Benchmarking Explainable Vulnerability Detection with Calling Contexts

Anonymous ACL submission

Abstract

Large language models (LLMs) are increasingly applied to vulnerability detection in real-world software systems, where the task goes beyond binary classification and requires calling context exploration over large code repositories. However, existing benchmarks primarily formulate the vulnerability detection as a classification problem without calling contexts and provide no formalized explanations, making them inadequate for evaluating LLM-driven vulnerability detectors upon repositories. To fill the gap, we introduce ManBo, a benchmark designed to assess the capability of LLM-driven vulnerability detectors in the real-world scenarios. Specifically, ManBo provides human-verified vulnerability explanations together with calling contexts, enabling principled evaluation of the ability to both identify vulnerabilities and explain root causes. It contains 191 vulnerable instances and 158 safe instances collected from 74 high-profile C/C++ projects, covering three critical types of memory corruption. We evaluate a broad spectrum of LLM-driven detectors on ManBo, and find that effective calling context retrieval is essential for repository-level vulnerability detection, exposing a fundamental gap between the capabilities of existing LLM-driven detectors and real-world requirements.

1 Introduction

Large language models (LLMs) have demonstrated remarkable capabilities and are now widely applied across many domains (Achiam et al., 2023; Touvron et al., 2023; Liu et al., 2024). In software engineering, developers increasingly rely on LLMs to support tasks such as code generation and vulnerability detection (Xu et al., 2024; Chen, 2021). Among these tasks, vulnerability detection is the cornerstone of reliable software development in the AI era, attracting substantial research interest and industry attention (Cursor, 2025; CodeRabbit, 2025; Wang et al., 2024a; Guo et al., 2025b). With expanded context windows and improved reasoning abilities, LLMs are increasingly being deployed beyond controlled experimental settings and into real-world software projects, where they are used to uncover previously unknown vulnerabilities (Guo et al., 2025b,a; Zheng et al., 2025).

To evaluate the capability of LLM-driven vulnerability detection in real-world settings, researchers have proposed various benchmarks over the past few years, including Devign (Zhou et al., 2019), BigVul (Fan et al., 2020), PrimeVul (Ding et al., 2024), ReposVul (Wang et al., 2024b), and CrossVul (Nikitopoulos et al., 2021). These benchmarks collect vulnerable code and corresponding fixed versions from real-world projects and ask detectors to determine whether a given code snippet contains a vulnerability. While effective for evaluating classification accuracy, these benchmarks suffer from two fundamental limitations.

Limitation I: Lack of formalized explanations. Existing vulnerability detection benchmarks predominantly formulate vulnerability detection as a classification problem (Zhou et al., 2019; Steenhoek et al., 2024). As code snippets in these benchmarks are annotated only with binary labels, namely *Vulnerable* or *Safe*, they do not provide formalized explanations of how vulnerabilities arise within the repository. Such simple formulation is insufficient for real-world vulnerability detection, where detailed explanations, such as how vulnerabilities originate, evolve, and are ultimately triggered across the codebase, are essential for effective diagnosis and remediation (Do and Bodden, 2020). As illustrated in Figure 1 (a), the Use-After-Free (UAF) vulnerability spans three functions and involves non-trivial inter-procedural data dependencies. Without a formalized explanation, human auditors must invest significant effort to reconstruct the vulnerable path and identify its root cause. Moreover, the binary classification formulation provides weak evaluation criteria: binary labels alone cannot determine

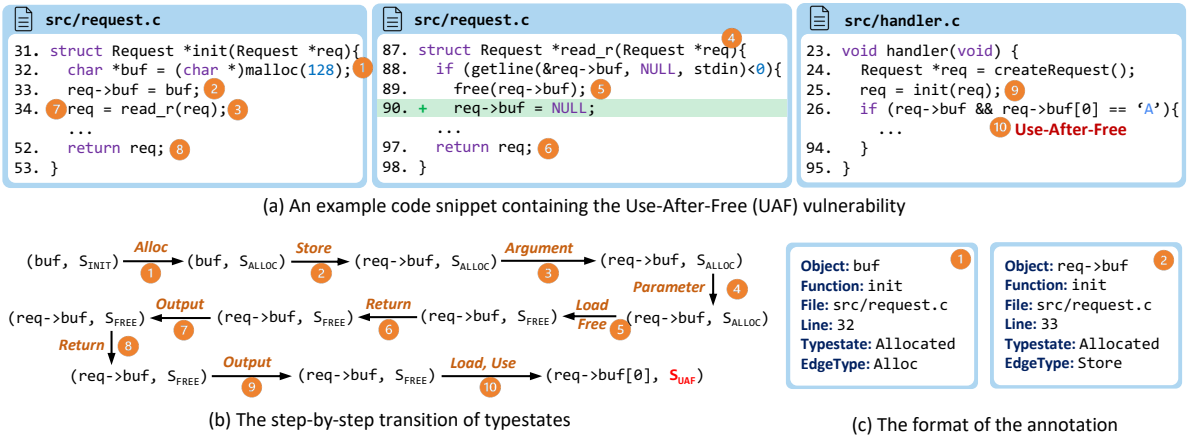


Figure 1: An example of Use-After-Free (UAF). The assignment req->buf = NULL is added in the fixing patch.

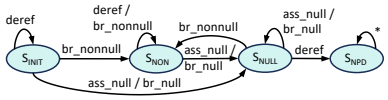
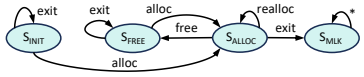
whether a detector reaches a correct prediction via valid reasoning paths (see Appendix E), which can consequently lead to false positives in evaluation.

Limitation II: Insufficient calling contexts. Existing benchmarks mainly define calling contexts based on patch commits (Fan et al., 2020; Ding et al., 2024; Wang et al., 2024b). During dataset construction, they locate a patch commit and treat the functions modified in the commit as related to the vulnerability, labeling the pre-patch version as vulnerable and the post-patch version as safe. Although the method of benchmarking is scalable, it cannot guarantee that the collected calling contexts are sufficient for reliable detection, as patch commits often modify only a subset of the functions involved in the complete vulnerable data flow. As shown in Figure 1 (a), the patch commit for the Use-After-Free (UAF) vulnerability only adds `req->buf = NULL` in the function `read_r`. Under the existing benchmarking strategy, the resulting calling context only includes the function `read_r`, while omitting other functions such as `init`, which allocates the memory, and `handler`, which later uses the freed memory. Obviously, evaluating LLM-driven detectors within such incomplete calling contexts leads to an ill-posed problem setting and can produce misleading conclusions (Risse et al., 2025).

Our Benchmark. To address these limitations, we introduce ManBo, a real-world vulnerability detection benchmark that provides both formalized explanations and sufficient calling contexts for evaluating LLM-driven vulnerability detectors. For rigorous formulation, we focus on *typestate vulnerabilities*, which cover a broad class of vulnerabilities involving resource and state mismanagement (Fink et al., 2008), and can be flexibly specified by users through customized typestate definitions. Specifically, we consider three representative vulnerability types, namely Null Pointer Dereference (NPD), Memory Leak (MLK), and Use-After-Free (UAF), which are the most common memory corruptions in real-world software systems (MITRE, 2025). Conceptually, we introduce the notion of *typestate slice* to uniformly depict the formalized explanations of various typestate vulnerabilities (see Section 3.2). Since existing program analysis techniques cannot reliably recover the program statements relevant to the target vulnerability upon the source-code level (See Appendix A), we design a semi-automated pipeline for benchmark construction, consisting of issue collection, semantic filtering, and manual annotation, which ensures the high quality of both the calling contexts and explanations. In total, ManBo consists of 191 vulnerable instances and 158 safe instances collected from 74 real-world C/C++ projects, which have an average of 14.8K GitHub stars and 340K lines of code.

We further propose two novel evaluation tasks, *Explainable Vulnerability Detection* and *Calling Context Retrieval*, to assess the capabilities of state-of-the-art LLM-driven detectors upon our contructed benchmark ManBo. Our evaluation shows that while LLMs achieve strong performance in vulnerability detection, they exhibit notable limitations in generating accurate vulnerability explanations. In particular, models may introduce hallucinations in the generated typestate slices, even when the predicted vulnerability label is correct. Moreover, our results demonstrate that the availability of sufficient calling context is a critical factor influencing vulnerability detection performance. However, existing LLM-based agents struggle to precisely retrieve the relevant calling contexts, achieving at best a precision of 33.64% and a recall of 50.74%, which substantially constrains their effectiveness in the end-to-end explainable vulnerability detection.

Table 1: The FSMs of Null Pointer Dereference (NPD), Memory Leak (MLK), and Use-After-Free (UAF).

$\mathbf{FSM_{NPD}}$ $\langle \Sigma, \mathbb{S}, S_{\text{INIT}}, \delta, S_{\text{NPD}} \rangle$	$\mathbf{FSM_{MLK}}$ $\langle \Sigma, \mathbb{S}, S_{\text{INIT}}, \delta, S_{\text{MLK}} \rangle$	$\mathbf{FSM_{UAF}}$ $\langle \Sigma, \mathbb{S}, S_{\text{INIT}}, \delta, S_{\text{UAF}} \rangle$
$\mathbb{S} = \{S_{\text{INIT}}, S_{\text{NON}}, S_{N}, S_{\text{NPD}}\}$ $S_{\textbf{NON}}$: The variable is non-NULL. $S_{\textbf{NULL}}$: The variable is NULL.	$\mathbb{S} = \{S_{\text{INIT}}, S_{\text{ALLOC}}, S_{\text{FREE}}, S_{\text{MLK}}\}$ S_{ALLOC}: A heap object is allocated and not freed. S_{FREE}: A heap object is freed.	$\mathbb{S} = \{S_{\text{INIT}}, S_{\text{ALLOC}}, S_{\text{FREE}}, S_{\text{UAF}}\}$ S_{ALLOC}: A heap object is allocated. S_{FREE}: The heap object is freed.
$\Sigma = \{ass_null, br_null, br_nonnull, deref\}$ ***ass_null***: Assign NULL to a pointer. ***br_null***: Execute a branch where the pointer is NULL. ***br_nonnull***: Execute a branch where the pointer is non-NULL. ***deref***: Dereference a pointer.	$\Sigma = \{alloc, realloc, free, exit\}$ ***alloc***: Allocate a heap memory object. ***recalloc***: Reallocate a heap memory object. ***free***: Free a heap object. ***exit***: Return from main function.	$\Sigma = \{alloc, realloc, free, use, ass_null\}$ ***alloc***: Allocate a heap memory object. ***recalloc***: Reallocate a heap memory object. ***free***: Free a heap object. ***use***: use a heap object. ***ass_null***: Assign NULL to the memory pointer.
		

2 Related Work

Recent studies have applied LLMs to real-world vulnerability detection tasks (Guo et al., 2025b; Li et al., 2025; Zheng et al., 2025; Guo et al., 2025a). As these techniques are increasingly evaluated on real-world codebases, the quality of benchmarks becomes critical for assessing their effectiveness and reliability. However, existing benchmarks for real-world vulnerability detection often lack sufficient context information and formalized explanations, which limits their ability to support such evaluations (Risse et al., 2025). Several widely used datasets, such as BigVul (Fan et al., 2020) and Devign (Zhou et al., 2019), construct vulnerability instances from patch commits. Although this approach enables large-scale dataset construction, the resulting instances typically provide incomplete calling contexts and only binary *Vulnerable* / *Safe* labels, without explicit explanations of root causes. Subsequent benchmarks, including PrimeVul (Ding et al., 2024), CrossVul (Nikitopoulos et al., 2021), and DiverseVul (Chen et al., 2023), build upon BigVul and improve data quality through additional filtering and cleaning, but they largely inherit the same fundamental limitations. Defects4J (Just et al., 2014) adopts a different construction methodology by incorporating human verification, ensuring that each vulnerability is executable and can be triggered by at least one test case. Nevertheless, Defects4J still does not provide formalized explanations that characterize the root cause or propagation of vulnerabilities, making it difficult to systematically evaluate a model's semantic reasoning ability.

To better evaluate LLM-driven vulnerability detectors, we focus on *typestate vulnerabilities*, a broad class that subsumes many common real-world vulnerability types (Fink et al., 2008; Li et al., 2022) and enables uniform, formalized explanations with necessary calling contexts. Additional preliminaries are introduced in Section 3.

3 Preliminaries

This section introduces the notion of *typestate vulnerabilities* and the formal model used to characterize them, followed by the definition of *typestate slices* as their formalized explanations.

3.1 Typestate Vulnerabilities

A *typestate* describes the abstract state of a program entity and how this state evolves as program operations are applied to it (Strom and Yemini, 1986). A *typestate vulnerability* occurs when an operation is executed in a state where it is not semantically valid, thereby violating the intended usage protocol of the entity. Typestate vulnerabilities constitute a broad class of real-world defects, encompassing various memory corruptions (e.g., Null Pointer Dereference (NPD), Memory Leak (MLK), and Use-after-Free (UAF)) as well as various resource misuse issues (e.g., file descriptor leaks).

To detect typestate vulnerabilities, prior work models typestate evolution using finite-state machines (FSMs) (Fink et al., 2008; Li et al., 2022). An FSM formalizes relevant typestates and depicts how they are updated as the program executes. Specifically, an FSM abstracts program execution as transitions over a finite set of states. Formally, a typestate vulnerability is defined by an FSM

$$\text{FSM}_{\text{type}} = \langle \Sigma, \mathbb{S}, S_{\text{INIT}}, \delta, S_{\text{type}} \rangle.$$

Here, Σ is the set of operations that may affect the typestate, namely *typestate-transition operations*. $\mathbb{S}$ is the set of typestates and S_{INIT} is the initial state. The transition function $\delta : \mathbb{S} \times \Sigma \rightarrow \mathbb{S}$ defines how typestates are updated in response to typestate-transition operations. S_{type} denotes an error typestate whose reachability indicates a potential vulnerability.

Table 1 illustrates the FSMs formulating NPD, MLK, and UAF. The states S_{NPD}, S_{MLK}, and S_{UAF}

denote the error states for the three vulnerability types, respectively. If program execution reaches one of these error states, the corresponding typestate vulnerability is reported. The error state has a self-loop and behaves as a trap state.

3.2 Typestate Slices

Detecting typestate vulnerabilities requires reasoning about both pointer manipulations and inter-procedural value flows. Specifically, pointer load and store operations enable flexible propagation of program values through memory, while passing values as function arguments and return values allows their typestates to evolve across different functions. To formalize how a typestate vulnerability arises in a program, we introduce the notion of a *typestate slice*, which captures the essential reasoning steps leading to the vulnerability. Formally, it is defined as an ordered sequence of program statements

$$\mathcal{S} = \langle s_1, \ldots, s_n \rangle$$

where each statement s_i belongs to one of the following three categories: (1) *typestate-transition operations* defined in the corresponding FSM, which trigger typestate transitions; (2) *pointer load and store operations*, which propagate vulnerability-related values through memory; and (3) *inter-procedural value-passing operations*, including function argument passing and return value propagation across call boundaries. Particularly, the first statement s_1 and the last statement s_n are the typestate-transition operations that change the typestate from the initial state S_{INIT} to a new typestate and update the typestate into an error typestate S_{type}, respectively.

Figure 1 (b) shows an example of typestate slice along with the updated typestates. To simplify the formulation, we only maintain the relevant statements and do not explicitly store typestates in the typestate slice. The FSM formulating the UAF is shown in the last column in Table 1. In the function `init`, a memory object `buf` is *allocated* via `malloc` at line 32 and subsequently *stored* into `req->buf` with the typestate S_{ALLOC}. The pointer `req` is then passed as the first *argument* to `read_r`. Later, the object is *loaded* and *freed* at line 89 in `read_r` and *used* at line 26 in `handler`, resulting in a UAF vulnerability.

Notably, typestate slices precisely capture how the typestate vulnerabilities arise by modeling the sequences of operations upon the concerned program values, making them suitable for evaluating the reasoning chains of the LLM-driven vulnerability detectors. With the inter-procedural propagation information, a typestate slice maintain the functions relevant to a given vulnerability as the calling context. Based on this essential concept, we will construct the benchmark and annotate the formalized explanation for a typestate vulnerability.

4 Our Benchmark: ManBo

In this section, we introduce a high-quality real-world vulnerability detection benchmark that provides both formalized vulnerability explanations and sufficient calling context. Unlike prior benchmarks that primarily evaluate the binary classification performance of LLM-driven vulnerability detectors, our benchmark assesses a broader set of capabilities essential for real-world vulnerability detection, including calling context retrieval and explainable vulnerability detection.

4.1 Benchmark Construction

We first introduce the construction of our benchmark. Since existing program analysis techniques cannot reliably recover complete typestate slices (see Appendix A), we develop a semi-automated pipeline that consists of three stages: issue collection, semantic filtering, and manual annotation.

4.1.1 Issue Collection

We begin by crawling real-world issues from GitHub repositories and selecting projects using two criteria: (1) large scale, with the project size exceeding 100K LoC, and (2) well-maintained, with more than 500 stars in GitHub. In total, we collect 75 projects with an average of 340K lines of code and 14K stars.

For each issue, we parse the description and apply keyword matching to identify those related to the three target vulnerability types. For each potentially relevant issue, we further retrieve supplementary information, including comments, vulnerable commit (i.e., the commit that contains the vulnerability), and reference commits. For the vulnerable commit, we first attempt to extract the commit identifier directly from the issue description. When this information is unavailable, we approximate the referenced version by selecting the commit immediately preceding the issue creation time. This heuristic increases the likelihood that the extracted code matches the version discussed in the report. Reference commits include all commits that mention or link to the issue, many of which contain potential patches.

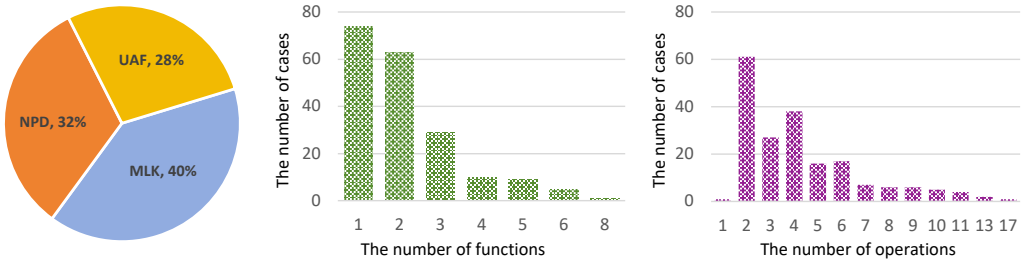


Figure 2: The detailed statistics of vulnerable cases in our benchmark ManBo.

4.1.2 Semantic Filtering

Next, we employ a lightweight LLM agent to analyze each issue together with its supplementary information. The agent performs semantic filtering by retaining only issues that satisfy three constraints: (1) the issue has been confirmed or fixed by the developers; (2) it explicitly describes the vulnerable path; and (3) it contains function names or file paths that correspond to the vulnerable path. An issue is retained only when all three constraints are satisfied. It is important to note that the goal of this semantic filtering is not to guarantee soundness, but rather to reduce manual annotation effort by prioritizing issues that are more likely to be high quality.

After this stage, we obtain 384 real-world issues. These issues contain project identifiers, vulnerability type, vulnerable commit, and fix commits, but they still lack complete typestate slices and may contain low-quality or ambiguous cases, which must be manually filtered and annotated.

4.1.3 Manual Annotation

In the annotation stage, we label the explanation of each vulnerability in the form of typestate slices. As stated in Section 3.2, a typestate slice begins with the initialization of the object and follows its forward propagation along the execution path. Whenever a typestate transition, an inter-procedural propagation, or a pointer load or store operation is encountered, we record the corresponding function name, file path, line number, and the resulting typestate of the object, as illustrated in Figure 1 (c).

In addition, for pointer load and store operations, we annotate the corresponding memory location (as shown by the second annotation in Figure 1(c)), and for inter-procedural propagation operations, we record the indices of the associated arguments and parameters. These annotations enable precise tracking of data flow across function boundaries. This typestate slice constitutes a minimal sequence of operations that fully characterizes the vulnerability and provides line-level explanation, thereby supporting precisely evaluating vulnerability detection.

Wrapper Functions. During the manual annotation, we only consider standard library functions when identifying typestate transition operations. However, many real-world projects encapsulate standard malloc or free primitives within customized helper or wrapper functions. Expanding all propagation steps inside these wrappers would significantly increase the depth of calling contexts. Following common practice in program analysis (Lyu et al., 2022), when encountering a wrapper function, we treat the invocations of wrapper functions as explicit typestate-transition operations instead of inlining its internal propagations. This strategy yields a minimal and concise typestate slice without sacrificing semantic soundness.

All issues are independently annotated by two authors and, in cases of disagreement, reviewed and resolved by a third, more senior author. All annotators have more than five years of experience in program analysis. To avoid ambiguity, we exclude cases in which multiple vulnerable execution paths converge at the same program location and ensure that each task instance contains exactly one typestate slice as the ground truth.

4.2 Dataset Details

By applying the benchmarking methodology stated in Section 4.1, we obtain a total of 191 vulnerable cases, comprising 62 NPD cases, 76 MLK cases, and 53 UAF cases, accounting for 32%, 40%, and 28% of the vulnerable cases, respectively. Each case is annotated with a human-verified typestate slice as the formalized explanation. As illustrated in Figure 2, 38.74% (74/191) of the cases are intra-procedural, involving only a single relevant function, while the remaining cases are inter-procedural, spanning up to 8 functions and 17 operations along the annotated typestate slice. Among the 191 vulnerable cases, 158 have associated fix commits and can be paired with their corresponding fixed versions as safe cases, resulting in a final benchmark of 349 instances with a well-balanced composition of vulnerable and safe cases.

4.3 Task Design

Our benchmark is designed to evaluate the performance of LLM-driven detectors in realistic vulnerability detection scenarios. Specifically, we introduce two complementary tasks, namely *explainable vulnerability detection* and *calling context retrieval*, which are described in detail below.

4.3.1 Explainable Vulnerability Detection

This task evaluates the ability of LLMs to detect vulnerabilities via prompt engineering. Specifically, we supply the LLMs with a *calling context* that contains all relevant functions introducing a vulnerability at a specific program location. To ensure that the LLMs follow the prescribed format of the typestate slice shown in Section 3.2, we explicitly specify the format in the prompt and provide three few-shot examples for each vulnerability type.

For evaluation, we assess both the performance of the vulnerability identification and the quality of the generated typestate slice with the groundtruth. For the vulnerability identification, we consider both vulnerable and safe cases and evaluate whether the predicted label (i.e., Vulnerable and Safe) matches the ground-truth label, which yield the statistics of the precision, recall, and F1 score. To quantify the quality of the generated typestate slices, we only consider vulnerable instances and evaluate the consistency between the generated typestate slices and the groundtruth. Specifically, we compute the *typestate-slice coverage* that measures the fraction of the operations in the groundtruth that are covered by the generated typestate slice. More details are provided in Appendix B.

4.3.2 Calling Context Retrieval

In Section 4.3.1, we provide the calling contexts of a vulnerability as input to the explainable vulnerability detection task. However, it remains an open problem that how to automatically collect such calling contexts in a way that effectively facilitates vulnerability detection. Hence, we introduce the task of calling context retrieval, in which we evaluate state-of-the-art LLM-based coding agents on their ability to retrieve relevant functions that serve as calling contexts. In this task, an agent is provided with only a program location and the corresponding function. The agent is required to retrieve all relevant functions needed for determining whether a typestate bug can arise at the specified program location or not.

Basically, the goal of calling context retrieval is to retrieve all relevant functions while minimizing the inclusion of irrelevant ones. Hence, we measure the precision and recall of the calling context retrieval by comparing the retrieved functions with the ones appearing in the labeled typestate slices. To quantify the efficiency, we measure the financial costs and execution time of each agent.

5 Evaluation

We organize our evaluation around four research questions.

- **RQ1:** How do LLMs perform in the explainable vulnerability detection with calling contexts?
- **RQ2:** How do LLMs perform in the explainable vulnerability detection without calling contexts?
- **RQ3:** How do existing agents perform in the calling context retrieval?
- **RQ4:** How do existing agents perform in the end-to-end explainable vulnerability detection?

Specifically, RQ1 and RQ2 target the explainable vulnerability detection task within a given scope of code, which is a primitive task that does not require interaction with the program environment. RQ3 and RQ4 target and require the calling context retrieval, respectively, which involves the interaction with the codebase to retrieve relevant functions. Hence, we use LLMs to investigate RQ1 and RQ2, and an LLM-based agent to investigate RQ3 and RQ4.

5.1 RQ1: Detection with Calling Contexts

We evaluate eight LLMs on the explainable vulnerability detection task. The details of LLM settings are provided in Appendix C. For each task instance, we provide the LLMs with a specific program location that may trigger a vulnerability, together with all relevant functions in the calling context, and ask the LLMs to determine whether the vulnerability exists or not and also generate a typestate slice as the formalized explanation of the vulnerability. The detailed task design and evaluation metrics are described in Section 4.3.1.

As shown in Table 2, all evaluated models achieve relatively strong performance on vulnerability identification, with an average F1 score of 0.70. However, the average coverage of the generated typestate slices is only 0.48. This discrepancy indicates that, although models can often predict the correct label, they frequently fail to recover the correct typestate slice, which represents the actual root cause and is critical in real-world vulnerability detection

Table 2: The performance of the LLMs in the explainable vulnerability detection. **P**, **R**, and **F1** indicate precision, recall, and F1 score, respectively. **Cov** indicates typestate-slice coverage. **Time** and **Cost** indicate the total time and financial cost, respectively.

Model	P (%)	R (%)	F1	Cov	Time (s)	Cost ($)
GPT 5	75.54	71.43	0.73	0.51	12,587	1.49
Gemini 2.5 Pro	70.52	82.99	**0.76**	**0.58**	17,382	2.90
Claude Sonnet 4.5	62.07	73.47	0.67	0.49	14,083	5.07
GPT 5 mini	68.31	65.99	0.67	0.49	9,716	0.38
Gemini 2.5 Flash	67.74	71.43	0.70	0.53	17,229	0.70
Claude Haiku 4.5	74.34	76.87	**0.76**	0.51	15,184	1.69
Qwen3 Coder 30B	74.74	48.30	0.59	0.30	6,735	N/A
GPT OSS 120B	76.42	63.95	0.70	0.45	5,508	N/A

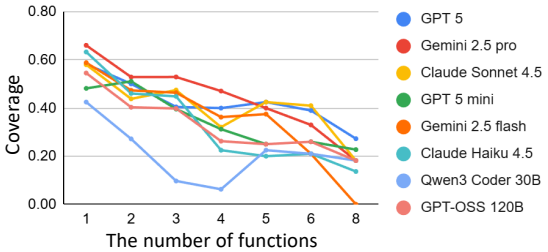


Figure 3: The typestate-slice coverage under increasing inter-procedural complexity.

scenarios. Without a correct explanation, developers cannot reliably locate and fix the vulnerability, even when the model produces a correct prediction. Moreover, models may generate typestate slices that correspond to safe execution paths while still predicting the correct vulnerability label, highlighting a gap between existing evaluation metrics and the requirements of realistic vulnerability detection. The detailed case study is provided in Appendix E.

Figure 3 further illustrates typestate slices coverage under increasing inter-procedural complexity. As the number of functions involved in a typestate slice increases, the coverage scores of all models decline, highlighting that reasoning over inter-procedural data dependencies remains a major challenge for existing models.

> **Finding 1:** LLMs achieve promising performance in vulnerability identification but often fail to offer high-quality explanations.

5.2 RQ2: Detection without Calling Contexts

To assess the effect of calling context on vulnerability detection, we introduce a controlled experiment in which the model is provided only with the vulnerability triggering program location without calling contexts. The evaluation metrics follow RQ1.

As shown in Table 3, removing calling contexts leads to substantial degradation in both F1 score

Table 3: Comparison of explainable vulnerability detection performance with and without calling context.

Model	With Context		Without Context	
	F1	Cov	F1	Cov
GPT 5	0.73	0.51	0.49	0.28
Gemini 2.5 Pro	0.76	0.58	0.64	0.32
Claude Sonnet 4.5	0.67	0.49	0.51	0.31
GPT 5 mini	0.67	0.49	0.39	0.23
Gemini 2.5 Flash	0.70	0.53	0.51	0.26
Claude Haiku 4.5	0.76	0.51	0.61	0.29
Qwen3 Coder 30B	0.59	0.30	0.48	0.14
GPT OSS 120B	0.70	0.45	0.51	0.16
Average	**0.70**	**0.48**	**0.52**	**0.25**

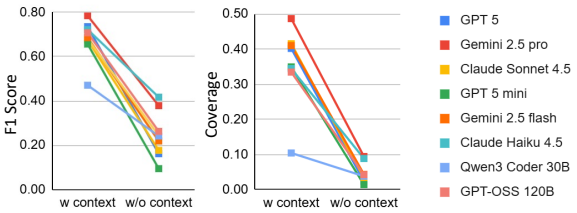


Figure 4: The effect of calling context on explainable vulnerability detection in inter-procedural cases.

and typestate-slice coverage. Specifically, the average F1 score drops from 0.70 to 0.52, while the average coverage decreases from 0.48 to 0.25. The performance gap becomes more pronounced in inter-procedural cases. As illustrated in Figure 4, the average F1 score decreases sharply from 0.68 to 0.24, while the average coverage score drops from 0.36 to 0.05. This performance degradation aligns with the intuition: without access to sufficient calling context, it is difficult to draw a precise conclusion about whether a vulnerability exists. Although models may attempt to infer missing context based on prior knowledge, such inferences are highly uncertain and prone to hallucination.

> **Finding 2:** Calling context is indispensable for effective vulnerability detection.

5.3 RQ3: Calling Context Retrieval

To evaluate the calling context retrieval capability of LLM-based coding agents, we select OpenHands (Wang et al., 2024c), a widely used code agent in practice, and RepoAudit (Guo et al., 2025b), a state-of-the-art vulnerability detection agent, as baselines. Detailed descriptions of these agents are provided in Appendix D. Each agent is provided with the vulnerability triggering program location and the corresponding function, and is tasked with retrieving the calling context required for vulnerability detection. The detailed task design and evaluation metrics are described in Section 4.3.2.

Table 4: Performance of different agents in the calling context retrieval task. **Time** and **Cost** indicate the total time and financial cost, respectively.

Agent	Model	P (%)	R (%)	Time (s)	Cost ($)
OpenHands	GPT 5 mini	**33.64**	**50.74**	24,142	4.95
	Gemini 2.5 Flash	11.77	25.55	11,333	3.79
	GPT OSS 120B	12.95	16.32	21,540	N/A
RepoAudit	GPT 5 mini	**28.78**	**48.56**	14,713	0.91
	Gemini 2.5 Flash	25.93	40.94	13,289	0.92
	GPT OSS 120B	17.61	30.23	7,487	N/A

As shown in Table 4, OpenHands powered by GPT 5 mini achieves the best overall performance, with a precision of 33.64% and a recall of 53.50%. Even in this best case, the agent retrieves only 53.50% of the relevant calling contexts, underscoring the challenge of precise calling context retrieval for LLM-based coding agents. Moreover, because OpenHands relies heavily on the LLM to perform multiple subtasks, it exhibits substantial performance variance across different models, with GPT 5 mini outperforming Gemini 2.5 Flash by more than 2 times and GPT OSS 120B by more than 3 times. Although RepoAudit demonstrates more stable performance across models, it still retrieves fewer than half of the relevant calling contexts.

In terms of efficiency, OpenHands takes 100 seconds on average to analyze a single program location at a cost of $0.03, whereas RepoAudit requires 62 seconds with a cost of $0.005. Given that a real-world project may contain thousands of program locations to analyze, the resulting time and financial overhead can become a significant efficiency bottleneck for LLM-based coding agents. Compared to RepoAudit, OpenHands incurs higher cost because it relies on LLM-generated grep-style searches and must process a large volume of returned results. In contrast, RepoAudit selectively retrieves only call-graph-adjacent functions, reducing the amount of context passed to the LLM and thus lowering both time and cost.

> **Finding 3:** Precise and efficient calling context retrieval remains a significant challenge for LLM-based coding agents.

5.4 RQ4: End-to-End Detection

We evaluate the end-to-end vulnerability detection capability of existing LLM-based coding agents. The input setting remains the same as that of RQ3. Each agent is required to first retrieve relevant calling context from the codebase and then perform explainable vulnerability detection.

Table 5: Vulnerability detection and explanation performance of different agents and backbone models.

Agent	Model	P (%)	R (%)	F1	Cov
OpenHands	GPT 5 mini	71.30	61.59	**0.66**	**0.33**
	Gemini 2.5 Flash	72.30	58.06	0.64	0.31
	GPT OSS 120B	61.39	53.52	0.57	0.19
RepoAudit	GPT 5 mini	71.43	55.54	0.62	0.37
	Gemini 2.5 Flash	73.10	53.09	**0.62**	**0.40**
	GPT OSS 120B	75.35	48.47	0.59	0.29

The results are summarized in Table 5. On average, OpenHands achieves an F1 score of 0.63 with a typestate-slice coverage of 0.28, while RepoAudit attains a comparable F1 score of 0.61 but a higher coverage of 0.35. This improvement stems from RepoAudit's neural-symbolic retrieval workflow, which instructs the LLM to extract data-flow paths that are dependent on the given program location from the retrieved functions and performs detection based on these paths, yielding more targeted context than directly analyzing all retrieved functions.

Compared with the results in RQ2, both OpenHands and RepoAudit outperform LLMs operating without calling context in explainable vulnerability detection. However, due to the limitation in calling context retrieval capability, their coverage scores remain lower than those observed in RQ1 (0.49 for GPT 5 mini and 0.54 for Gemini 2.5 Flash), where LLMs are provided with sufficient calling context.

> **Finding 4:** Calling context retrieval is essential for effective vulnerability detection by LLM-based coding agents.

6 Conclusion

We present ManBo, a benchmark designed to evaluate vulnerability detection under realistic, repository-scale settings. We formulate two core tasks, namely calling context retrieval and explainable vulnerability detection, and systematically evaluates existing LLMs and LLM-driven coding agents on both tasks. Our evaluation reveals that deficiencies in calling context retrieval constitute a major bottleneck, substantially hindering the effectiveness of end-to-end explainable vulnerability detection. We expect ManBo to serve as a foundation for future research on calling context retrieval and code explanation generation in real-world codebases, extending beyond vulnerability detection to broader program reasoning tasks. We have uploaded the benchmark as supplementary data on OpenReview and will release it publicly upon acceptance.

7 Limitations

ManBo has several limitations that suggest directions for future extensions. First, the benchmark currently focuses on calling context for repository-level reasoning. Program values propagated through global variables or concurrent interactions across threads are not explicitly modeled, which limits coverage of certain real-world vulnerabilities in concurrent or stateful systems. Second, ManBo targets typestate-related vulnerabilities, with a focus on three representative memory corruption issues. Other vulnerability types, such as buffer overflows or numeric errors, are not yet included. Nevertheless, the data format introduced in Section 4.2 is general and can be naturally extended to support additional vulnerability categories. Finally, the benchmark construction relies primarily on manual annotation, which incurs non-trivial human effort and may introduce minor labeling errors despite multi-annotator cross-validation. A promising direction is to integrate static analysis tools to automatically generate candidate typestate paths, followed by human verification and refinement, to reduce annotation cost while maintaining data quality.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, and 1 others. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. 2014. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 259–269. ACM.

Thoms Ball. 1999. The concept of dynamic analysis. *ACM SIGSOFT Software Engineering Notes*, 24(6):216–234.

Max Brunsfeld. 2018. Tree-sitter-a new parsing system for programming tools. In *Strange Loop Conference,. Accessed–. URL: https://www. thestrangeloop. com//tree-sitter—a-new-parsing-system-for-programming-tools. html*.

Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224. USENIX Association.

Mark Chen. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

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*, pages 654–668.

CodeRabbit. 2025. Coderabbit: Ai-powered code review for github. https://www.coderabbit.ai/. Accessed: 2025-07-07.

Cursor. 2025. Bugbot documentation. https://docs.cursor.com/bugbot. Accessed: 2025-06-26.

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*.

Lisa Nguyen Quang Do and Eric Bodden. 2020. Explaining static analysis with rule graphs. *IEEE Transactions on Software Engineering*, 48(2):678–690.

Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. 2001. Dynamically discovering likely program invariants to support program evolution. *IEEE transactions on software engineering*, 27(2):99–123.

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.

Stephen J Fink, Eran Yahav, Nurit Dor, G Ramalingam, and Emmanuel Geay. 2008. Effective typestate verification in the presence of aliasing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(2):1–34.

GitHub. 2025. CodeQL. https://codeql.github.com/. [Online; accessed 13-July-2025].

Jinyao Guo, Chengpeng Wang, Dominic Deluca, Jinjie Liu, Zhuo Zhang, and Xiangyu Zhang. 2025a. Bugscope: Learn to find bugs like human. *arXiv preprint arXiv:2507.15671*.

Jinyao Guo, Chengpeng Wang, Xiangzhe Xu, Zian Su, and Xiangyu Zhang. 2025b. Repoaudit: An autonomous llm-agent for repository-level code auditing. *Preprint*, arXiv:2501.18160.

René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*, pages 437–440.

Tuo Li, Jia-Ju Bai, Yulei Sui, and Shi-Min Hu. 2022. Path-sensitive and alias-aware typestate analysis for detecting os bugs. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 859–872.

Ziyang Li, Saikat Dutta, and Mayur Naik. 2025. Iris: Llm-assisted static analysis for detecting security vulnerabilities. In *The Thirteenth International Conference on Learning Representations*.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, and 1 others. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.

Yunlong Lyu, Yi Fang, Yiwei Zhang, Qibin Sun, Siqi Ma, Elisa Bertino, Kangjie Lu, and Juanru Li. 2022. Goshawk: Hunting memory corruptions via structure-aware and object-centric memory operation synopsis. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 2096–2113. IEEE.

MITRE. 2025. Common Weakness Enumeration (CWE). `https://cwe.mitre.org/index.html`. Accessed: 2025-07-11.

Georgios Nikitopoulos, Konstantina Dritsa, Panos Louridas, and Dimitris Mitropoulos. 2021. Crossvul: a cross-language vulnerability dataset with commit data. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1565–1569.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. 2024. Training software engineering agents and verifiers with swe-gym. *arXiv preprint arXiv:2412.21139*.

Abhinav Rastogi, Adam Yang, Albert Q Jiang, Alexander H Liu, Alexandre Sablayrolles, Amélie Héliou, Amélie Martin, Anmol Agarwal, Andy Ehrenberg, Andy Lo, and 1 others. 2025. Devstral: Fine-tuning language models for coding agent applications. *arXiv preprint arXiv:2509.25193*.

Henry Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical society*, 74(2):358–366.

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):388–410.

Saurabh Sinha, Alessandro Orso, and Mary Jean Harrold. 2004. Automated support for development, maintenance, and testing in the presence of implicit flow control. In *Proceedings. 26th International Conference on Software Engineering*, pages 336–345. IEEE.

Yannis Smaragdakis, George Balatsouras, and 1 others. 2015. Pointer analysis. *Foundations and Trends® in Programming Languages*, 2(1):1–69.

Benjamin Steenhoek, Hongyang Gao, and Wei Le. 2024. Dataflow analysis-inspired deep learning for efficient vulnerability detection. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*, pages 16:1–16:13. ACM.

Robert E. Strom and Shaula Yemini. 1986. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, SE-12(1):157–171.

Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, pages 265–266. ACM.

Hugo Touvron and 1 others. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Chengpeng Wang, Wuqi Zhang, Zian Su, Xiangzhe Xu, Xiaoheng Xie, and Xiangyu Zhang. 2024a. LLMDFA: Analyzing dataflow in code with large language models. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Xinchen Wang, Ruida Hu, Cuiyun Gao, Xin-Cheng Wen, Yujia Chen, and Qing Liao. 2024b. 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.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, and 1 others. 2024c. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*.

Xiangzhe Xu, Zian Su, Jinyao Guo, Kaiyuan Zhang, Zhenting Wang, and Xiangyu Zhang. 2024. Prosec: Fortifying code llms with proactive security alignment. *arXiv preprint arXiv:2411.12882*.

Mingwei Zheng, Chengpeng Wang, Xuwei Liu, Jinyao Guo, Shiwei Feng, and Xiangyu Zhang. 2025. An llm agent for functional bug detection in network protocols. *Preprint*, arXiv:2506.00714.

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 Challenges for Automation

As discussed in Section 4, typestate slices correspond to fundamentally semantic program properties. According to Theorem (Rice, 1953), deciding whether a semantic program property holds or not is inherently undecidable. As a consequence, it is theoretically infeasible to design a fully automated algorithm that can precisely determine such properties and produce perfect ground truth annotations. We initially explored the possibility of leveraging existing static and dynamic analysis techniques to automatically recover the typestate slices. However, our investigation revealed that current techniques are insufficient to substantially reduce the annotation burden in practice.

Existing Static Analyzers. Existing static analysis tools, including SVF (Sui and Xue, 2016), FlowDroid (Arzt et al., 2014), CodeQL (GitHub, 2025), and KLEE (Cadar et al., 2008), predominantly operate on intermediate representations (IRs) rather than directly on source code. The IRs generated by compiler are well suited for foundational analyses such as pointer analysis (Smaragdakis et al., 2015). As a consequence, the outputs of these analyzers typically characterize relationships among IR-level program values instead of source-level variables and expressions. However, LLMs perform reasoning directly over source code. This mismatch introduces a semantic gap between traditional static analysis tools and the source-level vulnerability detection capability that our benchmark is designed to evaluate. As a result, existing static analyzers cannot be directly leveraged to generate source-level typestate slices required by our benchmark. Hence, automating data annotation using existing static analysis tools remains impractical at present.

Existing Dynamic Analyzers. Dynamic analysis (Ernst et al., 2001; Ball, 1999; Sinha et al., 2004) observes program behavior by executing the program on concrete inputs and analyzing properties of its runtime execution (e.g., via traces/profiles and observed variable values at selected program points). It can capture execution-specific behaviors and has been widely used for program understanding and vulnerability detection. However, dynamic analysis (especially when used as raw tracing) does not by itself guarantee a minimal, source-level explanation of a vulnerability, such as a compact typestate-transition sequence or a summarized interprocedural pointer-propagation narrative, without additional abstraction or inference. In practice, execution traces can remain large and include operations irrelevant to the root cause, which makes post-mortem inspection difficult without further structuring/summarization. We also explored leveraging dynamic analysis to automatically generate vulnerability traces, but found that the generated traces could not precisely or consistently capture the typestate transitions and propagation semantics required for our benchmark.

B Typestate-Slice Coverage

In this section, we describe the procedure used to compute the typestate slice coverage score. For each vulnerable instance, we treat the annotated typestate slice as the groundtruth representing the minimal sequence of operations required to precisely characterize the vulnerability. From this slice, we extract the set of operations, where each operation is represented as a tuple consisting of a function name and the corresponding source-code line number. We then extract the set of operations from the generated typestate slice and compare it against the groundtruth. Since a single operation may span multiple lines of code (up to three lines), we consider a groundtruth operation to be covered if there exists a generated operation with the same function name whose line number falls within a window of ±3 lines around the groundtruth line number. Based on this criterion, the coverage score is defined as the proportion of groundtruth operations covered by the generated typestate slice

$$\text{Coverage} = \frac{|\text{Ops}_{\text{generated}} \cap \text{Ops}_{\text{groundtruth}}|}{|\text{Ops}_{\text{groundtruth}}|}.$$

C Details of LLM Settings

We evaluate eight LLMs, including three flagship commercial models: GPT 5, Gemini 2.5 Pro, and Claude Sonnet 4.5; three cost-efficient commercial models: GPT 5 Mini, Gemini 2.5 Flash, and Claude Haiku 4.5; and two open-source models: Qwen3 Coder 30B A3B Instruct (denoted as Qwen3 Coder 30B) and GPT OSS 120B. The open-source models are hosted on our local servers equipped with eight NVIDIA RTX A6000 GPUs, while all commercial models are accessed via their respective platform APIs [1][2][3].

[1] https://platform.openai.com/docs/models
[2] https://ai.google.dev/gemini-api/docs
[3] https://platform.claude.com/docs/en/about-claude/models/overview

For Gemini and Claude models, we configure an output token limit of 4096 and a reasoning token budget of 2048. For GPT 5 and GPT 5 Mini, which do not expose explicit controls over token limits, we use the default configuration and set the reasoning effort to *medium*. For GPT OSS 120B, we set the output token limit to 4096 and the reasoning budget to 2048 tokens. For Qwen3 Coder 30B, which is not a reasoning model, we set the output token limit to 4096 and the decoding temperature to 0 to minimize randomness.

D Details of Evaluated Agents

To evaluate the calling context retrieval capability of LLM-based coding agents, we select OpenHands (Wang et al., 2024c), one of the most widely adopted coding agents in practice, and RepoAudit (Guo et al., 2025b), a state-of-the-art vulnerability detection agent, as complementary baselines.

OpenHands provides the underlying LLM with an execution environment that allows it to generate and execute search scripts, enabling grep-style context retrieval over the codebase. The execution results are returned to the LLM for further analysis. Completing a single task involves multiple iterations, during which the LLM performs planning, invokes execution tools, analyzes retrieved results, and determines whether the task has been completed. Following common practice in prior work (Rastogi et al., 2025; Pan et al., 2024), we set the maximum number of iterations to 50 during evaluation.

While OpenHands is general and effective for broad context exploration, it does not explicitly model the structural properties of the program, such as the call graph. As a complementary baseline, we also evaluate RepoAudit, a vulnerability detection agent that performs calling-context retrieval in a structure-aware manner. RepoAudit first leverages static analysis tools (e.g, Tree-sitter (Brunsfeld, 2018)) to construct a call graph. It then selectively traverses call graph edges to retrieve the relevant calling context. Starting from a given program location, the LLM extracts the data-flow path associated with that location and triggers the retrieval of adjacent functions along the call graph whenever inter-procedural dependencies are involved. During evaluation, we set the maximum call-depth to 8, which matches the maximum number of functions in our dataset.

E Case Study

In this section, we present representative false positive cases caused by hallucinations in LLM reasoning. Although the models correctly predict the final decision as *Vulnerable*, the generated typestate slices do not correspond to actual vulnerabilities in the code.

E.1 False Positive 1

Listing 1 shows a code fragment containing a memory leak vulnerability. The wrapper function `LZ4F_calloc` allocate a memory object at line 3 inside function `LZ4F_createDecompressionContext_advanced`. The allocated memory is then returned to the caller function `LZ4F_createDecompressionContext` and subsequently propagated to `LZ4F_readOpen`. Within `LZ4F_readOpen`, the memory object is stored in the pointer `ret` at line 17. When `LZ4F_readOpen` returns along an error-handling branch at line 27, the allocated memory is not freed, resulting in a memory leak.

In the Explainable Vulnerability Detection task, the LLM correctly predicts the final decision as *Vulnerable*; however, the generated typestate slice does not correspond to the actual vulnerability. Specifically, the model-generated path starts from a `calloc` operation at line 12 in `LZ4F_readOpen` and ends at the same `return` statement at line 27. In this inferred path, the memory is already freed at line 26 prior to the return, and therefore no memory leak occurs along this data flow. As a result, the model produces a false positive at the explanation level for the memory leak.

E.2 False Positive 2

Listing 2 presents a code fragment that contains a use-after-free vulnerability. In function `JS_NewObjectFromShape`, the wrapper function `js_malloc` allocates a memory object at line 6. This object is subsequently returned through `JS_NewObjectProtoClass` and `js_create_from_ctor` to the function `JS_CallConstructorInternal`. The memory object is then passed as the third argument to the callee function `JS_CallInternal`, where it is stored in the pointer `val` at line 23 and further propagated to the stack pointer `sp` at line 27. Later, the object is loaded from `sp` at line 29 and freed via the wrapper function `JS_FreeValue` at line 30. Subsequently, in the

Listing 1: A memory leak example.

```
LZ4F_dctx* LZ4F_createDecompressionContext_advanced(LZ4F_CustomMem customMem, unsigned version)
{
    LZ4F_dctx* const dctx = (LZ4F_dctx*)LZ4F_calloc(sizeof(LZ4F_dctx), customMem);
    if (dctx == NULL) return NULL;

    dctx->cmem = customMem;
    dctx->version = version;
    return dctx;
}

LZ4F_errorCode_t LZ4F_createDecompressionContext(LZ4F_dctx** LZ4F_decompressionContextPtr, unsigned versionNumber)
{
    assert(LZ4F_decompressionContextPtr != NULL);
    RETURN_ERROR_IF(LZ4F_decompressionContextPtr == NULL, parameter_null);

    *LZ4F_decompressionContextPtr = LZ4F_createDecompressionContext_advanced(LZ4F_defaultCMem, versionNumber);
    if (*LZ4F_decompressionContextPtr == NULL) {
        RETURN_ERROR(allocation_failed);
    }
    return LZ4F_OK_NoError;
}

LZ4F_errorCode_t LZ4F_readOpen(LZ4_readFile_t** lz4fRead, FILE* fp)
{
  char buf[LZ4F_HEADER_SIZE_MAX];
  size_t consumedSize;
  LZ4F_errorCode_t ret;
  LZ4F_frameInfo_t info;

  if (fp == NULL || lz4fRead == NULL) {
    return -LZ4F_ERROR_GENERIC;
  }

  *lz4fRead = (LZ4_readFile_t*)calloc(1, sizeof(LZ4_readFile_t));
  if (*lz4fRead == NULL) {
    return -LZ4F_ERROR_allocation_failed;
  }

  ret = LZ4F_createDecompressionContext(&(*lz4fRead)->dctxPtr, LZ4F_getVersion());
  if (LZ4F_isError(ret)) {
    free(*lz4fRead);
    return ret;
  }

  (*lz4fRead)->fp = fp;
  consumedSize = fread(buf, 1, sizeof(buf), (*lz4fRead)->fp);
  if (consumedSize != sizeof(buf)) {
    free(*lz4fRead);
    return -LZ4F_ERROR_GENERIC;
  }

  ret = LZ4F_getFrameInfo((*lz4fRead)->dctxPtr, &info, buf, &consumedSize);
  if (LZ4F_isError(ret)) {
      LZ4F_freeDecompressionContext((*lz4fRead)->dctxPtr);
      free(*lz4fRead);
      return ret;
    }
  ...
}
```

caller function `JS_CallConstructorInternal`, the freed memory is used at line 13, resulting in a use-after-free vulnerability.

When provided with these functions, the LLM correctly predicts the final decision as *Vulnerable* and successfully identifies the memory object returned by `js_create_from_ctor`. However, it incorrectly interprets the operation at line 10 in `JS_CallConstructorInternal` as a store operation that stores the memory object from pointer `obj` into `ret`. In the subsequently generated reasoning path, the model assumes that the memory is freed at line 13 and returned to `JS_CallInternal` at line 14. It then treats the statement `*sp++ = ret_val;` at line 14 in `JS_CallInternal` as the operation that triggers the vulnerability. Although the generated path appears plausible and correctly tracks the target memory object, it is ultimately a false positive caused by the model's hallucinated interpretation of the functionality of `JS_CallInternal`.

Listing 2: A use after free example.

```
static JSValue JS_NewObjectFromShape(JSContext *ctx, JSShape *sh, JSClassID class_id)
{
    JSObject *p;

    js_trigger_gc(ctx->rt, sizeof(JSObject));
    p = js_malloc(ctx, sizeof(JSObject));
    ...
    return JS_MKPTR(JS_TAG_OBJECT, p);
}
```

```
JSValue JS_NewObjectProtoClass(JSContext *ctx, JSValueConst proto_val,
                               JSClassID class_id)
{
    ...
   return JS_NewObjectFromShape(ctx, sh, class_id);
}
```

```
static JSValue js_create_from_ctor(JSContext *ctx, JSValueConst ctor,
                                   int class_id)
{
    ...
    obj = JS_NewObjectProtoClass(ctx, proto, class_id);
    JS_FreeValue(ctx, proto);
    return obj;
}
```

```
static JSValue JS_CallConstructorInternal(JSContext *ctx,
                                          JSValueConst func_obj,
                                          JSValueConst new_target,
                                          int argc, JSValue *argv, int flags)
{
    ...
    obj = js_create_from_ctor(ctx, new_target, JS_CLASS_OBJECT);
    if (JS_IsException(obj))
        return JS_EXCEPTION;
    ret = JS_CallInternal(ctx, func_obj, obj, new_target, argc, argv, flags);
    if (JS_VALUE_GET_TAG(ret) == JS_TAG_OBJECT ||
        JS_IsException(ret)) {
        JS_FreeValue(ctx, obj);
        return ret;
    } else {
        JS_FreeValue(ctx, ret);
        return obj;
    }
}
```

```
static JSValue JS_CallInternal(JSContext *caller_ctx, JSValueConst func_obj,
                       JSValueConst this_obj, JSValueConst new_target,
                       int argc, JSValue *argv, int flags)
{
    ...
    ret_val = JS_CallConstructorInternal(ctx, call_argv[-2],
                                         call_argv[-1],
                                         call_argc, call_argv, 0);
    if (unlikely(JS_IsException(ret_val)))
        goto exception;
    for(i = -2; i < call_argc; i++)
        JS_FreeValue(ctx, call_argv[i]);
    sp -= call_argc + 2;
    *sp++ = ret_val;
    ...
    JSValue val;
    uint32_t tag = JS_VALUE_GET_TAG(this_obj);
    if (likely(tag == JS_TAG_OBJECT))
        goto normal_this;
    if (tag == JS_TAG_NULL || tag == JS_TAG_UNDEFINED) {
        val = JS_DupValue(ctx, ctx->global_obj);
    } else {
        val = JS_ToObject(ctx, this_obj);
        if (JS_IsException(val))
            goto exception;
    }
    *sp++ = val;
    ...
    for(pval = local_buf; pval < sp; pval++) {
        JS_FreeValue(ctx, *pval);
    }
    ...
}
```