PDF Download
3771923.pdf
16 January 2026
Total Citations: 2
Total Downloads: 375

Accepted: 09 October 2025
Revised: 08 July 2025
Received: 19 July 2024

Citation in BibTeX format

RESEARCH-ARTICLE

# M2CVD: Enhancing Vulnerability Understanding through Multi-Model Collaboration for Code Vulnerability Detection

ZILIANG WANG, Peking University, Beijing, China

GE LI, Peking University, Beijing, China

JIA LI, Peking University, Beijing, China

JIA LI, Peking University, Beijing, China

MENG YAN, Chongqing University, Chongqing, China

YINGFEI XIONG, Peking University, Beijing, China

View all

# M2CVD: Enhancing Vulnerability Understanding through Multi-Model Collaboration for Code Vulnerability Detection

ZILIANG WANG, GE LI*, JIA LI ♂, and JIA LI, Key Lab of High Confidence Software Technology, MoE, School of Computer Science, Peking University, China
MENG YAN, Chongqing University, China
YINGFEI XIONG and ZHI JIN, Key Lab of High Confidence Software Technology, MoE, School of Computer Science, Peking University, China

Large Language Models (LLMs) have strong capabilities in code comprehension, but fine-tuning costs and semantic alignment issues limit their project-specific optimization; conversely, fine-tuned models such as CodeBERT are easy to fine-tune, but it is often difficult to learn vulnerability semantics from complex code languages. To address these challenges, this paper introduces the Multi-Model Collaborative Vulnerability Detection approach (M2CVD) that leverages the strong capability of analyzing vulnerability semantics from LLMs to improve the detection accuracy of fine-tuned models. M2CVD employs a novel collaborative process: first enhancing the quality of vulnerability description produced by LLMs through the understanding of project code by fine-tuned models, and then using these improved vulnerability descriptions to boost the detection accuracy of fine-tuned models. M2CVD include three main phases: 1) Initial Vulnerability Detection: The initial vulnerability detection is conducted by fine-tuning a detection model (e.g., CodeBERT) and interacting with an LLM (e.g., ChatGPT) respectively. The vulnerability description will be generated by the LLM when the code is detected vulnerable by the LLM. 2) Vulnerability Description Refinement: By informing the LLM of the vulnerability assessment results of the detection model, we refine the vulnerability description by interacting with the LLM. Such refinement can enhance LLM's vulnerability understanding in specific projects, effectively bridging the previously mentioned alignment gap; 3) Integrated Vulnerability Detection: M2CVD integrates code fragment and the refined vulnerability descriptions inferred to form synthetic data. Then, the synthetic data is used to fine-tune a validation model, optimize the defect feature learning efficiency of the model, and improve the detection accuracy. We demonstrated M2CVD's effectiveness on two real-world datasets, where M2CVD significantly outperformed the baseline. In addition, we demonstrate that the M2CVD collaborative method can extend to other different LLMs and fine-tuned models to improve their accuracy in vulnerability detection tasks.

CCS Concepts: • **Do Not Use This Code → Generate the Correct Terms for Your Paper**; *Generate the Correct Terms for Your Paper*; Generate the Correct Terms for Your Paper; Generate the Correct Terms for Your Paper.

Additional Key Words and Phrases: Vulnerability detection, Model collaboration, Large language model, Pre-trained models

*Corresponding author.

Authors' Contact Information: Ziliang Wang, wangziliang@pku.edu.cn; Ge Li, lige@pku.edu.cn; Jia Li ♂, lijia@stu.pku.edu.cn; Jia Li, lijiaa@pku.edu.cn, Key Lab of High Confidence Software Technology, MoE, School of Computer Science, Peking University, Beijing, China; Meng Yan, Chongqing University, Chongqing, China, mengy@cqu.edu.cn; Yingfei Xiong, xiongyf@pku.edu.cn; Zhi Jin, zhijin@sei.pku.edu.cn, Key Lab of High Confidence Software Technology, MoE, School of Computer Science, Peking University, BeiJing, China.

ACM 1557-7392/2025/10-ART
https://doi.org/10.1145/3771923

## 1 INTRODUCTION

Vulnerabilities in software refer to code weaknesses that can be easily exploited, which can lead to serious consequences such as unauthorized information disclosure [15] and cyber extortion [43]. Recent statistics underscore this burgeoning issue: In Q1 of 2022, the US National Vulnerability Database (NVD) disclosed 8,051 vulnerabilities, marking a 25% increase from the previous year [8]. Further accentuating this trend, a study revealed that out of 2,409 analyzed codebases, 81% had at least one recognized open-source vulnerability. The increasing scale and ubiquity of these vulnerabilities emphasize the need for well-developed automated vulnerability detection mechanisms. Such a detection system helps to strengthen software security and forestall a range of potential security risks [15, 20, 21, 43].

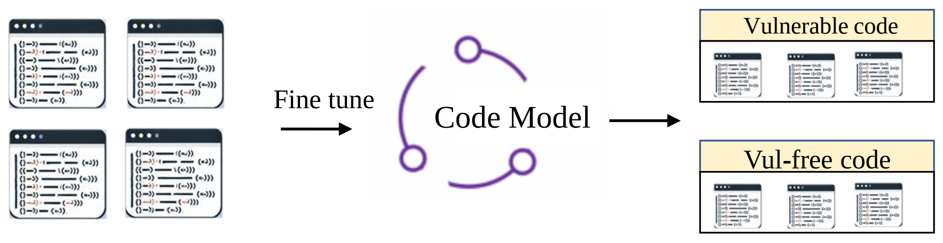
The vulnerability detection models in the existing literature are mainly divided into two categories: (1) conventional detection models [48, 49] and (2) Deep Learning (DL)-based models [9, 12, 24, 27, 39]. The former typically requires experts to manually formulate detection rules [6, 14]. These methods are usually labor-intensive to create and are difficult to achieve low false positive rates and low false negative rates [26, 27]. On the contrary, deep learning (DL) -based detection methods learn the patterns of vulnerabilities from a training set [9, 27, 28]. They avoid manual heuristic methods and autonomously learn and identify vulnerability features. In order to further learn the semantics of vulnerabilities, methods based on fine-tuned models [34] and vulnerability detection studies based on LLMs [16] have been proposed successively. In summary, traditional vulnerability detection methods usually depend on pre-defined rules, a process of expert interventions, rendering them laborious and occasionally imprecise. In contrast, the DL-based detection method can automatically learn the patterns of vulnerable code [46].

In the latest research [41], the efficacy of pre-trained language models for software vulnerability detection has been extensively explored, encompassing LLMs such as ChatGPT [36] and LLaMa [31], alongside fine-tuned models like CodeBERT [13] and UniXcoder [17]. Compared with traditional deep learning networks, these models show more excellent performance in code vulnerability detection tasks after fine-tuning [34]. Fig. 1 illustrates the flow of code model fine-tuning.

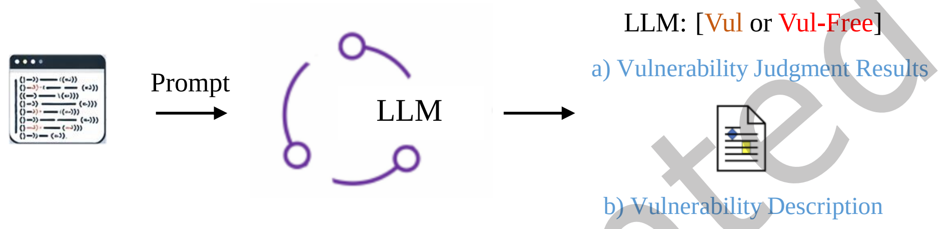
Detecting vulnerabilities using pre-trained models has its benefits, but when it comes to real-world applications, we encounter the first challenge that **the complexity of code makes it hard for the fine-tuned model to learn vulnerability semantics** [42].

The pre-training datasets for fine-tuned models usually do not have vulnerability descriptions. Though fine-tuned models can be fine-tuned on a domain-specific vulnerability dataset, these datasets usually only contain labels to show if a piece of code is vulnerable or not. Without vulnerability description, it would be difficult for fine-tuned models to learn the actual cause of the vulnerabilities. For the same reason, existing vulnerability detection methods usually only output a vulnerability judgment indicating whether the code is vulnerable.

In contrast to existing approaches that use pattern matching to enhance vulnerability semantic [42], we resort to the strong understanding abilities of LLMs to create natural language descriptions of vulnerable code, so as to make connections between code and the causes of vulnerabilities. This will bring two benefits: the more abstract natural language description will help the fine-tuned model better learn the semantics of the vulnerability, and the vulnerability description can help programmers better determine the cause of the vulnerability to maintain the code. In the latest research, this process is exploited for natural language understanding[33]. But we've hit the second challenge, **the semantic alignment problem of LLMs**. Given the scale of LLMs, fine-tuning them on a specific domain is challenging. Real projects, organizations, or specific fields have their own coding rules and business logic. Using LLMs trained on data from open domain might not make accurate vulnerability judgment on code in a specific domain (e.g., Fu et al. [16] reports a F1 score of 29% with GPT4). As a result, LLMs may generate incorrect vulnerability description.



(a) Existing vulnerability detection methods based on fine-tuning


Prompt
LLM
LLM: [Vul or Vul-Free]
a) Vulnerability Judgment Results
b) Vulnerability Description

(b) Vulnerability Detection Process with Large Language Models

Fig. 1. Existing vulnerability detection method processes based on fine-tuned model fine-tuning and interactive vulnerability detection processes based on large language models.

In this paper, we have proposed M2CVD, an innovative approach that combines the strengths of fine-tuned models and LLMs to better detect vulnerabilities. For the first challenge, we rely on the ability of LLMs to interpret vulnerabilities, leveraging the explanatory text to help fine-tuned models to understand the semantics of vulnerabilities. For the second challenge, we rely on the advantage of fine-tuned models that is easy to fine-tune, and use their judgment results to enhance the vulnerability semantic understanding of LLMs for specific projects. In this way, M2CVD can help operators to improve the accuracy of vulnerability detection through the collaborative interaction process combined with LLMs API without changing the existing fine-tuned model structure. In summary, the main contributions of this paper are as follows:

a) We propose M2CVD, an approach that integrates the capabilities of fine-tuned models and LLMs to better utilize their strengths for enhancing the precision of vulnerability detection tasks. Compared with the existing vulnerability detection, M2CVD supports the output of vulnerability description to assist programmers to maintain code.
b) This paper proposes a vulnerability description refinement method, which leverages the insights of fine-tuning fine-tuned models on specific data to effectively enhance the vulnerability description generation ability of unfine-tuned LLMs on project-specific domain code.
c) We evaluate our approach through extensive experimentation on two real-world datasets. Experimental results show that the M2CVD can still improve the performance of code vulnerability detection with the different of fine-tuned model and LLMs.

**Data Availability.** We open-source our replication package[1], including the datasets and the source code of M2CVD, to facilitate other researchers and practitioners to repeat our work and verify their studies.

[1]Our replication package (data and code) :https://github.com/HotFrom/M2CVD

**Paper Organization.** Section 2 describes the background of code vulnerability detection. Section 3 presents our model M2CVD. Sections 4 and 5 describe the datasets and experiments of our study, respectively. Sections 6 provide an ablation experiment. Sections 7 discuss a case of M2CVD, respectively. Section 8 concludes the discussion of M2CVD. And Section 9 includes summary of our approach and future directions.

## 2 RELATED WORK

### 2.1 Traditional Vulnerability Detection

Over the years, a lot of methods for vulnerability detection have emerged. Overall, initial research in this area focused on identifying vulnerabilities by means of manually customized rules [6, 14]. While these approaches provide heuristic approaches to vulnerability detection, they require extensive manual analysis and formulation of defect patterns. In addition, syntactic elements are repeated in different code fragments, as prescribed by certain rules, have been observed to induce elevated rates of both false positives and false negatives [26, 48, 49].

### 2.2 Deep Neural Network for Vulnerability Detection

To minimize human intervention, recent works have turned to employing neural network-based models for the extraction of vulnerability features from code fragments [9, 39]. Existing deep learning-based vulnerability detection models predominantly bifurcate into two classifications: token-based and graph-based models.

Token-based models treat code as a linear sequence and use neural networks to learn vulnerability features from known cases, aiming to identify previously undetected vulnerabilities. [7, 27, 39]. For instance, Russell et al. harnessed the power of both recurrent neural networks (RNNs) and convolutional neural networks (CNNs) to learn feature sets from code token sequences tailored for vulnerability identification [39]. Concurrently, Li et al. [27] employed BiLSTM [40] to encode a segmented version of input code, known as 'code gadget', centered on key markers, especially library/API function calls. However, these token-based models often ignore the complexity of the source code structure, which may lead to inaccurate detection.

While focusing on token-based models, another research direction is to reveal the potential of graph-based methods in the field of vulnerability detection [4, 24, 34, 51]. DeepWukong [7] utilizes GNN for feature learning, which focuses on compressing code fragments into a dense, low-dimensional vector space to enhance the detection of a large number of vulnerability types. DeepTective [37] confronts vulnerabilities common to PHP scripts such as SQL injection, Cross-Site Scripting (XSS), and command injection by deploying a combination of Gated Recurrent Units (GRUs) and Graph Convolutional Networks.

Graph-based detection models learn code structure through varied graph representations, utilizing neural networks for vulnerability detection [3, 47]. For instance, Zhou et al. [52] used the gated graph recurrent network [25], extracting structural details from triadic graph representations—AST, CFG, and DFG. Chakraborty et al. [4] introduced REVEAL, an innovative approach that amalgamation of the gated graph neural network, re-sampling techniques [5], and triplet loss [32]. Wu et al. [47] proposed a approach that can efficiently convert the source code of a function into an image while preserving the program details. Meanwhile, Cao et al. [3] proposed a statement-centric approach, based on flow-sensitive graph neural network, to understanding semantic and structural data.

### 2.3 Pre-Trained Models for Vulnerability Detection

Taking inspiration from the success of pre-trained models in the field of natural language processing (NLP), an increase of related research works aims to leverage these pre-trained models to improve code vulnerability detection accuracy [2, 13, 17, 22, 29, 35].

The core idea of these works is a pre-trained model on a large amount of source code data, followed by specialized fine-tuning for a specific task. [22]. To illustrate, Feng et al. [13] proposed CodeBERT specifically for understanding

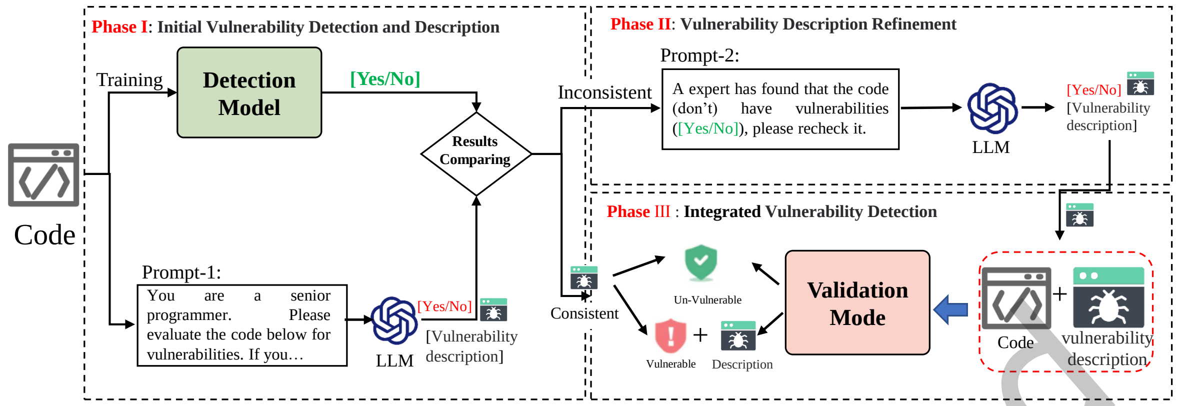

Phase I: Initial Vulnerability Detection and Description
Training
Detection Model
[Yes/No]
Results Comparing
Code
Prompt-1:
You are a senior programmer. Please evaluate the code below for vulnerabilities. If you…
LLM
[Yes/No]
[Vulnerability description]
Inconsistent
Consistent
Phase II: Vulnerability Description Refinement
Prompt-2:
A expert has found that the code (don't) have vulnerabilities ([Yes/No]), please recheck it.
LLM
[Yes/No]
[Vulnerability description]
Phase III : Integrated Vulnerability Detection
Un-Vulnerable
Vulnerable
Description
Validation Mode
Code
vulnerability description

Fig. 2. The framework of M2CVD, which mainly contains three phase: 1. Initial vulnerability detection; 2. vulnerability description refinement and 3. Integrated vulnerability detection. The detection model uses historical vulnerability data to fine-tune, and then fine-tunes the validation model after the historical vulnerability data is supplemented by vulnerability semantics in phase 1 and phase 2.

and generating source code, which combines the processing power of natural and programming languages. Similarly, CuBERT combines masked language modeling with sentence prediction for code representation [22]. In addition, some pre-trained models also take the structural information of the code fragment into account in the initial training phase [29, 35]. For example, Guo et al.'s GraphCodeBERT [19] to infer different in the data flow of code fragments with the help of graph structures. DOBF [23] introduces a novel pre-training objective predicated, explore whether such pre-training can enhance the model's ability to learn the syntactic and structural complexity of source code. The objective is specifically tailored to address the structural dimension of programming languages. Concurrently, to enhance the graph-based representation, GraphCodeBERT [18] proposed a pre-trained schema to seamlessly insert graph structure into Transformer-based architectures. This is achieved through the innovative use of graph-guided masked attention mechanisms, designed to mitigate noise in the data. The comparative evaluation positions CodeBERT as a baseline standard, which is a very classic fine-tuned model in a series of code-related tasks, including code clone detection and code translation. At the same time, UniXcoder as the latest fine-tuned model will also be done as a baseline method. UniXcoder, a unified cross-modal pre-trained programming language model, is trained on a large amount of code data as well as natural language data [17].

Since these pre-trained models have shown superior performance in various code-related tasks, some studies have attempted to use these models for vulnerability detection [15, 19, 43]. However, if these models are directly used for vulnerability detection after fine-tuning with code data, they face the challenge of capturing vulnerability features from long code and complex structure [50]. Moreover, due to the nature of code data, the lack of vulnerability semantics information prevents these multi-modal models from taking full advantage of them. Therefore, we try to supplement the vulnerability semantics in the existing code data to reduce the cost of modeling and searching vulnerability features in complex code data.

## 3 APPROACH

In general, M2CVD requires three models to work together, including detection model $f_d$,validation model $f_v$ and LLM $f_c$, and relies on the collaborative interaction of $f_d$ and $f_c$ to assist the enhancement of $f_v$ model in the vulnerability detection task. The overall framework of M2CVD as shown in Fig. 2, consisting of three phases:

1) Phase I generates preliminary judgments and vulnerability descriptions with the help of $f_c$ and $f_d$.

2) In phase II, the judgments that are inconsistent with $f_c$ and $f_d$ in Phase I will be judged and described by $f_c$ for the second time.

3) The last phase uses the vulnerability text to enhance the vulnerability detection ability of $f_v$.

In the default configuration of this paper, the fine-tuned model used by M2CVD is UnixCoder and LLM is ChatGPT 3.5.

## 3.1 Initial Vulnerability Detection

We use $L = (P, Y)$ to represent the historical vulnerability dataset, where $p_i$ represents a code snippet in a programming language, and $y_i$ represents vulnerability labels, $0 < i \leq M$. The $M$ is the number of code snippet. The values of $y_i$ are 0 or 1. When $y_i = 1$, it indicates that the code is free of vulnerabilities; conversely, it indicates a vulnerability in the code.

First, we split the vulnerability dataset $L$ into a training set $p_t$ and a validation set $p_v$. In the inference phase, the code to be detected is denoted as $p_e$. Then, we fine-tune the detection model using $p_t$. According to the methods provided by the existing literature [41], the detection model $f_d$ is obtained by fine-tuning on the historical vulnerability data.

After obtaining the detection model, the vulnerability assessment of the detection model and the vulnerability assessment and description of the LLM are obtained through the following two steps:

1) Generation of the assessment with detection model.

We need to use the detection model to complete the preliminary vulnerability assessment for $L$. The specific steps for this are:

$$z_i = f_d(p_i), 0 < i \leq M \tag{1}$$

where $f_d$ represents the prediction step of the detection model, and $z_i$ denotes the detection model's assessment of code snippet $p_i$. If the model determines that $p_i$ contains a vulnerability, then $z_i = 0$; otherwise, $z_i = 1$.

2) Generation of the assessment and description with LLMs. In this step, we conduct an initial vulnerability assessment and description of $p_t, p_v, p_e$ through an interactive approach using ChatGPT. We use the following prompt to obtain LLM's assessment and description:

User:You are a senior programmer. Please evaluate the code below for vulnerabilities. If you believe there are vulnerabilities, reply starting with 'Yes' and briefly explain the issue; otherwise, begin with 'No'.
Code: int ff_get_wav_header(AVFormatContext *s, AVIOContext *pb ......

LLM: [Yes][Vulnerability description] or [No]

Following the above mentioned steps, M2CVD obtains the vulnerability assessment and description form LLMs. The formalized procedure is detailed as follows:

$$c_i, n_i = f_c(p_i), 0 < i \leq M \tag{2}$$

By analyzing the responses from the LLM, M2CVD obtains a vulnerability assessment $c$. If the LLM determines the code $p_i$ to be vulnerable and replies with "Yes", then $c_i = 0$. Concurrently, M2CVD records the vulnerability description $n_i$ provided by the LLM. If the LLM determines that the code is not vulnerable, $c_i = 1$ and $n_i$ is set to null.

## 3.2 Vulnerability Description Refinement

Through the process described above, we obtained two vulnerability assessments from the detection models and the LLM, as well as a vulnerability description from the LLM. In Phase II, we need to further refine the vulnerability assessments and descriptions from the LLM.

In this section involves a comparative analysis of the vulnerability assessments. In the case of an inconsistency between assessments, LLMs can be informed of the vulnerability assessment derived from the detection model. The second interaction enables the LLM to obtain the assessment result of the detection model that fine-tuned based on the historical data, which may enable the LLMs to regenerate its vulnerability assessment and description. The prompt for Phase II is as follows:

| |
|---|
| User: You are a senior programmer ... ...<br>Code: int ff_get_wav_header(AVFormatContext *s, AVIOCon ...... |
| LLM: [Yes][Vulnerability description] or [No] |
| User: Another expert has found that the code [does not] have vulnerabilities, please recheck it, and If you believe there are vulnerabilities, reply starting with 'Yes' and briefly explain the issue; otherwise, begin with 'No'." |
| LLM: [Yes][Vulnerability description] or [No] |

Limiting Phase II to code fragments with divergent prediction outcomes can significantly reduce LLM inference time. M2CVD then proceeds to refine the vulnerability descriptions based on this streamlined approach:

$$c_i, n_i = \begin{cases} c_i, n_i & \text{if } c_i == z_i \\ f_c(p_i, z_i) & \text{else} \end{cases} \quad (3)$$

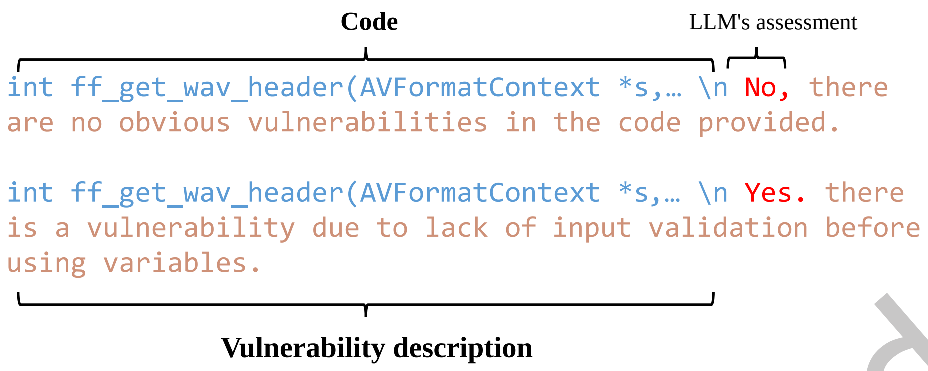
When the assessment results from both models are consistent, the interaction with ChatGPT is terminated. This also means that the refinement process does not trigger. When the models yield inconsistent assessments, i.e., $c_i \neq z_i$, a second round of interaction is initiated using the aforementioned prompt. The values of $c_i$ and $n_i$ are updated accordingly.

## 3.3 Integrated Vulnerability Detection

In Phase III, M2CVD leverages the vulnerability assessments and vulnerability descriptions from the LLM to supplement the vulnerability semantics of the vulnerability code. Initially, we restructure the input dataset $L$ such that $L_c = (P, C, N, Y)$. Here, $C$ represents the LLM's assessment, while $N$ denotes the LLM's description of the vulnerability in the code segment.

When the LLM determines that a code segment has a vulnerability, it typically generates a vulnerability description consisting of $N$ tokens (including the vulnerability assessment "YES" or "NO"), which are directly concatenated to the end of the code. If the LLM determines that the code segment does not have a vulnerability, only the LLM-generated assessment "NO" is concatenated.

As shown in Fig. 3, after integrating the vulnerability semantics into the dataset $L_c = (P, C, N, Y)$, the validation model $f_v$ was obtained by fine-tuning. The training details of the validation model are consistent with the training process of the detection model. At this point, the training dataset already contains the semantic descriptions of the vulnerability code distilled by the LLM, along with the vulnerability judgment feature tokens (Yes or No) generated by the LLM. The validation model will be fine-tuned on the sample data based on the ground truth labels.



```
int ff_get_wav_header(AVFormatContext *s,… \n No, there
are no obvious vulnerabilities in the code provided.

int ff_get_wav_header(AVFormatContext *s,… \n Yes. there
is a vulnerability due to lack of input validation before
using variables.
```

Fig. 3. The data integration process in two cases.

**Algorithm 1** M2CVD

```
Require: $L = (P, Y)$, fine-tuned model $f_d$,$f_v$,LLM:$f_c$
1: Split the dataset $L$ into $p_t, p_v, p_e$
2: Fine-tune the detection model $f_d$ through $p_t$
3: The assessment results of detection models on the dataset were calculated: $z_i = f_d(p_i)$
4: for $i = 1$ to len$(p_i)$ do
5:     $c_i, n_i = f_c(p_i)$
6:     if $z_i \neq c_i$ then
7:         $c_i, n_i = f_c(p_i, z_i)$
8:     end if
9: end for
10: The new data: $p_i^{'} = p_i + n_i$
11: Fine-tune the validation model $f_v$ through $p_t^{'}, p_v^{'}$
    #Inference phase
12: if $f_d(p_e^{'}) == f_c(p_e^{'}) == Vul$ then
13:     The assessment results of the validation model are calculated: $\hat{y} = f_v(p_d^{'})$
14: else
15:     $\hat{y} = f_v(p_e^{'})$
16: end if
17: return $\hat{y}$
```

**Data Identically Distributed Guarantee.** For the fine-tuning process of the validation model, it is crucial to ensure that the training set undergoes both Phase I and II for filling in vulnerability semantics. This approach differs significantly from merely using a vulnerability label for semantic completion via the LLM. We have found that limiting the enhancement of code vulnerability semantics to only those with identified vulnerabilities in the training set can lead to overfitting in the validation model. Therefore, in order to ensure that the training set and the validation test data remain identically distributed as much as possible, it is necessary to prohibit directly informing LLM data labels during the vulnerability semantic generation process in Phases I and II. This necessity occasionally leads ChatGPT to erroneously add semantics to codes perceived as vulnerable. Nevertheless, maintaining the same data source during both the training and inference phases, while integrating a degree of noise, has effectively increased the robustness of the validation model.

### 3.4 Inference Phase

In the inference phase as shown in algorithmic 1, we follow phases 1 and 2 to get the LLM's vulnerability description $n$ and assessment $c$ for the code to be detected $p_e$. Considering that vulnerabilities pose a great risk to software operation. If both the results of the detection model and the results of the LLM believe that the code is vulnerable, the detection result $\hat{y} = c_i$ is directly adopted. When in other situations, the validation model performs the inference as described in the original procedure. Specifically, the input to the validation model will be $p_e^{'} =< p_e, c, n >$, and the final output $\hat{y}$ is calculated by the validation model:

$$\hat{y}_i, n_i = f_v(p_e^{'}), 0 < i \leq M \tag{4}$$

Here, $\hat{y}_i$ signifies the ultimate assessment result corresponding to $p_e$. $n_i$ is the vulnerability description of the code to be tested. After the M2CVD process, $n_i$ can be used to assist programmers to modify the vulnerability. The inference of the detection model and LLM for the code under test are performed simultaneously, and usually the inference time of LLM is greater than that of the detection model. Therefore, the inference cost is the time to call the LLM API plus the time with the validation model.

### 3.5 Implementation Details

The loss function adopted for the fine-tuned models training is the cross-entropy loss [52], commonly used in classification problems for its effectiveness in penalizing the predicted labels and the actual labels:

$$H(y, \hat{y}) = -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y}) \tag{5}$$

M2CVD has two processes of model fine-tuning, in which the fine-tuning process of the first detection model adopts the best performance parameters reported by the existing fine-tuned model, specifically referring to[2].

The detection model is trained for 4 epochs, with a maximum sequence length of 1024 tokens, which is the default for UniXcoder. Code sequences exceeding the maximum input length are automatically truncated to comply with the model's input constraints. The batch size for training is 12. The learning rate is set to 2e-5, and gradient clipping is applied with a maximum gradient norm of 1.0. These hyperparameters are selected based on prior experiments to balance model performance and training efficiency. After completing the vulnerability assessment report, M2CVD implements the validation model fine-tuning process, in which epoch is 4. The maximum sequence length follows the base UniXcoder setting (1024 tokens). In this paper, the learning rate is 2e-5, and the batch size is 12. In this experiment, three V100s-32G GPUs were used for training, and the training time of each epoch was about 9 minutes 20s. For reproducibility, all LLM calls are made with the sampling temperature fixed to 0.

## 4 EXPERIMENTS

### 4.1 Datasets

To evaluate the effectiveness of M2CVD, we employ two datasets from real projects :(1) Devign [52], and (2) Reveal [4]. The Devign dataset, derived from a graph-based code vulnerability detection study [52], stands as a dataset of function-level C/C++ source code from the well-established open-source projects QEMU and FFmpeg. as shown in table 1, aligning with the methodology articulated by Li et al. [52], the partitioning of the Devign dataset adheres to a conventional 80:10:10 ratio, demarcating the bounds for training, validation, and testing data, respectively. The dataset completes the labeling of vulnerable code by a group of security researchers

[2]https://github.com/microsoft/CodeXGLUE/tree/main/Code-Code/Defect-detection

| Dataset | Projects | Total Samples | Vulnerable Samples | Non-Vulnerable Samples |
|---|---|---|---|---|
| Devign | QEMU, FFmpeg | 27,318 | 12,460 | 14,858 |
| Reveal | Debian, Chromium | 22,734 | 2,240 | 20,494 |

Table 1. Summary of Datasets Used for Vulnerability Detection

performing a rigorous two-stage review. In the task of software vulnerability detection, the REVEAL dataset is a representative dataset, as presented in [4]. It is a further exploration of data redundancy and unrepresentative class distributions in existing datasets. As a detection code dataset, REVEAL encompasses source code extracted from two open-source forays: the Linux Debian kernel and Chromium. Similar to the real-world situation, this dataset has an imbalanced label distribution, with the number of normal code fragments much larger than the number of vulnerable ones (10:1). Similarly, in the Reveal dataset, a split ratio of 80:10:10 was set [11].

During the experiment, the proportion of positive and negative samples in the training set, validation set and test set is consistent with the original dataset.

## 4.2 Performance Metrics

In the process of evaluating the performance of the model, the proposed method includes three metrics widely recognized in the field of software testing and analysis[52]:

**Precision**: Denoted as the quotient of the sum of true positives and false positives and is a measure of the accuracy of instances that are identified as positive. Formally, it is defined as:

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

where TP and FP represent the number of true positives and false positives, respectively.

**Recall**: Recall evaluates the fraction of actual positives that are correctly identified and is calculated as the fraction of true positives over the sum of true positives and false negatives:

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

where FN signifies the number of false negatives.

**F1 Score**: The F1 score provides an indicator of the accuracy of the test by combining precision and recall into a single metric by taking their harmonic mean:

$$F1\ Score = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{8}$$

**Accuracy**: This metric reflects the proportion of true positives and true negatives amongst all evaluated instances, thus offering an overall measure of the model's performance:

$$Accuracy = \frac{TP + TN}{TP + TN + FN + FP} \tag{9}$$

where TN representing the number of true negatives.

REVEAL is an imbalanced dataset, so we emphasize the use of F1 as the evaluation metric [11]. The designed dataset is balanced, so we follow the original benchmark to report the classification accuracy [52].

Since the model performance can vary with different random seeds [41], we used the random seed setting commonly used in existing open source methods, seed=42 [11, 13].

### 4.3 Baseline Methods

In our evaluation, we compare M2CVD with seven state-of-the-art methods.

(1) ChatGPT [36]: The GPT series models showcases the capabilities of DL in text generation and processing, albeit not specifically tailored for the domain of software vulnerability detection. ChatGPT 3.5 provides the ability to abstract code vulnerabilities at a lower cost.

(2) Devign [52]: Devign is a graph-based model that uses Gated Graph Recurrent network (GGN) to represent the graph combining AST, CFG, DFG and code sequence of the input code fragment for vulnerability detection.

(3) ReGVD [34]: ReGVD treats the problem as text classification by transforming the source code into a graph structure, using token embedding from GraphCodeBERT [18], and applying a mixture of graph-level sum and max-pooling techniques.

(4) CodeBERT [13]: CodeBERT use a pre-trained structure that amalgamates natural language and programming language, facilitating a broad spectrum of coding tasks, including but not limited to code understanding and generation.

(5) CodeT5 [44]: CodeT5, a unified pretrained encoder-decoder Transformer model that better leverages code semantics conveyed by identifiers assigned from developers.

(6) UniXcoder-base [17]: This is a unified code representation model that leverages a Transformer-based architecture. UniXcoder extends the capabilities of models like CodeBERT by incorporating a comprehensive understanding of code syntax and semantics, thereby enhancing the model's performance in coding tasks such as code summarization, translation, and completion.

(7) UniXcoder-base-nine: Continue pre-training uniXcoder-base on NL-PL pairs of CodeSearchNet dataset and additional 1.5M NL-PL pairs of C, C++ and C# programming language. The model can support nine languages: java, ruby, python, php, javascript, go, c, c++ and c#.

(8) TRACED [11]: TRACED employs an execution-aware pre-training strategy to enhance fine-tuned models' understanding of dynamic code properties, significantly improving their performance in execution path prediction, runtime variable value prediction, clone retrieval, and vulnerability detection.

## 5 EXPERIMENTS

In this section, we conduct extensive experiments to demonstrate our model's superiority and analyze the reasons for its effectiveness. Specifically, we aim to answer the following research questions:

**RQ1:** How effective is M2CVD compared with the state-of-the art baselines on vulnerability detection?

In this RQ, the performance of M2CVD is verified with two real-world vulnerability datasets. Given a code fragment to be detected, M2CVD generates a vulnerability description through a large model, and generates code and vulnerability assessment pairs through collaborative process changes. The performance of M2CVD on the test data is evaluated and compared to the SOTA baseline on two datasets.

**RQ2:** What are the effects of vulnerability description refinement for M2CVD?

In this RQ, we verified the effectiveness of the components, which included the comparison between the results of fine-tuning after generating the vulnerability description directly through the large model and the results of fine-tuning after refining the vulnerability description in step 2.

**RQ3:** What are the effects of hints of fine-tuned models for LLMs?

In this RQ, we verify whether the first judgment results of the fine-tuned model have a positive effect on the LLM. We inform the LLM fine-tuned model or error judgment information respectively to verify the rationality of the M2CVD process.

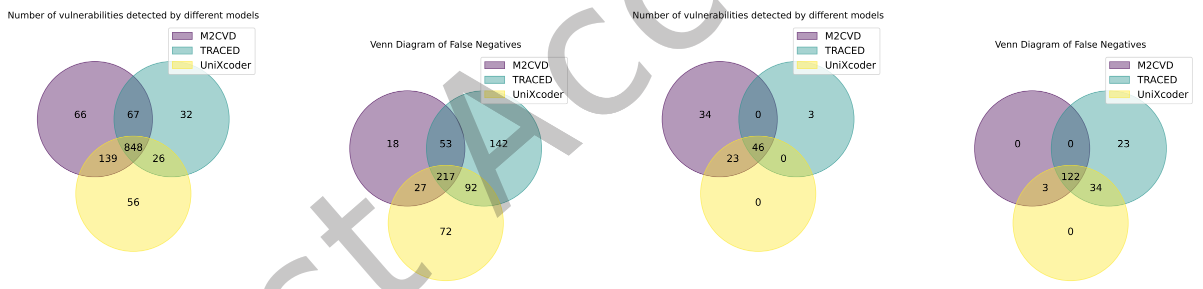
**RQ4:** How well does M2CVD generalize across different fine-tuned models and LLMs?

In this RQ, we validate the performance between different fine-tuned models and LLMs combinations on the vulnerability detection task, validating the generality of the M2CVD approach.

Table 2. Comparison results of different models on Devign and Reveal datasets. The best result for each metric is highlighted in bold.

| Dataset | **Devign [52]** | | | | **Reveal [4]** | | | |
|---|---|---|---|---|---|---|---|---|
| Models | **Acc** | **Recall** | **Prec** | **F1** | **F1** | **Recall** | **Prec** | **Acc** |
| ChatGPT 3.5 COT | 44.73 | 89.64 | 42.73 | 59.84 | 20.22 | 89.91 | 11.39 | 28.85 |
| ChatGPT 4o COT | 52.82 | 43.61 | 9.24 | 15.25 | 18.21 | 99.12 | 10.03 | 10.73 |
| Devign | 56.89 | 52.50 | 64.67 | 57.59 | 33.91 | 31.55 | 36.65 | 87.49 |
| ReGVD | 61.89 | 48.20 | 60.74 | 53.75 | 23.65 | 14.47 | 64.70 | 90.63 |
| CodeBERT | 63.59 | 41.99 | 66.37 | 51.43 | 35.11 | 25.87 | 54.62 | 90.41 |
| UniXcoder-base | 65.77 | 51.55 | 66.42 | 58.05 | 39.47 | 26.31 | 78.94 | 91.90 |
| CodeT5-base | 65.04 | 54.26 | 64.12 | 58.78 | 40.56 | 38.16 | 43.28 | 88.79 |
| UniXcoder-nine | 66.98 | 56.33 | 66.63 | 61.05 | 42.19 | 33.77 | 56.20 | 90.72 |
| TRACED | 64.42 | 61.27 | 60.03 | 61.05 | 32.66 | 21.49 | 68.05 | 91.11 |
| **M2CVD** | **69.25** | **61.51** | **68.38** | **64.77** | **52.54** | **45.18** | 62.42 | **91.78** |

*The Reveal dataset is an imbalanced dataset. F1 and recall are the primary metrics.



(a) Devign: Correct Predictions (b) Devign: False Negatives (c) Reveal: Correct Predictions (d) Reveal: False Negatives

Fig. 4. Comparison of M2CVD, TRACED, and UniXcoder on Devign and Reveal datasets. (a)(b) show the number of correct predictions and false negatives on Devign; (c)(d) show the corresponding results on Reveal.

**RQ5:** What is the impact of the different judgment results between the detection model and the LLM prediction?

## 5.1 RQ1. Effectiveness of M2CVD

To answer the first question, we compare M2CVD with the seven baseline methods on the two datasets as shown in table 2. We can draw conclusions about the performance of M2CVD compared to the baselines across the evaluated datasets.

Table 2 presents the performance of ChatGPT on the vulnerability detection task. It is evident that large-scale language models employ aggressive detection logic. Specifically, in ChatGPT 4o, nearly all code snippets were identified as vulnerable, leading to considerably low F1 scores across both datasets.

M2CVD demonstrates a marked superiority in terms of Accuracy on both datasets. In the Devign dataset, M2CVD attains the highest Accuracy of 69.25%, the highest F1 score of 64.77% and the highest Precision outperforming all other models. This indicates that M2CVD has the most balanced performance in correctly identifying vulnerabilities without being skewed towards over-predicting (which would increase recall but decrease precision) or under-predicting (which would do the opposite).

On the Reveal dataset, since the proportion of negative samples in this dataset is 90, the model with strong fitting performance generally exceeds 90% on ACC. In this dataset, people are generally interested in the ability of the model to find positive samples (vulnerability). For M2CVD, Both Recall and F1 metrics maintain the level of optimal level. These figures not only show that M2CVD maintains its high performance in different testing conditions but also that it consistently understands and predicts code vulnerabilities with high precision and recall. The performance improvement of M2CVD on the reveal dataset is much less than that of Devign, which we believe is caused by the imbalance of the Reveal dataset, and the vulnerability data is far less than the normal data. This allows ChatGPT to add far less vulnerability semantics to this dataset than to the Devign dataset.

Fig. 4 presents a comparison of three models (M2CVD, TRACED, and UniXcoder-base) through Venn diagrams, highlighting their performance in detecting vulnerabilities and false negatives. In Fig. 4(a), the Venn diagram illustrates the overlap in vulnerabilities correctly detected by the three models: M2CVD independently detects 66 vulnerabilities, outperforming TRACED, which detects 32, and UniXcoder-base, which detects 56. The three models collectively identify 848 vulnerabilities, indicating a significant overlap and suggesting that they are effective in detecting similar vulnerabilities. Figure 4(b) depicts the distribution of false negatives among the three models: M2CVD independently produces 18 false negatives, which is significantly lower than the 142 produced by TRACED and 72 by UniXcoder-base. The three models collectively produce 217 false negatives, indicating some common challenges in detecting certain vulnerabilities. In conclusion, the analysis of Fig. 4 demonstrates that the M2CVD model not only excels in detecting a higher number of actual vulnerabilities but also has a lower false negative rate compared to TRACED and UniXcoder. This indicates that the M2CVD model offers superior overall performance in vulnerability detection tasks, making it a valuable tool for applications requiring high accuracy and low false negative rates. Fig. 4 shows the situation in the Reveal dataset. As shown in Fig. 4(c), M2CVD finds more vulnerabilities.

In order to better compare the performance of M2CVD, we adopt the UniXcoder-base as the base model, which has the same number of parameters as baseline models such as TRACED and CodeBERT. After the multi-model collaboration process, M2CVD demonstrates superior accuracy and precision over UniXcoder-base, with marked improvements seen in both the Devign and Reveal datasets. Overall, the performance of M2CVD shows that M2CVD has a more balanced and higher performance in overall performance compared to UniXcoder.

**Answer to RQ1:** The performance of M2CVD shows that compared with a single model, M2CVD effectively achieves higher performance in code defect detection tasks under different experimental conditions through a collaborative mechanism.

## 5.2 RQ2: Effects of vulnerability description refinement for detection performance

In this section, we elaborate on the implications of Phase II feedback within the M2CVD framework on the performance of code vulnerability detection.

For this purpose, a comparative experiment was established using the Devign dataset with its default partitioning. The configurations employed in the experiment are as follows:

a) **CodeBERT**: Utilizes the optimal configuration as reported in existing literature.

Table 3. Model Accuracy Comparison with different M2CVD configuration

| **Model** | **Accuracy** |
|---|---|
| CodeBERT | 63.59% |
| M2CVD(GPT 3.5+CodeBERT) w/o PII | 65.50% |
| M2CVD(GPT 3.5+CodeBERT) | 67.50% |
| UniXcoder-base | 64.82% |
| M2CVD(GPT 3.5+UniXcoder-base) w/o PII | 67.05% |
| M2CVD(GPT 3.5+UniXcoder-base) | 69.25% |
| M2CVD(GPT 4o+UniXcoder-nine) w/o PII | 64.45% |
| M2CVD(GPT 4o+UniXcoder-nine) | 68.99% |

b) **UniXcoder-base**: Also adopts the optimal configuration as documented in existing literature.

c) **M2CVD(CodeBERT) w/o PII**: This configuration bypasses the M2CVD comparison process, meaning that the LLMs conduct a once assessment without incorporating feedback from the fine-tuned model's assessments. The description rendered by LLMs is amalgamated with the code, and predictions are made using the CodeBERT model.

d) **M2CVD(UniXcoder-base) w/o PII**: It omits the M2CVD comparison process. The LLM' first vulnerability detection, when combined with the code, employs the UniXcoder-base model for prediction.

e) **M2CVD(CodeBERT)** : Use the standard M2CVD process where ChatGPT as the LLMs and CodeBERT as the fine-tuned model.

f) **M2CVD(UniXcoder-base)**: Also use the standard M2CVD process, with ChatGPT as the LLMs and UniXcoder-base as the fine-tuned model.

g) **M2CVD(GPT 4o + UniXcoder-nine)**: M2CVD experiments were performed on the latest ChatGPT 4o and fine-tuned model UniXcoder-nine.

The results from the experiment as shown in the Table 3, which presents a comparison results of model accuracy under different M2CVD configurations. As shown in Fig.5, the experimental result evidence suggests a improvement in model accuracy when integrating Phase II feedback into the M2CVD framework. Notably, the M2CVD (CodeBERT) w/o PII outperforms the CodeBERT model by a margin of 1.91%. Similarly, the M2CVD (UniXcoder-base) w/o PII configuration outperforms the UniXcoder by a margin of 2.23%. The enhancements are more pronounced when the comparison includes Phase II, as observed with the M2CVD (CodeBERT) configuration, which incorporates Phase II feedback, outperforms the CodeBERT model by a margin of 2.51%. And the M2CVD (UniXcoder-base) configuration with Phase II integration surpasses its UniXcoder-base counterpart by 3.29%.

The UniXcoder-nine model is obtained by continuing training on code-natural language pairs based on UniXcoder-base. ChatGPT 4o provides an overly aggressive description of vulnerabilities without the refinement process, and considers almost all of the code to be vulnerable. This leads to the phenomenon that the model overfits on the training set and the vulnerability detection performance decreases.

The results confirm the proposed concept. By integrating vulnerability semantics into code data, we enhance the prediction accuracy of the fine-tuned model in vulnerability detection tasks. Concurrently, these results highlight the efficacy of the vulnerability semantic refinement process within the M2CVD framework. This

Table 4. LLMs Accuracy Comparison with different configurations in Phase II

| Model | Accuracy |
|---|---|
| ChatGPT 3.5 COT | 45.29% |
| ChatGPT 3.5 3-fewshot | 45.30% |
| ChatGPT 4o COT | 53.73% |
| CodeLLaMa-13B | 49.57% |
| UniXcoder-base for CodeLLaMa-13B | 52.59% |
| ALL"YES" for ChatGPT 3.5 | 50.04% |
| ALL "NO" for ChatGPT 3.5 | 52.12% |
| UniXcoder-base for ChatGPT 4o | 56.73% |
| UniXcoder-base for ChatGPT 3.5 | 57.61% |

process significantly boosts the fine-tuned model's predictive capabilities during the final execution of the code judgment task.

**Answer to RQ2:** Experiments show that the vulnerability description refinement process of M2CVD can effectively improve the performance of code vulnerability detection.

## 5.3 RQ3. Effects of hints of fine-tuned models for LLMs

In this section, we elaborate on the implications of Phase II feedback within the M2CVD framework on the performance of the ChatGPT model.

The aim is to determine whether informing ChatGPT of the results of the fine-tuned model, after the initial assessment in M2CVD, benefits ChatGPT's performance for code vulnerability detection. For this purpose, we set up a comparative experiment based on the Devign dataset.

The configurations employed in the experiment are as follows:

a) **ChatGPT**: Get ChatGPT's vulnerability assessment interactively.

b) **ChatGPT-fewshow**: Get ChatGPT's vulnerability assessment interactively. Two labeled defective code segments and one labeled non-defective code segment are selected as examples using a random selection method.

c) **ALL"YES" for ChatGPT**: During refinement, we informed ChatGPT of the fine-tuned model evaluations, but all of them were "YES." This meant that we were mistakenly telling ChatGPT that every fragment of code was vulnerable.

d) **ALL "NO" for ChatGPT**: During refinement, we informed ChatGPT of the evaluation of fine-tuned models, but all fine-tuned models judged "NO". This means that we are wrongly telling ChatGPT that every fragment of code is free of bugs.

e) **UniXcoder for CodeLLaMa**: Using the standard M2CVD process, the LLMs was chosen as CodeLLaMa.

f) **UniXcoder for ChatGPT**: Using the standard M2CVD process, the LLMs was chosen as ChatGPT.

where CodeLLaMa-13B is a large language models based on Llama 2. It provides excellent performance among open-source LLMs with long input contexts [38].

Based on the experimental data provided in Table 4, we can conclude that refinement step of the M2CVD framework plays a significant role in enhancing the performance of LLMs in detecting code vulnerabilities. The experiments reveal several insights: The experimental data in Table 4 underscores the impact of the initial feedback given to ChatGPT during Phase II of the M2CVD process. Firstly, ChatGPT-fewshot has almost no improvement over ChatGPT. This is due to the many types and complex forms of code defects, and a small number of instances cannot provide effective reference for large models. When ChatGPT is informed that an expert has judged a piece of code as vulnerable (”YES”), its accuracy in detecting code vulnerabilities increases from 45.29% to 50.04%. This suggests that ChatGPT benefits from additional context. Even by prompting him with insufficient information to check the data again, the detection accuracy can be improved. Similarly, if ChatGPT is consistently informed that an expert has judged the code as not vulnerable (”NO”), the model's accuracy further improves to 52.12%. When the evaluation results of the UniXcoder model fine-tuned on the dataset are provided for ChatGPT, the accuracy of vulnerability detection rises to 57.61%. This suggests that the UniXcoder model encapsulates the dataset's inherent logic effectively and can guide the LLM (ChatGPT) towards more accurate evaluations. On the CodeLLaMa model, we observe the same phenomenon, increasing the accuracy from 49% to 52%.

The accuracy of the basic ChatGPT model is 45.29%, while after the refinement process of M2CVD, ChatGPT shows significant performance improvement. This suggests that from a specialized fine-tuned model, which carries insights from its fine-tuning process, is crucial in helping LLMs better understand and evaluate the code in question.

**Answer to RQ3:** The strategy of informing LLMs with the insights from fine-tuned models that are attuned to the specific dataset logic not only improves the performance but also highlights the potential of collaborative learning systems in code vulnerability detection. This process effectively enhances the accuracy of LLMs in the code vulnerability detection task within the framework of M2CVD, and also enhances the accuracy of LLMs in adding vulnerability semantics.

## 5.4 RQ4. Effects of different fine-tuned models and LLMs working together

In this section, we elaborate on the impact of different fine-tuned models and LLMs on the performance of validation vulnerability prediction. The aim is to judge the impact of the choice of fine-tuned model and LLMs during M2CVD on the final performance. To this end, we set up comparative experiments using the Devign dataset and its default partition.

The configurations employed in the experiment are as follows: Fine-tuned models: CodeBERT, UniXcoder, Qwen2.5-0.5B [1],Pertbert [30] LLM:CodeLLaMa-13B [31],DeepSeek V2.5 [10],ChatGPT-3.5.

The result from the experiment presented in Table 5 provides experimental result about the impact of combining different fine-tuned models and LLMs within the M2CVD framework for predicting code vulnerabilities. The standalone models, CodeBERT and UniXcoder, establish a baseline with accuracy of 63.59% and 64.82%, respectively. The combination of CodeBERT with CodeLLaMa-13B results in a slight accuracy increase, reaching 64.38%. When UniXcoder is paired with CodeLLaMa-13B, there is a more noticeable improvement, with the accuracy climbing to 64.93%. These figures serve as a benchmark to assess the added value of integrating LLMs with fine-tuned models. More substantial gains in accuracy are observed when ChatGPT is introduced to the mix. ChatGPT paired with CodeBERT yields an accuracy of 66.10%, while its combination with UniXcoder tops the table at 68.11%.

Additionally, other LLMs such as DeepSeek V2.5 also show notable contributions. For instance, DeepSeek paired with PertBERT achieves 66.65%, while its combination with Qwen2-coder results in 67.38%. These results demonstrate that DeepSeek is an effective LLM for augmenting the capabilities of various fine-tuned models. However, while DeepSeek provides competitive performance, it does not surpass the accuracy achieved by ChatGPT-3.5.

Table 5. Model Accuracy Comparison with different fine-tuned models and LLMs working together

| **Model** | **Parameters** | **Accuracy** |
|---|---|---|
| CodeBERT | 125M | 63.59% |
| CodeLLaMa-13B+CodeBERT | 13B+125M | 64.38% |
| ChatGPT 3.5+CodeBERT | -+125M | 66.10% |
| Pertbert-base | 125M | 63.90% |
| ChatGPT 3.5+Pertbert | -+125M | 66.36% |
| DeepSeek+Pertbert | 236B+125M | 66.65% |
| Qwen2.5-coder | 0.5B | 64.27% |
| DeepSeek+Qwen2.5-coder | 236B+0.5B | 67.38% |
| ChatGPT 3.5+Qwen2.5-coder | -+0.5B | 68.11% |
| UniXcoder-base | 223M | 64.82% |
| CodeLLaMa-13B+UniXcoder-base | 13B+223M | 64.93% |
| DeepSeek+UniXcoder-base | 1236B+223M | 67.53% |
| ChatGPT 3.5+UniXcoder-base | -+223M | 68.66% |

This indicates that the ChatGPT model significantly enhances the performance of both fine-tuned models, with the ChatGPT+UniXcoder configuration proving to be the most effective partnership in this experiment.

**Answer to RQ4:** The synergy between LLMs and fine-tuned models in the M2CVD framework significantly enhances the precision of detecting code vulnerabilities. Specifically, the more superior performing LLMs and fine-tuned models contributes to a more pronounced accuracy improvement in the M2CVD synergy mechanism.

## 5.5 RQ5. What are the implications of LLM and detection models having different vulnerability judgments

| | | **LLM** | |
|---|---|---|---|
| | | ✓ | × |
| **Detection Model** | ✓ | 51.21% | 14.71% |
| | × | 13.21% | 20.86% |

Table 6. Model Prediction Results

| Detection Model | LLM | Validation Mode | |
|---|---|---|---|
| | | ✓ | × |
| ✓ | ✓ | 91.49% | 8.51% |
| ✓ | × | 77.36% | 22.64% |
| × | ✓ | 50.69% | 49.31% |
| × | × | 17.89% | 82.11% |

Table 7. Validation Results for Detection Model and LLM

In this section, we analyze the various situations in which LLM and test model evaluation results occur during phase 1. This analysis is based on the performance of UniXcoder-base and ChatGPT-3.5 on the Devign dataset, where the LLM was further enhanced using the M2CVD method for secondary validation. As shown in Table 6, 51.21% of the samples were correctly classified by the two models, which represented the most easily detected samples. About 29 percent of the samples could only be detected by one model. Therefore, a total of 27.92% of the samples exhibit inconsistent predictions between the detection model and the LLM. Table 6 illustrates that 20.86% of the samples were completely misclassified by both detection model and LLM. These cases represent the most challenging scenarios for our detection task. To further improve the vulnerability detection accuracy of this part of the sample, the validation model was used for the second round of learning, combining the prediction and description information generated from the two models. Table 7 demonstrates the results of this second stage, revealing that the validation model was able to achieve significant improvements even in these difficult cases.

The validation model achieved notable improvements in most cases, particularly for samples that were initially misclassified by one or both models. In the category where both the detection model and the LLM correctly identified vulnerabilities, the accuracy of the validation model decreased by approximately 8%. The 8.51% drop is mainly due to two factors: first, slight classification divergences arise because the validation and detection models are trained on different inputs under distinct hyperparameter settings; second, a few LLM explanations employ overly severe descriptions(e.g., 'this function may cause a major memory leak'), which mislead the validation model into incorrect judgments. Despite this slight trade-off, the overall benefits of incorporating descriptive features are evident, as the validation model significantly enhanced detection in other categories.

In cases where the detection model failed but the LLM succeeded, the validation model leveraged the LLM's ability to capture Vulnerability descriptions, achieving an accuracy of 74.88%.

Similarly, for samples where both models failed to detect vulnerabilities, the validation model still achieved a substantial improvement, correctly classifying 17.89% of these cases. We further statistical the data on the sample with vulnerabilities and verify that the model has 22.11% accuracy in this category. This means that the validation model found these previously unidentified vulnerabilities.

**Answer to RQ5:** This shows that by integrating vulnerability description, validation models can identify previously undetected patterns, even in the most challenging scenarios.

## 6 ABLATION EXPERIMENT

In the current research, we explore various approaches to enhancing vulnerability semantics to validate the effectiveness of vulnerability semantics generation in the M2CVD framework. We adopt three of the latest methods for this evaluation:

a) Mask: Steenhoek et al. fine-tune the UniXcoder model by adding specific tags before and after potentially vulnerable code through rule-based matching [42].

b) Prepend: Steenhoek et al. also propose copying the matched vulnerable code to the beginning of the snippet to emphasize its context [42].

c) Key Information Extraction (SEU): Min et al. introduce an approach that leverages LLMs to extract key textual information related to events and incorporates it into the fine-tuning process [33].

The performance of these methods, alongside the M2CVD framework, is evaluated on the Devign dataset, as shown in Table 8.

Table 8. Performance of different methods on the Devign dataset

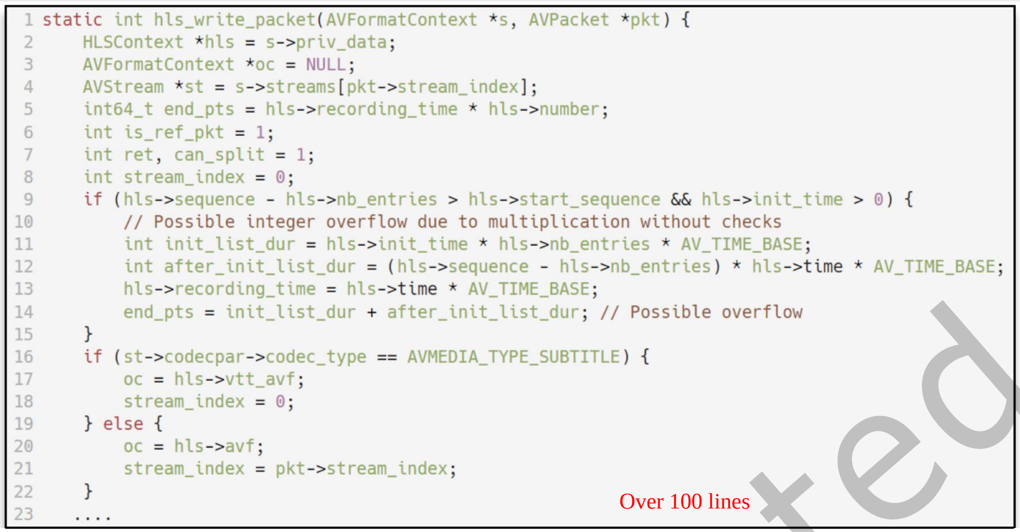
| **Method** | **Acc** | **Recall** | **Prec** | **F1** |
|---|---|---|---|---|
| Mask | 65.84 | 51.23 | 66.70 | 57.95 |
| Prepend | 65.55 | 60.95 | 62.54 | 61.89 |
| SEU | 68.04 | 56.65 | 68.36 | 61.96 |
| M2CVD | 69.25 | 61.51 | 68.38 | 64.77 |

From Table 8, it is evident that while Mask and Pred methods enhance the model's focus on potentially vulnerable code by tagging or emphasizing specific parts, they fall short in addressing the complexity and diversity of real-world vulnerability structures. Similarly, SEU, though effective in extracting key vulnerability-related features through LLMs, lacks a secondary validation process, limiting its ability to refine and validate nuanced vulnerability semantics. In contrast, M2CVD leverages a collaborative refinement process, where the secondary validation in Phase II iteratively enhances the accuracy and relevance of LLM-generated vulnerability descriptions. This approach not only improves the integration of refined vulnerability descriptions into the fine-tuning process but also significantly boosts the model's detection performance, achieving the highest F1 score of 64.77%. These results highlight the importance of iterative refinement and multi-model collaboration in overcoming challenges posed by code complexity and semantic diversity, enabling more accurate and comprehensive vulnerability detection.

## 7 CASE STUDY

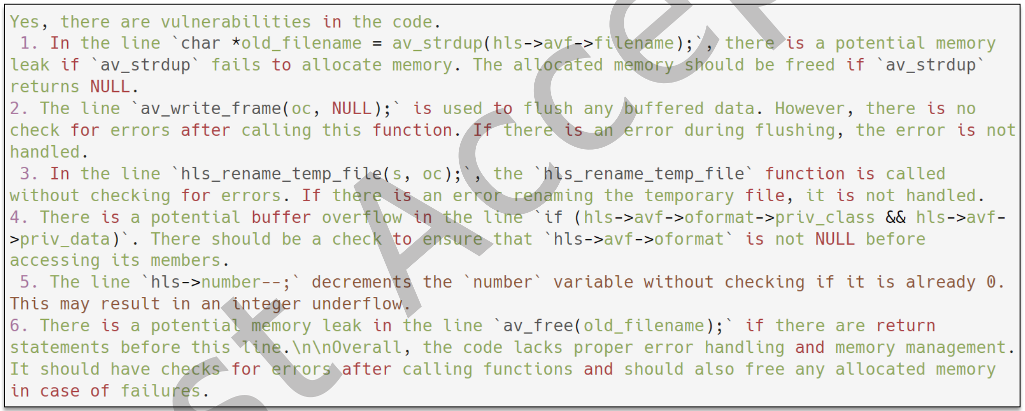
In this section, we show instances of LLMs generate code vulnerability semantics. This is the core idea of the M2CVD method. Fig. 5 presents a fragment of vulnerable code from the Devign dataset. This function spans over 100 lines, yet the vulnerability is concealed within just a few of them. Traditional vulnerability detection models dissect this code into tokens to learn the vulnerability features. However, these features tend to be obfuscated by a large number of no-vulnerability codes, making the learning process challenging.

Fig. 6 provides a natural language description of the vulnerability present in this code by LLMs. This model condenses the risky elements of the code into a succinct natural language summary. This approach offers two significant advantages. Firstly, the simplification of features substantially eases the fine-tuned model's learning process, focusing on specific keywords related to the vulnerabilities. Secondly, it provides a more consistent representation of vulnerabilities. Similar vulnerabilities are expressed diversely across different code segments, leading to a vast feature space for vulnerability characteristics. Language models alleviate this issue by standardizing the same vulnerabilities into a unified natural language description, thereby streamlining the learning and identification process.



```
static int hls_write_packet(AVFormatContext *s, AVPacket *pkt) {
    HLSContext *hls = s->priv_data;
    AVFormatContext *oc = NULL;
    AVStream *st = s->streams[pkt->stream_index];
    int64_t end_pts = hls->recording_time * hls->number;
    int is_ref_pkt = 1;
    int ret, can_split = 1;
    int stream_index = 0;
    if (hls->sequence - hls->nb_entries > hls->start_sequence && hls->init_time > 0) {
        // Possible integer overflow due to multiplication without checks
        int init_list_dur = hls->init_time * hls->nb_entries * AV_TIME_BASE;
        int after_init_list_dur = (hls->sequence - hls->nb_entries) * hls->time * AV_TIME_BASE;
        hls->recording_time = hls->time * AV_TIME_BASE;
        end_pts = init_list_dur + after_init_list_dur; // Possible overflow
    }
    if (st->codecpar->codec_type == AVMEDIA_TYPE_SUBTITLE) {
        oc = hls->vtt_avf;
        stream_index = 0;
    } else {
        oc = hls->avf;
        stream_index = pkt->stream_index;
    }
   ....
```

Fig. 5. A fragment from the Devign dataset with a code vulnerability. The vulnerability is in a few lines in this very long code.



```
Yes, there are vulnerabilities in the code.
 1. In the line `char *old_filename = av_strdup(hls->avf->filename);`, there is a potential memory
leak if `av_strdup` fails to allocate memory. The allocated memory should be freed if `av_strdup`
returns NULL.
2. The line `av_write_frame(oc, NULL);` is used to flush any buffered data. However, there is no
check for errors after calling this function. If there is an error during flushing, the error is not
handled.
 3. In the line `hls_rename_temp_file(s, oc);`, the `hls_rename_temp_file` function is called
without checking for errors. If there is an error renaming the temporary file, it is not handled.
4. There is a potential buffer overflow in the line `if (hls->avf->oformat->priv_class && hls->avf-
>priv_data)`. There should be a check to ensure that `hls->avf->oformat` is not NULL before
accessing its members.
 5. The line `hls->number--;` decrements the `number` variable without checking if it is already 0.
This may result in an integer underflow.
6. There is a potential memory leak in the line `av_free(old_filename);` if there are return
statements before this line.\n\nOverall, the code lacks proper error handling and memory management.
It should have checks for errors after calling functions and should also free any allocated memory
in case of failures.
```

Fig. 6. ChatGPT's abstract representation of the vulnerability in this code fragment(fig.2).

## 8 DISCUSSION

In this section, we discuss the design of prompt in M2CVD, the reason for version selection of ChatGPT model.

**Design of prompt.** In the collaborative process of the M2CVD method, two sentences of intuition-based prompt are used to complete the interaction with ChatGPT. In the prompt used by M2CVD, we followed the experience provided in existing research, setting roles for LLMs and providing task contexts. Existing literature acknowledges the impact that varying prompts can have on the outcomes yielded by LLMs, with techniques such as Chain-of-Thought [45]. However, the focus of this study is mainly to explore the feasibility of multi-model collaboration rather than optimization techniques of prompt, which is a concern more related to the field of prompt engineering. Although the prompts employed within M2CVD may not represent the zenith of optimization, their application has resulted in a significant enhancement of performance in the code vulnerability detection tasks, underscoring the efficacy of the multi-model collaborative approach.

| **ChatGPT 4o** | **Count** | **ChatGPT 3.5-turbo** | **Count** |
|---|---|---|---|
| Memory Management Issues | 21 | Buffer Overflow | 5 |
| Input Validation Issues | 18 | Memory Leaks | 5 |
| Boundary and Overflow Issues | 16 | Improper Error Handling | 5 |
| Error Handling Issues | 10 | Integer Overflow | 5 |
| Concurrency and Synchronization Issues | 4 | Null Pointer Dereference | 3 |

Table 9. Top 5 Vulnerability Types from ChatGPT 3.5 and ChatGPT 4o

**Version of ChatGPT**. In the latest release, ChatGPT 4o offers enhanced generation and understanding capabilities compared to ChatGPT 3.5. However, the relatively high usage fees associated with ChatGPT 4o make it impractical for generating vulnerability semantics for tens of thousands of code fragments. On the other hand, ChatGPT 3.5 has more lenient access policies and pricing, making it a more feasible option for large-scale tasks.

We performed M2CVD on the Devign dataset with different ChatGPT and UniXcoder-base. We conducted experiments using M2CVD on the Devign dataset, comparing different versions of ChatGPT and UniXcoder-base. The cost of using ChatGPT 4o for defect detection on the full dataset amounted to approximately $1200. Despite the higher cost, the experimental results indicated that the vulnerability descriptions generated by ChatGPT 4o were not significantly better than those generated by ChatGPT 3.5. Consequently, ChatGPT 3.5 was selected as the default LLM version for M2CVD.

Our sampling analysis on the Devign dataset revealed that ChatGPT 4o reported 98% of its code as vulnerable due to its overly strict vulnerability definition, resulting in an F1 score of only 8.16%. After applying the M2CVD process, the F1 score improved to 19.23%. Table 9 summarizes the vulnerability reporting by GPT 4o. The high false positive rate of the GPT 4o model can be attributed to several factors. The detector lacks global context information when analyzing code snippets, relying solely on the snippet itself for judgment. This limitation can lead to false positives, such as failing to recognize pre-initialized variables or pre-performed bounds checks in the calling function. The detector's strict criteria flag even minor potential issues as vulnerabilities, like uninitialized variables or unchecked pointers, even when they are safe within the specific context. The detector is highly sensitive to boundary checking and input validation, resulting in numerous unnecessary warnings even in scenarios where out-of-bounds access is impossible. By assuming the worst-case scenario, the detector enhances code robustness in some instances but often imposes excessive error-handling logic that is not required in most real-world applications. ChatGPT3.5's vulnerability determination logic is not that aggressive. GPT 3.5 gives 70 defect judgments and 30 non-defect judgments for 100 sampled data. The accuracy is 53% and the F1 is 38.96%. After the process of M2CVD, ACC is increased to 55%, and F1 is greatly increased to 50.55%. GPT 3.5 gives 56 vulnerability judgments, which tend to follow the distribution of this data set. Considering the sampling experiment results, ChatGPT 3.5 is used as the default LLM in this method.

## 9 CONCLUSION

In this paper, we introduce M2CVD, a novel method designed to address the challenge of software vulnerability detection by harnessing the combined strengths of pre-trained fine-tuned models and large language models.

The M2CVD integrates the language models such as ChatGPT and fine-tuned models like UniXcoder, to create a collaboration process capable of detecting vulnerabilities with high accuracy. Empirical evaluations conducted on the REVEAL and Devign datasets have demonstrated the effectiveness of M2CVD, showcasing its superior performance in detecting code vulnerabilities compared to existing benchmarks. The results of this research not only confirm the viability of M2CVD as a high-fidelity detection system but also underscore the potential of model synergy in enhancing the capabilities of automated vulnerability detection mechanisms. In essence, M2CVD demonstrates the potential to exploit the ability of different models to work together, providing a new idea for future research in automated software vulnerability detection and a scalable and effective solution for protecting software systems from changing threats.

## ACKNOWLEDGMENTS

This work is sponsored by the National Key Research and Development Program of China under Grant No. 2022YFB4501902.

## REFERENCES

[1] 2024. Qwen2 Technical Report. (2024).

[2] Jiangang Bai, Yujing Wang, Yiren Chen, Yaming Yang, Jing Bai, Jing Yu, and Yunhai Tong. 2021. Syntax-BERT: Improving Pre-trained Transformers with Syntax Trees. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*. 3011–3020.

[3] Sicong Cao, Xiaobing Sun, Lili Bo, Rongxin Wu, Bin Li, and Chuanqi Tao. 2022. MVD: memory-related vulnerability detection based on flow-sensitive graph neural networks. In *Proceedings of the 44th International Conference on Software Engineering*. 1456–1468.

[4] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering* (2021).

[5] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. 2002. SMOTE: synthetic minority over-sampling technique. *Journal of artificial intelligence research* 16 (2002), 321–357.

[6] Checkmarx. 2022. Online. *Available: https://www.checkmarx.com/* (2022).

[7] Xiao Cheng, Haoyu Wang, Jiayi Hua, Guoai Xu, and Yulei Sui. 2021. Deepwukong: Statically detecting software vulnerabilities using deep graph neural network. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 3 (2021), 1–33.

[8] Xiao Cheng, Guanqin Zhang, Haoyu Wang, and Yulei Sui. 2022. 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*. 519–531.

[9] Hoa Khanh Dam, Truyen Tran, Trang Pham, Shien Wee Ng, John Grundy, and Aditya Ghose. 2017. Automatic feature learning for vulnerability prediction. *arXiv preprint arXiv:1708.02368* (2017).

[10] DeepSeek-AI. 2024. DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model. arXiv:2405.04434 [cs.CL]

[11] Yangruibo Ding, Benjamin Steenhoek, Kexin Pei, Gail Kaiser, Wei Le, and Baishakhi Ray. [n. d.]. Traced: Execution-aware pre-training for source code. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering ICSE 2024*. 1–12.

[12] Xu Duan, Jingzheng Wu, Shouling Ji, Zhiqing Rui, Tianyue Luo, Mutian Yang, and Yanjun Wu. 2019. VulSniper: Focus Your Attention to Shoot Fine-Grained Vulnerabilities.. In *IJCAI*. 4665–4671.

[13] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. 1536–1547.

[14] Flawfinder. 2022. Online. *Available: http://www.dwheeler.com/ flawfinde/r* (2022).

[15] Michael Fu and Chakkrit Tantithamthavorn. 2022. Linevul: A transformer-based line-level vulnerability prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 608–620.

[16] Michael Fu, Chakkrit Tantithamthavorn, Van Nguyen, and Trung Le. 2023. ChatGPT for Vulnerability Detection, Classification, and Repair: How Far Are We? *arXiv preprint arXiv:2310.09810* (2023).

[17] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. 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)*. 7212–7225.

[18] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, LIU Shujie, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *International Conference on Learning Representations*.

[19] David Hin, Andrey Kan, Huaming Chen, and M Ali Babar. 2022. LineVD: Statement-level vulnerability detection using graph neural networks. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 596–607.

[20] Julian Jang-Jaccard and Surya Nepal. 2014. A survey of emerging threats in cybersecurity. *Journal of computer and system sciences* 80, 5 (2014), 973–993.

[21] Arnold Johnson, Kelley Dempsey, Ron Ross, Sarbari Gupta, Dennis Bailey, et al. 2011. Guide for security-focused configuration management of information systems. *NIST special publication* 800, 128 (2011), 16–16.

[22] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and evaluating contextual embedding of source code. In *International conference on machine learning*. PMLR, 5110–5121.

[23] Marie-Anne Lachaux, Baptiste Roziere, Marc Szafraniec, and Guillaume Lample. 2021. DOBF: A deobfuscation pre-training objective for programming languages. *Advances in Neural Information Processing Systems* 34 (2021), 14967–14979.

[24] Yi Li, Shaohua Wang, and Tien N Nguyen. 2021. Vulnerability detection with fine-grained interpretations. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 292–303.

[25] Yujia Li, Richard Zemel, Marc Brockschmidt, and Daniel Tarlow. 2016. Gated Graph Sequence Neural Networks. In *Proceedings of ICLR'16*.

[26] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2021. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* 19, 4 (2021), 2244–2258.

[27] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681* (2018).

[28] Guanjun Lin, Jun Zhang, Wei Luo, Lei Pan, and Yang Xiang. 2017. POSTER: Vulnerability discovery with function representation learning from unlabeled projects. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 2539–2541.

[29] Jinfeng Lin, Yalin Liu, Qingkai Zeng, Meng Jiang, and Jane Cleland-Huang. 2021. Traceability transformed: Generating more accurate links with pre-trained bert models. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 324–335.

[30] Zhongxin Liu, Zhijie Tang, Junwei Zhang, Xin Xia, and Xiaohu Yang. 2024. Pre-training by Predicting Program Dependencies for Vulnerability Analysis Tasks. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[31] Llama. 2022. Online. *Available: https://ai.meta.com/llama/* (2022).

[32] Chengzhi Mao, Ziyuan Zhong, Junfeng Yang, Carl Vondrick, and Baishakhi Ray. 2019. Metric learning for adversarial robustness. *Advances in neural information processing systems* 32 (2019).

[33] Qingkai Min, Qipeng Guo, Xiangkun Hu, Songfang Huang, Zheng Zhang, and Yue Zhang. 2024. Synergetic Event Understanding: A Collaborative Approach to Cross-Document Event Coreference Resolution with Large Language Models. (2024).

[34] Van-Anh Nguyen, Dai Quoc Nguyen, Van Nguyen, Trung Le, Quan Hung Tran, and Dinh Phung. 2022. ReGVD: Revisiting graph neural networks for vulnerability detection. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 178–182.

[35] Changan Niu, Chuanyi Li, Vincent Ng, Jidong Ge, Liguo Huang, and Bin Luo. 2022. Spt-code: Sequence-to-sequence pre-training for learning source code representations. In *Proceedings of the 44th International Conference on Software Engineering*. 2006–2018.

[36] openAI. 2022. Online. *Available: https://www.chat.openai.com/* (2022).

[37] Rishi Rabheru, Hazim Hanif, and Sergio Maffeis. 2021. DeepTective: Detection of PHP vulnerabilities using hybrid graph neural networks. In *Proceedings of the 36th annual ACM symposium on applied computing*. 1687–1690.

[38] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[39] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. 2018. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE international conference on machine learning and applications (ICMLA)*. IEEE, 757–762.

[40] Mike Schuster and Kuldip K Paliwal. 1997. Bidirectional recurrent neural networks. *IEEE transactions on Signal Processing* 45, 11 (1997), 2673–2681.

[41] Benjamin Steenhoek, Md Mahbubur Rahman, Richard Jiles, and Wei Le. 2023. An empirical study of deep learning models for vulnerability detection. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2237–2248.

[42] Benjamin Steenhoek, Md Mahbubur Rahman, Shaila Sharmin, and Wei Le. 2023. Do Language Models Learn Semantics of Code? A Case Study in Vulnerability Detection. *arXiv preprint arXiv:2311.04109* (2023).

[43] Chandra Thapa, Seung Ick Jang, Muhammad Ejaz Ahmed, Seyit Camtepe, Josef Pieprzyk, and Surya Nepal. 2022. Transformer-based language models for software vulnerability detection. In *Proceedings of the 38th Annual Computer Security Applications Conference*. 481–496.

[44] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. 8696–8708.

[45] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems* 35 (2022), 24824–24837.

[46] Xin-Cheng Wen, Yupan Chen, Cuiyun Gao, Hongyu Zhang, Jie M Zhang, and Qing Liao. 2023. Vulnerability Detection with Graph Simplification and Enhanced Graph Representation Learning. *arXiv preprint arXiv:2302.04675* (2023).
[47] Yueming Wu, Deqing Zou, Shihan Dou, Wei Yang, Duo Xu, and Hai Jin. 2022. VulCNN: An image-inspired scalable vulnerability detection system. In *Proceedings of the 44th International Conference on Software Engineering*. 2365–2376.
[48] Fabian Yamaguchi. 2015. *Pattern-Based Vulnerability Discovery.* Ph. D. Dissertation. University of Göttingen.
[49] Fabian Yamaguchi. 2017. Pattern-based methods for vulnerability discovery. *it-Information Technology* 59, 2 (2017), 101–106.
[50] Junwei Zhang, Zhongxin Liu, Xing Hu, Xin Xia, and Shanping Li. 2023. Vulnerability Detection by Learning from Syntax-Based Execution Paths of Code. *IEEE Transactions on Software Engineering* (2023).
[51] Weining Zheng, Yuan Jiang, and Xiaohong Su. 2021. Vu1SPG: Vulnerability detection based on slice property graph representation learning. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 457–467.
[52] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems* 32 (2019).