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

Anonymous ACL submission

Abstract

Large Language Models (LLMs) have strong capabilities in code comprehension, but fine-tuning costs and semantic alignment issues limit their project-specific optimization; conversely, code models such 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 code models. M2CVD employs a novel collaborative process: first enhancing the quality of vulnerability semantic description produced by LLMs through the understanding of project code by code models, and then using these improved vulnerability semantic description to boost the detection accuracy of code models. 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 code models to improve their accuracy in vulnerability detection tasks.

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 (Fu and Tantithamthavorn, 2022) and cyber extortion (Thapa et al., 2022). Recent statistics underscore this burgeoning issue: In Q1 of 2022, the US National Vulnerability Database (NVD) disclosed 8,051 vulnerabilities, marking a 25% surge from the previous year (Cheng et al., 2022). 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 (Fu and Tantithamthavorn, 2022; Thapa et al., 2022; Jang-Jaccard and Nepal, 2014; Johnson et al., 2011).

The vulnerability detection models in the existing literature are mainly divided into two categories: (1) conventional detection models (Yamaguchi, 2015, 2017) and (2) Deep Learning(DL)-based models (Li et al., 2021a; Duan et al., 2019; Russell et al., 2018; Li et al., 2018; Dam et al., 2017) The former typically requires experts to manually formulate detection rules (Checkmarx, 2022; Flawfinder, 2022). These methods are usually labor-intensive to create and are difficult to achieve low false positive rates and low false negative rates (Li et al., 2018, 2021b). On the contrary, deep learning (DL) -based detection methods learn the patterns of vulnerabilities from a training set (Li et al., 2018; Dam et al., 2017; Lin et al., 2017). They avoid manual heuristic methods and autonomously learn and identify vulnerability features. In order to further learn the semantics of vulnerabilities, methods based on pre-trained code models (Nguyen et al., 2022) and vulnerability detection studies based on LLMs (Fu et al., 2023) have been proposed successively. In summary, traditional vulnerability detection methods are usually depend on pre-defined rules, a process of expert interventions, rendering them laborious and occasionally imprecise. The DL-based detection methods, by contrast, show better detection ability by learning vulnerable code patterns automatically.

In the latest research (Steenhoek et al., 2023a), the efficacy of pre-trained language models for software vulnerability detection has been extensively explored, encompassing LLMs such as ChatGPT (openAI, 2022) and LLaMa (Llama, 2022), alongside pre-trained code models like CodeBERT (Feng et al., 2020) and UniXcoder (Guo et al., 2022). Compared with traditional deep learning networks, these models show more excellent performance in code vulnerability detection tasks after fine-tuning (Nguyen et al., 2022).

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 code model to learn vulnerability semantics** (Steenhoek et al., 2023b). The pre-training datasets for code models usually do not have vulnerability descriptions. Though code 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 semantic description, it would be difficult for code 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 (Steenhoek et al., 2023b), we resort tothe 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 code 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. 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 tough. 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. (2023) reports a F1 score of 29% with GPT4). As a result, LLMs may generate incorrect vulnerability description.

In this paper, we've proposed M2CVD, an innovative approach that combines the strengths of pre-trained code 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 hellp code models to understand the semantics of vulnerabilities. For the second challenge, we rely on the advantage of code 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 code model structure. In summary, the main contributions of this paper are as follows:

a) We propose M2CVD, an approach that integrates the capabilities of pre-trained code 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 semantic description to assist programmers to maintain code.

b) This paper proposes a vulnerability semantic description refinement method, which leverages the insights of fine-tuning pre-trained code 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 pre-trained code model and LLMs. [1].

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 (Checkmarx, 2022; Flawfinder, 2022). 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 (Yamaguchi, 2015, 2017; Li et al., 2021b).

[1]Our anonymous replication package (data and code) :https://figshare.com/articles/dataset/M2CVD/24831567

2.2 Deep Learning 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 (Dam et al., 2017; Russell et al., 2018). 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. (Russell et al., 2018; Li et al., 2018; Cheng et al., 2021). 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 (Li et al., 2021a; Chakraborty et al., 2021; Zheng et al., 2021; Nguyen et al., 2022). Graph-based detection models learn code structure through varied graph representations, utilizing neural networks for vulnerability detection (Wu et al., 2022; Cao et al., 2022).

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 (Feng et al., 2020; Kanade et al., 2020; Niu et al., 2022; Lin et al., 2021; Bai et al., 2021; Guo et al., 2022). 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. (Kanade et al., 2020). In addition, some pre-trained models also take the structural information of the code fragment into account in the initial training phase (Niu et al., 2022; Lin et al., 2021). 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 (Guo et al., 2022).

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 (Fu and Tantithamthavorn, 2022; Thapa et al., 2022; Hin et al., 2022). 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 (Zhang et al., 2023).

3 Approach

In general, M2CVD requires three models to work together, including code model f_d,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 1, 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.

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 $y_i = 0$, 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 (Steenhoek et al., 2023a), 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:

$$z_i = f_d(p_i), 0 < i \leq M \qquad (1)$$

where f_d represents the prediction step of the detection model, and z_i denotes the detection model's

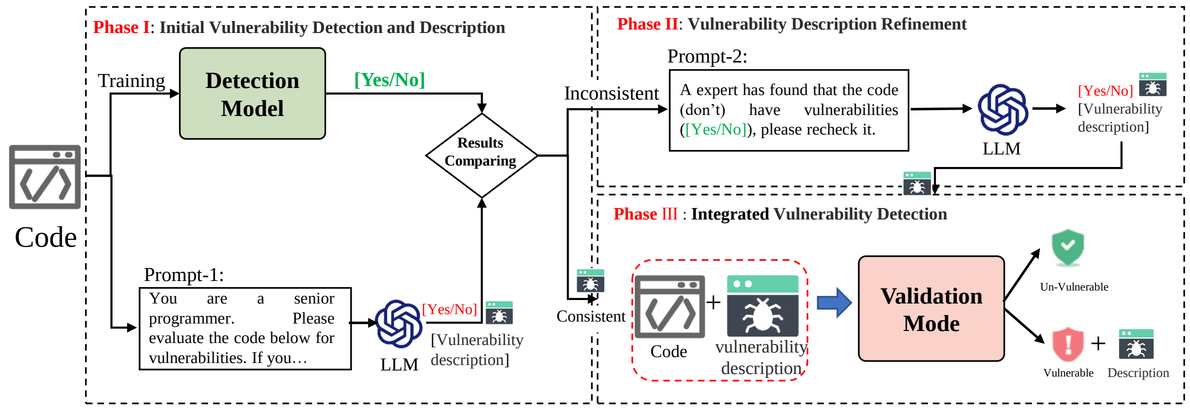

Phase I: Initial Vulnerability Detection and Description
Training
Detection Model
[Yes/No]
Results Comparing
Code
Prompt-1:
You are a senior programmer. Please evaluate the code below for vulnerabilities. If you…
LLM
[Yes/No]
[Vulnerability description]
Inconsistent
Consistent
Phase II: Vulnerability 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
Code
vulnerability description
Validation Mode
Un-Vulnerable
Vulnerable
Description

Figure 1: The framework of M2CVD, which mainly contains three phase: 1. Initial vulnerability detection; 2. vulnerability description refinement and 3. Integrated vulnerability detection. The final output of vulnerability judgment and vulnerability description assists programmers to maintain the code.

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:

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

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 ... ... 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:

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

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 $L_c = (P, C, N)$. The C represents ChatGPT's assessment, while N denotes the LLM's description of the vulnerability in the code segment. After integrating the vulnerability semantics into the dataset $L_c = (P, C, N)$, the **validation model** f_v was obtained by fine-tuning.

Algorithm 1 M2CVD

```
Input: $L = (P, Y)$, code 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: The assessment results of validation model
    were calculated: $\hat{y} = f_v(p_e^{'})$
13: return $\hat{y}$
```

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. Consequently, to adequately supplement the entire training set's semantic content, historical data plays a pivotal role in the vulnerability semantic generation process during 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. Finally, it was combined as the input of the validation model to complete the vulnerability detection task. The code to be detection: $p_e^{'} =< p_e, c, n >$ The formal procedure is articulated as follows:

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

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 Loss Function

The loss function adopted for the code models training is the cross-entropy loss (Zhou et al., 2019), commonly used in classification problems for its effectiveness in penalizing the predicted labels and the actual labels:

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

4 Experiments

4.1 Datasets

To evaluate the effectiveness of M2CVD, we employ two datasets from real projects :(1) Devign (Zhou et al., 2019), and (2) Reveal (Chakraborty et al., 2021). The Devign dataset, derived from a graph-based code vulnerability detection study (Zhou et al., 2019), stands as a dataset of function-level C/C++ source code from the well-established open-source projects QEMU and FFmpeg. The Devign dataset is a label balanced dataset with the ratio of positive and negative samples close to 1 to 1. In the task of software vulnerability detection, the REVEAL dataset is a representative dataset, as presented in (Chakraborty et al., 2021). It is a further exploration of data redundancy and unrepresentative class distributions in existing datasets. The REVEAL dataset is a label imbalance dataset with a 1:10 ratio of vulnerability data to normal data. Similar to known studies, we sample 8:1:1 ratio partition dataset.

Table 1: Comparison results for different models on Devign and Reveal datasets.The best result for each metric is highlighted in bold. The cells in grey represent the performance of the top-3 best methods in each metric, with darker colors representing better performance.

Dataset	Devign (Zhou et al., 2019)				Reveal (Chakraborty et al., 2021)			
Models	**Accuracy**	**Recall**	**Precision**	**F1**	**Accuracy**	**Recall**	**Precision**	**F1**
Devign	56.89	52.50	64.67	57.59	87.49	31.55	36.65	33.91
SySeVR	47.85	46.06	58.81	51.66	74.33	40.07	24.94	30.74
ReGVD	61.89	48.20	60.74	53.75	90.63	14.47	64.70	23.65
CodeBERT	63.59	41.99	66.37	51.43	91.33	34.64	62.20	44.50
UniXcoder	64.82	54.48	63.53	58.94	91.90	26.31	78.94	39.47
CodeT5	65.04	54.26	64.12	58.78	88.79	38.16	43.28	40.56
M2CVD	68.11	54.50	69.51	61.09	92.12	26.75	83.56	40.56

4.2 Baseline Methods

In our evaluation, we compare M2CVD with six baselines: Devign (Zhou et al., 2019), ReGVD (Nguyen et al., 2022), SySeVR (Li et al., 2021b), CodeBERT (Feng et al., 2020), CodeT5 (Wang et al., 2021), UniXcoder (Guo et al., 2022).

4.3 Main Results

In the experiments, the basic models of M2CVD are UniXcoder and ChatGPT. The UniXcoder model configuration is consistent with the baseline approach, and ChatGPT uses the GPT-3.5 API provided by Open AI. To eliminate the randomness of the LLM, the experiment was performed three times and averaged. As shown in tab 1, M2CVD demonstrates a marked superiority in terms of Accuracy on both datasets. In the Devign dataset, M2CVD attains the highest Accuracy of 68.11%, the highest F1 score of 61.09% 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, M2CVD again stands out with the highest Accuracy of 92.12%. Both Precision 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. Experimental reality, ChatGPT has ultra-high recall on both datasets. After case analysis, we found that ChatGPT tends to think that the code does not have vulnerabilities when the code context is not provided.

It is worth noting that the underlying code model for M2CVD is the UniXcoder. After the multi-model collaboration process, M2CVD demonstrates superior accuracy and precision over UniXcoder, with marked improvements seen in both the Devign and Reveal datasets. Overall, the performance comparison of M2CVD shows that the M2CVD collaborative process effectively enhances the performance of the base model without changing the original model training process.

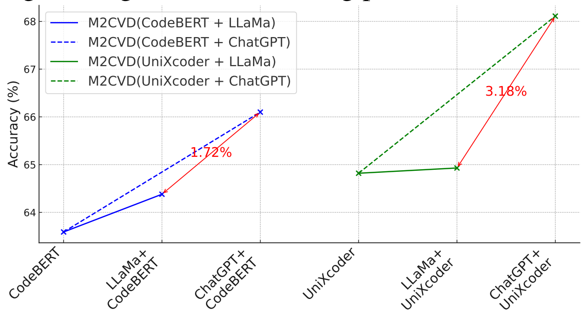


Figure 2: Effects of different code models and LLMs.

5 Ablations

5.1 Effects of different pre-trained code models and LLMs working together

In this section, we elaborate on the impact of different code models and LLMs on the performance of M2CVD vulnerability prediction. The aim is to judge the effectiveness of the proposed collaborative approach in the context of the increasing development of code model and LLM techniques.

To this end, we set up comparative experiments using the Devign dataset and its default partition.

The result from the experiment presented in fig 2 provides experimental result about the impact of combining different code 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 code 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%. Experimental results show that with the performance enhancement of LLM and code mode, the collaborative mechanism of M2CVD is more effective in vulnerability prediction tasks.

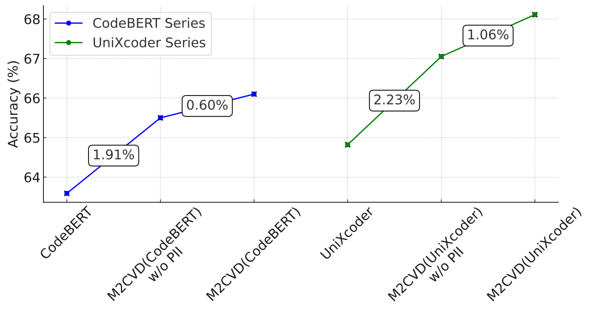


Figure 3: Effects of vulnerability description refinement for M2CVD.

5.2 Effects of vulnerability description refinement for M2CVD

In this section, we elaborate on the implications of Phase II feedback within the M2CVD framework on the performance of code vulnerability detection. In order to enhance the experimental validity, we use two different code models to conduct experiments. For this purpose, a comparative experiment was established using the Devign dataset with its default partitioning.

The results from the experiment as shown in the fig 3, which presents a comparison results of model accuracy under different M2CVD configurations. 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) 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) configuration with Phase II integration surpasses its UniXcoder counterpart by 3.29%.

The results confirm the proposed concept. By integrating vulnerability semantics into code data, we enhance the prediction accuracy of the pre-trained code model in vulnerability detection tasks. Concurrently, these results highlight the efficacy of the vulnerability semantic refinement process within the M2CVD framework. This process significantly boosts the code model's predictive capabilities during the final execution of the code judgment task.

5.3 Effects of vulnerability description refinement 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 code 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. And ALL"YES"("NO") for ChatGPT mean during refinement, we informed ChatGPT of the code model evaluations, but all of them were "YES."("NO.") This meant that we were mistakenly telling ChatGPT that every fragment of code was vulnerable or not vulnerable. The UniXcoder for CodeLLaMa(ChatGPT) means the standard M2CVD process, the LLMs was chosen as CodeLLaMa(ChatGPT).

The experiments reveal several insights: The experimental data in Table 2 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") or ("NO"), its accuracy in detecting code vulnerabilities increases from 45.29% to 50.04% and 52.12%. This suggests that ChatGPT benefits from additional context. Even by prompting him with in-

Table 2: LLMs Accuracy Comparison with different configurations in Phase II

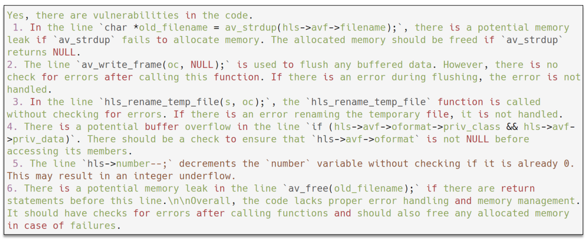
Model	**Accuracy**
ChatGPT	45.29%
ChatGPT-fewshot	45.30%
CodeLLaMa-13B	49.57%
UniXcoder for CodeLLaMa-13B	52.59%
ALL“YES” for ChatGPT	50.04%
ALL "NO" for ChatGPT	52.12%
UniXcoder for ChatGPT	57.61%

sufficient information to check the data again, the detection accuracy can be improved. 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 code model, which carries insights from its fine-tuning process, is crucial in helping LLMs better understand and evaluate the code in question.

6 Case Study

In this section, we show instances of LLMs generate code vulnerability semantics. This is the core idea of the M2CVD method. Figure 4 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 code 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.



```
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.
```

Figure 4: Semantic description of a vulnerability generated by LLMs. When the code is finally determined to be vulnerable, these descriptions are provided to the programmers to help maintain the code.

7 DISCUSSION

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 (Wei et al., 2022). 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 technology. 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.

8 CONCLUSION

In this paper, we introduced M2CVD, a novel method designed to address the challenge of software vulnerability detection by harnessing the combined strengths of pre-trained code models and large language models. The M2CVD integrates the language models such as ChatGPT and code 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.

9 Limitation

While our approach has yielded promising results, there are areas that require further improvement. Firstly, when processing long code snippets, including natural language descriptions intensifies the model's input length limitations. Secondly, our collaborative approach lowers false positives in datasets lacking enough defective code but doesn't significantly increase accuracy as it does in balanced datasets. Thirdly, only simple prompt are used in this paper, and using more complex and effective prompt maybe further improve the performance of M2CVD.

References

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*, pages 3011–3020.

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*, pages 1456–1468.

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

Checkmarx. 2022. Online. *Available: https://www.checkmarx.com/*.

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):1–33.

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*, pages 519–531.

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*.

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*, pages 4665–4671.

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*, pages 1536–1547.

Flawfinder. 2022. Online. *Available: http://www.dwheeler.com/ flawfinde/r*.

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*, pages 608–620.

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*.

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)*, pages 7212–7225.

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*, pages 596–607.

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

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):16–16.

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

Yi Li, Shaohua Wang, and Tien N Nguyen. 2021a. 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*, pages 292–303.

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

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*.

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*, pages 2539–2541.

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)*, pages 324–335. IEEE.

Llama. 2022. Online. *Available: https://ai.meta.com/llama/*.

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*, pages 178–182.

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*, pages 2006–2018.

openAI. 2022. Online. *Available: https://www.chat.openai.com/*.

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)*, pages 757–762. IEEE.

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

Benjamin Steenhoek, Md Mahbubur Rahman, Shaila Sharmin, and Wei Le. 2023b. Do language models learn semantics of code? a case study in vulnerability detection. *arXiv preprint arXiv:2311.04109*.

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*, pages 481–496.

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*, pages 8696–8708.

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*, 35:24824–24837.

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*, pages 2365–2376.

Fabian Yamaguchi. 2015. *Pattern-Based Vulnerability Discovery.* Ph.D. thesis, University of Göttingen.

Fabian Yamaguchi. 2017. Pattern-based methods for vulnerability discovery. *it-Information Technology*, 59(2):101–106.

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*.

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)*, pages 457–467. IEEE.

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.