HANDPICK: A Nice Defect Prediction In Complex Kinds

Anonymous ACL submission

Abstract

Software defect prediction serves as a critical precursor task to software defect detection. In recent years, most research efforts have focused on leveraging static code metrics for this task, yet such approaches face cross-project generalization challenges due to the absence of code semantic features. While emerging studies recognize the importance of code semantics, the lack of high-quality open-source datasets persists due to the prohibitive costs of large-scale manual annotation. With the remarkable capabilities demonstrated by Large Language Models (LLMs) like GPT in data synthesis tasks, we propose leveraging LLMs for automated software defect data synthesis and partially open-sourcing the generated datasets. Our methodology employs Common Weakness Enumeration(CWE) as the defect taxonomy standard, designs structured prompts grounded in software engineering and defect detection principles for data sampling and labeling, and systematically analyzes both model-specific synthesis limitations and dataset quality. The experimental results reveal intriguing insights that provide new perspectives for automated software defect annotation research. (For dataset access inquiries, please contact us via email[1] at your convenience.)

1 Introduction

The later latent defects are detected during the software development lifecycle, the higher the cost of remediation becomes. Post-deployment defect detection and repair costs escalate dramatically (Chan et al., 2023; Weiss et al., 2007). While traditional static analysis tools (e.g., Semgrep) can identify patterned defects, they rely on manual rule maintenance and exhibit high false-positive rates, struggling to adapt to rapidly evolving programming practices. For developers, manually addressing detected software vulnerabilities (Britton et al., 2013) remains non-trivial and time-consuming.

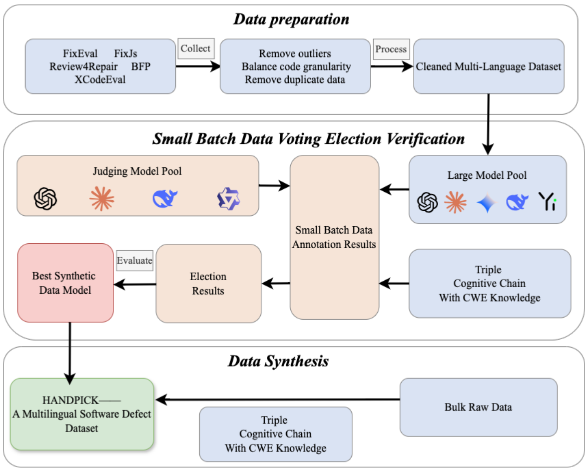

Figure 1: How Does TriCogVuln-LLM Work?

To enhance software quality more effectively, defect prediction techniques should be integrated earlier into the entire software development lifecycle to minimize both the false positive and false negative rates of defects.Existing research has achieved progress in software defect prediction, yet most efforts remain constrained to single programming languages or lack recognized defect taxonomy frameworks, limiting adaptability to multilingual or diverse development environments.

Recently, LLMs have been successfully applied to various code-related tasks (Wang et al., 2024; Zhang et al., 2023a,b). Their exceptional capability in understanding and generating natural language explanations positions them as promising candidates for code review automation. Nevertheless, the application of LLMs to code review – particularly in automated defect annotation – remains underexplored.

This research aims to bridge this gap by proposing an LLM-based automated software defect annotation method. We pioneer the integration of CWE

[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.

to construct "HandPick", a high-quality multilingual defect dataset, designed to enhance prediction accuracy and consistency while advancing software defect prediction research. Key contributions include:

- **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;
- **First LLM-Training-Ready Multilingual Dataset**: We have compiled and released HandPick, the first open-source multilingual dataset specifically designed for LLMs training;
- **Multi-Step Prompt Engineering**: By employing multi-step prompt engineering, we integrate discrete tasks into a cohesive workflow, thereby enhancing the accuracy of LLMs annotations;
- **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 research efforts primarily focused on the Within Project Defect Prediction (WPDP) problem, in practical software development scenarios, the target project requiring defect prediction may be newly initiated or have scarce existing training data, prompting the study of Cross Project Defect Prediction (CPDP).Both WPDP and CPDP predominantly rely on static data from a single programming language (e.g., Java, C/C++)(Wang and Yao, 2013; Nam et al., 2013; Pradel et al., 2020).

Traditional methods depend on static features like lines of code and cyclomatic complexity. Traditional code representation methods leverage pre-trained models (e.g., CodeBERT, GraphCodeBERT) to learn syntax and structural features (Feng et al., 2020; Guo et al., 2020). Wang et al. (Jiang et al., 2021) explored multimodal approaches combining code with text (comments) for automatic program repair. Although promising in merging code and text, these methods still face challenges in aligning code semantics with structured vulnerability patterns.

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.

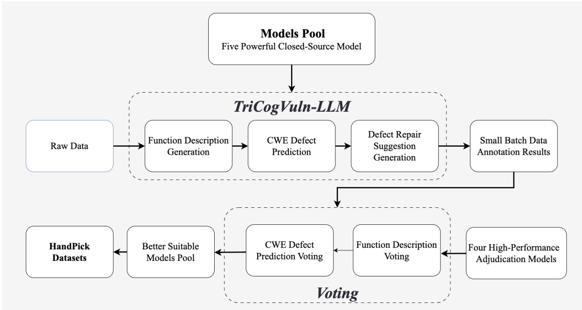

Figure 2: Flowchart of the Main Methods for Dataset Construction

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

Traditional software defect prediction typically encompasses two main tasks: defect prediction and defect repair suggestion. Initially, our approach followed a two-step design: "CWE defect prediction $\rightarrow$ defect repair suggestion generation" to assist LLMs in fulfilling these tasks. However, code defects are not solely determined by the semantics of the code; they are also deeply influenced by functional requirements. Recognizing that functional requirements are often underrepresented or insufficiently addressed, we introduced an additional preliminary step: "code function description." By incorporating this step, we sought to provide richer contextual information for defect prediction and to pave the way for further investigations. As a result, we restructured the original process into three fundamental tasks: "function description generation $\rightarrow$ CWE defect prediction $\rightarrow$ defect repair suggestion generation." This decomposition is intended to improve prediction accuracy and enhance the utility of the data.

3.2.1 Function Description Generation

In the initial phase of our multi-hop reasoning chain, the model is tasked with generating a description of the functionality encapsulated in the provided code. Given that our code operates at the function/class level granularity, we term this phase "Function Description Generation." The user prompt is formulated as: "Please carefully read the following code and describe its functionality in no more than thirty words" + the corresponding code code. The model is expected to infer the original functionality of the function based on the code. Based on our analysis of defect prediction behaviors, we find that the set of defect predictions when the function's functionality is unknown(ϕ_1) and when it is known (ϕ_2) should satisfy the relation $\phi_1 \cap \phi_2 = \phi_3$ and $\phi_3 \neq \emptyset$. Furthermore, in practical scenarios, it is often observed that $\phi_3 \nsubseteq \phi_1$ and $\phi_3 \nsubseteq \phi_2$. Consequently, the code requirements should be provided to the predictor to "delimit a specific scope" for the defect prediction behavior and facilitate subsequent code repair steps. This ensures that the repaired code adheres to the original requirements and aligns with real-world engineering practices.

3.2.2 CWE Defect Prediction

Following the function description generation, we move to the defect prediction phase within the defined scope ϕ_3. In this phase, we meticulously design the system prompt and user prompt, clearly distinguishing their roles. The system prompt is responsible for assigning the LLMs a contextual identity, available techniques, and operational considerations. In contrast, the user prompt provides detailed instructions, introduces structured domain knowledge, and specifies the required output format for the model.

To inform the design of defect prediction techniques and considerations, we conducted interviews with several frontline developers, gathering insights on software testing and code reviews. This practical knowledge was combined with established software engineering principles and prior testing experience, forming the foundation of the system prompt's content. Additionally, we analyzed the CWE-TOP25 lists from 2019 to 2024 and identified the ten consistently highest-ranked CWE types—referred to as the CWE-TOP10—as the focus for this study. The descriptions of these selected CWE types were integrated into the user prompt to enhance prediction accuracy, particularly for the TOP10. Additionally, we specified in the user prompt that the model should output the reasoning behind its predictions, thereby improving the interpretability of the defect prediction process. The output is also structured in JSON format to ensure it is directly usable.

3.2.3 Defect Repair Suggestion Generation

Given the shared techniques between defect prediction and code repair, we tailored the system prompt from "CWE Defect Prediction" by refining the role context, removing unnecessary instructions, and incorporating supplementary development techniques and background details. Additionally, we enhanced the user prompt by specifying a structured JSON format for the output.

3.3 Model Pool

Given the extensive variety of current LLMs, all of which claim to excel in understanding and processing code, we propose using a model pool composed of multiple models to sample synthetic data. This approach aims to obtain a more diverse datasets and avoid introducing "noise" due to overly similar data distributions. After evaluating factors such as model performance, cost, and processing speed, we initially selected five powerful closed-source models. We then conducted a small-scale validation experiment for each model to identify "unusable models," ensuring higher quality data when synthesizing large amounts of data.

3.4 Voting

After obtaining the results of the batch validation, we explored various methods and ultimately chose the simplest LLMs expert voting method, adhering to Occam's Razor principle. This method involves having the models vote on "unusable" models. Since the original datasets lacks ground truth, we made a strong assumption: the model that diverges most significantly from the others is considered unusable. The problem now reduces to "how to select an appropriate voting model." We expect the selected model to balance variance, bias, and cost, and thus, it should be chosen from the initial model pool. To mitigate the issue where LLMs tend to favor data with probability distributions similar to their own, we introduced an external expert to assist in the evaluation. Among the high-performance closed-source models not selected, we included Qwen-max as an auxiliary external expert for selecting the judging model, and conducted further ablation experiments based on the results.

3.5 Evaluation

After applying the voting model, we derived various conclusions about the models, focusing primarily on "which models are unusable for which tasks" and "which model performs relatively well under the current task settings." Since the final data is synthesized by sampling from the model pool, and in the absence of real labels for defect prediction data, we used the output of the best-performing model as "pseudo-labels," combined with "fix_code" from the original datasets, to evaluate datasets quality. Given that the three tasks we designed have distinct characteristics, we established a set of evaluation criteria:

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

In Equation 1, $sample_i$ denotes the outcome of the i-th round in the model pool sampling process, while $pseudo_i$ refers to the outcome of the i-th round in the pseudo-labeling model.

Specifically, due to the differing characteristics of the task outputs, we designed the scoring function in three parts: he first task, "function description," produces short outputs, so we used an embedding model to convert the model pool's sampled output and "pseudo-labels" into embeddings, and then computed their cosine similarity. The second task, "CWE defect prediction," generates predictions that can be mapped to specific CWE types, processed into an $[m, n]$ matrix where m represents the number of evaluations and n represents the number of CWE types. Since the matrix is a 0-1 sparse matrix, we used Jaccard similarity to calculate the second round's score. The third task, "defect repair," involves "fix_code" from the datasets. We tokenized both fix_code and the synthetic data and computed their similarity, defined as:

$$Similarity(sample_3, fix\_code) = \frac{1}{n} \left(\sum repetition(sample_3, fix\_code)\right) + \min\left(1, \frac{extra(sample_3, fix\_code)}{n}\right) \quad (2)$$

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 a high-quality software defect datasets, we meticulously selected five closed-source LLMs that excel in natural language processing and code generation. These models have demonstrated robust performance across multiple benchmarks and are capable of handling complex programming tasks. The selected models include GPT-4o, DeepSeek V3, Claude-3.5-Sonnet, Gemini-1.5-Pro-latest, and Yi-lightning (detailed model settings are provided in Appendix B).

To enhance the quality and reliability of the annotated data, we implemented a voting mechanism to filter the annotation results from models processing 500 small-batch data samples. This approach aimed to select large models that offer a balance between performance and economic cost. In addition to the voting models GPT-4o, DeepSeek V3, and Yi-lightning, we introduced Qwen-max as an external expert to vote on the annotation results of the small-batch data. This inclusion increased the diversity of the voting models and mitigated the impact of single-model bias on the voting outcomes.

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 is divided into two phases: small-batch data annotation and model evaluation, and large-scale data annotation.

In the small-batch annotation phase, we randomly selected 500 data points from the preprocessed datasets and independently annotated them using the five LLMs described in Section 4.1. We designed multi-step CoT prompts for the five models, directing the model to perform a step-by-step analysis of the code: first, identify the code's functionality; then, assess potential vulnerabilities or issues, providing reasoned judgments and explanations; finally, revise the code in accordance with its intended functionality and the identified issues. Consequently, we consistently used multi-step CoT prompts in subsequent annotation work. Each model generated corresponding defect labels and explanations for each code data point based on our carefully designed multi-step CoT prompts. After completing the small-batch annotation, we obtained different annotation results from the five models for the same batch of data, providing a foundation for subsequent model evaluation and selection.

In the large-scale annotation phase, we used the best model selected through evaluation (detailed in Section 4.4) to annotate the remaining 25,000 data points. The annotation process was consistent with the small-batch phase, also employing multi-step CoT prompts to ensure the quality of the large-scale annotated data.

4.3.2 Best Annotation Model Selection

To select a model that strikes the optimal balance between performance and cost for large-scale annotation, we implemented a voting mechanism to systematically eliminate less effective annotation models.

The voting process used LLMs as judges, inputting the data annotated during the small-batch validation phase into the LLMs for voting. In the absence of ground truth, decisions were guided by the "majority is right" strategy, leveraging the LLMs' ability to perform semantic difference identification. Specifically, for each vote, we provided the original datasets(code) along with the data synthesized by the annotation models. The system prompt was configured to simulate the role of a "teacher," tasked with assessing the quality of the "student"(the LLM used during the annotation phase) and identifying the "least suitable" item. Additionally, we had three tasks, with the first "function description" and the second "CWE defect prediction" as the focus.Since the third task ("defect repair suggestion generation") relied heavily on the outputs of the first two tasks, it was given lower priority during the voting process. Consequently, each voting session was restricted to two rounds.Specific user prompts and system prompts can be found in Appendix D.

Although the tasks in the first and second rounds of voting both involved "semantic difference identification," there were subtle differences. The output of the "function description" task was relatively simple, so the synthesized data from different models might be semantically similar. Thus, the voting model was required to select "at most one" unsuitable item in this round. The "CWE defect prediction" task was more challenging, as differences in model capabilities or inherent probability distributions might lead to significant differences in synthesized data. Therefore, the voting model was required to select "at least one" unsuitable item in this round.

We aimed for a good voting model to exhibit low bias and low variance. Since we lacked true labels, the first vote focused on variance, leading to a "model unsuitable" candidate conclusion. For the first attempt, we selected the voting model from our model pool, preferring cost-effective options. Thus, we chose "DeepSeek V3" and "Yi-lightning" for the first vote. Each voting model conducted three votes, and the results are shown in Figure 3.

Based on the analysis of Figure 3a-d, the following preliminary conclusions can be drawn:

The average variance of the three votes in the first round for D V3 is 15.2, compared to 24.5333 for Yi-lightning. In the second round, the variance of DeepSeek V3 is 19.15, significantly lower than 81.5333 for 01, indicating that the variance of DeepSeek V3 is reliable.

The Claude-3.5-Sonnet model performed poorly in the first round, and Gemini-1.5-Pro-latest was deemed "inappropriate" by both voting models. Consequently, Claude-3.5-Sonnet is unsuitable for the "function description" task, and Gemini-1.5-Pro-latest is effectively rejected.

Yi-lightning exhibits a relatively high variance and was frequently self-voted out, leading to its elimination. In the voting between the two models, the GPT-4o model performed better, prompting the hypothesis that GPT-4o could also serve as a voting model. If the voting trends of DeepSeek V3 and GPT-4o are similar, then the bias of DeepSeek V3 may be acceptable. To test this, three votes were conducted using GPT-4o, yielding the following results:

From Figure 3ef, the following conclusions are drawn:

The average variances of GPT-4o in the two rounds are 27.8333 and 64.1833, respectively, with DeepSeek V3's variance consistently lower than that of GPT-4o, further supporting the reliability of DeepSeek V3's variance.

The voting trends of DeepSeek V3 and 4o are similar, suggesting that, under the proposed hypothesis, the bias of DeepSeek V3 as a voting model can be trusted.

It is concluded that Cluade-3.5-Sonnet should not be entirely rejected, as it may be usable in the second round. Additionally, ablation experiments were conducted, detailed in Appendix E.

The comprehensive voting experiments lead to the conclusion that DeepSeek V3 is a viable voting model. Further insights into the composition of the model pool for subsequent large-scale data

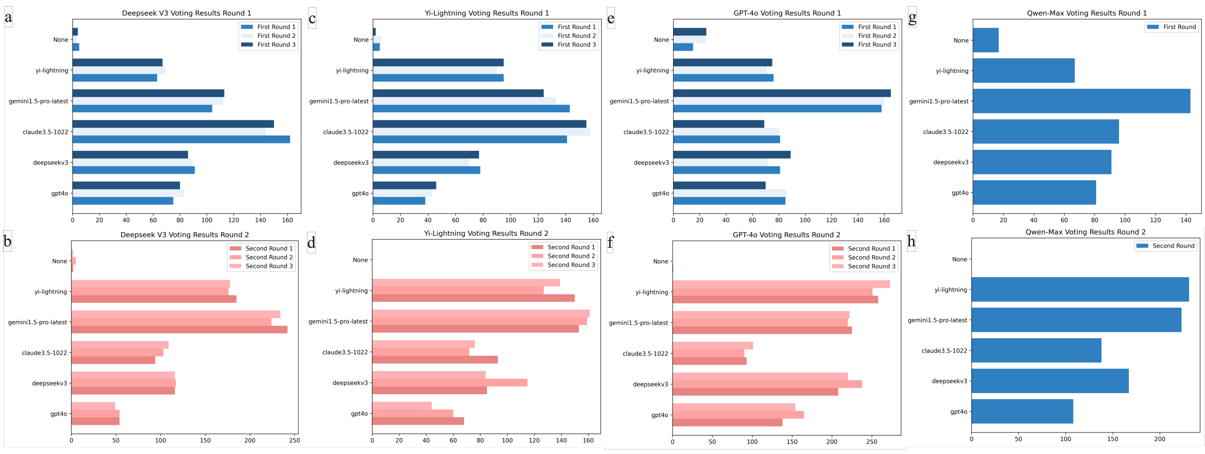

Figure 3: Validation Results of the Voting Model on Small-Scale Datasets

synthesis are as follows:

For the first round of "function description," the model pool consists of (4o, DeepSeek V3), with a sampling ratio of 1:2, reflecting their comparable performance and adherence to the "low cost" principle.

For the second round of "CWE defect prediction," the model pool includes (GPT-4o, DeepSeek V3, Cluade-3.5-Sonnet), with a ratio of 4:3:3 as suggested by the voting model.

For the third round of "defect repair suggestion generation," which builds upon the previous tasks and is considered less challenging, Gemini-1.5-Pro-latest is also included, resulting in a model pool of (GPT-4o, DeepSeek V3, Cluade-3.5-Sonnet, Gemini-1.5-Pro-latest) with a ratio of 1:1:1:1.

4.4 Evaluation of Synthetic Datasets Quality

After a small-scale validation, we finalized the model pool for large-scale data synthesis, annotating a total of 25,000 data entries. We then evaluated the models using the approach outlined in section 3.5. During the evaluation process, GPT-4o consistently outperformed other models. Consequently, in the tasks of "function description" and "CWE defect prediction," we used GPT-4o's outputs as "pseudo-labels" for reference. Additionally, we incorporated the fix_code from the original datasets for a third round of scoring to ensure a comprehensive assessment. From the generated datasets, we selected 500 entries for quality evaluation.

For the "function description" task, we employed the "m3e" embedding model to process the synthetic data from GPT-4o and the data sampled from our model pool. We then calculated the pairwise cosine similarity for the 1000 data entries, resulting in an average similarity score of 0.74.

In the "CWE defect prediction" task, we extracted the CWE type numbers from the outputs of GPT-4o and our model pool. Following the evaluation method in section 3.5, we first counted the occurrences of each CWE defect type. Both GPT-4o and our sampled data contained 34 distinct CWE types, with 18 types being repeated. The most frequently occurring CWE type was "CWE-20 (Improper Input Validation)," indicating that many programs are vulnerable to "injection attacks." This suggests that developers may overly rely on "client-side security checks" or "hidden form fields," which can be bypassed or altered. Additionally, "CWE-787 (Out-of-bounds Write)" and "CWE-125 (Out-of-bounds Read)" were also common. While these defects may not immediately cause exceptions, under certain conditions, they could lead to program crashes, categorizing them as "undefined exceptions." Some instances in the defect prediction results were marked as "pass!", indicating no defects were predicted. We recorded these instances as "0." This process yielded two binary sparse matrices of shape (500, 50). We calculated the Jaccard similarity between these matrices, resulting in a score of 0.54.

For the "defect repair suggestions" task, we compared the repair outputs from LLMs, the fix_code from the original datasets, and the differences with the original code at the "line" level. Following the method in section 3.5, we calculated 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 when compared to the ground truth from the original datasets. Additionally, the number of "pass!" instances in our model pool's results was 75, compared to 92 for GPT-4o, further demonstrating the superiority of our model pool. Given that the code repaired by LLMs for defect prediction is more complete than the original datasets, it is highly likely that $min(1, extra(sample\_3, fix\_code))$ will be 1. The experimental result was 5.851, yielding a score of 1.62 for the "defect repair suggestions" task.

In summary, the overall quality evaluation score for our dataset is approximately 0.725, indicating a highly satisfactory dataset.

We posit that the quality of the dataset surpasses the current evaluation scores, supported by the following rationale: ①The SDK (encompassing the compiler, interpreter, library classes, etc.) utilized by the data is outdated, and subsequent updates to the related SDK versions have rendered the issues obsolete, thereby preventing the LLM from predicting them; ②Certain defects are associated with functional requirements, for which we lack the necessary data; ③The "CWE defect prediction" task yielded a low score, as it is inherently challenging, and our evaluation benchmark, GPT-4o, exhibited high bias during the voting process, suggesting that GPT-4o may not be suitable as a "pseudo-label". Consequently, we contend that our dataset is superior to the current evaluation outcomes. Moreover, our model pool sampling and the proposed framework facilitate a nuanced equilibrium among the cost of data synthesis, data quality, and diversity.

5 Conclusion

In this study, we contributed along three key dimensions: (1) developing the chain-of-thought framework TriCogVuln-LLM, designed specifically for defect prediction tasks, (2) constructing HandPick, the first bilingual, multi-language dataset for defect prediction tasks, comprising approximately 25,000 entries and evaluation methodologies tailored to this framework, and (3) releasing a curated subset of 100 entries as a benchmark for defect prediction involving pre-trained models. Notably, our dataset achieved a high score of 72.5 on our proposed metrics, underscoring its effectiveness for defect prediction across diverse programming languages. By addressing the significant gap in available defect prediction datasets tailored for LLMs, our work offers novel perspectives and resources for advancing defect prediction research. To foster further collaboration, we have made portions of the HandPick dataset, along with the benchmark, publicly available on HuggingFace[2] and GitHub[3].

Moving forward, we will focus on refining the prompt design within our framework to enhance dataset quality and scalability. Additionally, we aim to explore the broader applicability of our methods to a wider array of programming languages and software engineering tasks. We hope that our open-source contributions serve as stepping stones for future research and progress in this critical domain.

6 Limitations

Currently, our dataset covers only four common programming languages. Given that our primary application scenario is centered around Chinese and Java, there is a notable absence of data exploration in other programming languages and English. Although we have divided the defect prediction task into three subtasks, our original aim was to further refine and decompose this task into additional, more granular subtasks. Moreover, while our experimental design and ablation studies are methodologically robust, we must acknowledge the limitation that the datasets lacks genuine, human-authenticated labels, which remains an unresolved issue. Another limitation stems from the underlying assumption in the small-batch validation phase—specifically, that the most dissimilar instances are the least reliable. This assumption may introduce biases and affect the evaluation of the datasets.Additionally, the inherent limitations of large language models (LLMs), including hallucination and restricted capabilities, further compromise datasets quality. Looking ahead, we plan to refine our framework tasks and associated prompts, expand our dataset by incorporating a wider array of programming languages, and address dataset quality concerns through an evident strategy: training LLM with the existing datasets, deploying the updated LLM in real-world development environments to collect richer and more varied data, and applying an iterative self-training approach to enhance the LLM's performance over time.

References

Hafsa Ashraf, Mamdouh Alenezi, Muhammad Nadeem, and Yasir Javid. 2019. Security assessment frame-

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

work for educational erp systems. *International Journal of Electrical and Computer Engineering*, 9(6):5570.

Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. 2013. Reversible debugging software-quantify the time and cost saved using reversible debuggers. *University Cambridge: Cambridge, UK*.

Aaron Chan, Anant Kharkar, Roshanak Zilouchian Moghaddam, Yevhen Mohylevskyy, Alec Helyar, Eslam Kamal, Mohamed Elkamhawy, and Neel Sundaresan. 2023. Transformer-based vulnerability detection in code at edittime: Zero-shot, few-shot, or fine-tuning? *arXiv preprint arXiv:2306.01754*.

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*.

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*.

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. *arXiv preprint arXiv:2002.08155*.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*.

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.

Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. Cure: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1161–1173. IEEE.

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.

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.

Jaechang Nam, Sinno Jialin Pan, and Sunghun Kim. 2013. Transfer defect learning. In *2013 35th international conference on software engineering (ICSE)*, pages 382–391. IEEE.

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.

Michael Pradel et al. 2020. Dynamic software analysis: Challenges and opportunities. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, FSE.

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*.

Shuo Wang and Xin Yao. 2013. Using class imbalance learning for software defect prediction. *IEEE Transactions on Reliability*, 62(2):434–443.

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.

Cathrin Weiss, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. 2007. How long will it take to fix this bug? In *fourth international workshop on mining software repositories (MSR'07: ICSE Workshops 2007)*, pages 1–1. IEEE.

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 (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.

B Three-Step Chain of Thought Prompt

Function description generation involves guiding the model to infer the intended function based on the structure and content of the code and generate a description of the code's functionality. The prompts utilized for this task are depicted in Figure 4, with the English version provided as follows:

Function Description

System Prompt

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.

User Prompt

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

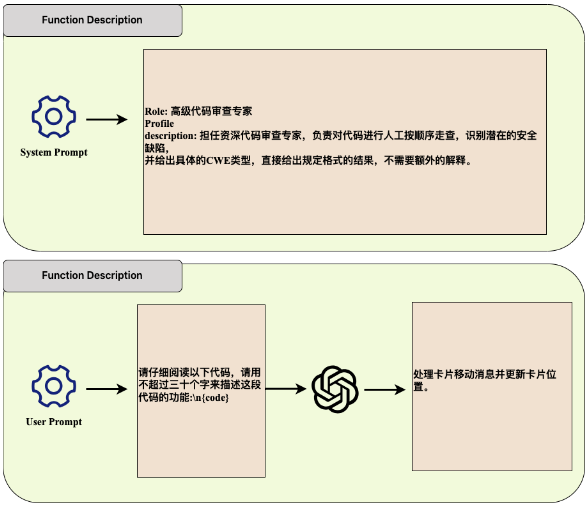

Figure 4: Function Descriptions Prompt Terms and Examples of Model Responses

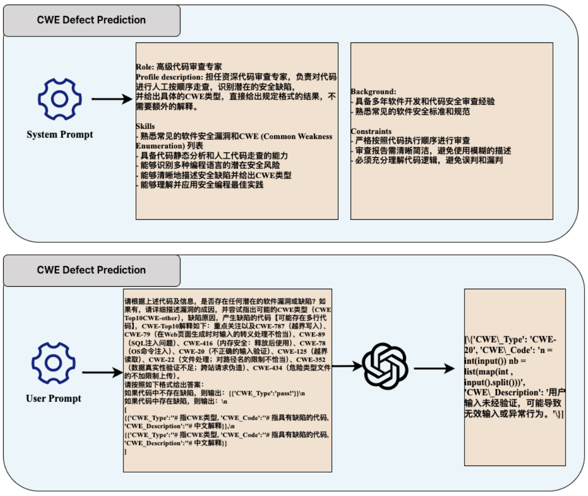

Figure 5: CWE Defect Prediction Prompt Terms and Examples of Model Responses

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

A Example Of Model Output

Process the card movement message and update the card position.

Drawing on both software engineering knowledge and software testing experience, we have developed specific techniques and guidelines for defect prediction. The prompts utilized for this task are depicted in Figure 5, with the English version provided as follows:

CWE Defect Prediction

System Prompt

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.

Capable of conducting static code analysis and manual code reviews.

Able to identify potential security risks in multiple programming languages.

Able to clearly describe security defects and provide the corresponding CWE types.

Able to understand and apply secure coding best practices.

Background:

Several years of experience in software development and code security reviews.

Familiarity with common software security standards and regulations.

Constraints:

Code reviews must strictly follow the order of code execution.

Review reports need to be clear and concise, avoiding vague descriptions.

A thorough understanding of the code logic is required to prevent misjudgment and omissions.

User Prompt

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-79 (Improper Neutralization of Input During Web Page Generation), CWE-89 (SQL Injection), CWE-416 (Use After Free), CWE-78 (OS Command Injection), CWE-20 (Improper Input Validation), CWE-125 (Out-of-bounds Read), CWE-22 (Path Traversal), CWE-352 (Cross-Site Request Forgery), and CWE-434 (Unrestricted File Upload). Your response should follow this format:If no defects are found in the code, output: 'CWE_Type':'pass!'.If defects are present, output:[{{'CWE_Type':' Indicate CWE type', 'CWE_Code':' Indicate the code with defects', 'CWE_Description':' Provide a Chinese explanation'}},

{{'CWE_Type':'# Indicate CWE type', 'CWE_Code':'# Indicate the code with defects', 'CWE_Description':'# Provide a Chinese explanation'}}]

A Example Of Model Output

[{'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.'}]

Building upon the system prompt utilized in "CWE Defect Prediction" we devised a defect repair suggestion generation step, aiming to leverage the large model for repairing defective code. The prompts utilized for this task are depicted in Figure 6, with the English version provided as follows:

Defect Repair Suggestion Generation

System Prompt

Role: Senior Code Repair Expert

Profile Description: You are a senior code repair expert who needs to fix the identified CWE types

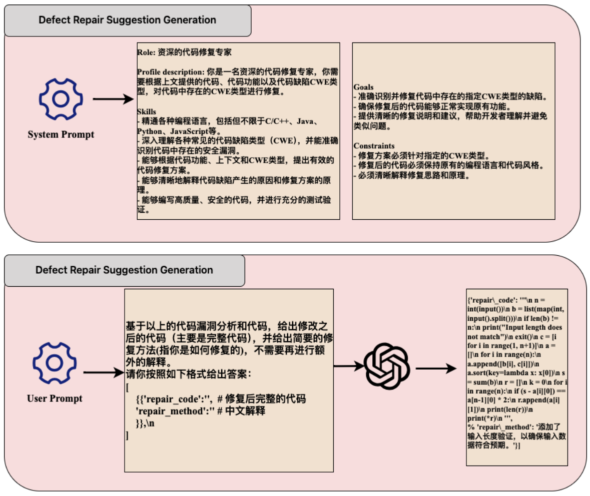

Figure 6: Defect Repair Suggestion Generation Prompt Terms and Examples of Model Responses

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.

Ability to propose effective code repair solutions based on code functionality, context, and CWE types. Capable of clearly explaining the causes of code defects and the principles behind repair solutions. Able to write high-quality, secure code and conduct thorough testing and validation.

Background:

Several years of experience in software development and code security reviews.

Familiarity with common software security standards and regulations.

Constraints:

Code reviews must strictly follow the order of code execution.

Review reports need to be clear and concise, avoiding vague descriptions.

A thorough understanding of the code logic is required to prevent misjudgment and omissions.

User Prompt

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 in Chinese }}]

A Example Of Model Output

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

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.

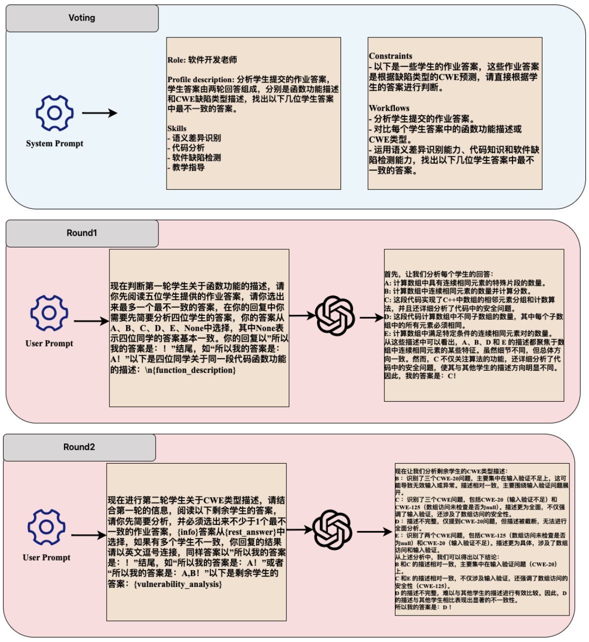

Figure 7: Voting Prompt Terms and Examples of Model Responses

D Prompt for Expert Model Election Voting

We designed a two-round voting process for the expert model. The first round focused on voting for the function description, while the second round targeted the CWE defect prediction. The prompts utilized for this task are depicted in Figure 7, with the English version provided as follows:

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

Constraints

The following are some students' homework answers, which are based on CWE defect type predictions. Please directly judge based on the students' answers.

Workflows

Analyze Student Submissions: Review the function descriptions and CWE defect type descriptions provided by the students.Compare Answers: Look for discrepancies in the function functionality descriptions or CWE types among the students' answers.Identify Inconsistencies: Use semantic difference recognition, code knowledge, and software defect detection abilities to find the most inconsistent answers among the submissions.

User Prompt in Round1

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. Your response should end with 'So my answer is: !' followed by your choice, such as 'So my answer is: A!' Below are the descriptions of the function's functionality from the four students regarding the same piece of code: {function_description}

A Example Of Model Output in Round1

First, let's analyze each student's answer:A: Count the number of special segments with consecutive identical elements in the array. -BCount the number of consecutive identical elements in the array and calculate the score. C This piece of code implements an adjacent element grouping and counting algorithm for an array in C++, and it also provides a detailed analysis of the safety issues in the code. D: This code calculates the number of distinct subarrays in an array, where all elements in each subarray must be identical. E: Count the pairs of consecutive identical elements in the array that satisfy specific conditions. 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!

User Prompt in Round2

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}. If multiple students show inconsistencies, respond with the results separated by English commas. For identical answers, conclude with: "So my answer is: A!" or "So my answer is: A, B!" as appropriate. Below are the remaining students' answers: {vulnerability_analysis}.

A Example Of Model Output in Round2

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). The descriptions are more comprehensive, highlighting not only input validation but also the security of array access.D : Incomplete description, mentioning only the CWE-20 issue, but the description is truncated and cannot be fully analyzed.E : Identified two CWE issues, including CWE-125 (array access not checked for null) and CWE-20 (insufficient input validation). The descriptions are more specific, addressing both array access and input validation. From the above analysis, we can conclude:The descriptions from B and C are relatively consistent, primarily focusing on input validation issues (CWE-20).The descriptions from C and E are relatively consistent, addressing not only input validation but also the security of array access (CWE-125).The description from D is incomplete, making it difficult to effectively compare with those of the other students. Thus, D's description shows significant inconsistency compared to the others.So my answer is: D!

E Ablation Experiments And The Corresponding Analysis Of Results

Given the critical importance of determining the voting model, we conducted extensive ablation ex-

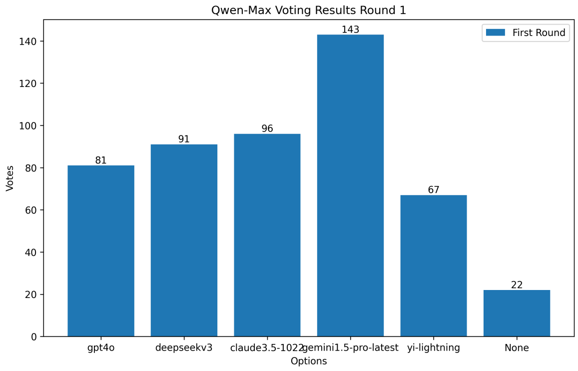

Figure 8: Qwen-max Voting Results Round1

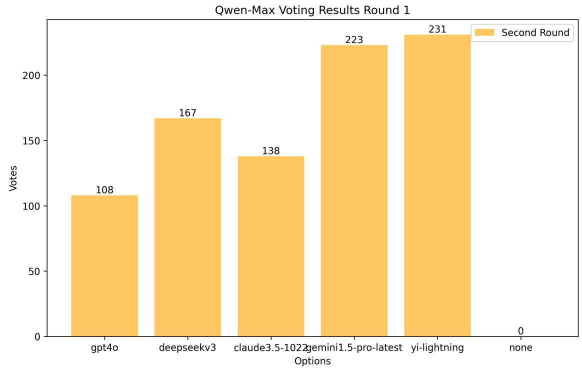

Figure 9: Qwen-max Voting Results Round2

periments 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

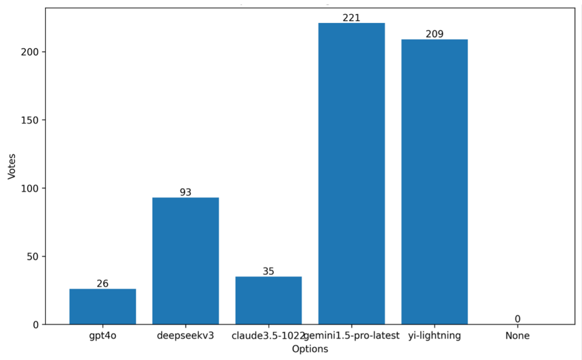

Figure 10: The voting results of the ablation experiment one on DeepSeekV3

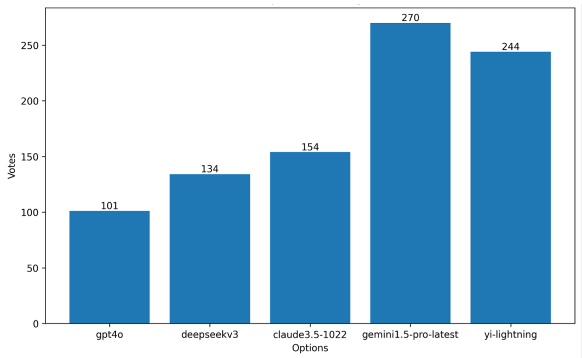

Figure 11: The voting results of the ablation experiment two on DeepSeekV3

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-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.

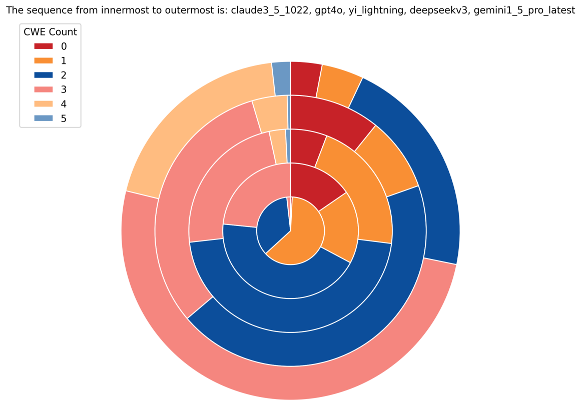

Figure 12: Statistical results of the CWE-Type from the small batch data labeling model

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.

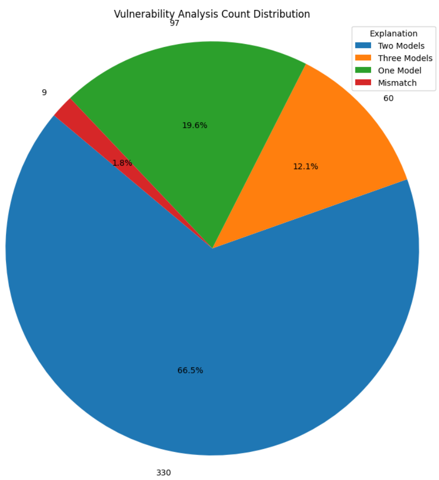

Figure 13: Vulenrability Analysis Count Distribution