Latest updates: https://dl.acm.org/doi/10.1145/3797277

PDF Download
3797277.pdf
06 April 2026
Total Citations: 7
Total Downloads: 504

Published: 20 February 2026
Accepted: 20 January 2026
Revised: 15 December 2025
Received: 09 October 2025

Citation in BibTeX format

RESEARCH-ARTICLE

Vul-RAG: Enhancing LLM-based Vulnerability Detection via Knowledge-level RAG

XUEYING DU, Fudan University, Shanghai, China

GENG ZHENG, Alibaba Group Holding Limited, Hangzhou, Zhejiang, China

KAIXIN WANG, Fudan University, Shanghai, China

YI ZOU, Fudan University, Shanghai, China

YUJIA WANG, Fudan University, Shanghai, China

WENTAI DENG, Fudan University, Shanghai, China

View all

Open Access Support provided by:

Fudan University

Alibaba Group Holding Limited

Sun Yat-Sen University

ACM Transactions on Software Engineering and Methodology
https://doi.org/10.1145/3797277
EISSN: 1557-7392

Vul-RAG: Enhancing LLM-based Vulnerability Detection via Knowledge-level RAG

XUEYING DU, Department of Computer Science, Fudan University, China
GENG ZHENG, Alibaba Group, China
KAIXIN WANG, Department of Computer Science, Fudan University, China
YI ZOU, Department of Computer Science, Fudan University, China
YUJIA WANG, Department of Computer Science, Fudan University, China
WENTAI DENG, Department of Computer Science, Fudan University, China
JIAYI FENG, Department of Computer Science, Fudan University, China
MINGWEI LIU, School of Software Engineering, Sun Yat-sen University, China
BIHUAN CHEN, Department of Computer Science, Fudan University, China
XIN PENG, Department of Computer Science, Fudan University, China
TAO MA, Alibaba Group, China
YILING LOU*, Department of Computer Science, Fudan University, China

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 0.06 - 0.14 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-dimensional 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 16% - 24% in identifying vulnerable and patched code. Additionally, vulnerability knowledge generated by Vul-RAG can further (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 with 6 assigned CVEs.

CCS Concepts: • **Software and its engineering** → *Maintaining software*; *Application specific development environments*; • **Computing methodologies** → *Artificial intelligence*.

*Corresponding author

Authors' Contact Information: Xueying Du, xueyingdu21@m.fudan.edu.cn, Department of Computer Science, Fudan University, Shanghai, China; Geng Zheng, zhenggeng.zg@alibaba-inc.com, Alibaba Group, Hangzhou, China; Kaixin Wang, kxwang23@m.fudan.edu.cn, Department of Computer Science, Fudan University, Shanghai, China; Yi Zou, yzou21@m.fudan.edu.cn, Department of Computer Science, Fudan University, Shanghai, China; Yujia Wang, yujiawang24@m.fudan.edu.cn, Department of Computer Science, Fudan University, Shanghai, China; Wentai Deng, wtdeng24@m.fudan.edu.cn, Department of Computer Science, Fudan University, Shanghai, China; Jiayi Feng, 23210240148@m.fudan.edu.cn, Department of Computer Science, Fudan University, Shanghai, China; Mingwei Liu, liumw26@mail.sysu.edu.cn, School of Software Engineering, Sun Yat-sen University, Zhuhai, China; Bihuan Chen, bhchen@fudan.edu.cn, Department of Computer Science, Fudan University, Shanghai, China; Xin Peng, pengxin@fudan.edu.cn, Department of Computer Science, Fudan University, Shanghai, China; Tao Ma, boyu.mt@alibaba-inc.com, Alibaba Group, Hangzhou, China; Yiling Lou, yilinglou@fudan.edu.cn, Department of Computer Science, Fudan University, Shanghai, China.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM 1557-7392/2026/2-ART
https://doi.org/10.1145/3797277

Just Accepted

Additional Key Words and Phrases: Large Language Model, Retrieval-Augmented Generation, Vulnerability Detection

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 boosted 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 [27, 32, 47, 49, 56, 58, 61, 64].

While significant research has been dedicated to evaluating LLMs for vulnerability detection [30], 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, Ding et al. [24] conducted the first empirical investigation into the ability of LLMs to differentiate between vulnerable and patched code, revealing LLMs' inherent limitations. However, their study primarily focuses on early, small-scale code LLMs and outdated general LLMs, and relies on relatively simple prompt templates. Building on their findings, we conduct a more comprehensive empirical study to assess the capabilities of state-of-the-art LLMs equipped with advanced prompt strategies in distinguishing vulnerable and patched code. We first construct a new benchmark PairVul, which includes 586 high-quality pairs of vulnerable and patched functions extracted from real-world CVEs of complicated software systems. Our results show that even the latest LLMs still struggle significantly with this task: in the vast majority of cases (86%–94%), current LLMs fail to simultaneously identify the vulnerable version as vulnerable and the patched version as benign. We further investigate how advanced prompts proposed in recent vulnerability detection work [55, 64] 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 to LLMs, with only 0.05 - 0.20 accuracy in correctly identifying both vulnerable and patched code. 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 struggle to distinguish the subtle textual difference between vulnerable and patched code, such as relocated method invocations or modified conditional checks. **Overall, existing state-of-the-art 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 Generation (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

Just Accepted

and patched code (i.e., achieving 16% -24% improvements in pair accuracy). Meanwhile, Vul-RAG achieves a 9%-14%/7%-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-based and fine-tuning-based baselines, i.e., 16%-27% and 22%-26% 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 helpfulness, preciseness, and generalizability. (3) *Case Study on Detecting Previously-Unknown Vulnerabilities.* To evaluate whether Vul-RAG can detect new vulnerabilities, we apply Vul-RAG to the recent Linux kernel release (v6.9.6). Vul-RAG detects 10 previously-unknown bugs with 6 assigned CVEs. **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:

- **Comprehensive Empirical Study:** We perform a comprehensive empirical study to reveal the limited capabilities of state-of-the-art LLMs in differentiating vulnerable code from patched code.
- **Innovative Methodology:** 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.
- **Extension Evaluation:** 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 [7] with MIT license.*

Just Accepted

2 Related Work

Empirical Studies. Many efforts have been dedicated to evaluating LLMs in vulnerability detection [25, 26, 30, 31, 46, 63], 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. Several prior studies explored similar tasks. Risse et al. [42] evaluated this capability using small pre-trained models such as CodeBERT, UniXcoder, and PLBart. Ding et al. [24] assessed code LLMs (e.g., StarCoder2) and outdated closed-source LLMs (e.g., GPT-3.5), while Zibaeirad et al. [67] conducted evaluations using outdated open-source models (e.g., LLaMA3). Building on these efforts, our study provides a more comprehensive evaluation of stronger LLMs with broader coverage of prompt engineering strategies. In addition, Ullah et al. [50] investigated this capability using a very small sample of only 30 vulnerable–patched pairs. In contrast, we conduct an extensive empirical study on 597 pairs, combining in-depth quantitative evaluation with detailed qualitative analysis.

Enhancing LLMs in Vulnerability Detection. The majority of existing work focuses on prompt engineering [57, 64], such as chain-of-thought [54, 62] and few-shot learning [17], to facilitate more powerful LLM-based vulnerability detection. Recent work also explores fine-tuning approaches [38, 47, 58] or integration with static analysis [32, 33, 35, 49, 55] to enhance LLMs in vulnerability detection. In addition, a recent study [52] has leveraged agent frameworks that utilize the planning capabilities of LLMs and adopt a hypothesis-validation paradigm to enhance vulnerability analysis. 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 [53], code translation [16], program repair [51], and vulnerability reasoning and detection [22, 29, 39, 48, 59, 60]. LLM4Vuln [48] and LLM4VFD [59] retrieve bug reports and their LLM-generated summaries as vulnerability knowledge to enhance LLMs' reasoning capabilities. ProveRAG [29] retrieves CVE documentation and summarizes evidence to support vulnerability explanation generation. Llbezpeky [39] combines code context with documentation to construct a vulnerability knowledge base for RAG-enhanced LLM vulnerability reasoning, while VulScribeR [22] leverages historical vulnerable–patched code pairs to perform code-level retrieval augmentation. However, these approaches either rely primarily on code-level RAG (i.e., retrieving and augmenting with historical raw code) or depend on bug reports, existing documentation, and coarse-grained summaries as vulnerability knowledge. In contrast, Vul-RAG introduces a structured, high-level, and generalizable representation of vulnerability knowledge, and automatically distilled knowledge base from historical vulnerable and patched code. This structured and high-level knowledge representation enables more effective retrieval and substantially enhances LLMs' vulnerability reasoning and detection capabilities.

3 Empirical Study

3.1 Experimental Setup

We answer the following RQs to comprehensively evaluate the capabilities of state-of-the-art LLMs equipped with advanced prompt strategies in distinguishing vulnerable and patched code.

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

3.1.1 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-4o [5], Claude Sonnet 3.5 [3], and two open-source models, i.e., Qwen2.5-Coder-32B-Instruct [6], DeepSeek-V2-Instruct [4].

In RQ1, we evaluate the capabilities of studied LLMs with a basic prompt [41]. In RQ2, we investigate three state-of-the-art prompting strategies proposed in recent LLM-based vulnerability detection work [55, 64]. 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 cause reasoning of vulnerabilities (denoted Cot-1 and Cot-2); and (2) a prompt enhanced with CWE descriptions (denoted CWE-enhanced). The detailed prompts are as follows:

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

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 the above analysis, determine whether the code is vulnerable, and conclude your answer with either YES or NO.
Code Snippet: *[Code Snippet]*.

CWE-enhanced 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]*

3.1.2 Benchmark. Existing widely-used vulnerability detection benchmarks, such as BigVul [28], Devign [65] and Reveal [19] are not directly applicable for our study, due to (1) the lack of corresponding patched versions

Just Accepted

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, many prior studies [21, 23, 24, 43] extensively examined the BigVul dataset and consistently reported issues that its patches may have been subsequently modified in later CVEs, making their correctness unreliable (e.g., modification in CVE-2013-3222 was reverted in CVE-2013-7266). Therefore, we construct a new benchmark PairVul, which specifically targets high-quality pairs of vulnerable functions and their corresponding patched functions.

Construction Procedure. Our benchmark construction process includes the following three key steps.

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). To ensure the representativeness of our dataset, we focus on Top-10 prevalent CWEs (i.e., 416, 476, 362, 119, 787, 20, 200, 125, 264, 401). Particularly, we first collect all the CVEs from [14], 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.

Filtering Unrelated Vulnerability Modifications. To reduce the impact of code changes unrelated to vulnerabilities, we applied the following filtering criteria. (1) We retained only modifications to functions in source code files (e.g., `.c`, `.cpp`) and excluded changes to configuration files, test code, and header files. (2) We removed function pairs involving only formatting changes or comments, thereby filtering out non-functional edits. After applying these filters, we obtained a dataset comprising 4,667 vulnerable–patched function pairs spanning 2,174 CVEs. Since most CVE-related commits modify multiple functions, and commits involving changes to only a single function are relatively rare, we did not restrict our dataset to commits containing only one modified function. This design choice ensures sufficient dataset scale, and also consistent with prior dataset construction practices [15, 20, 28]

Patched Code Verification. To ensure the correctness of the patched code, we derived a set of filtering rules to verify that: (1) No vulnerable–patched function pair exhibits identical code before and after the commit; and (2) No patched code has been subsequently reverted or modified by later 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 code remains unchanged before and after the commit, the triplet forms a self-loop. 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 remove all nodes within the loop. This process systematically filters out all incorrect patched code snippets, ensuring the correctness and label consistency of our benchmark dataset. In total, we identified 183 chains containing more than two nodes and removed 203 vulnerable–patched pairs whose patches were modified by subsequent commits. We further detected 8 loops and 699 self-loops; all such cases were excluded from the final dataset.

Data Statistics. We focus on the Top-10 most prevalent CWEs to construct our datasets. The detailed information for each CWE is presented in Table 1. Consequently, we obtain 2903 function pairs of vulnerable-patched pairs across 1325 CVEs. To ensure the diversity of the benchmark and control the benchmark scale, We then randomly divide these data into 1:4 for the test set and training set, with the test set serving as our benchmark dataset PairVul. PairVul includes 586 pairs across 420 CVEs. The statistical data for each CWE category within PairVul are detailed in Table 2. The training set is used to construct our knowledge base, including 2317 pairs of vulnerable and patched code across 1154 CVEs that do not overlap with PairVul. Table 3 presents the distribution of the five CWE categories within the knowledge base.

Just Accepted

Table 1. CWE Definition

CWE	Name	Definition
CWE-416	Use After Free	The product reuses or references memory after it has been freed.
CWE-476	NULL Pointer Dereference	The product dereferences a pointer that it expects to be valid but is NULL.
CWE-362	Race Condition	Concurrent code sequences can simultaneously access a shared resource without proper synchronization.
CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	The program accesses memory outside its allocated buffer boundaries.
CWE-787	Out-of-bounds Write	The product writes data past the end, or before the beginning, of the intended buffer.
CWE-20	Improper Input Validation	Input validation is missing or inadequate for safe data processing.
CWE-200	Exposure of Sensitive Information to an Unauthorized Actor	The product leaks sensitive data to unauthorized parties.
CWE-125	Out-of-bounds Read	The product reads data past the end, or before the beginning, of the intended buffer.
CWE-264	Permissions, Privileges, and Access Controls	Access control issues arise from poor management of permissions, privileges, and security features.
CWE-401	Missing Release of Memory after Effective Lifetime	The product has memory leaks due to poor deallocation practices.

Table 2. Statistics of PairVul

CWE	CWE-416	CWE-476	CWE-362	CWE-119	CWE-787	CWE-20	CWE-200	CWE-125	CWE-264	CWE-401
CVE Num.	117	58	53	36	40	36	31	29	13	23
Pair Num.	166	71	81	44	47	46	39	35	31	26

Table 3. Statistics of Training Set

	CWE-416	CWE-476	CWE-362	CWE-119	CWE-787	CWE-20	CWE-200	CWE-125	CWE-264	CWE-401
CVE Num.	300	163	159	111	107	79	92	89	41	76
Pair Num.	660	281	320	173	187	182	152	140	120	101

Just Accepted

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.

3.1.3 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.

3.2 RQ1: Basic Differentiating Capabilities

Table 4 presents the effectiveness of LLMs with the basic prompt. All LLMs show limited capabilities in distinguishing vulnerable and patched code. The low pairwise accuracy (i.e., 0.06 - 0.14) shows that LLMs fail to accurately identify a pair of vulnerable and patched code for the majority of cases (86% - 94%). Additionally, all LLMs show limited balanced recall and precision (not more than 0.52), which is similar as random guess.

Table 4. Evaluation of Basic LLMs

LLMs	Pair Acc.	Bal. Recall	Bal. Pre.
GPT-4o	0.08	0.50	0.49
Claude	0.06	0.50	0.50
Qwen	0.07	0.52	0.52
DeepSeek	0.14	0.51	0.52

3.3 RQ2: Impact of Advanced Prompting

Table 5 shows that the advanced prompts bring limited improvements on the distinguishing capabilities of LLMs. Even for the best case (CoT-1 for GPT-4o), its pairwise accuracy is only improved to 0.20, 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 0.55).

Table 5. Impacts of Enhancement Techniques

Tech.	LLM	Pair Acc.	Bal. Recall	Bal. Pre.
CoT-1	GPT-4o	**0.20**	0.52	0.52
	Claude	0.13	0.51	0.51
	Qwen	0.09	0.50	0.50
	DeepSeek	0.17	0.53	0.53
CoT-2	GPT-4o	0.18	0.50	0.50
	Claude	0.12	0.51	0.51
	Qwen	0.17	0.52	0.52
	DeepSeek	0.15	0.53	0.54
CWE Enhanced	GPT-4o	0.13	0.51	0.51
	Claude	0.12	0.53	0.53
	Qwen	0.05	0.51	0.51
	DeepSeek	0.18	**0.55**	**0.55**

Table 6. Vulnerable Code Identification Ratio

Technique	GPT-4o	Claude	Qwen	DeepSeek
Basic LLMs	0.77	0.49	0.71	0.81
CoT-1	0.48	0.34	0.17	0.72
CoT-2	0.67	0.66	0.61	0.80
CWE-Enhanced	0.36	0.34	0.20	0.40

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

Unstable Bias. To investigate how prompts influence different LLMs' decision bias, Table 6 shows the ratio of cases in PairVul that LLMs identify the code as vulnerable under different prompts. Interestingly, we find that

Just Accepted

LLMs show unstable biases, although different prompts without inductive instructions. With the basic prompt, GPT-4o and DeepSeek tend to consider the majority code (over 75%) as vulnerable. The CoT-1 (explaining the code behaviors first) and CWE-enhanced (including the relevant CWE descriptions) dramatically lead most LLMs (except DeepSeek) 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. 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 1 illustrates three specific examples, with the patch diffs highlighted in yellow.

Summary of Empirical Studies. Overall, our empirical study reveals that LLMs cannot distinguish between vulnerable and patched code (i.e., pair accuracy lower than 0.14 and balanced recall/precision lower than 0.52), 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 2, Vul-RAG includes three phases: offline vulnerability knowledge base construction, vulnerability knowledge retrieval, and knowledge-augmented vulnerability detection.

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's intention; and (2) *Detailed behavior* is the detailed description of the code's 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 a more concrete description of the cause.

Just Accepted

```
1. static struct inet_frag_queue
   *inet_frag_intern(struct netns_frags *nf,
2.          struct inet_frag_queue *qp_in, struct
         inet_frags *f, void *arg)
3. {
4.     struct inet_frag_bucket *hb;
5.     struct inet_frag_queue *qp;
6.     unsigned int hash;
…
38.     atomic_inc(&qp->refcnt);
39.     hlist_add_head(&qp->list, &hb->chain);
40.     spin_unlock(&hb->chain_lock);
41.     read_unlock(&f->lock);
42.     inet_frag_lru_add(nf, qp);
43.     return qp;
44. }
```

Vulnerable Code

```
1. static struct inet_frag_queue
   *inet_frag_intern(struct netns_frags *nf,
2.          struct inet_frag_queue *qp_in, struct
         inet_frags *f, void *arg)
3. {
4.     struct inet_frag_bucket *hb;
5.     struct inet_frag_queue *qp;
6.     unsigned int hash;
…
38.     atomic_inc(&qp->refcnt);
39.     hlist_add_head(&qp->list, &hb->chain);
40.     inet_frag_lru_add(nf, qp);
41.     spin_unlock(&hb->chain_lock);
42.     read_unlock(&f->lock);
43.
44.     return qp;
45. }
```

Non-vulnerable Code

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

```
1. static int da9150_charger_remove(struct
         platform_device *pdev)
2. {
3.     struct da9150_charger *charger =
         platform_get_drvdata(pdev);
…
19.     if (!IS_ERR_OR_NULL(charger->usb_phy))
20.         usb_unregister_notifier(charger->usb_phy,
        &charger->otg_nb);
21.
22.     power_supply_unregister(charger->battery);
23.     power_supply_unregister(charger->usb);
…
31.     return 0;
32. }
```

Vulnerable Code

```
1. static int da9150_charger_remove(struct
         platform_device *pdev)
2. {
3.     struct da9150_charger *charger =
         platform_get_drvdata(pdev);
…
19.     if (!IS_ERR_OR_NULL(charger->usb_phy))
20.         usb_unregister_notifier(charger->usb_phy,
        &charger->otg_nb);
21.
22.     cancel_work_sync(&charger->otg_work);
23.     power_supply_unregister(charger->battery);
24.     power_supply_unregister(charger->usb);
…
32.     return 0;
33. }
```

Non-vulnerable Code

(b) Example 2: Adding a method invocation

```
1. static void nft_set_commit_update(struct list_head
         set_update_list)
2. {
3.     struct nft_set *set, *next;
4.
5.     list_for_each_entry_safe(set, next,
         set_update_list, pending_update) {
6.         list_del_init(&set->pending_update);
7.
8.         if (!set->ops->commit)
9.             continue;
10.
11.         set->ops->commit(set);
12.     }
13. }
```

Vulnerable Code

```
1. static void nft_set_commit_update(struct list_head
*set_update_list)
2. {
3.     struct nft_set *set, *next;
4.
5.     list_for_each_entry_safe(set, next,
         set_update_list, pending_update) {
6.         list_del_init(&set->pending_update);
7.
8.     if (!set->ops->commit || set->dead)
9.             continue;
10.
11.         set->ops->commit(set);
12.     }
13. }
```

Non-vulnerable Code

(c) Example 3: Adding a conditional check

Fig. 1. Examples of vulnerable code and similar-but-benign patched code.

Just Accepted

- **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

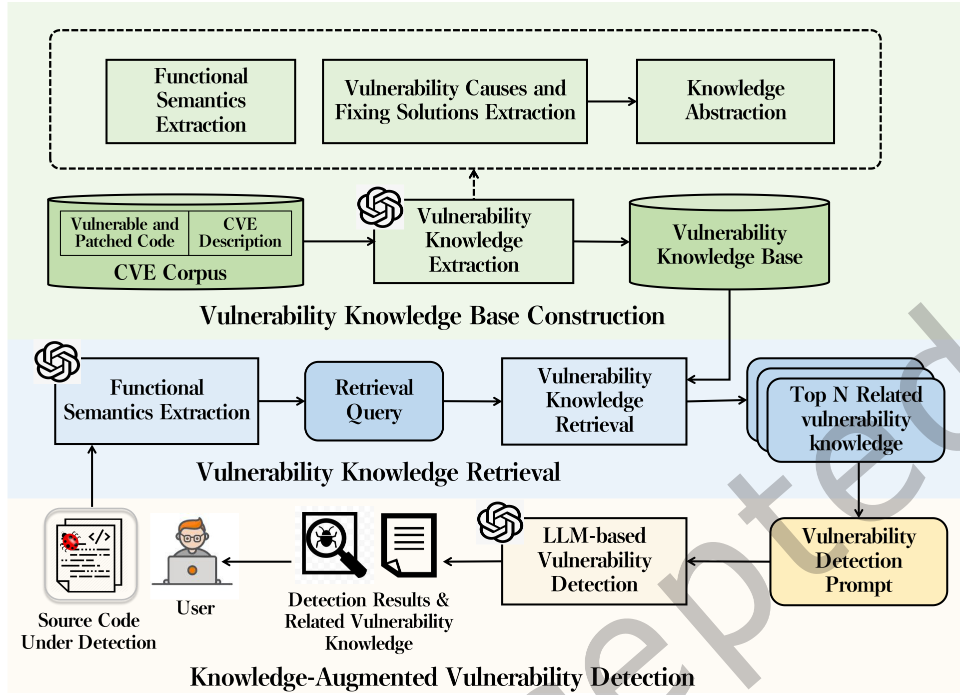


Fig. 2. Overview of Vul-RAG

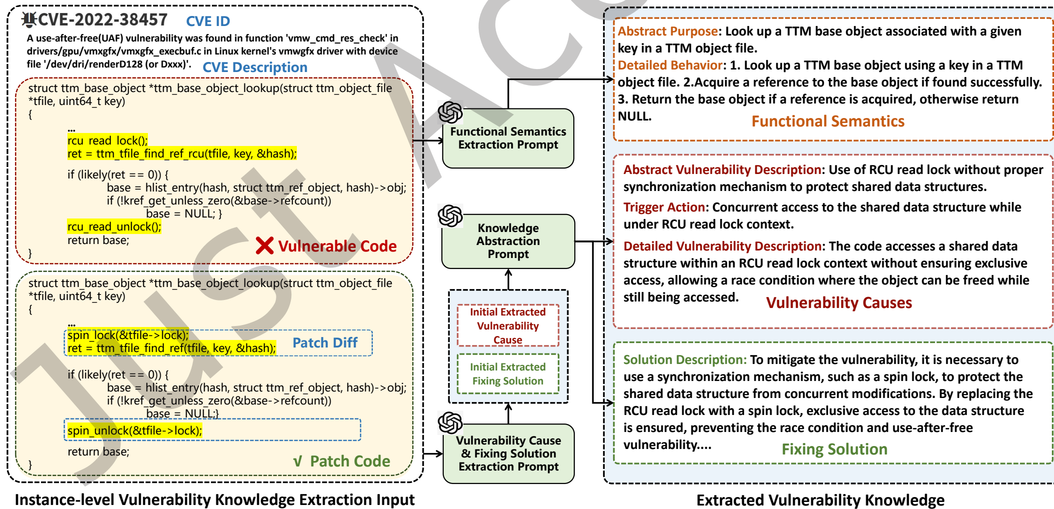


Fig. 3. An Example of Vulnerability Knowledge Extraction from CVE-2022-38457

subsequent online detection process (Section 4.3). Figure 3 exemplifies the multi-dimension representation for the real-world vulnerability CVE-2022-38457.

Just Accepted

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 as follows, 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..."

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 [34, 40, 54, 62]. 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 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]*

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 bound 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. We illustrate two abstraction 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").

Following these guidelines, we design detailed prompts for knowledge abstraction as follows:

Just Accepted

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] ...

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, we use the training set (2317 pairs of vulnerable and patched code) to construct the vulnerability knowledge base, ensuring that there is no data overlap between the evaluation benchmark and the knowledge base (Section 3.1.2).

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 [53] 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 shares high-level functional similarity with 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 the 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 [44] for similarity calculation, a method widely used in search engines due to its efficiency and effectiveness [49]. Previous work [45, 66] demonstrates that BM25 performs comparably to semantic, embedding-based retrieval methods across various RAG scenarios. Given its simplicity, efficiency, and stable performance, we adopt BM25 as the retriever in Vul-RAG. Specifically, 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.

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

Before calculating BM25 similarity, both the query and the retrieval documentation undergo standard preprocessing procedures, including tokenization, lemmatization, and stop word removal [18].
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. 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.

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

Just Accepted

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 [37]. 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 are as follows.

Prompt for Finding Vulnerability Causes: I want you to act as a vulnerability detection expert, given the following code snippet and related vulnerability knowledge, please detect whether there is a similar vulnerability in the code snippet.

Code Snippet: *[Code Snippet]*

Vulnerability Knowledge: In a similar code scenario, the following vulnerabilities have been found: *[Vulnerability causes][Fixing Solution].*

Please check if the above code snippet contains similar vulnerability behaviors mentioned in the vulnerability knowledge. Perform a step-by-step analysis and conclude your response with either <result> YES </result> or <result> NO </result>.

Prompt for Finding Fixing Solutions: I want you to act as a vulnerability detection expert, given the following code snippet and related vulnerability knowledge, please detect whether there are similar necessary solution behaviors in the code snippet, which can prevent the occurrence of related vulnerabilities in the vulnerability knowledge.

Code Snippet: *[Code Snippet]*

Vulnerability Knowledge: In a similar code scenario, the following vulnerabilities have been found: *[Vulnerability causes][Fixing Solution]*

Please check if the above code snippet contains similar solution behaviors mentioned in the vulnerability knowledge. Perform a step-by-step analysis and conclude your response with either <result> YES </result> or <result> NO </result>.

Just Accepted

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-RAG generated knowledge help manual vulnerability comprehension?
- **RQ5 (Bad Cases Analysis):** What is the limitation of Vul-RAG?
- **RQ6 (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 [2], given its rapid response and cost-effectiveness in generating a large volume of vulnerability-related knowledge items [49]. For the online knowledge retrieval, we use Elasticsearch [1] as our search engine. We use 2317 vulnerable-patched

pairs across 1154 CVEs that do not overlap with PairVul as the training set for knowledge base Construction. Detailed dataset division and training data statistics are in Section 3.1.2. For the online knowledge-augmented detection, we study the same four LLMs (GPT-4o, Claude Sonnet 3.5, Qwen2.5-Coder-32B-Instruct, and DeepSeek-R1) as in the study. Vul-RAG and all studied baselines are trained and tested on a Linux server (CPU: Intel Xeon Platinum 8358P, GPU: NVIDIA A800 Tensor Core GPU).

5.1 RQ3: Overall Improvements

Baselines. Besides the basic prompt and three advanced prompts studied in RQ1 and RQ2, we further include a **code-level RAG** baseline and a **fine-tuning** based baselines.

Code-based RAG enhances basic LLMs by retrieving similar code snippets from training datasets to enrich the prompts. In particular, comparing Vul-RAG with code-level RAG can investigate the contribution of our knowledge-level representation. The detailed prompt design of code-level RAG is in Figure 6 (b). Comparing Vul-RAG with code-level RAG can investigate the contribution of our knowledge-level representation.

LLMAO [58] fine-tunes the LLM on the Devign dataset [65] for vulnerability detection. We directly utilize the public implementation of this baseline. To adapt the techniques to our benchmark, we (i) replace the original codegen-16b model with the more advanced Qwen2.5-Coder-32b and DeepSeek-R1-7b model and (ii) fine-tune the baseline on our training set for 10 epochs. Additionally, as LLMAO is a line-level vulnerability detection technique, we adapt it to the function level by regarding the function that has any line with higher-than-0.5 suspiciousness scores as vulnerable.

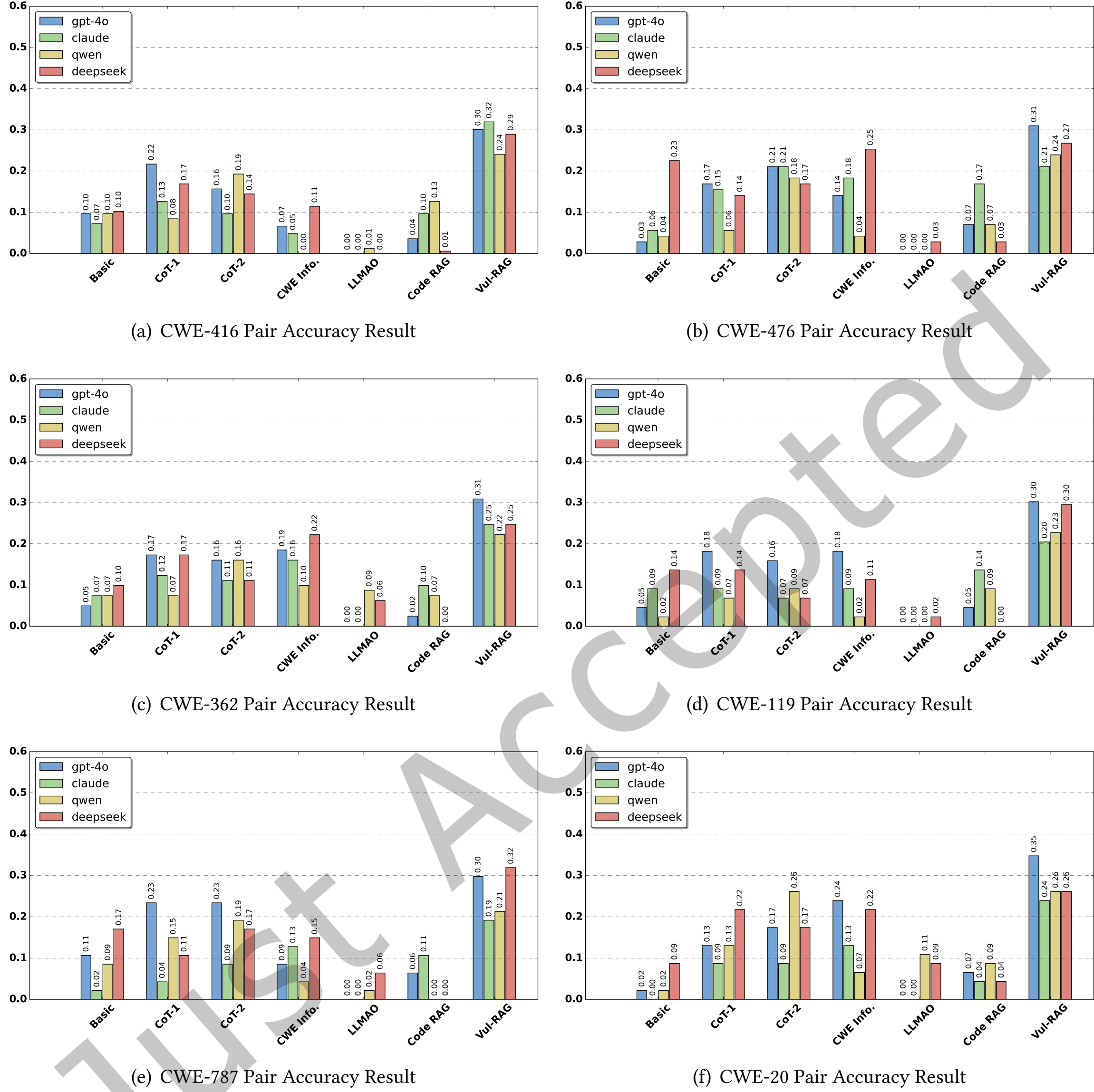
Table 7. Effectiveness of Vul-RAG

Tech.	LLM	Pair Acc.	Bal. Recall	Bal. Pre.
Code RAG	GPT-4o	0.05	0.51	0.54
	Claude	0.11	0.52	0.53
	Qwen	0.09	0.54	0.59
	DeepSeek	0.05	0.52	0.60
LLMAO	Qwen	0.04	0.51	0.51
	DeepSeek	0.04	0.51	0.51
Vul-RAG	GPT-4o	**0.32**	0.58	**0.63**
	Claude	0.27	**0.61**	0.62
	Qwen	0.26	0.59	0.61
	DeepSeek	0.30	**0.61**	0.62

Results. Table 7 compares Vul-RAG with code-level RAG and LLMAO on PairVul. Figure 4 and Figure 5 present the detailed comparison between Vul-RAG and all baselines across the 10 CWE categories. Overall, Vul-RAG substantially outperforms all baselines in all metrics. Particularly, Vul-RAG not only improves the pair accuracy of LLMs (with 16% - 24% increase) but also improves the balanced precision and recall by 9%-14% and 7%-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

Just Accepted



(a) CWE-416 Pair Accuracy Result

(b) CWE-476 Pair Accuracy Result

(c) CWE-362 Pair Accuracy Result

(d) CWE-119 Pair Accuracy Result

(e) CWE-787 Pair Accuracy Result

(f) CWE-20 Pair Accuracy Result

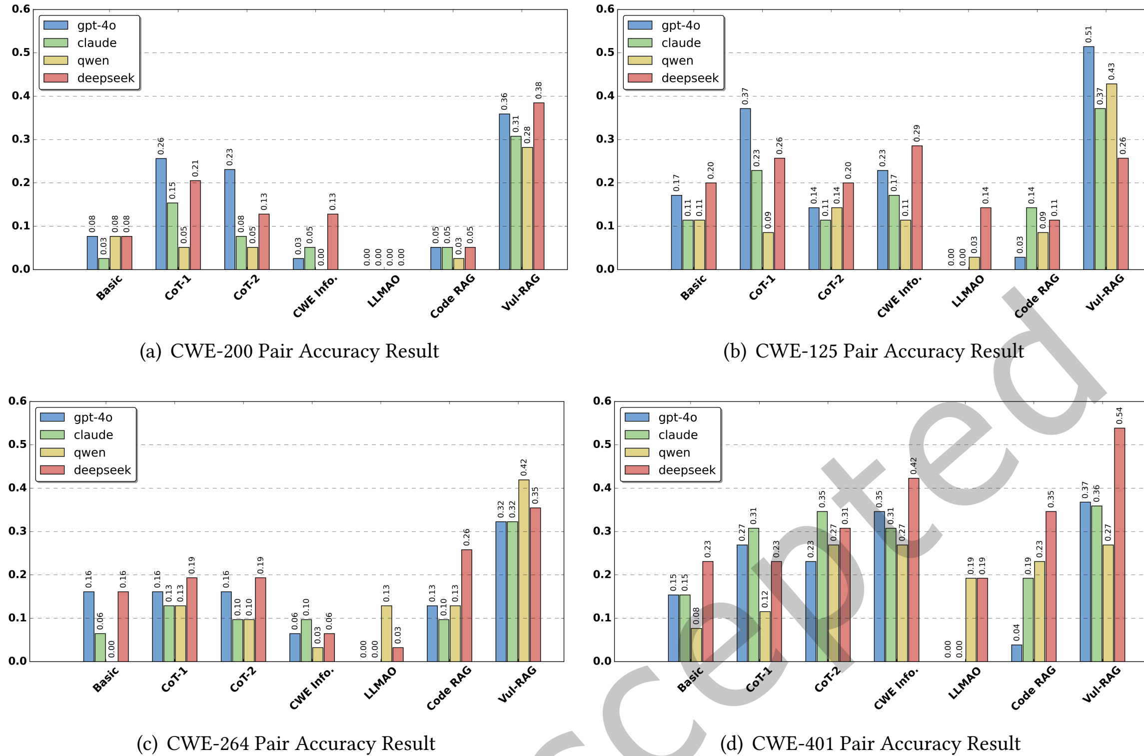
Fig. 4. Comparison of performance for VUL-RAG and Baselines

Just Accepted

vulnerability knowledge provided by VUL-RAG can more accurately prompt LLMs while the plain representation of code pairs used in code-level RAG cannot.

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 strategy.

Knowledge Representation. Figure 6 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-4o. When



(a) CWE-200 Pair Accuracy Result

(b) CWE-125 Pair Accuracy Result

(c) CWE-264 Pair Accuracy Result

(d) CWE-401 Pair Accuracy Result

Fig. 5. Comparison of performance for Vul-RAG and Baselines

detecting the given code from CVE-2023-30772, the basic GPT-4o fails to identify the real cause of the vulnerability (as shown in Figure 6 (A)). GPT-4o 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 real 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-4o detects the corresponding patched code, leading to false positives and affecting the pairwise accuracy. Enhancing GPT-4o with code-based RAG also fails to detect the vulnerability. As shown in Figure 6 (B), although the retrieved code pair contains a similar functional semantics and vulnerability cause, GPT-4o 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-4o not only successfully detects the vulnerability root cause in the vulnerable code but also accurately identifies the patched code (Figure 6 (C)). The comparison demonstrates that high-level vulnerability knowledge can effectively help LLMs understand the behavior of the vulnerable code, thereby improving the accuracy of vulnerability detection.

Retrieval Strategy. Figure 7 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 7, 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 differs 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

Just Accepted

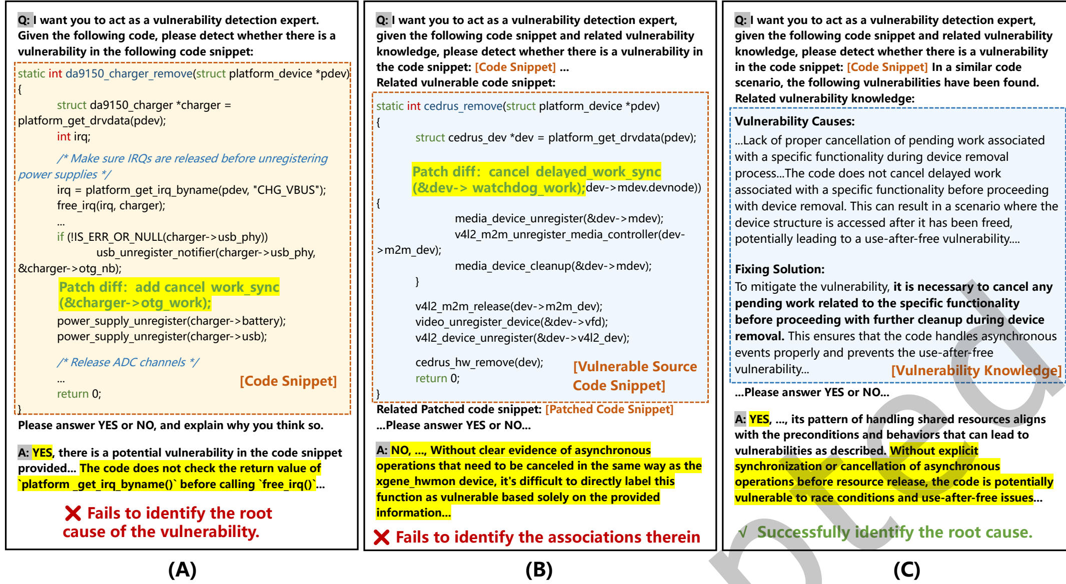


Fig. 6. An example of vulnerability knowledge representation

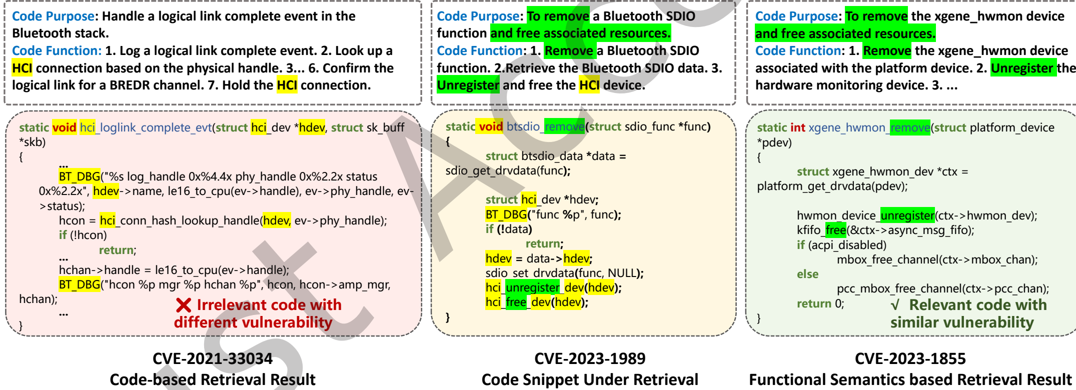


Fig. 7. An example of knowledge retrieval strategy

Just Accepted

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.

5.2 RQ4: Usability for Developers

We conduct a user study to investigate the quality of the vulnerability knowledge generated by Vul-RAG, and whether it can help developers to more accurately identify true positives and more effectively filter out false positives during the manual validation stage in code review.

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 12 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 is paid with 250$ for the experiments. The procedure is approved by the Institutional Review Board (IRB) at our institution.

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.

Metrics. 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 [36] (*i.e.,* 1-disagree; 2-somewhat disagree; 3-somewhat agree; 4-agree).

- **Helpfulness**: The vulnerability knowledge provided by Vul-RAG helps understand the vulnerability and verify detection labels.
- **Preciseness**: The vulnerability knowledge offers 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).

In this evaluation, each human annotator is provided with the following instructions: *"You will be presented with several C/C++ code snippets, each accompanied by a label indicating whether it is predicted as vulnerable or not. For each snippet, your task is to determine whether the code is truly vulnerable. For some snippets, you will also receive additional knowledge generated by the analysis tool to assist in your assessment. In these cases, you will be asked to evaluate the usefulness, clarity, and generalizability of the provided knowledge."*

Results. Participants rate the helpfulness, preciseness, and generalizability with average scores of 3.03, 3.17, and 3.00, 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 with a statistically significant improvement (*i.e.,* 78% detection accuracy with knowledge v.s. 58% detection accuracy without knowledge, $p = 0.01$). It confirms the usability of Vul-RAG generated knowledge for manual vulnerability comprehension.

Just Accepted

5.3 RQ5: Bad Cases Analysis

To understand the limitations of Vul-RAG, we first manually analyze the bad cases, including both false negatives and false positives reported by Vul-RAG. Specifically, we focused on the five most frequently occurring CWE categories (i.e., CWE-416, CWE-476, CWE-362, CWE-119, and CWE-787) for in-depth analysis. For each CWE category, we randomly sampled six false negatives and six false positives, resulting in a total of 60 failure cases for manual inspection. Table 8 summarizes the causes of these bad cases.

In particular, the reasons for false negatives are classified into three primary categories:

Table 8. FN/FP analysis in five most frequent CWE categories

Type	Reason	Number
FN	Inaccurate vulnerability knowledge descriptions.	7
	Irrelevant vulnerability knowledge retrieval	5
	Non-existent relevant vulnerability knowledge.	18
FP	Mismatched fixing solutions.	11
	Irrelevant vulnerability knowledge retrieval	19

- **Inaccurate Vulnerability Knowledge Descriptions.** We observe that for 7 instances (23.3%), 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).
- **Irrelevant vulnerability knowledge retrieval.** We observe that for 5 cases (16.7%) Vul-RAG fails to retrieve relevant vulnerability knowledge, thus leading to false negatives. Although there are instances in the knowledge base that share 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, 18 cases (60%) in this category are caused 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 positives can be classified into the following two categories:
- **Mismatched Fixing Solutions.** There are 11 cases (36.7%) 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 (63.3%) 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 GPT-4o to precisely identify the vulnerability.

Impact of Code Complexity. Table 9 further analyzes the impact of code complexity by comparing the average number of code lines in the full test set with those in bad cases across different CWE categories. Overall, across

Just Accepted

all vulnerability types, bad cases consistently exhibit longer code contexts, with average increases ranging from 3.69 to 91.95 lines. This trend indicates that increased code complexity substantially increases the difficulty of LLM-based reasoning. Specifically, the impact of code complexity is relatively limited for CWE-119 and CWE-787, where bad cases exhibit only marginal increases (approximately three lines) in average code length. Our manual inspection suggests that, for these memory-safety vulnerabilities, vulnerability root causes and repair patterns are largely localized to specific operations and less dependent on overall code length. In contrast, code complexity exerts a much stronger influence on CWE-200 and CWE-264, where bad cases involve code that is more than 40 lines longer on average. These access-control logic vulnerability categories are more tightly coupled with broader program context. As code complexity increases, identifying the underlying vulnerability causes becomes substantially more challenging, posing greater challenges to LLM-based analysis.

Table 9. Impact of Code Complexity

CWE Type	Avg. Code Lines (All samples)	Avg. Code Lines (Error samples)	Pair Acc.
CWE-20	53.59	84.41 (+**30.82**)	0.35
CWE-119	83.14	86.83 (+**3.69**)	0.30
CWE-125	55.09	75.50 (+**20.41**)	0.51
CWE-200	108.13	200.08 (+**91.95**)	0.36
CWE-264	64.26	104.50 (+**40.24**)	0.32
CWE-362	61.63	83.25 (+**21.62**)	0.31
CWE-401	75.46	92.67 (+**17.21**)	0.37
CWE-416	43.88	55.38 (+**11.50**)	0.30
CWE-476	75.44	92.90 (+**17.46**)	0.31
CWE-787	72.87	76.79 (+**3.92**)	0.30

5.4 RQ6: 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 1,568 functions in total. We apply Vul-RAG on GPT-4o-mini to extract the functional semantics required for retrieval and GPT-4o to perform vulnerability detection on these functions. Vul-RAG reported 34 warnings. After manual inspection, we identified 10 previously unknown bugs, and 6 of them have been confirmed as real bugs by the Linux community with assigned CVEs. The confirmed CVEs include: CVE-2024-47747 [8], CVE-2024-49924 [10], CVE-2024-49874 [9], CVE-2024-50059 [11], and CVE-2024-50061 [12], CVE-2025-37838 [13]. Notably, although the resulting precision is relatively low (0.30), it already represents a substantial improvement over existing LLM-based approaches. For example, prompt-engineering–based baselines tend to severely over-predict vulnerabilities in real-world projects, often flagging 50% or more of functions as potentially vulnerable, which renders them impractical for actual use. Moreover, since 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 to write high-quality bug-reporting emails. For the 6 confirmed bugs, we further submitted patches based on the fix solutions provided by Vul-RAG, all of which have already been accepted.

Cost Analysis. Processing all 1,568 functions required approximately 12 hours in total, with an overall cost of $29.8. On average, each case required 27.69 seconds and cost $0.019. Overall, Vul-RAG demonstrates reasonable usability, offering acceptable time and monetary costs for practical use.

Just Accepted

Case Analysis. Figure 8 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 a 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_ 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.

```
1. static void switchtec_ntb_remove(struct device *dev)
2. {
3.   struct switchtec_dev *stdev = to_stdev(dev);
4.   struct switchtec_ntb *sndev = stdev->sndev;
5.
6.   if (!sndev)
7.    return;
8.
9.    stdev->link_notifier = NULL;
10.   stdev->sndev = NULL;
11.   ntb_unregister_device(&sndev->ntb);
12.   switchtec_ntb_deinit_db_msg_irq(sndev);
13.   switchtec_ntb_deinit_shared_mw(sndev);
14.   switchtec_ntb_deinit_crosslink(sndev);
15.   kfree(sndev);
16.   dev_info(dev, "ntb device unregistered\n");
17. }
```

```
1. static int switchtec_ntb_add(struct device *dev)
2. {
…
7.   stdev->sndev = NULL;
…
16.   sndev->stdev = stdev;
17.   rc = switchtec_ntb_init_sndev(sndev);
18.   if (rc)
19.    goto free_and_exit;
…
49.   rc = ntb_register_device(&sndev->ntb);
50.   if (rc)
51.    goto deinit_and_exit;
52.   stdev->sndev = sndev;
53.   stdev->link_notifier = switchtec_ntb_link_notification;
54.   dev_info(dev, "NTB device registered\n");
…}
```

```
1. static int switchtec_ntb_init_sndev(struct
          switchtec_ntb *sndev)
2. {
...
7.   sndev->ntb.pdev = sndev->stdev->pdev;
8.   sndev->ntb.topo = NTB_TOPO_SWITCH;
9.   sndev->ntb.ops = &switchtec_ntb_ops;
10.
11.   INIT_WORK(&sndev->check_link_status_work,
          check_link_status_work);
…
```

```
1. static void switchtec_ntb_link_notification(struct
          switchtec_dev *stdev)
2. {
3.  struct switchtec_ntb *sndev = stdev->sndev;
4.
5.  switchtec_ntb_check_link(sndev, MSG_CHECK_LINK);
6. }
```

```
1. static void switchtec_ntb_check_link(struct switchtec_ntb *sndev,
2.           enum switchtec_msg msg)
3. {
…
7.  schedule_work(&sndev->check_link_status_work);
8. }
```

Fig. 8. An example of a previously-unknown bug in the Linux kernel reported by Vul-RAG

Just Accepted

6 Threats To Validity

Threats in benchmarks. A primary threat concerns the correctness of benchmark labels. Since a single CVE may involve modifications to multiple functions, it is difficult to guarantee that every pre-commit function is vulnerable and that every post-commit function is fully non-vulnerable. Moreover, commits may include code changes unrelated to the vulnerability fix. To mitigate these issues, we designed a set of filtering rules to remove vulnerability-irrelevant modifications and constructed a patch graph to verify patch correctness. While these measures substantially improve label reliability, fully eliminating label noise remains a challenging open problem in vulnerability dataset construction and will be explored in future work.

Threats in generalization. Our benchmark focuses on the Linux kernel CVEs due to their prevalence and rich vulnerability information[41], which might limit the generalization of results. However, our approach is not limited to the Linux kernel CVEs and can be extended to CVEs of other systems in the future. In addition, the

incompleteness of the knowledge base can limit the performance of Vul-RAG in practice. Given the diversity of vulnerabilities, there may be no relevant historical vulnerabilities for the code under detection, which is also a common pain spot for RAG techniques. Therefore, we 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.

7 Conclusion

This work reveals the limitations of LLMs in distinguishing between vulnerable and patched code, and proposes a novel knowledge-level RAG framework Vul-RAG, which enhances LLMs with multi-dimensional vulnerability knowledge distilled from historical vulnerabilities and fixes. 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 with assigned CVEs.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (Grant No.62332005 and 62372114).

References

[1] 2023. *ElasticSearch, https://github.com/elastic/elasticsearch.*
[2] 2023. *GPT-3-5-turbo Documentation, https://platform.openai.com/docs/models/gpt-3-5-turbo.*
[3] 2024. *Claude Sonnet 3.5, https://www.anthropic.com/news/claude-3-family.*
[4] 2024. *DeepSeek-Coder-V2-Instruct, https://huggingface.co/deepseek-ai/DeepSeek-Coder-V2-Instruct.*
[5] 2024. *GPT-4-turbo, https://platform.openai.com/docs/models/gpt-4-and-gpt-4-turbo.*
[6] 2024. *Qwen2.5-Coder-32B- 256 Instruct, https://huggingface.co/Qwen/Qwen2.5-Coder-32B-Instruct.*
[7] 2024. *Replication Package, https://github.com/KnowledgeRAG4LLMVulD/KnowledgeRAG4LLMVulD.*
[8] 2024. *The website of CVE-2024-47747.* https://nvd.nist.gov/vuln/detail/CVE-2024-47747
[9] 2024. *The website of CVE-2024-49874.* https://nvd.nist.gov/vuln/detail/CVE-2024-49874
[10] 2024. *The website of CVE-2024-49924.* https://nvd.nist.gov/vuln/detail/CVE-2024-49924
[11] 2024. *The website of CVE-2024-50059.* https://nvd.nist.gov/vuln/detail/CVE-2024-50059
[12] 2024. *The website of CVE-2024-50061.* https://nvd.nist.gov/vuln/detail/CVE-2024-50061
[13] 2025. *The website of CVE-2025-37838.* https://nvd.nist.gov/vuln/detail/CVE-2025-37838
[14] 2025. *The website of Linux Kernel CVEs, https://lore.kernel.org/linux-cve-announce.*
[15] Guru Bhandari, Amara Naseer, and Leon Moonen. 2021. CVEfixes: automated collection of vulnerabilities and their fixes from open-source software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*. 30–39.
[16] 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* (2024).
[17] 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 (2020), 1877–1901.
[18] 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 (Lecture Notes in Computer Science, Vol. 9489)*, Sabri Arik, Tingwen Huang, Weng Kin Lai, and Qingshan Liu (Eds.). Springer, 168–176. https://doi.org/10.1007/978-3-319-26532-2_19
[19] 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 (2022), 3280–3296. https://doi.org/10.1109/TSE.2021.3087402
[20] Yizheng Chen, Zhoujie Ding, Lamya Alowain, Xinyun Chen, and David Wagner. 2023. Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*. 654–668.

Just Accepted

[21] Roland Croft, M Ali Babar, and M Mehdi Kholoosi. 2023. Data quality for software vulnerability datasets. In 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). *IEEE, 121ś133* (2023).
[22] Seyed Shayan Daneshvar, Yu Nong, Xu Yang, Shaowei Wang, and Haipeng Cai. 2025. VulScribeR: Exploring RAG-based Vulnerability Augmentation with LLMs. *ACM Transactions on Software Engineering and Methodology* (Aug. 2025). https://doi.org/10.1145/3760775
[23] Charlie Dil, Hui Chen, and Kostadin Damevski. 2025. Towards higher quality software vulnerability data using LLM-based patch filtering. *Journal of Systems and Software* (2025), 112581.
[24] Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. 2025. Vulnerability Detection with Code Language Models: How Far Are We?. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering* (Ottawa, Ontario, Canada) *(ICSE '25)*. IEEE Press, 1729–1741. https://doi.org/10.1109/ICSE55347.2025.00038
[25] Xueying Du, Jiayi Feng, Yi Zou, Wei Xu, Jie Ma, Wei Zhang, Sisi Liu, Xin Peng, and Yiling Lou. 2026. Reducing False Positives in Static Bug Detection with LLMs: An Empirical Study in Industry. arXiv:2601.18844 [cs.SE] https://arxiv.org/abs/2601.18844
[26] Xueying Du, Mingwei Liu, Hanlin Wang, Juntao Li, Xin Peng, and Yiling Lou. 2025. Exploring Large Language Models in Resolving Environment-Related Crash Bugs: Localizing and Repairing. arXiv:2312.10448 [cs.SE] https://arxiv.org/abs/2312.10448
[27] Xueying Du, Kai Yu, Chong Wang, Yi Zou, Wentai Deng, Zuoyu Ou, Xin Peng, Lingming Zhang, and Yiling Lou. 2025. Minimizing False Positives in Static Bug Detection via LLM-Enhanced Path Feasibility Analysis. *CoRR* abs/2506.10322 (2025). https://doi.org/10.48550/ARXIV.2506.10322 arXiv:2506.10322
[28] 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*, Sunghun Kim, Georgios Gousios, Sarah Nadi, and Joseph Hejderup (Eds.). ACM, 508–512. https://doi.org/10.1145/3379597.3387501
[29] Reza Fayyazi, Stella Hoyos Trueba, Michael Zuzak, and Shanchieh Jay Yang. 2025. ProveRAG: Provenance-Driven Vulnerability Analysis with Automated Retrieval-Augmented LLMs. arXiv:2410.17406 [cs.CR] https://arxiv.org/abs/2410.17406
[30] 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 (2023). https://doi.org/10.48550/ARXIV.2311.12420 arXiv:2311.12420
[31] 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 (2023). https://doi.org/10.48550/ARXIV.2311.16169 arXiv:2311.16169
[32] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2023. The Hitchhiker's Guide to Program Analysis: A Journey with Large Language Models. (2023). arXiv:2308.00245 [cs.SE]
[33] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2024. Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 111 (apr 2024), 26 pages. https://doi.org/10.1145/3649828
[34] Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2023. Structured chain-of-thought prompting for code generation. *arXiv preprint arXiv:2305.06599* (2023).
[35] Ziyang Li, Saikat Dutta, and Mayur Naik. 2024. LLM-Assisted Static Analysis for Detecting Security Vulnerabilities. arXiv:2405.17238 [cs.CR] https://arxiv.org/abs/2405.17238
[36] Rensis Likert. 1932. A technique for the measurement of attitudes. *Archives of psychology* (1932).
[37] 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 (2023). https://doi.org/10.48550/ARXIV.2307.03172 arXiv:2307.03172
[38] Qiheng Mao, Zhenhao Li, Xing Hu, Kui Liu, Xin Xia, and Jianling Sun. 2024. Towards Effectively Detecting and Explaining Vulnerabilities Using Large Language Models. arXiv:2406.09701 [cs.SE] https://arxiv.org/abs/2406.09701
[39] Noble Saji Mathews, Yelizaveta Brus, Yousra Aafer, Meiyappan Nagappan, and Shane McIntosh. 2024. LLbezpeky: Leveraging Large Language Models for Vulnerability Detection. arXiv:2401.01269 [cs.CR] https://arxiv.org/abs/2401.01269
[40] 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 (2024). https://doi.org/10.48550/ARXIV.2402.17230 arXiv:2402.17230
[41] 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*. IEEE, 112–119. https://doi.org/10.1109/ISSREW60843.2023.00058
[42] 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*, Davide Balzarotti and Wenyuan Xu (Eds.). USENIX Association. https://www.usenix.org/conference/usenixsecurity24/presentation/risse
[43] Niklas Risse, Jing Liu, and Marcel Böhme. 2025. Top score on the wrong exam: On benchmarking in machine learning for vulnerability detection. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 388–410.
[44] 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.*

Just Accepted

Dublin, Ireland, 3-6 July 1994 (Special Issue of the SIGIR Forum). ACM/Springer, 232–241. https://doi.org/10.1016/0306-4573(88)90021-0
[45] Tao Shen, Guodong Long, Xiubo Geng, Chongyang Tao, Yibin Lei, Tianyi Zhou, Michael Blumenstein, and Daxin Jiang. 2024. Retrieval-Augmented Retrieval: Large Language Models are Strong Zero-Shot Retriever. In *Findings of the Association for Computational Linguistics: ACL 2024*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 15933–15946. https://doi.org/10.18653/v1/2024.findings-acl.943
[46] Ze Sheng, Zhicheng Chen, Shuning Gu, Heqing Huang, Guofei Gu, and Jeff Huang. 2025. Llms in software security: A survey of vulnerability detection techniques and insights. *Comput. Surveys* 58, 5 (2025), 1–35.
[47] 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 (2024). https://doi.org/10.48550/ARXIV.2401.17010 arXiv:2401.17010
[48] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Wei Ma, Lyuye Zhang, Yang Liu, and Yingjiu Li. 2025. LLM4Vuln: A Unified Evaluation Framework for Decoupling and Enhancing LLMs' Vulnerability Reasoning. arXiv:2401.16185 [cs.CR] https://arxiv.org/abs/2401.16185
[49] 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. arXiv:2308.03314 [cs.CR]
[50] 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*. IEEE, 862–880. https://doi.org/10.1109/SP54263.2024.00210
[51] 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* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 146–158. https://doi.org/10.1145/3611643.3616256
[52] Ziliang Wang, Ge Li, Jia Li, Hao Zhu, and Zhi Jin. 2025. VulAgent: Hypothesis-Validation based Multi-Agent Vulnerability Detection. arXiv:2509.11523 [cs.SE] https://arxiv.org/abs/2509.11523
[53] 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? arXiv:2406.14497 [cs.SE] https://arxiv.org/abs/2406.14497
[54] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems* 35 (2022), 24824–24837.
[55] 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, Article 168 (jun 2024), 34 pages. https://doi.org/10.1145/3653718
[56] Ratnadira Widyasari, David Lo, and Lizi Liao. 2024. Beyond ChatGPT: Enhancing Software Quality Assurance Tasks with Diverse LLMs and Validation Techniques. arXiv:2409.01001 [cs.SE] https://arxiv.org/abs/2409.01001
[57] 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 (2023). https://doi.org/10.48550/ARXIV.2312.05275 arXiv:2312.05275
[58] 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*. ACM, 17:1–17:12. https://doi.org/10.1145/3597503.3623342
[59] Xu Yang, Wenhan Zhu, Michael Pacheco, Jiayuan Zhou, Shaowei Wang, Xing Hu, and Kui Liu. 2025. Code Change Intention, Development Artifact and History Vulnerability: Putting Them Together for Vulnerability Fix Detection by LLM. arXiv:2501.14983 [cs.SE] https://arxiv.org/abs/2501.14983
[60] Jeffy Yu. 2024. Retrieval Augmented Generation Integrated Large Language Models in Smart Contract Vulnerability Detection. *ArXiv* abs/2407.14838 (2024). https://api.semanticscholar.org/CorpusID:271328891
[61] 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 (2023). https://doi.org/10.48550/ARXIV.2308.12697 arXiv:2308.12697
[62] Zhuosheng Zhang, Aston Zhang, Mu Li, and Alex Smola. 2022. Automatic chain of thought prompting in large language models. *arXiv preprint arXiv:2210.03493* (2022).
[63] Xin Zhou, Sicong Cao, Xiaobing Sun, and David Lo. 2025. Large language model for vulnerability detection and repair: Literature review and the road ahead. *ACM Transactions on Software Engineering and Methodology* 34, 5 (2025), 1–31.
[64] Xin Zhou, Ting Zhang, and David Lo. 2024. Large Language Model for Vulnerability Detection: Emerging Results and Future Directions *(ICSE-NIER'24)*. Association for Computing Machinery, New York, NY, USA, 47–51. https://doi.org/10.1145/3639476.3639762
[65] 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*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 10197–10207. https://proceedings.neurips.cc/paper/2019/hash/49265d2447bc3bbfe9e76306ce40a31f-Abstract.html

Just Accepted

[66] Yutao Zhu, Huaying Yuan, Shuting Wang, Jiongnan Liu, Wenhan Liu, Chenlong Deng, Zhicheng Dou, and Ji rong Wen. 2023. Large Language Models for Information Retrieval: A Survey. *ACM Transactions on Information Systems* (2023). https://api.semanticscholar.org/CorpusID:260887838

[67] Arastoo Zibaeirad and Marco Vieira. 2024. VulnLLMEval: A Framework for Evaluating Large Language Models in Software Vulnerability Detection and Patching. arXiv:2409.10756 [cs.SE] https://arxiv.org/abs/2409.10756

Just Accepted