SecCVE: Can Large Language Models Really Reason About Vulnerabilities?

Anonymous ACL submission

Abstract

Vulnerability analysis is a fundamental yet challenging task for large language models. Although recent code-oriented LLMs have achieved strong performance on snippet-level vulnerability benchmarks, it remains unclear whether they can reason about vulnerabilities at the repository level. In particular, models may identify code that appears vulnerable locally, yet fail to determine whether the vulnerable behavior is reachable from external entry points in practice. To address this gap, we introduce **SecCVE**, a benchmark for evaluating repository-level vulnerability reasoning. SecCVE is constructed through an automated pipeline that extracts real-world vulnerabilities from existing CVE repositories. For each CVE, we align vulnerable and fixed snapshots, map security patch hunks to enclosing functions as evidence anchors, and expand through the repository to enumerate candidate entry-to-evidence call chains. We then apply constrained LLM-based pruning guided by the CVE description, followed by AST-based structural validation and manual auditing. This process provides reliable supervision by connecting external entry points to vulnerability fixes. SecCVE supports three quantitative tasks: (1) vulnerability localization, identifying vulnerable locations in a repository; (2) entry-point inference, predicting external entry files given a vulnerability sink; and (3) chain reconstruction, generating a faithful call-chain explanation from entry point to fix. The benchmark covers 6,156 CVEs from 2,455 open-source projects across 6 programming languages, with 4,688 instances providing audited entry-to-fix supervision. We will release the complete benchmark, including all datasets, construction pipelines, and evaluation scripts.

1 Introduction

In modern software development, vulnerability analysis is essential for maintaining the security and reliability of complex systems (Gonçalves et al., 2025). Recent LLMs have shown strong potential across a wide range of programming tasks, including code generation (Chen, 2021), debugging (Chen et al., 2023a), and program repair (Huang et al., 2025). They have also achieved promising results on vulnerability benchmarks, suggesting the ability to recognize common insecure coding patterns and security-related code changes.

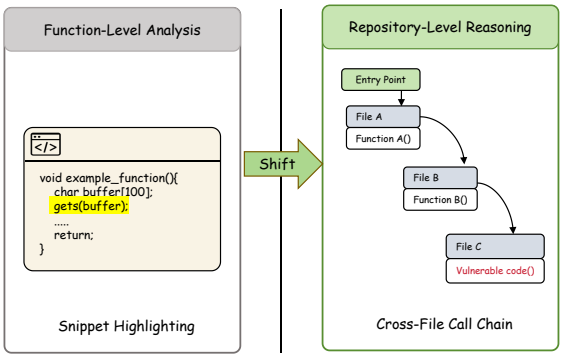

Figure 1: SecCVE reformulates vulnerability analysis from function-level to cross-file chain.

However, a key mismatch remains between existing evaluations and real-world security auditing. Most benchmarks reward models for locating code that appears vulnerable in isolation, whereas practical vulnerability analysis requires determining whether the vulnerable behavior is actually reachable from attacker-controlled entry points. In production systems, the security impact of a code fragment depends not only on its local syntax (Wei et al., 2022), but also on whether an external attacker can trigger it through entry points such as API routes, RPC handlers, or CLI commands, and propagate execution through cross-file call chains. Existing benchmarks (Black, 2018; Fan et al., 2020) typically focus on single snapshots, functions, or localized annotations, where the goal is to identify suspicious lines or files within a narrow scope. As a result, they provide limited support for evaluating repository-scale reasoning over entry points,

inter-procedural dependencies, and cross-module reachability. This creates a blind spot: models may generate explanations that are plausible yet executionally impossible. Standardized evaluation for repository-scale vulnerability reasoning, especially **Entry-to-Fix** consistency, therefore remains limited.

To address this gap, we introduce **SecCVE**, a benchmark for evaluating repository-level vulnerability reasoning in LLMs. SecCVE is constructed from real-world CVEs through an automated data collection and verification pipeline. For each CVE, we align the vulnerable and fixed versions of the corresponding repository and use the security patch as an anchor for identifying vulnerability-related code. Specifically, we map patch hunks to their enclosing functions and treat these functions as evidence anchors. Starting from these patch-related functions, we expand through the repository to enumerate statically plausible entry-to-fix call chains. Since static call graphs are often noisy and over-approximate possible executions, this expansion can produce many spurious paths. We therefore apply a constrained LLM-based pruning step guided by the CVE description and repository evidence. The selected paths are then validated using AST-based structural checks over call relations, and ambiguous cases are further manually audited. This refine-and-verify process proposes higher-fidelity entry-to-fix supervision.

SecCVE supports three quantitative tasks that capture complementary aspects of vulnerability reasoning. The first task, vulnerability localization, evaluates whether a model can retrieve files that need to be modified given a CVE and a repository. The second task, entry-point inference, assesses whether a model can identify externally reachable entry files given a vulnerability evidence sink. The third task, end-to-end chain reconstruction, requires models to generate a faithful call-chain explanation from an entry point to the vulnerability fix. SecCVE covers 6,156 CVEs from 2,455 open-source projects across 6 programming languages, with 4,688 instances providing audited entry-to-fix supervision. By grounding evaluation in real-world vulnerabilities and repository-level structure, SecCVE provides a practical testbed for studying the limitations and capabilities of LLMs beyond isolated vulnerability snippets.

The main contributions of our paper are as follows:

- We identify a mismatch between existing vulnerability benchmarks and real-world security auditing, and formalize repository-scale Entry-to-Fix reasoning as a missing evaluation dimension for large language models.
- We introduce **SecCVE**, a benchmark constructed from real-world CVEs that provides entry-to-fix supervision by combining patch-based evidence anchoring, repository-level call-chain expansion, constrained LLM pruning, structural validation, and manual auditing.
- We define three evaluation tasks, vulnerability localization, entry-point inference, and end-to-end chain reconstruction—and systematically evaluate LLMs under these settings, showing that strong localization performance does not necessarily translate into faithful entry-to-fix vulnerability reasoning.

2 Related works

In this section, we provide an overview of related work on vulnerability benchmarks, repository-level program analysis and LLM-based methods, highlighting their relevance and the fundamental differences between existing approaches and our proposed "entry-to-fix" reasoning frame work.

2.1 Vulnerability Benchmarks

Traditional vulnerability benchmarks typically formulate detection as a function-level classification problem. Datasets such as Big-Vul (Fan et al., 2020), Devign (Zhou et al., 2019), and Diverse-Vul (Chen et al., 2023b) focus on isolated functions or small code snippets, making them suitable for learning local vulnerability patterns. Megavul (Ni et al., 2024) further improves benchmark reliability by addressing noisy labels, duplicated samples, and unrealistic data splits. LineVul (Fu and Tantithamthavorn, 2022) studies finer-grained localization by identifying vulnerable lines within functions. However, these benchmarks mainly focus on local code regions, which differs from practical security analysis, which often need to inspect code in repository context and reason about interactions across components.

Recent work has introduced repository-level or CVE-linked datasets to better reflect real-world workflows. ReposVul (Wang et al., 2024) leverages static analysis to recover inter-procedural de-

pendencies and associate vulnerable code with related functions in the same repository. JITVul (Yildiz et al., 2025) links vulnerable functions to specific commits, incorporating repository history and code changes as additional signals. VulDetectBench (Liu et al., 2024) evaluates the vulnerability detection capability of LLMs under different settings. While these efforts provide richer context than function-level benchmarks, they usually do not offer structured supervision connecting external entry points, such as routes or handlers, to vulnerable code.

SecCVE targets this missing information by introducing Entry-to-Fix supervision. Each instance connects an external entry point with the corresponding vulnerability fix, enabling evaluation of whether a model can not only localize the code, but also trace a structurally valid path from the entry point to the patched hunk.

2.2 Repository-Level Program Analysis

Repository-level vulnerability analysis remains challenging because it requires reasoning over long call chains that span multiple modules and functions. To address this challenge, VulChecker (Mirsky et al., 2023) proposes a graph-based framework that encodes repository structure for vulnerability detection. GRACE (Lu et al., 2024) incorporates graph structure into LLM-based vulnerability detection, demonstrating the value of structural code information. For vulnerability localization, CoSIL (Jiang et al., 2025) leverages module-level call graphs to narrow the search space of potential bug locations. More recently, tool-augmented LLM systems such as AgentFL (Qin et al., 2024) and LocAgent (Chen et al., 2025) have explored repositories iteratively or with graph-guided reasoning for code localization. RepoAudit (Guo et al., 2025) further investigates autonomous LLM agents for repository-level code auditing.

While these methods make repository-scale reasoning more feasible, they also reveal a practical bottleneck: static call graphs are often noisy and overly dense, which can cause path explosion and introduce many spurious call chains. Consequently, simply enumerating paths from a suspicious function does not necessarily capture real reachability. Our work addresses this limitation by combining constrained LLM-based pruning with AST-based structural validation, thereby producing higher-fidelity call chains for reachability evaluation.

2.3 LLMs for Vulnerability Localization and Reasoning

Recent progress in reasoning-capable LLMs has encouraged vulnerability research to move beyond local pattern recognize. LLaVul (Jararweh et al., 2025) generates human-readable explanations for reported vulnerabilities. JITVul (Yildiz et al., 2025) benchmarks LLMs and LLM-based agents on practical vulnerability detection in code repositories. From Trace to Line (Xi et al., 2025) uses execution traces to help LLM agents localize vulnerable lines in real open-source projects. SEC-bench (Lee et al., 2026) evaluates LLM agents on real-world software security tasks, showing the growing need for realistic security-focused evaluation.

Despite these advances, one common limitation remains. LLMs can often produce convincing reasoning explanations, but the call chains they describe may not be reachable in the actual program. SecCVE addresses this problem by providing entry-to-fix paths that are manually checked and structurally validated. This makes it possible to evaluate whether model reasoning is consistent with feasible execution paths.

3 Method

As shown in Figure 2, the SecCVE benchmark includes a unified construction pipeline and three evaluation tasks for repository-level vulnerability reasoning. The pipeline builds audited entry points and entry-to-fix call chains from real-world CVEs, which are then used as supervision for evaluation. Based on this data, SecCVE defines three tasks: localization, entry-point inference, and end-to-end chain reconstruction. Each task focuses on a different aspect of repository-scale reasoning.

3.1 Data Construction Pipeline

Patch Extraction Given a repository

$$\mathcal{R} = (\mathcal{F}, \mathcal{U}, \mathcal{E}, \phi), \tag{1}$$

where $\mathcal{F}$ denotes the set of source files, $\mathcal{U}$ denotes the set of code units, such as functions or methods, $\mathcal{E} \subseteq \mathcal{U} \times \mathcal{U}$ denotes a directed call graph, and $\phi : \mathcal{U} \to \mathcal{F}$ maps each code unit to its enclosing file.

We collect CVE-to-repository mappings from GitHub Advisory records and retain cases whose vulnerability-fixing commit messages explicitly contain the corresponding CVE identifier. From these commits, we obtain the vulnerable snapshot

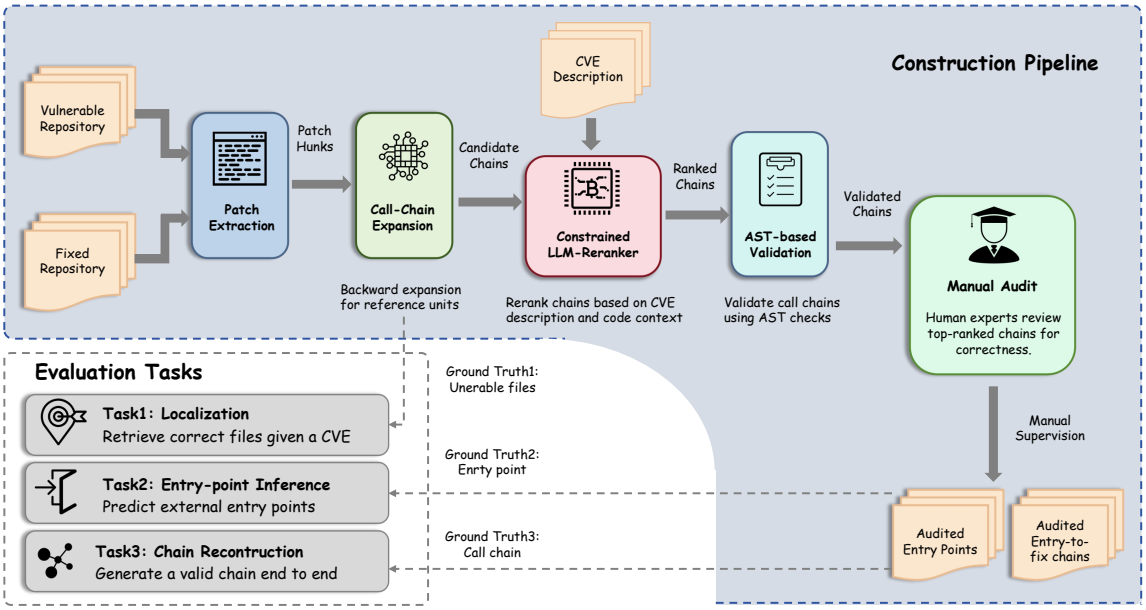

Figure 2: SecCVE construction pipeline and evaluation tasks. We align vulnerable and fixed snapshots to extract security patches, map them to enclosing functions, and expand inter-procedural call chains to construct candidate Entry-to-Fix paths. After LLM-based pruning, AST validation, and expert audit, we obtain verified entry points and call chains. SecCVE defines three tasks: Localization, Entry-point Inference, and Chain Reconstruction.

and the corresponding fixed snapshot. Given a vulnerable snapshot $\mathcal{R}^{\text{vul}}$ and a fixed snapshot $\mathcal{R}^{\text{fix}}$, we compute the patch diff and extract a set of patch hunks $\mathcal{H}$.

To avoid benchmark leakage and simple string matching, we clean the CVE descriptions by removing or replacing repository-specific names, such as file names, function names, API names, and module names. Therefore, models must rely on repository structure and code reasoning instead of memorizing public CVE reports.

Call-chain Expansion Each $h \in \mathcal{H}$ is mapped to its enclosing unit $u(h) \in \mathcal{U}$. We define the set of reference vulnerability units as

$$\mathcal{S} = u(h) \mid h \in \mathcal{H} \subseteq \mathcal{U}. \tag{2}$$

$\mathcal{S}$ serves as a patch-anchored reference for evaluation. We use $\mathcal{T} \subseteq \mathcal{U}$ to denote the set of candidate external entry units, such as request handlers, CLI entry functions, and exported APIs. A call chain is a directed path in the call graph:

$$\pi = (u_0, u_1, \ldots, u_\ell) \tag{3}$$

Starting from $\mathcal{S}$, we enumerate candidate Entry-to-Fix call chains by expanding outward in the call graph. Let $\text{Pred}(u) = v \in \mathcal{U}$ denote the set of predecessor callers of u. We perform a bounded backward expansion to collect units that can reach the patched units. We then enumerate candidate paths Π_{raw} that end in $\mathcal{S}$ and start from the candidate external entry set $\mathcal{T}$. This set is obtained through a simple call-graph traversal. Its goal is to cover reachable paths as much as possible, while accepting that many of the resulting call chains may be spurious.

Constrained LLM Pruning The candidate set Π_{raw} can be large and noisy. We use minimax-2.7 as a constrained LLM ranker to filter more realistic trigger paths. For each candidate chain $\pi \in \Pi_{\text{raw}}$, we build a compact evidence bundle which includes: (i) the CVE description d, (ii) function signatures and brief summaries of the units on the chain, and (iii) local context around the patch-related units. minimax-2.7 is only allowed to score or rank the given candidates, rather than generate new chains or modify call-chain structures. We then select the top R candidates for the next processing stage.

$$\Pi_{\text{prune}} = \text{TopR}\left[\text{Score}_{\text{LLM}}\big(\pi \mid \text{Ev}(\pi)\big)\right]. \tag{4}$$

This design reduces hallucinated call-chain structures by restricting minimax-2.7 (MiniMax, 2026) to candidate ranking and encourages the model to

rank candidates based on textual evidence, such as the CVE description, and repository-level signals.

Bias Control in LLM-assisted Labeling. LLM-assisted labeling may introduce model-specific bias if the same model is used both to construct candidate labels and to perform downstream evaluation. To reduce this risk, SecCVE uses minimax-2.7 for the pruning stage, rather than any of the evaluated frontier models.

AST-based Validation The repository-level call graphs are built with language-specific static analysis frontends, which extract function or method units, import dependencies, class or module structure, and call expressions. Since these graphs are over-approximations, LLM-based pruning may still keep chains with incorrect call edges. Therefore, we validate each predicted edge $(u \rightarrow v)$ with AST-based call-site checking on the vulnerable version of the code. Specifically, we verify the called functions, upstream and downstream call relationships, and scope reachability, ensuring that each candidate call chain is syntactically valid. Call chains that violate these structural constraints are removed or truncated.

Manual Audit for Ambiguous Cases Some CVEs may have more than one reasonable entry point or more than one feasible path. In these cases, static checking alone may not be sufficient to identify the correct path. For all audited instances, five vulnerability experts conduct cross-review on the top-ranked and structurally valid candidates in Π, based on the CVE description and the repository context. Each audited instance is independently reviewed by two experts, and disagreements are resolved by a third expert. The five experts have prior experience in vulnerability analysis. The audit focuses on whether the selected external entry points are valid and whether the corresponding call chains correctly connect the entry points to the vulnerability fixes. This process produces: (i) an entry set $\mathcal{T}^\star$, and (ii) one or more accepted chains $\Pi^\star$. Thus the ground truth for reachability reasoning is reliable and of high quality.

3.2 Evaluation Tasks and Metrics

We evaluate frontier LLMs (Qwen, 2025; GLM, 2026; Moonshot AI, 2026; OpenAI, 2026; Anthropic, 2026) on repository-level vulnerability reasoning using the CVE descriptions and repository context from SecCVE. The experiments directly evaluate whether models can reason over Entry-to-Fix consistency, including external entry points and inter call chains. The input context is task-specific: vulnerability localization uses the CVE description and repository file tree, while entry-point inference and chain reconstruction additionally use the patch-anchored sink information. All reported metrics are computed automatically against ground truth. We provide detailed mathematical formulations and scoring derivations in the appendix.

3.2.1 Benchmark Instance

Each benchmark instance is defined as

$$x = (\mathcal{R}^{\text{vul}}, d, \mathcal{S}, \mathcal{T}^\star, \Pi^\star), \tag{5}$$

where $\mathcal{R}^{\text{vul}} = (\mathcal{F}, \mathcal{U}, \mathcal{E}, \phi)$ denotes the vulnerable repository snapshot. Here, $\mathcal{F}$ is the file tree, $\mathcal{U}$ is the set of code units, $\mathcal{E}$ is the static call graph, and ϕ maps each code unit to its enclosing file. The CVE description is denoted by d, and $\mathcal{S}$ is the patch-anchored sink set. We define the vulnerable file set as

$$\mathcal{F}^\star = \{\phi(u) \mid u \in \mathcal{S}\}. \tag{6}$$

For audited instances, we additionally provide manually verified external entry points $\mathcal{T}^\star$ and end-to-end call chains $\Pi^\star$.

3.2.2 Task1: Vulnerable File Localization

Given the repository file tree $\mathcal{F}$ and the CVE description, the model outputs a ranked list of files. We report **R-loc@K**, a top-K hit metric that checks whether at least one ground-truth vulnerable file in $\mathcal{F}^\star$ appears among the top-K predicted files. The metric is computed per instance and then averaged over the benchmark. It reflects whether the model can localize vulnerability-relevant regions of the repository.

3.2.3 Task2: Entry-point Inference

Given $(\mathcal{R}^{\text{vul}}, d, \mathcal{S})$, the model predicts a ranked list of external entry points, represented at the file level. We evaluate these predictions against the audited entry files derived from $\mathcal{T}^\star$. We report **R-entry@K**, which is satisfied if any audited entry file appears among the top-K predicted entry files. This metric is averaged over audited instances and measures whether the model can identify at least one valid external entry into the vulnerable flow.

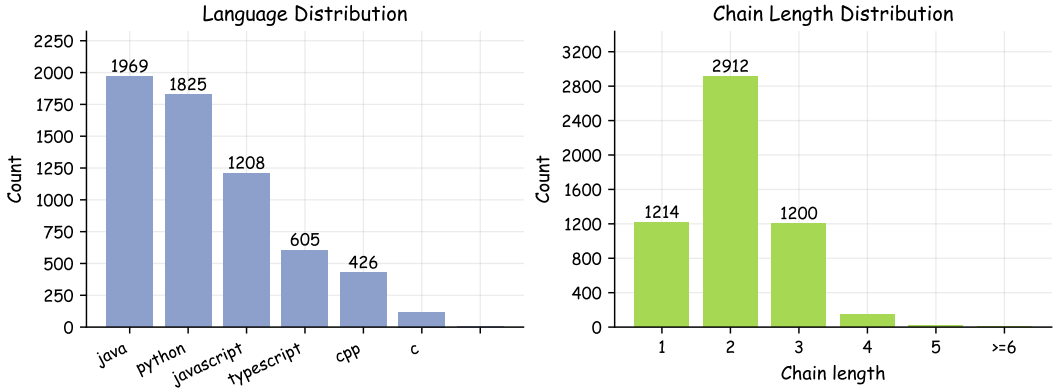

Figure 3: Statistics of the SecCVE dataset. The left plot shows the programming-language distribution of the collected CVEs, covering 6 languages with Java and Python as the 2 largest categories. The right plot presents the distribution of vulnerability chain lengths among audited instances, where most CVEs require two-step or three-step chain reconstruction, reflecting the multi-hop nature of real-world security fixes.

3.2.4 Task 3: Chain Reconstruction

Given $(\mathcal{R}^{\text{vul}}, d, \mathcal{S})$, the model generates a ranked list of call chains from external entry points to patch-anchored sinks. Each chain is converted into an ordered sequence of files and evaluated against the audited chain set $\Pi^{\star}$.

To account for partial correctness and structural ambiguity, we use a soft chain similarity score, **Sim-chain**, between a predicted chain and a ground-truth chain. This score combines two signals: an order-sensitive common-prefix score and an order-insensitive file-set overlap score:

$$\text{Sim-chain} = \alpha \cdot \text{Prefix} + (1-\alpha) \cdot \text{Jaccard} \quad (7)$$

A predicted chain is considered matched if its score to any audited chain is at least a threshold τ.

We report three chain-level metrics. **Cov-chain@τ** measures the fraction of audited ground-truth chains that are covered by at least one predicted chain. **R-chain@K@τ** is a top-K hit metric that checks whether any of the top-K predicted chains matches an audited chain under threshold τ. **MRR-chain@K@τ** measures how early the first sufficiently matched chain appears in the ranked list. All chain metrics are averaged over audited instances.

4 Benchmarking LLMs on SecCVE

In this section, we report detailed statistics of the SecCVE dataset and evaluate the performance of multiple frontier large language models on this benchmark, with further analysis of the results.

4.1 Dataset Statistics

We collect **6156** real-world CVEs from **2455** open-source repositories using GitHub Advisory records and CVE-bearing vulnerability-fixing commits, covering **6** programming languages and a wide range of application domains, with detailed statistics shown in Figure 3. SecCVE supports three evaluation tasks: entry-point inference over **5920** instances, and end-to-end chain reconstruction over the **4688** audited instances.

4.2 Experimental results

The evaluation results for Task1, Task2, and Task3 are reported as follows.

4.3 Can LLMs Reason about Vulnerabilities?

Our evaluation on SecCVE leads to three key findings that may challenge the common assumptions about LLM-based vulnerability understanding.

Finding 1: High localization accuracy overestimates true vulnerability reasoning ability. Results on Task 1 show that LLMs achieve strong performance in vulnerable file localization, with several models exceeding 0.5 on **R-location@1**. However, this apparent success does not imply true vulnerability reasoning. File localization mainly relies on semantic matching between CVE descriptions and code artifacts, without requiring models to reason about how a vulnerability is triggered or propagated. The sharp performance drop from Task 1 to Task 2 indicates that identifying where vulnerable code resides is substantially easier than inferring how an attacker can reach it. This suggests that prior benchmarks focusing solely on lo-

Model	Task 1: File Loc.		Task 2: Entry Inf.		Task 3: Chain Recon.		
	R-loc@1	R-loc@3	R-ent@1	R-ent@3	Cov@0.8	R@3@0.8	MRR@3@0.8
Non-Thinking Models							
Qwen3.6	0.278	0.352	0.169	0.178	0.042	0.051	0.011
GLM-5.1	0.379	0.391	0.292	0.315	0.063	0.030	0.009
Kimi-K2.6	0.442	0.476	0.258	0.381	0.070	0.053	0.012
DeepSeek-V4	0.458	0.515	0.402	0.487	0.112	0.059	**0.052**
Gemini-3.1	0.541	0.558	0.423	0.445	0.161	**0.123**	**0.063**
GPT-5.5	**0.636**	**0.678**	**0.506**	**0.532**	**0.195**	0.120	0.060
Thinking Models							
Qwen3.6	0.443	0.500	0.304	0.389	0.104	0.079	0.032
DeepSeek-V4	0.536	0.616	0.438	0.510	0.119	0.083	0.069
Kimi-K2.6	0.595	0.624	0.528	0.581	0.154	0.096	0.052
Claude-opus-4.6	0.670	0.729	**0.555**	0.613	**0.250**	**0.190**	**0.1531**
Gemini-3.1	0.677	0.763	0.507	0.580	0.201	0.136	0.071
GPT-5.5	**0.807**	**0.869**	0.535	**0.647**	0.236	0.143	0.121

Table 1: Results on SecCVE for Task 1 (Vulnerable File Localization), Task 2 (Entry-point Inference), and Task 3 (End-to-End Chain Reconstruction). Best results are in bold and second-best are underlined.

K	R-chain@K@τ				MRR-chain@K@τ			
	$\tau = 0.2$	$\tau = 0.4$	$\tau = 0.6$	$\tau = 0.8$	$\tau = 0.2$	$\tau = 0.4$	$\tau = 0.6$	$\tau = 0.8$
1	0.246	0.142	0.103	0.084	0.114	0.095	0.054	0.024
3	0.284	0.216	0.133	0.120	0.154	0.122	0.078	0.060
5	0.318	0.257	0.161	0.143	0.169	0.129	0.087	0.076
8	**0.351**	**0.293**	**0.177**	**0.198**	**0.183**	**0.141**	**0.097**	**0.084**

Table 2: Ablation over K and τ for R-chain and MRR-chain. All experiments are based on GPT-5.5.

calization may significantly overestimate LLMs' security reasoning capabilities.

Finding 2: Reasoning models help, but do not close the gap to repository-level understanding. Models equipped with explicit reasoning mechanisms consistently outperform non-thinking models across all tasks, confirming the importance of multi-step reasoning for security analysis. Nevertheless, the gains diminish as tasks require reasoning across multiple functions and files. While reasoning models show clear improvements on entry-point inference (Task 2), their advantage narrows on end-to-end exploit chain reconstruction (Task 3). Even the strongest models achieve limited coverage and low reciprocal ranks, indicating that current reasoning models remain insufficient for modeling realistic exploit paths spanning multiple functions and files.

Finding 3: End-to-end chain reconstruction exposes a fundamental capability gap. Task 3 shows a clear weakness in current evaluation settings. Even when models are given the same CVE descriptions and repository context, they rarely generate call chains that match the audited Entry-to-Fix paths. The consistently low **Cov-chain@0.8** scores show that many ground-truth chains are not covered by any predicted chain, even with a relatively loose matching threshold. This suggests that current LLMs do not understand program execution as a complete process. Instead, they often combine locally reasonable code fragments that do not form a valid exploit chain. As a result, strong performance on snippet-level or file-level benchmarks does not transfer well to repository-level vulnerability reasoning. This limitation is also supported by our ablation study on the Task 3 evaluation settings. As shown in Table 4, increasing the number

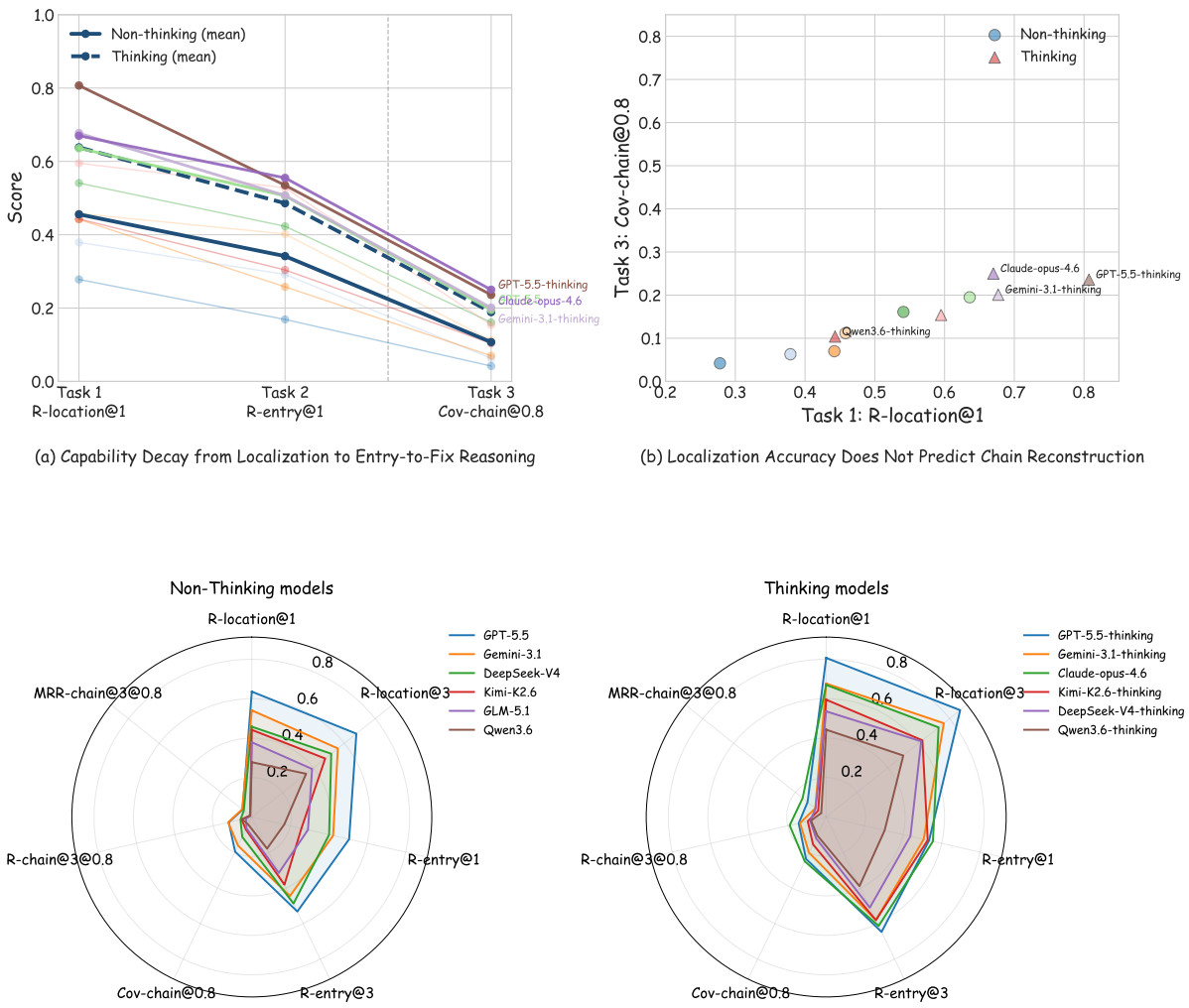

Figure 4: SecCVE benchmark analysis and model performance comparison. Top: reasoning capability decay from Task 1 to Task 3, and the correlation between file localization accuracy and chain reconstruction performance. Bottom: comparison between non-thinking and thinking LLMs across the three SecCVE tasks.

of predicted chains K improves both recall and ranking metrics, while stricter matching thresholds τ reduce performance. However, even with large K values and loose τ settings, the overall results are still limited. These findings show that the low coverage and ranking quality in Task 3 are not caused by overly strict evaluation rules. Instead, they reflect a real weakness of current models in reconstructing coherent end-to-end exploit paths.

5 Acknowledgment

Large language models were used only for language polishing and writing assistance. All technical design, experiments, analysis, and conclusions were completed and verified by the authors.

6 Conclusion

This paper introduces SecCVE, a benchmark for evaluating entry-to-fix vulnerability reasoning in real-world repositories. By providing structured supervision across file localization, entry-point inference, and end-to-end chain reconstruction, SecCVE exposes limitations of existing snippet-level evaluations. Experiments show that while current models perform well on localization, their performance drops sharply on multi-hop vulnerability reasoning, even with thinking-enabled architectures. These results highlight a persistent gap between surface-level pattern recognition and repository-level reasoning. We hope SecCVE will facilitate more realistic evaluation and drive progress toward robust vulnerability understanding in large codebases.

Limitations

SecCVE has several limitations. First, although the benchmark is built from real-world CVEs and diverse repositories, its supervision quality still depends on the availability and correctness of vulnerability-fixing commits, CVE descriptions, and repository snapshots. Second, the entry-to-fix chains are constructed using static call-graph expansion, LLM-based pruning, AST validation, and manual auditing. While this improves reliability, it cannot fully capture dynamic dispatch, runtime behavior, data-flow constraints, or environment-dependent exploitability. Third, SecCVE focuses on source-level reachability from external entry points to patched code, and does not directly evaluate other important security tasks such as exploit generation, patch correctness, vulnerability severity, or runtime verification. Finally, the current benchmark only covers selected languages, repositories, and prompting settings. Future work can extend SecCVE to broader software ecosystems and incorporate stronger dynamic analysis signals.

References

Anthropic. 2026. Claude opus 4.6. https://www.anthropic.com. Accessed: 2026-05-21.

Paul E Black. 2018. A software assurance reference dataset: Thousands of programs with known bugs. *Journal of research of the National Institute of Standards and Technology*, 123:1.

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

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023a. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*.

Yizheng Chen, Zhoujie Ding, Lamya Alowain, Xinyun Chen, and David Wagner. 2023b. 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.

Zhaoling Chen, Robert Tang, Gangda Deng, Fang Wu, Jialong Wu, Zhiwei Jiang, Viktor Prasanna, Arman Cohan, and Xingyao Wang. 2025. Locagent: Graph-guided llm agents for code localization. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8697–8727.

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.

Michael Fu and Chakkrit Tantithamthavorn. 2022. Linevul: A transformer-based line-level vulnerability prediction. In *Proceedings of the 19th international conference on mining software repositories*, pages 608–620.

GLM. 2026. Glm-5: from vibe coding to agentic engineering. *arXiv preprint arXiv:2602.15763*.

José Gonçalves, Miguel Silva, Bernardo Cabral, Tiago Dias, Eva Maia, Isabel Praça, Ricardo Severino, and Luís Lino Ferreira. 2025. Evaluating llama 3.2 for software vulnerability detection. In *European Interdisciplinary Cybersecurity Conference*, pages 38–51. Springer.

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

Kai Huang, Jian Zhang, Xinlei Bao, Xu Wang, and Yang Liu. 2025. Comprehensive fine-tuning large language models of code for automated program repair. *IEEE Transactions on Software Engineering*.

Ala Jararweh, Michael Adams, Avinash Sahu, Abdullah Mueen, and Afsah Anwar. 2025. Llavul: A multimodal llm for interpretable vulnerability reasoning about source code. In *2025 5th Intelligent Cybersecurity Conference (ICSC)*, pages 232–241. IEEE.

Zhonghao Jiang, Xiaoxue Ren, Meng Yan, Wei Jiang, Yong Li, and Zhongxin Liu. 2025. Cosil: Software issue localization via llm-driven code repository graph searching. *arXiv preprint arXiv:2503.22424*.

Hwiwon Lee, Ziqi Zhang, Hanxiao Lu, and Lingming Zhang. 2026. Sec-bench: Automated benchmarking of llm agents on real-world software security tasks. *Advances in Neural Information Processing Systems*, 38:116342–116378.

Yu Liu, Lang Gao, Mingxin Yang, Yu Xie, Ping Chen, Xiaojin Zhang, and Wei Chen. 2024. Vuldetectbench: Evaluating the deep capability of vulnerability detection with large language models. *arXiv preprint arXiv:2406.07595*.

Guilong Lu, Xiaolin Ju, Xiang Chen, Wenlong Pei, and Zhilong Cai. 2024. Grace: Empowering llm-based software vulnerability detection with graph structure and in-context learning. *Journal of Systems and Software*, 212:112031.

MiniMax. 2026. Minimax language models. https://www.minimaxi.com. Accessed: 2026-05-21.

Yisroel Mirsky, George Macon, Michael Brown, Carter Yagemann, Matthew Pruett, Evan Downing, Sukarno Mertoguno, and Wenke Lee. 2023. {VulChecker}: Graph-based vulnerability localization in source code. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6557–6574.

Moonshot AI. 2026. Kimi-k2.6. https://platform.moonshot.ai. Accessed: 2026-05-21.

Chao Ni, Liyu Shen, Xiaohu Yang, Yan Zhu, and Shaohua Wang. 2024. Megavul: Ac/c++ vulnerability dataset with comprehensive code representations. In *Proceedings of the 21st International Conference on Mining Software Repositories*, pages 738–742.

OpenAI. 2026. Gpt-5.5. https://platform.openai.com. Accessed: 2026-05-21.

Yihao Qin, Shangwen Wang, Yiling Lou, Jinhao Dong, Kaixin Wang, Xiaoling Li, and Xiaoguang Mao. 2024. Agentfl: Scaling llm-based fault localization to project-level context. *arXiv preprint arXiv:2403.16362*.

Qwen. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

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

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Haoran Xi, Minghao Shao, Brendan Dolan-Gavitt, Muhammad Shafique, and Ramesh Karri. 2025. From trace to line: Llm agent for real-world oss vulnerability localization. *arXiv preprint arXiv:2510.02389*.

Alperen Yildiz, Sin G Teo, Yiling Lou, Yebo Feng, Chong Wang, and Dinil Mon Divakaran. 2025. Benchmarking llms and llm-based agents in practical vulnerability detection for code repositories. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 30848–30865.

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.