FAULTLOC: Evaluating Coding Agents For Fault Localization

Jianhong Tu[1] Shubham Gaur[1] Rathik Murtinty[1] Zhun Wang[2]
Tianneng Shi[2] Dawn Song[2] Chenguang Wang[1]*

[1]University of California, Santa Cruz [2]University of California, Berkeley

Abstract

Coding agents powered by large language models have demonstrated strong performance on security tasks such as vulnerability discovery and automated patching. However, existing benchmarks primarily evaluate patch generation through functional tests, allowing agents to succeed by mitigating symptoms at the crash site without identifying the underlying root cause. As a result, it remains unclear whether these systems perform genuine code reasoning or rely on superficial fixes. We introduce FAULTLOC, a benchmark designed to evaluate fault localization in large codebases. Given a bug report and a proof-of-concept input, agents must identify the code region responsible for the vulnerability, with predictions evaluated against developer patches. FAULTLOC comprises 900 real-world vulnerabilities with verified PoCs and fixes across widely-used open-source projects. Our evaluation shows that even frontier agents fail to recover the developer-identified root-cause region in most cases, achieving only 40% overlap with ground-truth buggy lines and frequently mislocalizing vulnerabilities at the file level. These results highlight the need for agents that reason about causal structure rather than merely suppress observable failures.

1 Introduction

Recent advancements in large language models (LLMs) have given rise to autonomous coding agents capable of addressing real-world software engineering tasks (Jimenez et al., 2024; Wang et al., 2025a; Xia et al., 2025), driven by increasingly proficient long-horizon reasoning and tool-use capabilities (Team, 2026; Anthropic, 2026; OpenAI, 2026). These agents are now being deployed as automated vulnerability repair systems, demonstrating the potential to reshape the cybersecurity landscape (Guo et al., 2025). It is therefore critical to develop a holistic understanding of the capability boundaries of frontier agentic systems, encompassing both offensive and defensive competencies.

Extensive prior work has proposed cybersecurity benchmarks spanning vulnerability detection, exploitation, and patching (Zhang et al., 2025a; Wang et al., 2025b; Meta, 2025; Zhang et al., 2025b; Ahmed et al., 2025). Among these, automatic patch generation has received particular attention given its direct impact on hardening codebases and deployed services against known vulnerabilities. These benchmarks typically rely on predefined proof-of-concept (PoC) tests to verify that the reported bug no longer manifests and that existing functionality remains intact. However, a fundamental limitation arises: a coding agent can craft a narrow guardrail that suppresses the specific PoC input without addressing the underlying root cause. Such a patch may pass all provided tests yet leave the broader vulnerability class exploitable, as a single PoC is rarely exhaustive enough to cover every manifestation of the flaw. In other words, passing the PoC-based test suite is a necessary but insufficient condition for confirming that the root vulnerability has been eliminated.

To address this limitation, we shift the evaluation target from patch correctness to root-cause identification, which requires a genuine understanding of the defect rather than surface-level symptom suppression. We propose FAULTLOC, a large-scale fault localization

*Corresponding author: chenguangwang@ucsc.edu

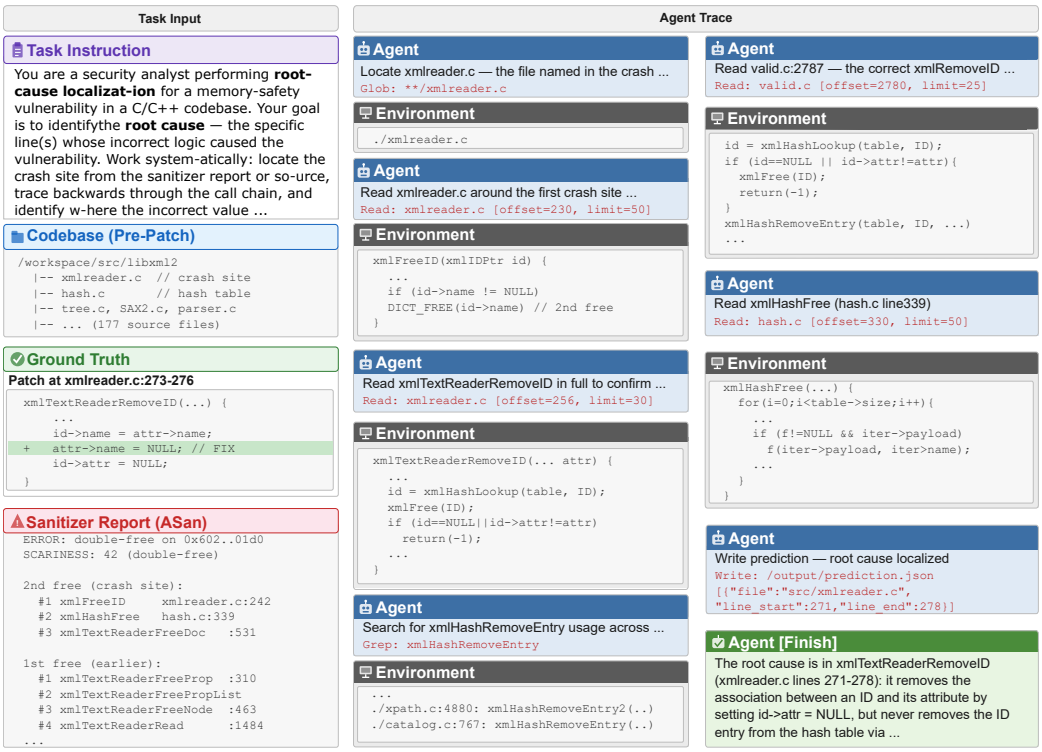

Figure 1: Qualitative example of a fault localization task in FAULTLOC. Each task is initialized with four inputs: a task instruction, the vulnerable codebase, a PoC, and a sanitizer report. The agent analyzes the call chain to trace from the crash site to the root cause and predicts the suspicious code regions. The ground truth patch is hidden from the agent and used during scoring.

benchmark designed for modern AI coding agents, comprising vulnerabilities discovered in real-world projects. Fault localization—the precise identification of the defective code region responsible for ill program behavior—is a prerequisite for patch generation and instrumental to software quality assurance (Wong et al., 2016). Crucially, fault localization also serves as a natural probe into agent reasoning: it exposes *where* the agent believes the defect resides, offering a window into its intermediate understanding of the codebase and isolating analytical capability from code-editing proficiency.

FAULTLOC contains 900 instances collected from 124 popular open-source projects. Each vulnerability was originally discovered and reported by OSS-Fuzz (Serebryany, 2017), an automated service that stress-tests programs with mutated inputs to find bugs at scale. As illustrated in Fig. 1, each task challenges a coding agent to locate all defective code regions by predicting line spans within files, given the sanitizer report produced by a PoC. The agent must precisely document all such line spans across the entire codebase. We rigorously evaluate the predicted localizations against ground-truth patches written by human developers, reporting accuracy at the file, hunk, and line levels to cover all hierarchical granularities. To solve these tasks, agents must perform multi-hop reasoning, tracing through the call chain from the crash site to the upstream root cause while strategically navigating source codes comprising thousands of files and spanning distinct vulnerability types. Successful agents pinpoint the relevant code sections without including non-essential surrounding context, thereby attaining high precision and recall.

We extensively benchmark 8 proprietary and open-source models across 4 agent harnesses on FAULTLOC. Our results reveal that fault localization remains challenging even for today's frontier agents, which frequently diverge from human ground truth and can miss the underlying root cause. The best-performing model, GPT-5.3-Codex (OpenAI, 2026), achieves a Line F1 score of 24.90, approximately 4 points above the strongest open-source

model, Qwen3.5-397B-A17B (Qwen Team, 2026). Notably, larger model size does not always yield better performance: Qwen3.5-27B (Qwen Team, 2026) surpasses the substantially larger Kimi K2.5 (Team, 2026) (1-trillion parameters) and demonstrates competitive performance against its own larger variant, attaining an F1 score of 20.43. Our analysis further indicates that instances requiring multi-site fixes are considerably harder and that models do not memorize exact vulnerability fixes but must reason over the codebase to deduce the answer.

To summarize, we make the following key contributions:

- A large-scale cybersecurity fault localization benchmark for coding agents, built on real codebases and rigorously validated vulnerabilities.
- Clean, human-authored ground-truth patches for security issues, free of noisy comments, changelogs, and irrelevant additions.
- A holistic evaluation spanning state-of-the-art proprietary and leading open-source models.

2 Related Works

Coding Agents & Benchmarks The growing long-horizon planning and tool-use capabilities of LLMs have enabled general-purpose coding agents that interact with computer environments through file I/O and terminal commands. Notable research-oriented agents include SWE-Agent, which defines a specialized agent–computer interface for general agentic coding (Jimenez et al., 2024). SWE-Bench (Jimenez et al., 2024), a companion benchmark, evaluates agents on resolving real GitHub issues. SWE-Bench Pro (Deng et al., 2025) and SWE-Bench Verified (OpenAI, 2024) are hardened variants featuring more complex tasks that require multi-site modifications and stricter selection criteria. While these benchmarks assess whether agent-generated patches resolve coding issues, SWT-Bench (Mündler et al., 2024) measures how effectively agents generate tests that validate reported bugs. Terminal-Bench (Merrill et al., 2026) curates real-life terminal-based workflows, evaluating agents' ability to automate diverse tasks beyond software engineering.

Cybersecurity Benchmarks & Fault Localization The cybersecurity domain has seen an increasing application of coding agents. Early benchmarks feature Capture-the-Flag style challenges: the NYU CTF Bench (Shao et al., 2024) comprises 200 challenges across cryptography, web exploitation, reverse engineering, and three additional domains, while Cybench (Zhang et al., 2025b) presents 40 complex tasks that human experts require hours to complete. A separate line of benchmarks curates test instances from real-world projects: SEC-Bench (Lee et al., 2025) and CyberGym (Wang et al., 2025b) evaluate coding agents on PoC generation and automatic vulnerability patching using software sourced from the National Vulnerability Database (National Institute of Standards and Technology, 2024) and OSS-Fuzz (Serebryany, 2017). BountyBench (Zhang et al., 2025a) contains tasks spanning detection, exploitation, and patching, with estimated dollar-value measurements of severity. Among cybersecurity tasks, agent performance on fault localization remains relatively understudied. Prior benchmarks dedicated to this problem, including LinuxFLBench (Zhou et al., 2025) and T2L-ARVO (Xi et al., 2025), are limited to single-file bugs, a narrow range of agents, and a small number of benchmark instances.

3 Benchmark Construction

3.1 Preliminaries

The seed codebases used in our dataset are sourced from OSS-Fuzz (Serebryany, 2017), Google's continuous fuzzing service. Its bug-finding pipeline leverages *fuzzers* and *sanitizers* to stress-test programs and automatically discover security vulnerabilities at scale. Fuzzing is an automated software testing technique that repeatedly feeds randomly mutated or generated inputs to a program to trigger unexpected behaviors such as crashes, memory corruption, or assertion failures (Manès et al., 2021). Sanitizers such as AddressSanitizer

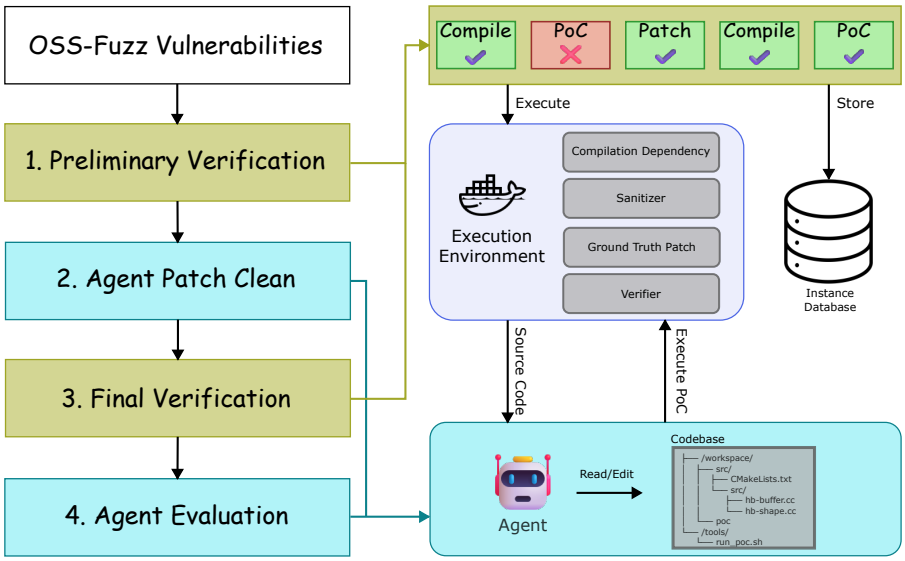

Figure 2: Overview of the FAULTLOC construction and evaluation pipeline. Each OSS-Fuzz vulnerability undergoes preliminary verification, patch cleaning, and final verification. A general architecture is used for patch cleaning and agent evaluation with an appropriate prompt and a selected set of input assets. The verifier computes scores by comparing the agent's prediction against the hidden ground truth.

(ASan) (Serebryany et al., 2012) and MemorySanitizer (MSan) (Stepanov & Serebryany, 2015) are compile-time instrumentation tools that detect memory-safety violations—including buffer overflows, use-after-free, and uninitialized reads—at runtime, producing detailed diagnostic reports. A fuzzer-generated input that causes a sanitizer to report a violation constitutes a confirmed vulnerability. OSS-Fuzz supports projects written in C/C++, Java, Python, and other mainstream languages. As of 2025, the service has identified over 13,000 vulnerabilities across more than 1,000 projects (Google, 2025).

3.2 Task Formulation

Task Input In the main evaluation, given a software repository in its vulnerable state along with the accompanying sanitizer report and PoC input, the agent is instructed to identify the root-cause locations responsible for the crash. We define root causes as incorrect or missing logic in the code that must be added, deleted, or modified. The input artifacts are organized into three categories: (1) *Code*: the original source files from a vulnerable snapshot of the project. (2) *Sanitizer*: the sanitizer dependency, the compiled software executable, and a script to execute the PoC. An example sanitizer report is provided in Appendix F; (3) *PoC & Metadata*: the byte-level PoC input, along with a text description of the crash type and sanitizer type; All three input categories are included in the main experiments to provide agents with sufficient information and to mirror the real-world fault localization workflow. Certain inputs are deliberately omitted to create more challenging evaluation settings in the ablation studies. We specify how the agent instructions differ across evaluation settings in Appendix E.

Task Output After exploring the code repository, the agent must submit a prediction in the form of a list of spans, each identifying a contiguous range of lines within a single file. All spans together constitute the localization of a single vulnerability, as the root cause may involve logic flaws spanning multiple functions and files. For code that must be deleted or modified, the agent includes all affected lines directly. However, for logic that must be added—code absent from the original codebase—we instruct the agent to pinpoint the line where the function requires hardening, guided by a demonstrative example in the prompt.

For instance, a missing `else if` branch may be needed after an `if` statement to properly handle an edge case, and the agent should flag the end of the *if* segment.

Ground Truth & Evaluation Metrics Following prior fault localization research, we curate the code patch that addresses each vulnerability and extract the modified lines as the ground truth for the localization task (Chen et al., 2025; Just et al., 2014). We measure the accuracy of agent predictions at three hierarchical levels: file, hunk, and line. A hunk is defined as one contiguous block of changed lines together with its surrounding context. A hunk is considered correctly predicted if the agent flags any line within it, thereby allowing fuzzy matching and awarding credit even when the exact line is missed. At the file and hunk levels, we report Recall, defined as the proportion of ground-truth files or hunks included in the agent's prediction. At the line level, we report Recall, Precision, and macro F1, so that trivial predictions spanning the entire file are penalized.

3.3 Vulnerability Curation Pipeline

We initialize the data curation process by drawing code environments from the ARVO database (Mei et al., 2024), which contains over 6,000 reproducible security vulnerabilities discovered by OSS-Fuzz. In this section, we detail how we ensure the quality of the ground-truth patches and construct the agent evaluation environment.

Preliminary Verification From the ARVO database, we extract container environments equipped with source code, sanitizer configurations, and PoC inputs, and implement a multi-stage verification pipeline illustrated in Fig. 2. Prior to verification, we initialize from the original container but remove extraneous build artifacts and untracked files to restore the codebase to a clean state. We then programmatically verify three assumptions: (1) the program compiles within a reasonable time without critical errors; (2) the PoC input triggers a sanitizer-reported error on the vulnerable version but is properly guarded after the patch is applied; and (3) the patch matches the current state of the codebase. Concretely, we run compilation and PoC tests both before and after patching, expecting all operations to succeed except the pre-patch PoC execution, which must trigger the sanitizer. At this stage, we identified over 2,000 patches that failed to apply due to context mismatches, which could be fixed and included in our benchmark as future work.

Agent Patch Cleaning Once we collect a set of reproducible vulnerabilities with functional patches, we employ coding agents to automatically inspect and clean the patch content in two stages. To allow the unsupervised agent to explore the codebase without disrupting the compilation environment, we provision a separate container to host the agent harness and mount the source code directory into it. The mounted directory then becomes the default workspace for the agents, who also have unlimited access to all directories in the agent container. Access to PoC execution and compilation is exposed to the agent via a dedicated remote-execution tool. In the first stage, we instruct the agent to examine each patch and identify common issues, including excessive comments, test additions, changelog modifications, and code changes irrelevant to the vulnerability itself. Agent categorizations are sampled and manually inspected by human experts to ensure quality. In the second stage, the agent cleans the patches based on the preceding diagnosis and the bug description, producing a minimal patch containing only the core fix.

Final Verification In the final stage, we re-validate the cleaned patches by running the verification pipeline described in the previous section, discarding all instances that violate our requirements regarding compilation and PoC tests. We also inspect the distribution of patch sizes and remove outliers that rewrite hundreds of lines across dozens of files. After filtering, we retain 900 high-quality instances from 124 open-source projects. Tab. 1 summarizes the patch sizes; additional benchmark statistics are provided in Appendix C.

Table 1: Patch edit-size statistics across benchmark instances.

Metric	Mean	Median	Max
# Files edited	1.39	1	15
# Hunks edited	2.79	1	83
# Lines edited	8.17	2	220

We categorize a benchmark instance as multi-site if the ground-truth patch contains more than one hunk, whether spanning different files or different regions within the same file. Roughly 55% of the instances involve single-site modifications, while the remainder require multi-site fixes.

Agent Evaluation For the evaluation experiments, we reuse the agent–environment architecture from the patch cleaning pipeline. Verified codebases, sanitizer scripts, and PoC inputs are mounted in the agent container, with the program pre-compiled for fast PoC execution. Certain artifacts are withheld depending on the experimental setting. Git-related commands are disabled system-wide in the agent container to prevent leakage of the ground-truth patch.

4 Experimental Evaluation

4.1 Experiment Setup

We present a comprehensive evaluation of frontier LLMs paired with mainstream coding agent harnesses. Our model lineup includes GPT-5.3-Codex (OpenAI, 2026), Claude Sonnet 4.6 (Anthropic, 2026), MiniMax-M2.5 (MiniMax, 2026), Kimi K2.5 (Team, 2026), GLM-4.7 (Team et al., 2025), Qwen3.5-397B-A17B (Qwen Team, 2026), Qwen3.5-27B (Qwen Team, 2026), and GPT-OSS-120B (OpenAI, 2025b). In practice, Claude Sonnet 4.6 occasionally enters extended thinking loops, creating a long tail of costly runs. We impose a per-instance budget cap of \$3—approximately 4.6× the model's average cost of \$0.65 per instance—to bound these outliers while leaving typical runs unaffected. Claude must submit its best guess when running out of budget. For reference, GPT-5.3-Codex averages \$0.35 per instance. All remaining models are self-hosted and evaluated without restrictions.

We evaluate four agent harnesses: GPT-5.3-Codex is paired with Codex CLI (OpenAI, 2025a); Claude Sonnet 4.6 uses Claude Code (Anthropic, 2025); all remaining LLM backbones are paired with OpenCode (Scoutos, 2025); and Kimi K2.5 and MiniMax-M2.5 are additionally evaluated with mini-SWE-agent (Yang et al., 2024), the successor of SWE-agent. To restrict agents to the provided resources, we disable WebFetch, WebSearch, and similar web-access tools, as well as `wget` and `curl` in the bash terminal. We also explicitly instruct each agent not to retrieve information from online sources. In our benchmark, a strong agent would achieve high File and Line Recall while maintaining high Line Precision, accurately identifying all defective lines without inflating predictions with non-essential surrounding code or false-positive locations.

4.2 Main Results

Tab. 2 presents the evaluation results in the main setting across all three hierarchical levels. Overall, the best-performing model, GPT-5.3-Codex, attains the highest recall at the file, hunk, and line levels, with scores of 74.13, 41.44, and 37.03, respectively. The imperfect performance—particularly the limited line-level alignment with human ground truth—underscores the difficulty of FAULTLOC. The sharp drop of roughly 33 percentage points from file- to hunk-level recall indicates that models may still miss the defective code region even after identifying the correct file. Moreover, Hunk Recall generally exceeds Line Recall by only 1–4 percentage points across agents, suggesting that agents occasionally produce near-correct predictions despite failing to match the ground truth exactly. Although our budget-capped Claude Sonnet 4.6 does not match

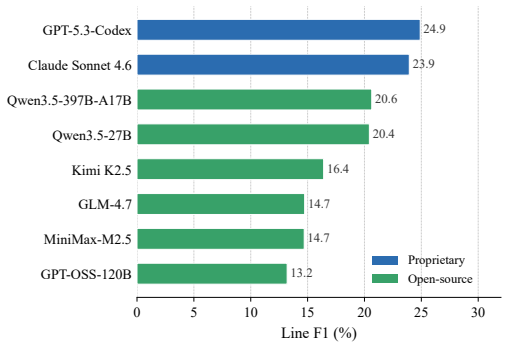

Figure 3: Line F1 scores on FAULTLOC. Proprietary and open-source models use native harnesses and OpenCode, respectively.

Table 2: Main fault localization results on FAULTLOC. We report Recall at the file, hunk, and line levels, along with line-level Precision and macro F1. Best scores are shown in **bold**. GPT-5.3-Codex leads on all metrics except Line Precision.

Model	Harness	File Recall	Hunk Recall	Line Recall	Line Precision	Line F1
GPT-5.3-Codex	Codex	**74.13**	**41.44**	**37.03**	27.20	**24.90**
Claude Sonnet 4.6	Claude Code	70.52	34.61	30.32	**29.57**	23.94
MiniMax-M2.5	OpenCode	61.15	21.33	18.32	19.57	14.70
MiniMax-M2.5	mini-SWE-agent	63.36	22.71	19.98	16.71	14.01
Kimi K2.5	OpenCode	65.56	25.44	22.90	19.58	16.41
Kimi K2.5	mini-SWE-agent	66.97	28.46	26.51	22.03	18.89
GLM-4.7	OpenCode	61.05	23.19	19.59	19.19	14.74
Qwen3.5-397B-A17B	OpenCode	66.16	28.87	25.57	25.79	20.64
Qwen3.5-27B	OpenCode	64.77	29.20	26.29	23.98	20.43
GPT-OSS-120B	OpenCode	59.24	22.10	19.72	15.72	13.20

the highest recall scores, it outperforms GPT-5.3-Codex with the best Line Precision of 29.57. This result indicates that Claude tends to produce more precise predictions with generally narrower line spans. Additionally, we present a qualitative analysis of agent behavior in Fig. 1 in Appendix B as well as a discussion of common failure patterns.

Proprietary vs. Open-Source A notable gap emerges when comparing open-source models with proprietary ones: no open-source model attains a File Recall above 70 percentage points or a Hunk Recall above 30 percentage points. Even the best open-source model, Qwen3.5-397B-A17B, trails Claude Sonnet 4.6 by 4 percentage points in File Recall and 6 percentage points in Hunk Recall. We also observe that Qwen3.5-397B-A17B surpasses both Kimi K2.5 and MiniMax-M2.5 by approximately 6 points in Line F1, highlighting substantial performance variation even among leading open-source models.

Large vs. Small Models Notably, the smaller Qwen3.5-27B outperforms the 1-trillion-parameter Kimi K2.5 by a substantial margin and roughly matches the performance of its own larger variant, Qwen3.5-397B-A17B. This result suggests that model scale alone does not guarantee stronger performance on FAULTLOC; rather, agentic training and domain knowledge play a decisive role in code reasoning. A similar trend is observed for GPT-OSS-120B, which achieves Hunk and Line Recall comparable to those of the larger MiniMax-M2.5.

Harness Comparison Current agent harnesses are built with different tool sets, system instructions, and design goals. While OpenCode is similar to Codex CLI and Claude Code—featuring dedicated tools such as string-replacement editing and sub-agent support—mini-SWE-agent adopts a minimalist design with only a bash tool for string searching and file I/O. To study the impact of harness design on agent performance, we evaluate Kimi K2.5 and MiniMax-M2.5 on both harnesses and compare the results. Our analysis reveals that harness differences have a significant impact on performance. For both models, we observe a general increase in scores when switching to mini-SWE-agent, with a more pronounced effect on Kimi K2.5. For instance, Line Recall rises from 22.90 to 26.51 percentage points for Kimi K2.5, and from 18.32 to 19.98 for MiniMax-M2.5. However, MiniMax-M2.5 exhibits a 3-percentage-point decrease in Line Precision, suggesting a shift in behavior toward fuzzier predictions.

Single-Site vs. Multi-Site Based on the number of code regions modified by the ground-truth patch, we divide the benchmark instances into single-site and multi-site splits. Tab. 3 presents selected results across both splits. Consistent with the main setting, GPT-5.3-Codex achieves the best performance across both splits, except for single-site Line F1, where it is surpassed by Claude Sonnet 4.6. Overall, agents perform substantially better on the single-site split owing to its lower difficulty. For instance, GPT-5.3-Codex attains a Hunk Recall of 55.83 on single-site instances but only 24.33 on multi-site instances. Although the relative ranking of models remains consistent with the main evaluation, the gap between Claude Sonnet 4.6 and the leading open-source models narrows considerably on the multi-site split: Qwen3.5-397B-A17B trails by fewer than 1 percentage point in Hunk Recall. In contrast,

GPT-5.3-Codex leads Claude Sonnet 4.6 by a substantial margin, with approximately 8 percentage points higher Hunk Recall and 5 percentage points higher Line F1.

Table 3: Performance comparison on single-site versus multi-site instances in FAULTLOC. Multi-site instances, which require localizing defects across multiple code regions, are substantially harder: all models exhibit sharp declines in Hunk Recall and Line F1. Best scores per metric are shown in **bold**.

	Single-site			Multi-site		
Model	File Recall	Hunk Recall	Line F1	File Recall	Hunk Recall	Line F1
GPT-5.3-Codex	**83.44**	**55.83**	29.48	**63.05**	**24.33**	**19.45**
Claude Sonnet 4.6	81.39	49.90	**32.09**	57.55	16.38	14.22
MiniMax-M2.5	70.02	30.80	19.39	50.54	10.01	8.91
Kimi K2.5	74.23	35.79	21.33	55.18	13.06	10.52
GLM-4.7	67.33	31.68	18.35	55.96	12.04	9.62
Qwen3.5-397B-A17B	73.81	40.00	27.02	57.05	15.64	13.06
Qwen3.5-27B	72.88	41.82	27.44	55.20	14.29	12.57
GPT-OSS-120B	67.83	32.58	18.21	49.00	9.60	7.24

4.3 Ablation Studies

We conduct ablation studies by restricting the inputs available to the agent during environment initialization. Two settings are considered: *PoC+Code* and *Sanitizer+Code*, which intentionally omit sanitizer- and PoC-related artifacts, respectively, to challenge models with reduced visibility. In the *PoC+Code* setting, agents cannot execute PoC tests or reference the sanitizer report and are restricted to static analysis based on the PoC binary and vulnerability metadata, including crash types. In the *Sanitizer+Code* setting, the agent has access to the sanitizer report but cannot inspect the actual PoC input. Due to the prolonged evaluation time in the *PoC+Code* setting, we evaluate MiniMax-M2.5 and Kimi K2.5 on a reduced split of 100 randomly sampled instances. Both agents are run with the OpenCode harness.

Table 4: Ablation study on input composition using Kimi K2.5 and MiniMax-M2.5 with OpenCode on 100 sampled instances. Each setting modifies the task inputs relative to the Main configuration: *PoC+Code* removes the sanitizer report, while *Sanitizer+Code* removes the PoC. We report absolute scores and deltas (Δ) relative to Main.

		File Recall		Hunk Recall		Line F1	
Model	Setting	Score	Δ	Score	Δ	Score	Δ
MiniMax-M2.5	Main	48.86	–	17.34	–	11.30	–
MiniMax-M2.5	PoC+Code	29.39	-19.47	4.50	-12.84	2.45	-8.85
MiniMax-M2.5	Sanitizer+Code	54.28	+5.42	23.50	+6.16	15.79	+4.49
Kimi K2.5	Main	53.45	–	27.19	–	19.57	–
Kimi K2.5	PoC+Code	33.73	-19.72	6.90	-20.29	3.31	-16.26
Kimi K2.5	Sanitizer+Code	58.46	+5.01	26.60	-0.59	15.51	-4.06

As shown in Tab. 4, the sanitizer report is a high-signal input for agents, removing which substantially increases task difficulty. Hunk Recall drops by 12 and 20 percentage points for MiniMax-M2.5 and Kimi K2.5, respectively, with the latter exhibiting greater sensitivity. In the *Sanitizer+Code* setting, the two models exhibit distinct patterns. MiniMax-M2.5 experiences a performance boost when the PoC is unavailable: its scores across all levels increase by 4–6 percentage points, suggesting that the PoC acts as a distractor and the model cannot leverage the additional information effectively. Kimi K2.5, by contrast, shows a more nuanced response: its File Recall increases by 5 percentage points, while Hunk Recall and Line F1 decrease by 0.5 and 4 percentage points, respectively. This indicates that Kimi K2.5 effectively incorporates the PoC input into its reasoning, thereby producing more precise predictions. Supporting this interpretation, Line Precision drops from 26 to 20 percentage

points when the PoC is removed. Since Hunk Recall decreases only slightly, we conclude that without the PoC, Kimi K2.5 predicts wider line spans and includes more speculative code regions.

Table 5: Performance lower-bound under the *No-Code* setting, in which agents have no access to the source code. We report absolute scores and deltas (Δ) relative to the Main setting for MiniMax-M2.5 and Kimi K2.5 on OpenCode.

		File Recall		Hunk Recall		Line F1	
Model	Setting	Score	Δ	Score	Δ	Score	Δ
Minimax-M2.5	Main	61.15	–	21.33	–	14.70	–
Minimax-M2.5	No-Code	62.13	+0.98	12.67	-8.66	7.67	-7.03
Kimi K2.5	Main	65.56	–	25.44	–	16.41	–
Kimi K2.5	No-Code	65.75	+0.19	15.32	-10.12	7.61	-8.80

4.4 Lower-Bound Analysis

LLMs pretrained on internet-scale data may have memorized the ground-truth patches in our benchmark. Curating instances after each models' knowledge cutoff is commonly used for assessing the memorization issue, which we cannot afford since FAULTLOC contains historical vulnerabilities. Following Liang et al. (2025), we instead design a surrogate evaluation setting by removing the codebase entirely from the agent workspace, testing whether agents can recall the fix from parametric memory alone. This *No-Code* setting also establishes a performance lower bound without code reasoning or agentic behavior. We evaluate MiniMax-M2.5 and Kimi K2.5 with OpenCode on all 900 instances.

Tab. 5 reveals that both Hunk Recall and Line F1 nearly halve when the source code is unavailable. For example, the Line F1 of Kimi K2.5 drops from 16.41 to 7.61. Scores do not fall to zero because the sanitizer report contains a stack trace generated from the crash, allowing the root cause to be partially identified from function-level information alone. The persistent File Recall above 60% indicates that both agents rely heavily on files explicitly mentioned in the sanitizer report and fail to sufficiently explore other files in the repository. In summary, we find no convincing evidence that models can faithfully recall exact patches from parametric memory; in-depth code exploration and reasoning remain essential to succeed on FAULTLOC.

5 Conclusion & Future Work

We present FAULTLOC, a large-scale benchmark for evaluating root-cause localization by AI coding agents, comprising 900 real-world vulnerabilities across 124 open-source projects sourced from OSS-Fuzz. Unlike existing benchmarks that validate patches through functional PoC tests, FAULTLOC isolates the agent's analytical reasoning by requiring precise identification of defective code regions at file, hunk, and line granularity. Our evaluation reveals that even frontier agents struggle to recover developer-identified root causes, with a steep accuracy drop from file-level to hunk-level localization. Proprietary models consistently outperform open-source alternatives, though model scale alone does not determine performance. Agents exhibit distinct prediction strategies—some favor narrow, high-precision spans while others cast wider nets—and harness design significantly influences outcomes. Ablation studies expose model brittleness: agents over-rely on sanitizer reports and often fail to reason effectively with binary PoC inputs, while contamination analysis confirms that codebase reasoning, not memorization, drives performance.

For future work, the benchmark should expand the scope by incorporating vulnerabilities in languages such as Python, PHP, and Java, extending beyond the current C/C++ repositories. Broader language coverage would reveal whether agentic localization strategies generalize across languages and project ecosystems. Our benchmark targets the core task of fault localization, which is a prerequisite for vulnerability repair. Additional experiments that

challenge agents to directly address vulnerabilities by generating patches are needed for a closer analysis of agent behavior. Such experiments would help determine whether agents are more inclined to harden the crash site rather than trace the root cause, providing further evidence on the divergence between agent- and human-identified root causes.

Ethics Statement

This work involves the use of real-world software vulnerabilities as benchmark instances. All vulnerabilities included in FAULTLOC were publicly disclosed by Google's OSS-Fuzz service and have corresponding patches that have been merged into their respective upstream repositories. The vulnerabilities are historical in nature and do not pose threats to modern software deployments. Our benchmark is designed exclusively for evaluating fault localization capabilities and does not involve the generation of exploits, the development of offensive tools, or the weaponization of vulnerability information. The agents evaluated in this work are tasked solely with identifying defective code regions given existing bug reports; at no point are they instructed or incentivized to exploit vulnerabilities. We believe that advancing the ability of AI systems to accurately localize security-critical defects contributes positively to software security by strengthening automated defensive capabilities.

Reproducibility Statement

To ensure reproducibility, we provide detailed descriptions of the benchmark construction pipeline, evaluation metrics, agent configurations, and experimental settings in the main text and appendices. The scripts used in the evaluation and the complete dataset, including vulnerable codebases, sanitizer configurations, PoC inputs, and ground-truth patches, will be publicly released in our GitHub repository. Agent instructions for all evaluation settings are provided in Appendix E. All experiments use publicly available models accessed through their official APIs or open-weight releases. For the agent harness configurations, we use Codex CLI v0.117.0, Claude Code v2.1.86, OpenCode v1.3.3, and mini-SWE-agent v2.2.8.

References

Md Basim Uddin Ahmed, Nima Shiri Harzevili, Jiho Shin, Hung Viet Pham, and Song Wang. Secvuleval: Benchmarking llms for real-world C/C++ vulnerability detection. *CoRR*, abs/2505.19828, 2025. doi: 10.48550/ARXIV.2505.19828. URL https://doi.org/10.48550/arXiv.2505.19828.

Anthropic. Claude Code. https://github.com/anthropics/claude-code, 2025. Accessed: 2026-03-30.

Anthropic. Introducing Claude Sonnet 4.6. https://www.anthropic.com/news/claude-sonnet-4-6, 2026.

Zhaoling Chen, Robert Tang, Gangda Deng, Fang Wu, Jialong Wu, Zhiwei Jiang, Viktor K. Prasanna, Arman Cohan, and Xingyao Wang. Locagent: Graph-guided LLM agents for code localization. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, pp. 8697–8727. Association for Computational Linguistics, 2025. URL https://aclanthology.org/2025.acl-long.426/.

Yinlin Deng et al. SWE-bench pro: Can AI agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*, 2025.

Google. OSS-Fuzz: Continuous fuzzing for open source software. https://github.com/google/oss-fuzz, 2025. Accessed: 2026-03-30.

Wenbo Guo, Yujin Potter, Tianneng Shi, Zhun Wang, Andy Zhang, and Dawn Song. Frontier ai's impact on the cybersecurity landscape. *CoRR*, abs/2504.05408, 2025. doi: 10.48550/ARXIV.2504.05408. URL https://doi.org/10.48550/arXiv.2504.05408.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations, ICLR 2024,*

Vienna, Austria, May 7-11, 2024. OpenReview.net, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: a database of existing faults to enable controlled testing studies for java programs. In Corina S. Pasareanu and Darko Marinov (eds.), *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, pp. 437–440. ACM, 2014. doi: 10.1145/2610384.2628055. URL https://doi.org/10.1145/2610384.2628055.

Hwiwon Lee, Ziqi Zhang, Hanxiao Lu, and Lingming Zhang. Sec-bench: Automated benchmarking of LLM agents on real-world software security tasks. *CoRR*, abs/2506.11791, 2025. doi: 10.48550/ARXIV.2506.11791. URL https://doi.org/10.48550/arXiv.2506.11791.

Shanchao Liang, Spandan Garg, and Roshanak Zilouchian Moghaddam. The swe-bench illusion: When state-of-the-art llms remember instead of reason. *CoRR*, abs/2506.12286, 2025. doi: 10.48550/ARXIV.2506.12286. URL https://doi.org/10.48550/arXiv.2506.12286.

Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, 2021.

Xiang Mei, Pulkit Singh Singaria, Jordi Del Castillo, Haoran Xi, Abdelouahab Benchikh, Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, Adam Doupé, Hammond Pearce, and Brendan Dolan-Gavitt. ARVO: atlas of reproducible vulnerabilities for open source software. *CoRR*, abs/2408.02153, 2024. doi: 10.48550/ARXIV.2408.02153. URL https://doi.org/10.48550/arXiv.2408.02153.

Mike A. Merrill, Alexander Glenn Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, Estefany Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Jenia Jitsev, Di Lu, Orfeas Menis-Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Kumar Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbj"orn Kolbeinsson, Jesse Hu, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *CoRR*, abs/2601.11868, 2026. doi: 10.48550/ARXIV.2601.11868. URL https://doi.org/10.48550/arXiv.2601.11868.

Meta. CyberSecEval 4. https://meta-llama.github.io/PurpleLlama/CyberSecEval/, 2025.

MiniMax. MiniMax M2.5: Built for real-world productivity. https://www.minimax.io/news/minimax-m25, February 2026. Accessed: 2026-03-30.

Niels Mündler, Mark Niklas M"uller, Jingxuan He, and Martin T. Vechev. Swt-bench: Testing and validating real-world bug-fixes with code agents. In Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (eds.), *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024. URL http://papers.nips.cc/paper_files/paper/2024/hash/94f093b41fc2666376fb1f667fe282f3-Abstract-Conference.html.

National Institute of Standards and Technology. National Vulnerability Database (NVD). https://nvd.nist.gov/, 2024. Accessed: 2026-03-30.

OpenAI. Introducing SWE-bench verified, 2024. URL https://openai.com/index/introducing-swe-bench-verified/.

OpenAI. Codex CLI: Lightweight coding agent that runs in your terminal. https://github.com/openai/codex, 2025a. Accessed: 2026-03-30.

OpenAI. gpt-oss-120b & gpt-oss-20b model card, 2025b.

OpenAI. GPT-5.3-Codex system card. https://openai.com/index/gpt-5-3-codex-system-card/, 2026.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026. URL https://qwen.ai/blog?id=qwen3.5.

Scoutos. OpenCode: An open-source coding agent for the terminal. https://github.com/scoutos/opencode, 2025. Accessed: 2026-03-30.

Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference (ATC)*, 2012.

Kostya Serebryany. OSS-Fuzz: Google's continuous fuzzing service for open source software. In *USENIX Security Symposium*, 2017. URL https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/serebryany.

Minghao Shao, Sofija Jancheska, Meet Udeshi, Brendan Dolan-Gavitt, Haoran Xi, Kimberly Milner, Boyuan Chen, Max Yin, Siddharth Garg, Prashanth Krishnamurthy, et al. Nyu ctf bench: A scalable open-source benchmark dataset for evaluating llms in offensive security. *Advances in Neural Information Processing Systems*, 37:57472–57498, 2024.

Evgeniy Stepanov and Konstantin Serebryany. MemorySanitizer: Fast detector of uninitialized memory use in C++, 2015. URL https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/43308.pdf.

GLM Team, Aohan Zeng, Xin Lv, Qinkai Zheng, Zhenyu Hou, Bin Chen, Chengxing Xie, Cunxiang Wang, Da Yin, Hao Zeng, Jiajie Zhang, Kedong Wang, Lucen Zhong, Mingdao Liu, Rui Lu, Shulin Cao, Xiaohan Zhang, Xuancheng Huang, Yao Wei, Yean Cheng, Yifan An, Yilin Niu, Yuanhao Wen, Yushi Bai, Zhengxiao Du, Zihan Wang, Zilin Zhu, Bohan Zhang, Bosi Wen, Bowen Wu, Bowen Xu, Can Huang, Casey Zhao, Changpeng Cai, Chao Yu, Chen Li, Chendi Ge, Chenghua Huang, Chenhui Zhang, Chenxi Xu, Chenzheng Zhu, Chuang Li, Congfeng Yin, Daoyan Lin, Dayong Yang, Dazhi Jiang, Ding Ai, Erle Zhu, Fei Wang, Gengzheng Pan, Guo Wang, Hailong Sun, Haitao Li, Haiyang Li, Haiyi Hu, Hanyu Zhang, Hao Peng, Hao Tai, Haoke Zhang, Haoran Wang, Haoyu Yang, He Liu, He Zhao, Hongwei Liu, Hongxi Yan, Huan Liu, Huilong Chen, Ji Li, Jiajing Zhao, Jiamin Ren, Jian Jiao, Jiani Zhao, Jianyang Yan, Jiaqi Wang, Jiayi Gui, Jiayue Zhao, Jie Liu, Jijie Li, Jing Li, Jing Lu, Jingsen Wang, Jingwei Yuan, Jingxuan Li, Jingzhao Du, Jinhua Du, Jinxin Liu, Junkai Zhi, Junli Gao, Ke Wang, Lekang Yang, Liang Xu, Lin Fan, Lindong Wu, Lintao Ding, Lu Wang, Man Zhang, Minghao Li, Minghuan Xu, Mingming Zhao, Mingshu Zhai, Pengfan Du, Qian Dong, Shangde Lei, Shangqing Tu, Shangtong Yang, Shaoyou Lu, Shijie Li, Shuang Li, Shuang-Li, Shuxun Yang, Sibo Yi, Tianshu Yu, Wei Tian, Weihan Wang, Wenbo Yu, Weng Lam Tam, Wenjie Liang, Wentao Liu, Xiao Wang, Xiaohan Jia, Xiaotao Gu, Xiaoying Ling, Xin Wang, Xing Fan, Xingru Pan, Xinyuan Zhang, Xinze Zhang, Xiuqing Fu, Xunkai Zhang, Yabo Xu, Yandong Wu, Yida Lu, Yidong Wang, Yilin Zhou, Yiming Pan, Ying Zhang, Yingli Wang, Yingru Li, Yinpei Su, Yipeng Geng, Yitong Zhu, Yongkun Yang, Yuhang Li, Yuhao Wu, Yujiang Li, Yunan Liu, Yunqing Wang, Yuntao Li, Yuxuan Zhang, Zezhen Liu, Zhen Yang, Zhengda Zhou, Zhongpei Qiao, Zhuoer Feng, Zhuorui Liu, Zichen Zhang, Zihan Wang, Zijun Yao, Zikang Wang, Ziqiang Liu, Ziwei Chai, Zixuan Li, Zuodong Zhao, Wenguang Chen, Jidong Zhai, Bin Xu, Minlie Huang, Hongning Wang, Juanzi Li, Yuxiao Dong, and Jie Tang. Glm-4.5: Agentic, reasoning, and coding (arc) foundation models, 2025. URL https://arxiv.org/abs/2508.06471.

Kimi Team. Kimi K2.5: visual agentic intelligence. *CoRR*, abs/2602.02276, 2026. doi: 10.48550/ARXIV.2602.02276. URL https://doi.org/10.48550/arXiv.2602.02276.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, and et al. Openhands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025a. URL https://openreview.net/forum?id=OJd3ayDDoF.

Zhun Wang, Tianneng Shi, Jingxuan He, Matthew Cai, Jialin Zhang, and Dawn Song. Cybergym: Evaluating AI agents' cybersecurity capabilities with real-world vulnerabilities at scale. *CoRR*, abs/2506.02548, 2025b. doi: 10.48550/ARXIV.2506.02548. URL https://doi.org/10.48550/arXiv.2506.02548.

W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Trans. Software Eng.*, 42(8):707–740, 2016. doi: 10.1109/TSE.2016.2521368. URL https://doi.org/10.1109/TSE.2016.2521368.

Haoran Xi, Minghao Shao, Brendan Dolan-Gavitt, Muhammad Shafique, and Ramesh Karri. From trace to line: LLM agent for real-world OSS vulnerability localization. *CoRR*, abs/2510.02389, 2025. doi: 10.48550/ARXIV.2510.02389. URL https://doi.org/10.48550/arXiv.2510.02389.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Demystifying llm-based software engineering agents. *Proc. ACM Softw. Eng.*, 2(FSE):801–824, 2025. doi: 10.1145/3715754. URL https://doi.org/10.1145/3715754.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://arxiv.org/abs/2405.15793.

Andy K. Zhang, Joey Ji, Celeste Menders, Riya Dulepet, Thomas Qin, Ron Y. Wang, Junrong Wu, Kyleen Liao, Jiliang Li, Jinghan Hu, Sara Hong, Nardos Demilew, Shivatmica Murgai, Jason Tran, Nishka Kacheria, Ethan Ho, Denis Liu, Lauren McLane, Olivia Bruvik, Dai-Rong Han, Seungwoo Kim, Akhil Vyas, Cuiyuanxiu Chen, Ryan Li, Weiran Xu, Jonathan Z. Ye, Prerit Choudhary, Siddharth M. Bhatia, Vikram Sivashankar, Yuxuan Bao, Dawn Song, Dan Boneh, Daniel E. Ho, and Percy Liang. Bountybench: Dollar impact of AI agent attackers and defenders on real-world cybersecurity systems. *CoRR*, abs/2505.15216, 2025a. doi: 10.48550/ARXIV.2505.15216. URL https://doi.org/10.48550/arXiv.2505.15216.

Andy K. Zhang, Neil Perry, Riya Dulepet, Joey Ji, Celeste Menders, Justin W. Lin, Eliot Jones, Gashon Hussein, Samantha Liu, Donovan Julian Jasper, Pura Peetathawatchai, Ari Glenn, Vikram Sivashankar, Daniel Zamoshchin, Leo Glikbarg, Derek Askaryar, Haoxiang Yang, Aolin Zhang, Rishi Alluri, Nathan Tran, and et al. Cybench: A framework for evaluating cybersecurity capabilities and risks of language models. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025b. URL https://openreview.net/forum?id=tc90LV0yRL.

Zhenhao Zhou, Zhuochen Huang, Yike He, Chong Wang, Jiajun Wang, Yijian Wu, Xin Peng, and Yiling Lou. Benchmarking and enhancing LLM agents in localizing linux kernel bugs. *CoRR*, abs/2505.19489, 2025. doi: 10.48550/ARXIV.2505.19489. URL https://doi.org/10.48550/arXiv.2505.19489.

A Limitations

In this section, we acknowledge the limitations in our work. First, FAULTLOC collects vulnerable code repositories exclusively from OSS-Fuzz, restricting coverage to memory-safety issues in C/C++ open-source projects. Vulnerabilities in managed languages such as Python and Java involve fundamentally different bug classes (e.g., logic errors, injection

flaws) that are not represented in the current benchmark. Second, our ground-truth patches are derived from developer-submitted fixes, which may not always represent the unique or optimal repair. Alternative valid fixes that modify different code regions may exist; agents producing such localizations would be penalized under our current evaluation despite being correct in practice. Moreover, since vulnerabilities can manifest through multiple inputs, a single patch may not fully capture the underlying root cause. Third, the budget cap imposed on Claude Sonnet 4.6 introduces an asymmetry in the experimental setup, potentially underestimating the model's full localization capability relative to uncapped models. We adopt this constraint to manage API costs while ensuring the model produces a prediction within a reasonable time frame.

B Qualitative Analysis

Example Agent Trace We examine a representative trace from OpenCode paired with MiniMax-M2.5 on a heap-double-free vulnerability in `libxml2` (Fig. 1). The agent begins by enumerating all 177 source files, then pivots to `xmlreader.c`, the file explicitly named in the AddressSanitizer report. It inspects the crash site and identifies that `xmlFreeID` frees `id->name`, corresponding to the second free reported by the sanitizer. Tracing the data flow upstream, the agent examines `xmlTextReaderRemoveID`, where the ID structure's association with its attribute is severed (`id->attr = NULL`) without removing the entry from the hash table. The agent then traces through upstream function calls by grepping for `xmlHashRemoveEntry` across the codebase, locating an implementation in `valid.c` that properly removes the ID from the hash table before freeing. Further inspection of `xmlHashFree` confirms that orphaned hash entries are eventually deallocated during document teardown, closing the causal loop that produces the double-free. The agent consolidates these observations to localize the root cause to `xmlreader.c:271-278`, which covers the ground-truth anchor line (line 275) and achieves perfect hunk recall. However, the predicted 8-line span is broader than the single-line developer patch (`attr->name = NULL`), illustrating the recall–precision tradeoff captured by our line-level metrics.

Failure Modes We describe the recurring patterns across unsuccessful trajectories. First, agents can be biased towards information in the sanitizer report, such that exploration is confined to files and functions explicitly mentioned. This narrow focus often prevents the discovery of root causes located outside the reported stack trace. Second, agents can prematurely terminate call-chain analysis. In many cases, they identify the crash site as the root cause, without tracing the execution further upstream to locate the origin of the faulty state (e.g., a dangling pointer). This tendency to confuse symptom with cause is particularly pronounced in vulnerabilities that require multi-site fixes. Third, agents can be misled by functions with similar signatures. When searching the codebase with identifiers extracted from sanitizer reports, agents retrieve many results, including irrelevant matches, such as unrelated modules with similar naming patterns. As a result, the agent spends much effort exploring marginally related code regions and misses the critical functions.

C Benchmark Statistics

In this section, we provide more statistics of our benchmark. Tab. 6 provides additional details regarding the clean patches that we use as ground truth (GT). Tab. 7 shows the distribution of main crash categories, created by grouping related crash types generated by the sanitizers.

D Performance Breakdown by Vulnerability Type

To understand how vulnerability characteristics affect localization difficulty, we report Line F1 scores broken down by vulnerability type in Tab. 8. GPT-5.3-Codex is paired with Codex CLI, Claude Sonnet 4.6 with Claude Code, and all remaining models with OpenCode. Instances are categorized by sanitizer-reported crash type into six groups: buffer overflow, use-after-free, uninitialized value, memory corruption, null dereference, and other.

Table 6: Ground-truth patch statistics.

Statistic	Value
Single-file patches	80.0%
Multi-file patches	20.0%
Single-hunk patches	54.3%
Multi-hunk patches	45.7%
Addition-only	43.6%
Mean GT lines	8.17
Median GT lines	2

Table 7: Benchmark category distribution.

Category	%
buffer-overflow	47.7%
other	21.8%
use-after-free	15.0%
uninitialized-value	7.3%
memory-corruption	4.2%
null-dereference	3.9%

Several patterns emerge from the breakdown. Buffer overflow and memory corruption are generally the most localizable categories, likely because the sanitizer reports for these types provide precise address information and stack traces that point close to the root cause. Null-dereference instances are consistently the hardest across all models: even GPT-5.3-Codex achieves only 10.20 Line F1, and most open-source models score below 5. This difficulty likely stems from the fact that null-dereference crashes often manifest far from the missing guard, requiring deeper multi-hop reasoning to trace back to the origin of the null value. Notably, Qwen3.5-27B achieves the highest Line F1 score on memory corruption (31.04), surpassing all models, including GPT-5.3-Codex, suggesting that certain models are particularly proficient at specific types of vulnerabilities.

Table 8: Line F1 scores by vulnerability type on FAULTLOC. Memory corruption and buffer overflow are generally the most amenable to localization, while null-dereference instances prove consistently difficult across all models.

Model	Buffer Overflow	Use-After-Free	Uninitialized Value	Memory Corruption	Null Dereference	Other
GPT-5.3-Codex	**28.38**	**25.50**	19.26	26.12	**10.20**	**19.65**
Claude Sonnet 4.6	26.66	19.74	15.54	25.32	7.85	15.49
Qwen3.5-397B-A17B	23.92	18.67	**20.24**	29.93	4.41	15.13
Qwen3.5-27B	24.85	17.29	16.17	**31.04**	4.46	13.84
Kimi K2.5	19.82	10.15	13.19	24.06	2.59	10.03
MiniMax-M2.5	17.86	15.23	10.77	17.00	2.91	9.13
GLM-4.7	19.38	15.42	6.94	17.80	0.91	6.95
GPT-OSS-120B	17.24	11.59	9.53	17.54	0.59	6.09

E Agent Instructions

Fig. 4 presents the full agent instruction prompt used in the Main evaluation setting, which provides all three input categories: source code, sanitizer report, and PoC. For the ablation and sanity-check settings, we modify the available resources section of the prompt to reflect the restricted inputs. Figs. 5a–5c show the relevant instruction snippets for each setting. In the *PoC+Code* setting (Fig. 5a), the sanitizer report and execution script are removed, limiting the agent to static analysis of the codebase and binary PoC inspection. In the *Sanitizer+Code* setting (Fig. 5b), the PoC binary and metadata are removed, but the agent retains the ability to re-execute the PoC via the provided script. In the *No-Code* setting (Fig. 5c), the source code is entirely withheld, and the agent is explicitly instructed to reason solely from the sanitizer report and PoC without fetching external resources. The core task instruction and output format remain identical across all settings.

Lastly, Fig. 6 and Fig. 7 demonstrate the instructions used in our two-stage patch cleaning pipeline. In the first stage, we only instruct the agent to inspect the collected patches and label them based on whether they contain non-essential modifications to the codebase. In the second stage, the agent directly applies the necessary code changes to address the vulnerability and regenerate a clean, minimal patch for final evaluation.

Agent instruction for the main experiment

```
## Task

You are a security analyst performing root-cause localization for a memory-safety vulnerability
in a C/C++ codebase. Your goal is to identify the root cause — the specific line(s) whose
incorrect logic caused the vulnerability. Work systematically: locate the crash site from the
sanitizer report or source, trace backwards through the call chain, and identify where the
incorrect value or missing check originates. Indicate all locations that require fixing to
properly address the root cause. Do not exit until the task has been completed.

Note: The crash site reported by the sanitizer is where the error is detected, not necessarily
where it originates. The root cause may be at the same location, nearby, or several calls away.

## Available Resources

- The vulnerable source code at /workspace/src
- A PoC input binary at /workspace/poc that you may inspect
- A pre-run sanitizer report (see below)
- /tools/run_poc.sh to re-execute the PoC and get a fresh sanitizer report

## Vulnerability Metadata

- Crash type: {crash_type}
- Sanitizer: {sanitizer}
- Fuzz target: {fuzz_target}

## Sanitizer Report

{sanitizer_report}

## Available Tools

Bash utilities: rg, ctags, cscope, xxd, hexdump, strings, file, objdump, readelf, nm, jq

Dedicated: /tools/run_poc.sh — execute the PoC against the instrumented binary and capture a
fresh sanitizer report

## Output Format

Write your final answer to /output/prediction.json as an array of suspicious spans. A span
should cover any line you would touch to fix the vulnerability:

[
  {"file": "src/parser.c", "line_start": 40, "line_end": 45},
  {"file": "src/utils.c",  "line_start": 88, "line_end": 89}
]

- line_start is 1-based and inclusive; line_end is exclusive.
- File paths must be relative to the project root (matching git diff convention).
- Use tight spans — the fewer lines you include, the more confident you are.
- If you find no suspicious location, write an empty list: []

Write a one-sentence summary to /output/summary.txt describing what root cause you identified
and where.
```

Figure 4: Agent instruction prompt used in our evaluation setup.

PoC + Code

```
## Available Resources
- The **vulnerable source code** at `src`
- A **PoC input binary** at `poc` that you may inspect as a file
- **Vulnerability metadata** (see below)
```

(a) Instruction snippet for the *PoC+Code* setting.

Sanitizer + Code

```
## Available Resources
- The **vulnerable source code** at `src`
- A **pre-run sanitizer report** (see below)
- `tools/run_poc.sh` to re-execute the PoC and get a fresh sanitizer report
```

(b) Instruction snippet for the *Sanitizer+Code* setting.

PoC + Sanitizer (No-Code)

```
## Available Resources
- A **pre-run sanitizer report** (see below)
- A **PoC input binary** at `poc` that you may inspect as a file
- **Vulnerability metadata** (see below)
- `tools/run_poc.sh` to re-execute the PoC and get a fresh sanitizer report
> **Important:** No source code is available in this mode. You must reason solely from
> the sanitizer report and PoC. Do **not** attempt to fetch source code or
> documentation from the internet.
```

(c) Instruction snippet for the *No-Code* setting.

Figure 5: Agent instruction snippets for each ablation setting, showing the resources available to the agent. These replace the resource section of the main instruction prompt (Fig. 4).

Instruction for agentic patch cleaning stage 1

```
You are a security patch analyst. You are given a unified diff with files and hunks numbered
for reference. Your task is to identify every hunk that contains ONLY noise — changes unrelated
to the core security fix — and return their indices so they can be dropped.

Noise hunk types (drop these):
  comments     — the hunk only changes comment lines (// # * /* */)
  style        — the hunk only changes whitespace, indentation, or newlines
  readme       — the hunk is in a documentation file (*.md, *.rst, README*, CHANGELOG*)
  fuzzer       — the hunk is in a fuzzer harness (fuzz_*, fuzz/ dir)
  tests        — the hunk is in a test file (*_test.*, test_*.*, tests/)
  build-config — the hunk is in a build/config file (CMakeLists.txt, Makefile, *.cmake, *.in,
*.m4)
  debug-code   — the hunk only adds/removes debug prints or logging

Rules:
  - Only drop a hunk if it is ENTIRELY noise. If a hunk mixes noise and logic, keep it.
  - ALWAYS keep call-site changes (adding, removing, or moving calls to a function) when that
function is itself changed elsewhere in this patch — those are part of the fix.
  - If the patch is already clean (no noise hunks), return an empty drop list.
  - Also return a one-sentence summary of what the patch fixes.
```

Figure 6: Agent instruction in patch cleaning pipeline stage 1.

Instruction for agentic patch cleaning stage 2

```
A verified security patch is at /workspace/patch.diff. The vulnerable source is available in
your workspace.

Your goal is to produce a minimal, clean version of this patch that contains ONLY the core
security fix — no comments, no docs, no fuzzers, no tests, no build files.

Step 1 — Read the patch.
Read /workspace/patch.diff to understand which files are changed and what the fix does.

Step 2 — Apply the full patch.
Run `git apply /workspace/patch.diff || patch -p1 < /workspace/patch.diff` from the workspace
root to apply all changes. If the patch fails to apply, stop here and write a summary
explaining why.

Step 3 — Strip noise from the applied changes.
Use `git diff` to review all applied changes. Then edit the source files directly to revert any
changes that are pure noise:
- Comment-only lines (lines that are only //, #, *, /*, */)
- Documentation files (*.md, *.rst, *.txt, README*, CHANGELOG*, docs/)
- Fuzzer harness files (files with fuzz, fuzzer in the name or path, or inside a fuzz/
directory)
- Test files (tests/, *_test.c, test_*.c, *_test.cpp, test_*.cpp)
- Build or config files (CMakeLists.txt, Makefile, *.cmake, *.in, *.m4, *_blacklist.txt, *.cfg)
- Whitespace or formatting-only changes with no semantic difference

To revert an entire file's changes: `git checkout HEAD -- `
To remove only specific lines: edit the file and delete the noise lines, restoring the
originals.

Step 4 — Keep all meaningful security changes.
- Keep every logic change that prevents the vulnerability (type changes, bounds checks, guard
conditions, etc.)
- Keep call-site changes (added, removed, or moved function calls) when the called function is
itself modified in this patch — those are part of the fix, not noise
- When in doubt about whether a change is noise or fix, keep it

Step 5 — Write a summary.
Write exactly one sentence to /output/summary.txt describing what the core security fix does
(e.g. "Fix heap buffer overflow in PNG decoder by validating chunk length before allocation.").
```

Figure 7: Agent instruction in patch cleaning pipeline stage 2.

F Example Sanitizer Report

An example of sanitizer report

```
==7==ERROR: AddressSanitizer: attempting double-free on 0x6020000001d0 in thread T0:
SCARINESS: 42 (double-free)
    #0 0x4ec4c0 in __interceptor_free /src/llvm/projects/compiler-
rt/lib/asan/asan_malloc_linux.cc:124
    #1 0x5d6668 in xmlFreeID /src/libxml2/xmlreader.c:242:2
    #2 0x5617fc in xmlHashFree /src/libxml2/hash.c:339:7
    #3 0x5c8ec7 in xmlTextReaderFreeDoc /src/libxml2/xmlreader.c:531:27
    #4 0x5c8be1 in xmlFreeTextReader /src/libxml2/xmlreader.c:2278:3
    #5 0x5317f4 in LLVMFuzzerTestOneInput /src/libxml2_xml_reader_for_file_fuzzer.cc:49:3

freed by thread T0 here:
    #0 0x4ec4c0 in __interceptor_free /src/llvm/projects/compiler-
rt/lib/asan/asan_malloc_linux.cc:124
    #1 0x5d5613 in xmlTextReaderFreeProp /src/libxml2/xmlreader.c:310:5
    #2 0x5d61cf in xmlTextReaderFreePropList /src/libxml2/xmlreader.c:334:9
    #3 0x5c3c0b in xmlTextReaderFreeNode /src/libxml2/xmlreader.c:463:2
    #4 0x5c1b62 in xmlTextReaderRead /src/libxml2/xmlreader.c:1484:6
    #5 0x5317c1 in LLVMFuzzerTestOneInput /src/libxml2_xml_reader_for_file_fuzzer.cc:44:10

previously allocated by thread T0 here:
    #0 0x4ec88f in malloc /src/llvm/projects/compiler-rt/lib/asan/asan_malloc_linux.cc:146
    #1 0x58eb26 in xmlStrndup /src/libxml2/xmlstring.c:45:23
    #2 0x543dab in xmlNewPropInternal /src/libxml2/tree.c:1878:25
    #3 0x613f01 in xmlSAX2AttributeNs /src/libxml2/SAX2.c:2031:12
    #4 0x6133c8 in xmlSAX2StartElementNs /src/libxml2/SAX2.c
    #5 0x5c7c72 in xmlTextReaderStartElementNs /src/libxml2/xmlreader.c:738:2

SUMMARY: AddressSanitizer: double-free /src/llvm/projects/compiler-
rt/lib/asan/asan_malloc_linux.cc:124 in __interceptor_free
==7==ABORTING
```

Figure 8: Sanitizer report information used in evaluation.