LLMs Struggle to Differentiate Vulnerable Code from Patched Code: An Empirical Study and Knowledge-level Enhancement Framework

Anonymous ACL submission

Abstract

Although LLMs have shown promising potential in vulnerability detection, this study reveals their limitations in distinguishing between vulnerable and similar-but-benign patched code (only 0.04 - 0.06 accuracy). It shows that LLMs struggle to capture the root causes of vulnerabilities during vulnerability detection. To address this challenge, we propose enhancing LLMs with multi-dimension vulnerability knowledge distilled from historical vulnerabilities and fixes. We design a novel *knowledge-level* Retrieval-Augmented Generation framework VUL-RAG, which improves LLMs with an accuracy increase of 22% - 25% in identifying vulnerable and patched code. Additionally, VUL-RAG generated vulnerability knowledge can (1) serve as high-quality explanations to improve manual detection accuracy (from 60% to 77%), and (2) detect 10 previously-unknown bugs in the recent Linux kernel release (6 have been confirmed by developers).

1 Introduction

Software vulnerabilities can cause severe consequences. To date, there has been a large body of research on automated vulnerability detection, utilizing traditional program analysis or deep learning techniques. More recently, the advance of large language models (LLMs) further boosts learning-based vulnerability detection. Due to the strong code comprehension capabilities, LLMs show promise in analyzing malicious behaviors (e.g., detecting bugs or vulnerabilities) in code (Zhang et al., 2023; Yang et al., 2024; Shestov et al., 2024; Li et al., 2023a; Sun et al., 2023; Ding et al., 2024a; Widyasari et al., 2024; Zhou et al., 2024).

While significant research has been dedicated to evaluating LLMs for vulnerability detection (Ding et al., 2024b; Gao et al., 2023), their ability to accurately distinguish between vulnerable code and its corresponding patched code remains unclear. Given that vulnerable and patched code pairs often share high textual similarity, addressing this question can reveal whether LLMs genuinely capture the root causes of vulnerabilities or merely overfit to superficial code features when classifying code as vulnerable or benign. Additionally, this question is closely related to the robustness of LLMs in vulnerability detection, which reflects how well LLMs perform in distinguishing between similar code.

Empirical Study. To fill this gap, we perform an empirical study to evaluate the capabilities of LLMs in distinguishing between vulnerable and patched code. We first construct a new benchmark PairVul, which includes 592 high-quality pairs of vulnerable and patched functions extracted from real-world CVEs of complicated software systems. Our experiments reveal that existing LLMs struggle to distinguish between vulnerable and patched code: for majority (94% - 96%) cases, existing LLMs cannot identify the vulnerable code as vulnerable while identify its patched code as benign at the same time. In addition, we further investigate how advanced prompts proposed in recent vulnerability detection work (Wen et al., 2024; Zhou et al., 2024) can eliminate such limitations, including two Chain-of-Thought prompts and one CWE description enhanced prompt. We find that all these advanced strategies bring limited improvement for LLMs, with only 0.01 - 0.16 accuracy in correctly identifying both vulnerable and patched code at the same time. Based on further analysis, we find that LLMs show unstable bias by dominantly identifying most code as vulnerable or benign when working with different prompts. Particularly, LLMs fail to distinguish the subtle textual difference between vulnerable and patched code, such as relocating or replacing method invocations and modifying conditional checks. **In general, LLMs still fall short in understanding vulnerable behaviors in code.**

Enhancement Framework VUL-RAG. To address this challenge, we propose VUL-RAG, a novel knowledge-level Retrieval-Augmented Gen-

eration (RAG) framework to enhance LLM-based vulnerability detection. The key insight behind VUL-RAG is to distill *high-level, generalizable vulnerability knowledge* from historical vulnerabilities and fixes, which can guide LLMs to more accurately understand vulnerable and benign behaviors in code. Specifically, VUL-RAG proposes a novel multi-dimension representation (including perspectives of functional semantics, vulnerability root causes, and fixing solutions) for vulnerability knowledge. The representation focuses on high-level features of vulnerabilities rather than lexical code details. Based on this representation, VUL-RAG incorporates a three-step workflow for vulnerability detection. First, VUL-RAG constructs a vulnerability knowledge base by extracting multi-dimension knowledge from existing CVE instances and fixes via LLMs; Second, for the given code, VUL-RAG retrieves the relevant vulnerability knowledge with similar functional semantics; Finally, VUL-RAG uses LLMs to assess the vulnerability of the given code by reasoning through the presence of vulnerability causes and fixing solutions from the retrieved knowledge.

Evaluation. We evaluate VUL-RAG in extensive settings. (1) *Evaluation on Distinguishing Capabilities.* Our results show that VUL-RAG can substantially enhance the ability of various LLMs to distinguish between vulnerable and patched code (i.e., achieving 22% -25% improvements in pair accuracy). Meanwhile, VUL-RAG achieves an 11%-13%/11% increase in balanced precision/recall for vulnerability detection. Our ablation study shows the superiority of our knowledge-level RAG compared to existing code-level RAG, i.e., 17%-23% increase in pair accuracy. (2) *User Study on Manual Vulnerability Detection.* To evaluate the quality and usability of VUL-RAG generated vulnerability knowledge, we conduct a user study in which participants are asked to confirm vulnerability detection results (both true positives and false alarms) with or without the assistance of VUL-RAG generated vulnerability knowledge. The results show that the vulnerability knowledge improves manual confirmation accuracy from 60% to 77%. User feedback also confirms the high quality of the generated knowledge in terms of the helpfulness, preciseness, and generalizability. (3) *Case Study on Detecting Previously-Unknown Vulnerabilities.* To evaluate whether VUL-RAG generated vulnerability knowledge can detect new vulnerabilities, we apply VUL-RAG to the recent Linux kernel release (v6.9.6, June 2024). VUL-RAG detects 10 previously-unknown bugs, 6 of which have been confirmed by the Linux community. **Our extensive evaluation shows that high-level vulnerability knowledge is a promising direction for enhancing LLM-based vulnerability detection.**

This paper makes the following contributions:

- We perform the first study to reveal the limited capabilities of LLMs in differentiating vulnerable code from patched code.
- We propose VUL-RAG, a novel knowledge-level RAG framework to enhance LLM-based vulnerability detection with generalizable and multi-dimensional vulnerability knowledge distilled from historical vulnerabilities and fixes.
- We perform quantitative experiments, user study, and case analysis to extensively evaluate VUL-RAG. The results not only show the effectiveness of VUL-RAG in improving overall precision/recall and distinguishing capabilities of LLMs, but also show the usability of VUL-RAG in helping manual vulnerability comprehension and detecting previously-unknown bugs for complex software (e.g., Linux Kernel). *Data and code of our work are at (rep, 2024) with MIT license.*

2 Related Work

Empirical Studies. Many efforts have been dedicated to evaluating LLMs in vulnerability detection (Khare et al., 2023; Ding et al., 2024b; Gao et al., 2023), covering diverse benchmarks, LLMs, and metrics. Different from existing studies, we focus on evaluating the capabilities of LLMs in distinguishing between vulnerable and patched code. Risse et al. (Risse and Böhme, 2024) evaluate such capabilities of small pre-trained models (e.g., CodeBERT, UniXcoder, and PLBart), while we study more recent instructed and large LLMs. Ullah et al. (Ullah et al., 2024) evaluate such capabilities of LLMs on a small sample (only 30 pairs) while we extensively study 597 pairs with both quantitative and qualitative analysis.

Enhancing LLMs in Vulnerability Detection. The majority of existing work focuses on prompt engineering (Zhou et al., 2024; Wu et al., 2023), such as chain-of-thought (Wei et al., 2022; Zhang et al., 2022) and few-shot learning (Brown et al.,

2020), to facilitate more powerful LLM-based vulnerability detection. Additionally, recent work explores fine-tuning approaches (Yang et al., 2024; Shestov et al., 2024; Mao et al., 2024) or integration with static analysis (Li et al., 2023a; Sun et al., 2023; Li et al., 2024a; Wen et al., 2024; Li et al., 2024b) to enhance LLMs in vulnerability detection. As fine-tuning enhancement often works for small models with high-quality training data and static analysis enhancement often works on specific types of bugs, in this work, we mainly focus on enhancement techniques with prompt engineering.

Retrieval-Augmented Generation (RAG) for Code-related Tasks. RAG has been widely explored in many code-related tasks, including code generation (Wang et al., 2024), code translation (Bhattarai et al., 2024), program repair (Wang et al., 2023), and vulnerability detection in smart contracts (Yu, 2024). While existing work remains on code-level RAG (retrieving and augmenting with code), VUL-RAG is novel in using high-level, generalizable knowledge to augment generation for the source code vulnerability detection task.

3 Empirical Study

3.1 Experimental Setup

3.1.1 Research Questions of Study

The following RQs aim to evaluate how LLMs distinguish between vulnerable and patched code.

- **RQ1:** How effectively do LLMs distinguish between vulnerable and patched code?
- **RQ2:** How do state-of-the-art prompting strategies improve LLMs in distinguishing between vulnerable and patched code?

3.1.2 Studied LLMs and Baselines

We include four state-of-the-art LLMs that have been widely used in vulnerability detection, including two closed-source models, i.e., GPT-4-turbo (gpt, 2024), Claude Sonnet 3.5 (cla, 2024), and two open-source models, i.e., Qwen2.5-Coder-32B-Instruct (qwe, 2024), DeepSeek-V2-Instruct (dee, 2024).

In RQ1, we evaluate the capabilities of studied LLMs with a basic prompt (Purba et al., 2023). In RQ2, we investigate three state-of-the-art prompting strategies proposed in recent LLM-based vulnerability detection work (Wen et al., 2024; Zhou et al., 2024). These include (1) two prompts that combine role-oriented with chain-of-thought, one involving an initial explanation of code behavior, and the other focusing on the root causes reasoning of vulnerabilities (denoted Cot-1 and Cot-2); and (2) a prompt enhanced with CWE descriptions (denoted CWE-enhanced). The detailed prompts and baseline settings are in Appendix B.

3.1.3 Benchmark

Existing widely-used vulnerability detection benchmarks, such as BigVul (Fan et al., 2020), Devign (Zhou et al., 2019) and Reveal (Chakraborty et al., 2022) are not directly applicable for our study, due to (1) the lack of corresponding patched versions for vulnerable code (e.g., Devign and Reveal), and (2) the absence of verified correctness for patched code. For example, although BigVul includes patched code, its patches may have been subsequently modified in later CVEs, making their correctness unreliable. Therefore, we construct a new benchmark PairVul, which specifically targets high-quality pairs of vulnerable functions and their corresponding patched functions. Our benchmark construction process includes three key steps. (1) *Vulnerable and Patched Code Collection*: We extract function-level pairs of vulnerable and patched code, along with descriptions from existing CVEs of real-world systems (i.e., Linux kernel). Particularly, we focus on Top-5 prevalent CWEs (i.e., 416, 476, 362, 119, 787). (2) *Patched Code Verification*: To ensure the reliability of the patched code, we manually summarize multiple filtering rules to verify the patched code is not subsequently reverted/modified by other commits. (3) *Pair Selection:* to ensure the diversity of the benchmark and control the benchmark scale, we randomly sample one third of pairs of vulnerable and patched functions in each CVE, to further form our final benchmark. We exclude cases where the code length exceeds the current token limit of studied LLMs (i.e., 16,384 tokens). In this way, PairVul includes 592 pairs across 373 CVEs. Detailed construction procedure and benchmark statistics are in Appendix A.

3.1.4 Metrics

We focus on the following metrics. **pairwise accuracy** calculates among all pairs, the ratio of pairs whose vulnerable and patched code are both correctly identified. We use **Balanced Recall** (defined as $\left(\frac{\#\text{True}_{\text{vul}}}{\#\text{Total}_{\text{vul}}} + \frac{\#\text{True}_{\text{nvul}}}{\#\text{Total}_{\text{nvul}}}\right)/2$) and **Balanced Precision** (defined as $\left(\frac{\#\text{True}_{\text{vul}}}{\#\text{Predict}_{\text{vul}}} + \frac{\#\text{True}_{\text{nvul}}}{\#\text{Predict}_{\text{nvul}}}\right)/2$) to evaluate the precision and recall across both vulnerable and non-vulnerable instances. Notably,

Balanced Recall is equivalent to the overall accuracy given the even distribution of vulnerable and non-vulnerable samples on PairVul.

Table 1: Evaluation of Basic LLMs

LLMs	Pair Acc.	Bal. Recall	Bal. Pre.
GPT-4	0.05	0.50	0.50
Claude	0.05	0.49	0.49
Qwen	0.04	0.49	0.49
DeepSeek	0.06	0.48	0.48

3.2 RQ1: Basic Differentiating Capabilities

Table 1 presents the effectiveness of LLMs with the basic prompt. All LLMs show limited capabilities of distinguishing vulnerable and patched code. The low pairwise accuracy (i.e., 0.04 - 0.06) show that LLMs fail to accurately identify a pair of vulnerable and patched code for majority cases (94% - 96%). Additionally, all LLMs show limited balanced recall and precision (not more than 0.50), which is similar as random guess.

3.3 RQ2: Impact of Advanced Prompting

Table 2 shows that the advanced prompts bring limited improvements on the distinguishing capabilities of LLMs. Even for the best case (CoT-2 for Qwen), its pairwise accuracy is only improved to 0.16, while others bring fewer improvements and some (CWE-enhanced) even harm the pair accuracy. Additionally, the balanced precision and accuracy still remain limited (lower than 0.54).

Table 2: Impacts of Enhancement Techniques

Tech.	LLM	Pair Acc.	Bal. Recall	Bal. Pre.
CoT-1	GPT-4	0.14	0.49	0.49
	Claude	0.14	0.51	0.51
	Qwen	0.02	0.51	0.54
	DeepSeek	0.09	0.50	0.50
CoT-2	GPT-4	0.10	0.51	0.52
	Claude	0.14	0.53	0.53
	Qwen	0.16	0.52	0.52
	DeepSeek	0.12	0.51	0.51
CWE Enhanced	GPT-4	0.03	0.50	0.50
	Claude	0.04	0.50	0.51
	Qwen	0.03	0.50	0.49
	DeepSeek	0.01	0.50	0.42

Table 3: Vulnerable Code Identification Ratio

Technique	GPT-4	Claude	Qwen	DeepSeek
Basic LLMs	0.74	0.71	0.23	0.44
CoT-1	0.42	0.37	0.04	0.15
CoT-2	0.71	0.64	0.42	0.63
CWE-Enhanced	0.19	0.12	0.16	0.01

Further Analysis. We perform quantitative and qualitative analysis of RQ1 and RQ2 results.

Unstable Bias. Table 3 shows the ratio of cases that LLMs identify the code as vulnerable when working with different prompts. Interestingly, we find that LLMs show unstable biases although all the prompts are neutral without inductive instructions. With a basic prompt, GPT-4 and Claude tend to consider majority code (over 70%) as vulnerable while Qwen oppositely considers the majority code (77%) as benign. The CoT-1 (explaining the code behaviors first) and CWE-enhanced (including the relevant CWE descriptions) dramatically lead all LLMs to consider most code as benign. The results further confirm that LLMs cannot capture the semantic difference between vulnerable and patched code, thus showing unstable bias when instructed with different neutral prompts.

Case Analysis. We manually sample and analyze code pairs where all the studied LLMs fail to distinguish between vulnerable and patched code. We further confirm that it is challenging for LLMs to discern the subtle textual differences between two similar functions with opposing labels (i.e., vulnerable vs. benign), such as (1) relocating a method invocation, (2) replacing a method invocation, and (3) adding a conditional check. Detailed bad case examples are in Appendix F.1.

Summary of Empirical Studies. Overall, our empirical study reveals that LLMs cannot distinguish between vulnerable and patched code (i.e., pair accuracy lower than 0.06 and balanced recall/precision lower than 0.50), while the recent prompting techniques bring limited improvements. LLMs show unstable bias with different neutral prompts, and struggle to capture subtle textual differences between similar vulnerable code and patched code.

4 Enhancement Framework VUL-RAG

The findings suggest that LLMs require semantic-level guidance for vulnerability detection to avoid relying on superficial code features. Inspired by this, we propose leveraging *high-level vulnerability knowledge* to enhance LLMs in vulnerability detection. Particularly, we propose a novel *knowledge-level* Retrieval-Augmented Generation (RAG) framework VUL-RAG for vulnerability detection, which first distills multi-dimension vulnerability knowledge from existing CVEs and then leverages relevant knowledge items to guide LLM in comprehending the vulnerable behaviors of the given code. As illustrated in Figure 1, VUL-RAG includes three phases: offline vulnerability knowledge base construction, online vulnerability knowledge retrieval, and online knowledge-augmented vulnerability detection.

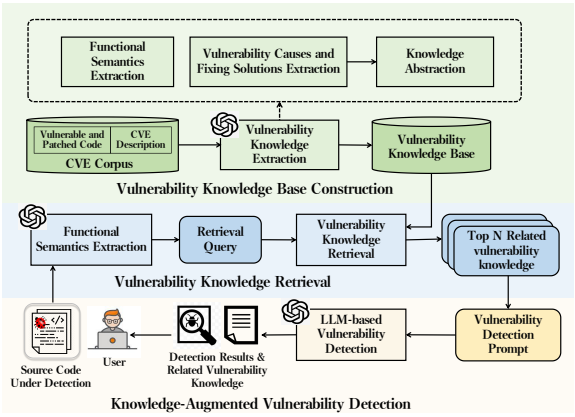


Figure 1: Overview of VUL-RAG

4.1 Vulnerability Knowledge Base Construction

VUL-RAG constructs a vulnerability knowledge base by automatically extracting multi-dimension knowledge via LLMs from existing vulnerabilities and fixes. Section 4.1.1 introduces our novel multi-dimension representation for vulnerability knowledge; Section 4.1.2 introduces the automatic pipeline of knowledge extraction.

4.1.1 Vulnerability Knowledge Representation

Inspired by how developers understand vulnerabilities, we propose a multi-dimensional representation, including seven elements from three dimensions, to describe each vulnerability as follows.

- **Functional Semantics.** This dimension summarizes the high-level functionality (*i.e.,* what this code is doing) of the vulnerable code: (1) *Abstract purpose* is the brief summary of the code intention; and (2) *Detailed behavior* is the detailed description of the code behavior.
- **Vulnerability Causes.** It describes the reasons for triggering vulnerable behaviors by comparing the vulnerable code and its corresponding patch. The cause can be described from three perspectives: (1)*Triggering action* describes the direct action triggering the vulnerability; (2) *Abstract vulnerability description* is the brief summary of the cause; and (3) *Detailed vulnerability description* is more concrete descriptions of the causes.
- **Fixing Solutions.** It summarizes the fixing of the vulnerability by comparing the vulnerable code and its corresponding patch.

Functional semantics are summarized from the vulnerable code, which describe code contexts where vulnerability occurs and are used to facilitate the subsequent retrieval process (Section 4.2); *vulnerability causes* and *fixing solutions* are summarized from the pair of vulnerable and patched code, which are used to facilitate the subsequent online detection process (Section 4.3). Figure 2 exemplifies the multi-dimension representation for the real-world vulnerability CVE-2022-38457.

4.1.2 Knowledge Extraction

For each existing CVE instance (including a pair of vulnerable and patched code and its CVE description), VUL-RAG first leverages LLM to extract each dimension of knowledge; then VUL-RAG performs a knowledge abstraction step to increase the generality of extracted knowledge items.

Functional Semantics Extraction. Given the vulnerable code, VUL-RAG prompts LLMs to extract both its abstract purpose and detailed behavior. The detailed prompt is in Appendix C.1.

Vulnerability Causes and Fixing Solutions Extraction. As the causes and fixing solutions are often logically connected, VUL-RAG extracts two dimensions together to maximize the reasoning capabilities of LLMs. Given a pair of vulnerable and patched code, VUL-RAG incorporates two rounds to extract the vulnerability causes and the corresponding fixing solutions. In the first round, VUL-RAG instructs LLMs to explain the modification from vulnerable code to patched code; in the second round, VUL-RAG further asks LLMs to extract relevant information in dimensions of causes and fixing solutions based on the explanations generated in the first round. Such a two-step strategy follows a CoT paradigm, which inspires LLM reasoning capabilities by thinking step-by-step (Wei et al., 2022; Zhang et al., 2022; Li et al., 2023b; Nong et al., 2024). Additionally, VUL-RAG includes two shots of demonstration examples to guide the output formats of LLMs. The detailed prompts for vulnerability causes and fixing solutions extraction are in Appendix C.1.

Knowledge Abstraction. As different vulnerability instances might share high-level commonality (*e.g.,* the similar causes and fixing solutions), VUL-RAG further performs abstraction to distill more general knowledge representation that is less bonded to concrete code implementation details. Particularly, VUL-RAG leverages LLMs to abstract the concrete code elements (*i.e.,* method invocations, variable names, and types) in the extracted vulnerability causes and fixing solutions. Detailed prompts for knowledge abstraction are in Appendix C.1. We further illustrate two abstraction

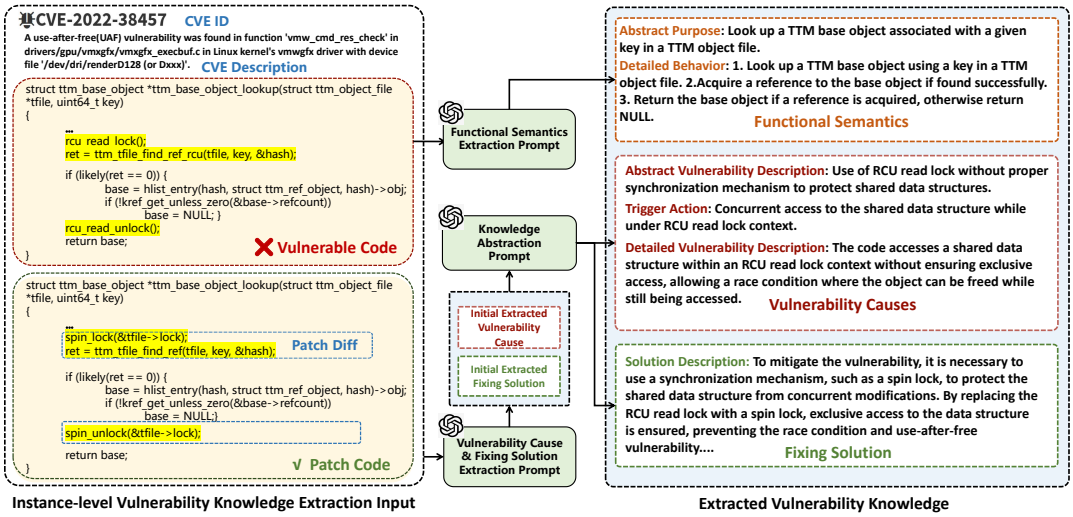


Figure 2: An Example of Vulnerability Knowledge Extraction from CVE-2022-38457

guidelines as follows.

- *Abstracting Method Invocations.* The extracted knowledge might contain concrete method invocations with detailed function identifiers (*e.g.,* `io_worker_handle_work function`) and parameters (*e.g.,* `mutex_lock(&dmxdev->mutex)`), which can be abstracted into the generalized description (*e.g.,* "during handling of IO work processes" and "employing a locking mechanism akin to mutex_lock()").
- *Abstracting Variable Names and Types.* The extracted knowledge might contain concrete variable names or types (*e.g.,* "without &dev->ref initialization"), which can be abstracted into the more general description (*e.g.,* "without proper reference counter initialization").

Vulnerability Knowledge Base. For each vulnerability instance, VUL-RAG generates a multi-dimensional knowledge item with the knowledge extraction and abstraction described above. All the knowledge items are aggregated to form the final *vulnerability knowledge base*. In our experiments, to construct the vulnerability knowledge base, we use the remaining 1,462 pairs of vulnerable and patched code that are not selected into our benchmark PairVul (Section 3.1.3), ensuring that there is no data overlap between the evaluation benchmark and the knowledge base. Detailed statistics within the knowledge base are in Appendix A.

4.2 Vulnerability Knowledge Retrieval

For a given code snippet under detection, VUL-RAG retrieves relevant vulnerability knowledge items from the constructed vulnerability knowledge base in a three-step retrieval process: semantic query generation, candidate knowledge retrieval, and candidate knowledge re-ranking.

Semantic Query Generation. Different from existing RAG pipelines for code-related tasks (Wang et al., 2024) that solely use code as the retrieval query, VUL-RAG uses a mixed query of both code and its functional semantics to find the knowledge item that share high-level functional similarity as the given code. VUL-RAG prompts LLMs to extract the functional semantics of the given code, using the method described in Section 4.1.2. The abstract purpose, detailed behavior, and code itself, form the query for the subsequent retrieval.

Candidate Knowledge Retrieval. VUL-RAG conducts similarity-based retrieval using above three query elements: the code, abstract purpose, and detailed behavior. It retrieves the Top-n knowledge items (where n=10 in our experiments) for each element, resulting in a total of 30 candidate items. Duplicates across query elements are removed to ensure uniqueness. The retrieval is based on the similarity between each query element and the corresponding elements of the knowledge items. VUL-RAG adopts BM25 (Robertson and Walker, 1988) for similarity calculation, a method widely used in search engines due to its efficiency and effectiveness (Sun et al., 2023). Before calculating BM25 similarity, both the query and the retrieval documentation undergo standard preprocessing procedures, including tokenization, lemmatization, and stop word removal (Çagatayli and Çelebi, 2015).

Candidate Knowledge Re-ranking. We re-rank candidate knowledge items with the Reciprocal Rank Fusion (RRF) strategy. For each retrieved

knowledge item k, we aggregate the reciprocal of its rank across all three query elements. If a knowledge item k is not retrieved by a particular query element, we assign its rank as infinity. Detailed formulas and implementation of the retrieval and re-ranking process are in Appendix D. In the end, we keep Top-10 candidate knowledge items with the highest re-rank scores as the final knowledge items for the subsequent vulnerability detection.

4.3 Knowledge-Augmented Vulnerability Detection

Based on the retrieved knowledge items, VUL-RAG leverages LLMs to reason whether the given code is vulnerable. However, directly incorporating all the retrieved knowledge items into one prompt can hinder the effectiveness of the models, as LLMs often perform poorly on long contexts (Liu et al., 2023). Therefore, VUL-RAG iteratively enhances LLMs with each retrieved knowledge item by sequentially checking whether the given code exhibits the same vulnerability cause without the corresponding fixing solutions. If the given code exhibits the same vulnerability cause as the knowledge item but without applying the relevant fixing solution, it is identified as *vulnerable*. Otherwise, VUL-RAG cannot identify the code as vulnerable with the current knowledge item and proceeds to the next iteration (*i.e.,* using the next retrieved knowledge item). If the code cannot be identified as vulnerable with any of the retrieved knowledge items, it is identified as *non-vulnerable*. The iteration process terminates when (1) the code is identified as vulnerable or (2) all the retrieved knowledge items have been considered. The detailed prompts of this phase are in Appendix C.2.

5 Evaluation for VUL-RAG

We answer the following RQs to extensively evaluate the effectiveness and usability of VUL-RAG.

- **RQ3 (Overall Improvements):** How does VUL-RAG improve LLMs in vulnerability detection?
- **RQ4 (User Study on Usability):** How is the quality of VUL-RAG generated knowledge? How can the VUL-RAGgenerated knowledge help manual vulnerability comprehension?
- **RQ5 (Case Study on Detecting New Vulnerabilities):** Can the VUL-RAG generated knowledge help detect previously-unknown vulnerabilities in real-world software systems?

Implementation. During the offline knowledge base construction, we employ GPT-3.5-turbo-0125 (cha, 2023), given its rapid response and cost-effectiveness in generating a large volume of vulnerability-related knowledge items (Sun et al., 2023). For the online knowledge retrieval, we use Elasticsearch (ela, 2023) as our search engine. For the online knowledge-augmented detection, we study the same four LLMs (GPT-4-turbo, Claude Sonnet 3.5, Qwen2.5-Coder-32B-Instruct, and DeepSeek-Coder-V2-Instruct) as in the study.

5.1 RQ3: Overall Improvements

Baselines. Besides the basic prompt and three advanced prompts studied in RQ1 and RQ2, we further include code-level RAG as the baseline. Code-level RAG is a default paradigm that has been widely used in code-related tasks, e.g., program repair (Wang et al., 2023) and code generation (Wang et al., 2024); it uses code similarity (BM25 in our experiments) to retrieve Top-10 similar vulnerable code from historical vulnerabilities (which is the same set for constructing the vulnerability knowledge base of VUL-RAG), and prompts LLMs to detect vulnerabilities with the retrieved pair of vulnerable and patched code into the prompt. The detailed prompt design of code-level RAG is in Figure 4 (b) in Appendix F.2. Comparing VUL-RAG with code-level RAG can investigate the contribution of our knowledge-level representation.

Table 4: Effectiveness of VUL-RAG

Tech.	LLM	Pair Acc.	Bal. Recall	Bal. Pre.
Code RAG	GPT-4	0.05	0.50	0.50
	Claude	0.12	0.51	0.51
	Qwen	0.07	0.51	0.52
	DeepSeek	0.10	0.47	0.47
VUL-RAG	GPT-4	0.28	0.61	0.61
	Claude	0.29	0.60	0.62
	Qwen	0.29	0.60	0.60
	DeepSeek	0.28	0.59	0.59

Results. Table 4 compares VUL-RAG and code-level RAG on PairVul. Due to space limits, here we do not repeat the results of other baselines (in Table 1 and Table 2). Detailed comparison between VUL-RAG and baselines in each CWE category is in Appendix E and the bad case analysis of VUL-RAG is in Appendix H. Overall, VUL-RAG substantially outperforms all baselines in all metrics. Particularly, VUL-RAG not only improves the pair accuracy of LLMs (with 22% - 25% increase) but also improves the balanced precision and recall by

11%-13% and 11%.

Compared to code-level RAG, VUL-RAG shows greater effectiveness in enhancing LLMs for vulnerability detection, with consistent improvements across all metrics. This highlights the contribution of our novel vulnerability knowledge representation and underscores the superiority of knowledge-level RAG over code-level RAG. We manually inspect cases where VUL-RAG successfully identifies vulnerable and patched code pairs that code-level RAG fails. We identify two key reasons for the superior performance of VUL-RAG. (1) In the retrieval phase, knowledge-level RAG more accurately retrieves semantically relevant vulnerabilities from the knowledge base, whereas code-level RAG often retrieves textually similar but semantically irrelevant vulnerabilities. As a result, the vulnerabilities retrieved by code-level RAG offer limited utility for or even mislead LLMs in vulnerability detection. (2) In the inference phase, even when retrieving the same vulnerabilities, the high-level representation of vulnerability knowledge provided by VUL-RAG can more accurately prompt LLMs while the plain representation of code pairs used in code-level RAG cannot. Appendix F.2 presents such two cases observed in our experiments.

5.2 RQ4: Usability for Developers

We conduct a user study to investigate the quality of VUL-RAG generated knowledge and whether the knowledge can help developers understand and check the vulnerabilities in code.

Tasks and Participants. We select 10 cases (5 true and 5 false positives) from PairVul for the user study. We invite 6 participants with 3-5 years c/c++ programming experience. Participants are tasked to identify whether the given code is vulnerable in two settings. (1) Basic setting: provided with the code and the detection labels generated by VUL-RAG; (2) Knowledge-accompanied setting: provided with the basic setting and VUL-RAG generated vulnerability knowledge. Beyond recording user outputs (*i.e.,* vulnerable or not) of each case, we further survey the participants on the helpfulness, preciseness, and generalizability of the vulnerability knowledge on a 4-point Likert scale (Likert, 1932) (*i.e.,* 1-disagree; 2-somewhat disagree; 3-somewhat agree; 4-agree). Detailed procedure and scoring criteria are in Appendix G.

Results. Participants rate the helpfulness, preciseness, and generalizability with average scores of 3.00, 3.20, and 2.97, respectively. It indicates the high quality of vulnerability knowledge generated by VUL-RAG. Additionally, participants provided with VUL-RAG generated vulnerability knowledge can more precisely identify the vulnerable and non-vulnerable code (*i.e.,* 77% detection accuracy with knowledge v.s. 60% detection accuracy without knowledge). It confirms the usability of VUL-RAG generated knowledge for manual vulnerability comprehension.

5.3 RQ5: Detecting New Vulnerabilities

We investigate whether VUL-RAG generated vulnerability knowledge can detect previously-unknown vulnerabilities in real-world software systems. In particular, we apply VUL-RAG on the recent Linux Kernel release (v6.9.6, June 2024) given the importance of Kernel systems. Given the large scale of Linux kernels, we randomly sample a set of files within the *drivers* component, including 1,867 functions in total. We apply VUL-RAG with GPT-4 on the 1,867 functions. VUL-RAG detects 10 previously-unknown bugs, and 6 of them have been confirmed as real bugs by the Linux community. In addition, as VUL-RAG not only generates the detection labels (*i.e.,* vulnerable or not) but also provides vulnerability knowledge with relevant vulnerability causes and fix suggestions, it is helpful for us writing high-quality bug-reporting emails. For the 6 confirmed bugs, we further write patches based on the fix solutions provided by VUL-RAG, and five submitted patches have already been accepted by the paper submission time. Appendix F.3 presents an example of our confirmed bug.

6 Conclusion

This work reveals the limitation of LLMs in distinguishing between vulnerable and patched code; and proposes a novel knowledge-level RAG framework VUL-RAG, which enhances LLMs with multi-dimention vulnerability knowledge. VUL-RAG outperforms all baselines in vulnerability detection; and VUL-RAG generated knowledge improves manual vulnerability detection by 17% accuracy increase. Additionally, VUL-RAG detects 10 previously-unknown bugs in the Linux kernel and 6 of them have been confirmed by the Linux community.

7 Limitations

The incompleteness of the knowledge base can limit the performance of VUL-RAG in practice. Given the diversity of vulnerabilities, it is possible that there is no relevant historical vulnerabilities for the code under detection, which is also a common pain spot for RAG techniques. Therefore, we plan to open source our vulnerability knowledge base, which can be further continuously maintained and extended by the community together. Furthermore, although we evaluated four LLMs, including both open-source and closed-source models, the generalizability of our findings to other LLMs requires further investigation.

References

2023. Elasticsearch.

2023. Gpt-3-5-turbo documentation.

2024. Claude sonnet 3.5.

2024. Deepseek-coder-v2-instruct.

2024. Gpt-4-turbo.

2024. Qwen2.5-coder-32b- 256 instruct.

2024. Replication package.

2024. The website of linux kernel cves.

2024. The website of ommon weakness enumeration.

Manish Bhattarai, Javier E Santos, Shawn Jones, Ayan Biswas, Boian Alexandrov, and Daniel O'Malley. 2024. Enhancing code translation in language models with few-shot learning via retrieval-augmented generation. *arXiv preprint arXiv:2407.19619*.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Mustafa Çagatayli and Erbug Çelebi. 2015. The effect of stemming and stop-word-removal on automatic text classification in turkish language. In *Neural Information Processing - 22nd International Conference, ICONIP 2015, Istanbul, Turkey, November 9-12, 2015, Proceedings, Part I*, volume 9489 of *Lecture Notes in Computer Science*, pages 168–176. Springer.

Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2022. Deep learning based vulnerability detection: Are we there yet? *IEEE Trans. Software Eng.*, 48(9):3280–3296.

Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. 2024a. Vulnerability detection with code language models: How far are we?

Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. 2024b. Vulnerability detection with code language models: How far are we? *arXiv preprint arXiv:2403.18624*.

Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. A C/C++ code vulnerability dataset with code changes and CVE summaries. In *MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020*, pages 508–512. ACM.

Zeyu Gao, Hao Wang, Yuchen Zhou, Wenyu Zhu, and Chao Zhang. 2023. How far have we gone in vulnerability detection using large language models. *CoRR*, abs/2311.12420.

Avishree Khare, Saikat Dutta, Ziyang Li, Alaia Solko-Breslin, Rajeev Alur, and Mayur Naik. 2023. Understanding the effectiveness of large language models in detecting security vulnerabilities. *CoRR*, abs/2311.16169.

Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2023a. The hitchhiker's guide to program analysis: A journey with large language models.

Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2024a. Enhancing static analysis for practical bug detection: An llm-integrated approach. *Proc. ACM Program. Lang.*, 8(OOPSLA1).

Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2023b. Structured chain-of-thought prompting for code generation. *arXiv preprint arXiv:2305.06599*.

Ziyang Li, Saikat Dutta, and Mayur Naik. 2024b. Llm-assisted static analysis for detecting security vulnerabilities.

Rensis Likert. 1932. A technique for the measurement of attitudes. *Archives of psychology*.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2023. Lost in the middle: How language models use long contexts. *CoRR*, abs/2307.03172.

Qiheng Mao, Zhenhao Li, Xing Hu, Kui Liu, Xin Xia, and Jianling Sun. 2024. Towards effectively detecting and explaining vulnerabilities using large language models.

Yu Nong, Mohammed Aldeen, Long Cheng, Hongxin Hu, Feng Chen, and Haipeng Cai. 2024. Chain-of-thought prompting of large language models for discovering and fixing software vulnerabilities. *CoRR*, abs/2402.17230.

Moumita Das Purba, Arpita Ghosh, Benjamin J. Radford, and Bill Chu. 2023. Software vulnerability detection using large language models. In *34th IEEE International Symposium on Software Reliability Engineering, ISSRE 2023 - Workshops, Florence, Italy, October 9-12, 2023*, pages 112–119. IEEE.

Niklas Risse and Marcel Böhme. 2024. Uncovering the limits of machine learning for automatic vulnerability detection. In *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. USENIX Association.

Stephen E. Robertson and Steve Walker. 1988. Some simple effective approximations to the 2-poisson model for probabilistic weighted retrieval. In *Proceedings of the 17th Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval. Dublin, Ireland, 3-6 July 1994 (Special Issue of the SIGIR Forum)*, pages 232–241. ACM/Springer.

Alexey Shestov, Anton Cheshkov, Rodion Levichev, Ravil Mussabayev, Pavel Zadorozhny, Evgeny Maslov, Chibirev Vadim, and Egor Bulychev. 2024. Finetuning large language models for vulnerability detection. *CoRR*, abs/2401.17010.

Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Haijun Wang, Zhengzi Xu, Xiaofei Xie, and Yang Liu. 2023. When gpt meets program analysis: Towards intelligent detection of smart contract logic vulnerabilities in gptscan.

Saad Ullah, Mingji Han, Saurabh Pujar, Hammond Pearce, Ayse K. Coskun, and Gianluca Stringhini. 2024. Llms cannot reliably identify and reason about security vulnerabilities (yet?): A comprehensive evaluation, framework, and benchmarks. In *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*, pages 862–880. IEEE.

Weishi Wang, Yue Wang, Shafiq Joty, and Steven C.H. Hoi. 2023. Rap-gen: Retrieval-augmented patch generation with codet5 for automatic program repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, page 146–158, New York, NY, USA. Association for Computing Machinery.

Zora Zhiruo Wang, Akari Asai, Xinyan Velocity Yu, Frank F. Xu, Yiqing Xie, Graham Neubig, and Daniel Fried. 2024. Coderag-bench: Can retrieval augment code generation?

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.

Cheng Wen, Yuandao Cai, Bin Zhang, Jie Su, Zhiwu Xu, Dugang Liu, Shengchao Qin, Zhong Ming, and Tian Cong. 2024. Automatically inspecting thousands of static bug warnings with large language model: How far are we? *ACM Trans. Knowl. Discov. Data*, 18(7).

Ratnadira Widyasari, David Lo, and Lizi Liao. 2024. Beyond chatgpt: Enhancing software quality assurance tasks with diverse llms and validation techniques.

Fangzhou Wu, Qingzhao Zhang, Ati Priya Bajaj, Tiffany Bao, Ning Zhang, Ruoyu Wang, and Chaowei Xiao. 2023. Exploring the limits of chatgpt in software security applications. *CoRR*, abs/2312.05275.

Aidan Z. H. Yang, Claire Le Goues, Ruben Martins, and Vincent J. Hellendoorn. 2024. Large language models for test-free fault localization. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*, pages 17:1–17:12. ACM.

Jeffy Yu. 2024. Retrieval augmented generation integrated large language models in smart contract vulnerability detection. *ArXiv*, abs/2407.14838.

Chenyuan Zhang, Hao Liu, Jiutian Zeng, Kejing Yang, Yuhong Li, and Hui Li. 2023. Prompt-enhanced software vulnerability detection using chatgpt. *CoRR*, abs/2308.12697.

Zhuosheng Zhang, Aston Zhang, Mu Li, and Alex Smola. 2022. Automatic chain of thought prompting in large language models. *arXiv preprint arXiv:2210.03493*.

Xin Zhou, Ting Zhang, and David Lo. 2024. Large language model for vulnerability detection: Emerging results and future directions. ICSE-NIER'24, page 47–51, New York, NY, USA. Association for Computing Machinery.

Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 10197–10207.

A Dataset

Construction Procedure. In this section, we elaborate more details on the following two key steps involved in the benchmark construction process.

Vulnerable and Patched Code Collection. We first collect all the CVEs from (lin, 2024), an open-source project dedicated to automatically tracking CVEs within the upstream Linux kernel. Based on the list of collected CVE IDs, we further extract corresponding CWE IDs and CVE descriptions from the National Vulnerability Database (NVD), enriching our dataset with detailed vulnerability

categorizations and descriptions. Based on the CVE ID list, we then parse the commit information for each CVE to extract function-level vulnerable and patched code pairs. Vulnerable code snippets prior to the commit diffs are labeled as positive samples and the patched code snippets as negative samples. In this way, we initially obtain a dataset of 4,667 function pairs of vulnerable and patched code across 2,174 CVEs.

Patched Code Verification. The patched code cannot always be non-vulnerable, thus it is important to double-check the correctness of the patched code. To this end, we further implement a filtering process to ensure the patched code is not subsequently reverted or modified by other commits. Specifically, we construct a patch graph in which vulnerable and patched code pairs are represented as independent triplets. Each triplet consists of a head node representing the vulnerable code and a tail node representing its corresponding patched code. If the patched code is modified or reverted by subsequent commits, the triplet evolves into a chain or a loop comprising multiple nodes. For chains in the graph, we retain only the vulnerable and patched code pairs linked by the final edge. For loops, we eliminate all nodes within the loop. This process systematically filters out all patched code snippets that have been altered or reverted, ensuring the correctness of our benchmark dataset.

PairVul includes 592 pairs across 373 CVEs. The statistical data for each CWE category within our benchmark are detailed as Table 5

Table 5: Statistics of PairVul

CWE	CWE-416	CWE-476	CWE-362	CWE-119	CWE-787
CVE Num.	145	60	81	42	45
Pair Num.	267	89	121	53	62

Data Format. Our benchmark PairVul contains the following information for each vulnerability. (i) CVE ID: the unique identifier assigned to a reported vulnerability in the Common Vulnerabilities and Exposures (CVE); (ii) *CVE Description*: descriptions of the vulnerability provided by the CVE system, including the manifestation, the potential impact, and the environment where the vulnerability may occur; (iii) *CWE ID*: the Common Weakness Enumeration identifier that categorizes the type of the vulnerability exploits; (iv) *Vulnerable Code*: the source code snippet containing the vulnerability, which will be modified in the commit; (v) *Patched Code*: the source code snippet that has been committed to fix the vulnerability in the vulnerable code; (vi) *Patch Diff*: a detailed line-level difference between the vulnerable and patched code with added and deleted lines.

Data for constructing knowledge base. We construct the knowledge base using 1462 pairs of vulnerable and patched code across 953 CVEs that do not overlap with PairVul. Table 6 presents the distribution of the five CWE categories within the knowledge base.

Table 6: Statistics of Training Set

	CWE-416	CWE-476	CWE-362	CWE-119	CWE-787
CVE Num.	339	194	169	129	122
Pair Num.	587	262	280	163	170

B Studied Baselines

In RQ1, we evaluate the capabilities of studied LLMs with following basic prompt:

> **Basic Prompt:** Is this code vulnerable? Answer in Yes or No.
> ### Code Snippet: *[Code Snippet]*.

In RQ2, we further investigate three state-of-the-art prompting strategies, including:

Chain-of-thought(CoT) strategies enhances the basic LLMs with two chain-of-thought prompt design, guiding LLMs step-by-step reasoning.

> **CoT-1 Prompt:** I want you to act as a vulnerability detection expert. Initially, you need to explain the behavior of the code. Subsequently, you need to determine whether the code is vulnerable. Answer in YES or NO.
> ### Code Snippet: *[Code Snippet]*.
> **CoT-2 Prompt:** I want you to act as a vulnerability detection system. Initially, you need to explain the behavior of the given code. Subsequently, analyze whether there are potential root causes that could result in vulnerabilities. Based on above analysis, determine whether the code is vulnerable, and conclude your answer with either YES or NO.
> ### Code Snippet: *[Code Snippet]*.

CWE-enhanced strategies enhances the basic LLM by incorporating CWE description information (cwe, 2024) as vulnerability knowledge to LLMs.

> **CWE-enhnced Prompt:** I want you to act as a vulnerability detection system. I will provide you with a code snippet and a CWE description. Please analyze the code to determine if it contains the vulnerability described in the CWE. Answer in YES or NO.
> ### Code Snippet: *[Code Snippet]*.
> ### CWE Description: *[CWE Description]*

C Prompt Design of VUL-RAG

C.1 Prompt Templates in Vulnerability Knowledge Base Construction

Given the vulnerable code snippet, VUL-RAG prompts LLMs with the following instructions to

summarize both the abstract purpose and the detailed behavior respectively, where the placeholder "`[Vulnerable Code]`" denotes the vulnerable code snippet.

Prompt for Abstract Purpose Extraction: `[Vulnerable Code]` What is the purpose of the function in the above code snippet? Please summarize the answer in one sentence with the following format: "Function purpose:".

Prompt for Detailed Behavior Extraction: `[Vulnerable Code]` Please summarize the functions of the above code snippet in the list format without any other explanation: "The functions of the code snippet are: 1. 2. 3..."

The detailed prompts for vulnerability causes and fixing solutions extraction are as follows, where the placeholders "`[Vulnerable Code]`", "`[Patched Code]`", and "`[Patch Diff]`" denote the vulnerable code, the patched code, and the code diff of the given vulnerability, and *[CVE ID]* and *[CVE Description]* denote the details of the given vulnerability.

Extraction Prompt in Round 1: This is a code snippet with a vulnerability *[CVE ID]*: `[Vulnerable Code]` The vulnerability is described as follows:*[CVE Description]* The correct way to fix it is by *[Patch Diff]* The code after modification is as follows: *[Patched Code]* Why is the above modification necessary?

Extraction Prompt in Round 2: I want you to act as a vulnerability detection expert and organize vulnerability knowledge based on the above vulnerability repair information. Please summarize the generalizable specific behavior of the code that leads to the vulnerability and the specific solution to fix it. Format your findings in JSON. Here are some examples to guide you on the level of detail expected in your extraction: *[Vulnerability Causes and Fixing Solution Example 1]* *[Vulnerability Causes and Fixing Solution Example 2]*

The detailed prompts for knowledge abstraction are as follows, which queries LLMs to abstract the method invocations and variable names.

Knowledge Abstraction Prompt: With the detailed vulnerability knowledge extracted from the previous stage, your task is to abstract and generalize this knowledge to enhance its applicability across different scenarios. Please adhere to the following guidelines and examples provided:

[Knowledge Abstraction Guidelines and Examples]

...

C.2 Prompt Templates in Knowledge-Augmented Vulnerability Detection

The prompts used for identifying the existence of vulnerability causes and the fixing solutions of the given code snippets are as follows.

Prompt for Finding Vulnerability Causes: Given the following code and related vulnerability causes, please detect if there is a vulnerability caused in the code. *[Code Snippet]*. In a similar code scenario, the following vulnerabilities have been found: *[Vulnerability causes][fixing solutions]*. Please use your own knowledge of vulnerabilities and the above vulnerability knowledge to detect whether there is a vulnerability in the code.

Prompt for Finding Fixing Solutions: Given the following code and related vulnerability fixing solutions, please detect if there is a vulnerability in the code. *[Code Snippet]*. In a similar code scenario, the following vulnerabilities have been found: *[Vulnerability causes][fixing solutions]*. Please use your own knowledge of vulnerabilities and the above vulnerability knowledge to detect whether there is a corresponding fixing solution in the code.

D Retrieval Implementation

VUL-RAG adopts BM25 (Robertson and Walker, 1988) for similarity calculation in retrieval process. Given a query q and the documentation d for retrieval, BM25 calculates the similarity score between q and d based on the following Equation 1, where $f(w_i, q)$ is the word w_i's term frequency in query q, $IDF\,(w_i)$ is the inverse document frequency of word w_i. The hyperparameters k and b (where k=1.2 and b=0.75) are used to normalize term frequencies and control the influence of document length.

$$Sim_{BM25}(q,d) = \sum_{i=1}^{n} \frac{\mathrm{IDF\,(w_i)} \times \mathrm{f\,(w_i, q)} \times (\mathrm{k}+1)}{\mathrm{f\,(w_i, q)} + \mathrm{k} \times \left(1\text{-b} + \mathrm{b} \times \frac{|\mathrm{q}|}{\mathrm{avgdl}}\right)} \tag{1}$$

We re-rank candidate knowledge items with the Reciprocal Rank Fusion (RRF) strategy. For each retrieved knowledge item k, the re-rank score for k is calculated using the following Equation 2. E denotes the set of all query elements (*i.e.,* the code, the abstract purpose, and the detailed behavior), $rank_t(k)$ denotes the rank of knowledge item k based on query element t.

$$ReRankScore_k = \sum_{t \in E} \frac{1}{rank_t(k)} \tag{2}$$

E Overall Performance

Table 7-Table 11 presents the performance of VUL-RAG and all baselines across the five CWE categories.

Table 7: Effectiveness in CWE-119

Tech.	LLM	Pair Acc.	Bal. Recall	Bal. Pre.
Basic LLM	GPT-4	0.05	0.50	0.50
	Claude	0.08	0.52	0.53
	Qwen	0.06	0.48	0.48
	DeepSeek	0.02	0.44	0.44
CoT-1	GPT-4	0.17	0.56	0.56
	Claude	0.19	0.53	0.53
	Qwen	0.06	0.53	0.59
	DeepSeek	0.13	0.50	0.50
CoT-2	GPT-4	0.11	0.53	0.53
	Claude	0.09	0.48	0.48
	Qwen	0.26	0.60	0.61
	DeepSeek	0.21	0.56	0.56
CWE Enhanced	GPT-4	0.08	0.52	0.52
	Claude	0.08	0.53	0.58
	Qwen	0.06	0.53	0.55
	DeepSeek	0.02	0.51	0.55
Code RAG	GPT-4	0.04	0.50	0.50
	Claude	0.11	0.51	0.51
	Qwen	0.04	0.49	0.47
	DeepSeek	0.02	0.43	0.43
VUL-RAG	GPT-4	0.30	0.62	0.62
	Claude	0.32	0.62	0.65
	Qwen	0.36	0.64	0.64
	DeepSeek	0.23	0.58	0.59

Table 8: Effectiveness in CWE-476

Tech.	LLM	Pair Acc.	Bal. Recall	Bal. Pre.
Basic LLM	GPT-4	0.06	0.51	0.51
	Claude	0.02	0.47	0.47
	Qwen	0.06	0.51	0.52
	DeepSeek	0.09	0.49	0.49
CoT-1	GPT-4	0.12	0.47	0.47
	Claude	0.11	0.48	0.48
	Qwen	0.02	0.51	0.63
	DeepSeek	0.08	0.51	0.51
CoT-2	GPT-4	0.09	0.52	0.52
	Claude	0.25	0.54	0.55
	Qwen	0.15	0.51	0.51
	DeepSeek	0.10	0.51	0.51
CWE Enhanced	GPT-4	0.02	0.49	0.49
	Claude	0.08	0.51	0.51
	Qwen	0.01	0.48	0.46
	DeepSeek	0.01	0.50	0.25
Code RAG	GPT-4	0.08	0.49	0.49
	Claude	0.04	0.47	0.47
	Qwen	0.08	0.48	0.47
	DeepSeek	0.08	0.46	0.46
VUL-RAG	GPT-4	0.28	0.62	0.62
	Claude	0.33	0.61	0.64
	Qwen	0.34	0.61	0.61
	DeepSeek	0.32	0.60	0.61

F Case Study

F.1 Case Study in Empirical Study

We sample and manually analyze pairs that all studied LLMs and advanced techniques in RQ1 and RQ2 fail to distinguish between vulnerable and patched code. Particularly, LLMs fail to distinguish the subtle textual difference between vulnerable code and patched code. Figure 3 illustrates three specific examples, with the patch diffs highlighted in yellow.

F.2 Case Study in Overall Improvements

we use two examples that VUL-RAG can successfully detect the vulnerability but code-level RAG cannot, to explain the superiority of VUL-RAG in both knowledge representation and retrieval strat-

Table 9: Effectiveness in CWE-787

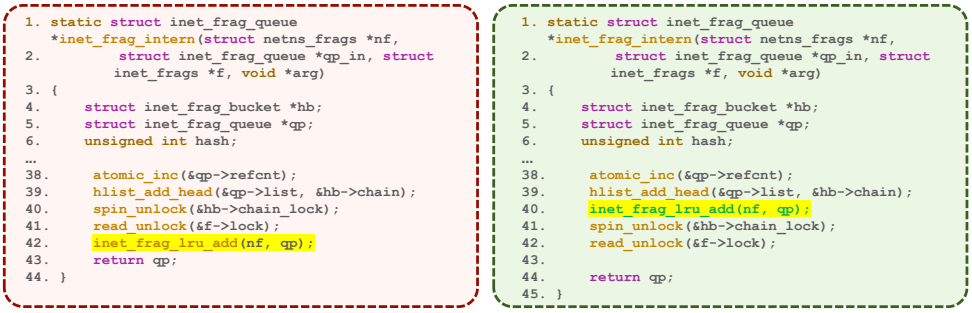
Tech.	LLM	Pair Acc.	Bal. Recall	Bal. Pre.
Basic LLM	GPT-4	0.08	0.52	0.53
	Claude	0.08	0.53	0.54
	Qwen	0.08	0.51	0.51
	DeepSeek	0.11	0.52	0.53
CoT-1	GPT-4	0.18	0.53	0.53
	Claude	0.16	0.54	0.54
	Qwen	0.08	0.53	0.59
	DeepSeek	0.08	0.51	0.51
CoT-2	GPT-4	0.11	0.54	0.55
	Claude	0.19	0.53	0.53
	Qwen	0.19	0.52	0.52
	DeepSeek	0.10	0.52	0.52
CWE Enhanced	GPT-4	0.03	0.50	0.50
	Claude	0.06	0.53	0.68
	Qwen	0.02	0.51	0.75
	DeepSeek	0.01	0.50	0.25
Code RAG	GPT-4	0.11	0.55	0.59
	Claude	0.13	0.54	0.54
	Qwen	0.05	0.52	0.55
	DeepSeek	0.11	0.49	0.49
VUL-RAG	GPT-4	0.25	0.61	0.61
	Claude	0.39	0.65	0.67
	Qwen	0.31	0.60	0.60
	DeepSeek	0.32	0.63	0.63

Table 10: Effectiveness in CWE-362

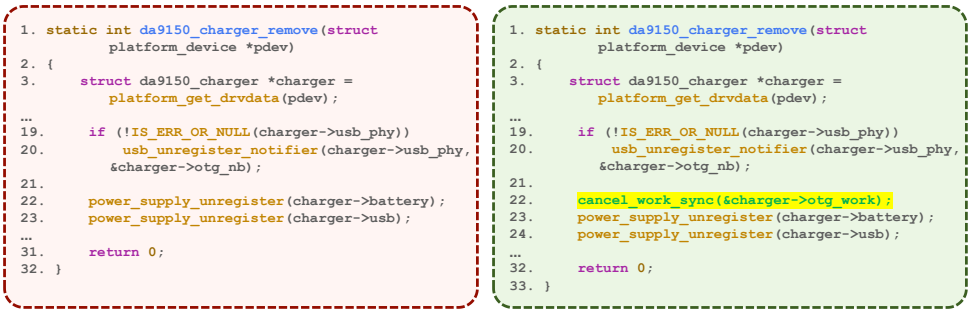
Tech.	LLM	Pair Acc.	Bal. Recall	Bal. Pre.
Basic LLM	GPT-4	0.04	0.50	0.51
	Claude	0.06	0.48	0.48
	Qwen	0.01	0.50	0.50
	DeepSeek	0.06	0.48	0.48
CoT-1	GPT-4	0.14	0.48	0.48
	Claude	0.15	0.52	0.52
	Qwen	0.02	0.51	0.54
	DeepSeek	0.11	0.50	0.51
CoT-2	GPT-4	0.11	0.50	0.50
	Claude	0.16	0.53	0.54
	Qwen	0.14	0.50	0.50
	DeepSeek	0.12	0.52	0.52
CWE Enhanced	GPT-4	0.04	0.47	0.47
	Claude	0.07	0.48	0.48
	Qwen	0.08	0.48	0.48
	DeepSeek	0.01	0.48	0.32
Code RAG	GPT-4	0.03	0.50	0.50
	Claude	0.12	0.49	0.49
	Qwen	0.11	0.54	0.59
	DeepSeek	0.12	0.48	0.48
VUL-RAG	GPT-4	0.26	0.59	0.59
	Claude	0.31	0.61	0.62
	Qwen	0.23	0.58	0.58
	DeepSeek	0.29	0.61	0.62

Table 11: Effectiveness in CWE-416

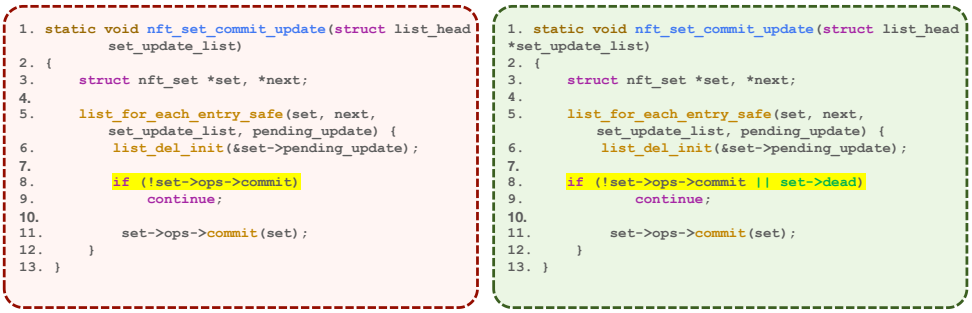
Tech.	LLM	Pair Acc.	Bal. Recall	Bal. Pre.
Basic LLM	GPT-4	0.05	0.49	0.48
	Claude	0.04	0.48	0.48
	Qwen	0.03	0.50	0.49
	DeepSeek	0.04	0.47	0.47
CoT-1	GPT-4	0.13	0.49	0.48
	Claude	0.13	0.51	0.51
	Qwen	0.02	0.50	0.48
	DeepSeek	0.08	0.50	0.50
CoT-2	GPT-4	0.10	0.51	0.51
	Claude	0.16	0.53	0.53
	Qwen	0.15	0.52	0.52
	DeepSeek	0.12	0.50	0.50
CWE Enhanced	GPT-4	0.02	0.50	0.52
	Claude	0.01	0.50	0.75
	Qwen	0.01	0.50	0.42
	DeepSeek	0.01	0.50	0.25
Code RAG	GPT-4	0.03	0.49	0.46
	Claude	0.14	0.52	0.52
	Qwen	0.06	0.50	0.50
	DeepSeek	0.12	0.48	0.48
VUL-RAG	GPT-4	0.29	0.61	0.61
	Claude	0.24	0.58	0.60
	Qwen	0.28	0.60	0.60
	DeepSeek	0.26	0.57	0.57



(a) Example 1: Moving the location of a method invocation



(b) Example 2: Adding a method invocation



(c) Example 3: Adding a conditional check

Figure 3: Examples of vulnerable code and similar-but-benign patched code.

egy.

Knowledge Representation. Figure 4 illustrates an example to show the benefits of our knowledge representation comparing VUL-RAG with basic LLM and code-level RAG baselines, all implemented on GPT-4. When detecting the given code from CVE-2023-30772, the basic GPT-4 fails to identify the real cause of the vulnerability (as shown in Figure 4 (A)). GPT-4 incorrectly suggests that the absence of a return value check in "`platform_get_irq_byname()`" could cause a vulnerability, whereas such a check is not required here. However, it overlooks the true issue, which is the improper handling of asynchronous events resulting in a race condition and subsequently a use-after-free vulnerability. This misunderstanding continues as GPT-4 detects the corresponding patched code, leading to false positives and affecting the pairwise accuracy. Enhancing GPT-4 with code-based RAG also fails to detect the vulnerability. As shown in Figure 4 (B), although the retrieved code pair contains a similar functional semantic and vulnerability cause, GPT-4 still struggles to associate the vulnerability knowledge implied in the retrieved source code with the target code under detection. In contrast, providing the distilled high-level vulnerability knowledge from our approach VUL-RAG, GPT-4 not only successfully detects the vulnerability root cause in the vulnerable code but also accurately identifies the patched code (Figure 4 (C)). The comparison demonstrates the high-level vulnerability knowledge can effec-

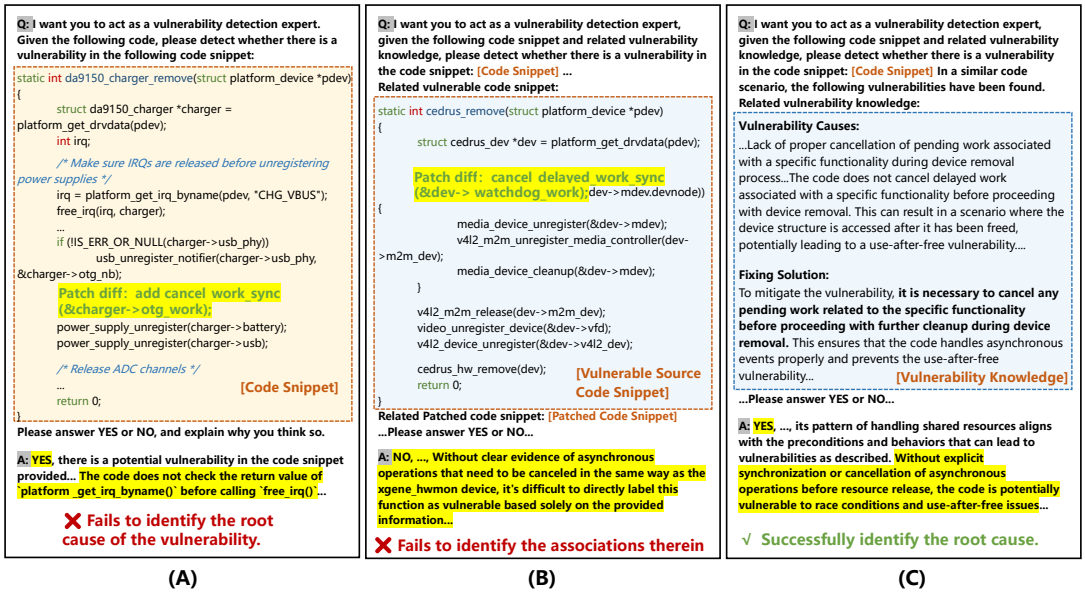


Figure 4: An example of vulnerability knowledge representation

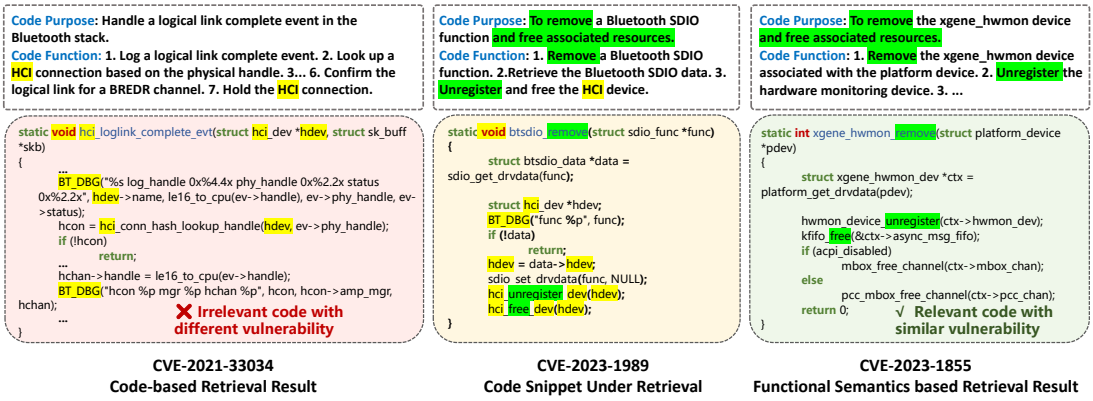


Figure 5: An example of knowledge retrieval strategy

tively help LLMs understand the behavior of the vulnerable code, thereby improving the accuracy of vulnerability detection.

Retrieval Strategy. Figure 5 compares the retrieving outcomes of code-based retrieval (*i.e.,* retrieving only by code snippet) and our retrieval strategy (*i.e.,* retrieving by both code snippet and extracted functional semantics) for the given code snippet. As shown in Figure 5, when detecting a given code snippet from CVE-2023-1989, the code-based retrieval finds a code snippet (from CVE-2021-33034) that shares more operational resources with the target code (highlighted in yellow), but differ significantly in their functional semantics, leading to disparate root causes of vulnerabilities. In contrast, our retrieval strategy finds a code snippet (from CVE-2023-1855) that shares more semantic similarity with the target code (highlighted in green). Furthermore, they share an identical vulnerability root cause, which lies in the failure to adequately handle asynchronous events during the device removal process. This indicates that our retrieval strategy can help LLMs find code pairs with more similar vulnerability causes.

F.3 Case Study of Previously-Unknown Vulnerability detected by VUL-RAG

Figure 6 shows a previously-unknown bug detected by VUL-RAG in Linux kernel v6.9.6. This vulnerability is a use-after-free (UAF) caused by race condition found in the *switchtec_ntb_remove* function located in *drivers/ntb/hw/mscc/ntb_hw_switchtec.c* file. In *switchtec_ntb_add* function, a call to switchtec_ntb_init_sndev binds &sndev->check_link_status_work with check_link_status_work. The *switchtec_ntb_link_notification* function may subsequently trigger the work by calling *switchtec_*

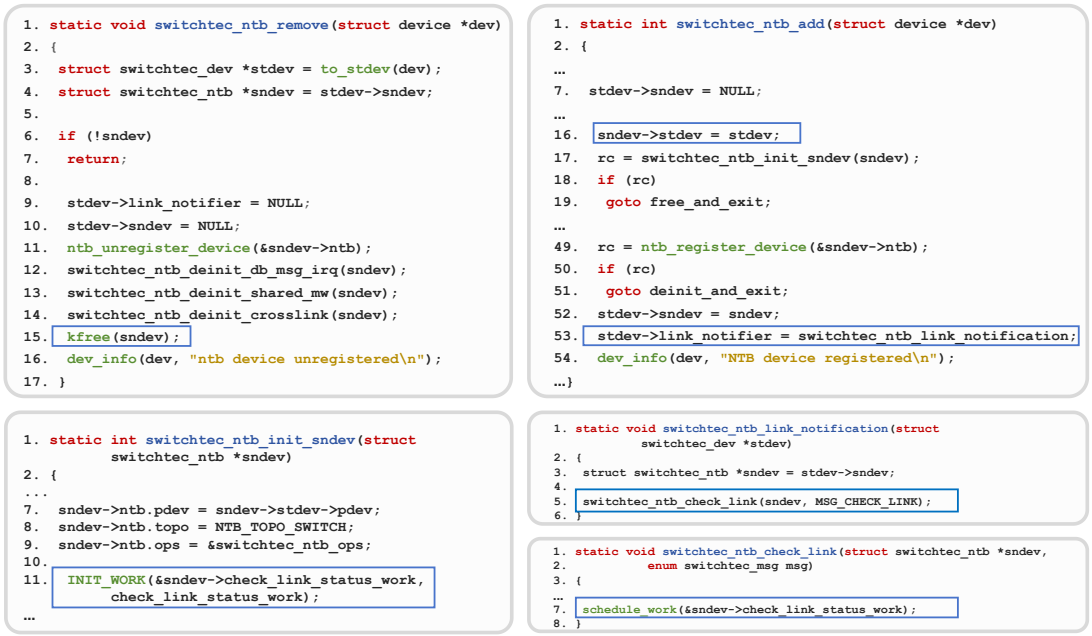


Figure 6: An example of a previously-unknown bug in Linux kernel reported by VUL-RAG

ntb _check_link. When *switchtec_ntb_remove* is called during cleanup, it frees *sndev* via kfree(sndev). If *sndev* is accessed by CPU 1 via *check_link_status_work* after being freed by CPU 0, it could result in a use-after-free (UAF) vulnerability. The vulnerability can be mitigated by ensuring that any pending work is canceled before the cleanup proceeds in *switchtec_ntb_remove*, preventing access to memory that has been freed. Both the root cause and fixing solutions for this vulnerability align with those retrieved from CVE-2023-30772 in our constructed vulnerability knowledge base, demonstrating the scalability and effectiveness of the knowledge captured by VUL-RAG.

G Usability for Developers

This section details the setup for our user study in investigating the quality of VUL-RAG generated knowledge and whether the knowledge can help developers understand and check the vulnerabilities.

Tasks and Participants. We select 10 cases from PairVul for the user study. Specifically, we randomly select two cases from each of the five CWE categories PairVul, including both true positive (*i.e.,* genuinely vulnerable code snippets) and false positive (*i.e.,* correct code snippets mistakenly predicted by VUL-RAG as vulnerable) instances. To ensure a balanced evaluation, we randomly assign the two cases from each CWE category into two equal groups (T_A and T_B), with each group comprising 5 cases. We invite 6 participants with 3-5 years c/c++ programming experience for the user study. We conduct a pre-experiment survey on their c/c++ programming expertise, based on which they are divided into two participant groups (G_A and G_B) of similar expertise distribution. Each participant are payed with 250$ with the experiments. The procedure is approved with Ethics Review Board.

Procedure. Each participant is tasked to identify whether the given code snippet is vulnerable. For comparison, participants are asked to identify vulnerability in two settings. (1) Basic setting: provided with the given code snippets and the detection labels generated by VUL-RAG; (2) Knowledge-accompanied setting: provided with the given code snippets, the detection labels generated by VUL-RAG, and the vulnerability knowledge generated by VUL-RAG. In particular, the participants in G_A are tasked to identify vulnerability in T_A with the knowledge-accompanied setting, and to identify vulnerability in T_B with the basic setting; conversely, the participants in G_B are tasked to identify vulnerability in T_A with the basic setting, and to identify vulnerability in T_B with the knowledge-accompanied setting. In addition to recording the outputs (*i.e.,* vulnerable or not) of each participant, we further survey the participants on the helpfulness, preciseness, and generalizability of the vulnerability knowledge on a 4-point Likert scale (Likert, 1932) (*i.e.,* 1-disagree; 2-somewhat disagree; 3-somewhat agree; 4-agree).

- **Helpfulness**: The vulnerability knowledge pro-

vided by VUL-RAG is helpful in understanding the vulnerability and verifying detection labels.

- **Preciseness**: The vulnerability knowledge offer precise and detailed descriptions of the vulnerability, avoiding overly generic narratives that do not adequately identify the root cause.
- **Generalizability**: The vulnerability knowledge maintains a degree of general applicability, eschewing overly specific descriptions that diminish its broad utility (*e.g.,* narratives overly reliant on variable names from the source code).

H Bad Case Analysis

To understand the limitation of VUL-RAG, we manually analyze the bad cases (*i.e.,* false negatives and false positives reported by VUL-RAG). In particular, we include all 19 FN and 21 FP cases from CWE-119 for manual analysis. Table 12 summarizes the reasons and distributions. In particular, the reasons for false negatives are classified into three primary categories:

Table 12: FN/FP analysis in CWE-119

Type	Reason	Number
FN	Inaccurate vulnerability knowledge descriptions.	5
	Unretrieved relevant vulnerability knowledge.	2
	Non-existent relevant vulnerability knowledge.	12
FP	Mismatched fixing solutions.	11
	Irrelevant vulnerability knowledge retrieval	10

- **Inaccurate Vulnerability Knowledge Descriptions.** We observe that for 5 instances (26.3%), VUL-RAG successfully retrieves relevant vulnerability knowledge but fails to detect the vulnerability due to the imprecise knowledge descriptions. For example, given the vulnerable code snippet of CVE-2021-4204, although VUL-RAG successfully retrieves the relevant knowledge of the same CVE, it yields a false negative due to the vague descriptions of vulnerability knowledge (*i.e.,* only briefly mentioning "*lacks proper bounds checking*" in the vulnerability cause and fixing solution description with explicitly stating what kind of bound checking should be performed).
- **Unretrieved Relevant Vulnerability Knowledge.** We observe that for 2 cases (15.8%) VUL-RAG fails to retrieve relevant vulnerability knowledge, thus leading to false negatives. Although there are instances in the knowledge base that share the similar vulnerability root causes and fixing solutions of the given code, their functional semantics are significantly different. Therefore, VUL-RAG fails to retrieve them from the knowledge base.
- **Non-existent Relevant Vulnerability Knowledge.** Based on our manual checking, the 12 cases (63.2 %) in this category are cased by the absence of relevant vulnerability knowledge in our knowledge base. Even though there are other vulnerable and patched code pairs of the same CVE, the vulnerability behaviors and fixing solutions are dissimilar, rendering these cases unsolvable with the current knowledge base. This limitation is inherent to the RAG-based framework. In future work, we will further extend the knowledge base by extracting more CVE information to mitigate this issue.

In addition, the reasons for false positive can be classified into the following two categories:

- **Mismatched Fixing Solutions.** There are 11 cases (52.4 %) that although VUL-RAG successfully retrieves relevant vulnerability knowledge, the code snippet is still considered vulnerable, as it is considered not applied to the fixing solution of the retrieved knowledge. This is because one vulnerability can be fixed by more than one alternative solution.
- **Irrelevant Vulnerability Knowledge Retrieval.** There are 10 (47.6%) false positives caused by VUL-RAG retrieving irrelevant vulnerability knowledge. Based on our manual inspection, these incorrectly-retrieved knowledge descriptions often generally contain "missing proper validation of specific values", which is too general for GPT4 to precisely identify the vulnerability.