect vulnerabilities. Therefore, MVulD integrates token-based features, graph-based features, and image features together to boost its effectiveness in vulnerability detection. We, therefore, want to investigate to what extent the detection performance can MVulD achieve with the multi-modal features.

To comprehensively compare the performance difference between the four baselines and MVulD, we train all the models on the training dataset and use the validation set to monitor the training process by epoch and select the best one based on the optimal F1-score, while the testing dataset is used to test. We also consider the five commonly-used performance measures including Accuracy, Precision, Recall, F1-score, and PR-AUC. In addition, we set the same seed and unify the version of *Joern* [15] used for the graph extraction phase in both baseline models and our MVulD to eliminate the potential bias of randomness.

TABLE III
THE TRUE POSITIVE RATE(TPR) OF MVULD AND UNIXCODER FOR FUNCTION-LEVEL VULNERABILITY DETECTION ON SOME VULNERABILITY TYPES.

CWE Type	Name	TPR		# Vuln.
		UniXcoder	MVulD	
CWE-772	Missing Release of Resource after Effective Lifetime	1/2	0/2	2
CWE-269	Improper Privilege Management	3/3	2/3	3
CWE-310	Cryptographic Issues	2/3	0/3	3
CWE-59	Link Following	2/3	1/3	3
CWE-404	Improper Resource Shutdown or Release	3/4	2/4	4
CWE-415	Double Free	3/7	5/7	7
CWE-416	Use After Free	5/10	2/10	10
CWE-362	Race Condition	8/12	4/12	12
CWE-190	Integer Overflow or Wraparound	13/14	9/14	14
CWE-189	Numeric Errors	11/16	12/16	16
CWE-476	NULL Pointer Dereference	7/17	5/17	17
CWE-125	Out-of-bounds Read	25/27	14/27	27
CWE-399	Resource Management Errors	20/30	18/30	30
CWE-20	Improper Input Validation	23/31	16/31	31
CWE-264	Permissions, Privileges, and Access Controls	18/31	15/31	31
CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	71/101	62/101	101

The evaluation results are provided in Table II with the best performances highlighted in bold. According to the results, we find that our MVulD outperforms all baseline methods by a large margin on almost all performance measures. Specifically, MVulD obtains 0.272 of F1-score, 0.880 of Accuracy, 0.183 of Precision, and 0.231 of PR-AUC, which improves baselines by 30.8%-81.3%, 12.8%-27.4%, 48.8%-115%, and 22.9%-141% respectively. However, MVulD shows a decrease in terms of Recall by 16.1% on average. Since UniXcoder performs best on Recall among all the approaches, we undertake a deeper analysis by comparing how UniXcoder and our MVulD perform on different types of vulnerabilities. In our experiment, we delete six CWEs with only one vulnerable instance since they may not properly reflect the true detect performance of models. Then, we use True Positive Rate (TPR) as the evaluation metric, which measures the proportion of the vulnerable functions that are accurately detected by methods and the number of real vulnerable functions. In Table III, the investigated vulnerability types are ranked according to the number of vulnerabilities, and only the types with differing performances between UniXcoder and MVulD are displayed. From the results, our MVulD (5/7 and 12/16) can find more vulnerable instances than UniXcoder (3/7 and 11/16) for CWE-415 and CWE-189. However, for the remaining CWE types, UniXcoder has shown better performance than MVulD. For example, UniXcoder can detect CWE-772 and CWE-310 correctly while MVulD cannot.

Answer to RQ-1► MVulD can achieve superior performance on function-level vulnerability detection than the state-of-the-art baselines, especially given the overwhelming results at the F1-score, Accuracy, Precision, and PR-AUC. It also indicates that the multi-modal features can bring significant improvement to the detection performance of MVulD.◄.

B. [RQ-2]: Effectiveness on Top-25 Most Dangerous CWEs

The 2022 Common Weakness Enumeration Top 25 Most Dangerous Software Weaknesses list[3] (CWE Top 25) demonstrates the currently most common and impactful software weaknesses. Those unresolved weaknesses associated with software systems may result in privacy risks and exploitable vulnerabilities, which allow attackers to completely take over a system, steal data, or prevent applications from working. To explore the effectiveness of MVulD on practical vulnerability detection scenarios, we make a further investigation to show the detection accuracy of MVulD for the Top-25 Most Dangerous CWEs.

In the testing set mentioned in Section IV-B, we only choose functions that belong to the Top-25 Most Dangerous CWEs as our studied dataset and then group them by their CWE type. Since not all of the Top-25 Most Dangerous CWEs are included, we finally collect 13 CWE types in Table IV. After that, we select the MVulD having the highest F1-score on the validation set to evaluate the accuracy of our method for the obtained CWEs. The accuracy measures the vulnerable functions that can be correctly detected by our method, which can be computed using TPR.

[3]https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html

TABLE IV
THE TPR OF OUR MVULD FOR THE TOP-25 MOST DANGEROUS CWES.

Rank	CWE Type	Name	TPR	Proportion
1	CWE-787	Out-of-bounds Write	0.833	5/6
3	CWE-89	SQL Injection	0	0/1
4	CWE-20	Improper Input Validation	0.516	16/31
5	CWE-125	Out-of-bounds Read	0.519	14/27
7	CWE-416	Use After Free	0.200	2/10
8	CWE-22	Path Traversal	1	1/1
11	CWE-476	NULL Pointer Dereference	0.294	5/17
13	CWE-190	Integer Overflow or Wraparound	0.643	9/14
16	CWE-862	Missing Authorization	0	0/1
19	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	0.614	62/101
22	CWE-362	Race Condition	0.333	4/12
23	CWE-400	Uncontrolled Resource Consumption	0	0/1
24	CWE-611	Improper Restriction of XML External Entity Reference	0	0/1
		Sum	**0.529**	**118/223**

Table IV shows the performance of our MVulD for each CWE type in terms of TPR. From the results, we can draw the following conclusions: 1) MVulD accurately detects 118 of the total 223 vulnerabilities affected by the Top-25 Most Dangerous CWEs, accounting for 52.9%. 2) The TPR of MVulD on Top-1 CWEs is up to 83.3%, while the accuracy on Top-5 CWEs is 53.8%. 3) When it comes to CWE-787 (Out-of-bounds Write), CWE-125 (Out-of-bounds Read), and CWE-119 (Improper Restriction of Operations within the Bounds of a Memory Buffer), MVulD can achieve an accuracy of 83.3%, 51.9%, and 61.4% respectively. It reveals that MVulD excels in detecting memory-related vulnerabilities.

However, MVulD fails to identify some types of vulnerabilities including CWE-89, CWE-862, CWE-400, and CWE-611. The lack of vulnerable instances of such CWEs may account for the poor accuracy of MVulD.

Answer to RQ-2► Generally, MVulD can correctly detect 52.9% of the vulnerable functions affected by the Top-25 Most Dangerous CWEs. It manifests that MVulD has the capacity to discover vulnerabilities in practical applications.◄

VI. CONCLUSION

In this paper, we propose MVulD, a novel DL-based approach that can learn multi-modal code features from different knowledge resources to detect function-level vulnerability. By leveraging UniXcoder to encode the text features, GNN models to encode the graph features, and SwinV2 to encode the image features, MVulD achieves a new state-of-the-art in comparison to other baseline approaches on real-world dataset (i.e., Big-Vul). Besides, exhaustive experiments have also been conducted to further study the effectiveness of different modalities of the function, the result shows that the diverse code representations are all indispensable and can be effectively leveraged by MVulD to capture rich program semantics and make vulnerability detection.

ACKNOWLEDGEMENTS

This research is supported by the Fundamental Research Funds for the Central Universities (No. 226-2022-00064), the National Natural Science Foundation of China (No. 62202419), the Ningbo Natural Science Foundation (No. 2022J184), and the State Street Zhejiang University Technology Center.

REFERENCES

[1] "Checkmarx," 2022. [Online]. Available: https://www.checkmarx.com/
[2] "Flawfinder," 2022. [Online]. Available: https://dwheeler.com/flawfinder/
[3] M. Backes, B. Köpf, and A. Rybalchenko, "Automatic discovery and quantification of information leaks," in *2009 30th IEEE Symposium on Security and Privacy*. IEEE, 2009, pp. 141–153.
[4] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner, "Detecting format string vulnerabilities with type qualifiers," in *10th USENIX Security Symposium (USENIX Security 01)*, 2001.
[5] L. K. Shar, L. C. Briand, and H. B. K. Tan, "Web application vulnerability prediction using hybrid program analysis and machine learning," *IEEE Transactions on dependable and secure computing*, vol. 12, no. 6, pp. 688–707, 2014.
[6] Y. Li, S. Wang, and T. N. Nguyen, "Vulnerability detection with fine-grained interpretations," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 292–303.
[7] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," *arXiv preprint arXiv:1801.01681*, 2018.
[8] 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.
[9] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet," *IEEE Transactions on Software Engineering*, 2021.
[10] B. Steenhoek, M. M. Rahman, R. Jiles, and W. Le, "An empirical study of deep learning models for vulnerability detection," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2237–2248.
[11] S. Cao, X. Sun, L. Bo, R. Wu, B. Li, and C. Tao, "Mvd: Memory-related vulnerability detection based on flow-sensitive graph neural networks," *arXiv preprint arXiv:2203.02660*, 2022.
[12] 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.
[13] L. Mou, G. Li, Z. Jin, L. Zhang, and T. Wang, "Tbcnn: A tree-based convolutional neural network for programming language processing," *arXiv preprint arXiv:1409.5718*, 2014.
[14] 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 *IJCAI*, 2019, pp. 4665–4671.
[15] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 590–604.
[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," *Advances in neural information processing systems*, vol. 32, 2019.
[17] J. Chen, K. Hu, Y. Yu, Z. Chen, Q. Xuan, Y. Liu, and V. Filkov, "Software visualization and deep transfer learning for effective software defect prediction," in *Proceedings of the ACM/IEEE 42nd international conference on software engineering*, 2020, pp. 578–589.
[18] Y. Wu, D. Zou, S. Dou, W. Yang, D. Xu, and H. Jin, "Vulcnn: An image-inspired scalable vulnerability detection system," 2022.
[19] J. K. Siow, S. Liu, X. Xie, G. Meng, and Y. Liu, "Learning program semantics with code representations: An empirical study," *arXiv preprint arXiv:2203.11790*, 2022.
[20] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," *arXiv preprint arXiv:2203.03850*, 2022.
[21] Z. Liu, H. Hu, Y. Lin, Z. Yao, Z. Xie, Y. Wei, J. Ning, Y. Cao, Z. Zhang, L. Dong *et al.*, "Swin transformer v2: Scaling up capacity and resolution," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 12 009–12 019.
[22] S. S. Inc., "Rough auditing tool for security (rats)," 2022. [Online]. Available: https://code.google.com/p/rough-auditing-tool-for-security/
[23] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck, "Automatic inference of search patterns for taint-style vulnerabilities," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 797–812.
[24] 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.
[25] 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.
[26] H. K. Dam, T. Tran, T. Pham, S. W. Ng, J. Grundy, and A. Ghose, "Automatic feature learning for vulnerability prediction," *arXiv preprint arXiv:1708.02368*, 2017.
[27] 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.
[28] B. Shi, X. Bai, and C. Yao, "An end-to-end trainable neural network for image-based sequence recognition and its application to scene text recognition," *IEEE transactions on pattern analysis and machine intelligence*, vol. 39, no. 11, pp. 2298–2304, 2016.
[29] X. Zhou, C. Yao, H. Wen, Y. Wang, S. Zhou, W. He, and J. Liang, "East: an efficient and accurate scene text detector," in *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, 2017, pp. 5551–5560.
[30] W. Wang, E. Xie, X. Li, W. Hou, T. Lu, G. Yu, and S. Shao, "Shape robust text detection with progressive scale expansion network," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 9336–9345.
[31] M. Liao, Z. Wan, C. Yao, K. Chen, and X. Bai, "Real-time scene text detection with differentiable binarization," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 34, no. 07, 2020, pp. 11 474–11 481.
[32] T. M. Quan, D. G. C. Hildebrand, and W.-K. Jeong, "Fusionnet: A deep fully residual convolutional neural network for image segmentation in connectomics," *Frontiers in Computer Science*, p. 34, 2021.
[33] V. Badrinarayanan, A. Kendall, and R. Cipolla, "Segnet: A deep convolutional encoder-decoder architecture for image segmentation," *IEEE transactions on pattern analysis and machine intelligence*, vol. 39, no. 12, pp. 2481–2495, 2017.
[34] L.-C. Chen, Y. Zhu, G. Papandreou, F. Schroff, and H. Adam, "Encoder-decoder with atrous separable convolution for semantic image segmentation," in *Proceedings of the European conference on computer vision (ECCV)*, 2018, pp. 801–818.
[35] C. Peng, X. Zhang, G. Yu, G. Luo, and J. Sun, "Large kernel matters–improve semantic segmentation by global convolutional network," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 4353–4361.
[36] S. Pouyanfar, S. Sadiq, Y. Yan, H. Tian, Y. Tao, M. P. Reyes, M.-L. Shyu, S.-C. Chen, and S. S. Iyengar, "A survey on deep learning: Algorithms, techniques, and applications," *ACM Computing Surveys (CSUR)*, vol. 51, no. 5, pp. 1–36, 2018.
[37] Graphviz, 2022. [Online]. Available: https://graphviz.org
[38] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.
[39] E. A. Smirnov, D. M. Timoshenko, and S. N. Andrianov, "Comparison of regularization methods for imagenet classification with deep convolutional neural networks," *Aasri Procedia*, vol. 6, pp. 89–94, 2014.
[40] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.
[41] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.
[42] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 3431–3440.

[43] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *International Conference on Medical image computing and computer-assisted intervention*. Springer, 2015, pp. 234–241.

[44] P. Luc, C. Couprie, S. Chintala, and J. Verbeek, "Semantic segmentation using adversarial networks," *arXiv preprint arXiv:1611.08408*, 2016.

[45] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," *arXiv preprint arXiv:1910.13461*, 2019.

[46] R.-M. Karampatsis and C. Sutton, "Scelmo: Source code embeddings from language models," *arXiv preprint arXiv:2004.13214*, 2020.

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

[48] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *stat*, vol. 1050, p. 20, 2017.

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

[50] X. Li and S. Jiang, "Know more say less: Image captioning based on scene graphs," *IEEE Transactions on Multimedia*, vol. 21, no. 8, pp. 2117–2130, 2019.

[51] X. Yang, K. Tang, H. Zhang, and J. Cai, "Auto-encoding scene graphs for image captioning," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2019, pp. 10 685–10 694.

[52] K. Li, Y. Zhang, K. Li, Y. Li, and Y. Fu, "Visual semantic reasoning for image-text matching," in *Proceedings of the IEEE/CVF international conference on computer vision*, 2019, pp. 4654–4662.

[53] C. Liu, Z. Mao, T. Zhang, H. Xie, B. Wang, and Y. Zhang, "Graph structured network for image-text matching," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2020, pp. 10 921–10 930.

[54] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "Ac/c++ code vulnerability dataset with code changes and cve summaries," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 508–512.