Abundant Modalities Offer More Nutrients: Multi-Modal-Based Function-level Vulnerability Detection

CHAO NI*†, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
XIN YIN*, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
XINRUI LI, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
XIAODAN XU, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
ZHI YU‡§, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China

Software vulnerabilities are weaknesses in software systems that can lead to significant cybersecurity risks. Recently, several deep learning (DL)-based approaches have been proposed to detect vulnerabilities at the function level. These approaches typically utilize one or a few different modalities (e.g., text representation and graph-based representation) of the function, and have shown promising performance. However, existing studies have not fully leveraged diverse modalities, particularly those that use images to represent functions for vulnerability detection. These approaches often fail to make sufficient use of the important graph structure underlying the images. In this paper, we propose MVulD+, a multi-modal-based function-level vulnerability detection approach, which fuses multi-modal features of the function (i.e., text representation, graph representation, and image representation) to detect vulnerabilities. Specifically, MVulD+ leverages a pre-trained model (i.e, UniXcoder) to capture the semantic information of the textual source code, uses a graph neural network to extract graph representations, and employs computer vision techniques to obtain image representations while preserving the graph structure of the function. To investigate the effectiveness of MVulD+, we conduct a large-scale experiment by comparing our approach with nine state-of-the-art baselines. Experimental results demonstrate that MVulD+ improves the DL-based baselines by 24.3%-125.7%, 5.2%-31.4%, 40.6%-192.2%, and 22.3%-186.9% in terms of F1-score, Accuracy, Precision, and PR-AUC, respectively.

CCS Concepts: • **Security and privacy → Software security engineering**.

Additional Key Words and Phrases: Vulnerability Detection, Computer Vision, Deep Learning, Multi-Modal Code Representations

*Equal contribution.
†Chao Ni is also with Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security.
‡Zhi Yu is also with Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security.
§Zhi Yu is the corresponding author.

Authors' Contact Information: Chao Ni, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, China, chaoni@zju.edu.cn; Xin Yin, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, China, xyin@zju.edu.cn; Xinrui Li, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, China, lixinrui@zju.edu.cn; Xiaodan Xu, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, China, xiaodanxu@zju.edu.cn; Zhi Yu, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, China, yuzhirenzhe@zju.edu.cn.

ACM 1557-7392/2025/4-ART
https://doi.org/10.1145/3731557

1 INTRODUCTION

Software vulnerabilities have been serious threats to software systems. Hackers can exploit undetected vulnerabilities and may cause significant losses to both enterprises and users. The OSSRA report[1] shows that, among studied 2,409 codebases, 81% contain at least one known open-source vulnerability. This underscores the need for increased focus on vulnerability detection (VD). To effectively detect software vulnerabilities, many automated VD approaches have been proposed [18, 36], which can be broadly categorized into three types: program analysis (PA)-based, machine learning (ML)-based, and deep learning (DL)-based approaches. PA-based approaches [1, 2, 24] focus on specific vulnerability types (e.g., BufferOverflow[2]) using manually defined rules or patterns, which rely heavily on expert knowledge. ML-based approaches [3, 41, 50, 51, 67, 68] employ human-crafted metrics to characterize vulnerabilities and train machine learning models on these metrics to detect vulnerabilities. DL-based approaches [6, 10, 12, 30, 32–34, 44, 48, 70] build deep learning models to automatically learn both the metrics [10, 12] and detection models [6, 30, 32–34, 44, 48, 70] simultaneously.

DL-based approaches have shown promising results in vulnerability detection at the function-level. A function can be represented in various forms, such as textural source code, abstract syntax tree (AST), control flow graph (CFG), control dependency graph (CDG), and data dependency graph (DDG). Existing approaches typically use one or more of these representations. For example, some approaches [33, 48] treat source code as text and leverage natural language processing techniques to detect vulnerabilities. Instead of treating source code as text, Mou et al. [40] proposed a tree-based convolutional neural network using ASTs to extract structural information from the code. To better capture program semantics, some approaches use various graph representations (e.g., CFG, CDG, and DDG) and build VD models with graph neural networks [6, 12, 30, 66, 70]. More recently, several studies [7, 63] have explored the feasibility of representing functions as images and adopted computer vision techniques to detect vulnerabilities. For example, DTL-DP [7] converts code into images based on the ASCII decimal values of code characters, while VulCNN [63] transforms code into PDG and applies centrality analysis to generate the corresponding image.

Building on previous efforts that use a single representation of functions, some studies in vulnerability detection leverage multi-modal features of source code [17, 58]. For example, Devign [70] builds the VD model by simultaneously considering multiple representations of functions, including AST, CFG, PDG, and code sequences. Reveal [6] uses the code property graph (CPG) [66] and the gated graph neural network to detect vulnerabilities, where CPG combines properties of AST, CFG, and PDG in a joint data structure. These multi-modal approaches are demonstrated to produce improved performance.

Though existing approaches have shown promising performance in detecting vulnerability by utilizing various modalities of functions, there are still a few limitations: ① Different modalities offer varying representation capabilities of a function, as highlighted in previous findings [53]. However, these modalities are not fully utilized by existing works, especially the image modality, which has also proven effective for VD tasks. ② Computer vision techniques can help to detect vulnerabilities in functions that are represented as images. However, existing approaches [7, 63] focus solely on the importance of the images, overlooking the need for feature alignment with other modalities, such as text and graph representations. In this way, they destroy the overall semantics of the function which is crucially important.

In this paper, to fully exploit the code semantics from multiple modalities, we propose a novel multi-Modal-based function-level Vulnerability Detection approach called MVulD+, which is an improved version of our previously proposed approach [42]. MVulD+ focuses on leveraging multi-modal content including text information, image information, and graph information of the source code. More specifically, for the text modality, MVulD+ extracts the semantic information of a function using the widely used pre-trained model UniXcoder [22] as the code

[1] https://electrosource.com/wp-content/uploads/2022/09/rep-ossra-20221.pdf
[2] https://cwe.mitre.org/data/definitions/120.html

encoder. For the graph modality, MVulD+ first analyzes the function to construct the corresponding graph (i.e., CPG) and then employs graph-based models as a multi-modal graph encoder. Meanwhile, the CPG is transformed into an image, and MVulD+ uses a transformer-based computer vision model (i.e., SwinV2 [37]) as the image encoder to capture global information from the image modality. We also employ local encoding, which integrates information related to each statement (i.e., the node in the graph and image). Each statement in the image is matched with its corresponding code and graph. For instance, we pair the embedding of a statement obtained from the code and graph encoders with the local image embedding corresponding to that statement. Additionally, we use degree centrality values to weight all local embeddings, resulting in the final local feature. Our local encoder enables the model to learn a more effective representation of the code and facilitates the alignment and fusion of information across different modalities (i.e., aligning the code, graph, and image modalities at the statement-level). Finally, the various code representations from the four dimensions are aligned and fused to generate the multi-modal features of the function. These features are then used to build the final classification model.

To evaluate the effectiveness of different modalities, we perform an ablation study (cf. Section 6.1) to quantify the performance improvements obtained by the addition of each modality. Following that, we validate the ability of the multi-modal graph presentation (cf. Section 6.2) to learn from the multi-modal information of the code and establish relationships between them. Furthermore, to evaluate the effectiveness of MVulD+, we conduct a large-scale study (cf. Section 6.3) by comparing our approach with nine state-of-the-art baselines using five widely used performance measures. The experimental results indicate that our approach achieves better performance than the baselines. In particular, MVulD+ obtains 0.307 of F1-score, 0.908 of Accuracy, 0.225 of Precision, and 0.241 of PR-AUC, which improves DL-based baselines by 24.3%-125.7%, 5.2%-31.4%, 40.6%-192.2%, and 22.3%-186.9%, respectively. Finally, to explore the effectiveness of MVulD+ in practical vulnerability detection scenarios, we show the detection capabilities of MVulD+ on the Top-25 Most Dangerous CWEs in Big-Vul dataset and real-world vulnerabilities (cf. Section 6.4). In summary, this paper makes the following contributions:

- We propose an effective multi-modal-based function-level vulnerability detection approach named MVulD+. To the best of our knowledge, MVulD+ is the first to fuse the image modality with text and graph modalities, where the generated image fully leverages the structural information of the function.
- We design a novel multi-modal graph-based module that integrates textual and visual features jointly with multi-modal alignment to perform reasoning.
- We conduct large-scale experiments on 25,816 functions and experimental results demonstrate that MVulD+ outperforms the state-of-the-art baselines.
- To support the open science community, we publish the studied dataset and source code of our model with supporting scripts on GitHub (https://github.com/vinci-grape/MVulD), which provides a ready-to-use implementation of our model for future research about comparison.

Paper Organization. The organization of this paper is as follows: Section 2 provides the relevant background information. Section 3 presents a vulnerable sample to illustrate the motivation of our paper. Section 4 details the proposed MVulD+. Section 5 presents the experiment settings including the studied dataset, the considered baselines, and the implementation details. Section 6 presents the experimental results. Section 7 discusses the findings and addresses potential threats to validity. Section 8 describes the related work. Finally, Section 9 concludes our work and outlines future research directions.

2 BACKGROUND

Recently, many novel approaches [5, 6, 10, 12, 30, 32–34, 40, 48, 70] have been proposed for software vulnerability detection, utilizing various program representation techniques such as feature-based, sequence-based, tree-based, and graph-based code representations. These approaches aim to capture the deeper semantics behind the textual

code and have achieved promising results. Meanwhile, Siow et al. [53] conducted a comprehensive code-related study and found that ❶ **different modalities have varying representation abilities of a function.** We also experiment by building a model with only one modality on the Big-Vul [13] dataset and the results are shown in Table 1. According to the results, we find that different modalities of a function exhibit varying effectiveness in detecting vulnerabilities.

Table 1. Vulnerability detection results of various models built on different modalities

Methods	F1-score	Accuracy	Recall	Precision	PR-AUC
Text (Func)	0.200	0.765	0.695	0.117	0.182
AST	0.177	0.732	0.683	0.102	0.136
CFG	0.173	0.736	0.654	0.099	0.125
PDG	0.180	0.751	0.649	0.105	0.157
Image	0.124	0.674	0.548	0.070	0.077

Besides, many vulnerabilities (e.g., memory leak) are too subtle to be spotted, suggesting that ❷ **adopting a joint representation of the composite code semantics [66, 70] is a good choice for software vulnerability detection**. For example, previous work [66] found that a model built solely with ASTs could only identify insecure arguments, but this could be enhanced by combining ASTs with CFGs, enabling the detection of additional vulnerability types, such as resource leaks and use-after-free errors. By further incorporating the three graph-based modalities including AST, CFG, and PDG, the model can be further improved to discover the majority of vulnerability types.

In addition to converting the source code of the function into graphs, there has been an attempt [63] to convert the source code into a vector representation, and subsequently transform the vector into an image. The experimental results indicate that ❸ **computer vision techniques also demonstrate strong capabilities in performing software vulnerability detection.**

Although prior works have shown promising results, there are still limitations in fully leveraging the different modalities of a function for vulnerability detection. We conduct a thorough analysis of the results obtained in Table 1 and draw a Venn diagram (Fig. 1) to illustrate the performance differences among various modalities concerning specific vulnerability types. From Table 1, we can conclude that different code representations exhibit varying capacities for vulnerability detection. The top half of Fig. 1 displays the overlapping results among models built with graph, text, or image representations, while the bottom half provides details on graphs, such as AST, CFG, and PDG. The illustrated results indicate that, although models built with single modality information can correctly detect many instances in common, there are also specific vulnerabilities that can only be identified by models utilizing particular modalities. For example, for CWE-20, three models (graph, text, and image) can jointly identify 14 vulnerabilities, while one vulnerability can only be detected exclusively by either the image model or the graph model. Therefore, we find that ❹ **different modalities have complementarity in vulnerability detection**.

3 MOTIVATION

3.1 A Motivation Example

Fig. 2 shows a *bypass of CSRF (Cross Site Request Forgery) protection* vulnerability *CVE-2016-7401* in *Django*, which is disclosed during the interaction between Google Analytics and Django's cookie parsing. The top-left sub-figure illustrates the source code of the function. According to the source code, we can observe that there is no return operation in the *else* branch (line 6) after the call of statement *follow_dotdot(nd)* (line 7), which ignores the result of *follow_dotdot()* and always returns a zero (line 9). Such a vulnerability will allow an attacker to set arbitrary

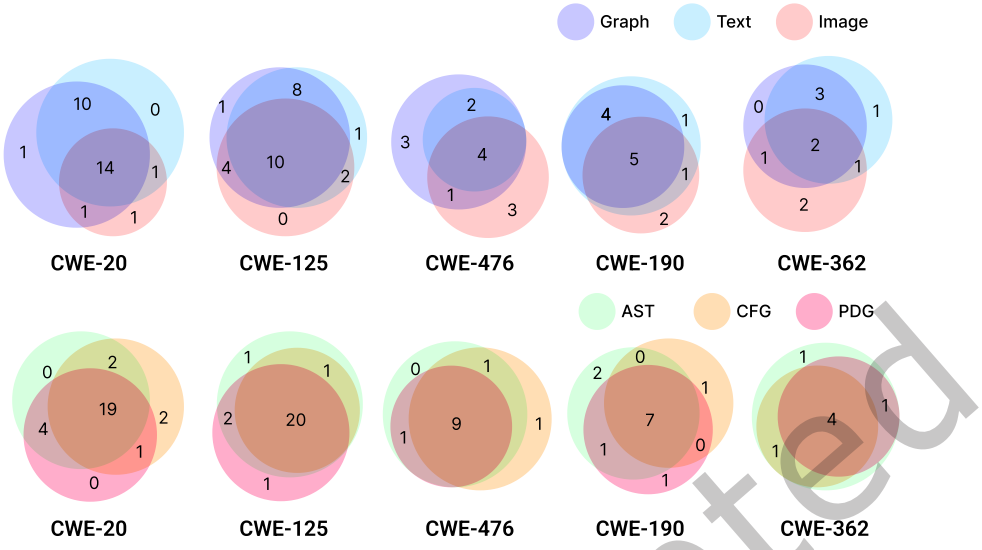


Fig. 1. The complementary analysis of various models built on different modalities

cookies leading to a bypass of CSRF protection. To fix this vulnerability, it is necessary to conduct suitable operations on the returned results of *follow_dotdot()*.

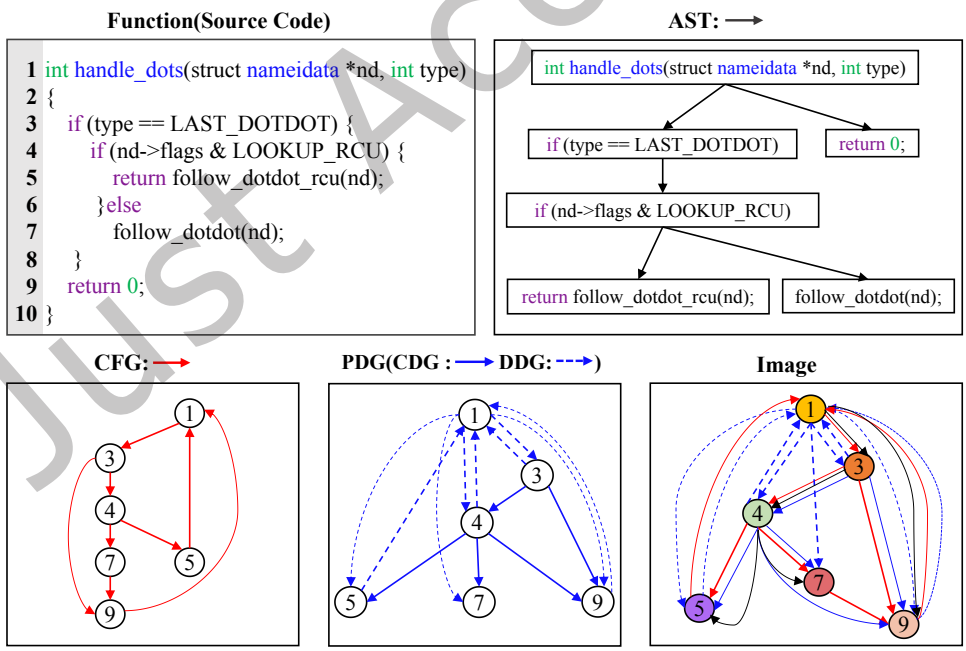


Fig. 2. The multi-modal information of a vulnerability from CVE-2016-7401

Observation 1. A function can be represented in various ways, including text (source code), abstract syntax tree (AST), control flow graph (CFG), program dependence graph (PDG), and image. As shown in Fig. 2, we illustrate the vulnerable function using different representations, which are referred to as the various modalities of a function [17, 53, 58, 63]. The AST is an ordered tree where inner nodes represent operators and leaf nodes correspond to operands (e.g., constants or identifiers). ASTs are widely used for identifying semantically similar codes. The CFG explicitly describes the execution order of code statements, with edges indicating control flow. The PDG explicitly represents dependencies among statements, which is constructed using two types of edges: data dependency edges reflecting the influence of one variable on another and control dependency edges corresponding to the impact of predicates on the values of variables. The image modality represents the edges and nodes that integrate the AST, CFG, and PDG properties into a unified data structure.

Observation 2. Although different modalities offer varying representation capabilities of a function, as highlighted in previous findings [53], the information from these modalities has not been effectively utilized, and the information across modalities tends to be relatively independent. This is particularly evident in the case of image modalities, where existing approaches [7, 63] focus solely on the importance of images, neglecting the need for feature alignment with other modalities, such as text and graph representations. Such independence limits the approach's ability to comprehensively understand the function's semantics, thereby affecting the effectiveness of vulnerability detection.

3.2 Key Ideas

Based on the above observations, we propose a multi-modal approach, which fuses and aligns the multiple modalities of functions to detect vulnerability at the function-level.

(1) Multi-Modal-based Vulnerability Detection. Building a model that utilizes these complementary modalities of a function can enhance its effectiveness in vulnerability detection. By leveraging the strengths of each modality, such as the syntactic structure provided by the AST, CFG, and PDG, this approach aims to create a more holistic understanding of the function's behavior. This comprehensive view facilitates improved identification of potential vulnerabilities, as it enables the model to recognize patterns and anomalies that may not be evident when analyzing a single modality in isolation.

(2) Multi-Modal Fusion and Alignment. To address the issue of independent information across various modalities, we propose a multi-modal fusion and alignment approach that integrates the distinct features of each representation into a unified framework. This process involves fusing and aligning the representations to ensure that complementary information is effectively captured, allowing for a richer understanding of the function. By fusing the representations derived from text, graph, and image modalities, this approach enhances the semantic interpretation of the function's behavior, thereby improving the overall vulnerability detection capabilities. This alignment facilitates deeper semantic analysis and enables the model to discern complex relationships among code representations, ultimately leading to more accurate vulnerability detection.

4 APPROACH

As illustrated in Fig. 3, MVulD+ consists of four main phases: ❶ graph extraction aims to obtain the structure representation of the function, ❷ image generation transforms the structural graph into a graphical representation, ❸ multi-modal feature fusion builds the relationship of various modalities to obtain enriched code representations, and ❹ classification is applied to detect whether a function is vulnerable. Details of MVulD+ are presented in the following subsections.

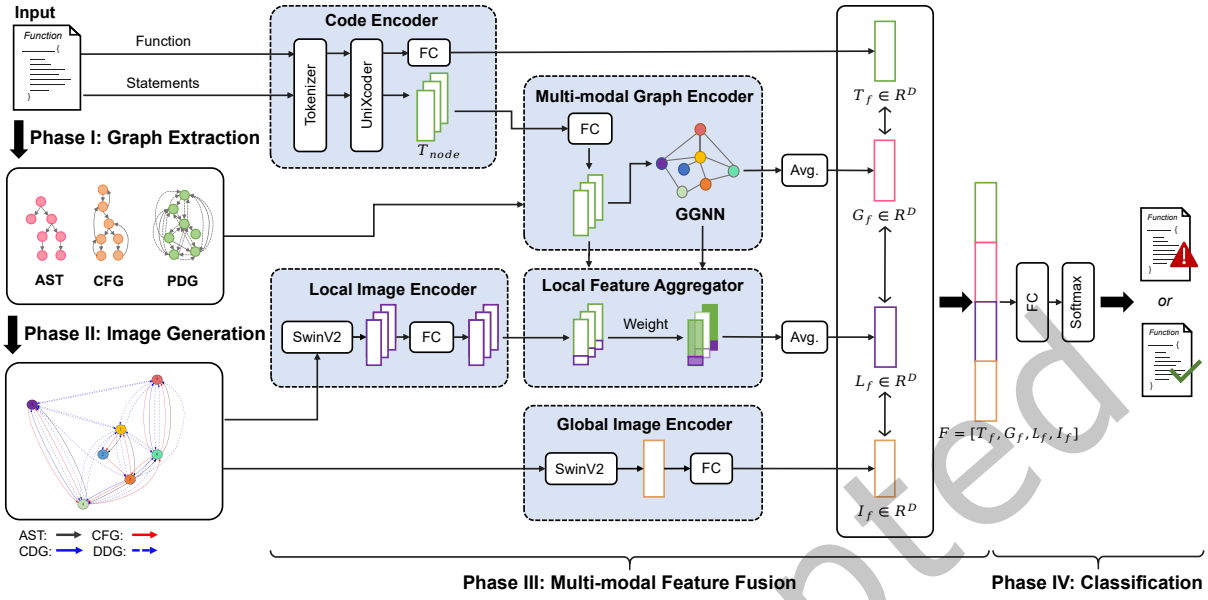


Fig. 3. An overview architecture of MVulD+, which fuses the multiple modalities of a function including text, graph, and image representations to detect function-level vulnerability

4.1 Graph Extraction

During the graph extraction phase, we extract the code property graph (CPG) [66] 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. Following previous works [43, 53, 63, 69], we utilize the *Joern* program [66] to extract all graphs. Specifically, when parsing the source code of a function with *Joern*, we obtain a CPG composed of nodes that represent statements and edges that indicate the flow of information among those statements.

4.2 Image Generation

During the image generation phase, we employ the tool named *Graphviz* [20] to transform the CPG generated by *Joern* to an image. Each statement node of the function is represented by a numbered circular node in various colors, with the number indicating the corresponding line number. To effectively differentiate the three structural types represented in the image, we use distinct colors and line styles for the connections between statement nodes. This visualization allows developers to easily identify the structures (AST, CFG, and PDG) and the corresponding statement nodes.

4.3 Multi-modal Feature Fusion

To obtain representations of different modalities, we design four encoders: ① global image encoder, ② code encoder, ③ multi-modal graph encoder, and ④ local encoder. These encoders extract four types of code features (i.e., global image features, text features, graph features, and local features). MVulD+ then aligns and fuses these features to produce enriched multi-modal code representations. The details of each encoder and fusion method are introduced as follows.

4.3.1 Global Image Encoder. To extract global image features, MVulD+ utilizes the well-known computer vision (CV) model named SwinV2 [37], which is pre-trained on ImageNet [11] and demonstrates strong performance across various tasks (e.g., image classification [35, 64], object detection [57, 61], and semantic segmentation [27, 65]). Specifically, for a given image I, SwinV2 processes it as input and produces a vectorized feature representation. The dimension of the output feature vector is 1,536, which is then projected to 768 dimensions through a fully connected layer, resulting in the final global image features denoted as $I_f \in \mathbb{R}^{1\times D}$, where D equals 768.

4.3.2 Code Encoder. Apart from the structure information of a function, the semantic information of the source code is also significant. UniXcoder, introduced by Guo et al. [22], is a unified pre-trained model that integrates semantic and syntactic information from both code comments and ASTs. We utilize UniXcoder as the code encoder to embed code features at both the function and statement levels. Specifically, MVulD+ first divides the source code of a function into individual statements. The entire function and these individual statements (code lines) are then tokenized using UniXcoder's pre-trained Byte Pair Encoding tokenizer [22]. Following that, UniXcoder transforms the input function to vectors at both the function and statement levels. This process results in a total of $n + 1$ code representations: one embedding for the overall function and n embeddings for each line of code.

Similar to obtaining the final image representation of a function, MVulD+ further projects the obtained function-level code feature and statement-level code feature to 768 dimensions through a fully connected layer to get the final encoded code feature. The final function-level code feature can be denoted as $T_f \in \mathbb{R}^{1\times D}$, while the statement-level code features $T_{node} = \{t_1, t_2, ..., t_n\}$ will serve as the initial node embedding of the graph (i.e., CPG).

4.3.3 Multi-modal Graph Encoder. To enhance graph feature extraction, MVulD+ constructs a multi-modal graph encoder primarily based on a gated graph recurrent network (GGNN) [28]. We begin by inputting the enriched statement feature embeddings generated by the code encoder into the GGNN. Between these, we apply GGNN to the CPG. Given that CPG is a directed graph, GGNN is capable of flexibly handling asymmetric relationships and directionalities between nodes. We take n statement embeddings (i.e., T_{node}) as input, and the GGNN model is adopted to optimize them on the basic CPG. Specifically, GGNN utilizes the adjacent matrix to aggregate the neighborhood information of each node and embeds edge information between neighboring statements incrementally to propagate the information. After GGNN, we can obtain the enhanced node embedding: $G_{node} = \{g_1, g_2, ..., g_n\}$. Global structure information is attached to each line of code (i.e., node). Therefore, the final graph feature representation G_f is obtained by averaging the information from all nodes, which can be represented as: $G_f = \frac{1}{n}\sum_{i=1}^{n} g_i$.

4.3.4 Local Encoder. To establish correspondence between three features (i.e., text feature, graph feature, and image feature), we have implemented a local encoder. Initially, we obtain the in-degree and out-degree information for each node in the CPG extracted by *Joern*. The degree centrality [14] of a node represents the score of its connected nodes and has been used in many different areas (e.g., biological network [25] and transportation network [21]). Consequently, we calculate the degree of centrality for each node. The degree centrality values are normalized by dividing them by the maximum possible degree in the graph, given by the formula $x_i = \frac{deg(i)}{n-1}$, where n is the number of nodes in the graph. Following this, we match each node in the image with its corresponding code, for example, pairing the vector of the first line statement obtained by the code encoder with the local image embedding corresponding to that statement in the image, forming the first local embedding: $l_1 = l_{t1} + l_{i1}$, where l_{t1} represents local text feature, and l_{i1} represents local image feature. Finally, we use the degree centrality values to weight all the local embeddings to obtain the final local feature: $L_f = \sum_{i=1}^{n} (l_i \cdot x_i)$. Our designed local encoder not only enables the model to learn how to better represent code but also can align and fuse information from different modalities (i.e., aligning the text of the code, graph, and image at the statement-level).

4.3.5 Multi-modal Aligner. We use contrastive learning [16, 46, 62] to align information from different modalities. The training objective is to fine-tune the network so that the distance between different modalities should be as close as possible, which is illustrated below:

$$max(||T_f - G_f|| + ||T_f - I_f|| + ||T_f - L_f||, 0) \tag{1}$$

We employ the kl-divergence loss used in R_Drop [62] to quantify the differences between modalities. The final loss function consists of both classification cross-entropy loss and kl-divergence loss, which can be described by the following equation:

$$\mathcal{L}_{ce} = -\sum_{i} y_i \log(\hat{y}_i) \tag{2}$$

$$\mathcal{L} = \mathcal{L}_{ce} + \alpha \cdot \mathcal{L}_{kl} \tag{3}$$

Before making classifications, MVulD+ concatenates the four obtained features T_f, G_f, I_f, and L_f from respective encoder together to obtain the multi-modal code features $F = [T_f, G_f, I_f, L_f]$.

4.4 Classification

The target of MVulD+ is to detect whether a function is vulnerable, which is a typical binary classification task. To achieve this, we pass the final representation F through a fully connected layer (the final classification layer) and then through a $softmax$ layer to obtain the probability distribution of class labels for the input function. During the training phase, we use cross-entropy loss to guide the optimization process of MVulD+ by comparing the predicted probabilities with the ground-truth labels. The objective is to minimize the cross-entropy loss, thereby bringing the predicted results closer to the actual labels. Additionally, kl-divergence loss is employed to align different modalities during training.

5 EXPERIMENTAL SETTINGS

In this section, we first present an introduction to our studied datasets, then we introduce the state-of-the-art baselines. Finally, we describe the implementation details, including hyperparameters and training settings.

5.1 Datasets

In our experiment, we use the Big-Vul dataset provided by Fan et al. [13], which is one of the largest vulnerability datasets collected from 348 open-source GitHub projects spanning 91 different vulnerability types. All vulnerabilities in this dataset are linked to the public Common Vulnerabilities and Exposures (CVE) database. Meanwhile, Fan et al. spent a substantial amount of manual resources to ensure the quality of the dataset and ended up with 188,636 functions, which contains 10,900 vulnerable samples and 177,736 non-vulnerable samples. Its large diversity in projects and vulnerability types allows us to conduct a deep analysis of the characteristics of different vulnerabilities.

Since MVulD+ focuses on detecting vulnerability at the function-level, we perform three filtering steps on the original dataset to obtain valid functions, and the filtering results of each step of the dataset are displayed in Table 2.

Step-1: To clean and normalize the dataset, we start by removing empty lines, leading and trailing spaces in each line, as well as comments from the source code. Then we remove the improperly truncated functions, which may fail when parsing them. Since there are two versions of a specific function included in the dataset: function before (the function before the vulnerability is fixed) and function after (the function after the vulnerability is

fixed), we also remove vulnerable functions whose code remains unchanged between the two versions. In this step, we obtain the dataset containing 183,750 functions.

Step-2: To extract the graph representations (i.e., CPG) for all functions in the datasets, we leverage a widely-used code analysis tool, namely *Joern* [66]. However, due to some inner compile errors and exceptions in *Joern*, we can only obtain CPGs for part of the functions. Therefore, we filter out those functions without control or data dependency edges and finally collect 101,568 functions in this step.

Step-3: We also conduct an analysis on the size of functions and find that the majority (i.e., 95%) of the functions have less than 100 lines of code (LOC), while the remaining ones have varying function sizes ranging from 101 LOCs to 6,434 LOCs. Considering the trade-off between detection performance and computational efficiency (cf. Section 7.2.1), we focus our research on functions (LOCs<=100). In this step, we finally obtain 96,529 functions.

Data Splitting. After the filtering steps, we randomly split the final dataset (LOCs<=100) into three parts: training, validation, and testing sets, which account for 80%, 10%, and 10%, respectively. For the training dataset, we undersample the non-vulnerable functions to produce an approximately balanced dataset at the function level, while the validation and testing dataset remains in the original imbalanced ratio.

Though vulnerabilities contain different types, we do not distribute each type proportionally across the training, validation, and test sets for the following two reasons: (1) Not all types of vulnerabilities have enough instances for dataset partitioning (i.e., 80%:10%:10%) and some vulnerabilities have only one or two instances. For example, there are only two instances of CWE-90, one for CWE-290 and one for CWE-352. (2) Existing studies [15, 23, 30, 46] all treat the dataset as a binary classification dataset (i.e., vulnerable ones and non-vulnerable ones) with the consideration of inherent imbalance in vulnerability datasets. For a better comparison, we follow the strategy used in existing literature.

Table 2. 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%

5.2 Baseline Models

To comprehensively evaluate the performance of MVulD+ with prior works, in total, we consider nine state-of-the-art approaches described as follows:

- **UniXcoder:** Guo et al. [22] propose a unified pre-trained model to support code-related tasks. UniXcoder leverages cross-modal contents (i.e., AST and code comment) and builds a multi-layer Transformer framework to learn code representation from different knowledge sources.
- **IVDetect:** Li et al. [30] propose an interpretable vulnerability detection approach. IVDetect leverages feature-attention graph convolutional network (FA-GCN) [52] for vulnerability detection using five types of feature representations (i.e., sequence of sub-tokens, AST, variable names and types, data dependency context and control dependency context) of the source code.
- **Devign:** Zhou et.al. [70] propose Devign, a DL-based approach that applies gated graph neural network [29] on a rich set of code semantic representations (i.e., AST, CFG, PDG, and code sequences) for function-level vulnerability detection.

- **Reveal:** Chakraborty et.al. [6] propose a novel approach named Reveal. It uses a code property graph (CPG) to extract graph-based features and then leverages a gated graph neural network to learn the graph properties of source code.
- **LineVD**: Hin et.al. [23] propose a novel deep learning framework, LineVD. It redefines the task of statement-level vulnerability detection as a node classification problem. It harnesses graph neural networks to capture control and data dependencies among statements and employs a transformer-based model to encode the raw source code tokens. Notably, LineVD effectively enhances prediction performance by addressing conflicting outputs arising from function-level and statement-level information.
- **LineVul:** Fu et al. [15] propose LineVul, a Transformer-based line-level vulnerability prediction approach. It leverages BERT architecture with self-attention layers which can capture long-term dependencies within a long sequence. Moreover, it adopts the attention mechanism of BERT architecture to locate the vulnerable lines for finer-grained detection.
- **VulCNN:** Wu et al. [63] propose VulCNN, which employs Joern and Sent2Vec for producing PDGs from source code. It calculates degree centrality, katz centrality, and closeness centrality on the PDG. Utilizing these three centrality measures, VulCNN generates three vectors, considering them as the three channels of the image. Subsequently, it utilizes a CNN to accomplish the classification task.
- **VulDeeLocator:** Li et al. [31] propose VulDeeLocator, a vulnerability detector with high detection capability and precise locating. It uses slicing criteria from a user-specified API call to form a subgraph on a program dependence graph (PDG). The subgraph consolidates multiple paths, leveraging intermediate code for semantic information. The model employs granularity refinement to pinpoint vulnerability locations accurately.
- **Cppcheck:** Daniel [39] proposes Cppcheck, a tool for static C/C++ code analysis Overview Cppcheck is an analysis tool for C/C++ code. Unlike C/C++ compilers and many other analysis tools, it doesn't detect syntax errors. Cppcheck only detects the types of bugs that the compilers normally fail to detect.

5.3 Implementation Details

Our evaluation is conducted on a 32-core workstation equipped with an Intel(R) Xeon(R) Platinum 8358P CPU @ 2.60GHz, 768GB RAM, and 4×NVIDIA GeForce RTX 2080 GPU, running Ubuntu 20.04.6 LTS. The RTX 2080 features 2944 CUDA cores, 8 GB of GDDR6 memory running at 14 Gbps, and supports Tensor Cores for accelerated deep learning operations. This setup is powered by two key Python libraries: Transformers [60] and Pytorch [47]. To extract the semantic information of the source code, we download the UniXcoder tokenizer and UniXcoder model pre-trained by Guo et al. [22]. As for the global image features, we employ a hierarchical vision transformer model called SwinV2 [37], which is pre-trained on ImageNet and further fine-tuned on our training dataset with the input image size set to 256×256 considering the limited GPU memory. The best SwinV2 was selected according to the optimal F1-score on the validation set.

To optimize MVulD+, we choose Exponential Linear Units (ELU) [8] as the non-linear activation function to avoid the problem of vanishing gradients and dying neurons. It can shorten the training time and get higher accuracy as compared to Rectified Linear Unit (ReLU) [19] and its variants. During training, we employ the AdamW optimizer [38], which is widely adopted to fine-tune Transformer-based models. We also use a cosine decay learning rate scheduler with 5 epochs of linear warm-up to update the model and minimize the loss function. The initial learning rate and the weight decay are set to 5e-5 and 0.005, respectively. We train MVulD+ for at most 500 epochs, the best MVulD+ was selected based on the optimal F1-score against the validation set with the early stopping epoch set to 50.

6 EXPERIMENTAL RESULTS

We construct our research questions and categorize them into four domains: the influence of modalities on MVulD+, the impact of code structures on MVulD+, the effectiveness and efficiency of MVulD+, and the ability of MVulD+ to detect real-world vulnerabilities:

- RQ-1: How do different modalities affect the performance of MVulD+?
- RQ-2: How do different types of graph representations affect vulnerability detection performance?
- RQ-3: How does MVulD+ perform compared to baselines for function-level vulnerability detection?
- RQ-4: How does MVulD+ perform in detecting the Top-25 Most Dangerous CWEs in Big-Vul and real-world vulnerabilities?

Specifically, both RQ-1 and RQ-2 focus on investigating the influence of both different code modalities and different types of graph representations on MVulD+, showcasing the complementary effects of various modalities and graph representations and revealing the role of multi-modal alignment. Following that, RQ-3 aims to evaluate MVulD+'s capability in addressing vulnerability detection tasks when fusing multi-modalities of codes, including a comparison with baselines in terms of effectiveness and efficiency. The goal of RQ-4 is to explore MVulD+'s capability in real-world vulnerability detection, highlighting MVulD+'s exceptional abilities in handling unknown vulnerabilities and specific vulnerability types.

6.1 [RQ-1]: Effectiveness of Different Modalities

Objective: Source code has rich program semantics and can be learned by different code representation techniques. The varying capacity of different code features to represent the function is also proved by Siow et al. [53]. Since a comprehensive and rich code representation is fundamental for code intelligence tasks (e.g., vulnerability detection), our MVulD+ integrates various program semantics from different dimensions, including text, image, and graph features. In RQ-1, we conduct an ablation study to assess the impact of different code features, aiming to understand the performance gains achieved by fusing each modality.

Experiment Design: Different code representations (i.e., text representation, graph representation, and image representation) can be obtained using four encoders and then fused to obtain the final multi-modal code features. Therefore, we consider the image feature I_f, graph feature G_f, text feature T_f, and local feature L_f to further study their respective contributions to vulnerability detection tasks. For example, to evaluate the effectiveness of the image feature, we remove the image feature vector obtained by the image encoder and retrain our MVulD+ model using the remaining three features (i.e., text feature, graph feature, and local feature with no image feature). We then compare MVulD+ (No Graph Feature) with MVulD+ (All) to observe the performance improvements brought by the image feature.

Moreover, we compare MVulD+ with models built on one single modality to investigate the detection performance of different modalities. For instance, we build MVulD+ (Image Feature) using only the image feature to observe the performance difference. Additionally, we utilize five performance metrics (i.e., Accuracy, F1-score, Recall, Precision, and PR-AUC) to investigate the impacts of different features on vulnerability detection. Furthermore, the experimental setup is consistent with that in RQ-3 (cf. Section 6.3).

Results: We discuss the results from the perspectives of the ablation study and the mono-modality study, respectively.

Ablation Study. Table 3 illustrates the effectiveness of different modalities and the better performance is highlighted in bold. Firstly, we observe that the MVulD+ (All) with comprehensive multi-modal information achieves improved performance in terms of F1-score (0.307), Accuracy (0.908), Precision (0.225), and PR-AUC (0.241). Besides, removing the global image feature from MVulD+ (All) results in a decrease of 18.9%, 7.6%, 31.1%, and 8.3% in F1-score, Accuracy, Precision, and PR-AUC, respectively, indicating that the image feature enhances the model's learning capability. We also find that the text feature significantly contributes to MVulD+

by comparing the second line and the bottom line. Specifically, excluding the text feature (i.e., MVulD+ (No Text Feature)) results in a decrease in F1-score (0.196), Accuracy (0.766), Precision (0.115), and PR-AUC (0.150) by 36.2%, 15.6%, 48.9%, and 37.8%, respectively. A further decrease occurs when the graph representation G_f is removed. Comparing MVulD+ (All) with MVulD+ (No Graph Feature), we observe reductions from 0.307 to 0.263, 0.908 to 0.894, 0.225 to 0.186, and 0.241 to 0.213 for F1-score, Accuracy, Precision, and PR-AUC, corresponding to 14.3%, 1.5%, 17.3%, and 11.6% decreases, respectively. Furthermore, removing the local feature from MVulD+ (All) results in a decrease of 13.7%, 4.4%, 23.1%, and 6.2% in F1-score, Accuracy, Precision, and PR-AUC, highlighting the effect of the local feature in multi-modal alignment.

Table 3. The vulnerability detection performance of MVulD+ compared with other types of variants

Features	Modality	F1-score	Accuracy	Recall	Precision	PR-AUC	# TP	# FP
G_f + T_f + L_f	MVulD+ **(No Image Feature)**	0.249	0.839	0.634	0.155	0.221	258	1,409
I_f + G_f + L_f	MVulD+ **(No Text Feature)**	0.196	0.766	**0.678**	0.115	0.150	**276**	2,132
I_f + T_f + L_f	MVulD+ **(No Graph Feature)**	0.263	0.894	0.450	0.186	0.213	183	802
I_f + G_f + T_f	MVulD+ **(No Local Feature)**	0.265	0.868	0.565	0.173	0.226	230	1,100
I_f + G_f + T_f + L_f	MVulD+ **(All)**	**0.307**	**0.908**	0.486	**0.225**	**0.241**	198	**683**

Note: I_f: Image feature; T_f: Text feature; G_f: Graph feature; L_f: Local feature.

From the results in Table 3, we can conclude that the absence of any modality influences the model's performance: (1) The lack of text modality hinders the model's ability to understand the semantics of the code, leading to lower Precision and a high count of false positives; (2) The lack of the graph and image modalities impedes the model's capability to capture the structural aspects of the code, resulting in reduced Precision and a higher incidence of false positives. However, MVulD+ (All) does not achieve the best Recall, falling below that of MVulD+ (No Image Feature), MVulD+ (No Text Feature), and MVulD+ (No Local Feature). To analyze the reasons in detail, we present the number of true positives (TPs) and false positives (FPs). Although MVulD+ (All) identifies fewer TPs compared to other variants (except MVulD+ (No Graph Feature)), it also has fewer FPs. This indicates that MVulD+ (All) is more cautious in identifying vulnerabilities and is less prone to misclassify negative instances as positive ones. In contrast, MVulD+ (No Image Feature), MVulD+ (No Text Feature), and MVulD+ (No Local Feature) achieve higher TPs but also relatively high FPs, indicating a greater likelihood of misclassifying non-vulnerable code as vulnerable. When removing the graph feature, the TP and FP of MVulD+ (No Graph Feature) show changes compared to MVulD+ (All). Specifically, the TP drops to a lower value (i.e., 183), while FP increases by 119. This indicates that the absence of graph information negatively impacts the capacity of vulnerability detection. The increase in FP after removing the modalities further emphasizes the role of the different modalities, with the text modality providing code semantic information, while the graph and image modalities provide code structural information.

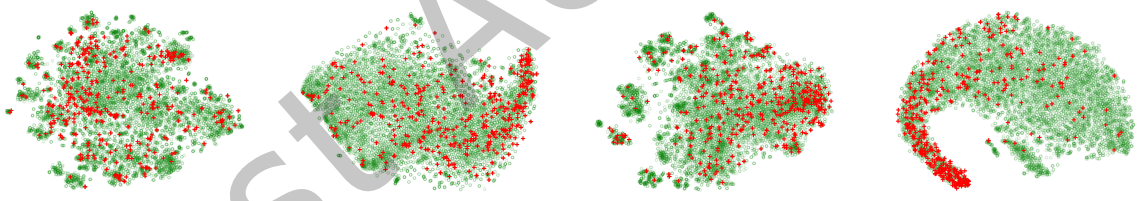
Mono-modality Study. Table 4 shows the performance of MVulD+ built on a single modality and better performances are highlighted in bold. For better comparison, we also include the performance of VulCNN. Based on the table, we can make the following observations: (1) When using only the image modality, MVulD+ outperforms VulCNN. We attribute this to our images containing richer graph information (i.e., AST, CFG, and PDG), while VulCNN relies solely on PDG, which lacks comprehensive details. Additionally, we utilize the advanced SwinV2 model to better capture the semantics within the image, whereas VulCNN is based on CNN. (2) Different modalities demonstrate varying vulnerability detection performance, with the text modality showing better results in terms of F1-score (0.205), Accuracy (0.783), and PR-AUC (0.191). (3) The aggregated MVulD+ fusing all modalities achieves improved results compared to the single-modality variants and VulCNN across the F1-score, Accuracy, and PR-AUC.

Table 4. The vulnerability detection performance of MVulD+ compared with single-modality variants and VulCNN

Model Type	F1-score	Accuracy	PR-AUC
VulCNN **(Image Feature)**	0.136	0.704	0.084
MVulD+ **(Image Feature)**	0.153	0.806	0.091
MVulD+ **(Text Feature)**	0.205	0.783	0.191
MVulD+ **(Graph Feature)**	0.179	0.814	0.110
MVulD+ **(All)**	**0.307**	**0.908**	**0.241**

Furthermore, we aim to understand why fusing the three modalities would be a better design than using individual modalities. DL-based vulnerability detection approaches have a strong ability to learn a feature representation to distinguish vulnerable functions and non-vulnerable ones. Therefore, the effectiveness of the models in detecting vulnerabilities largely depends on the separability of the feature representations for these two types of functions (i.e., vulnerable and non-vulnerable). The greater the separability, the easier it is for a model to distinguish between them. To illustrate this separability, we employ the widely used *t*-SNE [54], which is a visualization method that maps high-dimensional features into two-dimensional features. For a better illustration, we extract the hidden vectors before making the final binary classification decision as the high-dimensional features.

Fig. 4 illustrates the visualization of separating vulnerable functions from non-vulnerable functions. From the visualization results (Fig. 4(a)–(c)), we observe that the majority of the functions are mixed, and the boundaries between different function types are not clearly defined. This indicates the challenges faced by single-modality variants in establishing a clear decision boundary. In contrast, Fig.4(d) demonstrates the separability of MVulD+ with the multi-modal feature. We can observe that MVulD+ with multi-modal features performs more effectively in distinguishing vulnerable functions from non-vulnerable ones. This suggests that through multi-modal alignment and fusion, MVulD+ can learn better semantic embedding of functions.



(a) Text Feature (b) Graph Feature (c) Image Feature (d) All Feature

Fig. 4. Visualization of the separation between vulnerable (denoted by +) and non-vulnerable (denoted by ◯) functions

We conduct a deep analysis of the results obtained in Table 4 and visually represent the performance distinctions among various single-modality variants through a Venn diagram (Fig. 5), highlighting specific vulnerability types. From Table 4, we find that different variants show varying capacities for vulnerable detection. Fig. 5 illustrates the overlapping results among the MVulD+ built with text, graph, or image modalities. The results show that while these models utilizing single-modality information can correctly detect many instances in common, there are also some vulnerability instances that only a specific modality can detect. For instance, regarding CWE-119, all three variants (Text, Graph, and Image) are capable of detecting 19 vulnerabilities. However, 2 vulnerabilities in CWE-119 can only be detected by MVulD+ (Graph Feature) and MVulD+ (Image Feature), while MVulD+ (Text Feature) fails to detect them. This observation highlights the complementarity of different modalities in vulnerability detection.

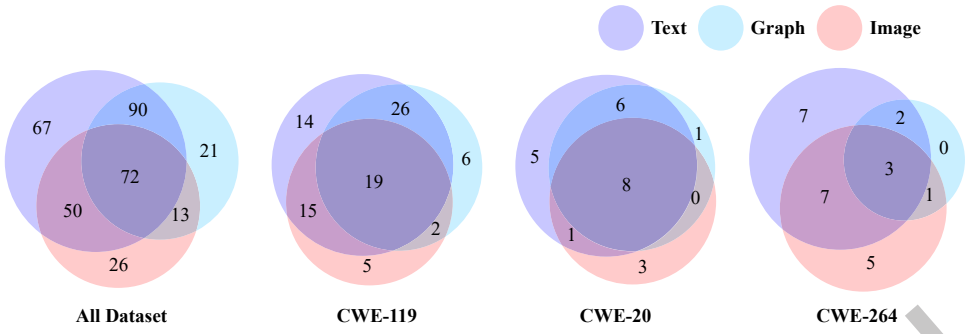


Fig. 5. The complementary analysis of single-modality variants of MVulD+

We also provide a specific example that can only be detected by MVulD+ (All). By incorporating this example, we aim to enhance the credibility of the fusion of different modalities and more convincingly demonstrate its practical value. As shown on the left side of Fig. 6, this example represents a vulnerability of CWE category CWE-20. The aspath_put function in bgpd/bgp_aspath.c in Quagga project allows remote attackers to cause a denial of service (session drop) via BGP UPDATE messages. This occurs because the AS_PATH size calculation for long paths counts certain bytes twice, resulting in the construction of an invalid message. Specifically, the vulnerable line within the function is Line 20: bytes += ASSEGMENT_SIZE (written, use32bit);.

To better explain why MVulD+ can detect this vulnerability, we follow prior works [15, 26, 59] and employ attention mechanisms to investigate the content that the model focuses on during the recognition process. We find that MVulD+ pays more attention to the lines of code where vulnerabilities are located, as well as the surrounding lines (i.e., near line 20). Since MVulD+'s image modality is implemented by the Swim v2 model based on the transformer architecture, we can still use attention to explore the image information the model focuses on when identifying this vulnerability. The left part of Fig. 7 shows the image generated from the CPG of the aspath_put function, where CALL_20 represents the node at Line 20 (i.e., vulnerability line). In the right part of Fig.7, we present the attention of MVulD+ when recognizing this vulnerability. As shown in the figure, the bottom of the image appears darker, while other areas are lighter, indicating that the model pays more attention to this region (i.e., the location of the vulnerability, as well as the edges and nodes connecting to the "CALL_20" node). This observation indicates that MVulD+ can learn effective vulnerability patterns from the image modality composed of AST, CFG, and PDG. Since the image modality integrates the structures of various graphs, the fused images fully utilize the structural information within functions. Consequently, fusing and aligning the graph modality and image modality provides the model with additional structural insights.

✍ **RQ-1** ► *Different modalities capture various program semantics and contribute differently to vulnerability detection. Among them, the text modality T_f provides greater contributions than both the graph modality G_f and the image modality I_f. However, fusing all these modalities can achieve better performance.* ◄

6.2 [RQ-2]: Effectiveness of different Graph Representations

Objective: In our study, we extract the function's CPG, a multi-graph that integrates the properties of AST, CFG, and PDG into a unified data structure. Each graph representation provides a unique perspective on the source code, emphasizing different aspects of the underlying program. Specifically, the AST precisely encodes how statements are nested to produce programs, the CFG provides control flow information, and the PDG conveys data flow information. Meanwhile, GNN has shown the capability of learning structural and semantic information

```
01 aspath_put (struct stream *s, struct aspath *as, int use32bit )
02 {
03   struct assegment *seg = as->segments;
04   size_t bytes = 0;
05   if (!seg || seg->length == 0)
06     return 0;
07   if (seg)
08   {
09     while (seg && (ASSEGMENT_LEN(seg, use32bit) <= STREAM_WRITEABLE(s)))
10     {
11       struct assegment *next = seg->next;
12       int written = 0;
13       int asns_packed = 0;
14       size_t lenp;
15       while ( (seg->length - written) > AS_SEGMENT_MAX)
16       {
17         assegment_header_put (s, seg->type, AS_SEGMENT_MAX);
18         assegment_data_put (s, seg->as, AS_SEGMENT_MAX, use32bit);
19         written += AS_SEGMENT_MAX;
20         bytes += ASSEGMENT_SIZE (written, use32bit);
21       }
22       lenp = assegment_header_put (s, seg->type, seg->length - written);
23       assegment_data_put (s, (seg->as + written), seg->length - written, use32bit);
24       while (next && ASSEGMENTS_PACKABLE (seg, next))
25       {
26         assegment_data_put (s, next->as, next->length, use32bit);
27         stream_putc_at (s, lenp, seg->length - written + next->length);
28         asns_packed += next->length;
29         next = next->next;
30       }
31       bytes += ASSEGMENT_SIZE (seg->length - written + asns_packed, use32bit);
32       seg = next;
33     }
34   }
35   return bytes;
36 }
```

```
01 aspath_put (struct stream *s, struct aspath *as, int use32bit )
02 {
03   struct assegment *seg = as->segments;
04   size_t bytes = 0;
05   if (!seg || seg->length == 0)
06     return 0;
07   if (seg)
08   {
09     while (seg && (ASSEGMENT_LEN(seg, use32bit) <= STREAM_WRITEABLE(s)))
10     {
11       struct assegment *next = seg->next;
12       int written = 0;
13       int asns_packed = 0;
14       size_t lenp;
15       while ( (seg->length - written) > AS_SEGMENT_MAX)
16       {
17         assegment_header_put (s, seg->type, AS_SEGMENT_MAX);
18         assegment_data_put (s, seg->as, AS_SEGMENT_MAX, use32bit);
19         written += AS_SEGMENT_MAX;
20         bytes += ASSEGMENT_SIZE (written, use32bit);
21       }
22       lenp = assegment_header_put (s, seg->type, seg->length - written);
23       assegment_data_put (s, (seg->as + written), seg->length - written, use32bit);
24       while (next && ASSEGMENTS_PACKABLE (seg, next))
25       {
26         assegment_data_put (s, next->as, next->length, use32bit);
27         stream_putc_at (s, lenp, seg->length - written + next->length);
28         asns_packed += next->length;
29         next = next->next;
30       }
31       bytes += ASSEGMENT_SIZE (seg->length - written + asns_packed, use32bit);
32       seg = next;
33     }
34   }
35   return bytes;
36 }
```

Fig. 6. An example that can only be detected by MVulD+ (All)

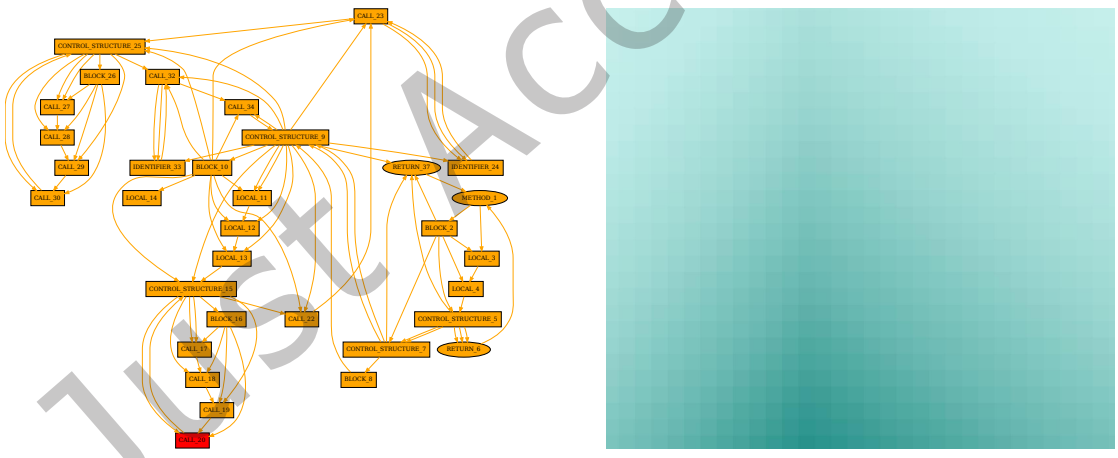


Fig. 7. The image generated from the aspath_put function's CPG and the image attention of MVulD+ when recognizing this vulnerability

from a given graph [6, 30, 53, 70]. MVulD+ utilizes a GNN component (i.e., GGNN) as the basic part of the multi-modal graph encoder to extract enriched graph features. Since our CPG is a directed graph, GGNN can flexibly handle asymmetric relationships and directionalities between nodes. Therefore, in this RQ, we aim to explore how different graph presentations impact the effectiveness of vulnerability detection.

Experiment Design: To verify the effects of different graph presentations on vulnerability detection, we treat MVulD+ as the base model. Then, we remove one structure (e.g., AST structure) from the base model to build model variants with different graph presentations. The experimental setting and dataset are set to the same as RQ-1. Meanwhile, the same optimization operations are performed on the parameters of each model on the validating set. All model types are described in Table 5 and we evaluate them with three comprehensive performance measures the same as RQ-1.

Table 5. The performance difference among model variants with different graph presentations

Model Type	F1-score	Accuracy	PR-AUC
CFG+PDG (-AST)	0.262	0.878	0.205
AST+PDG (-CFG)	0.245	0.866	0.216
AST+CFG (-PDG)	0.261	0.845	0.225
MVulD+	**0.307**	**0.907**	**0.241**

Results: Table 5 presents the comparative performance results among the different model variants. According to the results, we obtain the following observations: (1) Different graph representations exhibit varying capabilities in extracting function semantics, leading to different performances. For instance, CFG+PDG (0.262) outperforms AST+PDG (0.245) in terms of F1-score, but performs worse in terms of PR-AUC, with scores of 0.205 compared to AST+PDG's 0.216. (2) Combining all three graph representations of code enhances the models' performance. In particular, the original model (i.e., MVulD+), which integrates AST, CFG, and PDG, demonstrates improved performance among all model variants, with 0.307 F1-score, 0.908 Accuracy, and 0.241 PR-AUC.

✍ **RQ-2** ► *The three graph presentations of code (i.e., AST, CFG, and PDG) have varying impacts on learning program semantics. Combining all three graph presentations contributes to improved performance of vulnerability detection.* ◄

6.3 [RQ-3]: Compare with baselines on Function-level Vulnerability Detection

Objective: Various advanced DL-based approaches have been proposed for function-level vulnerability detection using different code representation techniques. Siow et al. [53] demonstrated that different code representations contain varying program semantics. Meanwhile, some works [7, 63] have attempted to visualize source code as images and input them into a CV model for vulnerability detection. However, these methods have limitations in fully leveraging the diverse modalities for detecting vulnerabilities. Therefore, MVulD+ integrates text-based, graph-based, and image features to enhance its effectiveness in vulnerability detection. Meanwhile, efficiency in vulnerability detection is crucial for practical applications, as it directly impacts the real-time responsiveness of security systems. Therefore, considering efficiency is imperative to ensure the timely identification and mitigation of potential vulnerabilities in software. In RQ-3, we investigate the detection performance (i.e., effectiveness and efficiency) that MVulD+ can achieve with multi-modal features and compare our approach with the state-of-the-art baseline approaches.

Experiment Design: We consider nine state-of-the-art baselines: UniXcoder [22], IVDetect [30], Devign [70], Reveal [6], LineVD [23], LineVul [15], VulCNN [63], VulDeeLocator [31], and Cppcheck [39]. To comprehensively compare the performance of these baselines with MVulD+, we train all models on the training set and monitor the training process using the validation set, selecting the best model based on the optimal F1-score. The testing set is used for evaluation. The three datasets are detailed in Section 5.1. We also evaluate effectiveness using five commonly used performance measures, including Accuracy, Precision, Recall, F1-score, and PR-AUC. The F1-score is the harmonic mean of Precision and Recall. PR-AUC is used due to the imbalanced nature of our studied dataset,

Table 6. Vulnerability detection results of MVulD+ compared against nine baselines

Methods	F1-score	Accuracy	Recall	Precision	PR-AUC
UniXcoder	0.208	0.780	**0.688**	0.123	0.188
IVDetect	0.160	0.737	0.595	0.093	0.106
Devign	0.157	0.720	0.619	0.090	0.112
Reveal	0.150	0.691	0.649	0.085	0.096
LineVD	0.241	0.848	0.570	0.152	0.185
LineVul	0.247	0.863	0.533	0.160	0.197
VulCNN	0.136	0.704	0.553	0.077	0.084
VulDeeLocator	0.176	0.746	0.644	0.102	0.121
Cppcheck	0.021	0.952	0.012	0.076	-
MVulD+	**0.307**	**0.908**	0.486	**0.225**	**0.241**

measuring the area under the Precision-Recall curve, which is suitable for imbalanced problems [49]. To eliminate potential randomness bias, we set the same seed and unify the version of *Joern* [66] used for graph extraction in both baseline models and MVulD+. Furthermore, we perform a comprehensive evaluation of the runtime of MVulD+ with studied datasets. We not only separately calculate the time occupied by each main step of MVulD+, but also compare MVulD+ with eight DL-based baselines.

Results: We discuss the results from the aspects of effectiveness and efficiency, respectively.

Effectiveness of Function-level Vulnerability Detection. The evaluation results are presented in Table 6, with the best performances highlighted in bold. The results indicate that all the baselines consistently exhibit poor performance on our dataset. While the DL-based models (i.e., UniXcoder, IVDetect, Devign, Reveal, LineVD, LineVul, VulCNN, and VulDeeLocator) show strong performance in their respective studies, their effectiveness in vulnerability detection is diminished in our research. This discrepancy may be due to the complexity and concealment of vulnerabilities, as well as the highly imbalanced distribution of vulnerabilities in our studied dataset. Notably, our MVulD+ outperforms baselines across almost all performance measures. Specifically, MVulD+ achieves 0.307 in F1-score, 0.908 in Accuracy, 0.255 in Precision, and 0.241 in PR-AUC, improving upon the baselines by 24.3%-125.7%, 5.2%-31.4%, 40.6%-192.2%, and 22.3%-186.9%, respectively. We also evaluate Cppcheck, a representative static analysis tool, on the same dataset. However, Cppcheck performs poorly, achieving 0.021 in F1-score, 0.012 in Recall, and 0.074 in Precision.

Besides, we find that LineVul only obtains 0.247 of F1-score, a strong contrast to 90+% on the Big-Vul dataset in the original paper. We first verify the correctness of our LineVul reproduction using the original dataset provided by LineVul's official source and verify its correctness. Then, we inspect each step of the data preprocessing process, as outlined in Section 4.1. In particular, "Step-1" involves three preprocessing tasks: removing blank lines, removing comments, and trimming leading and trailing spaces from lines. We preprocess the original dataset of LineVul, retrain, and test the model under the same parameter settings. The results are shown in Table 9 and we obtain the following conclusions: (1) Our reproduced LineVul performs similarly to the original version; (2) Removing blank lines and comments does not significantly affect LineVul's results; (3) Trimming leading and trailing spaces from lines leads to a drastic decrease in LineVul's performance.

Generally, for C/C++ source code, removing leading and trailing spaces does not affect the semantics of the code. To verify whether this holds for other transformer-based models, we conduct an additional experiment on UniXcoder (a widely used transformer-based pre-trained model) by using the same filtering operations. The results are presented in the right part of Table 7. Table 7 shows that the UniXcoder's performance closely resembled LineVul's before the third step of processing. However, after preprocessing, UniXcoder's performance similarly

plummeted. This suggests that such operations can have adverse effects on transformer-based models, as these models attend to each token, even if some tokens lack semantic meaning in the context of source code (e.g., spaces at the beginning and end of a line). Based on this observation, we conclude that the vulnerability detection effectiveness of LineVul after space removal is correct and the performance results obtained in our paper are reasonable.

Table 7. The reproduced results for LineVul and UniXcoder

Datasets	LineVul				UniXcoder			
	F1-score	Accuracy	Recall	Precision	F1-score	Accuracy	Recall	Precision
Origin dataset	0.90	0.95	0.86	0.95	0.86	0.98	0.82	0.90
Remove empty lines	0.85	0.98	0.79	0.93	0.85	0.98	0.82	0.88
Remove comments	0.86	0.99	0.81	0.93	0.85	0.98	0.81	0.90
Remove spaces	**0.40**	**0.94**	**0.37**	**0.45**	**0.26**	**0.95**	**0.16**	**0.68**

We also find that the average Recall of MVulD+ decreases by 19.9%. Since UniXcoder performs best in Recall among all approaches, we conduct a deeper analysis comparing the performance of UniXcoder and MVulD+ on different types of vulnerabilities from our testing set. In this experiment, we use True Positive Rate (TPR) as the evaluation metric, which measures the proportion of vulnerable functions accurately detected by the methods against the total number of actual vulnerable functions. In Table 8, the investigated vulnerability types (CWE types) are ranked according to the number of vulnerabilities, displaying only those types with differing performances between UniXcoder and MVulD+. We also exclude some CWEs that lack sufficient vulnerability instances in our dataset, as they may not accurately reflect the true detection performance of the models. The results indicate that UniXcoder outperforms MVulD+ for the majority of CWE types. For instance, UniXcoder successfully detects CWE-310, while MVulD+ does not. However, MVulD+ detects more vulnerable instances than UniXcoder for CWE-415 and CWE-189.

The performance differences in Recall between MVulD+ and UniXcoder can be attributed to the inherent nature of their respective modalities. The integration of text, image, and graph data may introduce complexities in capturing certain vulnerabilities. To illustrate this point, we analyze CWE-310 (Cryptographic Issues) as a specific example. CWE-310 vulnerabilities typically manifest as specific code patterns or improper use of cryptographic algorithms, which are often clearly reflected in the code text itself. For instance, hardcoded keys or weak cryptographic algorithms can be difficult to capture through control flow, data flow, or image representations. Graph and image features may fail to capture these cryptographic details, as they are typically tightly associated with the algorithms themselves and may not be explicitly represented in structural data. UniXcoder, as a textual modality model, may excel at capturing the textual patterns and context related to cryptographic issues. In contrast, MVulD+, with its multi-modal approach, might struggle to effectively identify and recall instances of cryptographic vulnerabilities.

We present fine-grained results of MVulD+ compared to DL-based baselines on our dataset. The results in Table 9 indicate that while our proposed MVulD+ shows a slight deficit in the number of true positives (TP) compared to the baseline methods, it achieves a favorable performance in terms of false positives (FP) and true negatives (TN). This observation suggests that although MVulD+ may overlook some vulnerabilities, it effectively reduces the false positive rate and demonstrates a strong ability to classify non-vulnerable samples as negatives.

Efficiency of Function-level Vulnerability Detection. MVulD+ comprises four main steps to complete vulnerability detection: Graph Extraction, Image Generation, Multi-modal Embedding, and Classification. We present the average running time of each main step as follows.

Table 8. The True Positive Rate (TPR) of MVulD+ and UniXcoder for function-level vulnerability detection on some vulnerability types

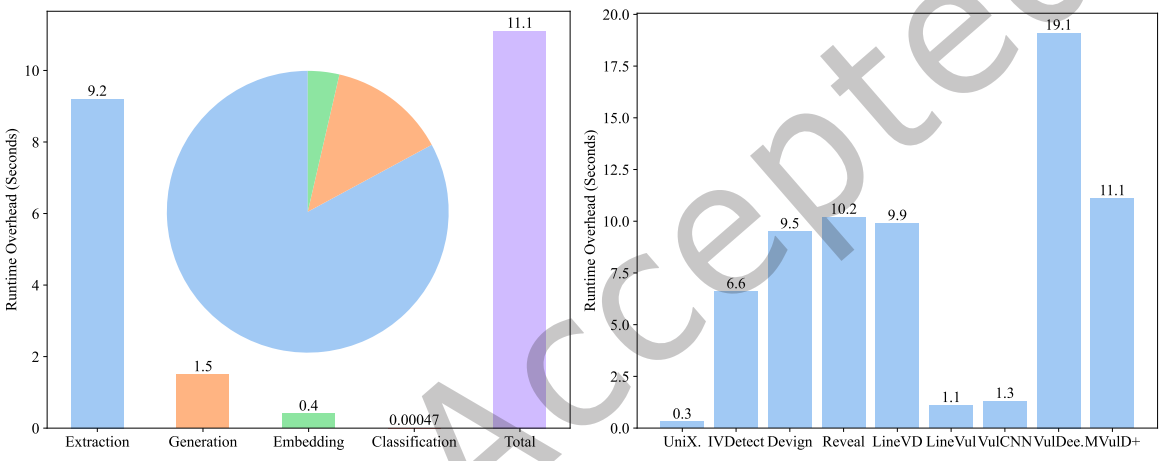
CWE Type	Name	TPR		# Vul.
		UniXcoder	MVulD+	
CWE-59	Link Following	2/3	1/3	3
CWE-310	Cryptographic Issues	2/3	0/3	3
CWE-269	Improper Privilege Management	3/3	2/3	3
CWE-254	Security Features	2/4	1/4	4
CWE-404	Improper Resource Shutdown or Release	3/4	2/4	4
CWE-415	Double Free	3/7	4/7	7
CWE-284	Improper Access Control	4/8	3/8	8
CWE-416	Use After Free	5/10	4/10	10
CWE-362	Race Condition	8/12	4/12	12
CWE-190	Integer Overflow or Wraparound	13/14	10/14	14
CWE-189	Numeric Errors	11/16	12/16	16
CWE-476	NULL Pointer Dereference	7/17	5/17	17
CWE-200	Exposure of Sensitive Information to an Unauthorized Actor	14/23	7/23	23
CWE-125	Out-of-bounds Read	25/27	18/27	27
CWE-399	Resource Management Errors	20/30	16/30	30
CWE-264	Permissions, Privileges, and Access Controls	18/31	11/31	31
CWE-20	Improper Input Validation	23/31	13/31	31
CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	71/101	52/101	101

Table 9. Fine-grained results of MVulD+ compared against DL-based baselines on our dataset

Methods	TP	FP	TN	FN
UniXcoder	**280**	2,001	7,245	**127**
IVDetect	242	2,370	6,876	165
Devign	252	2,552	6,694	155
Reveal	264	2,844	6,402	143
LineVD	232	1,290	7,956	175
LineVul	217	1,136	8,110	190
VulCNN	225	2,680	6,566	182
VulDeeLocator	262	2,305	6,941	145
MVulD+	198	**683**	**8,563**	209

- **Graph Extraction.** Given the source code of a function, the first step of MVulD+ is to extract the CPG. Fig. 8(a) presents the runtime consumed by graph extraction on our dataset. On average, it takes 9.2 seconds to extract a CPG of a function.
- **Image Generation.** After extracting all the code into a graph, MVulD+ subsequently utilizes *Graphviz* to transform the CPG into an image. Fig. 8(a) presents the average runtime for this step is 1.5 seconds.
- **Multi-modal Embedding.** Subsequently, MVulD+ embeds the code into corresponding vectors through four embedding processes: code embedding, graph embedding, image embedding, and local embedding. As illustrated in Fig. 8(a), this process requires approximately 0.4 seconds.
- **Classification.** Finally, MVulD+ detects whether a function contains a vulnerability. It only consumes about 4.7×10^{-4} seconds to complete the classification.

Fig. 8(b) shows the average runtime for each baseline on the right. We find that three approaches (i.e., UniXcoder, LineVul, and VulCNN) perform fast, while the others are comparatively slower. Specifically, UniXcoder and LineVul require only 0.3 seconds and 1.1 seconds to finish the analysis of a function in our datasets since they only simply analyze lexical to obtain the source code tokens and directly adopt a BERT-based model to detect vulnerability. As for VulCNN, it takes 1.3 seconds to execute since it only needs to extract PDG from the code and convert it to an image. Additionally, it adopts a simple CNN architecture which contributes to its high efficiency. As for IVDetect, Devign, Reveal, LineVD, and our MVulD+, they all need to extract multiple graphs to support the slice generation, which results in lower efficiency. Regarding VulDeeLocator, it needs to compile the source code since it is an LLVM-IR-based vulnerability detection approach, which is a time-consuming process and consequently leads to the longest running time in detecting software vulnerabilities. Overall, though some approaches including MVulD+ need tens of seconds to finish software vulnerability detection tasks, it still is an acceptable time in practical usage.



(a) The runtime utilized in each main step

(b) Comparative runtime overheads of baselines and MVulD+

Fig. 8. The running time used by both MVulD+ and the baselines. (a). The average time used in each step of MVulD+. (b). The average time used by each baseline.

✍ **RQ-3** ► *MVulD+ demonstrates better performance in vulnerability detection compared to the baseline methods. This indicates that the fusion of multi-modal features enhances the capabilities of MVulD+. Furthermore, MVulD+ maintains acceptable efficiency for practical applications.* ◄

6.4 [RQ-4]: Effectiveness on Top-25 Most Dangerous CWEs in Big-Vul and real-world vulnerabilities

Objective: The 2022 Common Weakness Enumeration Top 25 Most Dangerous Software Weaknesses list[3] highlights the currently most common and impactful software weaknesses. Unresolved weaknesses in software systems can lead to privacy risks and exploitable vulnerabilities, enabling attackers to potentially take control of a system, steal data, or disrupt application functionality. Therefore, we aim to investigate the effectiveness of MVulD+ in practical vulnerability detection scenarios, particularly in identifying the Top 25 Most Dangerous

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

CWEs within the Big-Vul dataset. However, the Big-Vul dataset only includes CVE entries from 2002 to 2019 and is no longer updated, meaning it does not encompass recently discovered CVE entries. To address this, we introduce a new vulnerability dataset to evaluate how MVulD+ performs with these recent CVE entries.

Experiment Design: To investigate the performance of MVulD+ on the Top-25 Most Dangerous CWEs, we select functions that belong to these CWEs within the Big-Vul dataset and categorize them according to their respective CWE types. For the analysis of recently discovered CVE entries, we utilize the newly established entries from the MegaVul [45] dataset, which is a recently released function-level vulnerability dataset containing details of CVE entries as of October 2023. The MegaVul dataset comprises 17,000 identified vulnerable functions and 320,000 non-vulnerable functions, extracted from 9,000 real-world vulnerability fix commits. It provides multi-dimensional data to facilitate the validation of vulnerability detectors' performance. Given that the projects included in the Big-Vul dataset overlap with those in MegaVul, we excluded duplicate examples by matching commit IDs to prevent data leakage. As a result, we obtained a total of 265,960 unique examples. To evaluate whether MVulD+ can detect vulnerabilities in unknown real-world projects, we utilize the entire filtered dataset from MegaVul as the testing set. This setting entails training and validating MVulD+ on the Big-Vul dataset and testing it on MegaVul.

Notice that, since not all of the Top-25 Most Dangerous CWEs are included in both datasets, we eventually collect 13 CWE types from Big-Vul and 24 CWE types from MegaVul in Table 10 and Table 11, respectively. After that, we select the version of MVulD+ that achieves the highest F1-score on the validation set to evaluate the accuracy of MVulD+ for the obtained CWEs. The accuracy measures the vulnerable functions that can be correctly detected by our method, which can be computed using TPR. We further compare the vulnerability detection performance of MVulD+ with LineVul, identified as the best-performing baseline (cf. Section 6.3).

Table 10. The TPR of MVulD+ for the Top-25 Most Dangerous CWEs in Big-Vul

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	0.419	13/31
5	CWE-125	Out-of-bounds Read	0.667	18/27
7	CWE-416	Use After Free	0.400	4/10
8	CWE-22	Path Traversal	0	0/1
11	CWE-476	NULL Pointer Dereference	0.294	5/17
13	CWE-190	Integer Overflow or Wraparound	0.714	10/14
16	CWE-862	Missing Authorization	0	0/1
19	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	0.525	53/101
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	0.502	112/223

Results: We analyze the results in Big-Vul and MegaVul, respectively.

Effectiveness on Big-Vul. Table 10 shows the performance of MVulD+ for each CWE type in terms of TPR. From the results, we can draw the following conclusions: (1) MVulD+ successfully detects 112 out of 223 vulnerabilities associated with the Top-25 Most Dangerous CWEs, resulting in an overall TPR of 50.2%. (2) Specifically, for 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+ achieves TPR of 83.3%, 66.7%, and 52.5%, respectively. These results indicate that MVulD+ performs well in detecting memory-related vulnerabilities.

(3) However, MVulD+ fails to identify some types of vulnerabilities including CWE-89, CWE-22, CWE-862, CWE-400, and CWE-611. The insufficient number of vulnerable instances for these CWEs may contribute to the observed lower detection accuracy of MVulD+.

Effectiveness on MegaVul. Overall, MVulD+ demonstrates better performance on MegeVul as evidenced by multiple evaluation metrics. Specifically, MVulD+ obtains 0.201 of F1-score, 0.893 of Accuracy, 0.158 of Precision, and 0.124 of PR-AUC, which improves the best-performing LineVul by 16.9%, 9.0%, 41.1%, and 11.7%, respectively. Table 11 further illustrates the effectiveness of MVulD+ on Top-25 Most Dangerous CWEs. The results indicate that when dealing with previously unseen real-world vulnerabilities, MVulD+ performs well, achieving a TPR of 0.226. Particularly, MVulD+ achieves TPR values of 0.538, 0.556, and 0.429 on CWE-79, CWE-306, and CWE-918, respectively. These findings underscore the robustness and effectiveness of MVulD+ in detecting authentication-related vulnerabilities.

Table 11. The TPR of MVulD+ for the Top-25 Most Dangerous CWEs in MegaVul

Rank	CWE Type	Name	TPR	Proportion
1	CWE-787	Out-of-bounds Write	0.289	367/1,269
2	CWE-79	Cross-site Scripting	0.538	28/52
3	CWE-89	SQL Injection	0.129	4/31
4	CWE-20	Improper Input Validation	0.197	158/804
5	CWE-125	Out-of-bounds Read	0.348	350/1,007
6	CWE-78	OS Command Injection	0.135	7/52
7	CWE-416	Use After Free	0.170	129/761
8	CWE-22	Path Traversal	0.385	45/117
9	CWE-352	Cross-Site Request Forgery	0	0/1
10	CWE-434	Unrestricted Upload of File with Dangerous Type	0.188	3/16
11	CWE-476	NULL Pointer Dereference	0.166	175/1,053
12	CWE-502	Deserialization of Untrusted Data	0.364	8/22
13	CWE-190	Integer Overflow or Wraparound	0.319	156/489
14	CWE-287	Improper Authentication	0.157	25/159
16	CWE-862	Missing Authorization	0.063	1/16
17	CWE-77	Command Injection	0.263	5/19
18	CWE-306	Missing Authentication for Critical Function	0.556	5/9
19	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	0.231	245/1,062
20	CWE-276	Incorrect Default Permissions	0.261	6/23
21	CWE-918	Server-Side Request Forgery	0.429	3/7
22	CWE-362	Race Condition	0.143	49/342
23	CWE-400	Uncontrolled Resource Consumption	0.213	54/253
24	CWE-611	Improper Restriction of XML External Entity Reference	0.077	1/13
25	CWE-94	Code Injection	0.214	3/14
		Sum	0.226	1,711/7,577

✍ **RQ-4** ► *Overall, MVulD+ successfully detects 50.2% of the Top-25 Most Dangerous CWEs in the Big-Vul dataset and 22.6% in real-world vulnerabilities. These results demonstrate that MVulD+ is capable of identifying vulnerabilities in practical applications.* ◄

7 DISCUSSION

7.1 Data Quality Concerns with Big-Vul

7.1.1 The impact of data quality issues in Big-Vul on the reliability of MVulD+. The application of learning-based techniques for automated software vulnerability detection has long been a focus within the software security field. These data-driven approaches rely on extensive software vulnerability datasets for training. However, Croft et al. [9] identify that all examined datasets, including Big-Vul, exhibit various data quality concerns. To assess how these data quality issues in Big-Vul influence the reliability of MVulD+ trained on this dataset, we follow previous work [9] to explore the effects of inaccurate labels. We retrain MVulD+ using manually validated samples from Big-Vul as a distinct holdout test set and evaluate model performance by comparing the original labels with the manually corrected ones. Our observations reveal a 36% decrease in precision for Big-Vul. This decline is attributed to incorrect labels for vulnerabilities, which lead the models to identify erroneous patterns for this class. As a result, the models learn patterns associated with vulnerabilities that are, in fact, non-vulnerable. Consequently, instances previously classified as true positives are reclassified as false positives during model evaluation. In contrast, we find that model recall remains largely unaffected, as we only identify label inaccuracies within the vulnerable class, resulting in a consistent count of false negatives. Nonetheless, these effects are significant, as they can lead to elevated false positive rates in MVulD+, substantially increasing the effort required for inspection during practical applications.

7.1.2 The extended experiments using newer datasets. Alternative datasets such as MegaVul [45], CVEfixes [4], ReposVul [56], and REEF [55] offer potentially higher-quality data and could provide a more robust foundation for vulnerability detection. To enhance the reliability of our results, we conduct extended experiments using these newer datasets. For MegaVul, CVEfixes, ReposVul, and REEF, we utilize the C/C++ data and select LineVul as the baseline, as LineVul is identified as having the best performance (cf. Section 6.3). Table 12 presents the results of MVulD+ and LineVul across the four datasets. The results indicate that MVulD+ outperforms LineVul in terms of F1-score, accuracy, and precision across all datasets, achieving an average F1-score of 0.312, accuracy of 0.925, and precision of 0.213.

Table 12. Vulnerability detection results of MVulD+ compared against LineVul across four datasets

Datasets	MVulD+				LineVul			
	F1-score	Accuracy	Recall	Precision	F1-score	Accuracy	Recall	Precision
MegaVul	0.373	0.903	0.571	0.277	0.338	0.874	0.634	0.230
CVEFixes	0.368	0.931	0.602	0.255	0.328	0.902	0.713	0.213
ReposVul	0.250	0.946	0.689	0.152	0.168	0.899	0.780	0.094
REEF	0.256	0.920	0.526	0.169	0.195	0.868	0.607	0.116
Average	0.312	0.925	0.597	0.213	0.257	0.886	0.684	0.163

7.2 Data Pre-process and Hyperparameter Setting

7.2.1 The impact of limited LOCs. We focus our research on functions with LOCs less than or equal to 100. In this section, we discuss the impact of limiting LOCs on the application of MVulD+ in real-world scenarios from two perspectives:

(1) Efficiency perspective: We consider the efficiency of MVulD+ and the input limitation of 512 tokens in the code encoder (i.e., UniXcoder). Therefore, we follow previous work [63] and set a threshold of 100 lines of code. While utilizing models with larger input token capacities (e.g., DeepSeek-Coder 1.3B, which can handle over 2048

tokens) mitigates this issue, it also results in higher computational resource consumption. We replace UniXcoder with DeepSeek-Coder 1.3B to implement MVulD+, retrain and validate it on datasets containing more than 100 lines of code, and test it on a dataset with functions that contain 100 lines or fewer. MVulD+ achieves 0.331 in F1-score, 0.918 in accuracy, 0.484 in recall, and 0.252 in precision, outperforming the UniXcoder-based MVulD+ in terms of F1-score, accuracy, and precision. However, using DeepSeek-Coder 1.3B with larger token inputs incurs higher time and resource consumption. Specifically, MVulD+ (DeepSeek-Coder 1.3B) requires an average of 1.6×10^{-3} seconds to complete a classification, which is more than three times the time taken by MVulD+ (UniXcoder). During training, with a batch size of 1, the Code Encoder module of MVulD+ (DeepSeek-Coder 1.3B) requires over 37.4GB of GPU memory, while the Code Encoder module of MVulD+ (UniXcoder) requires only 3.7GB. Although using DeepSeek-Coder provides performance improvements, the increased resource consumption may not be feasible for smaller organizations and individual users.

(2) Data distribution perspective: Big-Vul is a dataset extracted from real-world C/C++ projects, and we find that vulnerable functions in the dataset average 95 lines of code. Moreover, we observe that the majority (i.e., 95%) of the functions contain fewer than 100 lines of code. Therefore, we focus our research on functions with LOCs less than or equal to 100, which encompasses most practical scenarios.

Considering both resource consumption and application scenarios, using UniXcoder with LOCs less than or equal to 100 is more appropriate.

7.2.2 The impact of Joern's failed extraction. We use *Joern* to extract the graph from the function. However, *Joern* might fail to compile functions in the following situations:

(1) Syntax errors or incomplete code: *Joern* might fail to correctly parse the code and extract relevant information if there are syntax errors (such as missing semicolons or unmatched parentheses) or incomplete code segments.

(2) Complex code structure: For example, if the code contains a significant amount of nesting or recursive calls, *Joern* may struggle to fully parse these structures, leading to extraction failures.

(3) Defined macros in the code: The presence of macros can complicate the parsing process, especially if they are heavily utilized or nested.

If *Joern* is unable to compile these functions, MVulD+ will not be able to obtain graph features and image features. However, this is not critical. As shown in the results from Table 4, MVulD+ (Text Feature) can still effectively perform vulnerability detection without relying on graph and image features. This indicates that the text features of MVulD+ can operate independently in practical applications, allowing for effective vulnerability detection.

7.2.3 The impact of hyperparameter α. To clarify the impact of hyperparameter α on the performance of MVulD+, we evaluate α values of 0, 0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1, 2, and 5. According to the results shown in Fig. 9, we find that: (1) when α is set to 0, the MVulD+ performs the worst performance. This occurs because an α value of 0 results in a kl-divergence loss of 0, leading to a lack of information alignment between modalities. (2) As α increases, the performance of MVulD+ gradually improves, peaking at $\alpha = 0.1$, after which it begins to decline. Consequently, we select 0.1 as the default value for the α.

7.3 Threat to Validity

Threats to internal validity may come from the two aspects. One is related to the potential sub-optimal hyperparameter settings when fine-tuning MVulD+. We mitigate this threat by using pre-trained Unixcoder and SwinV2, as hyperparameter tuning can be extremely expensive for a transformer-based model with millions of parameters. Consequently, we focus on tuning only the learning rate, batch size, optimizer, and other relevant parameters during model training. The other threat may come from the potential experimental mistakes in the

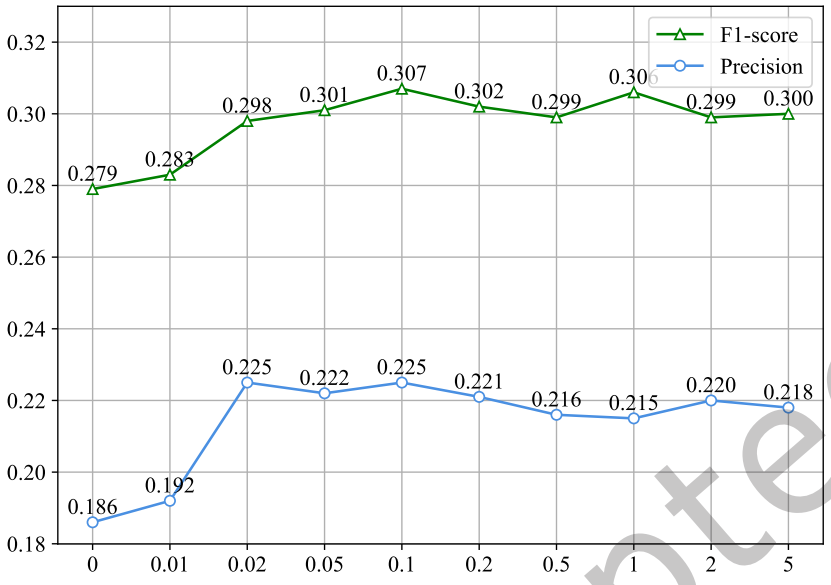


Fig. 9. The varying performance of MVulD+ with different hyperparameter α

implementation of our approach and the baseline methods. To minimize this threat, we implement our MVulD+ carefully and double-check the source code of each part as well as the experiment results. For the baseline methods, we attempt our best to reproduce the results of them by directly using the original source code from the GitHub repositories provided by corresponding authors.

Threats to external validity may exist in our studied dataset (i.e., Big-Vul). Since it is a large C/C++ code vulnerability dataset collected from open-source GitHub projects, the detection performance of MVulD+ in other programming languages (e.g., Java and Python) requires validation through additional experiments in the future. Furthermore, the performance of MVulD+ on commercial projects remains unexamined. To enhance the generalizability of MVulD+, it is essential to explore alternative, larger, and more diverse fine-grained vulnerability datasets in future work. Additionally, we follow prior works in treating the dataset as a binary classification problem, categorizing examples as either vulnerable or non-vulnerable, while neglecting the specific types of vulnerabilities during the partitioning into training, validation, and test sets. This approach may introduce bias in model performance and generalization issues. In future studies, we plan to collect a broader range of vulnerability types and conduct experiments that ensure a balanced representation of these types.

Threats to construct validity mainly relate to the adopted performance metrics in our evaluations. To mitigate these threats, we use widely used performance measures including F1-score, Accuracy, Recall, Precision, and PR-AUC.

8 RELATED WORK

Vulnerability detection (VD) is a challenging task in the field of security Many 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 based on pre-defined patterns that are manually generated by human experts. However, since it is impractical for experts to identify all possible patterns of various vulnerabilities,

PA-based vulnerability detectors (e.g., RATS [24], Checkmarx [1], and FlawFinder [2]) can only detect limited types of vulnerabilities.

ML-based methods train machine learning models on specific features to detect vulnerabilities [3, 50, 51, 68]. 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 [41, 67].

DL-based methods [6, 10, 12, 23, 30, 32–34, 40, 46, 48, 70] can automatically learn features from source code for VD. These methods can be further classified into three types based on their code representation techniques: ① Token-based methods: These methods treat source code as text and leverage natural language processing techniques to detect vulnerabilities. For example, Dam et al. [10] propose to learn features directly from source code based on long short-term memory (LSTM). Li et al. [33] propose a bidirectional long short-term memory (BLSTM) based model named VulDeePecker, which detects vulnerable code slices related to API calls. Additionally, Li et al. [32] propose SySeVR, a slice-level VD model based on the bidirectional gated recurrent unit (BGRU), designed to extract syntactic and semantic features from source code. ② Tree-based methods: Instead of treating source code as natural language sentences, tree-based methods leverage the structural information contained in a program. For example, Mou et al. [40] propose a tree-based convolutional neural network (TBCNN) based on abstract syntax trees (ASTs) of source code for program analysis. ③ Graph-based methods: To achieve a more comprehensive representation of source code, some approaches [6, 12, 23, 30, 66, 70] utilize various graph representations, such as control flow graphs (CFG), control dependency graphs (CDG), and data dependency graphs (DDG), employing graph neural networks (GNN) for VD. For example, Devign [70] applies GNN on graph-based representations (i.e., AST, CFG, DFG) to detect vulnerability. IVDetect [30] represents source code via program dependence graph (PDG) and treats the VD problem as graph-based classification via leveraging feature-attention graph convolutional network (FA-GCN) [52].

Additionally, several works have explored the potential of representing source code as images and have utilized computer vision techniques for VD. For example, Chen et al. [7] introduce a file-level defect prediction method named DTL-DP, which converts code into images based on the ASCII decimal values of code characters. Wu et al. [63] propose VulCNN, a function-level VD method that transforms code into images by applying centrality analysis on the PDG of source code. However, the generated image of these works damages the graph structure of the function which is crucially important.

Apart from only leveraging one single representation of the function, recent works have revealed the benefits of using multi-modal features in VD tasks. For example, UniXcoder [22] combines AST and code comments to learn code representation. Devign [70] integrates AST, CFG, DFG (Data Flow Graph), and code sequences to detect vulnerability. Reveal [6] leverages code property graph (CPG) [66] to build a VD model, which simultaneously considers AST, CFG, CDG, and DDG.

However, to the best of our knowledge, no prior work has attempted to fuse image features of source code with features from other modalities (i.e., text or graph) for vulnerability detection. Therefore, this paper is among the first to fuse image features, text features, and graph features of source code to facilitate function-level vulnerability detection.

9 CONCLUSION AND FUTURE WORK

In this paper, we propose MVulD+, a novel DL-based approach that learns multi-modal code features from different knowledge resources to detect function-level vulnerabilities. By leveraging UniXcoder to encode the text features, GGNN models to encode the graph features, and SwinV2 to encode the image features, MVulD+ achieves a new state-of-the-art performance compared to baseline approaches on the real-world dataset (i.e., Big-Vul). Additionally, extensive experiments have been conducted to further investigate the effectiveness of different modalities in function-level vulnerability detection. The results demonstrate that diverse code representations

are indispensable and can be effectively utilized by MVulD+ to capture rich program semantics and enhance vulnerability detection.

In the future, we aim to identify more efficient methods for code representation that preserve greater detail of the program. We also plan to explore alternative feature embedding and fusion methods to optimize the use of multi-modal information. Furthermore, we intend to investigate the underlying reasons for the varying detection capabilities of MVulD+ across different types of vulnerabilities.

ACKNOWLEDGEMENTS

This work was supported by Zhejiang Pioneer (Jianbing) Project (2025C01198(SD2)), the National Natural Science Foundation of China (Grant No.62202419), the Fundamental Research Funds for the Central Universities (No. 226-2022-00064), Zhejiang Provincial Natural Science Foundation of China (No. LY24F020008), the Ningbo Natural Science Foundation (No. 2022J184), the Key Research and Development Program of Zhejiang Province (No.2021C01105), and the State Street Zhejiang University Technology Center.

REFERENCES

[1] 2022. Checkmarx. https://www.checkmarx.com/
[2] 2022. FlawFinder. https://dwheeler.com/flawfinder/
[3] Michael Backes, Boris Köpf, and Andrey Rybalchenko. 2009. Automatic discovery and quantification of information leaks. In *2009 30th IEEE Symposium on Security and Privacy*. IEEE, 141–153.
[4] Guru Bhandari, Amara Naseer, and Leon Moonen. 2021. CVEfixes: automated collection of vulnerabilities and their fixes from open-source software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*. 30–39.
[5] Sicong Cao, Xiaobing Sun, Lili Bo, Rongxin Wu, Bin Li, and Chuanqi Tao. 2022. MVD: Memory-Related Vulnerability Detection Based on Flow-Sensitive Graph Neural Networks. *arXiv preprint arXiv:2203.02660* (2022).
[6] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering* (2021).
[7] Jinyin Chen, Keke Hu, Yue Yu, Zhuangzhi Chen, Qi Xuan, Yi Liu, and Vladimir Filkov. 2020. Software visualization and deep transfer learning for effective software defect prediction. In *Proceedings of the ACM/IEEE 42nd international conference on software engineering*. 578–589.
[8] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. 2015. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289* (2015).
[9] Roland Croft, M Ali Babar, and M Mehdi Kholoosi. 2023. Data quality for software vulnerability datasets. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 121–133.
[10] Hoa Khanh Dam, Truyen Tran, Trang Pham, Shien Wee Ng, John Grundy, and Aditya Ghose. 2017. Automatic feature learning for vulnerability prediction. *arXiv preprint arXiv:1708.02368* (2017).
[11] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 248–255.
[12] Xu Duan, Jingzheng Wu, Shouling Ji, Zhiqing Rui, Tianyue Luo, Mutian Yang, and Yanjun Wu. 2019. VulSniper: Focus Your Attention to Shoot Fine-Grained Vulnerabilities.. In *IJCAI*. 4665–4671.
[13] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. AC/C++ code vulnerability dataset with code changes and CVE summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 508–512.
[14] Linton C Freeman et al. 2002. Centrality in social networks: Conceptual clarification. *Social network: critical concepts in sociology. Londres: Routledge* 1 (2002), 238–263.
[15] Michael Fu and Chakkrit Tantithamthavorn. 2022. LineVul: A Transformer-based Line-Level Vulnerability Prediction. (2022).
[16] Tianyu Gao, Xingcheng Yao, and Danqi Chen. 2021. Simcse: Simple contrastive learning of sentence embeddings. *arXiv preprint arXiv:2104.08821* (2021).
[17] Yuexiu Gao and Chen Lyu. 2022. M2TS: Multi-Scale Multi-Modal Approach Based on Transformer for Source Code Summarization. *arXiv preprint arXiv:2203.09707* (2022).
[18] Seyed Mohammad Ghaffarian and Hamid Reza Shahriari. 2017. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Computing Surveys (CSUR)* 50, 4 (2017), 1–36.
[19] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. 2011. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings, 315–323.

[20] Graphviz. 2022. https://graphviz.org
[21] Roger Guimera, Stefano Mossa, Adrian Turtschi, and LA Nunes Amaral. 2005. The worldwide air transportation network: Anomalous centrality, community structure, and cities' global roles. *Proceedings of the National Academy of Sciences* 102, 22 (2005), 7794–7799.
[22] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. *arXiv preprint arXiv:2203.03850* (2022).
[23] David Hin, Andrey Kan, Huaming Chen, and M Ali Babar. 2022. LineVD: Statement-level vulnerability detection using graph neural networks. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 596–607.
[24] Secure Software Inc. 2022. Rough Auditing Tool for Security (RATS). https://code.google.com/p/rough-auditing-tool-for-security/
[25] Hawoong Jeong, Sean P Mason, A-L Barabási, and Zoltan N Oltvai. 2001. Lethality and centrality in protein networks. *Nature* 411, 6833 (2001), 41–42.
[26] Bonan Kou, Shengmai Chen, Zhijie Wang, Lei Ma, and Tianyi Zhang. 2023. Is model attention aligned with human attention? an empirical study on large language models for code generation. *arXiv preprint arXiv:2306.01220* (2023).
[27] Feng Li, Hao Zhang, Huaizhe Xu, Shilong Liu, Lei Zhang, Lionel M Ni, and Heung-Yeung Shum. 2023. Mask dino: Towards a unified transformer-based framework for object detection and segmentation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 3041–3050.
[28] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2015. Gated Graph Sequence Neural Networks. *Computer Science* (2015).
[29] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2015. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493* (2015).
[30] Yi Li, Shaohua Wang, and Tien N Nguyen. 2021. 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*. 292–303.
[31] Zhen Li, Deqing Zou, Shouhuai Xu, Zhaoxuan Chen, Yawei Zhu, and Hai Jin. 2021. Vuldeelocator: a deep learning-based fine-grained vulnerability detector. *IEEE Transactions on Dependable and Secure Computing* (2021).
[32] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2021. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* (2021).
[33] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681* (2018).
[34] Guanjun Lin, Jun Zhang, Wei Luo, Lei Pan, and Yang Xiang. 2017. POSTER: Vulnerability discovery with function representation learning from unlabeled projects. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2539–2541.
[35] Yutong Lin, Ze Liu, Zheng Zhang, Han Hu, Nanning Zheng, Stephen Lin, and Yue Cao. 2022. Could Giant Pre-trained Image Models Extract Universal Representations? *Advances in Neural Information Processing Systems* 35 (2022), 8332–8346.
[36] Bingchang Liu, Liang Shi, Zhuhua Cai, and Min Li. 2012. Software vulnerability discovery techniques: A survey. In *2012 fourth international conference on multimedia information networking and security*. IEEE, 152–156.
[37] Ze Liu, Han Hu, Yutong Lin, Zhuliang Yao, Zhenda Xie, Yixuan Wei, Jia Ning, Yue Cao, Zheng Zhang, Li Dong, et al. 2022. Swin transformer v2: Scaling up capacity and resolution. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 12009–12019.
[38] Ilya Loshchilov and Frank Hutter. 2017. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101* (2017).
[39] Daniel Marjamäki. 2013. Cppcheck: a tool for static c/c++ code analysis. *URL: https://cppcheck. sourceforge. io* (2013).
[40] Lili Mou, Ge Li, Zhi Jin, Lu Zhang, and Tao Wang. 2014. TBCNN: A tree-based convolutional neural network for programming language processing. *arXiv preprint arXiv:1409.5718* (2014).
[41] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. 2007. Predicting vulnerable software components. In *Proceedings of the 14th ACM conference on Computer and communications security*. 529–540.
[42] Chao Ni, Xinrong Guo, Yan Zhu, Xiaodan Xu, and Xiaohu Yang. 2023. Function-level vulnerability detection through fusing multi-modal knowledge. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1911–1918.
[43] Chao Ni, Liyu Shen, Wei Wang, Xiang Chen, Xin Yin, and Lexiao Zhang. 2023. FVA: Assessing Function-Level Vulnerability by Integrating Flow-Sensitive Structure and Code Statement Semantic. In *2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC)*. IEEE, 339–350.
[44] Chao Ni, Liyu Shen, Xiaodan Xu, Xin Yin, and Shaohua Wang. 2024. Learning-based models for vulnerability detection: An extensive study. *arXiv preprint arXiv:2408.07526* (2024).
[45] Chao Ni, Liyu Shen, Xiaohu Yang, Yan Zhu, and Shaohua Wang. 2024. MegaVul: AC/C++ Vulnerability Dataset with Comprehensive Code Representations. In *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*. IEEE, 738–742.
[46] Chao Ni, Xin Yin, Kaiwen Yang, Dehai Zhao, Zhenchang Xing, and Xin Xia. 2023. Distinguishing look-alike innocent and vulnerable code by subtle semantic representation learning and explanation. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1611–1622.

[47] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).
[48] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. 2018. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE international conference on machine learning and applications (ICMLA)*. IEEE, 757–762.
[49] Takaya Saito and Marc Rehmsmeier. 2015. The precision-recall plot is more informative than the ROC plot when evaluating binary classifiers on imbalanced datasets. *PloS one* 10, 3 (2015), e0118432.
[50] Umesh Shankar, Kunal Talwar, Jeffrey S Foster, and David Wagner. 2001. Detecting format string vulnerabilities with type qualifiers. In *10th USENIX Security Symposium (USENIX Security 01)*.
[51] Lwin Khin Shar, Lionel C Briand, and Hee Beng Kuan Tan. 2014. Web application vulnerability prediction using hybrid program analysis and machine learning. *IEEE Transactions on dependable and secure computing* 12, 6 (2014), 688–707.
[52] Min Shi, Yufei Tang, Xingquan Zhu, Yuan Zhuang, Maohua Lin, and Jianxun Liu. 2022. Feature-attention graph convolutional networks for noise resilient learning. *IEEE Transactions on Cybernetics* (2022).
[53] Jing Kai Siow, Shangqing Liu, Xiaofei Xie, Guozhu Meng, and Yang Liu. 2022. Learning Program Semantics with Code Representations: An Empirical Study. *arXiv preprint arXiv:2203.11790* (2022).
[54] Laurens van der Maaten and Geoffrey Hinton. 2008. Visualizing Data using t-SNE. *Journal of Machine Learning Research* 9, 86 (2008), 2579–2605. http://jmlr.org/papers/v9/vandermaaten08a.html
[55] Chaozheng Wang, Zongjie Li, Yun Pena, Shuzheng Gao, Sirong Chen, Shuai Wang, Cuiyun Gao, and Michael R Lyu. 2023. Reef: A framework for collecting real-world vulnerabilities and fixes. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1952–1962.
[56] Xinchen Wang, Ruida Hu, Cuiyun Gao, Xin-Cheng Wen, Yujia Chen, and Qing Liao. 2024. ReposVul: A Repository-Level High-Quality Vulnerability Dataset. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*. 472–483.
[57] Yixuan Wei, Han Hu, Zhenda Xie, Ze Liu, Zheng Zhang, Yue Cao, Jianmin Bao, Dong Chen, and Baining Guo. 2023. Improving clip fine-tuning performance. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 5439–5449.
[58] Martin Weyssow, Houari Sahraoui, and Bang Liu. 2022. Better Modeling the Programming World with Code Concept Graphs-augmented Multi-modal Learning. *arXiv preprint arXiv:2201.03346* (2022).
[59] Sarah Wiegreffe and Yuval Pinter. 2019. Attention is not not explanation. *arXiv preprint arXiv:1908.04626* (2019).
[60] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*. 38–45.
[61] Jiesheng Wu, Fangwei Hao, Weiyun Liang, and Jing Xu. 2023. Transformer fusion and pixel-level contrastive learning for RGB-D salient object detection. *IEEE Transactions on Multimedia* 26 (2023), 1011–1026.
[62] Lijun Wu, Juntao Li, Yue Wang, Qi Meng, Tao Qin, Wei Chen, Min Zhang, Tie-Yan Liu, et al. 2021. R-drop: Regularized dropout for neural networks. *Advances in Neural Information Processing Systems* 34 (2021), 10890–10905.
[63] Yueming Wu, Deqing Zou, Shihan Dou, Wei Yang, Duo Xu, and Hai Jin. 2022. VulCNN: An Image-inspired Scalable Vulnerability Detection System. (2022).
[64] Zhenda Xie, Zheng Zhang, Yue Cao, Yutong Lin, Jianmin Bao, Zhuliang Yao, Qi Dai, and Han Hu. 2022. Simmim: A simple framework for masked image modeling. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 9653–9663.
[65] Zhenda Xie, Zheng Zhang, Yue Cao, Yutong Lin, Yixuan Wei, Qi Dai, and Han Hu. 2023. On data scaling in masked image modeling. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 10365–10374.
[66] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 590–604.
[67] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. 2012. Generalized vulnerability extrapolation using abstract syntax trees. In *Proceedings of the 28th Annual Computer Security Applications Conference*. 359–368.
[68] Fabian Yamaguchi, Alwin Maier, Hugo Gascon, and Konrad Rieck. 2015. Automatic inference of search patterns for taint-style vulnerabilities. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 797–812.
[69] Xin Yin, Chao Ni, and Shaohua Wang. 2024. Multitask-based evaluation of open-source llm on software vulnerability. *IEEE Transactions on Software Engineering* (2024).
[70] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems* 32 (2019).