Benchmarking LLMs and LLM-based Agents in Practical Vulnerability Detection for Code Repositories

Alperen Yildiz[1], **Sin G. Teo**[2], **Yiling Lou**[3], **Yebo Feng**[4], **Chong Wang**[4*], **Dinil Mon Divakaran**[2]
[1]National University of Singapore, Singapore
[2]Institute for Infocomm Research (I^{2}R), A*STAR, Singapore
[3]Fudan University, China
[4]Nanyang Technological University, Singapore

Abstract

Large Language Models (LLMs) have shown promise in software vulnerability detection, particularly on function-level benchmarks like Devign and BigVul. However, real-world detection requires interprocedural analysis, as vulnerabilities often emerge through multi-hop function calls rather than isolated functions. While repository-level benchmarks like ReposVul and VulEval introduce interprocedural context, they remain computationally expensive, lack pairwise evaluation of vulnerability fixes, and explore limited context retrieval, limiting their practicality.

We introduce JITVUL, a JIT vulnerability detection benchmark linking each function to its vulnerability-introducing and fixing commits. Built from 879 CVEs spanning 91 vulnerability types, JITVUL enables comprehensive evaluation of detection capabilities. Our results show that ReAct Agents, leveraging thought-action-observation and interprocedural context, perform better than LLMs in distinguishing vulnerable from benign code. While prompting strategies like Chain-of-Thought help LLMs, ReAct Agents require further refinement. Both methods show inconsistencies, either misidentifying vulnerabilities or over-analyzing security guards, indicating significant room for improvement.

1 Introduction

Given the success of large language models (LLMs) across various application domains, researchers have begun exploring their effectiveness in software vulnerability detection. On well-known vulnerability detection benchmarks such as Devign (Zhou et al., 2019) and BigVul (Fan et al., 2020), LLMs—particularly those fine-tuned on code—have shown promising results, suggesting their potential for real-world applications.

Benchmark	# CVEs	# CWEs	Pairwise	Agents Eval
ReposVul	6,134	236	✗	✗
VulEval	4,196	5	✗	✗
Lomio et al.	27	11	✓	✗
JITVUL (ours)	879	91	✓	✓

Table 1: Comparison of JITVUL with existing benchmarks for repository-level vulnerability detection.

However, a significant gap exists between these widely used benchmarks and the requirements for real-world vulnerability detection in code repositories (Wang et al., 2024; Wen et al., 2024b). These benchmarks primarily focus on function-level vulnerability detection, where a single function is input to a detector for label prediction without considering the broader repository context. In contrast, real-world vulnerabilities—such as null pointer dereference (NPD)—often arise within multi-hop function call chains, and not in isolated functions. Detecting such vulnerabilities requires tracing interprocedural call relationships and understanding the relevant code elements like branch conditions (Risse and Böhme, 2024a).

To address these limitations, recent studies have shifted towards repository-level detection scenarios, enabling more realistic benchmarking of LLMs for vulnerability detection. ReposVul (Wang et al., 2024) and VulEval (Wen et al., 2024b) are two benchmarks that enhance the detection process by extracting callers and callees for a target function from the code repository, providing interprocedural context. These callers (functions that call the target function) and callees (functions called by the target function) are selectively fed into LLMs to assess the vulnerability of the target function. Findings from these benchmarks show that while LLMs benefit from the additional interprocedural context, they still exhibit low effectiveness in real-world scenarios, particularly when fine-tuning is not applied.

Although existing works have provided valu-

*Chong Wang is the corresponding author.

able insights into the effectiveness of LLMs for repository-level vulnerability detection, several key limitations remain in achieving more comprehensive benchmarking. First, in existing repository-level vulnerability detection approaches, all functions within a code repository are treated as target functions for vulnerability detection. This approach becomes computationally expensive and impractical, particularly for large repositories like the Linux kernel. Second, the benchmarks do not effectively assess the capability of LLMs in distinguishing between vulnerable functions and those where the vulnerability has been patched. As highlighted by a recent study (Risse and Böhme, 2024b), this is a critical limitation of machine learning-based vulnerability detection methods. Finally, the integration of interprocedural context (*i.e.,* callers and callees) has mostly been limited to retrieval-based strategies, leaving many potential approaches underexplored. LLM-based agentic methods, such as ReAct (Yao et al., 2022), offer the potential for on-demand, iterative acquisition of interprocedural context, enabling more adaptive analysis.

To bridge the gap, we target the task of ***just-in-time (JIT) vulnerability detection*** (Lomio et al., 2022), a more practical approach for identifying vulnerabilities in code repositories. Unlike prior methods that analyze all functions in a repository, JIT vulnerability detection is triggered only for functions modified in a commit, with interprocedural context provided. Inspired by prior research (Risse and Böhme, 2024b), we construct a pairwise benchmark called JITVUL for JIT vulnerability detection, where each target function is linked to both a vulnerability-introducing commit and a vulnerability-fixing commit. To achieve this, we first select 879 Common Vulnerabilities and Exposures (CVE) entries, each representing a unique vulnerability, from PrimeVul, a high-quality function-level detection dataset (Ding et al., 2024). We then extract target functions from the vulnerability-fixing commits explicitly referenced in the CVE entries, obtaining both their vulnerable and patched versions. Finally, we analyze the commit history of each vulnerable function to identify the corresponding vulnerability-introducing commit. The resulting JITVUL comprises 1,758 paired commits spanning 91 Common Weakness Enumerations (CWEs).

We implement LLMs and ReAct Agents with various prompting strategies and foundation models to assess their effectiveness in JIT vulnerability detection. Our evaluation on JITVUL uncovers several key findings. A higher F_1 score doesn't always reflect a method's ability to capture vulnerability characteristics, highlighting the need for pairwise evaluation. ReAct Agents, using their thought-action-observation framework and interprocedural context, better differentiate between vulnerable and benign versions. While strategies like CoT and few-shot examples boost LLMs' performance, ReAct Agents need more tailored designs. Both LLMs and ReAct Agents often exhibit inconsistent analysis patterns between the vulnerable and benign versions, indicating a lack of robustness in vulnerability analysis. These findings highlight key areas for future research in vulnerability detection: (i) developing more comprehensive evaluation guidelines for benchmarking LLMs and LLM-based agents in JIT vulnerability detection, (ii) exploring advanced prompting strategies for improving LLM-based agents in vulnerability detection, and (iii) designing robust reasoning models tailored to vulnerability analysis that capture the true essence of vulnerabilities, rather than relying on speculation.

This paper makes the following contributions:

- We introduce JITVUL, a benchmark for just-in-time (JIT) vulnerability detection in code repositories, consisting of 1,758 pairwise commits spanning 91 vulnerability types.
- We implement ReAct agents with various prompting strategies and foundation models and evaluate their effectiveness in leveraging interprocedural context for JIT detection.
- Our experimental results provide valuable insights into the application of LLMs and LLM-based agents for real-world vulnerability detection. We explore the necessity of pairwise evaluation, the advantages and disadvantages of LLMs and ReAct agents, and the impact of prompting designs and foundation models.
- We release all code and data at this repository.

2 Related Work

2.1 Vulnerability Detection Benchmarks

Several benchmarks have been proposed for function-level vulnerability detection. BigVul (Fan et al., 2020) collects C/C++ vulnerabilities from the CVE database, filtering out entries without public Git repositories, and labels functions as vulnerable or non-vulnerable

based on commit fixes. MegaVul (Ni et al., 2024) improves upon existing benchmarks by using code parsing tools for accurate function extraction and de-duplicating functions referenced by multiple CVEs. DiverseVul (Chen et al., 2023) ensures data quality by filtering vulnerability-introducing commits with specific keywords and deduplicating function bodies with hash functions. PrimeVul (Ding et al., 2024) addresses data quality challenges by proposing filtering rules to handle noise labels and duplicated functions.

For repository-level vulnerability detection, ReposVul (Wang et al., 2024) addresses issues with tangled and outdated patches, using trace-based filtering to ensure data quality and integrating repository-level features to provide richer context for detection. VulEval (Wen et al., 2024b) provides a framework that collects high-quality data from sources like Mend.io Vulnerability Database (Mend.io, 2025) and National Vulnerability Database (NIST, 2025), including contextual information like caller-callee relationships.

2.2 LLM-based Vulnerability Detection

Recent studies have explored the use of LLMs for vulnerability detection, highlighting their ability to enhance both the identification and explanation of software vulnerabilities. InferROI (Wang et al., 2025) enhances static resource leak detection with LLM-inferred resource operation oriented intentions. LLM4SA (Wen et al., 2024a) integrates language models with SAST tools, leveraging LLMs to inspect static analysis warnings and significantly reduce false positives. LLM4Vuln (Sun et al., 2024) enhances LLM execution by incorporating more context through a retrieval-augmented generation (RAG) pipeline and static analysis. LSAST (Keltek et al., 2024) further explores context augmentation, comparing multiple RAG pipelines using static analysis outputs, vulnerability reports, and code abstraction. Similarly, Vul-RAG (Du et al., 2024) constructs a vector database of vulnerability reports alongside an language model engine. (Zhou et al., 2024b) introduce a voting mechanism that combines SAST tools and LLMs for vulnerability detection.

2.3 LLMs and LLM-based Agents

Various methods have been explored to enhance LLM performance, with prompt augmentation being a key approach that enriches prompts to improve the model's reasoning process. Chain-of-thought (CoT) prompting (Wei et al., 2022) is one of the most widely used techniques, where instructions like "Let's think step by step" guide the model to break problems into sub-problems. Few-shot prompting (Brown et al., 2020) is another common method, providing example traces to enable in-context learning without modifying model weights. Both CoT and few-shot prompting are frequently employed in LLM-based vulnerability detection (Zhou et al., 2024b; Wen et al., 2024a).

Agentic architectures are among the most promising state-of-the-art technologies but remain underexplored in vulnerability detection (Zhou et al., 2024a). (Yao et al., 2022) introduce Reasoning and Acting (ReAct) agents, which generate reasoning and action traces in an interleaved manner to interact with their environment and analyze resulting observations. This iterative process continues until the agent determines a final answer. Reflexion agents (Shinn et al., 2023) are similar to ReAct agents however they focus on self-reflection and dynamic memory updates alongside with reinforcement learning. Self-refine agents (Madaan et al.) are another type of agents where the same language model is instructed to provide feedback based on the output. There have also been several multi-agent systems, such as Alpha-Codium (Ridnik et al.), where the agents are represented as nodes on graphs.

3 JitVul: Just-in-Time Vulnerability Detection for Code Repositories

In this section, we outline the requirements for just-in-time (JIT) vulnerability detection in the context of benchmarking LLMs and LLM-based agents, and introduce a benchmark derived from real-world vulnerabilities.

3.1 Problem Statement

Benchmarking vulnerability detection in real-world code repositories requires considering three key practicality requirements:

- **Interprocedural Context.** Many vulnerabilities originate from interprocedural interactions, even though their manifestation and required fixes often occur within individual functions (Wang et al., 2024). For instance, a null pointer dereference (NPD) vulnerability may arise when a pointer initialized as null in one function is improperly dereferenced in another function along the execution path. Identi-

fying such vulnerabilities necessitates analyzing interprocedural dependencies, as examining functions in isolation is insufficient. A recent study (Li et al., 2024) reports that approximately 24.3% of vulnerabilities in real-world C/C++ projects are interprocedural, typically requiring an average of 2.8 levels of call relationships.

- **Scalability.** A straightforward application of learning-based methods to vulnerability detection for code repositories entails scanning each function individually and predicting a binary label. However, in the context of LLMs and LLM-based agents, this approach becomes computationally infeasible for large-scale repositories due to the high processing costs and resource constraints associated with analyzing extensive codebases. A more practical strategy is to focus on a limited set of candidate functions. *Just-in-time* vulnerability detection (Lomio et al., 2022) exemplify this by prioritizing functions that have been newly introduced or modified in commits.
- **Pairwise Comparison.** Traditional evaluation methods for machine learning-based vulnerability detection present models with labeled vulnerable code alongside other functions in the repository. However, recent findings (Risse and Böhme, 2024b) suggest that models struggle to distinguish between vulnerable code and its patched, benign version, indicating an over-reliance on superficial patterns rather than meaningful vulnerability indicators. This highlights the necessity of pairwise benchmarking to ensure reliable evaluation of vulnerability detection methods.

Although recent works address some of these requirements, to the best of our knowledge, no study fully satisfies all three. For example, PrimeVul (Ding et al., 2024) is a high-quality pairwise dataset, but it focuses on function-level detection. On the other hand, VulEval (Wen et al., 2024b) is a repository-level detection benchmark, but it lacks pairwise evaluation and considers only callers and callees modified in the same commit when extracting interprocedural dependencies.

3.2 Task Definition

We define the task of *just-in-time vulnerability detection* as follows. Given a code repository $\mathcal{R}$ and a target function f modified in a commit, the task is formulated as:

$$\text{JITDETECT} : (\mathcal{R}, f) \rightarrow \{\texttt{vul}, \texttt{ben}\},$$

where f is classified as either vulnerable (`vul`) or benign (`ben`) based on the (interprocedural) context within $\mathcal{R}$. Building on this, we propose a pairwise benchmark to evaluate LLMs and LLM-based agents.

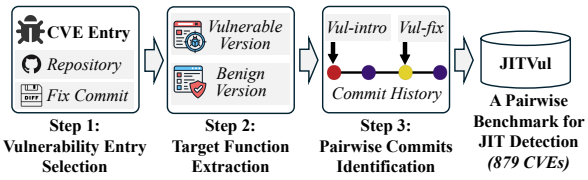

Figure 1: Construction process of JITVUL.

3.3 Benchmark Construction

We construct a benchmark called JITVUL for practical Just-In-Time (JIT) vulnerability detection for code repositories, building on the function-level detection dataset PrimeVul (Ding et al., 2024). As presented in Figure 1, the construction process involves three key steps: Vulnerability Entry Selection, Target Function Extraction, and Pairwise Commit Identification. Note that we did not use the JIT detection dataset curated by (Lomio et al., 2022) for the following reasons. First, it includes only 27 CVE entries across 11 CWEs in 9 Java projects, which is relatively small for effectively benchmarking the performance of LLMs and LLM-based agents. Second, as discussed in (Ding et al., 2024), the quality of the dataset should be ensured to avoid introducing noise into the evaluation.

Step 1: Vulnerability Entry Selection. We begin by selecting CVE entries that meet three key criteria: (i) the selected CVEs should cover a broad range of CWEs, representing different categories of vulnerabilities, (ii) each CVE should have an associated GitHub repository with a complete commit history, and (iii) each CVE should correspond to a commit that fixes the vulnerability. To satisfy these criteria, we leverage the PrimeVul dataset as our foundation. PrimeVul ensures high data quality by including only CVEs that were fixed in a single commit modifying a single function. From PrimeVul, we randomly select 879 CVE entries, ensuring coverage across 91 CWEs, with a maximum of ten CVEs per CWE.

Step 2: Target Function Extraction. For each selected CVE, we retrieve the target function and the commit that fixes the vulnerability directly from the PrimeVul dataset. The versions before

and after the fix correspond to the vulnerable and benign versions of the target function, denoted as f_{vul} and f_{ben}, respectively.

Step 3: Pairwise Commits Identification. Unlike function-level vulnerability detection studies like PrimeVul, our JIT detection requires identifying the commits that trigger vulnerability detection for both f_{vul} and f_{ben}, and obtaining the corresponding repository versions to provide necessary context like callers and callees. To achieve this, we extract the two commits responsible for introducing and fixing the vulnerability, referred to as the *vul-intro* and *vul-fix* commits, respectively.

The *vul-fix* commit can be directly obtained from the data retrieved in the previous step, and the repository version corresponding to it is denoted as $\mathcal{R}_{fix}$. Identifying the precise *vul-intro* commit is more challenging, as pinpointing the exact code change that introduces the vulnerability requires considering the complex interactions between functions. Following the methodology from prior JIT detection work (Lomio et al., 2022), we trace the change history of the target function to approximate the *vul-intro* commit. Specifically, we traverse the commit history backward, examining each commit until we identify the commit where the target function was last modified to become the f_{vul} version. This commit is then designated as the *vul-intro* commit, and the corresponding repository is denoted as $\mathcal{R}_{intro}$.

In the *vul-intro* and *vul-fix* commits, the target function is modified into the f_{vul} and f_{ben} versions, respectively, thereby triggering the JIT detection process.

Resulting Benchmark. After completing the above steps, JITVUL includes 1,758 pairwise data samples, with 879 labeled as vulnerable and 879 as benign. Each sample consists of a specific code repository version, a target function, and a ground-truth label (*i.e.,* `vul` or `ben`). These samples are derived from 879 CVE entries, spanning 91 CWEs. Regarding the code repositories, the average number of code files per repository is 2,955.9. Approximately 9% of the repositories primarily use C++, while 91% utilize C. The mean number of thousands of lines of code (KLoC) per repository is 988.8, with a median of 297.6 and a standard deviation of 1,991.2. For the target function, the average number of lines of code is 696.4.

4 Experimental Setup

4.1 Studied Methods

We investigate three categories of detection methods: Plain LLM, Dependency-Augmented (Dep-Aug) LLM, and ReAct Agent. We choose the ReAct agent over alternatives because its thought-action-observation workflow aligns well with the need for on-demand interprocedural analysis in JIT vulnerability detection.

- **Plain LLM** employs a single LLM with a prompt for vulnerability detection, resembling a function-level detection approach. The LLM is given only the target function to determine whether it is vulnerable. The detailed prompts can be found in Appendix A.1.
- **Dep-Aug LLM** extends the Plain LLM by incorporating Top-5 similar callers and callees of the target function into the prompt through a lexical retrieval approach, such as Jaccard Similarity. This method, proposed and evaluated in VulEval (Wen et al., 2024b), is reproduced in our work based on the original paper. We use it as a baseline that integrates interprocedural context into LLMs in a deterministic manner.
- **ReAct Agent** performs an iterative thought-action-observation process as illustrated in Appendix Figure 4, which is equipped with three tools for on-demand interprocedural context acquisition: (i) **get_callers** returns the function names and line numbers of callers for the input function; (ii) **get_callees** returns the function names and line numbers of callees for the input function; (iii) **get_definition** retrieves the complete function definition based on the input function name and line number. Using these three tools, our ReAct agent for JIT detection follows the workflow outlined below. In each iteration, the agent first reasons based on the current context and the observations from the previous iteration. It then decides whether to call the tools for interprocedural context or to stop the iteration and make a prediction. After observing the tool outputs, the agent proceeds to the next iteration.

For each category, besides the *vanilla* version with a basic prompt, we also design three variants based on different prompting strategies: *chain-of-thought (CoT)*, *few-shot examples (FS)*, and *combination of both CoT and FS*.

- **CoT**: We use a basic CoT approach by adding

the instruction "Solve this problem step by step..." to prompt the LLM and ReAct agent to break down the reasoning tasks. We avoid more complex CoT instructions, as summarizing reasoning patterns for vulnerabilities in advance is challenging, particularly given the wide variety of vulnerability types (CWEs).

- **FS**: We provide several pairwise examples, each consisting of a vulnerable code snippet with a detailed explanation of the vulnerability and its patched benign version with an explanation of the applied safeguard. These few-shot examples are expected to offer context for the LLM and ReAct agent to distinguish between vulnerable and benign code, helping them focus on the actual vulnerability features.

For each variant, we further employ two different foundation models: GPT-4o-mini and GPT-4o.

4.2 Metrics

For each *vul-intro* and *vul-fix* commit pair, we apply detection methods to the corresponding repository ($\mathcal{R}_{intro}$ or $\mathcal{R}_{fix}$) and target function (f_{vul} or f_{ben}), then compare predictions with ground truth. In addition to the commonly used F_1 score, we also assess effectiveness using *pairwise accuracy* (***pAcc***), inspired by PrimeVul (Ding et al., 2024). This metric reflects the proportion of pairs where both functions are correctly labeled, *i.e.,* JITDETECT$(\mathcal{R}_{intro}, f_{vul}) =$ `vul` and JITDETECT$(\mathcal{R}_{fix}, f_{ben}) =$ `ben`.

$$F_1 = \frac{2 \times TP}{2 \times TP + FP + FN},$$

$$pAcc = \frac{\textit{\# of Correctly Labeled Pairs}}{\textit{\# of Total Pairs}},$$

where TP is the number of true positives, FN is the number of false negatives, and FP is the number of false positives.

4.3 Implementation

Few-shot Example Creation. To support few-shot variants, we manually create ten example pairs for both the LLM and the agent. Each pair consists of a vulnerable example and a benign example, sourced from the 2024 CWE Top 25 Most Dangerous Software Weaknesses list[1] on the CWE website. For each weakness in the Top 25 list, we review its corresponding web page to identify a relevant C/C++ code snippet. These snippets typically serve as illustrative examples of how the vulnerability manifests, often accompanied by detailed explanations. We use the original example as the vulnerable code, making minor edits to the explanation for normalization (*e.g.,* merging two paragraphs before and after the code into a single cohesive text). Subsequently, we manually modify the code snippet to address the vulnerability, drafting an explanation of why the modified code is benign, referencing the original explanation of the vulnerable code. In this manner, we create ten pairs of vulnerable and benign examples, each accompanied by its respective explanation. Appendix Figure 6 provides an illustrative example.

[1] https://cwe.mitre.org/top25/index.html

Method	GPT-4o-mini		GPT-4o	
	F_1	*pAcc*	F_1	*pAcc*
Plain LLM				
- vanilla	56.00	3.36	65.96	1.02
- w/ CoT	65.10	3.36	62.22	15.02
- w/ FS	48.74	7.56	62.77	4.44
- w/ CoT+FS	64.65	11.76	64.44	17.63
Dep-Aug LLM				
- vanilla	52.68	2.05	63.30	1.03
- w/ CoT	66.05	4.86	62.60	18.66
- w/ FS	48.23	7.27	62.03	2.39
- w/ CoT+FS	65.01	4.68	61.12	18.79
ReAct Agent				
- vanilla	56.63	12.61	57.77	17.63
- w/ CoT	56.93	16.81	58.07	19.13
- w/ FS	56.81	20.17	56.42	18.91
- w/ CoT+FS	51.06	14.29	52.61	18.89

Table 2: Results of studied methods on JITVUL. The best and second-best results are highlighted.

Caller and Callee Extraction. Given a code repository and a function, we extract its callers and callees using CFlow (GNU, 2025), a widely used tool for on-demand call graph construction. For each caller or callee identified by CFlow, we then use CTags (universal ctags, 2025) to extract its complete function body from its definition in the code repository. These are implemented as Python functions using LangChain's tool decorator for integration into the ReAct workflow.

LLMs and Agents. We use two commercial, general-purpose LLMs, GPT-4o-mini and GPT-4o with the temperature set to 0. The pipeline is constructed using Transformers (v4.52.0) and LangChain (v0.3.14).

5 Results and Analyses

Table 2 presents the results of the studied detection methods on JITVUL.

5.1 Detection Method Comparison

We compare the three categories of detection methods based on both F_1 and $pAcc$ scores.

Results. The results indicate that ReAct Agents achieve higher $pAcc$ scores than other LLM-based methods across all prompting strategies, with improvements ranging from 0.1% to 16.61%, with the largest gain occurring with GPT-4o and the vanilla prompting. However, Plain LLMs and Dep-Aug LLMs generally achieve higher F_1 scores than ReAct under most settings, with improvements of 4.15%-13.95%, except when using GPT-4o-mini with the CoT and CoT+FS prompting strategies. Additionally, Dep-Aug LLMs do not show consistent improvements over Plain LLMs and even exhibit performance degradation with certain prompting strategies.

The LLM-based vulnerability detection methods often over-classify both vulnerable and benign instances as vulnerable, leading to pairwise accuracy that can fall below the level of random guessing, a well-documented issue. For example, GPT-4 achieves only 5.14% accuracy with two-shot prompting and 12.94% with chain-of-thought reasoning, compared to 22.70% for random guessing (Ding et al., 2024). Interestingly, larger models tend to perform worse than smaller ones, as they are more susceptible to over-predicting vulnerabilities (Ding et al., 2024; Zhou et al., 2024b). These challenges highlight the intrinsic difficulty of vulnerability detection, which demands deep comprehension of function semantics, interprocedural relationships, execution logic, vulnerability patterns, and complex reasoning. Although our evaluation shows that agent-based methods offer measurable improvements, substantial progress is still needed. We share several insights to guide future development of more effective approaches.

Findings. Two findings emerge from results.

A higher F_1 score does not necessarily indicate a detection method's superior ability to capture vulnerability characteristics. Vulnerability detection methods exhibit an inconsistent relationship between $pAcc$ (pairwise accuracy) and F_1 (isolated metric). LLM-based methods predict significantly more `vul` labels—exceeding 90% in certain settings—compared to ReAct Agents, leading to higher recall. However, precision remains similar across methods, around 50%, as observed on our *label-balanced* benchmark.. These factors explain the higher F_1 scores of LLM-based methods on JITVUL. This also highlights F_1's sensitivity to the data distribution, emphasizing the need for pairwise evaluation to better capture core vulnerability characteristics (Risse and Böhme, 2024b). We include an additional non-paired dataset to further demonstrate F_1's sensitivity. Specifically, we evaluate the performance of GPT-4o-mini using the original PrimeVul (Ding et al., 2024) test set, which contains 695 vulnerable and 25,216 benign functions. As shown in Table 3, the shift in data distribution significantly impacts the F_1 score, underscoring the importance of pairwise detection for more reliable benchmarking of LLMs and LLM-based agents in vulnerability analysis.

Prompting Method	F_1
- vanilla	9.20
- w/ CoT	5.41
- w/ FS	12.21
- w/ CoT+FS	5.86

Table 3: F_1 Score of GPT-4o-mini on a non-pair dataset with 695 vulnerable and 25,216 benign functions.

The thought-action-observation framework of ReAct Agents, combined with their effective use of interprocedural context, enhances their ability to capture vulnerability characteristics. ReAct Agents conduct in-depth, fine-grained analysis by iteratively and adaptively retrieving additional context, such as callers and callees, rather than relying on superficial analysis or speculation. This allows them to differentiate between code versions before and after vulnerability fix, leading to consistent improvements in $pAcc$. In contrast, while Dep-Aug LLMs incorporate interprocedural context, they rely on mechanical retrieval based on similarity metrics (*e.g.,* Jaccard Similarity), feeding retrieved Top-5 callers and callees all at once, which may introduce noise. This could explain why Dep-Aug LLMs sometimes show a lower $pAcc$ than Plain LLMs. In comparison, ReAct agents demonstrate average improvements in $pAcc$ of 9.46% over Dep-Aug LLMs with GPT-4o-mini and 8.42% with GPT-4o, across various prompting strategies. To demonstrate the adaptive use of interprocedural context of ReAct Agent, we present the distribution of tool invocations for ReAct with GPT-4o and vanilla prompting in Appendix 2. It shows that ReAct Agent dynamically invokes the tools one to three times to retrieve the necessary callers or callees for most cases, in contrast to Dep-Aug LLM, which feeds a fixed number of callers and callees.

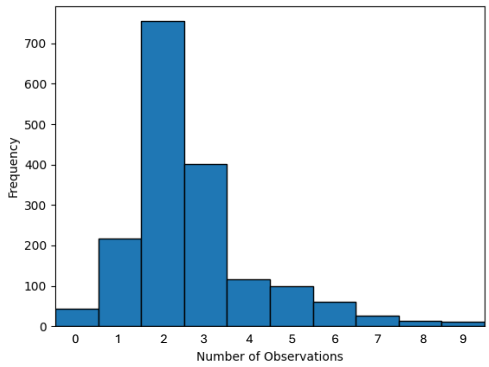

Figure 2: Distribution of tool invocations for ReAct Agent with GPT-4o and vanilla prompting.

5.2 Prompting Strategy Comparison

We also compare the three prompting strategies and perform a detailed analysis.

Results. When applying different prompting strategies, such as CoT and FS, detection methods show varying degrees of improvement in $pAcc$ scores, ranging from 1.26% to 17.76%. However, these prompting strategies do not consistently lead to F_1 improvements in the pairwise evaluation.

Findings. We find two key findings.

Popular prompting strategies like CoT and FS examples can enhance LLMs' performance in pairwise JIT evaluation. While these strategies sometimes reduce F_1 scores, the pairwise metric $pAcc$ shows that LLMs can significantly benefit from CoT instructions (even something as simple as "Solve this problem step by step...") and pairwise FS examples. This improvement is often overlooked when focusing solely on F_1 and should be considered when designing methods.

ReAct Agents require further design improvements when using prompting strategies. The current prompts are straightforward and align better with the inference process of LLMs, meaning the improvements from these strategies for ReAct Agents are relatively smaller than for LLM-based methods. For instance, the FS examples consist of singleton code snippets that do not require interprocedural analysis, which limits the benefit for ReAct Agents that rely on interprocedural context. This suggests a research opportunity in developing agent-oriented prompting strategies specifically for vulnerability detection.

5.3 Foundation Model Comparison

To evaluate the effectiveness of different foundation models across various detection methods, we conduct a comparative analysis. Additionally, we include two open-source models, LLaMA 3.1-8B and DeepSeek-Coder-V2-16B (a code specific model), for further comparison. Their results presented in Appendix B.

Results. GPT-4o outperforms GPT-4o-mini on average, while Llama-3.1-8B frequently fails to complete the analysis process and defaults to the `ben` label when integrated into ReAct Agents.

Findings. Two key findings are identified.

Different foundation models are sensitive to different prompting strategies. The results show that GPT-4o-mini and GPT-4o exhibit distinct improvement patterns with CoT and FS examples. This highlights the need to customize prompting strategies based on the selected foundation models. Moreover, larger models do not always outperform smaller models, emphasizing the importance of carefully designing methods that consider the characteristics of specific foundation models.

The execution of ReAct Agents depends on the instruction-following capability of foundation models. Inspection of the outputs reveals that the failures of Llama-3.1-8B in ReAct Agents are due to the model's frequent inability to follow output format requirements. This prevents the agents from linking outputs and inputs across components, thus failing to perform the thought-action-observation iterative framework. This is a known issue with some foundation models, which struggle to follow instructions effectively (Verma et al., 2024), reducing their effectiveness when used in agentic architectures.

5.4 Pairwise Comparison

The failures in pairwise evaluation can be categorized into three types: **pairwise vulnerable**, where both versions are labeled as vulnerable; **pairwise benign**, where both versions are labeled as benign; and **pairwise reversed**, where both versions are mislabeled. We conduct a more detailed pairwise comparison based on these types.

Results. The most prevalent pairwise inaccuracy is *pairwise vulnerable*, ranging from approximately 40% to 95% for LLMs and from around 35% to 50% for ReAct Agents (except those with Llama-3.1-8B). Typically, the occurrence of *pairwise reversed* increases as $pAcc$ improves.

Findings. We identify the following insights.

ReAct Agents are more effective at distinguishing between pairwise target functions. ReAct Agents demonstrate a stronger ability to differentiate between a vulnerable function and its patched benign version. This is because they leverage

the thought-action-observation framework to iteratively and adaptively retrieve additional context, such as callers and callees, which allows them to capture differences between the two versions. A related case study can be found in Appendix C.1.

LLMs and ReAct Agents sometimes fail to identify the causes in the vulnerable version, while tending to over-analyze the benign version after the vulnerability fix. LLMs and ReAct Agents often struggle to pinpoint the root causes (*e.g.,* insufficient input sanitization) of vulnerabilities in the vulnerable version. After the vulnerability is fixed, however, the detection methods tend to over-analyze the patched guards (*e.g.,* newly added sanitization statements) in the benign version, speculating about results and often misidentifying non-issues. This discrepancy arises because LLMs rely on broad, general patterns without the accurate reasoning capability needed for complex contexts. This highlights the need for more fine-grained analysis capabilities, such as reliable constraint solving, which should be integrated with LLMs or enhanced through program analysis techniques. A detailed case study can be found in Appendix C.2.

LLMs and ReAct Agents often exhibit inconsistent analysis patterns when analyzing pairwise target functions. When analyzing pairwise target functions (vulnerable and benign version) LLMs display significant inconsistencies in their evaluation patterns. For some pairs, they may provide thorough analysis for the vulnerable version but neglect important details in the benign version, or vice versa. In other cases, they may exhibit contrasting reasoning or focus on irrelevant aspects, leading to discrepancies in how they interpret the two versions. A clear example of this inconsistency is the significant difference in the average number of tool invocations by the ReAct Agent for the vulnerable and benign versions (*e.g.,* 5.85 vs. 1.79 when using GPT-4o-mini with vanilla prompting). These inconsistencies highlight the challenges LLMs face in handling the complexities of code analysis, where the differences between a vulnerable and benign version can be subtle and require more nuanced evaluation. A detailed case study can be found in Appendix C.1.

6 Conclusion

In this work, we introduced JITVUL, a benchmark for just-in-time (JIT) vulnerability detection that enables a comprehensive, pairwise evaluation of LLMs and LLM-based agents. Our results show that ReAct Agents, leveraging thought-action-observation and interprocedural context, demonstrate better reasoning but require further refinement, particularly in utilizing advanced prompting strategies. Additionally, LLMs and ReAct agents often misinterpret flaws by either overlooking critical issues or over-analyzing benign fixes. These findings highlight the need for improving agentic architectures, prompting techniques, dynamic interprocedural analysis, and robust reasoning models tailored to vulnerability analysis to enhance automated vulnerability detection.

7 Limitations

As the early work benchmarking LLM-based agents for JIT vulnerability detection in code repositories, we acknowledge several limitations. First, the construction of JITVUL may not perfectly trace the *vul-intro* commit due to the complexity of code evolution and function interactions. To mitigate this, we followed existing methodologies (Lomio et al., 2022) and manually inspected selected instances to ensure reliability. Second, while we emphasize pairwise evaluation for JIT detection, the label-balanced dataset may introduce bias in F_1 comparisons. In the future, we plan to incorporate more benign commits to improve the evaluation in F_1 metric. Third, some important statistics, such as the ratio of interprocedural vulnerabilities, are missing. Although we attempted manual annotation, it is labor-intensive and difficult to scale. We plan to leverage commercial annotation platforms to achieve the annotation and provide more fine-grained evaluations in future work.

Acknowledgment

This research/project is supported by the National Research Foundation, Singapore, and the Cyber Security Agency of Singapore under the National Cybersecurity R&D Programme and the CyberSG R&D Programme Office (Award CRPO-GC2-ASTAR-001). Any opinions, findings, conclusions, or recommendations expressed in these materials are those of the author(s) and do not reflect the views of the National Research Foundation, Singapore, the Cyber Security Agency of Singapore, or the CyberSG R&D Programme Office.

References

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.

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*, RAID '23, pages 654–668, New York, NY, USA. Association for Computing Machinery.

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

Xueying Du, Geng Zheng, Kaixin Wang, Jiayi Feng, Wentai Deng, Mingwei Liu, Bihuan Chen, Xin Peng, Tao Ma, and Yiling Lou. 2024. Vul-rag: Enhancing llm-based vulnerability detection via knowledge-level rag. *arXiv preprint arXiv:2406.11147*.

Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. Ac/c++ code vulnerability dataset with code changes and cve summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 508–512.

GNU. 2025. Gnu cflow: analyzing a collection of c source files, charting control flow within the program.

Mete Keltek, Rong Hu, Mohammadreza Fani Sani, and Ziyue Li. 2024. Lsast–enhancing cybersecurity through llm-supported static application security testing. *arXiv preprint arXiv:2409.15735*.

Zhen Li, Ning Wang, Deqing Zou, Yating Li, Ruqian Zhang, Shouhuai Xu, Chao Zhang, and Hai Jin. 2024. On the effectiveness of function-level vulnerability detectors for inter-procedural vulnerabilities.

Francesco Lomio, Emanuele Iannone, Andrea De Lucia, Fabio Palomba, and Valentina Lenarduzzi. 2022. Just-in-time software vulnerability detection: Are we there yet? *Journal of Systems and Software*, 188:111283.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. SELF-REFINE: Iterative Refinement with Self-Feedback.

Mend.io. 2025. Mend.io vulnerability database: The largest open source vulnerability database.

Chao Ni, Liyu Shen, Xiaohu Yang, Yan Zhu, and Shaohua Wang. 2024. MegaVul: A C/C++ Vulnerability Dataset with Comprehensive Code Representations. In *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*, pages 738–742. ISSN: 2574-3864.

NIST. 2025. National vulnerability database: Dashboard.

Ted Ridnik, Dedy Kredo, and Itamar Friedman. Code Generation with AlphaCodium: From Prompt Engineering to Flow Engineering.

Niklas Risse and Marcel Böhme. 2024a. Top score on the wrong exam: On benchmarking in machine learning for vulnerability detection. *arXiv preprint arXiv:2408.12986*.

Niklas Risse and Marcel Böhme. 2024b. Uncovering the limits of machine learning for automatic vulnerability detection. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 4247–4264.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language Agents with Verbal Reinforcement Learning. *arXiv preprint*. ArXiv:2303.11366 [cs].

Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Wei Ma, Lyuye Zhang, Yang Liu, and Yingjiu Li. 2024. LLM4Vuln: A Unified Evaluation Framework for Decoupling and Enhancing LLMs' Vulnerability Reasoning. *arXiv preprint*. ArXiv:2401.16185.

universal ctags. 2025. Universal ctags: A maintained ctags implementation.

Mudit Verma, Siddhant Bhambri, and Subbarao Kambhampati. 2024. On the brittle foundations of react prompting for agentic large language models. *Preprint*, arXiv:2405.13966.

Chong Wang, Jianan Liu, Xin Peng, Yang Liu, and Yiling Lou. 2025. Boosting static resource leak detection via llm-based resource-oriented intention inference. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 668–668. IEEE Computer Society.

Xinchen Wang, Ruida Hu, Cuiyun Gao, Xin-Cheng Wen, Yujia Chen, and Qing Liao. 2024. Reposvul: A repository-level high-quality vulnerability dataset. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, pages 472–483.

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. 2024a. Automatically inspecting thousands of static bug warnings with large language model: How far are we? *ACM Transactions on Knowledge Discovery from Data*, 18(7):1–34.

Xin-Cheng Wen, Xinchen Wang, Yujia Chen, Ruida Hu, David Lo, and Cuiyun Gao. 2024b. Vuleval: Towards repository-level evaluation of software vulnerability detection. *arXiv preprint arXiv:2404.15596*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.

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

Xin Zhou, Duc-Manh Tran, Thanh Le-Cong, Ting Zhang, Ivana Clairine Irsan, Joshua Sumarlin, Bach Le, and David Lo. 2024b. Comparison of static application security testing tools and large language models for repo-level vulnerability detection. *arXiv preprint arXiv:2407.16235*.

Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*, 32.

A Studied Methods

A.1 Prompt Templates in Plain LLM

Figure 3 illustrates the various prompting strategies used with Plain LLM. The *Vanilla Prompt* serves as the base prompt included in all variants, while *FS Examples* and *CoT Instruction* are selectively applied according to the specific strategies. In the template, "{target_function}" acts as a placeholder for the target function to be detected.

You are a security researcher tasked with identifying vulnerabilities in a codebase. You have been given a function to analyze. The function may or may not be vulnerable.

If you think it is vulnerable reply with @@VULNERABLE@@, otherwise reply with @@NOT VULNERABLE@@

If you think the function is vulnerable, please provide the CWE number that you think is most relevant to the vulnerability in the form of @@CWE: <CWE_NUMBER>@@
For example:
@@VULNERABLE@@
@@CWE: CWE-1234@@

Here is the function:
```c
{target_function}
```

Vanilla Prompt

Example Detections:
- Vulnerable Example 1
- Benign Example 1
…
- Vulnerable Example 10
- Benign Example 10

FS Examples

Solve this problem step by step. Carefully break down the reasoning process to arrive at the correct solution. Explain your reasoning at each step before providing the final answer.

CoT Instruction

Figure 3: Prompt templates used with Plain LLM.

A.2 ReAct Agent

Figure 4 illustrates the overall workflow of the ReAct Agent for JIT vulnerability detection. Figure 5 displays the prompt used with the ReAct Agent, implemented with the default LangChain framework. The "{agent_scratchpad}" is a one-time execution memory that holds tool descriptions, along with previous observations and reasoning traces. The "{input}" variable is used for the user prompt and can be further enhanced using prompt augmentation techniques.

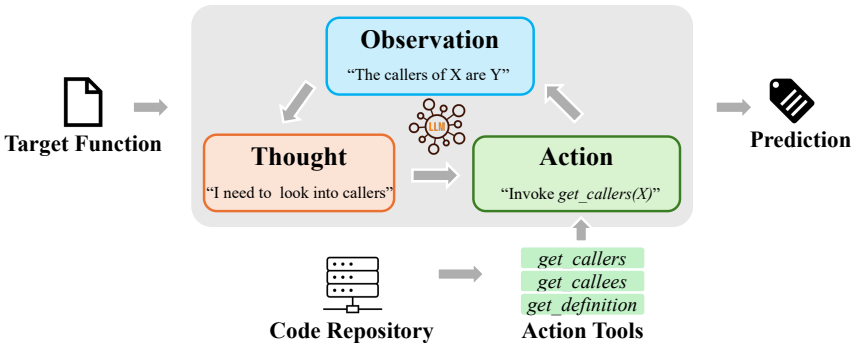

Figure 4: Workflow of ReAct Agent for JIT vulnerability detection.

Answer the following questions as best you can. You have access to the following tools:
- *get_callers:* Get callers for a function, function names are returned
- *get_callees:* Get callees for a function, function names are returned
- *get_definition:* Get definition code of a function based on the function name.

Use the following format:
- Question: the input question you must answer.
- Thought: you should always think about what to do
- Action: the action to take, should be one of *get_callers, get_callees, and get_definition*
  - Action Input: the input to the action
  - Observation: the result of the action
... (this Thought/Action/Action Input/Observation can repeat N times)
- Thought: I now know the final answer
- Final Answer: the final answer to the original input question

Begin!

Question: {input}
Thought:{agent_scratchpad}

Figure 5: Prompt template used with ReAct Agent.

A.3 Few-shot Example

Figure 6 presents a few-shot example based on the webpage for "CWE-787: Out-of-bounds Write". It highlights the key differences between the vulnerable and benign versions.

 # Code

```
char* trimTrailingWhitespace(char *strMessage, int length) {
  char *retMessage;
  char *message = malloc(sizeof(char)*(length+1));
  // copy input string to a temporary string
  char message[length+1];
  int index;
  for (index = 0; index < length; index++) {
    message[index] = strMessage[index];
  }
  message[index] = '\\0';
  // trim trailing whitespace
  int len = index-1;
  while (isspace(message[len])) {
    message[len] = '\\0';
    len--;
  }
  // return string without trailing whitespace
  retMessage = message;
  return retMessage;
}
```

 # Explanation

In the code, a utility function is used to trim trailing whitespace from a character string. The function copies the input string to a local character string and uses a while statement to remove the trailing whitespace by moving backward through the string and overwriting whitespace with a NULL character. However, this function can cause a buffer underwrite if the input character string contains all whitespace. On some systems the while statement will move backwards past the beginning of a character string and will call the `isspace()` function on an address outside of the bounds of the local buffer.

(a) Vulnerable Version

 # Code

```
char* trimTrailingWhitespace(char *strMessage, int length) {
  char *retMessage;
  char *message = malloc(sizeof(char)*(length+1));
  // copy input string to a temporary string
  char message[length+1];
  int index;
  for (index = 0; index < length; index++) {
    message[index] = strMessage[index];
  }
  message[index] = '\\0';
  // trim trailing whitespace
  int len = index-1;
  while (len >= 0 && isspace(message[len])) {
    message[len] = '\\0';
    len--;
  }
  // return string without trailing whitespace
  retMessage = message;
  return retMessage;
}
```

 # Explanation

In the code, a utility function is used to trim trailing whitespace from a character string. The function copies the input string to a local character string and uses a while statement to remove the trailing whitespace by moving backward through the string and overwriting whitespace with a NULL character. This function avoids a buffer underwrite by incorporating the boundary check `len >= 0` in the while loop condition. This ensures that the loop does not move past the beginning of the character string or call the `isspace()` function on an address outside the bounds of the buffer.

(b) Benign Version

Figure 6: A FS example of "CWE-787: Out-of-bounds Write", including both vulnerable version and benign version.

B Results of Llama-3.1 and DeepSeek-Coder-V2

Table 4 shows the results of LLaMA 3.1-8B and DeepSeek-Coder-V2-16B on JITVUL. ReAct Agents using these models exhibit significantly lower performance, with execution frequently failing due to formatting and parsing issues. As a result, the agents often default to the `ben` label. Notably, the results with DeepSeek-Coder suggest that code-specific fine-tuning enhances vulnerability detection performance, particularly in capturing inter-procedural context in Dep-Aug settings. This highlights a promising direction for future work: developing code-specific LLMs optimized for agent-based architectures.

Method	**Llama-3.1**		**DeepSeek-Coder-V2**	
	F_1	$pAcc$	F_1	$pAcc$
Plain LLM				
- vanilla	58.05	0.84	60.04	15.94
- w/ CoT	49.79	10.92	59.74	12.41
- w/ FS	54.48	1.68	48.26	8.30
- w/ CoT+FS	29.55	14.29	50.46	10.93
Dep-Aug LLM				
- vanilla	40.48	15.17	46.41	29.05
- w/ CoT	21.18	8.39	70.61	52.16
- w/ FS	27.37	10.88	59.06	41.72
- w/ CoT+FS	16.46	7.42	82.09	69.24
ReAct Agent				
- vanilla	9.09	4.20	19.40	8.39
- w/ CoT	14.67	3.36	30.00	16.81
- w/ FS	3.28	0.84	1.96	1.00
- w/ CoT+FS	3.28	1.68	31.97	21.39

Table 4: Results of studied methods on JITVUL with Llama3.1-8B and DeepSeek-Coder-V2-16B. The best and second-best results are highlighted.

C Case Study

We provide several examples to illustrate the inputs and outputs of the detection methods for a better understanding of the analysis.

C.1 CVE-2019-15164

Figure 7 illustrates the case study derived from CVE-2019-15164 (details at https://nvd.nist.gov/vuln/detail/CVE-2019-15164), with the left side showing the vulnerable code and the detection methods' responses, and the right side depicting the benign version and its corresponding responses. The vulnerable version of the function `daemon_msg_open_req` is susceptible to a "CWE-918: Server-Side Request Forgery (SSRF)" vulnerability due to the lack of validation for `source` before opening the device, which is read from the network socket. The benign version addresses this vulnerability by adding an if-condition to validate whether `source` is a valid URL, as highlighted in the figure.

Label Predictions. When using Plain LLM with GPT-4o and vanilla prompting, the analyses of both the vulnerable and benign versions focus on buffer operations and misclassify the benign as vulnerable. In contrast, when using the ReAct Agent, the predictions for both versions are correct. The agent is able to retrieve and analyze additional context, such as understanding its caller function `daemon_serviceloop` and surrounding function bodies. This contextual information enables the agent to better comprehend how the `daemon_msg_open_req` function is used within the broader codebase and recognize the risk introduced by the unvalidated URL input. Key points in the analysis process are highlighted to show the improved detection capability provided by the ReAct Agent.

CWE Predictions. However, upon examining the specific vulnerability categories predicted by Plain LLM and ReAct Agent, some fine-grained issues emerge. Plain LLM incorrectly predicts "CWE-120: Buffer Copy without Checking Size of Input" for both the vulnerable and benign versions, which is entirely inaccurate. On the other hand, ReAct Agent predicts "CWE-20: Improper Input Validation" for the vulnerable version. While this is not the correct classification, it is somewhat related to the ground-truth vulnerability of Server-Side Request Forgery (SSRF). The SSRF vulnerability arises from the improper validation of the `source` parameter before opening device, which the ReAct Agent's prediction partially captures, indicating a closer alignment to the actual issue.

Analysis Patterns. When delving into the detailed analysis processes, we observe that the ReAct Agent does not maintain consistent analysis patterns across both versions. For the vulnerable version, the agent focuses on buffer operation and input validation, while for the benign version, it conducts a more comprehensive check. However, in this case, the analysis patterns should be more similar, suggesting that the LLM behind the ReAct Agent lacks sufficient robustness to capture the actual vulnerability characteristics. This indicates a deficiency in its reasoning capabilities for accurate vulnerability reasoning.

C.2 CVE-2019-3877

Figure 8 illustrates the case study derived from CVE-2019-3877 (details at https://nvd.nist.gov/vuln/detail/CVE-2019-3877), with the left side showing the vulnerable code and the detection methods' responses, and the right side depicting the benign version and its corresponding responses. The vulnerable version of the function `am_check_url` is susceptible to a "CWE-601: URL Redirection to Untrusted Site" vulnerability due to the insufficient validation for `url`. The benign version addresses this vulnerability by adding an if-condition to validate whether backslash exists in `url`.

Label Predictions. Plain LLM with GPT-4o and vanilla prompting produces incorrect predictions, often reversing the labels.

Analysis Patterns. A closer look at the analysis process reveals that Plain LLM fails to identify the insufficient validation for the `url` in the vulnerable version, where an assert is highlighted. In contrast, it over-analyzes the validation requirements for the benign version. These lead to incorrect predictions for both versions.

Vulnerable Version

```
static int daemon_msg_open_req(uint8 ver, struct daemon_slpars *pars, uint32
plen, char *source, size_t sourcelen) {
  ...
  nread = sock_recv(pars->sockctrl, source, plen,...);
  ...
  source[nread] = '\0';
  plen -= nread;
  ...
  // Open the selected device
  ...
}
```

CWE-918: Server-Side Request Forgery (SSRF)

Benign Version

```
static int daemon_msg_open_req(uint8 ver, ..., char *source, size_t sourcelen) {
  ...
  nread = sock_recv(pars->sockctrl, source, plen,...);
  ...
  source[nread] = '\0';
  plen -= nread;
  ...
  if (is_url(source)){
    pcap_snprintf(errmsgbuf, PCAP_ERRBUF_SIZE, "Source ...");
    goto error;
  }
  // Open the selected device
  ...
}
```

Plain LLM Response

@@VULNERABLE@@
CWE-120: Buffer Copy without Checking Size of Input

The function daemon_msg_open_req is vulnerable to **a buffer overflow** due to improper handling of the source buffer. The issue arises because the function reads data into source using sock_recv without ensuring that the buffer is large enough to accommodate the incoming data. Although the function includes a check at the beginning to ensure that plen is not greater than sourcelen - 1, this does not fully prevent an overflow when sock_recv writes to source.
…

Plain LLM Response

@@VULNERABLE@@
CWE-120: Buffer Copy without Checking Size of Input

The function daemon_msg_open_req has **a buffer overflow vulnerability** due to improper handling of the source buffer. The issue arises because the function reads data into source using sock_recv without ensuring that the buffer is large enough to store the incoming data. While there is a check to ensure that plen is not greater than sourcelen - 1, this does not fully prevent an overflow when sock_recv writes to source. If plen exceeds sourcelen, the function could still write beyond the buffer's limits, causing a buffer overflow.
…

ReAct Agent Response

…
To determine if daemon_msg_open_req is vulnerable, I need to analyze its behavior and interactions with other functions in the codebase. Specifically, I should check for common vulnerabilities such as buffer overflows, improper input validation, and misuse of functions. Since the function body is already available, I will examine the functions that call daemon_msg_open_req and those that it calls. This will help me understand its execution context and identify potential vulnerabilities.

Action: get_callers
Action Input: "daemon_msg_open_req"
Observation: {'callers': ['daemon_serviceloop']}

Thought: The function daemon_msg_open_req is called by the function daemon_serviceloop. …

Action: get_definition
Action Input: "daemon_serviceloop"
Observation: <-Function Body->

Thought: The daemon_msg_open_req function is called within the daemon_serviceloop function. …
…
To further analyze potential vulnerabilities, I will check the functions that daemon_msg_open_req calls. This will help me identify if any of those functions might introduce vulnerabilities.

Action: get_callees
Action Input: "daemon_msg_open_req"
Observation: {'callees': ['pcap_snprintf', 'sock_recv', 'rpcapd_log', 'pcap_open_live', 'sock_bufferize', 'rpcap_createhdr', 'memset', 'htonl', 'pcap_datalink', 'pcap_close', 'sock_send', 'rpcap_senderror', 'rpcapd_discard']}

Thought: The daemon_msg_open_req function calls several functions, including pcap_snprintf, .

pcap_snprintf: …
…
rpcapd_discard: …

The main concern in daemon_msg_open_req is the handling of network data and ensuring that all inputs are properly validated. The function does perform some bounds checking, but it is crucial to ensure that all external data is sanitized and validated to prevent vulnerabilities such as buffer overflows or improper resource access.

Given the analysis, the function appears to have potential vulnerabilities related to improper input validation and buffer management, particularly with network data. Therefore, I would classify this function as potentially vulnerable.

Final Answer: @@VULNERABLE@@
CWE-20: Improper Input Validation

ReAct Agent Response

The function daemon_msg_open_req is given, and I need to determine if it is vulnerable. I should first analyze the function body to identify any potential vulnerabilities. Then, I can use the tools to get more context by examining the callers and callees of this function.

Let's start by examining the function body for potential vulnerabilities.

Buffer Overflow Risk: …
Source Length Check: …
Socket Operations: …
Device Opening: …
Error Handling: …

Now, let's use the tools to get more context by examining the callers and callees of this function.

Action: get_callers
Action Input: daemon_msg_open_req
Observation: {'callers': ['daemon_serviceloop']}

Thought: The function daemon_msg_open_req is called by the function daemon_serviceloop. …

Action: get_definition
Action Input: daemon_serviceloop
Observation: <-function body->

Thought: The daemon_msg_open_req function is called by daemon_serviceloop. By examining the daemon_serviceloop function, ...

The daemon_serviceloop function appears to perform several checks and handles different message types, including authentication and error handling. However, the key point of concern is how the source string is handled in daemon_msg_open_req.

Potential Vulnerabilities:
- Buffer Overflow: ... This is a good practice.
- URL Injection: The function checks if source is a URL using is_url(source) and rejects it if true. This is a security measure to prevent URL injection.
- Error Handling: …, which helps prevent buffer overflow.
- Use of External Functions: ...

Conclusion: The function daemon_msg_open_req seems to handle input validation and error handling appropriately. The primary concerns would be the security of the external functions it calls and ensuring that the buffers are always managed correctly.

Given the information available, the function does not exhibit any obvious vulnerabilities directly in its implementation. Therefore, based on the current analysis:

Final Answer: @@NOT VULNERABLE@@

Figure 7: Case study for CVE-2019-15164.

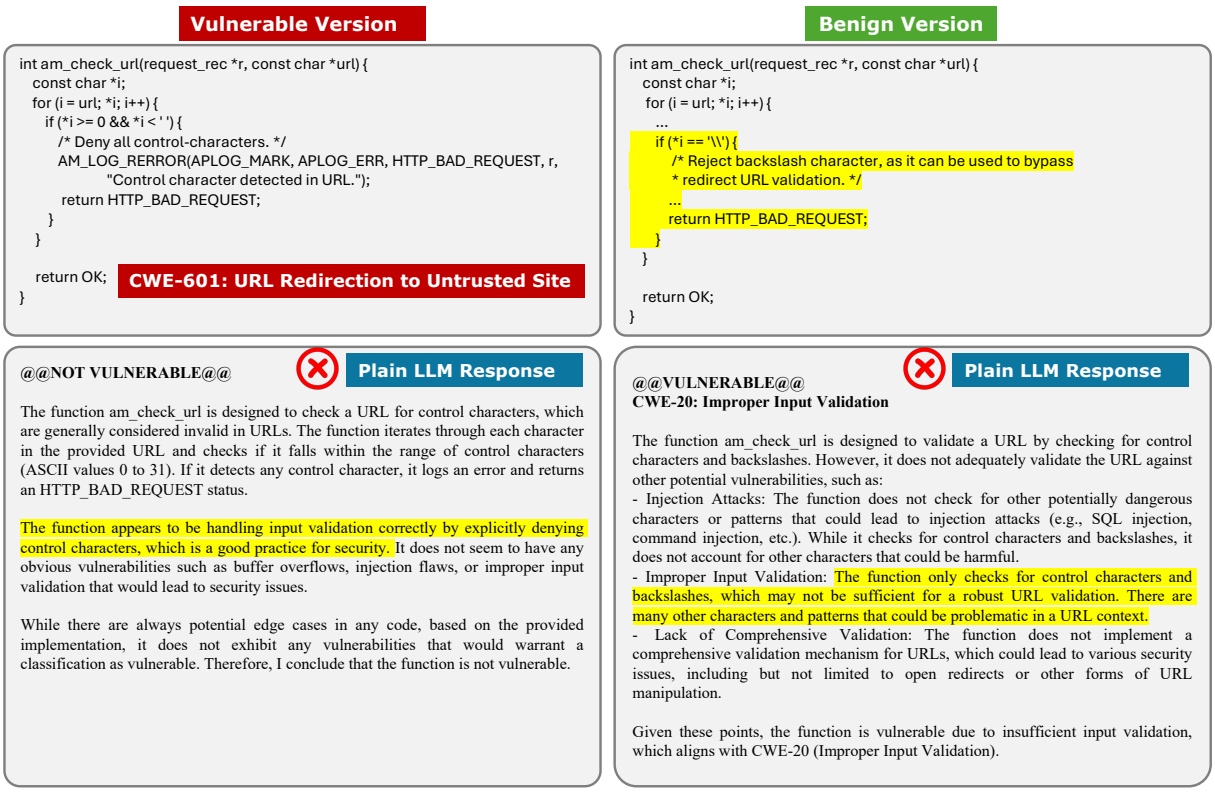

Figure 8: Case study for CVE-2019-3877.