HANDPICK:Construction of an LLM-Based Software Defect Prediction Dataset

Anonymous ACL submission

Abstract

Software defect prediction has become a significant research direction in software testing. The comprehensiveness of defect prediction directly impacts testing efficiency and program execution. In practical applications, there is a disconnect between detected software defects and their explanations or modification suggestions. Most methods stop at ranking the importance of static code features and fail to provide actionable repair recommendations. To address this issue, it is necessary to provide the location and a description of the predicted software defects. Therefore, this paper adopts the Common Weakness Enumeration (CWE) as the defect classification standard. Leveraging the remarkable capabilities demonstrated by large language models in code understanding tasks, we design structured prompts based on software engineering principles and prior defect knowledge for data sampling and labeling. Through a systematic analysis of the quality of the synthetic data, we identify a more suitable specific closed-source model pool. Experimental results demonstrate that our proposed method, Handpick—which automates the construction of software defect prediction datasets using large language models—can provide defect localization and repair suggestions during software defect prediction, thereby assisting developers in better rectifying software defects.(For dataset access inquiries, please contact us via email[1] at your convenience.)

1 Introduction

Software defect prediction is a crucial research area within software engineering. Its primary objective is to forecast potential future defects early in the development process, guiding the optimal allocation of testing resources, thereby reducing repair costs and enhancing software quality (Azam et al., 2022). Against the backdrop of expanding open-source project scales and the proliferation of automated development tools, defect prediction techniques should be integrated earlier into the entire software development life cycle to maximize software quality.

Existing research has made progress in software defect prediction; however, the increasing complexity of models has highlighted issues of interpretability: developers are unable to pinpoint the location of software defects and find it difficult to understand the model's decision-making logic, which limits trust and debugging efficiency in practical applications (Gezici Geçer and Kolukısa Tarhan, 2025).

Recently, large language models (LLMs) have been successfully applied to various code-related tasks (Wang et al., 2024; Zhang et al., 2023a,b). Their exceptional ability to understand and generate natural language explanations makes them promising candidates for automated code review. However, the application of LLMs in code review—particularly in automated software defect annotation—remains insufficiently explored.

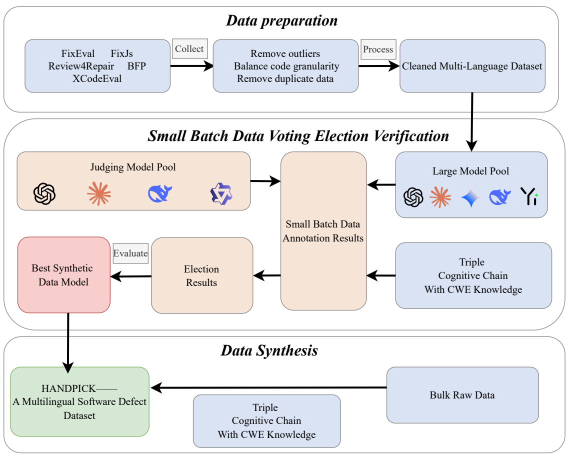

Data preparation
FixEval FixJs
Review4Repair BFP
XCodeEval
Collect
Remove outliers
Balance code granularity
Remove duplicate data
Process
Cleaned Multi-Language Dataset
Small Batch Data Voting Election Verification
Judging Model Pool
Small Batch Data Annotation Results
Large Model Pool
Best Synthetic Data Model
Evaluate
Election Results
Triple Cognitive Chain With CWE Knowledge
Data Synthesis
HANDPICK——
A Multilingual Software Defect Dataset
Triple Cognitive Chain With CWE Knowledge
Bulk Raw Data

Figure 1: How Does TriCogVuln-LLM Work?

[1]The data that support the findings of this study are available from the author, XXXXX, upon reasonable request. Interested readers may contact the author via email at mail to:XXXX@XXXX.com to request access to the dataset.

This study aims to bridge this gap by proposing an LLM-based automated software defect annotation method. Our goal is to construct a high-quality, multilingual dataset and enhance the interpretability of software defect prediction by introducing CWE (Common Weakness Enumeration), thereby advancing research in the direction of interpretable software defect prediction results. The main contributions include:

- **First LLM-Training-Ready Multilingual Dataset**: We have compiled and released HandPick, the open-source multilingual dataset specifically designed for LLMs training;
- **TriCogVuln-LLM Framework**: We introduce a novel framework, as illustrated in Figure 1, LLM-Enhanced Triple Cognitive Chain for Multilingual Code Vulnerability Mining with CWE Knowledge (TriCogVuln-LLM), designed to identify and extract vulnerability patterns in multilingual code;
- **Improving Software Defect Interpretability**:By locating the position of software defects and providing their CWE type, we help developers better undertake defect repair;
- **Task-Specific Evaluation Framework**:We have developed tailored evaluation methods for various tasks, specifically designed for the assessment of software defect prediction.

2 Related Work

This section reviews two main research areas relevant to this study: software defect prediction and the application of LLMs in software engineering.

2.1 Software Defect Prediction

Early methods relied on static code attributes (such as McCabe complexity) and statistical models (like logistic regression). With the advancement of machine learning (ML) and deep learning (DL), complex models such as Random Forest (RF), Gradient Boosting (GB), and Neural Networks (NN) have significantly improved prediction accuracy (Borandag, 2023; Li et al., 2021).

However, the prediction process of complex models (such as deep neural networks) is difficult to trace, making it impossible for developers to determine which code features lead to the defect prediction results. Although AST-based LSTM models excel in semantic feature extraction, their internal weights cannot be directly mapped to specific code logic (Ha et al., 2024). Existing explanation tools (such as LIME, SHAP) provide explanations through local approximations or feature importance ranking, but they may produce inconsistent results across different datasets or model settings. The explanations generated by LIME for the same prediction result can vary significantly under different sampling strategies, casting doubt on its reliability (Shin et al., 2021).

Meanwhile, the CWE offers a standardized defect classification system, facilitating efficient identification and handling of security vulnerabilities through unified terminology and classification methods. While CWE is widely adopted in industry, its integration with automated defect detection and labeling in academic research remains nascent, holding significant potential to enhance the accuracy and consistency of vulnerability detection(Nguyen et al., 2023; Ashraf et al., 2019; Kim et al., 2024).

2.2 Large Language Model

The widespread adoption of the Generative Pretrained Transformer (GPT) model (Ouyang et al., 2022) has demonstrated the substantial potential of LLMs in code-related tasks. Recent studies have explored the capabilities of LLMs in addressing various unique software engineering challenges (Cheng et al., 2024; Fan et al., 2024; Hou et al., 2024; Kulsum et al., 2024; Zhou et al., 2024b).

Prompt engineering is a critical step in interacting with LLMs to influence their responses. The characteristics of prompts, such as vocabulary, style, and tone, can significantly impact the responses generated by LLMs (Zamfirescu-Pereira et al., 2023). Well-crafted prompts can enhance the performance of LLMs in specific tasks. For instance, multi-hop Chain of Thought (CoT) (Wei et al., 2022) is a common prompt engineering technique that decomposes prompts into smaller, individual steps, thereby improving the reasoning abilities of LLMs. Wei et al. (Wei et al., 2022) introduced the CoT prompting strategy, which guides LLMs to generate intermediate reasoning steps, thereby enhancing their ability to solve complex problems.

Prompt engineering has proven to be highly effective in code-related tasks, enabling LLMs to overcome many limitations of earlier techniques (Hou et al., 2024; Liu et al., 2023). Concurrently, the power of LLMs has led researchers to leverage these models for vulnerability-related tasks, with a majority of prior work focusing on vulnerability detection (Zhou et al., 2024a). Notably, while the aforementioned studies focus on the application of LLMs to existing software defect detection datasets, our work aims to construct a model pool using high-performance closed-source large models, achieving data synthesis for code defect prediction and repair suggestions through multi-hop CoT. This approach not only improves the diversity and accuracy of annotations but also opens new avenues for code analysis as a structured information processing task.

3 Methodology

3.1 Original Multi-Programming Language Datasets

The original multi-programming language dataset we collected consists of code samples from four programming languages: Java, C (C++), Python, and JavaScript. This datasets includes both the original code and the corresponding fixed code. In this study, we exclusively utilized datasets that contain both pre-fix and post-fix code. For detailed information, please refer to Appendix A. Due to inconsistencies in data quality, the datasets underwent preprocessing, as detailed in Section 4.2.

3.2 Three-Step Chain of Thought Prompt

Conventional software defect prediction primarily focuses on binary classification of defective code segments. Our initial framework adopted a two-stage pipeline: "CWE Defect Prediction → Defect Repair Suggestion Generation" to enhance the task architecture. Nevertheless, code defects arise not merely from semantic patterns but are substantially governed by functional specifications. Given the frequent oversight of functional requirements in existing approaches, we propose incorporating a preliminary phase: "Code Functional Description." This additional step enriches contextual awareness for subsequent defect analysis. Consequently, we refine the workflow into three core components: (1) Functional Description Generation, (2) CWE Defect Prediction, and (3) Defect Repair Suggestion Generation. This progressive task decomposition demonstrates dual advantages in improving predictive performance and strengthening data interpretability.

3.2.1 Function Description Generation

During the first phase of our reasoning pipeline, the model generates functional descriptions for given code segments. The user instruction is structured as: "Analyze the following code and concisely describe its core functionality within 30 words:" accompanied by target code code. This setup requires the model to reconstruct the intended behavior from implementation patterns. Our defect prediction analysis reveals that the prediction scopes under unknown(ϕ_1) versus known functionality(ϕ_2) conditions must conform to:

$$\phi_1 \cap \phi_2 = \phi_3 \quad \text{and} \quad \phi_3 \neq \emptyset \tag{1}$$

This constraint necessitates explicit requirement integration, directing the predictor to identify defects within constrained operational contexts. Such targeted detection enables context-aware repair generation that preserves functional specifications while adhering to practical software engineering constraints.

3.2.2 CWE Defect Prediction

Upon completing functional description generation, the framework initiates defect detection within the formally defined context ϕ_3. The system prompt strategically configures three key elements for the LLM: (a) simulated code inspector roles, (b) applicable static analysis techniques, and (c) security-focused operational constraints. Corresponding user prompts deliver task-specific guidance through: (i) structured domain knowledge injection, (ii) defect pattern specifications, and (iii) rigorous output formatting requirements. Our prompt engineering methodology integrates empirical evidence from multiple sources. We interviewed several frontline developers and collected insights on software testing and code review. These insights, when cross-validated with historical testing logs, form the foundation of our diagnostic rules. Furthermore, longitudinal analysis of CWE-TOP25 rankings (2019-2024) reveals 10 persistently prevalent vulnerability types, designated as CWE-TOP10. Their characteristic patterns are encoded into prompt templates using security taxonomies. The prediction workflow mandates JSON-structured outputs.

3.2.3 Defect Repair Suggestion Generation

Based on the technical overlap between defect prediction and code repair, we optimized the system prompt for CWE defect prediction through three

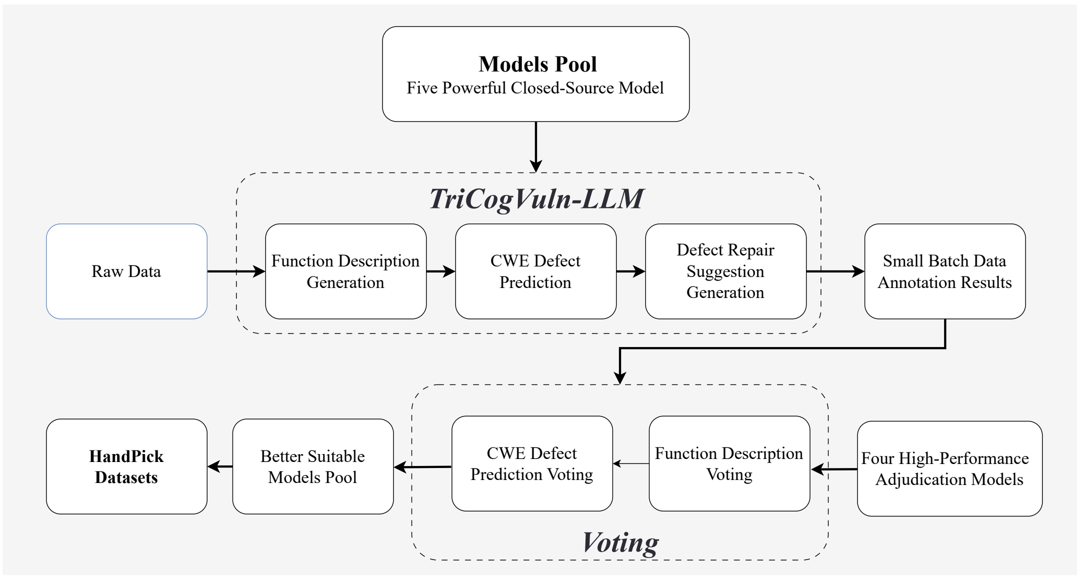

Models Pool
Five Powerful Closed-Source Model
TriCogVuln-LLM
Raw Data
Function Description Generation
CWE Defect Prediction
Defect Repair Suggestion Generation
Small Batch Data Annotation Results
HandPick Datasets
Better Suitable Models Pool
CWE Defect Prediction Voting
Function Description Voting
Four High-Performance Adjudication Models
Voting

Figure 2: Flowchart of the Main Methods for Dataset Construction

key modifications: (1) contextual role specification, (2) elimination of redundant instructional components, and (3) integration of enhanced developmental context with technical background specifications.

3.3 Model Pool

Given the proliferation of diverse large language models (LLMs) that all claim superior code comprehension capabilities, we propose employing a model collective to generate synthetic data. This methodology enhances distribution diversity in training datasets, effectively mitigating homogeneity-induced bias. Our initial selection comprises five state-of-the-art closed-source LLMs, with rigorous validation experiments conducted per model to ensure optimal data quality during large-scale synthesis operations.

3.4 Voting

Following small-scale annotation acquisition, we established an evaluation framework exploring multiple approaches before adopting a minimalist LLM consensus voting mechanism adhering to Occam's razor principles. Our methodology requires models to identify underperforming peers through comparative analysis. Given the absence of ground-truth labels, we operationalized a key assumption: models exhibiting maximum distributional divergence represent system outliers. This reduces our task to optimal voter selection, prioritizing balanced trade-offs between variance control, bias mitigation, and computational costs - therefore chosen from our initial model collective. To counter inherent self-preference tendencies in LLM probability matching, we implemented external validation through an independent domain expert.

3.5 Evaluation

Given the synthetic nature of our final dataset – generated through model collective sampling and devoid of defect prediction ground-truth labels – we employ a dual-strategy quality assessment: 1) voting-phase optimal performer outputs as pseudo-labels, and 2) integration with original "fix_code" instances. To address task-specific evaluation requirements across our three experimental scenarios, we developed a multi-dimensional metric framework:

$$\begin{aligned} Score = \frac{1}{4}(&\cos(sample_1, pseudo_1) \\ &+ Jaccard(sample_2, pseudo_2) \\ &+ Similarity(sample_3, fix\_code)) \end{aligned} \quad (2)$$

In Equation (2), let $sample_i$ and $pseudo_i$ respectively represent the ith iteration results from model collective sampling and pseudo-label generation. For the functional description task producing concise outputs: (1) We embed both model collective outputs and p_1 through an embedding model, then compute pairwise cosine similarities. The CWE defect prediction task outputs are structured as an m×n binary sparse matrix – with m evaluation samples and n CWE taxonomy IDs – where we compute Jaccard similarity scores. For defect repair evaluation involving "fix_code": (2) We perform lexical analysis on both original fix_code and synthetic data, defining their similarity metric as:

$$Similarity(sample_3, fix\_code) = \frac{1}{n}$$
$$(\sum repetition(sample_3, fix\_code)) \quad (3)$$
$$+ \min(1, \frac{extra(sample_3, fix\_code)}{n})$$

Here, $repetition()$ represents the code matching rate of the matched defect repair, $extra()$ represents the additional defect repairs by the LLMs, and n represents the number of lines modified by fix_code.

4 Experiments

In this section, we will present the experimental setup, describe the data collection methodologies, detail the implementation specifics, and assess the quality of the synthetic datasets.

4.1 Experimental Setup

To generate high-quality software defect prediction datasets, we selected five models including GPT-4o, DeepSeek V3, Claude-3.5-Sonnet, Gemini-1.5-Pro-latest, and Yi-lightning. These models have demonstrated robust performance across multiple benchmarks and are capable of handling complex programming tasks. (See Appendix C for detailed model configurations.)

To enhance the quality and reliability of our dataset, we employed a voting verification approach using annotation results from 500 data samples across different models, to select large language models that offer a balance between performance and economic cost. In addition to the annotation models, we used Qwen-max as an external expert to the voting process. This inclusion increased the diversity of the voting models and mitigated the impact of single-model bias on the voting results.

4.2 Data Collection

The datasets utilized in this study was compiled from multiple publicly available datasets for software defect identification and bug2fix(Haque et al., 2023; Huq et al., 2022; Tufano et al., 2018; Khan et al., 2023; Csuvik and Vidács, 2022), encompassing four mainstream programming languages: Java, C/C++, Python, and JavaScript. The objective was to construct a diversified, large-scale defect prediction datasets.

Due to the presence of outliers, duplicate data, and inconsistent code granularity in the original code dataset, preprocessing of the dataset is essential. The preprocessing steps are as follows:

Outlier Removal: Eliminate code segments that are excessively long or short, and verify the integrity of the remaining code.

Code Granularity Unification:Utilize regular expressions to identify and standardize the granularity of classes and functions.

Duplicate Data Removal:Remove duplicate data entries to ensure uniqueness.

Token Calculation:Estimate the average token size for each subset of the dataset.

Finally, all data were converted into a unified JSON format to facilitate subsequent processing and model input, yielding a clean, standardized dataset suitable for defect prediction tasks. In subsequent experiments, 500 data points were selected for small-scale experiments, and 25,000 data points were chosen for large-scale annotation.

4.3 Implementation Details

This section introduces the data annotation process and the model voting and selection procedure.

4.3.1 Data Annotation Process

The data annotation process was divided into two phases: small-batch data annotation by various models, and large-scale data annotation by the model pool.

In the small-batch annotation phase, we randomly selected 500 data points from the preprocessed dataset and independently annotated them using the five LLMs described in Section 4.1. We employed the multi-step prompting strategy outlined in Section 3.2: 1) identifying the code functionality; 2) predicting potential vulnerabilities or issues; and 3) modifying the code. After completing the small-batch annotation, we obtained different annotation results from the five models for the same batch of data, establishing the foundation for subsequent model evaluation and selection.

In the large-scale annotation phase, we employed the model pool selected through evaluation (detailed in Section 4.4) to annotate the remaining 25,000 data points. The annotation process remained consistent with that of the small-batch phase.

4.3.2 Best Annotation Model Selection

To select a large-scale annotation model pool that achieves the optimal balance between performance and cost, we implemented a voting mechanism to systematically eliminate underperforming annotation models.

We employed the voting method proposed in Section 3.4. Specifically, the system prompt for the voting models was configured to simulate the role of a "teacher" responsible for evaluating the quality of the "students" (LLMs used in the annotation phase) and identifying the "least appropriate" item. The first task of "functionality description" and the second task of "CWE defect prediction" were prioritized, while the third task ("defect repair suggestion generation") was heavily dependent on the output of the first two tasks and therefore did not participate in the voting process. The detailed user prompts and system prompts can be found in Appendix D.

Although both the first and second rounds of voting involved "semantic difference identification," there were subtle distinctions between them. The output of the "functionality description" task was relatively straightforward, with voting models being asked to select "at most one" inappropriate item in this round. The "CWE defect prediction" task was more challenging, as differences in model capabilities and probability distributions could lead to significant variations in results. Consequently, voting models were instructed to select "at least one" inappropriate item in this round.

4.4 Voting Results and Analysis

4.4.1 First Voting

We expect a good voting model to exhibit low bias and low variance. Due to the lack of ground truth labels, the first round of voting will focus on the variance metric. For the initial attempt, we selected voting models from the model pool, favoring cost-effective options. Therefore, we chose "DeepSeek V3" and "Yi-lightning" for the first round of voting. Each voting model conducted three votes, with the results shown in Figure 3 .

Based on the analysis of Figures 3, the following preliminary conclusions can be drawn:

Yi-lightning: The variance in the first round was 24.5333, and in the second round, it was 81.5333.

DeepSeek V3: The average variance in the first round was 15.2, and in the second round, it was 19.15, making it more reliable than Yi-lightning.

Claude-3.5-Sonnet: Performed poorly in the first round and was unsuitable for the "functionality description" task.

Gemini-1.5-Pro-latest: Was deemed "inappropriate" by both voting models and effectively rejected.

GPT-4o: Performed the best in evaluations by both voting models.

Yi-lightning exhibited relatively high variance and was often self-eliminated in the voting process, leading to its elimination, leaving DeepSeek V3.

4.4.2 Second Voting

GPT-4o demonstrated good performance during the evaluation process. Consequently, we used GPT-4o for the second round of voting, obtaining the results shown in Figure 3:

GPT-4o: The average variances for the two rounds were 27.8333 and 64.1833, respectively.

DeepSeek V3: Exhibited lower variance than GPT-4o, further supporting the reliability of DeepSeek V3's variance. The voting trends were similar, indicating that its bias is also acceptable.

Claude-3.5-Sonnet: Should not be entirely dismissed, as it performed well in "defect prediction."

Additionally, we conducted an ablation study using Qwen-max as an external expert. The results were similar to previous findings, and further details are provided in Appendix E.

4.4.3 Target Voting Model & Model Pool

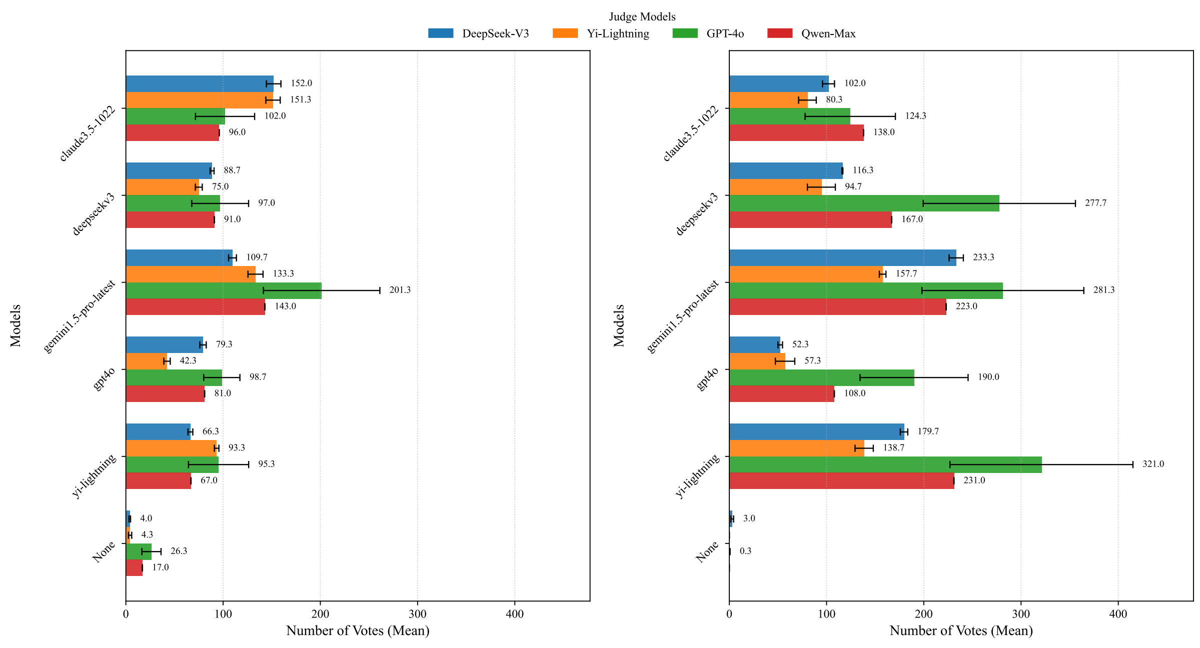
The comprehensive voting experiments concluded that DeepSeek V3 is a viable voting model. The composition of the model pool for subsequent large-scale data synthesis is as follows:

"Functionality Description": Composed of (GPT-4o, DeepSeek V3) with a sampling ratio of 1:2.

"CWE Defect Prediction": Composed of (GPT-4o, DeepSeek V3, Claude-3.5-Sonnet) with a ratio of 4:3:3.

"Defect Repair Suggestion Generation": This task builds on the first two tasks and is less challenging. Gemini-1.5-Pro-latest is also included, forming a model pool of (GPT-4o, DeepSeek V3, Claude-3.5-Sonnet, Gemini-1.5-Pro-latest) with a ratio of 1:1:1:1.

Throughout the process, we adhered to the principle of balancing performance, cost, speed, and diversity in model selection.



(A) The voting results of the first round by the Judge Models.

(B) The voting results of the second round by the Judge Models.

Figure 3: Validation Results of the Voting Model on Small-Scale Datasets

4.5 Evaluation of Synthetic Datasets Quality

4.5.1 Dataset Quality

After the voting process, we finalized the model pool used for large-scale data synthesis, annotating a total of 25,000 data entries. We then assessed the models using the method outlined in Section 3.5. Throughout the voting process, GPT-4o consistently outperformed the other models. Therefore, in the "Functionality Description" and "CWE Defect Prediction" tasks, we used the output from GPT-4o as "pseudo-labels" for reference. Additionally, we integrated the fix_code from the original dataset for a third round of scoring to ensure a comprehensive evaluation. From the generated dataset, we selected 500 entries for quality assessment.

For the **"Functionality Description**" task, we utilized the "m3e" embedding model to process the synthetic data from GPT-4o and the sampled data from our model pool. We then calculated the cosine similarity for 500 pairs of data entries, resulting in an average similarity score of 0.74.

In the **"CWE Defect Prediction"** task, we extracted CWE type numbers from the outputs of GPT-4o and our model pool. Following the evaluation method in Section 3.5, we counted the occurrences of each CWE defect type. Both GPT-4o and our sampled data included 34 different CWE types, with 18 types being common between them. The most frequently occurring CWE type was "CWE-20 (Improper Input Validation)." Instances where no defect was predicted were recorded as "CWE-0." This process resulted in two binary sparse matrices of shape (500, 50). We calculated the Jaccard similarity between these matrices, which scored 0.54. Additionally, the number of "pass!" instances in our model pool results was 75, compared to 92 for GPT-4o, further demonstrating the superiority of our model pool strategy.

For the **"Defect Repair Suggestion"** task, we compared the repair outputs of the LLMs at the "line" level with the fix_code from the original dataset and the differences from the original code, calculating the intra-line duplication rate. The scores for GPT-4o and our model pool were 0.6062 and 0.6190, respectively. This indicates that our method outperformed GPT-4o compared to the ground truth in the original dataset, resulting in a final "Defect Repair Suggestion" task score of 1.62.

4.5.2 Dataset Quality Analysis

The overall quality assessment score for our dataset is approximately 72.5, indicating that the dataset is very satisfactory. We believe the quality of the dataset exceeds the current evaluation score for the following reasons: 1)The SDK used for the data (including library classes, etc.) is outdated, and updates in subsequent versions have resolved these issues, preventing LLMs from predicting them.2)Some defects are related to functional requirements, for which we lack the necessary data.3)The "CWE Defect Prediction" task scored lower due to its inherent challenges and exhibited high variance during the voting process, suggesting that GPT-4o might not be suitable as a "pseudo-label." Therefore, we consider our dataset to be superior to the current evaluation results.

4.6 Engineering Practice

We used handpick to train **Qwen2.5-14B-Instruct** through multi-turn dialogues to predict our actual engineering code and conducted manual evaluations. The feedback received was qualitative: the model indeed has practical effects, but the current business code is highly encapsulated, preventing access to all code information, so some defects exist but do not need to be considered. We further conducted experiments using the public dataset MegaVul; detailed experiments are provided in Appendix F. The experimental results indicate that the Qwen2.5-14B-SFT model, when fine-tuned with our HandPick dataset, achieved significant improvements across key metrics for software defect prediction. Its performance not only surpassed that of the original model but also exceeded some of the compared large language models, such as DeepSeekV3-671B and Gemini 1.5 Pro, thereby robustly demonstrating the effectiveness of the HandPick dataset and the associated fine-tuning strategy in enhancing the model's software defect prediction capabilities.

5 Conclusion

In this study, we contributed from three key dimensions: 1) We proposed a multilingual, LLM-trainable, and interpretable defect prediction dataset—handpick; 2) We developed a chain-of-thought framework specifically designed for defect prediction tasks, TriCogVuln-LLM, which balances data synthesis cost, data quality, and diversity; 3) We designed a corresponding evaluation method for TriCogVuln-LLM. Notably, our dataset scored a high 72.5 using our designed evaluation method, highlighting its effectiveness in defect prediction across different programming languages. By addressing the lack of interpretability in existing defect prediction datasets, our work provides a novel perspective and resources to advance defect prediction research. To foster further collaboration, we have made part of the HandPick dataset publicly available on HuggingFace.[2] and GitHub[3].

6 Limitations

Currently, our dataset only covers four common programming languages. Given that our primary application scenario is centered around Chinese and Java, there is a noticeable lack of data exploration in other programming languages and English. Although our experimental design and ablation studies are methodologically robust, we must acknowledge the limitation of the dataset lacking real, manually verified labels, which remains an unresolved issue. Additionally, the inherent limitations of large language models (LLMs), including hallucinations and limited capabilities, further impair the quality of the dataset. Looking forward, we plan to optimize our framework tasks and related prompts, expand our dataset by incorporating a wider range of programming languages, and address the dataset quality issue with a clear strategy: annotating more defect data, using defect prediction models in actual development environments, collecting human feedback and more data in real scenarios, and applying an iterative self-training approach to gradually enhance the dataset quality and LLM performance.

References

Hafsa Ashraf, Mamdouh Alenezi, Muhammad Nadeem, and Yasir Javid. 2019. Security assessment framework for educational erp systems. *International Journal of Electrical and Computer Engineering*, 9(6):5570.

Muhammad Azam, Muhammad Nouman, and Ahsan Rehman Gill. 2022. Comparative analysis of machine learning technique to improve software defect prediction: Comparative analysis of machine learning technique to improve software defect prediction. *KIET Journal of Computing and Information Sciences*, 5(2).

Emin Borandag. 2023. Software fault prediction using an rnn-based deep learning approach and ensemble machine learning techniques. *Applied Sciences*, 13(3):1639.

Yiran Cheng, Lwin Khin Shar, Ting Zhang, Shouguo Yang, Chaopeng Dong, David Lo, Shichao Lv, Zhiqiang Shi, and Limin Sun. 2024. Llm-enhanced static analysis for precise identification of vulnerable oss versions. *arXiv preprint arXiv:2408.07321*.

[2]https://huggingface.co/datasets/pansysy/handdppick
[3]https://anonymous.4open.science/r/handdppick-5F25/

Viktor Csuvik and László Vidács. 2022. Fixjs: A dataset of bug-fixing javascript commits. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 712–716.

Lishui Fan, Jiakun Liu, Zhongxin Liu, David Lo, Xin Xia, and Shanping Li. 2024. Exploring the capabilities of llms for code change related tasks. *ACM Transactions on Software Engineering and Methodology*.

Bahar Gezici Geçer and Ayça Kolukısa Tarhan. 2025. Explainable ai framework for software defect prediction. *Journal of Software: Evolution and Process*, 37(4):e70018.

Thi Minh Phuong Ha, Thi Kim Ngan Nguyen, and Thanh Binh Nguyen. 2024. A comparative study of deep learning techniques in software fault prediction.

Md Mahim Anjum Haque, Wasi Uddin Ahmad, Ismini Lourentzou, and Chris Brown. 2023. Fixeval: Execution-based evaluation of program fixes for programming problems. In *2023 IEEE/ACM International Workshop on Automated Program Repair (APR)*, pages 11–18. IEEE.

Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 33(8):1–79.

Faria Huq, Masum Hasan, Md Mahim Anjum Haque, Sazan Mahbub, Anindya Iqbal, and Toufique Ahmed. 2022. Review4repair: Code review aided automatic program repairing. *Information and Software Technology*, 143:106765.

Mohammad Abdullah Matin Khan, M Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. 2023. xcodeeval: A large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. *arXiv preprint arXiv:2303.03004*.

Donghyun Kim, Seungho Jeon, Kwangsoo Kim, Jaesik Kang, Seungwoon Lee, and Jung Taek Seo. 2024. Guide to developing case-based attack scenarios and establishing defense strategies for cybersecurity exercise in ics environment. *The Journal of Supercomputing*, pages 1–34.

Ummay Kulsum, Haotian Zhu, Bowen Xu, and Marcelo d'Amorim. 2024. A case study of llm for automated vulnerability repair: Assessing impact of reasoning and patch validation feedback. In *Proceedings of the 1st ACM International Conference on AI-Powered Software*, pages 103–111.

Zhen Li, Tong Li, YuMei Wu, Liu Yang, Hong Miao, and DongSheng Wang. 2021. Software defect prediction based on hybrid swarm intelligence and deep learning. *Computational Intelligence and Neuroscience*, 2021(1):4997459.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36:21558–21572.

Dinh Huong Nguyen, Aria Seo, Nnubia Pascal Nnamdi, and Yunsik Son. 2023. False alarm reduction method for weakness static analysis using bert model. *Applied Sciences*, 13(6):3502.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744.

Jiho Shin, Reem Aleithan, Jaechang Nam, Junjie Wang, and Song Wang. 2021. Explainable software defect prediction: Are we there yet? *arXiv preprint arXiv:2111.10901*.

Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. An empirical investigation into learning bug-fixing patches in the wild via neural machine translation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 832–837.

Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*.

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.

JD Zamfirescu-Pereira, Richmond Y Wong, Bjoern Hartmann, and Qian Yang. 2023. Why johnny can't prompt: how non-ai experts try (and fail) to design llm prompts. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, pages 1–21.

Quanjun Zhang, Chunrong Fang, Yang Xie, Yaxin Zhang, Yun Yang, Weisong Sun, Shengcheng Yu, and Zhenyu Chen. 2023a. A survey on large language models for software engineering. *arXiv preprint arXiv:2312.15223*.

Ziyin Zhang, Chaoyu Chen, Bingchang Liu, Cong Liao, Zi Gong, Hang Yu, Jianguo Li, and Rui Wang. 2023b. Unifying the perspectives of nlp and software engineering: A survey on language models for code. *arXiv preprint arXiv:2311.07989*.

Xin Zhou, Sicong Cao, Xiaobing Sun, and David Lo. 2024a. Large language model for vulnerability detection and repair: Literature review and the road ahead. *ACM Transactions on Software Engineering and Methodology*.

Xin Zhou, Kisub Kim, Bowen Xu, DongGyun Han, and David Lo. 2024b. Out of sight, out of mind: Better automatic vulnerability repair by broadening input ranges and sources. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13.

A Data Collection

We gathered datasets from multiple programming languages, conducted data preprocessing, and the basic characteristics of the datasets are detailed in Table 1:

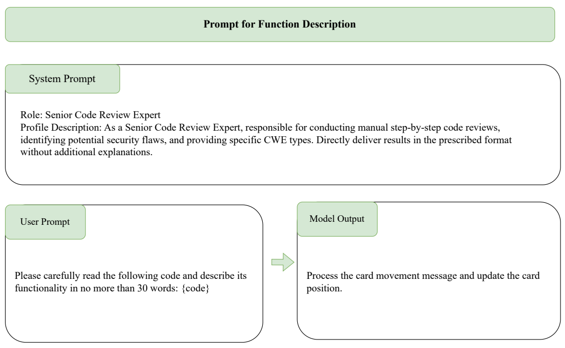
The FixEval datasets(Haque et al., 2023) is designed for evaluating program repair models, featuring pairs of buggy and fixed code in Java and Python. Data is sourced from programming competition platforms (e.g., AtCoder, Aizu Online Judge), with high complexity and problem difficulty levels (A-E). The extensive combinatorial search space necessitates a thorough understanding of the task for effective repair.

The Review4Repair datasets(Huq et al., 2022), targeting Java programs, includes 55,060 training and 2,961 test data points, leveraging code review (CR) information to facilitate repair.

Proposed by Tufano et al., the BFP datasets(Tufano et al., 2018) employs neural machine translation (NMT) to learn vulnerability repair models. Researchers extracted commits with the keyword "bug fix" from GitHub Archive, identifying around 10 million potential vulnerability repairs. Manual sampling confirmed 97.6% as genuine repairs, with the datasets focusing on small methods ($\leq$50 tokens).

XcodeEval(Khan et al., 2023), the largest multi-language, multi-task code benchmark, spans 17 programming languages and includes approximately 75,000 unique problems. It supports tasks such as code understanding, generation, translation, and retrieval, derived from competitive programming with a focus on advanced programming and mathematics.

Introduced by Viktor Csuvik and Laszlo Vidács in 2022, the FixJS datasets (Csuvik and Vidács, 2022) concentrates on JavaScript bug-fix commits. It was curated by selecting popular JavaScript projects from platforms like GitHub and analyzing version control history (e.g., git commits) to extract relevant bug-fix submissions.

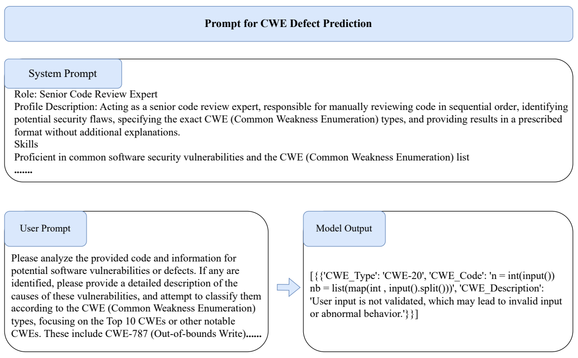


Role: Senior Code Review Expert
Profile Description: As a Senior Code Review Expert, responsible for conducting manual step-by-step code reviews, identifying potential security flaws, and providing specific CWE types. Directly deliver results in the prescribed format without additional explanations.

Please carefully read the following code and describe its functionality in no more than 30 words: {code}

Process the card movement message and update the card position.

Figure 4: Prompt For Function Description



Role: Senior Code Review Expert
Profile Description: Acting as a senior code review expert, responsible for manually reviewing code in sequential order, identifying potential security flaws, specifying the exact CWE (Common Weakness Enumeration) types, and providing results in a prescribed format without additional explanations.
Skills
Proficient in common software security vulnerabilities and the CWE (Common Weakness Enumeration) list
....

Please analyze the provided code and information for potential software vulnerabilities or defects. If any are identified, please provide a detailed description of the causes of these vulnerabilities, and attempt to classify them according to the CWE (Common Weakness Enumeration) types, focusing on the Top 10 CWEs or other notable CWEs. These include CWE-787 (Out-of-bounds Write)......

```
[{{'CWE_Type': 'CWE-20', 'CWE_Code': 'n = int(input())
nb = list(map(int , input().split()))', 'CWE_Description':
'User input is not validated, which may lead to invalid input
or abnormal behavior.'}}]
```

Figure 5: Prompt For CWE Detection

B Three-Step Chain of Thought Prompt

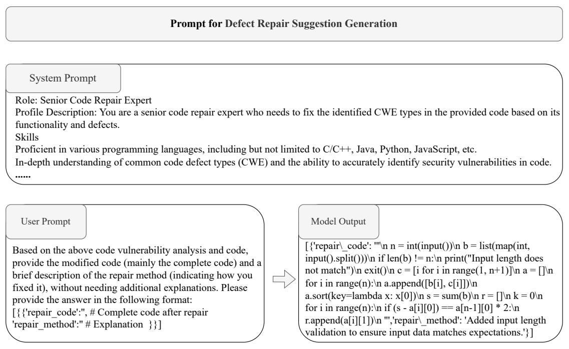
In our approach, we designed a set of Three-Step Chain of Thought Prompts to guide Large Language Models (LLMs) in data sampling and annotation, thereby automating the construction of software defect prediction datasets. This set of prompts targets three core stages: function description generation, CWE (Common Weakness Enumeration) defect type prediction, and defect repair suggestion generation.

The prompts for the function description generation stage are designed to guide the model to infer the intended functionality based on the code's structure and content, and to generate a concise description of the code's functionality. The detailed prompts utilized for this task are illustrated in Figure 4 . This figure details the specific content of the System Prompt, User Prompt, and provides an example of the model's response.

For the CWE defect prediction task, we developed specialized prompts by drawing upon software engineering knowledge and software testing experience. As shown in Figure 5 , these prompts instruct the model to act as a senior code review expert, meticulously examining the code, identifying potential security flaws, and specifying the precise

Language Type	Datasets Name	Data Size	Original Task Type
Java	FixEval	43000	Bugfix
	Review4Repair	59172	Bugfix
	BFP	1190331	Program Repair
	XCodeEval	574448	Program Repair
C++ (C)	XCodeEval	3409220	Program Repair
Python	XCodeEval	461356	Program Repair
JavaScript	FixJs	55551	Bugfix

Table 1: Datasets used in the experiment


Prompt for Defect Repair Suggestion Generation
System Prompt
User Prompt
Model Output

Role: Senior Code Repair Expert
Profile Description: You are a senior code repair expert who needs to fix the identified CWE types in the provided code based on its functionality and defects.
Skills
Proficient in various programming languages, including but not limited to C/C++, Java, Python, JavaScript, etc.
In-depth understanding of common code defect types (CWE) and the ability to accurately identify security vulnerabilities in code.
......

Based on the above code vulnerability analysis and code, provide the modified code (mainly the complete code) and a brief description of the repair method (indicating how you fixed it), without needing additional explanations. Please provide the answer in the following format:
[{{'repair_code':'', # Complete code after repair
'repair_method':'' # Explanation }}]

```
[{'repair\_code': '''\n n = int(input())\n b = list(map(int, input().split()))\n if len(b) != n:\n print("Input length does not match")\n exit()\n c = [i for i in range(1, n+1)]\n a = []\n for i in range(n):\n a.append([b[i], c[i]])\n a.sort(key=lambda x: x[0])\n s = sum(b)\n r = []\n k = 0\n for i in range(n):\n if (s - a[i][0]) == a[n-1][0] * 2:\n r.append(a[i][1])\n ''','repair\_method': 'Added input length validation to ensure input data matches expectations.'}]
```

Figure 6: Prompt for Defect Repair Suggestion Generation

CWE types. The figure also includes the detailed System Prompt, User Prompt, and an example of the model's output.

Building upon the system prompt for "CWE Defect Prediction," we designed corresponding prompts for generating defect repair suggestions, aiming to leverage LLMs to repair the identified defective code. The prompts used for this task are depicted in Figure 6 . These prompts require the model to assume the role of a senior code repair expert, providing the complete repaired code along with a brief Chinese explanation of the repair method.

C Large Language Model Pool

GPT-4o-2024-11-20(OpenAI): GPT-4o, developed by OpenAI, represents the latest advancement in language models, building upon GPT-4 with enhanced reasoning capabilities, faster response times, and improved multimodal understanding. GPT-4o excels in various NLP and code generation tasks.

DeepSeek V3 (DeepSeek): DeepSeek V3, the newest model from DeepSeek, is specifically tailored for code understanding and generation. It leads in multiple code-related benchmarks, particularly in managing complex code logic and producing high-quality code.

Claude-3.5-Sonnet-20241022 (Anthropic): Claude-3.5-Sonnet, part of Anthropic's Claude 3 series, is renowned for its robust security and reliability, alongside advanced natural language understanding and generation capabilities. It performs exceptionally in tasks demanding high security and reliability.

Gemini-1.5-Pro-latest (Google): Gemini-1.5-Pro, Google's latest multimodal large model, excels in processing and generating text, images, audio, and other data types, offering superior performance in cross-modal understanding tasks.

Yi-lightning (01.AI): Yi-lightning, a high-performance variant of the Yi series by 01.AI, is celebrated for its efficient inference speed and strong performance, with Yi-lightning pushing the boundaries in speed without compromising on performance.

Qwen-Max (Alibaba): Qwen-Max, the latest iteration in the Qwen series developed by Alibaba, represents a significant enhancement over its predecessors. It boasts superior reasoning capabilities, enhanced multimodal processing efficiency, and an expanded range of applications. Qwen-Max excels in various domains, including natural language processing, code generation, and multimodal tasks, with notable proficiency in complex logical reasoning and cross-modal comprehension. Beyond its technical superiority, Qwen-Max demonstrates exceptional stability and reliability in practical deployments, offering robust support for enterprise-level users.

D Prompt for Expert Model Election Voting

To select the optimal annotation model from a pool of candidates, we designed a specialized set of prompts for an election voting process. This voting process aims to comprehensively evaluate the overall capabilities of candidate models in understanding and predicting software defects through a two-stage assessment.

The first stage of voting focuses on the quality of function descriptions generated by each candidate model, primarily assessing their consistency and completeness. The second stage targets the CWE (Common Weakness Enumeration) defect type predictions made by candidate models, with a key evaluation of the congruence between their predictions and the actual defects in the code. Through this multi-dimensional evaluation, we aim to select the top-performing model ensemble that reflects a consensus.

The complete prompts for this expert model election voting process, encompassing the role definition for the language model executing the voting task, detailed review guidelines, specific user instructions for both voting rounds, and the expected output format, are consolidated and presented in Figure 7 .

E Ablation Experiments And The Corresponding Analysis Of Results

Given the critical importance of determining the voting model, we conducted extensive ablation experiments and analyzed the results. This appendix presents three ablation experiments and their conclusions, along with an additional related result analysis.

1. Fairness of the Model Voting Mechanisms

The voting models currently employed are all drawn from the initial model pool (GPT-4o, DeepSeek V3, Claude-3.5-Sonnet, Gemini-1.5-Pro-latest, Yi-lightning). Consequently, we are concerned that models may favor data aligning with their own probability distributions, such as knowledge distribution or syntactic structure, potentially leading to a reluctance to vote against themselves. Although we observed that the Yi-lightning model does not appear to favor its own data during "semantic difference recognition," the risk remains significant when a model serves as both a participant and an evaluator. To address this, we introduced an external expert, Qwen-max, to perform the same voting task. However, we conducted only one round of voting to assess whether the aforementioned risk necessitates attention. The results, depicted in Figure 8, suggest that concerns regarding the fairness of the models are unwarranted.

2. Effects of Including Both Pre-fix and Post-fix Code

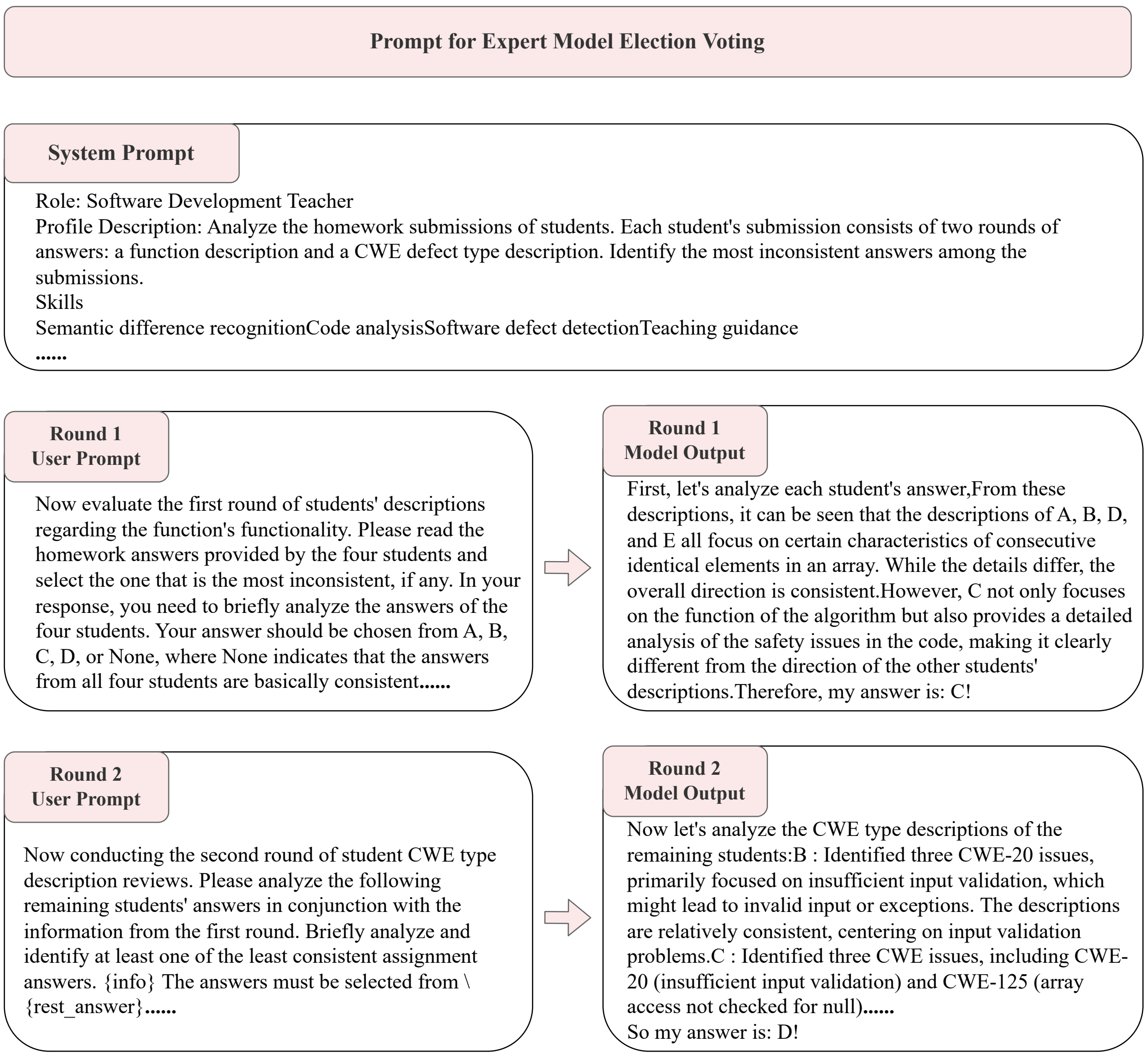
Our datasets includes instances with both pre-fix and post-fix code, whereas our proposed work focuses solely on predicting the original code. Therefore, we explored the potential utility of the post-fix code. During the initial design of the annotation prompt, we considered incorporating it, but this approach poses risks. Including optimized code might cause the model to focus more on the differences between pre-fix and post-fix code rather than the code itself or functional defects. This could result in identifying more errors or eliminating fewer models, deviating from the original goal of semantic difference recognition and potentially compromising annotation quality. Nonetheless, we proceeded with this ablation experiment. The results, illustrated in Figure 9, are noteworthy. When the original code is present, Claude-3.5-Sonnet exhibits a "polarization," becoming a highly "reliable" model. While we have speculated on the underlying reasons, we conclude that Claude-3.5-Sonnet warrants further exploration and consideration for inclusion in the model pool.

3. Impact of the First Round on the Second Round

In our previous task setup, models eliminated in the first round do not participate in the second round. Given that the voting models rarely vote "none" in the first round, and our subjective belief that "function description" is a relatively simple task, we posed the question: What is the impact of the first round on the second round? Consequently, we bypassed the first round of voting and directly conducted the second round, with the voting results shown in Figure 10 . This outcome is significant because, in prior model voting, Claude-3.5-Sonnet was seldom voted out in the second round, with two potential explanations: First, too many Claudes were voted out in the first round; Second, Claude-3.5-Sonnet is indeed "usable" in the second round. This ablation experiment clarifies that the latter explanation is accurate.

4. CWE Labeling by Various Models

We conducted a statistical analysis of the second round of "CWE defect prediction" on a small batch of annotations, as CWE-type is the only quantifiable label. We were particularly interested in the number of CWE-types generated by each model, and this statistical analysis provides an additional perspective on the conclusions drawn by the voting model. Using regular expression matching, we



Prompt for Expert Model Election Voting

System Prompt

Role: Software Development Teacher
Profile Description: Analyze the homework submissions of students. Each student's submission consists of two rounds of answers: a function description and a CWE defect type description. Identify the most inconsistent answers among the submissions.
Skills
Semantic difference recognitionCode analysisSoftware defect detectionTeaching guidance
......

Round 1
User Prompt

Now evaluate the first round of students' descriptions regarding the function's functionality. Please read the homework answers provided by the four students and select the one that is the most inconsistent, if any. In your response, you need to briefly analyze the answers of the four students. Your answer should be chosen from A, B, C, D, or None, where None indicates that the answers from all four students are basically consistent**......**

Round 1
Model Output

First, let's analyze each student's answer,From these descriptions, it can be seen that the descriptions of A, B, D, and E all focus on certain characteristics of consecutive identical elements in an array. While the details differ, the overall direction is consistent.However, C not only focuses on the function of the algorithm but also provides a detailed analysis of the safety issues in the code, making it clearly different from the direction of the other students' descriptions.Therefore, my answer is: C!

Round 2
User Prompt

Now conducting the second round of student CWE type description reviews. Please analyze the following remaining students' answers in conjunction with the information from the first round. Briefly analyze and identify at least one of the least consistent assignment answers. {info} The answers must be selected from \ {rest_answer}**......**

Round 2
Model Output

Now let's analyze the CWE type descriptions of the remaining students:B : Identified three CWE-20 issues, primarily focused on insufficient input validation, which might lead to invalid input or exceptions. The descriptions are relatively consistent, centering on input validation problems.C : Identified three CWE issues, including CWE-20 (insufficient input validation) and CWE-125 (array access not checked for null)**......**
So my answer is: D!

Figure 7: Prompt Details for Expert Model Election Voting

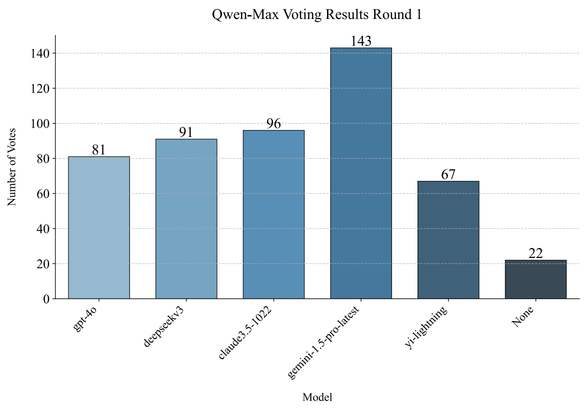


Figure 8: Qwen-max Voting Results Round1

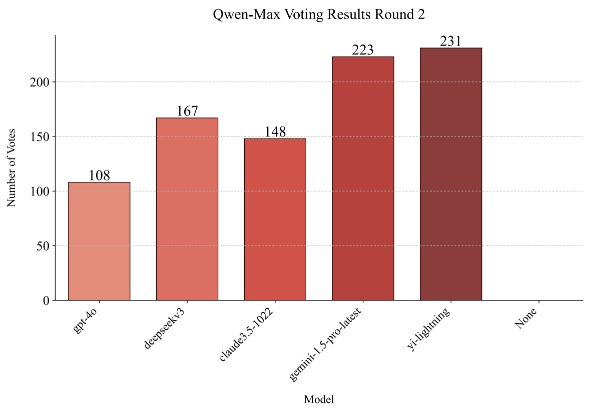


Figure 9: Qwen-max Voting Results Round2

obtained the final statistical results, as shown in Figure 12. The models, from inner to outer in the figure, are: Gemini-1.5-Pro-latest, DeepSeek V3, Yi-lightning, GPT-4o, and Claude-3.5-Sonnet. This figure illustrates the overall proportion of CWE-types labeled by each model, reflecting their preferences. It is evident that the models, from inner to outer, tend to predict a higher number of CWE-

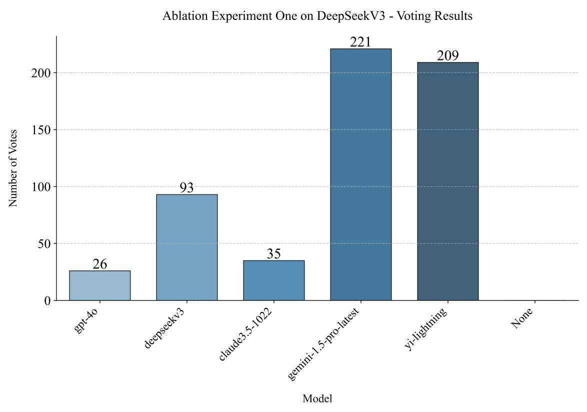


Figure 10: The voting results of the ablation experiment one on DeepSeekV3

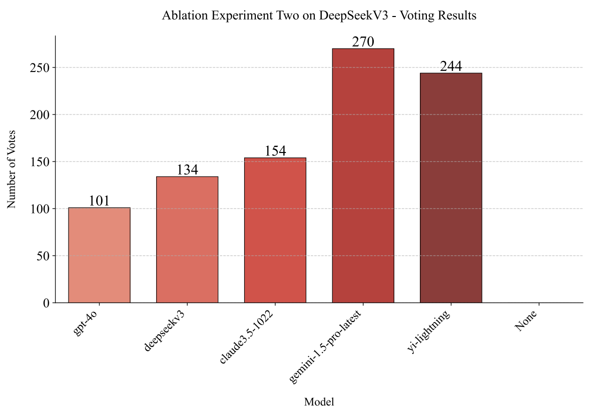


Figure 11: The voting results of the ablation experiment two on DeepSeekV3

types and exhibit a greater focus on identifying code defects.

5. Analysis of Voting Model Eliminations

We also examined the detailed voting patterns of the voting model to better understand the specific behaviors of each model. Since the first round of the voting model required the elimination of at most one model, we similarly focused on the "CWE defect prediction." Given the notable discrepancy in the acceptance of claude's synthesized data by the GPT model compared to other models during the second round of voting, we selected one round of voting from GPT-4o and analyzed the number of models eliminated in each voting round. The results, as depicted in Figure 13, are particularly noteworthy: (1) The highest probability was for the model to vote for the elimination of two models, and based on the voting results from GPT-4o, it is most likely that Yi-lightning and Gemini-1.5-Pro-latest were selected; (2) Interestingly, the probability of eliminating three models was also quite high.

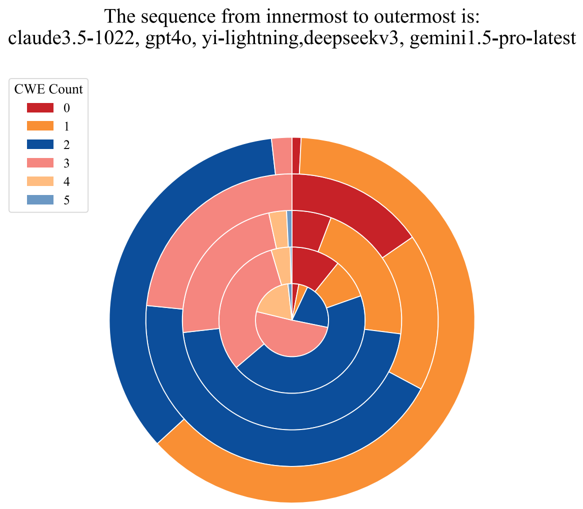


Figure 12: Statistical results of the CWE-Type from the small batch data labeling model

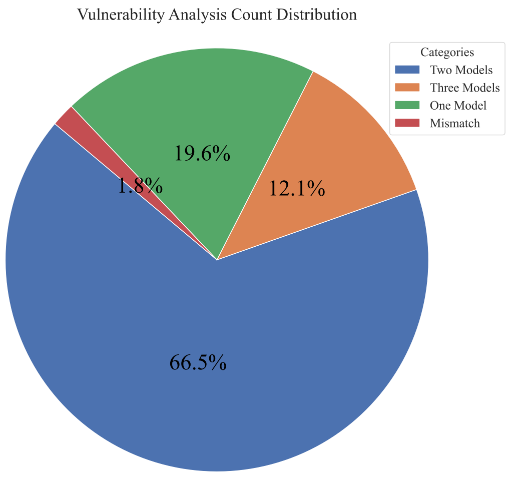


Figure 13: Vulenrability Analysis Count Distribution

We further analyzed the cases where three models were eliminated and found that Claude-3.5-Sonnet was selected in nearly half of these instances. Additionally, during the analysis of GPT-4o, it was observed that Yi-lightning and Claude-3.5-Sonnet produced similar results, but Yi-lightning provided additional insights, suggesting that Yi-lightning should have been favored. However, in the actual results, Yi-lightning was eliminated. These results are noteworthy, and we hypothesize the following reasons: (1) In ablation experiment 4, Claude-3.5-Sonnet's synthesized data demonstrated a preference for a higher number of CWE-types; (2) Claude-3.5-Sonnet had previously utilized GPT-4 data for RLAIF, which may have aligned Claude-3.5-Sonnet more closely with the preferences of GPT models.

F Validation of Model Fine-tuning Efficacy

F.1 Experimental Objective

This appendix details the experiments conducted to validate the effectiveness of the HandPick dataset in enhancing the software defect detection capabilities of large language models (LLMs). We fine-tuned the Qwen2.5-14B-SFT model and evaluated its performance before and after fine-tuning (Qwen2.5-14B-SFT and Qwen2.5-14B-SFT-fine-tuned, respectively) on the MegaVul dataset in a zero-shot setting. Furthermore, the performance of the fine-tuned model was benchmarked against several other advanced LLMs, including DeepSeekV3-671B, GPT-4o, Claude 3.5 Sonnet (claude3.5-1022 version), and Gemini 1.5 Pro (gemini-1.5-pro-latest version).

F.2 Experimental Design

The test dataset used in this experiment was derived from the public MegaVul dataset. Recognizing that the original MegaVul dataset encompasses an overly broad range of defect types, which could reduce the specificity of the evaluation, we filtered and restructured it. Specifically, we first extracted all vulnerable code samples belonging to the CWE Top 10 common defect types, totaling 861 samples. Subsequently, we randomly sampled 139 non-vulnerable code samples. These two subsets were combined to form a final test set of 1000 samples, comprising 861 vulnerable and 139 non-vulnerable samples.

The models evaluated include the Qwen2.5-14B-SFT model before fine-tuning, the Qwen2.5-14B-SFT model fine-tuned with the HandPick dataset, and, as benchmarks, the DeepSeekV3-671B, GPT-4o, Claude 3.5 Sonnet, and Gemini 1.5 Pro models.

All models employed an identical inference pipeline, analyzing the pre-fix version of each code sample in the test set. The inference process followed the three-step Chain-of-Thought (CoT) prompt structure adopted in this study for constructing the HandPick dataset. This structure includes a System Prompt and three User Prompts, guiding the model to: describe the code's functionality, analyze potential security vulnerabilities (with a particular focus on CWE Top 10 defects and requiring a specific identifier {{'CWE_Type':'pass!'}} for non-defective code), and generate code repair suggestions.

F.3 Evaluation Metrics

The performance of the models in the defect detection task was assessed using several key metrics. The four fundamental components of the confusion matrix are defined as follows: TP (True Positive) indicates that a sample is actually vulnerable, the model correctly predicts it as vulnerable (i.e., does not output 'pass!'), and at least one of the predicted CWE IDs matches a true CWE ID of the sample. FP (False Positive) indicates that a sample is actually non-vulnerable, but the model incorrectly predicts it as vulnerable. FN (False Negative) indicates that a sample is actually vulnerable, but the model fails to identify any of its true CWE types, potentially classifying it as non-vulnerable ('pass!' output) or predicting entirely incorrect CWE types. TN (True Negative) indicates that a sample is actually non-vulnerable, and the model correctly identifies it as such.

Based on these components, we calculated standard classification performance metrics, including Accuracy, Precision, Recall, and F1-score. The formulas are as follows:

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

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

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

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

F.4 Experimental Results

The detailed performance metrics of each model on the curated MegaVul test set of 1000 samples are presented in Table 2. The table lists Precision, Recall, F1-score, and Accuracy, along with the raw confusion matrix counts (s_TP, s_FP, s_FN, s_TN) that constitute these metrics.

F.5 Performance Analysis

The experimental results, as presented in Table 2, clearly demonstrate that fine-tuning with the HandPick dataset led to a significant leap in performance for the Qwen2.5-14B-SFT-fine-tuned model compared to its original version, Qwen2.5-14B-SFT.

Table 2: Performance of various models on the MegaVul test set.

Model	Precision	Recall	F1-Score	Accuracy	s_TP	s_FP	s_FN	s_TN
Qwen2.5-14B-SFT	0.5088	0.1672	0.2517	0.1440	144	139	717	0
Gemini 1.5 Pro	0.5900	0.2323	0.3333	0.2000	200	139	661	0
DeepSeekV3-671B	0.6091	0.2494	0.3539	0.2150	215	138	647	0
GPT-4o	0.7181	0.4116	0.5233	0.3544	354	139	507	0
Claude 3.5 Sonnet	0.7832	0.5878	0.6716	0.5055	502	139	353	0
Qwen2.5-14B-SFT (Fine-tuned)	**0.7017**	**0.3798**	**0.4928**	**0.3270**	**327**	**139**	**534**	**0**

Specifically, precision increased by 19.29 percentage points (from 0.5088 to 0.7017), recall substantially improved by 21.26 percentage points (from 0.1672 to 0.3798), the F1-score rose by 24.11 percentage points (from 0.2517 to 0.4928), and accuracy correspondingly increased by 18.30 percentage points (from 0.1440 to 0.3270). These figures robustly affirm the positive impact of the HandPick dataset on enhancing the model's targeted defect detection capabilities, particularly in reducing false negatives.

In comparison with other large language models, the performance of Qwen2.5-14B-SFT is also noteworthy. It comprehensively outperformed DeepSeekV3-671B (with 9.26, 13.04, 13.89, and 11.20 percentage point advantages in precision, recall, F1-score, and accuracy, respectively) and was also significantly superior to Gemini 1.5 Pro in this evaluation (F1-score higher by approximately 15.95 percentage points).

When compared to leading closed-source models, Claude 3.5 Sonnet exhibited the strongest overall performance in this assessment, achieving an F1-score of 0.6716. GPT-4o also performed commendably, with an F1-score of 0.5233, slightly higher than the fine-tuned Qwen2.5-14B-SFT (F1-score of 0.4928). Although these leading models demonstrate superior performance, the magnitude of improvement achieved by Qwen2.5-14B-SFT through fine-tuning with the specialized HandPick dataset, along with its competitive performance on specific metrics (such as its advantages over DeepSeekV3-671B and Gemini 1.5 Pro), amply showcases the substantial potential of high-quality, domain-specific datasets for optimizing and enhancing the efficacy of existing models.

F.6 Experimental Conclusion

The results of this validation experiment strongly attest to the value of the HandPick dataset in empowering large language models for software defect detection. By fine-tuning on this dataset, the Qwen2.5-14B-SFT model achieved significant advancements across all key performance indicators, especially in improving recall and F1-score, which are directly related to the effective discovery of actual vulnerabilities and the amelioration of false negatives.

A noteworthy common phenomenon observed in this testing was that all evaluated models, including Qwen2.5-14B-SFT and other LLMs, failed to correctly identify any of the 139 known non-vulnerable samples (i.e., s_TN was 0 for all models; s_FP was 139 for most, and 138 for DeepSeekV3-671B, implying that all or nearly all non-vulnerable samples were misclassified as vulnerable). This indicates that, under the current experimental setup and evaluation criteria, these models face challenges in distinguishing between genuinely defective code and harmless code. Alternatively, when prompted to search for specific CWE defects, they tend to report suspicious code as having some defect rather than confidently judging it as "non-defective."

Nevertheless, the performance advantages of the fine-tuned Qwen2.5-14B-SFT model over its original version, DeepSeekV3-671B, and Gemini 1.5 Pro are unequivocal. While leading models such as Claude 3.5 Sonnet and GPT-4o demonstrate superior absolute performance, the outcomes of this research highlight that fine-tuning with meticulously constructed datasets relevant to real-world application scenarios is a key strategy for enhancing software defect prediction efficacy. Future research could continue to optimize dataset construction methodologies, explore more robust prompt engineering techniques, and specifically address the challenge of models distinguishing non-defective code, aiming to improve overall precision and accuracy while maintaining high recall.