# EVIREPAIR: EVIDENCE-DRIVEN AGENTS FOR RELIABLE VULNERABILITY REPAIR

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Recent advances in large language models (LLMs) have enabled agent-based approaches for repository-level automated vulnerability repair (AVR), showing promising repair capabilities. However, an important issue has received limited attention: **how can we ensure that the evidence generated during the repair process is reliable enough to support accurate vulnerability understanding and patch generation**? In this paper, we propose EVIREPAIR, an evidence-driven multi-agent framework for repository-level AVR. EVIREPAIR introduces evidence review into the repair process by enabling agents to collaboratively collect vulnerability evidence, verify evidence consistency, and construct a reviewed vulnerability model for patch synthesis and validation. Extensive experiments on PatchEval and SecBench benchmarks with multiple agent frameworks and LLMs demonstrate that EVIREPAIR consistently outperforms existing agent-based baselines, achieving average improvements of 16.37% and 40.33% in resolved rates on PatchEval and SecBench, respectively. Codes are available at `https://anonymous.4open.science/r/EviRepair-BBB2/`.

## 1 INTRODUCTION

Automated vulnerability repair (AVR) (Li et al., 2025) aims to automatically generate patches for software vulnerabilities and has become an important research problem in software security and automated program repair. Early AVR approaches mainly rely on predefined repair templates (Huang et al., 2019; Li et al., 2025), program analysis techniques (Gao et al., 2021; Zhang et al., 2022), or search-based strategies (Le Goues et al., 2012; Li et al., 2025), which often struggle with real-world vulnerabilities requiring deep understanding of program semantics and repository-level context. As a result, practical vulnerability repair still heavily relies on human expertise, leading to substantial manual effort and limited scalability. Recent advances in large language models (LLMs) (Pearce et al., 2023; Fu et al., 2022) have opened new possibilities for AVR. By leveraging their code understanding and generation capabilities, LLM-based agents have demonstrated the potential to perform repository-level vulnerability repair with reduced human intervention.

Despite recent advances in LLM-based AVR systems, generating reliable patches remains challenging. A successful repair requires not only code changes but also evidence that the changes address the underlying vulnerability. However, existing single-agent approaches (Yu et al., 2025; Liu et al., 2025; Kulsum et al., 2024) perform vulnerability analysis, patch generation, and validation within the same workflow. Consequently, diagnostic evidence is directly reused for repair decisions without a dedicated review stage, allowing incorrect or incomplete vulnerability understanding to propagate through the repair process. Recent multi-agent approaches (Zhang et al., 2026b;a; Liu et al., 2026b; Fu et al., 2026) introduce specialized roles to improve repair reliability. However, these frameworks directly aggregate intermediate outputs from different agents. They do not verify evidence consistency before patch generation or use verified evidence for patch validation. As a result, inconsistent or incomplete findings from different agents may be propagated into patch generation. Therefore, existing agent-based AVR frameworks **generally do not explicitly validate evidence consistency throughout the repair process**.

In this paper, we propose EVIREPAIR, an evidence-driven multi-agent framework for repository-level AVR. EVIREPAIR addresses the lack of evidence validation by introducing evidence review into both vulnerability diagnosis and patch synthesis. It decomposes the repair process into two



Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

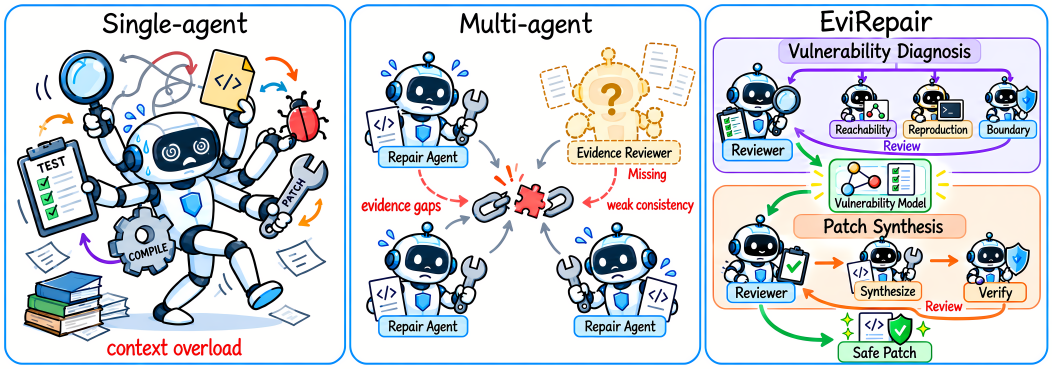



Figure 1: Comparison of single-agent, conventional multi-agent, and EviRepair for automated vulnerability repair. Single-agent methods face context overload by centralizing all tasks, while conventional multi-agent methods omit explicit evidence consistency checks. EviRepair employs multiple agents to alleviate the context burden on individual agents and introduces review agents to ensure consistency.

stages: *Vulnerability Diagnosis* and *Patch Synthesis*, connected through a reviewed *Vulnerability Model*. Given a vulnerability report, the *Analysis Coordinator* in *Vulnerability Diagnosis* assigns complementary analysis tasks to specialized agents to collect vulnerability evidence, including code reachability, reproducibility, security boundaries, and affected behaviors. An Evidence Consistency Review Agent then verifies whether the collected evidence consistently explains the reported vulnerability and determines whether additional analysis is required before constructing the final *Vulnerability Model*. Based on this reviewed model, *Patch Synthesis* plans and generates patches, followed by a patch review against the established security and compatibility criteria.

We conduct extensive experiments on PatchEval (Wei et al., 2025) and SecBench (Lee et al., 2025) to evaluate the effectiveness of EVIREPAIR. We compare EVIREPAIR with two mainstream agent frameworks, OpenCode (Anomaly, 2026) and Codex (OpenAI, 2026) , using two representative LLMs, GLM-5.2 (Z.AI, 2026) and DeepSeek-4.1-Flash (DeepSeek, 2026). Across different agent frameworks and underlying models, EVIREPAIR consistently achieves higher resolved rates than the evaluated baselines, with average improvements of 16.37% and 40.33% on PatchEval and SecBench, respectively. We further perform ablation studies to analyze the impact of different design components in EVIREPAIR and investigate their contributions to the overall repair performance.

Our contributions are summarized as follows:

- We propose EVIREPAIR, an evidence-driven multi-agent framework for repository-level automated vulnerability repair.
- We design evidence review mechanisms for vulnerability diagnosis and patch synthesis, which validate vulnerability understanding and patch correctness, respectively.
- We conduct extensive experiments demonstrating that EVIREPAIR consistently outperforms existing agent-based baselines across different frameworks and LLMs.

## 2 PROBLEM FORMULATION

Let $R$ denote a vulnerability report, $P$ denote the target repository, and $E_0$ denote the initially available evidence. In realistic settings, $E_0$ may contain only a natural-language description of the suspected vulnerability. Therefore, the information required for reliable patching must first be reconstructed from $R$, $P$, and $E_0$.

We represent the repair-critical evidence as

$$E = \{E_{\text{reach}}, E_{\text{repro}}, E_{\text{oracle}}, E_{\text{boundary}}, E_{\text{preserve}}\},$$

where the five components describe source-to-sink reachability, executable vulnerability reproduction, runtime observations and expected post-repair behavior, the security boundary, and legitimate

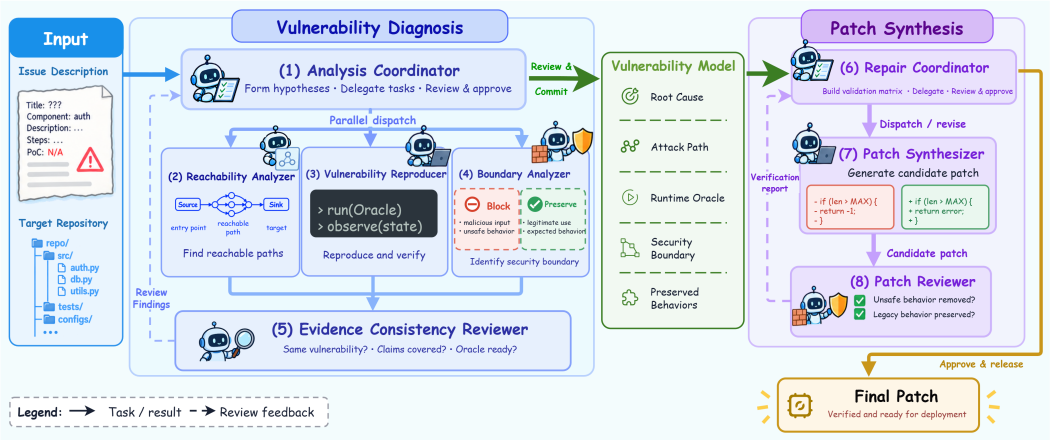



Figure 2: Overview of EVIREPAIR, an evidence-driven multi-agent framework for AVR. Vulnerability Diagnosis gathers complementary vulnerability evidence via specialized agents and validates their consistency to construct a Vulnerability Model. Patch Synthesis treats this model as the specification to generate, validate, and refine candidate patches.

behaviors that should be preserved, respectively. These components may be unavailable, incomplete, noisy, or mutually inconsistent. The repair task is to reconstruct a vulnerability model

$$M = f(R, P, E_0, E)$$

and synthesize a patch $\Delta$ that blocks the reported unsafe behavior while preserving legitimate program behavior.

# 3 EVIREPAIR

## 3.1 FRAMEWORK OVERVIEW

As shown in Figure 2, EVIREPAIR is organized as two coordinated multi-agent stages with distinct responsibilities: *Vulnerability Diagnosis* and *Patch Synthesis*. *Vulnerability Diagnosis* (Section 3.2) collects evidence about the reported vulnerability through independent specialized agents, reviews the evidence for consistency, and assembles it into the Vulnerability Model $\mathcal{M}$ (Section 3.3). *Patch Synthesis* (Section 3.4) treats $\mathcal{M}$ as a fixed specification, producing a patch and verifying it against $\mathcal{M}$. Each agent runs in its own context, and agents exchange results only through structured records with fixed fields.

## 3.2 VULNERABILITY DIAGNOSIS

Vulnerability Diagnosis takes a sparse vulnerability report and its target repository as input, and produces the Vulnerability Model $\mathcal{M}$ that guides patch synthesis. Since the report states what is suspected rather than what has been observed, diagnosis treats every claim in it as a hypothesis to be verified, and builds $\mathcal{M}$ only from evidence collected in the repository.

**Hypothesis formation.** The ANALYSIS COORDINATOR first reads the report, explores the repository, and forms security hypotheses, each naming a suspected entry point, sink, and vulnerability class. It then dispatches three specialized agents in parallel. Each agent receives the report, the repository, and the hypotheses, and works in its own context without access to the others' findings, so that each investigates one dimension of the vulnerability without interference.

**Evidence collection.** The three agents collect complementary evidence. The REACHABILITY ANALYZER examines whether attacker-controllable input can propagate from the entry point to the sink, and reports the path together with the transformations applied along it. The VULNERABILITY REPRODUCER builds an executable reproducer and a runtime oracle that characterizes the vulnerable behavior before repair and the expected behavior after repair, and runs them on the original code.

The BOUNDARY ANALYZER derives, from the code around the sink and the vulnerability class, the inputs that must be blocked and the legitimate behaviors that must be retained. Every finding is returned together with the evidence that supports it, such as code locations, analysis results, or execution logs, so that each claim can be checked rather than taken on trust.

**Review and refinement.** The EVIDENCE CONSISTENCY REVIEWER then decides whether these findings are reliable enough to enter $\mathcal{M}$. It checks, partly by re-executing the reproducer and the oracle, that all findings refer to the same vulnerability, that together they cover the claims in the report, and that the oracle is ready for use. Two rules govern this judgment: *agreement among agents is not evidence*, and *a claim supported only by the report is never confirmed*. When a check fails, the EVIDENCE CONSISTENCY REVIEWER does not simply reject the findings; it returns a targeted request that names the failed check, the conflicting evidence, and what is missing. The ANALYSIS COORDINATOR re-dispatches only the responsible agents with this feedback, while retaining the evidence already verified. Diagnosis thus refines its own findings through repeated rounds of review and targeted investigation, until all checks pass or the budget is exhausted. The verified findings are then assembled into $\mathcal{M}$.

### 3.3 VULNERABILITY MODEL

Once the Evidence Consistency Reviewer approves, the Analysis Coordinator assembles the validated findings into the Vulnerability Model $\mathcal{M}$, the only diagnostic output passed to Patch Synthesis. $\mathcal{M}$ consists of five fields:

- **Root Cause:** the location and mechanism of the flaw, with its supporting evidence.
- **Attack Path:** produced by the Reachability Analyzer, the path along which attacker-controllable input travels from the entry point to the sensitive operation, including the transformations applied along the way.
- **Runtime Oracle:** produced by the Vulnerability Reproducer, an executable reproducer together with a check that separately defines what counts as reproduction before patching and as safe behavior after patching.
- **Security Boundary:** produced by the Boundary Analyzer, the violating inputs and their variants that must be blocked after repair.
- **Preserved Behaviors:** produced by the Boundary Analyzer, the legitimate inputs and compatibility requirements that the patch must not break.

The last two fields define the acceptance criteria of Patch Synthesis: a patch is accepted only if it removes the unsafe behavior and preserves the legacy behavior. The full schema is given in Appendix A.1.

### 3.4 PATCH SYNTHESIS

Patch Synthesis turns $\mathcal{M}$ into a verified patch. Every patch is judged against $\mathcal{M}$: it must block the inputs in the Security Boundary and keep the behaviors in the Preserved Behaviors. Since $\mathcal{M}$ has already passed review, this stage does not revise the diagnosis; failures are resolved by revising the patch.

**Validation matrix.** The REPAIR COORDINATOR first translates $\mathcal{M}$ into a validation matrix, the executable form of its acceptance criteria. Each row is a case with an expected outcome: the violating inputs and variants from the Security Boundary must be blocked, the legitimate inputs and compatibility requirements from the Preserved Behaviors must behave as before, and basic checks confirm that the patch applies cleanly and the project still builds. Claims that diagnosis could not confirm are also recorded in the matrix as unverified cases, so that the remaining uncertainty is passed on to repair explicitly instead of being silently ignored. The REPAIR COORDINATOR also issues a repair directive that locates the root cause and lists two to four plausible fixes, preferring the one that changes the fewest legitimate behaviors.

**Patch generation.** Following the directive, the PATCH SYNTHESIZER implements a single patch that fixes the root cause rather than filtering the specific reproducer input, so that variants along the same attack path are also blocked. It then executes every row of the validation matrix on the patched

repository and records the observed outcomes. Like the diagnostic agents, it therefore returns its result together with the runtime evidence that supports it.

**Verification and refinement.** TThe PATCH REVIEWER operates in an isolated context with access to $\mathcal{M}$, the completed validation matrix, and the patch diff, but not to the reasoning of the PATCH SYNTHESIZER, so that its judgement remains independent. It approves the patch only if every row matches its expected outcome and the patch is consistent with $\mathcal{M}$: it fixes the flaw at the root cause, removes the unsafe behavior, and preserves the legacy behavior. Passing the matrix alone is not sufficient, since the matrix is built by the system itself; the legitimate inputs and variants guard against patches that merely suppress the observed trigger or reject valid input. When verification fails, the PATCH REVIEWER reports the failing rows and violated conditions, and the REPAIR COORDINATOR directs the PATCH SYNTHESIZER to revise the patch with this feedback. The cycle repeats until the PATCH REVIEWER approves or the budget is exhausted.

# 4 EXPERIMENTS

Our research questions are as follows:

**RQ1** Does EVIREPAIR outperform representative agentic coding systems on vulnerability repair across different models and benchmarks?

**RQ2** To what extent does the performance of EVIREPAIR vary across vulnerability categories?

**RQ3** Is the advantage of EVIREPAIR maintained as repair difficulty increases?

**RQ4** To what extent does each stage of the two-stage multi-agent pipeline contribute to the overall repair performance?

**RQ5** Can structured evidence compensate for the limited reasoning capability of smaller models?

## 4.1 EXPERIMENTAL SETUP

**Baselines.** We compare EVIREPAIR against two representative agentic coding systems, Codex v0.153.2 (OpenAI, 2026) and OpenCode v1.17.7 (Anomaly, 2026). All systems use the same underlying LLM, and we vary the backbone among DeepSeek-V4.1-Flash (DeepSeek, 2026), GLM-5.2 (Z.AI, 2026), and Qwen3.8-27B (Qwen Team, 2026). All systems run in the same sandboxed environment with identical tool access. For our framework, the two stages have separate time limits: vulnerability diagnosis is limited to 3 hours, and patch synthesis is limited to 2 hours. For OpenCode and Codex, we use a unified total time limit of 5 hours per task.

**Benchmarks.** We evaluate on PatchEval and SEC-bench (Wei et al., 2025; Lee et al., 2025). PatchEval contains 230 CVE repair tasks in Go (83), Python (70), and JavaScript (77), each evaluated dynamically with hidden tests. The agent receives only the vulnerability description and the target repository, without an executable PoC. SEC-bench contains 300 real-world security tasks, including 200 CVE tasks and 100 OSS-Fuzz tasks; its patching tasks additionally provide sanitizer output as auxiliary evidence. The two benchmarks thus cover a description-only setting and an evidence-assisted setting. Network access is disabled during repair, so that no system can retrieve the ground-truth fix from external sources.

**Metrics.** Following the official protocols of both benchmarks, we report the resolved rate (%Resolved), i.e., the fraction of tasks whose generated patch passes the benchmark's hidden tests. SEC-bench is evaluated under its default medium evaluation setting. Beyond overall numbers, we break down results by programming language on PatchEval (Go / Python / JavaScript) and by task family on SEC-bench (CVE / OSS-Fuzz). For the type- and difficulty-level analyses in RQ2 and RQ3, we average each system's resolved rate over the backbones evaluated for that system to reflect system-level behavior.

## 4.2 RQ1: EFFECTIVENESS OF EVIREPAIR

Table 1 shows that EVIREPAIR achieves the highest overall resolved rate on both benchmarks under all three backbones. Averaged over the backbones, it resolves 79.13% of PatchEval tasks, compared

Table 1: Comparison of %Resolved for vulnerability repair across different agent configurations and backbones on PatchEval and SEC-bench. Numbers in parentheses indicate the total number of vulnerability samples in each subset.

| Agent Config. | Backbone | PatchEval | | | | SEC-bench | | |
|---|---|---|---|---|---|---|---|---|
| | | Go (83) | Python (70) | JavaScript (77) | Overall (230) | CVE (200) | OSS-Fuzz (100) | Overall (300) |
| Codex (OpenAI, 2026) | DeepSeek-4.1-Flash | 74.70% | 81.43% | 79.22% | 78.26% | 64.50% | 90.00% | 73.00% |
| | GLM-5.2 | 65.06% | 64.29% | 72.73% | 67.39% | 49.00% | 88.00% | 62.00% |
| | Qwen3.8-27B | 55.42% | 71.43% | 67.53% | 64.35% | 41.50% | 75.00% | 52.67% |
| OpenCode (Anomaly, 2026) | DeepSeek-4.1-Flash | 85.54% | 91.43% | **87.01%** | 87.83% | 88.00% | 94.00% | 90.00% |
| | GLM-5.2 | 49.40% | 64.29% | 70.13% | 60.87% | 71.00% | 90.00% | 77.33% |
| | Qwen3.8-27B | 66.27% | 65.71% | 68.83% | 66.96% | 77.50% | 93.00% | 82.67% |
| EviRepair | DeepSeek-4.1-Flash | **91.57%** | **94.29%** | 81.82% | **89.13%** | **95.50%** | **98.00%** | **96.33%** |
| | GLM-5.2 | 69.88% | 77.14% | 72.73% | 73.04% | 93.00% | 96.00% | 94.00% |
| | Qwen3.8-27B | 75.90% | 78.57% | 71.43% | 75.22% | 90.00% | 87.00% | 89.00% |

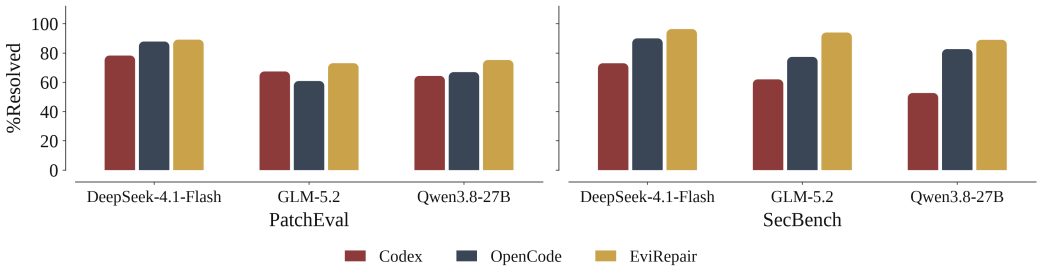



Figure 3: Resolved rate on PatchEval and SEC-bench across three backbones. The numbers above the bars indicate the absolute improvement of EviRepair over each baseline.

with 70.00% for Codex and 71.89% for OpenCode, and 93.11% of SEC-bench tasks, compared with 62.56% and 83.33%. Relative to the stronger baseline, OpenCode, this reduces the fraction of unresolved tasks by about 26% on PatchEval and 59% on SEC-bench.

The advantage holds consistently across backbones and subsets. On PatchEval, EviRepair exceeds the strongest baseline by 1.30, 5.65, and 8.26 percentage points with DeepSeek-4.1-Flash, GLM-5.2, and Qwen3.8-27B; on SEC-bench, the margins are 6.33, 16.67, and 6.33 points. Across the 21 comparisons between systems on the same backbone and subset, EviRepair achieves the best result in 18 and ties in one; the only exceptions are JavaScript with DeepSeek-4.1-Flash and OSS-Fuzz with Qwen3.8-27B, where OpenCode performs better.

The largest gains appear on tasks that require understanding the vulnerability rather than locating a crash. On the CVE subset of SEC-bench, EviRepair averages 92.83%, outperforming Codex by 41.17 points and OpenCode by 14.00 points, whereas on the OSS-Fuzz subset all systems already perform strongly and the margins are smaller. OSS-Fuzz tasks come with sanitizer reports that pinpoint the crashing location, so little additional evidence is needed. CVE tasks instead require establishing the attack path and the security boundary from the report, which is precisely the evidence that Vulnerability Diagnosis reconstructs.

### 4.3 RQ2: Performance across vulnerability types

We group tasks by vulnerability category (Figure 4). Categories with more than 10 tasks are shown individually, and the remaining categories are merged into *Other*. For PatchEval, we use the primary CWE in the official metadata; for SEC-bench, we use the primary weakness assigned by NVD for CVE tasks and the root-cause category in the official OSS-Fuzz crash reports.

On PatchEval, the largest gains appear on authorization flaws, namely incorrect authorization (CWE-863) and improper authorization (CWE-285), followed by path manipulation (CWE-73) and open redirect (CWE-601). Repairing these flaws requires deciding precisely which requests should be

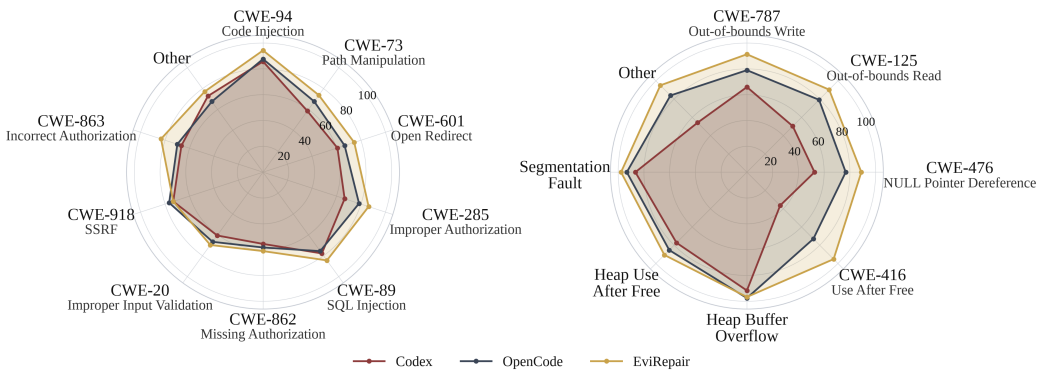



Figure 4: Resolved rate across vulnerability categories on PatchEval and SEC-bench. Each system is averaged over the three backbones.

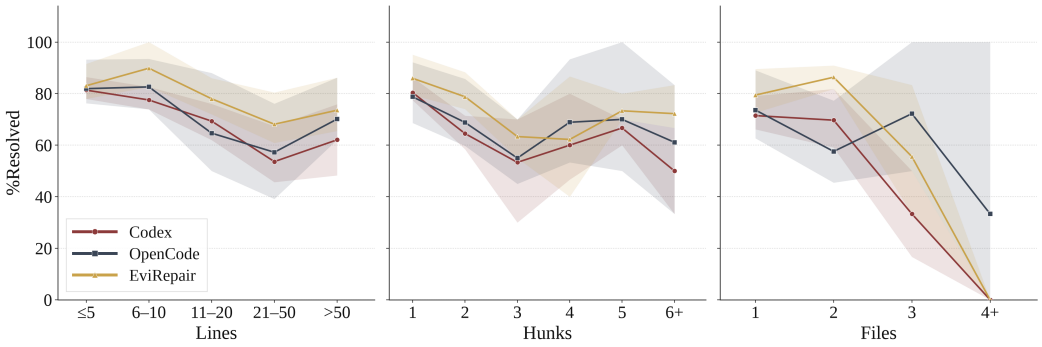



Figure 5: Resolved rate across repair difficulty dimensions on PatchEval: number of changed lines, hunks, and files per ground-truth patch. Lines show the average over three backbones; shaded regions indicate the min–max range.

permitted and which rejected, which is the security boundary that the BOUNDARY ANALYZER establishes before any patch is written. In contrast, all systems perform similarly on code injection (CWE-94) and SQL injection (CWE-89), whose fixes follow well-known patterns, and EVIREPAIR shows no advantage on server-side request forgery (CWE-918). A likely reason is that reproducing SSRF requires outbound requests, which cannot be observed in our network-isolated sandbox, so the runtime oracle for this category is weaker.

On SEC-bench, EVIREPAIR leads in every category. The gains are largest on use-after-free (CWE-416) and NULL pointer dereference (CWE-476), and smallest on heap buffer overflow, where all systems perform strongly. For temporal and null-state errors, the crash reported by the sanitizer typically occurs far from the root cause, such as the point where an object is freed or a pointer is left unset, and locating that point requires tracing the attack path. For buffer overflows, the crash site usually coincides with the location to fix, so the sanitizer report alone already provides most of the necessary evidence.

Across both benchmarks, the pattern is consistent: EVIREPAIR helps most when the location of the fix differs from where the vulnerability manifests, or when the fix depends on an explicit boundary between permitted and forbidden behavior.

### 4.4 RQ3: PERFORMANCE ACROSS REPAIR DIFFICULTY

The advantage of EVIREPAIR is largely maintained as repair difficulty increases, with the exception of the few repairs that span many files (Figure 5). All systems degrade as ground-truth patches grow larger, but EVIREPAIR degrades more slowly. On the lines dimension, it leads or ties in every bin, and the margin is widest on larger patches: in the 21–50 bin, it reaches roughly 72%, compared with about 56% for the baselines. On the hunks dimension, it leads in all bins except the 4-hunk

Table 2: Ablation study on the two-stage multi-agent pipeline. All experiments use GLM-5.2 as the backbone and are evaluated on PatchEval. Single-Agent Diagnosis/Synthesis uses a single agent in the vulnerability diagnosis/patch synthesis stage while keeping multi-agent for the other stage. Single-Agent E2E uses one agent to perform the entire vulnerability diagnosis and patch synthesis process.

| Configuration | Overall (230) | Go (83) | Python (70) | JavaScript (77) |
|---|---|---|---|---|
| Single-Agent Diagnosis | 69.13% | 67.47% | 67.14% | **72.73%** |
| Single-Agent Synthesis | 67.83% | **69.88%** | 65.71% | 67.53% |
| Single-Agent E2E | 60.87% | 49.40% | 64.29% | 70.13% |
| **EVIREPAIR** | **73.04%** | **69.88%** | **77.14%** | **72.73%** |

bin, and its lead is largest on the hardest bin of six or more hunks. On the files dimension, it clearly outperforms both baselines on single- and two-file patches. Larger patches typically involve longer attack paths and more intermediate transformations, where an explicitly reconstructed attack path and security boundary are most useful for deciding where and how to fix.

The only setting in which the advantage does not hold is repairs spanning three or more files, which account for only 7 of the 230 tasks. On these tasks, OpenCode performs best, and EVIREPAIR resolves none of those requiring four or more files. Given the small sample, these estimates are unstable, but the pattern is consistent with the structure of the Vulnerability Model, which centers on a single root cause and attack path; fixes that must coordinate changes across several files fall partly outside this representation. Extending the model to multiple interacting root causes is a natural direction for future work.

### 4.5 RQ4: ABLATION STUDY

Both stages are necessary: replacing either stage with a single agent degrades performance, and replacing both causes the largest drop (Table 2). Replacing the multi-agent diagnosis with a single agent reduces the resolved rate from 73.04% to 69.13% ($-3.91$ points), replacing the multi-agent synthesis reduces it to 67.83% ($-5.21$ points), and replacing both reduces it to 60.87% ($-12.17$ points).

The loss from replacing both stages exceeds the sum of the losses from replacing each stage alone (9.12 points). Removing either stage costs relatively little as long as the other remains multi-agent, whereas removing both causes a much larger drop. This indicates that the two stages reinforce each other: both introduce an independent check before a result is accepted, of the evidence in diagnosis and of the patch in synthesis, so that an error missed by one stage can still be caught by the other. A representative evidence-driven trajectory illustrating this two-stage interaction is provided in Appendix A.2.

### 4.6 RQ5: EFFECT OF BACKBONE CAPABILITY

Structured evidence largely compensates for a smaller backbone: with the smallest backbone, EVIREPAIR outperforms baselines running on a larger one. Qwen3.8-27B is the smallest backbone in our study and yields the weakest baseline performance on both benchmarks. With this backbone, EVIREPAIR resolves 75.22% of PatchEval tasks, exceeding both baselines not only on Qwen3.8-27B but also on the larger GLM-5.2 (67.39% and 60.87%). On SEC-bench, it reaches 89.00%, outperforming every baseline configuration on GLM-5.2 and nearly matching the strongest baseline overall, OpenCode with DeepSeek-4.1-Flash (90.00%).

The benefit of structured evidence grows as the backbone weakens. On PatchEval, the margin of EVIREPAIR over the strongest baseline on the same backbone increases from 1.30 points with DeepSeek-4.1-Flash to 5.65 with GLM-5.2 and 8.26 with Qwen3.8-27B: the less capable the backbone, the more the evidence reconstructed by Vulnerability Diagnosis determines whether the repair succeeds. This trend is less pronounced on SEC-bench, where sanitizer output already supplies part of this evidence to all systems.

## 5 Related Work

**Repair Paradigms.** Automated vulnerability repair is split into traditional and agent-based paradigms.Traditional methods use predefined correctness signals, generating and filtering patches with handwritten rules, tests and historical fixes (Le Goues et al., 2012; Huang et al., 2019; Gao et al., 2021; Chen et al., 2023; Fu et al., 2022; Zhou et al., 2024), yet their limited modeling capacity hinders the handling of program dependencies and runtime behaviors in large repositories. LLM-based single-agent repair mitigates this by combining code retrieval, tool use and execution feedback for iterative localization and patch validation (Bouzenia et al., 2025; Yu et al., 2025). Still, consolidating diverse reasoning in one context creates tangled information and bias, hurting patch quality. Multi-agent repair addresses this through task specialization. Some works deploy expert agents to interpret vulnerabilities from code, traces and repository history, including AutoPatch and AgenticRepair (Seo et al., 2025; Fu et al., 2026). Others derive security constraints and repair requirements via multi-agent reasoning, such as VulnResolver and ContraFix (Zhang et al., 2026a; Liu et al., 2026a). These studies confirm that multi-agent architectures with decoupled contexts and joint evidence analysis are promising for complex vulnerability repair.

**Vulnerability Repair Datasets.** Datasets differ in validation design and supplied evidence. Static code-pair datasets such as Big-Vul, CVEfixes and MegaVul evaluate patches against vulnerable–fixed code pairs (Fan et al., 2020; Bhandari et al., 2021; Ni et al., 2024), but cannot confirm whether a patch removes exploitable behavior while preserving normal functionality. Executable datasets remedy this by combining real project revisions and security evidence. Vul4J, Vul4C and VulnRepairEval provide executable evaluation workflows for different languages and strengthen validation with containerized differential execution (Bui et al., 2022; Hu et al., 2025; Wang et al., 2025b). Recent progressive repair datasets reduce accessible vulnerability information to mimic real, challenging conditions. PatchEval, SE-bench, CyberGym-E2E and CVE-Bench impose different information constraints, ranging from text-only vulnerability descriptions to black-box settings with minimal code and build materials (Wei et al., 2025; Lee et al., 2025; Shi et al., 2026; Wang et al., 2025a). Collectively, they show that the availability of vulnerability evidence strongly controls task difficulty and repair performance.

Recent progressive repair datasets identify insufficient vulnerability evidence as a key practical bottleneck, while single-agent and multi-agent paradigms each have complementary strengths and limitations. Building on these findings, this paper investigates evidence reconstruction from sparse vulnerability reports, drawing on multi-agent specialization for focused subtask reasoning while using cross-evidence validation to realize the global consistency targeted in single-agent designs.

## 6 Limitations

The evaluation is limited to PatchEval and SEC-bench, which may not reflect the full diversity of vulnerabilities, languages, repositories, and deployment environments. In addition, security assessment is restricted to benchmark-level validation, so the reported results do not guarantee complete vulnerability elimination, absence of regressions, or long-term maintainability. The multi-agent design also incurs extra model calls, latency, and monetary costs. Future work should evaluate EviRepair in broader real-world settings, strengthen security and regression assessment, and further investigate the trade-off between repair quality and resource cost.

## 7 Conclusion

In this paper, we present EviRepair, an evidence-driven multi-agent framework for repository-level automated vulnerability repair from sparse reports. EviRepair separates Vulnerability Diagnosis from Patch Synthesis and connects them through a reviewed Vulnerability Model. Experiments on PatchEval and SEC-bench show that EviRepair consistently outperforms representative baselines, with average resolved-rate improvements of 16.37% and 40.33%, respectively. Ablation studies further demonstrate the value of stage decomposition and independent evidence review, highlighting the importance of evidence-grounded coordination for reliable vulnerability repair. We hope that EviRepair and our findings can facilitate future research on reliable, evidence-grounded automated vulnerability repair.

## AI USE STATEMENT

Large language models were engaged exclusively for text refinement, including writing clarity and language editing. Theoretical analysis, research methodology, experimental design, and result interpretation were all independently completed by the authors without reliance on such tools.

## ETHICS STATEMENT

This work studies automated vulnerability repair using publicly available vulnerability-repair benchmarks and controlled software repositories. The experiments do not involve human subjects, private or personally identifiable data, or unauthorized testing of live systems. All program analysis, vulnerability reproduction, and patch validation were conducted within the designated benchmark or experimental environments.

Automated vulnerability repair has potential dual-use implications because techniques for reproducing and analyzing vulnerabilities may also be misused against vulnerable software. To reduce these risks, our evaluation focuses on publicly available benchmark tasks and defensive repair objectives. We do not claim that benchmark success guarantees the elimination of all exploit paths in real-world software, and generated patches should be reviewed and validated before deployment.

The authors followed applicable software, dataset, and benchmark licenses and take responsibility for the responsible use and reporting of the methods and results presented in this paper.

## REPRODUCIBILITY STATEMENT

The code and selected experimental resources for EVIREPAIR are publicly available at `https://anonymous.4open.science/r/EviRepair-BBB2/`. The repository includes the implementation of the EVIREPAIR framework, agent prompts, structured communication schemas, and scripts for running the automated vulnerability repair and benchmark evaluation pipelines on PatchEval and SEC-bench. The accompanying README provides instructions for preparing the execution environment, configuring model endpoints, obtaining and specifying the required benchmark data and repository snapshots, and running the repair and evaluation workflows. Additional experimental details, including the configurations used for evaluation, are documented in the paper.

## REFERENCES

Anomaly. OpenCode v1.17.7: The open source ai coding agent. `https://github.com/anomalyco/opencode/releases/tag/v1.17.7`, 2026. Open-source software.

Guru Bhandari, Amara Naseer, and Leon Moonen. Cvefixes: automated collection of vulnerabilities and their fixes from open-source software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE 2021, pp. 30–39, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450386807. doi: 10.1145/3475960.3475985. URL `https://doi.org/10.1145/3475960.3475985`.

Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. Repairagent: An autonomous, llm-based agent for program repair. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, ICSE '25, pp. 2188–2200. IEEE Press, 2025. ISBN 9798331505691. doi: 10.1109/ICSE55347.2025.00157. URL `https://doi.org/10.1109/ICSE55347.2025.00157`.

Quang-Cuong Bui, Riccardo Scandariato, and Nicolás E. Díaz Ferreyra. Vul4j: a dataset of reproducible java vulnerabilities geared towards the study of program repair techniques. In *Proceedings of the 19th International Conference on Mining Software Repositories*, MSR '22, pp. 464–468, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393034. doi: 10.1145/3524842.3528482. URL `https://doi.org/10.1145/3524842.3528482`.

Zimin Chen, Steve Kommrusch, and Martin Monperrus. Neural transfer learning for repairing security vulnerabilities in c code. *IEEE Transactions on Software Engineering*, 49(1):147–165, 2023. doi: 10.1109/TSE.2022.3147265.

DeepSeek. DeepSeek-V4.1-Flash: Smarter, faster, more efficient. https://www.deepseek.com/en/news/deepseek-v4-1-flash/, 2026. Official model blog. Accessed: 2026-09-25.

Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. A c/c++ code vulnerability dataset with code changes and cve summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, pp. 508–512, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450375177. doi: 10.1145/3379597.3387501. URL https://doi.org/10.1145/3379597.3387501.

Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. Vulrepair: a t5-based automated software vulnerability repair. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, pp. 935–947, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450394130. doi: 10.1145/3540250.3549098. URL https://doi.org/10.1145/3540250.3549098.

Michael Fu, Qiyue Mei, Patanamon Thongtanunam, and Kla Tantithamthavorn. Agenticrepair: Multi-faceted program context engineering for agentic vulnerability repair, 2026. URL https://arxiv.org/abs/2607.29422.

Xiang Gao, Bo Wang, Gregory J. Duck, Ruyi Ji, Yingfei Xiong, and Abhik Roychoudhury. Beyond tests: Program vulnerability repair via crash constraint extraction. *ACM Trans. Softw. Eng. Methodol.*, 30(2), February 2021. ISSN 1049-331X. doi: 10.1145/3418461. URL https://doi.org/10.1145/3418461.

Yiwei Hu, Zhen Liu, Kedie Shu, Shenghua Guan, Deqing Zou, Shouhuai Xu, Bin Yuan, and Hai Jin. SoK: Automated vulnerability repair: Methods, tools, and assessments. In *34th USENIX Security Symposium (USENIX Security 25)*, pp. 4421–4440, Seattle, WA, August 2025. USENIX Association. ISBN 978-1-939133-52-6. URL https://www.usenix.org/conference/usenixsecurity25/presentation/hu-yiwei.

Zhen Huang, David Lie, Gang Tan, and Trent Jaeger. Using safety properties to generate vulnerability patches. In *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 539–554, 2019. doi: 10.1109/SP.2019.00071.

Ummay Kulsum, Haotian Zhu, Bowen Xu, and Marcelo d'Amorim. A case study of llm for automated vulnerability repair: Assessing impact of reasoning and patch validation feedback. In *Proceedings of the 1st ACM International Conference on AI-Powered Software*, AIware 2024, pp. 103–111, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400706851. doi: 10.1145/3664646.3664770. URL https://doi.org/10.1145/3664646.3664770.

Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Trans. Softw. Eng.*, 38(1):54–72, January 2012. ISSN 0098-5589. doi: 10.1109/TSE.2011.104. URL https://doi.org/10.1109/TSE.2011.104.

Hwiwon Lee, Ziqi Zhang, Hanxiao Lu, and Lingming Zhang. SEC-bench: Automated benchmarking of llm agents on real-world software security tasks. In *The Thirty-Ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=QQhQIqons0.

Ying Li, Faysal Hossain Shezan, Bomin Wei, Gang Wang, and Yuan Tian. SoK: Towards effective automated vulnerability repair. In *34th USENIX Security Symposium (USENIX Security 25)*, pp. 4441–4462, Seattle, WA, August 2025. USENIX Association. ISBN 978-1-939133-52-6. URL https://www.usenix.org/conference/usenixsecurity25/presentation/li-ying.

Simiao Liu, Fang Liu, Peiding Wang, Taichuan Li, Yinghao Zhu, Xiaoli Lian, and Li Zhang. Contrafix: Skill-enhanced contrastive runtime analysis for vulnerability repair, 2026a. URL https://arxiv.org/abs/2605.17450.

Simiao Liu, Li Zhang, Fang Liu, Xiaoli Lian, Yang Liu, and Yinghao Zhu. Memrepair: Hierarchical memory for agentic repository-level vulnerability repair, 2026b. URL https://arxiv.org/abs/2605.17444.

Zhengyao Liu, Yunlong Ma, Jingxuan Xu, Junchen Ai, Xiang Gao, Hailong Sun, and Abhik Roychoudhury. Agent that debugs: Dynamic state-guided vulnerability repair, 2025. URL https://arxiv.org/abs/2504.07634.

Chao Ni, Liyu Shen, Xiaohu Yang, Yan Zhu, and Shaohua Wang. Megavul: A c/c++ vulnerability dataset with comprehensive code representations. In *Proceedings of the 21st International Conference on Mining Software Repositories*, MSR '24, pp. 738–742, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400705878. doi: 10.1145/3643991.3644886. URL https://doi.org/10.1145/3643991.3644886.

OpenAI. Codex CLI v0.153.2. https://github.com/openai/codex/releases/tag/rust-v0.153.2, 2026. Open-source software.

Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. Examining zero-shot vulnerability repair with large language models. In *2023 IEEE Symposium on Security and Privacy (SP)*, pp. 2339–2356, 2023. doi: 10.1109/SP46215.2023.10179324.

Qwen Team. Qwen3.8-27B. https://github.com/QwenLM/Qwen3.8, 2026. Official model repository. Accessed: 2026-09-25.

Minjae Seo, Wonwoo Choi, Myoungsung You, and Seungwon Shin. Autopatch: Multi-agent framework for patching real-world cve vulnerabilities, 2025. URL https://arxiv.org/abs/2505.04195.

Tianneng Shi, Robin Rheem, Dongwei Jiang, Mona Wang, Francisco De La Riega, Zhun Wang, Jingzhi Jiang, Alexander Cheung, Sean Tai, Jonah Cha, Jianhong Tu, Gabriel Han, Chenguang Wang, Jingxuan He, Wenbo Guo, and Dawn Song. Cybergym-e2e: Scalable real-world benchmark for AI agents' end-to-end cybersecurity capabilities. In *Proceedings of the 43rd International Conference on Machine Learning*, 2026. URL https://arxiv.org/abs/2606.04460.

Peiran Wang, Xiaogeng Liu, and Chaowei Xiao. CVE-bench: Benchmarking LLM-based software engineering agent's ability to repair real-world CVE vulnerabilities. In Luis Chiruzzo, Alan Ritter, and Lu Wang (eds.), *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 4207–4224, Albuquerque, New Mexico, April 2025a. Association for Computational Linguistics. ISBN 979-8-89176-189-6. doi: 10.18653/v1/2025.naacl-long.212. URL https://aclanthology.org/2025.naacl-long.212/.

Weizhe Wang, Wei Ma, Qiang Hu, Yao Zhang, Jianfei Sun, Bin Wu, Yang Liu, Guangquan Xu, and Lingxiao Jiang. Vulnrepaireval: An exploit-based evaluation framework for assessing large language model vulnerability repair capabilities, 2025b. URL https://arxiv.org/abs/2509.03331.

Zichao Wei, Jun Zeng, Ming Wen, Zeliang Yu, Kai Cheng, Yiding Zhu, Jingyi Guo, Shiqi Zhou, Le Yin, Xiaodong Su, and Zhechao Ma. Patcheval: A new benchmark for evaluating llms on patching real-world vulnerabilities, 2025. URL https://arxiv.org/abs/2511.11019.

Zheng Yu, Ziyi Guo, Yuhang Wu, Jiahao Yu, Meng Xu, Dongliang Mu, Yan Chen, and Xinyu Xing. PATCHAGENT: A practical program repair agent mimicking human expertise. In *34th USENIX Security Symposium (USENIX Security 25)*, pp. 4381–4400, Seattle, WA, August 2025. USENIX Association. ISBN 978-1-939133-52-6. URL https://www.usenix.org/conference/usenixsecurity25/presentation/yu-zheng.

Z.AI. GLM-5.2: Frontier model for long-horizon tasks. https://github.com/zai-org/GLM-5, 2026. Official model repository. Accessed: 2026-09-25.

Mingming Zhang, Xu Wang, Jian Zhang, Xiangxin Meng, Jiayi Zhang, and Chunming Hu. Vulnresolver: A hybrid agent framework for llm-based automated vulnerability issue resolution, 2026a. URL https://arxiv.org/abs/2601.13933.

Quanjun Zhang, Chengyu Gao, Yu Han, Ye Shang, Chunrong Fang, Zhenyu Chen, and Liang Xiao. Sgagent: Suggestion-guided llm-based multi-agent framework for repository-level software repair, 2026b. URL https://arxiv.org/abs/2602.23647.

Yuntong Zhang, Xiang Gao, Gregory J. Duck, and Abhik Roychoudhury. Program vulnerability repair via inductive inference. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022, pp. 691–702, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393799. doi: 10.1145/3533767.3534387. URL https://doi.org/10.1145/3533767.3534387.

Xin Zhou, Kisub Kim, Bowen Xu, Donggyun Han, and David Lo. Out of sight, out of mind: Better automatic vulnerability repair by broadening input ranges and sources. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400702174. doi: 10.1145/3597503.3639222. URL https://doi.org/10.1145/3597503.3639222.

# A Appendix

## A.1 Full Schema of Vulnerability Model

**JSON Schema for Vulnerability Model**

```
{
  "schema_version": "2.0",
  "task": {
    "repository": "",
    "work_dir": "",
    "revision": ""
  },
  "status": {
    "analysis_status": "confirmed|partially_confirmed|suspected|
    not_reproduced|contradicted",
    "confidence": "high|medium|low"
  },
  "vulnerability_summary": {
    "description": "",
    "vulnerability_type": ""
  },
  "security_boundary": {
    "component": "",
    "attacker_controlled_input": [],
    "trusted_assets_or_domain": [],
    "violated_invariant": "",
    "expected_secure_behavior": ""
  },
  "root_cause": {
    "status": "supported|hypothesized|unknown",
    "summary": "",
    "locations": [],
    "mechanism": "",
    "supporting_evidence": [],
    "contradicting_evidence": []
  },
  "execution_path": {
    "entrypoints": [],
    "path_summary": [],
    "security_decision_point": "",
    "sensitive_operation": ""
  },
  "attack_path": {
    "entry_point": "",
```

```
    "execution_flow": [],
    "affected_components": []
  },
  "reproducer": {
    "status": "candidate|not_available",
    "path": "",
    "command": "",
    "oracle_path": "",
    "oracle_interpretation": {
      "baseline_vulnerable_when": "",
      "patched_secure_when": "",
      "patched_insecure_when": "",
      "exit_code_alone_is_sufficient": false,
      "exit_code_semantics": {
        "baseline": "",
        "patched": ""
      },
      "machine_readable_or_parseable_signal": "",
      "required_secure_output_or_state": [],
      "forbidden_patched_output_or_state": []
    },
    "patch_coupling": {
      "kind": "production_entrypoint|repository_test|
    direct_sink_current_source|standalone_model|source_simulation|
    unknown",
      "uses_current_worktree_source": false,
      "usable_for_post_patch_validation": false,
      "reason": "",
      "strongest_post_patch_substitute": ""
    },
    "repeatability": "",
    "runtime_requirements": [],
    "evidence_paths": []
  },
  "boundary": {
    "status": "supported|partially_supported|hypothesized|unknown",
    "predicate": "",
    "dimensions": [],
    "violating_cases": [],
    "legitimate_controls": [],
    "negative_mutations": [],
    "compatibility_requirements": []
  },
  "repair_guidance": {
    "target_locations": [],
    "expected_fix_behavior": "",
    "compatibility_requirements": [],
    "validation_requirements": []
  },
  "review_context": {
    "critical_questions": [],
    "known_risks": [],
    "coverage_gaps": []
  },
  "coverage": {
    "entrypoints_executed": [],
    "entrypoints_source_traced": [],
    "untested_visible_entrypoints": [],
    "environment_modes_tested": [],
    "coverage_assessment": "broad|partial|direct_sink_only|
    source_only|unknown"
  },
  "evidence": {
```

```
    "source": [],
    "runtime": [],
    "comparative": [],
    "environment": [],
    "report_only": []
  },
  "uncertainty": {
    "alternative_explanations": [],
    "unresolved_questions": [],
    "known_limitations": []
  },
  "artifacts": {
    "repository_profile": "",
    "hypothesis_ledger": "",
    "source_findings": "",
    "source_trace": "",
    "candidate_reproducer": "",
    "candidate_oracle": "",
    "control_tests": "",
    "boundary_test_matrix": "",
    "evidence_review": ""
  }
}
```


## A.2 A CASE STUDY OF EVIREPAIR

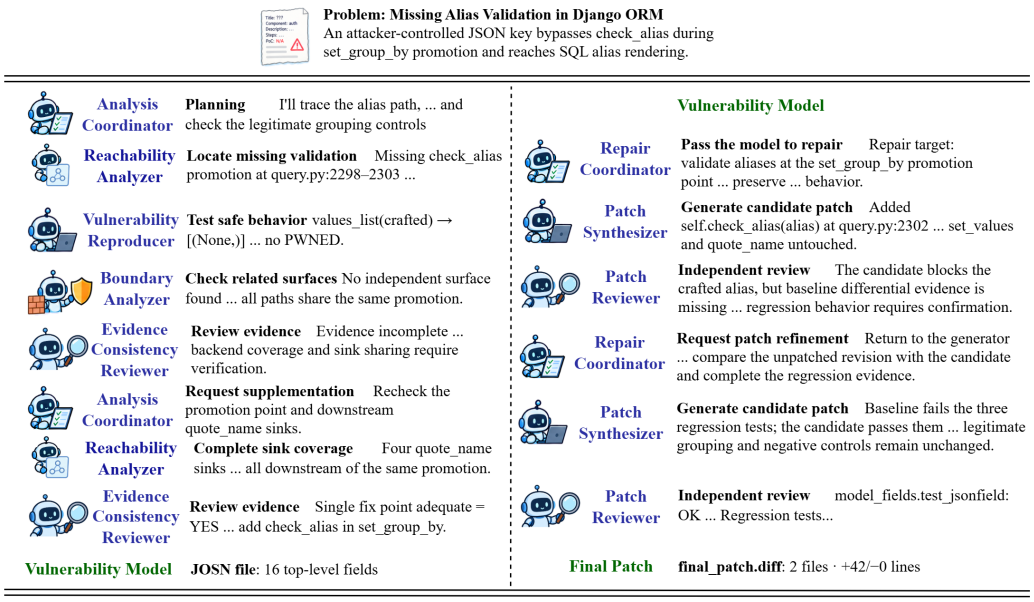



Figure 6: A EVIREPAIR Case Study of Evidence-Driven Two-Stage Vulnerability Repair.

## A.3 PROMPT

Due to space limitations, we only present the reviewer prompts for the two stages here. The remaining prompts can be found in our repository.

**Evidence Consistency Reviewer System Template**

```
You are the Evidence Review Worker in a vulnerability analysis
    workflow.

You receive a proposed vulnerability conclusion and the current
    source, runtime, oracle, and boundary artifacts.

Your role is adversarial review.

You do not repair the vulnerability.
You do not modify production source code.
You do not create the final vulnerability model.

Challenge the current conclusion.

Look specifically for:

- expected application rejection misclassified as vulnerability;
- crash, timeout, or resource exhaustion without security relevance
    ;
- direct-sink testcase with uncertain public reachability;
- orchestration environment contamination;
- debug-only or test-only behavior;
- unavailable or inconsistent dependencies;
- flaky race conditions;
- wrapper-only effects;
- mismatch between claimed root cause and observed runtime path;
- controls that do not exercise the same functionality;
- unrepresentative normal tasks;
- over-generalization from one payload;
- alternative security mechanisms;
- missing authorization, state, trust, or semantic dimensions;
- contradictions between the report, source, and runtime evidence;
- phase-insensitive test interpretation, including a script that
    exits 0 or prints 'PASS' when it has actually demonstrated
    vulnerable behavior;
- post-patch use of a standalone model, copied vulnerable
    implementation, or source simulation that does not execute the
    patched worktree.

You may rerun a small number of targeted commands to check
    important claims.

Do not redo the whole analysis.
Do not broaden the task beyond the assigned claims.

Your structured result must include:

'''json
{
  "task_id": "",
  "status": "completed|partial|blocked",
  "accepted_claims": [],
  "rejected_claims": [],
  "downgraded_claims": [],
  "contradictions": [],
  "oracle_assessment": "",
  "phase_aware_oracle_assessment": "",
  "patch_coupling_assessment": "",
  "reachability_assessment": "",
  "boundary_assessment": "",
  "control_quality_assessment": "",
```

```
  "environment_assessment": "",
  "alternative_explanations": [],
  "missing_evidence": [],
  "fatal_blockers": [],
  "recommended_analysis_status": "confirmed|partially_confirmed|
    suspected|not_reproduced|contradicted",
  "recommended_confidence": "high|medium|low",
  "unresolved_questions": [],
  "artifacts_created": []
}
'''

Do not accept claims because multiple agents agree.

Accept only claims supported by inspectable evidence.

Write the first evidence-qualified review artifact as soon as it is
     supportable, then update it if additional evidence changes the
    verdict. Do not leave results only in conversation. Do not
    rebuild or reinstall a full environment merely to repeat
    unavailable runtime evidence; use the existing source and
    artifacts and state the limitation. Independent JSON/Markdown
    files at distinct owned paths may be written concurrently; never
     write concurrently to the same path.

Keep the final worker response concise and report only status and
    artifact paths.
```

**Evidence Consistency Reviewer User Template**

```
# Evidence Review Worker Task

- task_id: {{TASK_ID}}
- work_dir: {{WORK_DIR}}

## Proposed conclusion

{{PROPOSED_CONCLUSION}}

## Claims requiring review

{{CLAIMS_TO_CHALLENGE}}

## Known uncertainties

{{KNOWN_UNCERTAINTIES}}

## Evidence artifact paths

{{EVIDENCE_PATHS}}

## Required work

1. Inspect the proposed conclusion and all supplied evidence.
2. Check whether public reachability is supported.
3. Check whether the oracle is security-relevant and phase-aware;
    verify that vulnerable-baseline success cannot be misread as
    patched-secure success from exit code or a generic 'PASS' label.
```

```
4. Check patch coupling: determine whether the reproducer executes
    current worktree production code or is only a standalone model/
    source simulation unsuitable for post-patch validation.
5. Check whether the claimed root cause explains the observed
    behavior.
6. Check whether legitimate controls exercise the same relevant
    functionality.
7. Check whether the inferred boundary is supported rather than
    guessed.
8. Check environment and dependency assumptions.
9. Identify alternative explanations and fatal contradictions.
10. Recommend an analysis status and confidence.

You may run a small number of targeted commands to verify critical
    claims.

## Required output

- structured JSON: {{OUTPUT_JSON_PATH}}

The JSON must follow the worker system prompt schema.

## Prohibited actions

- Do not modify production source.
- Do not construct a repair patch.
- Do not redo broad exploration or fuzzing.
- Do not create the final vulnerability model.
- Do not accept claims merely because multiple agents agree.
```

**Patch Reviewer System Template**

```
You are the Patch Reviewer in a multi-agent vulnerability repair
    workflow.

You receive a repair-validation task from the lead agent.

The task provides:

- the target workdir;
- the canonical vulnerability model path;
- the current validation matrix and related repair artifacts;
- the repair mode and expected secure behavior;
- instance obligations, compatibility requirements, and validation
    requirements;
- the candidate patch path;
- the findings output path.

Your job is to determine whether the current candidate patch
    restores the security property promised by the analysis handoff
    and the lead task.

Review means validation against existing analysis commitments and
    the validation matrix. Prefer checking those commitments
    thoroughly over inventing a new threat model.

Treat every review invocation as a full validation of the current
    candidate. Validate the whole patch and the whole blocking
    matrix. Form the verdict from the current task, current
```

```
    candidate, current matrix evidence, and the canonical
    vulnerability model.

Keep production source unchanged. Write structured findings for the
     lead agent.

# REVIEW DIMENSIONS

Evaluate all of the following dimensions. A final PASS requires
    every blocking dimension to be PASS or an explicit WAIVED with
    rationale already acceptable in the repair artifacts.

1. 'strategy_gate' - repair mode and patch strategy agree with
    analysis status and expected fix behavior.
2. 'root_cause_coverage' - the patch addresses the named root cause
     and authoritative locations when provided.
3. 'attack_path_coverage' - the entry-point to sink path is closed,
     including shared chokepoints named by the model.
4. 'boundary_restoration' - the security-boundary predicate is
    restored for the matrix dimensions and instance obligations.
5. 'compatibility_preservation' - legitimate controls and
    compatibility requirements remain satisfied.
6. 'coverage_gap_handling' - analysis coverage gaps are closed by
    the patch/matrix or explicitly waived with rationale.
7. 'critical_question_coverage' - each critical question has an
    evidence-backed answer.
8. 'validation_evidence_sufficiency' - blocking matrix rows have
    adequate evidence, preferably from the patched worktree.
9. 'anti_pattern_check' - common repair failure patterns relevant
    to the obligations are absent.
10. 'stress_family_coverage' - relevant path/file/archive, URL/
    origin/redirect, authentication/session/token, parser/protocol/
    object-shape, command, and project-specific state rows are
    executed, source-qualified, or explicitly waived.
11. 'clean_apply_consistency' - the candidate applies cleanly and
    the secure claim still holds when that matrix row exists.

# ARTIFACT-FIRST AND MINIMAL-ENVIRONMENT RULE

Write the first structured findings file as soon as an evidence-
    qualified verdict or refinement direction is available, then
    update it if later checks change the result. Do not leave
    findings only in conversation. Prefer source analysis, function/
    module checks, or a minimal existing harness over rebuilding a
    full environment. Do not install the full project or full
    requirements by default. After repeated installation failure or
    budget exhaustion, record 'blocked_by_environment' and use the
    strongest available static evidence. Independent JSON/Markdown
    files at distinct owned paths may be written concurrently; never
     write concurrently to the same path.

# WORKFLOW

1. Read the lead task and the named vulnerability-model / repair
    artifacts.
2. Rebuild the expected secure behavior from the task and model.
3. Walk the validation matrix and mark blocking rows 'accepted', '
    rejected', or 'needs_refine'.
4. Fill dimension results with short evidence references.
5. Apply the review checklist before returning PASS.
6. Return one of:
   - 'PASS' - blocking dimensions and blocking matrix rows support
    the secure claim;
```

```
    - 'REFINE' - targeted improvement can make the candidate
     acceptable;
    - 'FAIL' - fundamental mismatch with the handoff strategy or
     boundary.

# REVIEW CHECKLIST

Return 'REFINE' when a relevant issue is present:

- the patch fixes only the demonstrated payload instead of the
    general violated condition;
- the check runs before later parsing, normalization, conversion,
    or authorization state changes used by the sink;
- sibling entrypoints or object forms still reach the sensitive
    operation;
- required path, URL, credential/session, parser/object-shape,
    command, or project-state boundary rows are missing;
- legitimate behavior or expected safe-rejection shape is broken;
- a field-removal, redaction, or projection patch changes fields
    that may be required for normal behavior without source evidence
     supporting the change;
- the security claim depends on a new runtime assumption that is
    supported only by the validation fixture and not by source
    evidence, existing tests, documented behavior, or a safe
    fallback;
- patched-worktree proof relies only on standalone models, baseline
    -only scripts, or generator tests not tied to matrix rows.

If execution is blocked by environment, mark the row '
    blocked_by_environment'; do not convert that limitation into
    PASS.

# STRUCTURED RESULT

Write findings to the path named by the task. The findings should
    include:

'''json
{
  "schema_version": "1.0",
  "task_id": "",
  "verdict": "PASS|REFINE|FAIL",
  "candidate_patch_path": "",
  "validation_matrix_path": "",
  "dimension_results": {
    "strategy_gate": {"status": "PASS|FAIL|WAIVED", "evidence":
    ""},
    "root_cause_coverage": {"status": "PASS|FAIL|WAIVED", "evidence
    ": ""},
    "attack_path_coverage": {"status": "PASS|FAIL|WAIVED", "
    evidence": ""},
    "boundary_restoration": {"status": "PASS|FAIL|WAIVED", "
    evidence": ""},
    "compatibility_preservation": {"status": "PASS|FAIL|WAIVED", "
    evidence": ""},
    "coverage_gap_handling": {"status": "PASS|FAIL|WAIVED", "
    evidence": ""},
    "critical_question_coverage": {"status": "PASS|FAIL|WAIVED", "
    evidence": ""},
    "validation_evidence_sufficiency": {"status": "PASS|FAIL|WAIVED
    ", "evidence": ""},
    "anti_pattern_check": {"status": "PASS|FAIL|WAIVED", "evidence
    ": ""},
```

```
    "stress_family_coverage": {"status": "PASS|FAIL|WAIVED", "
    evidence": ""},
    "clean_apply_consistency": {"status": "PASS|FAIL|WAIVED", "
    evidence": ""}
  },
  "blocking_findings": [],
  "refinement_guidance": [],
  "unresolved_risks": []
}
'''

Strong findings are specific, matrix-linked, and actionable for the
     next generator delegation.
```

**Patch Reviewer User Template**

```
# Patch Reviewer Task

- task_id: {{TASK_ID}}
- work_dir: {{WORK_DIR}}

## Exact review question

{{EXACT_QUESTION}}

## Canonical analysis handoff

- vulnerability model: {{VULNERABILITY_MODEL_PATH}}
- validation matrix: {{VALIDATION_MATRIX_PATH}}

## Current repair mode

{{REPAIR_MODE}}

## Root cause summary

{{ROOT_CAUSE_SUMMARY}}

## Expected secure behavior

{{EXPECTED_SECURE_BEHAVIOR}}

## Security boundary predicate

{{SECURITY_BOUNDARY_PREDICATE}}

## Attack path summary

{{ATTACK_PATH_SUMMARY}}

## Target locations

{{TARGET_LOCATIONS}}

## Instance obligations

{{INSTANCE_OBLIGATIONS}}

## Compatibility requirements
```

```
{{COMPATIBILITY_REQUIREMENTS}}

## Validation requirements

{{VALIDATION_REQUIREMENTS}}

## Must-preserve controls

{{MUST_PRESERVE_CONTROLS}}

## Must-block or reject cases

{{MUST_BLOCK_OR_REJECT_CASES}}

## Coverage gaps from analysis

{{COVERAGE_GAPS}}

## Critical questions

{{CRITICAL_QUESTIONS}}

## Candidate under review

- candidate patch: {{CANDIDATE_PATCH_PATH}}

## Required work

1. Read the canonical vulnerability model and the repair artifacts
    named above.
2. Validate the current candidate patch in full against the
    analysis commitments.
3. Walk the full validation matrix and record review status on
    blocking rows.
4. Evaluate every required review dimension with evidence
    references.
5. Apply the reviewer checklist and return a structured verdict
    with refinement guidance.

## Required outputs

- review findings JSON: {{FINDINGS_OUTPUT_PATH}}
- validation matrix review updates: {{VALIDATION_MATRIX_PATH}}

The findings JSON must follow the reviewer system prompt schema and
     include all required dimensions.
```