Automated Software Engineering (2025) 32:67
https://doi.org/10.1007/s10515-025-00543-3

MCL-VD: Multi-modal contrastive learning with LoRA-enhanced GraphCodeBERT for effective vulnerability detection

Yi Cao[1] · **Xiaolin Ju**[1] · **Xiang Chen**[1] · **Lina Gong**[2]

Received: 23 December 2024 / Accepted: 20 July 2025

Abstract
Vulnerability detection in software systems is a critical challenge due to the increasing complexity of code and the rising frequency of security vulnerabilities. Traditional approaches typically rely on single-modality inputs and struggle to distinguish between similar code snippets. However, multi-modal methods find it challenging to balance performance and efficiency. To address these challenges, we propose MCL-VD, a framework that leverages multi-modal inputs including source code, code comments, and AST to capture complementary structural and contextual information. We employ LoRA, which reduces the computational burden by optimizing the number of trainable parameters without sacrificing performance. Additionally, we apply multi-modal contrastive learning to align and differentiate the representations across the three modalities, thereby enhancing the model's discriminative power and robustness. We designed and conducted experiments on three public benchmark datasets, i.e., Devign, Reveal, and Big-Vul. The experimental results show that MCL-VD significantly outperforms the best-performing baselines, achieving F1-score improvements ranging from 4.86% to 17.26%. These results highlight the effectiveness of combining multi-modal contrastive learning with LoRA optimization, providing a powerful and efficient solution for vulnerability detection.

Keywords Vulnerability detection · Contrastive learning · Low-rank adaptation · Deep learning

Extended author information available on the last page of the article

Published online: 28 July 2025

1 Introduction

Vulnerability detection in software systems is critical to ensure the security, integrity, and reliability of applications (Neuhaus et al. 2007; Chakraborty et al. 2021; Yang et al. 2022). As technology advances and software systems grow increasingly complex, identifying and addressing vulnerabilities before they are exploited has become paramount. The National Vulnerability Database[1] recorded over 20,000 new vulnerabilities in 2023 alone, marking a consistent upward trend in the discovery of software vulnerabilities over the past decade. Furthermore, a report published by Palo Alto Networks Unit 42[2] highlights that 75% of exploited vulnerabilities in 2023 were associated with unpatched software, emphasizing the critical need for proactive detection and mitigation measures in cybersecurity. Exploited vulnerabilities frequently serve as entry points for attackers to gain unauthorized access, exfiltrate sensitive data, or cause severe system failures. These breaches can lead to severe consequences, including data leaks, financial losses, reputational harm, and legal repercussions. The challenges of vulnerability detection are further amplified in modern distributed architectures and microservices-based systems, underscoring the urgent need for robust and scalable detection techniques.

Early methods for vulnerability detection primarily relied on static analysis techniques, which examined source code without executing it. Tools such as FindBugs (Livshits and Lam 2005) and PMD (Ayewah et al. 2008) automated the detection of known vulnerabilities by matching predefined patterns or rules. These methods effectively identified well-understood vulnerabilities, such as buffer overflows, SQL injection, and cross-site scripting (XSS) (Chess and McGraw 2004). However, static analysis techniques faced several inherent limitations (Shin and Williams 2008; Nunes et al. 2019; Li et al. 2018). They struggled to detect complex or novel vulnerabilities that did not conform to existing rules, resulting in high false-positive rates that required manual effort to review and confirm potential issues. Moreover, static analysis was ineffective for detecting vulnerabilities arising from runtime behavior, as it did not execute the code.

Dynamic analysis techniques were introduced to complement static analysis (Kim et al. 2016; Liu et al. 2016; Li et al. 2024; Kan et al. 2021). These methods monitored program execution to detect vulnerabilities during runtime, such as memory corruption or race conditions. Tools like Valgrind (Nethercote and Seward 2007) and DynamoRIO (Bruening and Amarasinghe 2004) enabled developers to observe and analyze the actual behavior of programs under various conditions. Although dynamic analysis offered deeper insights into runtime vulnerabilities, it was computationally expensive and impractical for large-scale applications due to the need for extensive test cases and resource-intensive simulations.

In recent years, deep learning has significantly advanced the field of vulnerability detection. Models such as Convolutional Neural Networks (CNNs) (LeCun et al. 1998) and Recurrent Neural Networks (RNNs) (Rumelhart et al. 1986) have been effectively utilized to analyze source code. For instance, VulDeePecker (Li

[1] https://nvd.nist.gov

[2] https://unit42.paloaltonetworks.com/

et al. 2018) leveraged CNNs to identify vulnerabilities in code slices, while SySeVR (Li et al. 2021) employed RNNs to capture semantic relationships within code sequences. These approaches surpassed traditional rule-based methods by learning patterns directly from data, enabling better generalization across diverse types of vulnerabilities. Building on this progress, Graph Neural Networks (GNNs) (Ayewah et al. 2008) have further expanded deep learning capabilities by modeling structural relationships in code. For example, Devign (Zhou et al. 2019) utilized GNNs to represent function call graphs and variable dependencies, achieving more accurate vulnerability detection by integrating both the syntactic and semantic aspects of code.

The development of pre-trained language models has revolutionized the field, providing even greater capabilities for vulnerability detection. Models such as CodeBERT (Feng et al. 2020), GraphCodeBERT (Guo et al. 2020), and CodeT5 (Wang et al. 2021) are pre-trained on massive corpora of programming languages and natural language, enabling them to understand both syntactic and semantic structures of code. These models have been successfully applied to various tasks, including defect detection, code summarization, and vulnerability detection. For instance, CodeBERT learns source code representations and comments, while GraphCodeBERT incorporates structural information from AST to enhance its understanding of code. Additionally, ContraBERT (Liu et al. 2023) applies contrastive learning to improve vulnerability detection performance by aligning code representations with contextual information. These advancements have significantly improved the performance of vulnerability detection tasks.

Despite significant advancements in vulnerability detection, existing methods still face several key challenges. One major limitation is the lack of multi-modal inputs, as most models rely predominantly on source code alone (Sharma et al. 2021; Russell et al. 2018; Hanif and Maffeis 2022; Marjanov et al. 2022). This approach fails to leverage complementary information from other modalities, such as AST and code comments, which provide critical structural and contextual insights for detecting subtle or complex vulnerabilities. Additionally, many datasets suffer from the absence of code comments (Svyatkovskiy et al. 2020; Wen et al. 2019), thereby reducing the potential value of this modality.

Another challenge is the computational overhead introduced by incorporating multiple modalities (Lahat et al. 2015), which can make model training and optimization more resource-intensive. While combining multiple modalities has improved detection accuracy, aligning and integrating these diverse data types remain difficult. Many existing methods struggle to effectively align and differentiate embeddings from various modalities, limiting the model's ability to capture the complex relationships among them. Furthermore, directly fusing multiple modalities can lead to noisy or redundant information, impeding the model's learning efficiency.

A particularly pressing issue in vulnerability detection is the challenge of distinguishing between similar code snippets (Chakraborty et al. 2021; Liu et al. 2023; Jiang et al. 2024). Vulnerable code is often very similar to benign code, making it difficult for models to differentiate between the two. This challenge is exacerbated by many existing methods struggle with fine-grained feature extraction and alignment, hindering the model's ability to capture subtle differences between vulnerable and non-vulnerable code. Additionally, many vulnerability detection models lack robust-

ness, making them sensitive to minor variations in code that do not significantly affect their functionality but can still alter the model's predictions. This lack of robustness is particularly problematic in real-world scenarios, where code may vary in style or contain small changes that do not alter its underlying behavior but can negatively impact vulnerability detection performance.

To address these challenges, we propose a novel approach called **MCL-VD** (Multi-modal Contrastive Learning for Vulnerability Detection). Our method incorporates a three-modal input representation, combining source code, AST, and comments to understand code comprehensively. To overcome the issue of missing comments in datasets, we employ the large language model GPT-4o-mini to generate synthetic comments, enriching the comment modality and ensuring consistency across all samples. To manage the increased computational cost and parameter load introduced by the three modalities, we apply **LoRA** (Low-Rank Adaptation) (Hu et al. 2021), enabling efficient fine-tuning of model parameters without excessive computational overhead. Finally, we design a multi-modal contrastive learning framework to align and differentiate embeddings across the three modalities, capturing both shared and modality-specific semantic attributes to improve the robustness and accuracy of vulnerability detection.

To evaluate the effectiveness of MCL-VD, we conducted experiments on three widely used vulnerability detection datasets: Devign, Reveal, and Big-Vul (Zhou et al. 2019; Chakraborty et al. 2021; Fan et al. 2020). These datasets provide diverse real-world vulnerabilities, offering a comprehensive benchmark for our approach. We compared MCL-VD against eight state-of-the-art baselines, including SySeVR, VulDeePecker, Devign, ReGVD, CodeBERT, GraphCodeBERT, ContraBERT, and SCL-CVD (Li et al. 2021, 2018; Zhou et al. 2019; Nguyen et al. 2022; Feng et al. 2020; Guo et al. 2020; Liu et al. 2023; Wang et al. 2024). The results demonstrate the advantages of our multi-modal framework in achieving superior vulnerability detection performance across various software environments.

The main contributions of this paper are:

- We propose a novel multi-modal framework for vulnerability detection that incorporates code, AST, and comments to create a comprehensive representation.
- We generate supplemental comments during preprocessing to address the issue of incomplete or missing comments, enriching the comment modality for better multi-modal learning.
- We utilize LoRA to fine-tune the model, thereby reducing its computational cost and training time without compromising performance.
- We apply contrastive learning to improve alignment and differentiation across modalities, thereby enhancing the robustness and discriminative power of the detection process.

We share MCL-VD[3] to encourage future studies on vulnerability detection. The rest of this paper is organized as follows. Section 2 summarizes related work of our study. Next, Section 3 presents the design of MCL-VD. Section 4 analyzes the experimental

[3] https://github.com/olivergo7/MCL-VD

settings, including the comparative baselines and performance metrics considered. Section 5 reports the experimental results. Then, Section 6 introduces some discussions and threats to validity. Finally, we conclude our work and show future studies in Section 7.

2 Related work

2.1 Vulnerability detection

Early work on vulnerability detection focused heavily on static analysis methods, which often suffered from high false negative rates due to their inability to capture dynamic program behavior (Russell et al. 2018; Li et al. 2018). Static analysis was typically used to analyze source code token sequences, but this approach was limited in its ability to model complex relationships and dependencies within the code.

With the advent of machine learning, traditional approaches began to evolve. The application of machine learning models to vulnerability detection has gained traction, enabling more automated and efficient analysis. Early machine learning-based techniques utilized classifiers, such as Support Vector Machines, on features extracted from source code (Li et al. 2018). However, these methods often struggled with scalability and the complexity of modern software.

Recent advancements in deep learning have introduced models such as RNNs and CNNs for the automated extraction of meaningful features from source code (Li et al. 2021, 2018; Russell et al. 2018). These deep learning models have demonstrated effectiveness in detecting vulnerabilities by modeling sequential patterns within code token sequences. However, they still faced challenges in capturing the structural dependencies within the source code. These structural relationships are critical for accurately identifying vulnerabilities, as they encapsulate the contextual and hierarchical nature of software systems.

To address these limitations, researchers began exploring the use of GNNs for vulnerability detection. GNN-based approaches represent source code as graph structures, capturing the sequential relationships and complex dependencies — such as data flow, control flow, and function call interactions — among various program components. For example, Devign (Zhou et al. 2019) proposed a GNN-based model that learns from a rich set of semantic representations of the code, significantly improving performance compared to traditional methods. Additionally, GraphCodeBERT (Guo et al. 2020) and VulCNN (Wu et al. 2022) leveraged GNNs to process code as graph structures, enhancing vulnerability detection by incorporating graph-based representations of code.

Our approach builds upon these advancements by combining multiple modalities of data into a unified model, such as source code, AST, and code comments. By fusing these modalities, we provide a more comprehensive representation of the code, capturing syntactic and semantic information critical for vulnerability detection.

2.2 Low-rank adaptation of pre-trained models

Low-rank adaptation has emerged as a compelling alternative to previous parameter-efficient fine-tuning (PEFT) methods (Hu et al. 2021). LoRA provides a way to fine-tune large pre-trained models with a significantly reduced number of trainable parameters while maintaining competitive generalization performance compared to full fine-tuning. One of its key advantages is that it does not introduce any additional latency during the inference phase, making it particularly attractive for real-time applications.

Since the introduction of LoRA, several efforts have been made to enhance its learning efficiency and performance. Recent research has focused on improving the learning curve of LoRA-based models by exploring novel optimization strategies and regularization techniques (Liu et al. 2024; Hayou et al. 2024; Meng et al. 2024). These advancements aim to accelerate the convergence of the model and improve its adaptability to new tasks. Additionally, some studies have sought to further reduce the number of trainable parameters in LoRA to make it even more computationally efficient, especially in resource-constrained environments (Zhang et al. 2023; Kopiczko et al. 2023; Renduchintala et al. 2023).

Another promising direction involves training LoRA with quantized pre-trained weights to reduce the memory footprint during training. This approach has significantly decreased memory requirements without sacrificing model performance, making it particularly suitable for large-scale deployment on edge devices or in environments with limited resources (Dettmers et al. 2023; Li et al. 2023).

In the context of vulnerability detection, LoRA enables the efficient fine-tuning of pre-trained models, such as GraphCodeBERT, without requiring the retraining of the entire network. This is particularly important for multi-modal tasks, where different data types (e.g., code, AST, and comments) are processed simultaneously. By applying LoRA, we reduce the computational and storage costs of fine-tuning large models while maintaining their generalization power.

2.3 Contrastive learning for code representation learning

Contrastive learning, initially developed for visual representation learning (Chen et al. 2020; He et al. 2020), aims to maximize the similarity between related samples while minimizing the similarity between unrelated ones. Over the years, contrastive learning has found applications in various domains, including natural language processing (NLP) and code analysis (Neelakantan et al. 2022; Xu et al. 2022; Aberdam et al. 2021), where it has been utilized to learn robust representations of source code by contrasting different code variants.

In early applications, contrastive learning relied on data augmentation techniques, such as rotation, scaling, and cropping, in visual tasks. Similar data augmentation strategies have been employed for NLP and code tasks, such as generating different variants of code through obfuscation or code transformation (Jain et al. 2020). For instance, ContraBERT (Liu et al. 2023) employed data augmentation methods to create code variants and then applied contrastive learning to fine-tune a pre-trained model for tasks like code clone detection, defect detection, and code search.

In the context of vulnerability detection, contrastive learning has demonstrated significant promise in enhancing the robustness of models. By creating positive and negative pairs of code samples, where positive pairs consist of semantically similar code variants, and negative pairs consist of unrelated code samples, contrastive learning helps models better understand the underlying structure and semantics of the code. For example, ContraFlow (Cheng et al. 2022) employed contrastive learning to strengthen the model's ability to recognize vulnerabilities despite code syntax and structure variations.

Our approach adopts contrastive learning to enhance the robustness of multi-modal models for vulnerability detection. Introducing positive and negative pairs of multi-modal code samples enables our model to learn richer and more robust representations of code, AST, and comments. This contrastive learning process enables the model to distinguish between meaningful code features and noise, thereby improving the accuracy and robustness of vulnerability detection.

3 Approach

In this study, we propose MCL-VD, a vulnerability detection model that leverages multi-modal contrastive learning and LoRA to enhance the model's ability to capture and differentiate semantic features across multiple code representations. The overall architecture of MCL-VD is shown in Fig. 1. MCL-VD consists of three main steps: ① **Code Annotation Supplementation**: Missing or incomplete comments are supple-

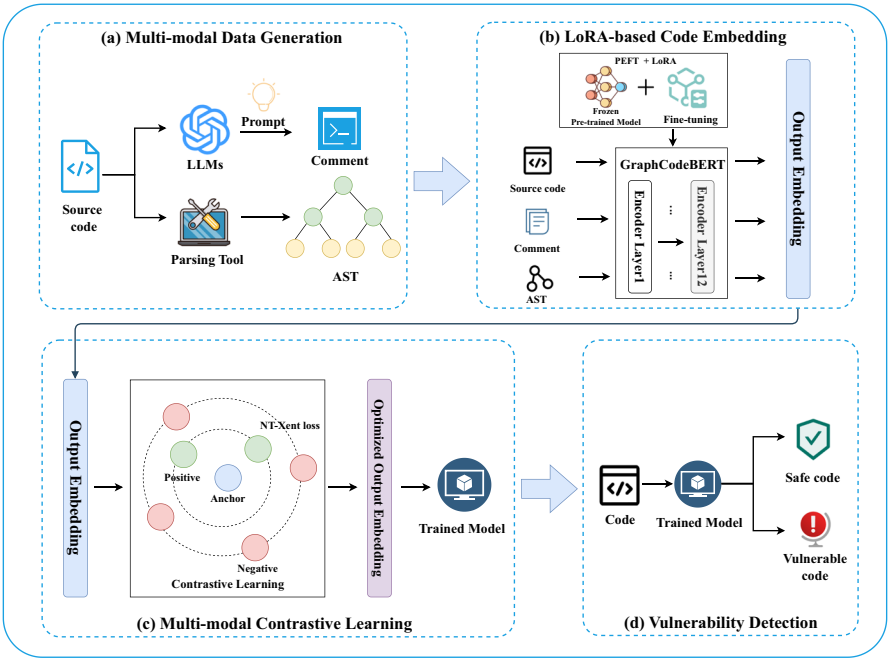


Fig. 1 The architecture of our approach MCL-VD

mented by generating synthetic annotations using GPT-4o-mini, guided by prompt tuning to produce contextually relevant comments that enrich the comment modality; ② **LoRA-based Fine-tuning for Multi-modal Representation Learning**: LoRA is applied to fine-tune the model and reduce computational cost while maintaining performance. By adapting each modality (source code, comments, AST) with a unique LoRA-based mechanism, we minimize the number of trainable parameters, decreasing both model loss and training time. These modality-specific representations are then fed into GraphCodeBERT for further representation learning; ③ **Cross-modality Alignment with Contrastive Learning**: Contrastive learning is employed to align and differentiate the embeddings across the three modalities, ensuring semantic consistency while preserving the unique characteristics of each modality for robust vulnerability detection. Details of each step in MCL-VD are presented in the following subsections.

3.1 Code comment supplementation with GPT-4o-mini

In many real-world code repositories, the absence or insufficiency of comments poses a significant challenge, as it limits the contextual information available for multi-modal models. To address this issue, we utilize GPT-4o-mini, a version of GPT-4 optimized for code-related tasks, to automatically generate annotations for code snippets that lack comments.

The generated comments are guided by a specifically designed prompt instructing GPT-4o-mini to provide explicit, concise, and contextually relevant annotations. Figure 2 illustrates the prompt structure that guides the model in generating these comments.

The design of this prompt is intentional, as it ensures that the generated comments contain all the essential elements needed for a thorough understanding of the code. By requesting a brief function summary, the prompt guides the model to capture the high-level purpose of the code. This provides a quick overview of what the function is designed to do, which is crucial for understanding its behavior and identifying any potential vulnerabilities.

Including input parameter descriptions and expected output or return values further enriches the comment. This approach ensures that the model understands the function's purpose and gains insight into how the inputs interact with the function, as well as what the expected result is. This level of detail is crucial for identifying vulnerabilities related to incorrect input handling or unintended side effects in the output.

Additionally, instructing the model to document any exceptions raised is particularly significant, as it focuses the model's attention on potential error-handling mechanisms. Exceptions often highlight edge cases or unexpected behaviors that could result in security vulnerabilities. By incorporating this information, we ensure the model considers all possible scenarios when evaluating the function's security implications.

By leveraging GPT-4o-mini in this manner, we supplement missing or incomplete comments, enriching the comment modality and improving the model's ability to learn from a more complete set of features. The generated comments fill in gaps and

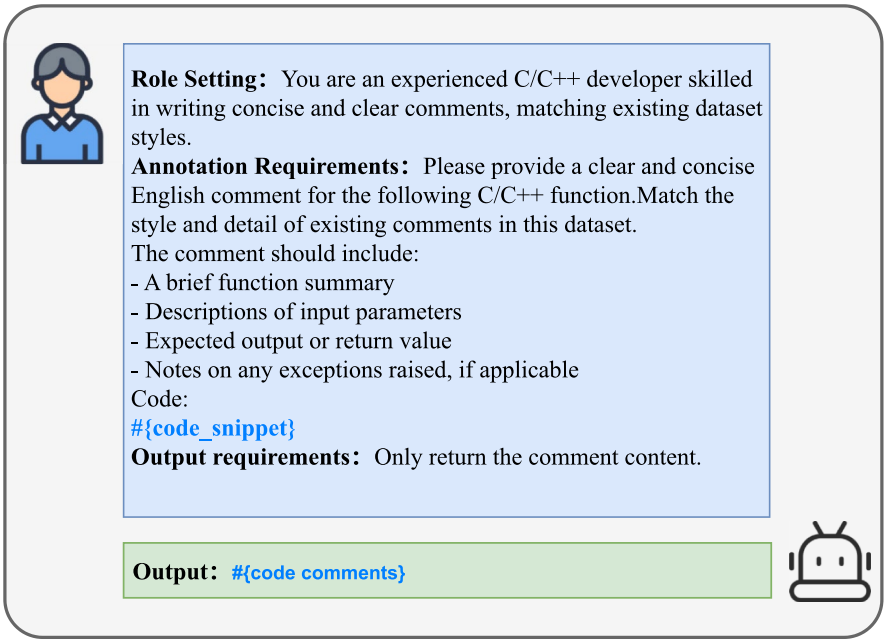


Fig. 2 Prompt design for code comment generation

provide additional context essential for detecting vulnerabilities, particularly those that may be reflected in both the code's logic and its natural language description.

3.2 Low-rank adaptation for multi-modal inputs

This section introduces an enhanced version of LoRA specifically designed to improve vulnerability detection with multi-modal inputs. Traditional LoRA techniques apply a uniform low-rank adaptation across model parameters, which works well for single-modality tasks. However, when dealing with multiple modalities, such as code, AST, and code comments, a single low-rank adaptation does not fully exploit the distinct characteristics inherent to each modality. We propose a modality-specific low-rank adaptation approach to address this limitation, enabling the model to fine-tune each modality independently while preserving shared semantics through contrastive learning.

In multi-modal vulnerability detection, different input types (code tokens, AST structures, and code comments) each provide unique information that cannot be fully captured by a single low-rank update. To address this, we introduce a unified base weight W_{base} for the pre-trained GraphCodeBERT model and apply separate low-rank adapters to each modality. Specifically, for modalities $m \in \{\text{code}, \text{ast}, \text{comment}\}$, we learn two small matrices $A_m \in \mathbb{R}^{D \times r}$ and $B_m \in \mathbb{R}^{r \times D}$ (with rank $r \ll D$) and fuse them into the model as:

$$W_{\text{total}} = W_{\text{base}} + \alpha\big(A_{\text{code}}B_{\text{code}} + A_{\text{ast}}B_{\text{ast}} + A_{\text{comment}}B_{\text{comment}}\big) \quad (1)$$

where α scales the low-rank updates. This formulation ensures that each modality contributes its specialized adjustment while sharing the bulk of the pre-trained parameters.

For the code modality, $A_{\text{code}}, B_{\text{code}}$ capture syntactic and control-flow patterns that signal vulnerabilities. For the AST modality, $A_{\text{ast}}, B_{\text{ast}}$ emphasize hierarchical relationships and data-flow dependencies. For the comment modality, $A_{\text{comment}}, B_{\text{comment}}$ extract semantic cues from natural language annotations, aligning contextual descriptions with code behavior. By cumulatively adding these low-rank adapter outputs to the shared W_{base}, the model retains a compact parameter footprint—only $3Dr$ additional parameters—while independently tuning each modality's representation.

We further integrate this modality-specific LoRA within our multi-modal contrastive learning framework, so that shared semantics across modalities are preserved and aligned, enhancing robustness without incurring the cost of full-parameter tuning.

3.3 Enhanced multi-modal contrastive learning for representation differentiation

Input: Dataset $\mathcal{D}$ with fields {code, ast, comments, label}, Temperature τ, Batch B
Output: Optimized Multi-modal Embeddings **H**

for epoch in training epochs **do**
  for each B in $\mathcal{D}$ **do**
    Encode source code in B to obtain code_reps
    Encode AST in B to obtain ast_reps
    Encode comments in B to obtain comment_reps
    Calculate classification loss $\mathcal{L}_{\text{cls}}$ using code_reps and batch labels
    if batch sizes of code, AST, and comments match **then**
      Calculate contrastive loss $\mathcal{L}_{\text{code-ast}}$ between code_reps and ast_reps using NT-Xent loss
      Calculate contrastive loss $\mathcal{L}_{\text{code-comment}}$ between code_reps and comment_reps using NT-Xent loss
      Calculate contrastive loss $\mathcal{L}_{\text{ast-comment}}$ between ast_reps and comment_reps using NT-Xent loss
      $\mathcal{L}_{\text{contrast\_total}} = \frac{1}{3}(\mathcal{L}_{\text{code-ast}} + \mathcal{L}_{\text{code-comment}} + \mathcal{L}_{\text{ast-comment}})$
      Compute total loss: $\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{cls}} + \mathcal{L}_{\text{contrast\_total}}$
    else
      Set total loss as classification loss: $\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{cls}}$
    end if
    Update model parameters to minimize $\mathcal{L}_{\text{total}}$
  end for
end for
return Optimized Multi-modal Embeddings **H**

Algorithm 1 Enhanced Multi-Modal Contrastive Learning for MCL-VD

To address the unique challenges of multi-modal vulnerability detection, we introduce an enhanced multi-modal contrastive learning framework designed to leverage and differentiate the representations of code, AST, and comment modalities. Unlike traditional contrastive learning frameworks, which are typically designed for single-modality or homogeneous data, our cross-modal approach modifies the framework

to suit the heterogeneous and complementary nature of code, AST, and comments, allowing the model to capture shared insights while maintaining modality-specific distinctions crucial for effective vulnerability detection.

Our enhanced multi-modal learning strategy computes pairwise contrastive losses tailored for each modality pair, specifically targeting the unique semantic relationships between code, AST, and comments. By defining modality-specific contrastive objectives, our approach ensures that the model can learn common and distinctive patterns across modalities, ultimately improving the robustness of the multi-modal embeddings.

Algorithm 1 outlines the steps involved in our enhanced contrastive learning framework. Specifically, within each mini-batch, the embeddings of code, AST, and comments originating from the same function are treated as positives, while all embeddings belonging to different functions in that batch serve as negatives. For example, a code embedding $\mathrm{h}_{\mathrm{code}}^{(i)}$ uses $\{\mathrm{h}_{\mathrm{ast}}^{(i)}, \mathrm{h}_{\mathrm{comment}}^{(i)}\}$ as its positive set and all $\{\mathrm{h}_{\mathrm{code}}^{(j)}, \mathrm{h}_{\mathrm{ast}}^{(j)}, \mathrm{h}_{\mathrm{comment}}^{(j)}\}$ for $j \neq i$ as its negative set.

For the multi-modal contrastive loss calculations, we employ the NT-Xent loss function (Normalized Temperature-scaled Cross Entropy Loss), which maximizes similarity for positive pairs while ensuring distinctiveness for negative pairs. For each positive pair $(\mathrm{h}_m, \mathrm{h}_n)$ across modalities, the contrastive loss is defined as:

$$\mathcal{L}_{\mathrm{contrast}} = -\log \frac{\exp\big(\mathrm{sim}(\mathrm{h}_m, \mathrm{h}_n)/\tau\big)}{\sum_{a \in \mathcal{N}(\mathrm{h}_m)} \exp\big(\mathrm{sim}(\mathrm{h}_m, \mathrm{h}_a)/\tau\big)}, \tag{2}$$

where $\mathrm{sim}(\cdot,\cdot)$ denotes cosine similarity, τ is the temperature, and $\mathcal{N}(\mathrm{h}_m)$ is the set of negatives for h_m. We then combine this contrastive term with our classification loss $\mathcal{L}_{\mathrm{cls}}$, which is the average binary cross-entropy over the batch:

$$\mathcal{L}_{\mathrm{cls}} = -\frac{1}{B} \sum_{i=1}^{B} \big[y_i \log \hat{p}_i + (1 - y_i) \log(1 - \hat{p}_i) \big], \tag{3}$$

where B is the batch size, $y_i \in \{0, 1\}$ is the ground-truth label for the i-th sample, and $\hat{p}_i$ is the model's predicted probability that the sample is vulnerable. The total loss is thus expressed as a convex combination of the two terms:

$$\mathcal{L}_{\mathrm{total}} = (1 - \lambda)\, \mathcal{L}_{\mathrm{cls}} + \lambda\, \mathcal{L}_{\mathrm{contrast}}. \tag{4}$$

Experiments show that setting $\lambda = 0.3$ yields the best F1 performance, and this value is used in all our reported results. The model learns multi-modal embeddings that balance shared semantics with modality-specific distinctions by applying this weighted objective across code–AST, code–comment, and AST–comment pairs.

4 Experimental setup

4.1 Research questions

To evaluate MCL-VD, we aim to answer the following five research questions:

RQ1: How effective is MCL-VD in vulnerability detection compared to existing methods?

To answer this question, we compare MCL-VD with eight state-of-the-art approaches for vulnerability detection. The aim is to evaluate its accuracy, efficiency, and applicability across various vulnerabilities.

RQ2: How does multi-modal integration of code, AST, and comments impact the performance of MCL-VD?

To address this question, we evaluate how the combination of source code, AST, and comments as separate modalities affects the performance of MCL-VD. We aim to measure the improvements in vulnerability detection accuracy, efficiency, and robustness when integrating these modalities.

RQ3: How does multi-modal contrastive learning enhance the alignment and differentiation of embeddings in MCL-VD?

Multi-modal contrastive learning is a cornerstone of MCL-VD, enabling the model to align representations across modalities while maintaining their distinctiveness. This RQ investigates how contrastive learning improves embeddings' semantic alignment and differentiation, enhancing the model's robustness in detecting vulnerabilities.

RQ4: What role does LoRA-based fine-tuning play in optimizing MCL-VD for vulnerability detection?

LoRA is employed in MCL-VD to enable efficient fine-tuning while preserving pre-trained knowledge and reducing model consumption. This RQ examines the impact of LoRA-based fine-tuning on the model's performance, focusing on its role in lowering model resource usage, such as reducing trainable parameters and accelerating training time, while maintaining detection accuracy.

RQ5: How does supplementing incomplete comments during preprocessing improve the performance of MCL-VD?

In MCL-VD, the source code in the datasets often contains incomplete or missing comments. To address this limitation, we supplement the missing comments using GPT-4o-mini during the preprocessing phase. This RQ explores how enriching the comment modality affects the model's performance in vulnerability detection.

4.2 Datasets

To evaluate the effectiveness of MCL-VD, we conduct experiments on three widely used datasets for vulnerability detection: Devign (Zhou et al. 2019), Reveal (Chakraborty et al. 2021), and Big-Vul (Fan et al. 2020). The details of these datasets are as follows:

- **Devign** (Zhou et al. 2019): This dataset comprises functions collected from FFmpeg and QEMU, including approximately 12k vulnerable and about 14k

non-vulnerable functions. It is a relatively balanced dataset, with a substantial number of both vulnerable and non-vulnerable functions.

- **Reveal** (Chakraborty et al. 2021): The Reveal dataset, collected from the Linux Debian Kernel and Chromium, contains around 1.6k vulnerable functions and about 16k non-vulnerable functions. This dataset is imbalanced, with a significant disparity between the number of vulnerable and non-vulnerable functions.
- **Big-Vul** (Fan et al. 2020): The Big-Vul dataset, collected by Fan et al., consists of approximately 10k vulnerable functions and around 176k non-vulnerable functions. Like the other datasets, Big-Vul is imbalanced, with a notable disparity between the number of vulnerable and non-vulnerable functions.

Table 1 presents the details of the three datasets used in our experiments. Due to the length of some code snippets in the datasets, which made it impossible to generate their ASTs, we excluded those code snippets. The datasets include vulnerable and non-vulnerable code, with the proportion of vulnerable code varying across the datasets. Big-Vul contains 187,318 samples, with 5.5% being vulnerable and 94.5% non-vulnerable. Reveal consists of 18,169 samples, with 9.16% vulnerable code, and Devign includes 26,621 samples with 45.5% vulnerable code.

4.3 Baselines

We compare our proposed approach against eight state-of-the-art baselines in vulnerability detection, covering a variety of models from traditional deep learning methods to transformer-based architectures:

- **SySeVR** (Li et al. 2021): A syntax-based vulnerability detection model that uses code representations and syntactic features to classify vulnerable and non-vulnerable code.
- **VulDeePecker** (Li et al. 2018): A CNN-based model designed to detect vulnerabilities by learning deep representations of source code sequences.
- **Devign** (Zhou et al. 2019): A graph-based model for vulnerability detection that leverages graph neural networks to capture structural dependencies in code.
- **ReGVD** (Nguyen et al. 2022): A transformer-based approach tailored for vulnerability detection, primarily designed to capture semantic and structural context in source code.
- **CodeBERT** (Feng et al. 2020): A pre-trained transformer model that learns code and natural language representations, commonly used for code-related tasks, including vulnerability detection.
- **GraphCodeBERT** (Guo et al. 2020): A transformer-based model pre-trained on code and graphs to enhance code understanding by incorporating data flow and structural information.

Table 1 Statistics of the datasets

Dataset	Samples	Non-vul	Vul	Non-vul (%)	Vul (%)
Big-Vul	187,318	176,951	10,367	94.5%	5.5%
Reveal	18,169	16,505	1,664	90.8%	9.2%
Devign	26,621	14,493	12,128	54.5%	45.5%

- **ContraBERT** (Liu et al. 2023): Introduces nine distinct data augmentation operators, encompassing both simple and complex transformations, applicable to programming language (PL) and natural language (NL) data. These operators are employed to generate diverse model variants. Additionally, it further pre-trains existing models using masked language modeling (MLM) and contrastive learning on both original and augmented samples to enhance robustness.
- **SCL-CVD** (Wang et al. 2024): Supervised Contrastive Learning for Code Vulnerability Detection via GraphCodeBERT. This method represents each function as a flattened AST graph along with its code tokens, applies a single NT-Xent contrastive loss over graph–code pairs, and employs R-Drop regularization to improve robustness.

These baselines represent various state-of-the-art approaches in vulnerability detection, providing a comprehensive comparison across diverse approaches. SySeVR and VulDeePecker represent earlier models that utilize syntax-based and convolutional neural network (CNN) techniques to extract syntactic features for vulnerability identification. Devign, GraphCodeBERT, and SCL-CVD adopt graph-based techniques, emphasizing the importance of capturing code structure and data flow. Notably, SCL-CVD integrates supervised contrastive learning applied to graph–code pairs, further enhancing its analytical capabilities. Transformer-based models, including ReGVD, CodeBERT, and ContraBERT offer advanced solutions for vulnerability detection. ReGVD combines semantic and structural analysis to address vulnerabilities comprehensively, while CodeBERT serves as a versatile pre-trained model for general-purpose code understanding. ContraBERT leverages contrastive learning and data augmentation techniques to refine representation quality, improving the model's robustness and accuracy. These baselines ensure a robust comparison across different methodologies, demonstrating the effectiveness of the proposed multi-modal contrastive learning approach in advancing vulnerability detection.

4.4 Evaluation metrics

In this study, we employed the following four widely used evaluation metrics to assess the performance of MCL-VD:

True positive (TP) TP refers to the instances where the model correctly predicts the presence of vulnerabilities in the code. These are the cases where the model identifies vulnerabilities in the code, ensuring that genuine issues are not overlooked.

True negative (TN) TN represents the cases where the model correctly identifies code as free from vulnerabilities. In other words, when the model assesses code that does not contain vulnerabilities, it classifies it correctly as benign, preventing unnecessary alerts.

False positive (FP): FP occurs when the model incorrectly classifies code without vulnerabilities as containing vulnerabilities. This results in the model raising false alarms, wasting time and resources for developers who investigate non-issues.

False negative (FN): FN represents the instances where the model fails to detect a vulnerability in the code. These missed vulnerabilities are critical because they can go unnoticed.

Accuracy quantifies the proportion of correct predictions made by the model out of all predictions. It is a general measure of model performance, though it may not be sufficient in cases where class imbalance is present. The formula is defined as:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{5}$$

Precision measures the proportion of correctly predicted vulnerable instances among all instances predicted as vulnerable by the model. Precision is especially crucial when the cost of false positives is high, as it reflects the model's ability to avoid incorrectly labeling benign instances as vulnerable. The formula is:

$$\text{Precision} = \frac{TP}{TP + FP} \tag{6}$$

Recall refers to the proportion of actual vulnerable instances correctly identified by the model. It is crucial when the cost of false negatives is high, as it reflects the model's ability to detect all vulnerable instances. The formula for recall is:

$$\text{Recall} = \frac{TP}{TP + FN} \tag{7}$$

F1-score provides a balanced measure of both Precision and Recall by calculating their harmonic mean. It is particularly valuable in cases of class imbalance, where the model must navigate the trade-off between false positives and false negatives. The F1-score is calculated as:

$$\text{F1-score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{8}$$

These metrics collectively offer a comprehensive evaluation of the model's ability to detect vulnerabilities across different datasets. In this study, the F1-score is particularly interesting as it provides a robust assessment of the model's performance, especially in imbalanced class distributions.

4.5 Experimental settings

In terms of model reproduction, we followed all baseline experimental settings. The datasets were divided into training, validation, and test sets in an 8:1:1 ratio, following established practices (Li et al. 2021; Feng et al. 2020). This partitioning was uniformly applied across all baselines and our proposed model. For the Devign dataset,

where pre-processed code was unavailable, we replicated the methodology described in Reveal (Chakraborty et al. 2021) to ensure consistency.

GraphCodeBERT was employed as the pre-trained backbone with a maximum input sequence length of 512 tokens in our experiments. If the input consisted of two modalities, the tokens from each modality were evenly distributed with a 1:1 ratio. The tokens were distributed in a 2:1:1 ratio for three-modal inputs, with code, comments, and AST tokens occupying the respective proportions. The learning rate was set to 5×10^{-5}, and gradient accumulation was utilized to simulate an adequate batch size of 16 with mini-batches of size 2. For LoRA, we set the rank $r = 16$ and applied a dropout rate of 0.1 to regularize the model and prevent overfitting. Training was performed on an NVIDIA GeForce RTX 4090 GPU, leveraging mixed precision to improve computational efficiency and reduce memory usage. The model was trained for up to 50 epochs, with early stopping triggered if the validation F1-score did not improve for 10 consecutive epochs.

5 Result analysis

5.1 RQ1: effectiveness of MCL-VD

To address this research question, we compared MCL-VD against eight state-of-the-art methods, including graph-based, sequence-based, and LLM-based. As shown in Table 2, MCL-VD demonstrates superior performance across all three datasets, i.e., Devign, Reveal, and Big-Vul, outperforming the best existing methods in most evaluated metrics. Specifically, on the Devign dataset, MCL-VD improves the F1-score by **8.49%** compared to the best baseline SCL-CVD. On the Reveal dataset, the F1-score increases by **4.86%** compared to the best-performing baseline. The largest gains are observed on the Big-Vul dataset, where MCL-VD achieves an F1-score improvement of **17.26%**, demonstrating the performance improvements of the proposed method.

In terms of performance categories, pre-trained models such as GraphCodeBERT and ContraBERT show better results compared to traditional sequence-based methods (e.g., VulDeePecker and SySeVR) and graph-based methods (e.g., SCL-CVD and ReGVD). However, MCL-VD consistently outperforms these models, confirming multi-modal integration and contrastive learning advantages. Unlike pre-trained models that primarily leverage semantic and syntactic information, MCL-VD effectively integrates diverse modalities such as code, AST, and comments. This allows the model to capture rich structural, contextual, and annotation-based features critical for accurate vulnerability detection.

The improvements brought by MCL-VD can be attributed to several key factors. First, multi-modal embeddings leverage complementary information from source code, AST, and comments, enabling better feature representation. Second, applying contrastive learning ensures effective alignment and differentiation of embeddings across modalities, improving generalization to unseen code structures. Finally, LoRA fine-tuning enhances the model's efficiency by optimizing specific parameters without overfitting.

Table 2 Comparison results between MCL-VD and baselines on the three datasets in vulnerability detection

Metrics(%) / Dataset	Devign				Reveal				Big-Vul			
Baseline	Accuracy	Precision	Recall	F1	Accuracy	Precision	Recall	F1	Accuracy	Precision	Recall	F1
SySeVR	48.59	47.08	60.02	52.77	73.21	43.56	27.84	33.97	90.10	30.91	14.08	19.34
VulDeePecker	50.12	47.89	33.34	39.31	78.51	20.63	14.59	17.09	81.19	38.44	12.75	19.15
Devign	60.61	58.56	48.76	53.22	87.49	31.55	21.77	25.76	92.78	30.61	15.96	20.98
ReGVD	62.15	61.72	46.13	52.80	91.16	53.33	25.33	34.35	94.75	64.57	13.65	22.54
CodeBERT	59.77	57.98	56.59	57.28	91.09	59.38	31.67	41.30	64.25	56.74	54.58	55.64
GraphCodeBERT	63.26	59.81	55.62	57.64	90.90	**60.22**	23.55	33.86	95.38	69.36	32.08	43.86
ContraBERT	65.70	**64.53**	56.34	60.16	90.80	34.04	**60.09**	43.46	94.89	67.66	67.63	67.64
SCL-CVD	63.58	60.55	64.76	62.58	91.42	46.70	47.53	47.11	95.83	62.26	35.26	45.02
MCL-VD	**66.20**	55.67	**86.97**	**67.89**	**91.53**	42.91	58.22	**49.40**	**96.82**	**91.54**	**69.98**	**79.32**

Answer to RQ1: The proposed approach, MCL-VD, significantly outperforms all baselines in vulnerability detection across all datasets. It achieves F1-score improvements ranging from 12.85% to 17.26% over the best-performing baselines. These results demonstrate the effectiveness of our method.

5.2 RQ2: Impact of multi-modal integration

To investigate the impact of multi-modal integration, we conducted an ablation study using different combinations of modalities across three datasets: Devign, Reveal, and Big-Vul. Specifically, MCL-A represents using only the AST, MCL-C represents using only the source code, and MCL-M represents using only the comments. MCL-A-C represents combining code and AST, MCL-A-M represents combining AST and comments, and MCL-C-M represents combining code and comments. Finally, MCL-all represents using all three modalities: code, AST, and comments.

The results shown in Table 3 demonstrate that integrating multiple modalities significantly improves performance in terms of F1-score, with all multi-modal configurations outperforming single-modality models.

On the Devign dataset, incorporating all three modalities yields substantial relative improvements compared to both single- and double-modal models. Relative to MCL-A, MCL-M, and MCL-C, the MCL-all configuration achieves F1-score increases of 6.77%, 5.86%, and 6.31%, respectively. Furthermore, when compared with the double-modal baselines MCL-A-C, MCL-A-M, and MCL-C-M, MCL-all outperforms them by around 3.10%, 5.29%, and 2.03%, respectively.

On the Reveal dataset, the advantages of multi-modal integration are more pronounced. In comparison to the single-modal baselines MCL-A, MCL-C, and MCL-M, the MCL-all's F1-score improves by 24.96%, 19.38%, and 17.59%, respectively. Similarly, against the double-modal configurations MCL-A-C, MCL-A-M, and MCL-C-M, MCL-all achieves relative gains of about 7.32%, 3.91%, and 5.67%, respectively.

On the Big-Vul dataset, the triple-modal approach also demonstrates robust performance gains. Compared to the single-modal models MCL-A, MCL-C, and MCL-M, MCL-all's F1-score increases by 11.61%, 11.14%, and 8.64%, respectively. Against the double-modal approaches MCL-A-C, MCL-A-M, and MCL-C-M, MCL-all achieves relative improvements of around 6.45%, 5.65%, and 4.77%, respectively.

The performance improvement achieved by multimodality can be attributed to the complementarity of different information sources. The code provides syntactic structure, the AST captures structural relationships within the code, and the comments offer additional contextual information. Combining these modalities allows the model to leverage all available information, leading to better performance in detecting vulnerabilities.

Table 3 The impact of different input modalities on the performance of MCL-VD on the three datasets

Metrics(%) / Dataset / Method	Devign				Reveal				Big-Vul			
Method	Accuracy	Precision	Recall	F1	Accuracy	Precision	Recall	F1	Accuracy	Precision	Recall	F1
MCL-A	60.17	47.80	94.97	63.59	78.12	30.73	55.40	39.53	95.40	82.71	62.30	71.07
MCL-C	61.25	49.86	88.79	63.86	88.25	33.73	53.52	41.38	96.52	85.16	61.43	71.37
MCL-M	57.27	47.48	**98.76**	64.13	85.57	34.01	54.93	42.01	96.05	84.91	64.03	73.01
MCL-A-C	63.71	50.85	93.40	65.85	88.90	41.51	51.64	46.03	95.51	85.13	66.25	74.51
MCL-A-M	61.29	48.83	94.89	64.48	89.84	42.18	54.46	47.54	95.13	88.17	65.38	75.08
MCL-C-M	62.71	53.82	87.14	66.54	86.74	41.22	53.99	46.75	94.83	89.37	65.67	75.71
MCL-all	**66.20**	**55.67**	86.97	**67.89**	**91.53**	**42.91**	**58.22**	**49.40**	**96.82**	**91.54**	**69.98**	**79.32**

Answer to RQ2: Multi-modal integration significantly improves performance across all datasets. The combination of code, AST, and comments consistently achieves the highest F1-scores, surpassing both single-modal and double-modal combinations. This demonstrates that combining multiple modalities enhances the model's ability to detect vulnerabilities.

5.3 RQ3: Influence of multi-modal contrastive learning

To address this research question, we investigated the influence of multi-modal contrastive learning by comparing the performance of MCL-VD with and without contrastive learning (MCL-C and MCL-wo-C) across three datasets: Devign, Reveal, and Big-Vul. As shown in Fig. 3, applying contrastive learning consistently improved the model's performance across all metrics on three datasets.

Specifically, on the Devign dataset, the F1-score improved by 8.05%, when contrastive learning was applied. Additionally, accuracy increased by 4.78%, and recall showed a significant improvement of 29.77%. This demonstrates that contrastive learning effectively enhances the model's ability to identify vulnerabilities.

On the Reveal dataset, the F1-score increased by 7.09%, while accuracy improved by 3.96%. Precision also saw a modest increase of 2.91%, and recall improved by 6.92%, confirming the benefits of contrastive learning in enhancing both precision and recall.

The Big-Vul dataset exhibited the largest performance gains. The F1-score improved by 3.27%, and accuracy showed a 1.25% increase. Precision and recall also saw improvements of 1.94% and 4.20%, respectively. These results suggest that contrastive learning has a powerful impact on performance in large-scale datasets.

To investigate the impact of multi-modal contrastive learning, we utilized t-distributed stochastic neighbor embedding (t-SNE) (Van der Maaten and Hinton 2008) to visualize the effect of contrastive learning on the model's ability to distinguish between vulnerable and benign code. t-SNE allows us to map high-dimensional feature spaces into a two-dimensional representation, providing a clear visual understanding of how the model separates the two types of code. In Fig. 4, the purple and red points represent benign and vulnerable code, respectively.

We randomly selected 1000 samples for each dataset, proportional to the vulnerability distribution, to ensure a balanced representation in the visualizations. The scatter plots clearly show the differences in the separability of the two classes with and without contrastive learning.

In the case of the Devign dataset (Fig. 4a), when contrastive learning is not applied, the vector representations of benign and vulnerable code overlap significantly, indicating that the model struggles to distinguish between the two classes. However, when contrastive learning is applied, the separation between the two classes becomes clearer, with the two categories becoming more distinct, and the samples of the same class are more tightly clustered.

A similar trend is observed in the Reveal (Fig. 4b) and Big-Vul (Fig. 4c) datasets. Without contrastive learning, the embeddings of benign and vulnerable code are

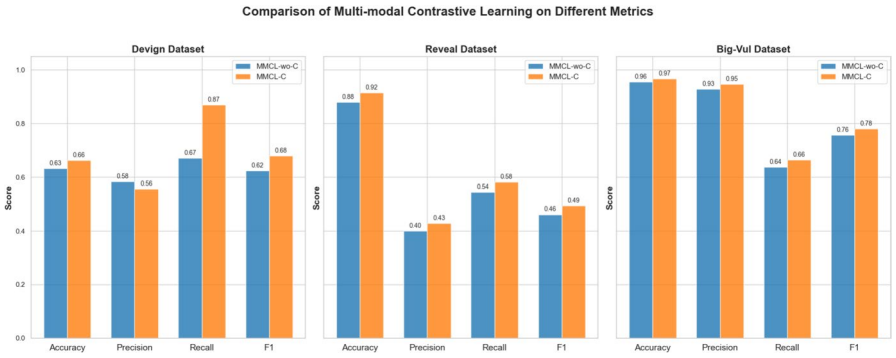


Fig. 3 The impact of multi-modal contrastive learning on the performance of MCL-VD on the three datasets

mixed, making it difficult to distinguish between them. In contrast, applying contrastive learning leads to a more defined separation, where the vulnerable and benign code samples are more distinguishable and better separated in the feature space.

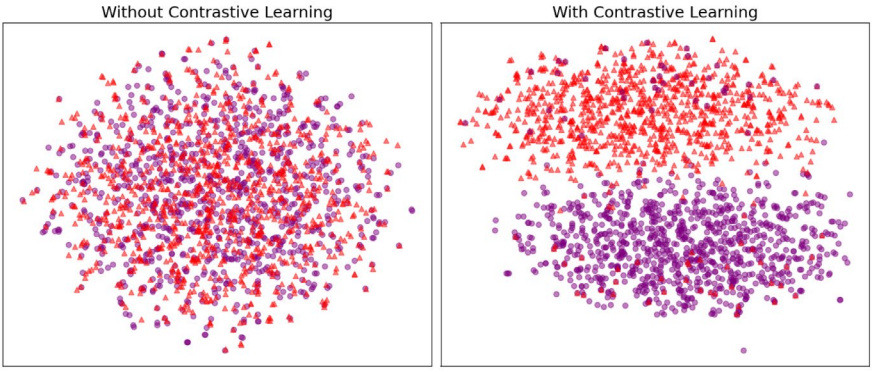
These visualizations demonstrate that contrastive learning enhances the model's ability to differentiate between vulnerable and benign code. Contrastive learning improves the overall model performance by forcing the model to learn embeddings that better separate these two classes, making it more effective in vulnerability detection.

> **Answer to RQ3:** The application of multi-modal contrastive learning significantly improves the performance of MCL-VD across all datasets. The F1-score improvements range from 3.27% to 8.05%, with similar gains observed in accuracy, precision, and recall. These results demonstrate the effectiveness of contrastive learning in enhancing the model's ability to detect vulnerabilities.

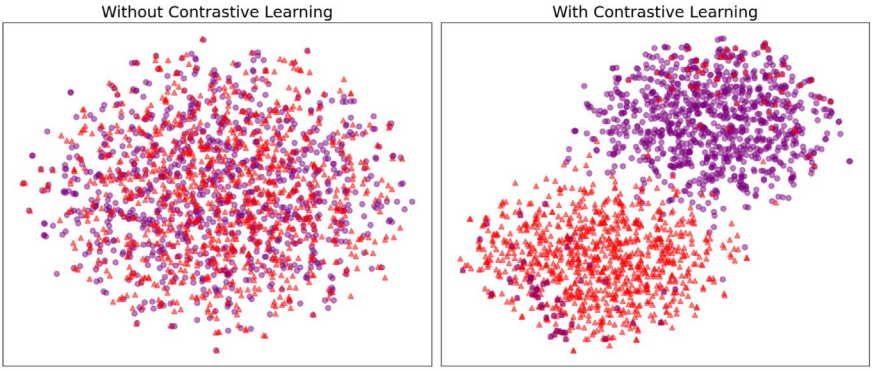
5.4 RQ4: LoRA in fine-tuning efficiency

To investigate the impact of LoRA on model performance, we compared the MCL-L model (which incorporates LoRA) with the MCL-wo-L model (which does not). The comparison was conducted across three datasets: Devign, Reveal, and Big-Vul. As shown in Table 4, the results demonstrate the efficiency gains achieved by LoRA in terms of trainable parameters and training time, without compromising the model's ability to detect vulnerabilities.

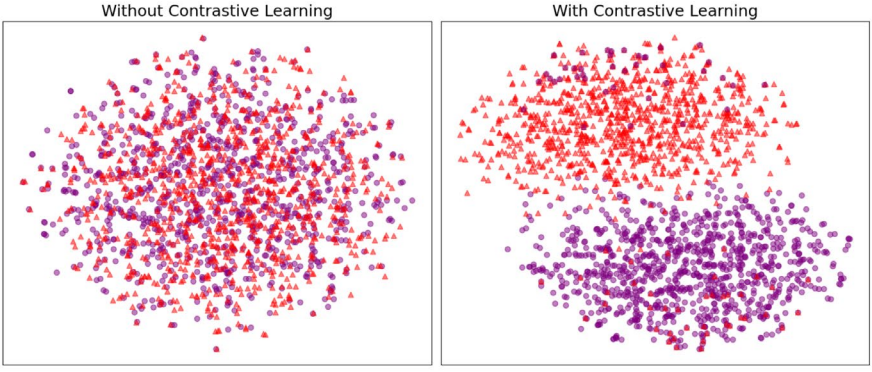
On the Devign dataset, MCL-L reduces trainable parameters from 125.83 million to 1.77 million, and the training time per epoch decreases from 24 minutes to 18 minutes. Similarly, training time decreases from 18 to 14 minutes on the Reveal dataset. On the Big-Vul dataset, the training time drops from 120 minutes to 89 minutes, with trainable parameters remaining reduced across all datasets. Despite these significant reductions in trainable parameters and training time, the model's perfor-



(a) t-SNE on Devign



(b) t-SNE on Reveal



(c) t-SNE on Big-Vul

Fig. 4 t-SNE visualization on Devign, Reveal, and Big-Vul datasets

Table 4 The impact of LoRA on the performance of MCL-VD

Metrics / Dataset / Method	Devign			Reveal			Big-Vul		
Method	Trainable Params (M)	Time (min)	F1(%)	Trainable Params (M)	Time (min)	F1(%)	Trainable Params (M)	Time (min)	F1(%)
MCL-wo-L	125.83M	24	67.64	125.83M	18	49.26	125.83M	120	79.25
MCL-L	1.77M	18	67.89	1.77M	14	49.40	1.77M	89	79.32

mance remains unchanged, with MCL-L achieving similar F1-scores as MCL-wo-L across all three datasets.

The results highlight the effectiveness of LoRA in reducing the complexity of the model while maintaining its performance. By applying low-rank updates to only a subset of the model's parameters, LoRA reduces the number of trainable parameters and the computational cost associated with training, allowing the model to adapt more efficiently without compromising performance. This leads to a more efficient model that retains its vulnerability detection capabilities while requiring fewer resources for training.

Answer to RQ4: The proposed approach significantly reduces the number of trainable parameters and training time while maintaining the same level of performance. This demonstrates that LoRA enables more efficient model adaptation without sacrificing detection accuracy.

5.5 RQ5: Effect of comment supplementation in preprocessing

To investigate the effect of supplementing incomplete comments, we compared the performance of MCL-VD with and without comment supplementation across three datasets: Devign, Reveal, and Big-Vul. In the absence of comments, the model is trained with the original datasets containing incomplete or missing comments. In contrast, the enriched dataset uses GPT-4o-mini to supplement the missing comments, providing more complete context and improving the quality of the comment modality.

As shown in Table 5, the results indicate that supplementing comments with GPT-4o-mini generally enhances the model's performance across the evaluated datasets. Compared to MCL-wo-S, the MCL-S model trained with supplemented comments achieves notable relative improvements in most metrics. On the Devign dataset, MCL-S provides approximately a 6.45% relative increase in accuracy, a 16.46% increase in recall, and a 5.88% increase in F1-score, though it experiences a slight 0.91% relative decrease in precision. The Reveal dataset shows a similar pattern, with about a 3.76% improvement in accuracy, a 1.18% improvement in precision, a 22.31% improvement in recall, and a 10.11% improvement in F1-score. For the Big-Vul dataset, MCL-S demonstrates relative improvements of roughly 2.16% in accuracy, 4.49% in precision, 3.67% in recall, and 4.03% in F1-score. These find-

Table 5 Comparison of performance with and without comment supplementation across three datasets

Dataset	Method	Accuracy (%)	Precision (%)	Recall (%)	F1-score (%)
Devign	MCL-wo-S	62.19	56.18	74.68	64.12
	MCL-S	66.20	55.67	86.97	67.89
Reveal	MCL-wo-S	88.21	42.41	47.60	44.86
	MCL-S	91.53	42.91	58.22	49.40
Big-Vul	MCL-wo-S	94.77	87.61	67.50	76.25
	MCL-S	96.82	91.54	69.98	79.32

ings suggest that comment supplementation can be a valuable strategy for enhancing multi-modal vulnerability detection, even though the magnitude of improvement may vary across different datasets and metrics.

Answer to RQ5: Supplementing incomplete comments during preprocessing significantly improves the performance of MCL-VD across all datasets. The enriched comments help the model to capture more context, resulting in better accuracy, precision, recall, and F1-score, thus enhancing the model's vulnerability detection capabilities.

6 Discussion

6.1 The impact of different lora parameter settings on model performance

In this section, we explore the effect of varying the LoRA parameter, specifically the rank r, on the performance of our model. LoRA introduces a low-rank matrix to modify the model's weights, enhancing computational efficiency while maintaining model performance. We experiment with several values of r, including $r = 4, 8, 16, 32, 64$, to evaluate how different ranks impact our model's performance. All experiments are conducted on the Devign dataset.

As shown in Fig. 5, we observed the impact of varying LoRA ranks on the model's performance across several metrics. Specifically, we measured accuracy, precision, recall, and F1-score for each rank r value. The experimental results indicate that the rank $r = 16$ strikes the best balance between model performance and computational efficiency.

Our experiments indicate that while increasing the rank to $r = 32$ and $r = 64$ resulted in higher computational costs, the performance improvements were marginal and did not surpass the results achieved with $r = 16$. Conversely, smaller ranks, such as $r = 4$ and $r = 8$, provided a good trade-off between performance and efficiency, though with a slight drop in performance. Therefore, our findings suggest that $r = 16$ offers the optimal configuration, balancing performance and computational efficiency, making it the ideal choice for vulnerability detection without incurring unnecessary computational overhead. This outcome is likely because $r = 16$ strikes a sufficient balance between model complexity and the data's capacity to capture

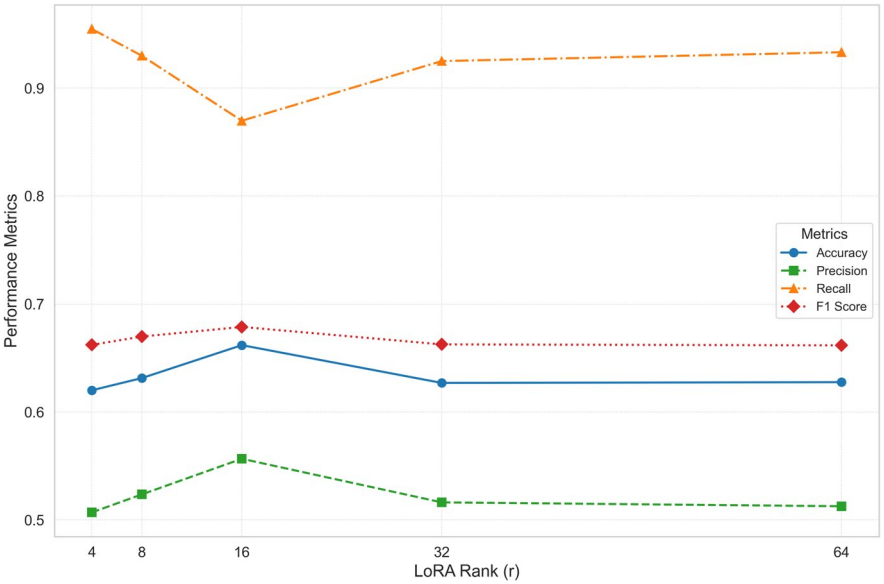


Fig. 5 Impact of different ranks on the performance of MCL-VD

meaningful features. At the same time, higher ranks add more trainable parameters, which, although they increase the model's expressiveness, lead to diminishing returns in performance but significantly raise the computational cost.

6.2 The effect of different pre-trained models on performance

In our experiments, we evaluate four pre-trained models: CodeBERT, GraphCodeBERT, CodeT5, and UniXcoder (Wang et al. 2021; Guo et al. 2022, 2020; Feng et al. 2020). Each model has its unique strengths in handling different types of code representations. Each model has unique strengths in handling different code representations. For models without a native graph encoder (CodeBERT, CodeT5, UniXcoder), we serialize the AST into a token sequence using the Structure-Based Traversal (SBT) algorithm (Wu et al. 2021), then concatenate it as the input sequence. GraphCodeBERT, by contrast, consumes the AST directly via its built-in graph module.

We assessed the performance of these pre-trained models on the Devign dataset, measuring key evaluation metrics including accuracy, precision, recall, and F1-score. The results are summarized in Table 6, where we compare the performance of these models across various metrics.

As shown in Table 6, GraphCodeBERT consistently outperforms the other models across key metrics: F1-score, accuracy, and precision. This performance advantage is likely attributed to its ability to integrate semantic and structural code information, essential for effectively detecting vulnerabilities. CodeBERT, while still performing well, does not achieve the same level of performance, possibly due to its limited ability to capture graph-based contextual information compared to GraphCodeBERT.

Table 6 Performance of different pre-trained models on vulnerability detection

Model	Accuracy (%)	Precision (%)	Recall (%)	F1-score (%)
GraphCodeBERT	66.20	55.67	86.97	67.89
CodeBERT	62.45	51.54	92.58	66.21
CodeT5	55.71	47.61	95.14	63.36
UniXcoder	62.19	53.13	91.18	67.30

Table 7 Efficiency and performance comparison: Full-Parameter vs. LoRA fine-tuning

Method	Devign			Reveal			Big-Vul		
	F1 (%)	GPU Mem (GB)	Time (min)	F1 (%)	GPU Mem (GB)	Time (min)	F1 (%)	GPU Mem (GB)	Time (min)
Full FT	67.79	12.0	24	49.43	11.5	18	79.22	12.3	120
LoRA	67.89	1.8	18	49.40	1.7	14	79.32	1.9	89

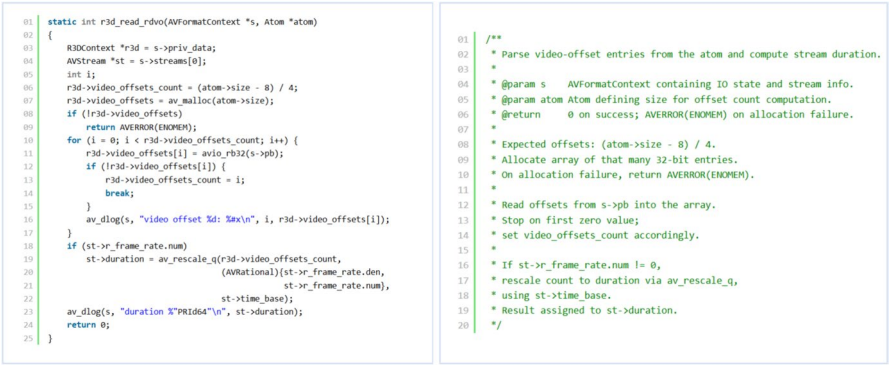
CodeT5 and UniXcoder also show competitive performance but slightly trail GraphCodeBERT in all metrics. CodeT5 demonstrates solid performance, particularly in tasks requiring code-to-code translation. However, it appears less effective in capturing the structural features necessary for vulnerability detection. UniXcoder, with its cross-lingual capabilities, shows promising results in general code representation. Still, it falls short in precision and recall on the Devign dataset compared to GraphCodeBERT, indicating that its performance is sensitive to dataset characteristics.

GraphCodeBERT's superior performance can be attributed to its integration of both semantic and structural information. By incorporating graph-based representations, GraphCodeBERT can capture hierarchical relationships and syntactic dependencies within the code, making it particularly effective in identifying vulnerabilities that rely on understanding complex code structures. This ability to capture high-level semantic meaning and low-level syntactic patterns gives GraphCodeBERT a significant edge in vulnerability detection tasks.

6.3 How LoRA compares to full-parameter fine-tuning in efficiency and performance

LoRA was originally proposed to reduce the memory footprint when fine-tuning large models (7B+ parameters). However, our experiments show that applying modality-specific LoRA adapters to a mid-sized model like GraphCodeBERT can achieve substantial resource savings without compromising vulnerability detection performance. We compare standard full-parameter fine-tuning with LoRA fine-tuning under identical hyperparameter settings.

As Table 7 shows, LoRA fine-tuning reduces GPU memory requirements by approximately 85% and cuts per-epoch training time by about 25%, while maintaining F1-scores within 0.5 percentage points of full-parameter tuning on all three datasets. These savings arise because LoRA replaces full-gradient updates with low-rank adapter updates, capturing essential modality-specific adjustments with far fewer trainable weights. Moreover, by assigning separate adapters to code, AST,



```
static int r3d_read_rdvo(AVFormatContext *s, Atom *atom)
{
    R3DContext *r3d = s->priv_data;
    AVStream *st = s->streams[0];
    int i;
    r3d->video_offsets_count = (atom->size - 8) / 4;
    r3d->video_offsets = av_malloc(atom->size);
    if (!r3d->video_offsets)
        return AVERROR(ENOMEM);
    for (i = 0; i < r3d->video_offsets_count; i++) {
        r3d->video_offsets[i] = avio_rb32(s->pb);
        if (!r3d->video_offsets[i]) {
            r3d->video_offsets_count = i;
            break;
        }
        av_dlog(s, "video offset %d: %#x\n", i, r3d->video_offsets[i]);
    }
    if (st->r_frame_rate.num)
        st->duration = av_rescale_q(r3d->video_offsets_count,
                                    (AVRational){st->r_frame_rate.den,
                                                 st->r_frame_rate.num},
                                    st->time_base);
    av_dlog(s, "duration %"PRId64"\n", st->duration);
    return 0;
}
```

```
/**
 * Parse video-offset entries from the atom and compute stream duration.
 *
 * @param s    AVFormatContext containing IO state and stream info.
 * @param atom Atom defining size for offset count computation.
 * @return     0 on success; AVERROR(ENOMEM) on allocation failure.
 *
 * Expected offsets: (atom->size - 8) / 4.
 * Allocate array of that many 32-bit entries.
 * On allocation failure, return AVERROR(ENOMEM).
 *
 * Read offsets from s->pb into the array.
 * Stop on first zero value;
 * set video_offsets_count accordingly.
 *
 * If st->r_frame_rate.num != 0,
 * rescale count to duration via av_rescale_q,
 * using st->time_base.
 * Result assigned to st->duration.
 */
```

Fig. 6 Example of source code and GPT-4o-mini–generated comment

Table 8 Human evaluation of generated comment quality (1 = worst, 5 = best)

Metric	Mean Score	Std. Dev.
Relevance	4.2	0.5
Clarity	4.0	0.6

and comments, LoRA enhances feature separation and reduces overfitting risk compared to monolithic full-parameter updates. This demonstrates that LoRA provides an effective and efficient fine-tuning strategy for multi-modal vulnerability detection in resource-constrained settings.

6.4 Generated-comment quality assessment

To illustrate the style and content of GPT-4o-mini's automatic annotations, Fig. 6 shows one representative example: the source code on the left and the generated comment on the right.

We then conducted human evaluation on 50 randomly sampled samples in Devign. Three senior software engineers rated each generated comment based on two criteria—Relevance to the code's functionality and Clarity of the comment—using a 1–5 score. Table 8 summarizes the results.

These high scores demonstrate that GPT-4o-mini's annotations reliably capture both the semantic intent of the code and the readability of the comments, supporting the validity of MCL-VD.

6.5 How do different contrastive learning loss functions impact performance?

To investigate the impact of different contrastive learning loss functions on the performance of our model, we conducted experiments with several commonly used loss functions: NT-Xent Loss (our method), InfoNCE Loss, Triplet Loss, and SimCLR Loss. In our experiments, we evaluated the performance of these contrastive learning formulations on the Devign dataset, measuring accuracy, precision, recall, and F1-score. The results of these experiments are presented in Table 9.

Table 9 Performance comparison of different contrastive learning loss functions

Loss Function	Accuracy (%)	Precision (%)	Recall (%)	F1-score (%)
NT-Xent Loss (Ours)	66.20	55.67	86.97	67.89
InfoNCE Loss	64.24	54.10	85.20	66.10
Triplet Loss	63.21	53.01	84.50	65.30
SimCLR Loss	65.15	54.85	85.92	66.75

As shown in Table 9, NT-Xent loss method consistently outperforms the other contrastive learning formulations across all evaluated metrics. Specifically, it achieves the highest F1-score of 67.89%, as well as the best precision and recall values. The InfoNCE Loss, Triplet Loss, and SimCLR Loss exhibit comparable performance, with slight variations across the metrics. However, none of these alternatives surpassed the performance of our NT-Xent loss approach, confirming the efficacy of our approach in improving multi-modal representation learning for vulnerability detection.

The superior performance of our method can be attributed to the way the NT-Xent loss ensures that each modality (code, AST, and comments) is treated separately, yet aligned with the other modalities. This helps the model capture modality-specific features while maintaining the shared semantics necessary for accurate vulnerability detection.

6.6 Effectiveness in distinguishing similar code snippets

To investigate the effectiveness of our method in distinguishing similar code snippets, we conducted experiments. Figure 7 presents a case study of a CWE-17 Denial of Service vulnerability from the Big-Vul dataset. Snippet (a) implements DCCP connection tracking but never verifies that the corresponding kernel module (`nf_conntrack_dccp`) is loaded before use; the code immediately proceeds to build the DCCP attributes, which can cause a crash or DoS if the module is absent. Snippet (b) handles SCTP connection tracking instead, and it correctly checks `ATTR_SCTP_STATE` before proceeding, so it cannot trigger the same failure.

Traditional methods such as GraphCodeBERT and ReGVD both fail to detect the missing kernel-module check in snippet (a), incorrectly labeling it as safe and resulting in a false negative. However, our approach correctly identifies snippet (a) as vulnerable and snippet (b) as safe, demonstrating its effectiveness in distinguishing similar code snippets.

6.7 Robustness testing

To evaluate the robustness of our method, we conducted a robustness test using the Metropolis-Hastings Modifier (MHM) perturbation method (Zhang et al. 2020). This approach introduces modifications to the code by perturbing variable names, function names, and other identifiers, while preserving the code's overall functionality and structure. The goal is to simulate real-world code changes, such as refactoring and renaming, and assess how well the model performs under such small perturbations.

```
01 static void build_l4proto_dccp(const struct nf_conntrack *ct, struct nethdr *n)
02 {
03   ct_build_group(ct, ATTR_GRP_ORIG_PORT, n, NTA_PORT,
04                  sizeof(struct nfct_attr_grp_port));
05         if (!nfct_attr_is_set(ct, ATTR_DCCP_STATE))
06               return;
07
08         ct_build_u8(ct, ATTR_DCCP_STATE, n, NTA_DCCP_STATE);
09         ct_build_u8(ct, ATTR_DCCP_ROLE, n, NTA_DCCP_ROLE);
10 }
```

(a)

```
01 static void build_l4proto_sctp(const struct nf_conntrack *ct, struct nethdr *n)
02   {
03         /* SCTP is optional, make sure nf_conntrack_sctp is loaded */
04         if (!nfct_attr_is_set(ct, ATTR_SCTP_STATE))
05               return;
06
07         ct_build_group(ct, ATTR_GRP_ORIG_PORT, n, NTA_PORT,
08                        sizeof(struct nfct_attr_grp_port));
09         ct_build_u8(ct, ATTR_SCTP_STATE, n, NTA_SCTP_STATE);
10         ct_build_u32(ct, ATTR_SCTP_VTAG_ORIG, n, NTA_SCTP_VTAG_ORIG);
11         ct_build_u32(ct, ATTR_SCTP_VTAG_REPL, n, NTA_SCTP_VTAG_REPL);
12 }
```

(b)

Fig. 7 Case study of CWE-17 DoS vulnerability. (a) `build_l4proto_dccp` without module check leads to potential crash or DoS. (b) `build_l4proto_sctp` includes module check and is safe

Table 10 Performance comparison before and after perturbations using the Metropolis-Hastings Modifier on three datasets

Dataset	F1 before Perturbation	F1 after Perturbation	Change in F1
Devign	0.68	0.67	-0.01
Reveal	0.49	0.47	-0.02
Big-Vul	0.79	0.77	-0.02

We applied MHM perturbations to the source code across three datasets and evaluated the F1-scores before and after the perturbations to assess the impact of these changes on model performance. The results, presented in Table 10, demonstrate that the drop in F1-score due to MHM perturbations is minimal, ranging from 0.01 to 0.02 across all datasets. These findings affirm the robustness of MCL-VD.

6.8 How accurate is MCL-VD for predicting the Top-25 Most Dangerous CWEs?

To evaluate the practical effectiveness of MCL-VD in real-world security scenarios, we conduct an additional evaluation focusing on the Top-25 most dangerous CWEs[4]. These CWEs represent the most common and impactful vulnerabilities over the past two years.

[4] https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html

Table 11 The Accuracy of MCL-VD for the Top-25 Most Dangerous CWEs

Rank	CWE	Name	Proportion	TPR
1	CWE-79	Cross-site Scripting	20/25	0.80
5	CWE-22	Path Traversal	5/5	1.00
6	CWE-125	Out-of-bounds Read	38/46	0.83
12	CWE-20	Improper Input Validation	102/121	0.84
13	CWE-77	Command Injection	3/3	1.00
17	CWE-200	Information Exposure	46/53	0.87
20	CWE-119	Improper Memory Access	180/198	0.91
23	CWE-190	Integer Overflow	29/31	0.94
Average			**423/482**	**0.88**

Since the dataset does not sufficiently represent all of the Top-25 most dangerous CWEs, we selected a subset of these CWEs for evaluation. Due to the limited number of examples available for certain CWE IDs, traditional metrics such as F1-score, precision, and recall may lack stability and reliability in providing meaningful insights. Consequently, we employed the True Positive Rate (TPR) as the primary evaluation metric to ensure robust assessment of the model's performance.

As shown in Table 11, MCL-VD achieves high true positive rates across a variety of critical vulnerability types. For instance, the model reaches perfect TPR for CWE-22 (Path Traversal) and CWE-77 (Command Injection), which often exhibit distinctive patterns, making them easier to detect accurately. On the other hand, CWE-79 (Cross-site Scripting) shows a relatively lower TPR of 80%, likely due to the diverse and context-dependent nature of input sanitization. Overall, MCL-VD achieves an average TPR of 88% on these selected high-risk CWEs, demonstrating its effectiveness in identifying critical software vulnerabilities.

The differentiated performance across Top-25 CWEs reflects MCL-VD's architectural alignment with vulnerability characteristics. Structural vulnerabilities, such as CWE-22 and CWE-77, achieve near-perfect TPR due to AST's explicit syntactic pattern recognition, which detects risky API calls and unsanitized input flows. Memory-related flaws (CWE-119/125/190) exhibit 90% TPR through synergistic AST-boundary checks and comment-augmented context, enabling precise correlation of code semantics with buffer allocations. Context-dependent vulnerabilities, such as CWE-79, exhibit a moderated TPR due to semantic ambiguity: the AST struggles to distinguish between secure encoding functions and vulnerable outputs, while comments often lack security-specific nuance.

These variations highlight the reliance of MCL-VD on explicit structural patterns for high-confidence detections and reveal challenges in resolving semantic ambiguity within complex web security contexts. The multi-modal fusion demonstrates superior effectiveness for vulnerabilities with clear syntactic markers compared to those requiring deeper semantic understanding. This performance dichotomy stems from the inherent tradeoff between structural precision and semantic flexibility in the model's design.

6.9 Threats to validity

In this section, we discuss potential threats to the validity of our research results and the measures taken to mitigate them.

Threats to internal validity primarily concern factors like hyperparameter selection that could influence experimental outcomes. LoRA depends on hyperparameters such as rank r, learning rate, and the number of layers, which can significantly affect performance. We conducted comprehensive experiments to mitigate this, selecting reasonable and stable parameter values based on standard practices and preliminary trials. While our main objective was to demonstrate the model's effectiveness rather than optimize every parameter, we believe that the chosen configurations fairly reflect the model's capabilities.

Threats to external validity involve the generalizability of our results to other datasets and programming languages. To alleviate this threat, our evaluation was conducted on the Devign, Reveal, and Big-Vul datasets, which are widely used in the research community for vulnerability detection tasks. Despite this, these datasets consist primarily of code written in specific programming languages, which may limit the applicability of our findings to other languages or domains. Furthermore, because our comment modality relies on annotations generated by a large language model, which can exhibit inherent stochastic variation, reproducing results exactly may be challenging. To address broader applicability, we have designed the model to remain flexible and easily adaptable to other domains, though our experiments were focused on these specific datasets.

Threats to construct validity relate to the suitability of the metrics used to evaluate our model. To alleviate this threat, we employed a set of well-established metrics, including F1-score, accuracy, precision, and recall, to provide a balanced assessment of the model's performance. These metrics are widely used in the vulnerability detection literature, providing a comprehensive view of the model's classification capabilities. However, to mitigate potential issues with class imbalance in the datasets, we also considered the weighted and effort-aware evaluation metrics to ensure the robustness of our evaluation.

7 Conclusion and future work

This paper introduces MCL-VD, a multi-modal vulnerability detection model that leverages contrastive learning and LoRA for enhanced model performance and computational efficiency. By incorporating contrastive learning with pre-trained models like GraphCodeBERT, MCL-VD generates more effective semantic representations of source code, improving its ability to distinguish between benign and vulnerable code. Furthermore, integrating LoRA optimization ensures a good trade-off between performance and efficiency by reducing the number of trainable parameters without compromising overall detection capabilities. We designed and experimented on three widely recognized vulnerability detection datasets, i.e., Devign, Reveal, and Big-Vul, demonstrating that MCL-VD significantly outperforms existing state-of-the-art methods across all key performance metrics. The integration of multi-modal contrastive learning leads to substantial improvements in F1-score, accuracy, recall, and precision, with F1-score gains ranging from 4.86% to 17.26%.

In future work, we plan to expand the application of MCL-VD to additional datasets and programming languages to further validate its generalizability. We also aim to explore more advanced contrastive learning techniques, such as self-supervised learning and domain adaptation, to further enhance model performance. Finally, we aim to refine the LoRA optimization process to strike a balance between model size, training efficiency, and performance across various vulnerability detection tasks.

Acknowledgements This work is partly supported by the National Natural Science Foundation of China (Grant No. 61872263) and the Postgraduate Research & Practice Innovation Program of Jiangsu Province (Grant No. SJCX25_2000).

Author Contributions All authors contributed significantly to the research and preparation of this paper. Yi Cao conceptualized the study, designed the MCL-VD framework, and led the development of the LoRA-enhanced GraphCodeBERT model. Xiaolin Ju implemented the multi-modal contrastive learning approach and conducted the experiments, including dataset preparation and performance evaluation. Xiang Chen focused on the theoretical aspects of contrastive learning and contributed to the analysis and interpretation of the results. Lina Gong provided expertise in vulnerability detection, reviewed existing methods, and assisted in benchmarking the proposed framework against state-of-the-art baselines. All authors contributed to the writing and editing of the manuscript, with Yi Cao and Xiaolin Ju coordinating the overall structure and ensuring technical accuracy. All authors reviewed and approved the final manuscript.

Data Availability No datasets were generated or analysed during the current study.

Declarations

Competing interests The authors declare no competing interests.

References

Aberdam, A., Litman, R., Tsiper, S., Anschel, O., Slossberg, R., Mazor, S., Manmatha, R., Perona, P.: Sequence-to-sequence contrastive learning for text recognition. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 15302–15312 (2021)

Ayewah, N., Pugh, W., Hovemeyer, D., Morgenthaler, J.D., Penix, J.: Using static analysis to find bugs. IEEE Softw. **25**(5), 22–29 (2008)

Bruening, D., Amarasinghe, S.: Efficient, transparent, and comprehensive runtime code manipulation (2004)

Chakraborty, S., Krishna, R., Ding, Y., Ray, B.: Deep learning based vulnerability detection: Are we there yet? IEEE Trans. Softw. Eng. **48**(9), 3280–3296 (2021)

Chen, T., Kornblith, S., Norouzi, M., Hinton, G.: A simple framework for contrastive learning of visual representations. In: International Conference on Machine Learning, pp. 1597–1607 (2020). PMLR

Cheng, X., Zhang, G., Wang, H., Sui, Y.: Path-sensitive code embedding via contrastive learning for software vulnerability detection. In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 519–531 (2022)

Chess, B., McGraw, G.: Static analysis for security. IEEE Sec. Privacy. **2**(6), 76–79 (2004)

Dettmers, T., Pagnoni, A., Holtzman, A., Zettlemoyer, L.: Qlora: efficient finetuning of quantized llms (2023). **52**:3982–3992 (2023). arXiv:2305.14314

Fan, J., Li, Y., Wang, S., Nguyen, T.N.: Ac/c++ code vulnerability dataset with code changes and cve summaries. In: Proceedings of the 17th International Conference on Mining Software Repositories, pp. 508–512 (2020)

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al.: Codebert: A pre-trained model for programming and natural languages (2020). arXiv:2002.08155

Guo, D., Lu, S., Duan, N., Wang, Y., Zhou, M., Yin, J.: Unixcoder: Unified cross-modal pre-training for code representation. In: Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pp. 7212–7225 (2022)
Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., et al.: Graphcodebert: Pre-training code representations with data flow (2020). arXiv:2009.08366
Hanif, H., Maffeis, S.: Vulberta: Simplified source code pre-training for vulnerability detection. In: 2022 International Joint Conference on Neural Networks (IJCNN), pp. 1–8 (2022). IEEE
Hayou, S., Ghosh, N., Yu, B.: Lora+: Efficient low rank adaptation of large models (2024). arXiv:2402.12354
He, K., Fan, H., Wu, Y., Xie, S., Girshick, R.: Momentum contrast for unsupervised visual representation learning. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 9729–9738 (2020)
Hu, E.J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., Chen, W.: Lora: Low-rank adaptation of large language models (2021). arXiv:2106.09685
Jain, P., Jain, A., Zhang, T., Abbeel, P., Gonzalez, J.E., Stoica, I.: Contrastive code representation learning (2020). arXiv:2007.04973
Jiang, C., Xu, H., Dong, M., Chen, J., Ye, W., Yan, M., Ye, Q., Zhang, J., Huang, F., Zhang, S.: Hallucination augmented contrastive learning for multimodal large language model. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 27036–27046 (2024)
Kan, X., Sun, C., Liu, S., Huang, Y., Tan, G., Ma, S., Zhang, Y.: Sdft: A pdg-based summarization for efficient dynamic data flow tracking. In: 2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS), pp. 702–713 (2021). IEEE
Kim, S., Kim, R.Y.C., Park, Y.B.: Software vulnerability detection methodology combined with static and dynamic analysis. Wireless Personal Commun. **89**, 777–793 (2016)
Kopiczko, D.J., Blankevoort, T., Asano, Y.M.: Vera: Vector-based random matrix adaptation (2023). arXiv:2310.11454
Lahat, D., Adali, T., Jutten, C.: Multimodal data fusion: an overview of methods, challenges, and prospects. Proceed. IEEE. **103**(9), 1449–1477 (2015)
LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proceed. IEEE **86**(11), 2278–2324 (1998)
Li, L., Ding, S.H., Walenstein, A., Charland, P., Fung, B.C.: Dynamic neural control flow execution: an agent-based deep equilibrium approach for binary vulnerability detection. In: Proceedings of the 33rd ACM International Conference on Information and Knowledge Management, pp. 1215–1225 (2024)
Li, Y., Yu, Y., Liang, C., He, P., Karampatziakis, N., Chen, W., Zhao, T.: Loftq: Lora-fine-tuning-aware quantization for large language models (2023). arXiv:2310.08659
Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., Zhong, Y.: Vuldeepecker: A deep learning-based system for vulnerability detection (2018). arXiv:1801.01681
Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., Chen, Z.: Sysevr: A framework for using deep learning to detect software vulnerabilities. IEEE Trans. Dependable Secure Comput. **19**(4), 2244–2258 (2021)
Liu, T., Curtsinger, C., Berger, E.D.: Doubletake: Fast and precise error detection via evidence-based dynamic analysis. In: Proceedings of the 38th International Conference on Software Engineering, pp. 911–922 (2016)
Liu, S.-Y., Wang, C.-Y., Yin, H., Molchanov, P., Wang, Y.-C.F., Cheng, K.-T., Chen, M.-H.: Dora: Weight-decomposed low-rank adaptation (2024). arXiv:2402.09353
Liu, S., Wu, B., Xie, X., Meng, G., Liu, Y.: Contrabert: Enhancing code pre-trained models via contrastive learning. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), pp. 2476–2487 (2023). IEEE
Livshits, V.B., Lam, M.S.: Finding security vulnerabilities in java applications with static analysis. In: USENIX Security Symposium, vol. 14, pp. 18–18 (2005)
Maaten, L., Hinton, G.: Visualizing data using t-sne. J. Mach. Learn. Res. **9**(11) (2008)
Marjanov, T., Pashchenko, I., Massacci, F.: Machine learning for source code vulnerability detection: What works and what isn't there yet. IEEE Sec. Privacy. **20**(5), 60–76 (2022)
Meng, F., Wang, Z., Zhang, M.: Pissa: Principal singular values and singular vectors adaptation of large language models (2024). arXiv:2404.02948
Neelakantan, A., Xu, T., Puri, R., Radford, A., Han, J.M., Tworek, J., Yuan, Q., Tezak, N., Kim, J.W., Hallacy, C., et al.: Text and code embeddings by contrastive pre-training (2022). arXiv:2201.10005
Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. ACM Sigplan Notices. **42**(6), 89–100 (2007)

Neuhaus, S., Zimmermann, T., Holler, C., Zeller, A.: Predicting vulnerable software components. In: Proceedings of the 14th ACM Conference on Computer and Communications Security, pp. 529–540 (2007)

Nguyen, V.-A., Nguyen, D.Q., Nguyen, V., Le, T., Tran, Q.H., Phung, D.: Regvd: Revisiting graph neural networks for vulnerability detection. In: Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings, pp. 178–182 (2022)

Nunes, P., Medeiros, I., Fonseca, J., Neves, N., Correia, M., Vieira, M.: An empirical study on combining diverse static analysis tools for web security vulnerabilities based on development scenarios. Computing. **101**, 161–185 (2019)

Renduchintala, A., Konuk, T., Kuchaiev, O.: Tied-lora: Enhacing parameter efficiency of lora with weight tying (2023). arXiv:2311.09578

Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Learning representations by back-propagating errors. Nature. **323**(6088), 533–536 (1986)

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

Sharma, T., Kechagia, M., Georgiou, S., Tiwari, R., Vats, I., Moazen, H., Sarro, F.: A survey on machine learning techniques for source code analysis (2021). arXiv:2110.09610

Shin, Y., Williams, L.: Is complexity really the enemy of software security? In: Proceedings of the 4th ACM Workshop on Quality of Protection, pp. 47–50 (2008)

Svyatkovskiy, A., Deng, S.K., Fu, S., Sundaresan, N.: Intellicode compose: Code generation using transformer. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1433–1443 (2020)

Wang, Y., Wang, W., Joty, S., Hoi, S.C.: Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation (2021). arXiv:2109.00859

Wang, R., Xu, S., Tian, Y., Ji, X., Sun, X., Jiang, S.: Scl-cvd: Supervised contrastive learning for code vulnerability detection via graphcodebert. Comput. Sec. **145**, 103994 (2024)

Wen, F., Nagy, C., Bavota, G., Lanza, M.: A large-scale empirical study on code-comment inconsistencies. In: 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC), pp. 53–64 (2019). IEEE

Wu, H., Zhao, H., Zhang, M.: Code summarization with structure-induced transformer. In: Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021, pp. 1078–1090 (2021)

Wu, Y., Zou, D., Dou, S., Yang, W., Xu, D., Jin, H.: Vulcnn: An image-inspired scalable vulnerability detection system. In: Proceedings of the 44th International Conference on Software Engineering, pp. 2365–2376 (2022)

Xu, S., Zhang, X., Wu, Y., Wei, F.: Sequence level contrastive learning for text summarization. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 36, pp. 11556–11565 (2022)

Yang, Y., Xia, X., Lo, D., Grundy, J.: A survey on deep learning for software engineering. ACM Comput. Surv. (CSUR). **54**(10s), 1–73 (2022)

Zhang, Q., Chen, M., Bukharin, A., Karampatziakis, N., He, P., Cheng, Y., Chen, W., Zhao, T.: Adalora: Adaptive budget allocation for parameter-efficient fine-tuning (2023). arXiv:2303.10512

Zhang, H., Li, Z., Li, G., Ma, L., Liu, Y., Jin, Z.: Generating adversarial examples for holding robustness of source code processing models. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 34, pp. 1169–1176 (2020)

Zhou, Y., Liu, S., Siow, J., Du, X., Liu, Y.: Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. Adv. Neural Inf. Process. Syst. **32** (2019)

Authors and Affiliations

Yi Cao[1] · Xiaolin Ju[1] · Xiang Chen[1] · Lina Gong[2]

✉ Xiaolin Ju
ju.xl@ntu.edu.cn

✉ Xiang Chen
xchencs@ntu.edu.cn

Yi Cao
ntucaoyi@outlook.com

Lina Gong
gonglina@nuaa.edu.cn

[1] School of Artificial Intelligence and Computer Science, Nantong University, Nantong, Jiangsu, China

[2] School of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, Jiangsu, China