Root-Cause-Driven Automated Vulnerability Repair

Hulin Wang
Arizona State University
Tempe, AZ, USA
hwang551@asu.edu

Zion Leonahenahe Basque
Arizona State University
Tempe, AZ, USA
zbasque@asu.edu

Jie Hu
Arizona State University
Tempe, AZ, USA
jiehu12@asu.edu

Ati Priya Bajaj
Arizona State University
Tempe, AZ, USA
atipriya@asu.edu

Yibo Liu
Arizona State University
Tempe, AZ, USA
yiboliu@asu.edu

Samuel Zhu
Arizona State University
Tempe, AZ, USA
sjzhu@asu.edu

Giorgi Kobakhia
Arizona State University
Tempe, AZ, USA
gkobakhi@asu.edu

Nikhil Chapre
Arizona State University
Tempe, AZ, USA
nchapre@asu.edu

Will Rosenberg
Arizona State University
Tempe, AZ, USA
whrosenb@asu.edu

Siddharth Mishra
Arizona State University
Tempe, AZ, USA
smish149@asu.edu

Aditya Maheshbhai Gabani
Arizona State University
Tempe, AZ, USA
agabani@asu.edu

Moritz Schloegel
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
moritz.schloegel@cispa.de

Adam Doupé
Arizona State University
Tempe, AZ, USA
doupe@asu.edu

Yan Shoshitaishvili
Arizona State University
Tempe, AZ, USA
yans@asu.edu

Ruoyu Wang
Arizona State University
Tempe, AZ, USA
fishw@asu.edu

Tiffany Bao
Arizona State University
Tempe, AZ, USA
tbao@asu.edu

Abstract

Recent LLM-based systems have made automated vulnerability repair increasingly practical, but two challenges remain. First, without strong signals about where a bug originates, repair agents drift toward shallow edits that silence the observed failure while leaving the underlying defect unresolved. Second, finding the root cause for bugs is hard: even developers familiar with the codebase frequently produce fixes that address symptoms rather than the root cause, and LLM-based agents, operating with noisier context and less program understanding, are no exception.

We present Kumushi[1], a root-cause-driven patching agent that addresses both challenges by combining diversified dynamic fault localization with evidence-weighted ranking to focus the LLM on the code most relevant to the defect. To rigorously measure whether Kumushi produces genuinely better patches, we also introduce a two-tier patch quality metric that pairs automated oracle validation with structured expert assessment of patches.

Evaluated on 178 C/C++ vulnerabilities, Kumushi substantially outperforms prior specialized repair agents under automated evaluation while matching a frontier commercial coding agent. Expert assessment then reveals differences that oracles cannot: Kumushi produces more root-cause fixes and fewer superficial patches, and is preferred in the majority of decisive pairwise comparisons. Together, these results demonstrate that progress in automated vulnerability repair requires not only stronger patching systems, but also richer evaluation methods capable of distinguishing genuine fixes from oracle-passing ones.

CCS Concepts

• **Security and privacy** → Use https://dl.acm.org/ccs.cfm to generate actual concepts section for your paper.

Keywords

automated program repair, patch generation, root cause

[1]Kumushi: "Kumu" is Hawaiian for "root," or "teacher," and "shi" is Chinese for "teacher"; together, the term denotes a teacher of root cause.

Anonymous Submission to ACM CCS 2017, Dallas, Texas

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Automated vulnerability repair is becoming increasingly critical as the volume of disclosed vulnerabilities continues to grow. Modern fuzzing infrastructure, such as OSS-Fuzz [20], has found thousands of bugs accompanied by reproducible proofs of concept (PoCs). More recently, efforts such as Project Glasswing [4] have uncovered thousands of more vulnerabilities in critical software and show no sign of waning. The resulting backlog far outpaces developers' triaging capacity, creating a strong demand for tools that can quickly generate correct patches for vulnerable source code [36].

Recent advances in large language models (LLMs) have given rise to a new class of LLM-based patching agents [30, 34, 56, 57, 61] that

address this problem through an iterative workflow: the agent first *localizes* the fault, then *synthesizes* a candidate patch, and finally *validates* it against replay-based test oracles comprising the original PoC and the developer test suite. If validation fails, the agent feeds the error signal back to drive the next round of localization and patch generation. The loop repeats until all oracles pass. Prior work has substantially improved this iterative loop by enabling agents to leverage program analysis techniques, such as static analysis [57], symbolic execution [56], and dynamic analysis approaches [61].

Despite recent advances in agentic patching, we observe that existing patching agents fail to reliably address the root causes of vulnerabilities and generate incomplete or incorrect patches. First, root cause localization is fundamentally limited by the underlying analysis techniques. Static approaches, such as PatchAgent [57], cannot precisely resolve indirect calls, leading to incorrect localization in cases like the heap out-of-bounds vulnerability in mruby [35]. In contrast, dynamic approaches based on symbolic execution (e.g., LoopRepair [56]) suffer from poor scalability and fail to handle this case in practice. Second, LLM-based patch generation is unstable due to its sensitivity to noisy context. Even when the correct fix location is identified, agents frequently drift and produce incorrect edits [2, 59]. For example, when fixing a heap buffer overflow in hunspell [40], PatchAgent correctly identifies the fix location but ultimately patches a nearby incorrect location, distracted by irrelevant context. Both limitations are critical and must be resolved for identifying and fixing root causes effectively.

In this paper, we present Kumushi, a root cause-driven patching agent that addresses the issues identified above. To balance the advantages and limitations of static and dynamic analyses, Kumushi employs *diversified dynamic fault localization*: instead of relying on static analysis alone, it collects execution evidence from a family of crash-triggering inputs, exposing root cause-correlated functions that a single trace would not surface. To prevent LLM's context drift, Kumushi employs an *evidence-weighted ranking* stage that consolidates the candidate pool into a ranked set of Functions of Interest, which concentrates the agent's attention on the most crash-relevant code before patch generation begins.

We evaluate the effectiveness of Kumushi and existing approaches in repairing the root causes of software vulnerabilities on real-world open-source projects. During the development of evaluation, we observe that assessing root cause repair is inherently challenging due to the lack of a *reliable ground truth*. This creates a chicken-and-egg problem: identifying the root cause of a vulnerability is itself non-trivial, which makes it difficult to determine whether a patch correctly addresses it. Automatic root cause identification could be an interesting solution, but it remains an open research problem. Prior work usually treats developer patches as ground truth; however, studies have shown that a significant fraction of developer patches fail to address the underlying defect. Even developers deeply familiar with the codebase frequently produce fixes that target symptoms rather than root causes [26, 32, 51].

An alternative is to rely on replay-based oracles [24], which re-execute the original PoC and developer test suites to verify that a patch suppresses the observed crash while preserving existing functionality. However, these oracles, analogous to dynamic testing in Programming Language theory, are complete but not sound[2]: a patch may eliminate the observed failure and pass all tests while leaving the underlying defect reachable through alternative execution paths [51], even when test suites are augmented with automated techniques such as fuzzing. This coverage gap limits the ability of oracle-based evaluation to distinguish true root-cause fixes from superficial repairs. Therefore, accurately assessing patch quality requires deeper analysis beyond what automated oracles alone can provide. However, such analysis typically requires security expertise and substantial manual effort, which does not scale to comprehensive, real-world evaluations. Consequently, it remains difficult to design an evaluation methodology that balances accuracy and practicality for large-scale, real-world systems.

To effectively evaluate Kumushi, we introduce a two-tier root-cause repair quality metric that combines automated oracle validation ("tier 1") with structured expert assessment ("tier 2") of patch completeness and correctness.

This framework enables a more rigorous evaluation of automated vulnerability patching systems such as Kumushi and provides stronger evidence of its ability to generate genuine root-cause fixes compared to existing state-of-the-art patching tools.

We evaluated Kumushi on 178 C/C++ vulnerabilities against three state-of-the-art works from academia and industry: PatchAgent [57], LoopRepair [56], and OpenAI's Codex [37]. Under tier-1 oracle-based evaluation, Kumushi reaches a plausible patch rate of 85%, outperforming PatchAgent (72%) and LoopRepair (2%), confirming the effectiveness of our root cause-driven localization.

However, we observe that Kumushi and Codex are statistically indistinguishable under all three replay-based oracles: replaying the PoC, executing the test suite, and replaying fuzzer-generated variant PoCs. This is a direct manifestation of the oracle coverage gap identified above. Both tools produce a similar percentage of plausible patches, but automated oracles alone cannot tell us which tool produces better patches.

Applying tier-2 expert assessment provides the deeper analysis needed to distinguish the two: Kumushi produces more root-cause fixes and fewer superficial patches than Codex. On the 144 bugs where both tools produced a plausible patch, human experts conducting an blind assessment preferred Kumushi on 68 (47%), Codex on 39 (27%), and judged 37 (26%) cases as equivalent; a sign test on the 107 decisive pairs gives $p = 0.0065$, confirming that Kumushi produces a statistically significantly greater number of better patches when bug difficulty is controlled.

Contributions: We make the following contributions:

- We identify two recurring limitations in state-of-the-art LLM-based patching tools: static-only fault localization consistently fails to pinpoint root causes, and unfiltered context causes agents to pivot toward crash-site suppression rather than root-cause repair. We further identify that existing oracle-based evaluation cannot reliably differentiate genuine root-cause fixes from superficial ones.
- We propose a root cause-driven patching approach that addresses both limitations through diversified dynamic fault localization

[2]In PL theory, a system is sound if a program that passes the system's checking will not exhibit certain forbidden behaviors at runtime.

and evidence-weighted FOI ranking, and implement and open-source Kumushi as a prototype of this approach.
- We conduct a two-tier evaluation on 178 C/C++ vulnerabilities. Kumushi outperforms PatchAgent by 18% in plausible patches (tier 1), and human experts prefer Kumushi patches over those of Codex by a 20% margin (tier 2). Notably, this preference gap is invisible to automated oracles and emerges only under human judgment.

2 Background and Motivation

We briefly discuss relevant background and investigate patch quality of existing approaches.

2.1 LLM-based Automated Program Repair

Software vulnerability repair has long been studied under the umbrella of Automated Program Repair (APR). Early approaches relied on templates [15, 29, 33, 46], search-based mutations [17, 21], semantic-based approaches[16, 18, 25, 44], and learning-based models [9, 62] to synthesize candidate patches. Recent advances in LLMs have changed the landscape, enabling repair tools that reason over code at the semantic level. LLM-based patch generation follows two general approaches. Prompt-based techniques [30, 34] query the model directly with engineered prompts that supply the bug report, surrounding code context, and optional chain-of-thought scaffolding, then treat the model's response as a candidate patch. On the other hand, agent-based techniques [56, 57, 61] embed the LLM in an iterative loop equipped with tools for code navigation, compilation, and test execution, allowing the agent to gather evidence, propose patches, and refine them across rounds. Within the agent-based approach, state-of-the-art systems differ chiefly in how they localize the root cause: PatchAgent equips the LLM with code-navigation tools (e.g., viewcode) and delegates localization entirely to the model. Starting from the sanitizer report, the agent retrieves source context iteratively and reasons about the defect statically, with no dynamic signal beyond the crash stack. LoopRepair localizes through concolic execution. Replaying the PoC under KLEE, it tracks how crash-relevant input bytes propagate through the program, retains the program points that carry the full taint, and ranks them by their distance to the crash. Each candidate is paired with a crash-free constraint, producing a semantically grounded set of suspect locations. CodeRover-S localizes through dynamic tracing. The PoC is executed on an instrumented binary that captures the full caller/callee graph of the buggy run. Functions that executed but are absent from the crash stack are resolved to source symbols and appended to the sanitizer report, giving the agent execution-grounded seeds beyond what the bare stack trace provides. More recently, general-purpose coding agents [3, 37, 39] have demonstrated strong competence on software-engineering benchmarks [27], vulnerability reproduction [50], PoC generation [31], code review [63] and can be applied to vulnerability repair tasks [31, 58] with minimal task-specific configuration efforts.

2.2 Patch Quality Evaluation in Practice

A high-quality patch should repair the faulty invariant at the source of a vulnerability rather than merely mitigate the observed crash. Verifying this property is itself a research problem, but two evaluation methods are common in practice: manual expert review and dynamic oracle validation.

In expert reviews, a reviewer reconstructs the root cause from the crash report and PoC behavior and judges whether the patch addresses the underlying defect rather than guarding the crash site. The judgment is usually reliable but labor-intensive, and it does not scale to address the size of modern vulnerability backlogs.

In terms of automated validation, the field usually relies on *replay-based oracles*, including PoC replay, developer test suite re-execution, and fuzzing-based variant exploration [24]. PoC replay checks if the reported crash no longer reproduces after the patch has been applied [34, 56]. Similarly, developer test suites are re-executed to check if documented behavior is preserved [42, 57]. Fuzzing campaigns seeded from the original PoC can be used to explore nearby inputs and provide a stronger guard against variants that re-trigger the defect [26, 51].

We observe that these oracles are complete but not sound. A patch may pass every replay-based check while leaving the underlying defect reachable through alternative execution paths [51]. Two patches that pass the same oracle suite can therefore differ substantially in whether they repair the root cause.

2.3 Preliminary Study

To understand the limitations of state-of-the-art LLM-based patching tools in practice, we conduct a preliminary study. We first run 10 independent trials of PatchAgent for each bug and identify 24 bugs that it fails to patch in every run. Using this subset of bugs, we examine whether other state-of-the-art tools, including LoopRepair [56], exhibit similar limitations.

We observe that PatchAgent fails to locate or examine the correct fix location for 9 out of 24 bugs, suggesting that static analysis alone is insufficient to pinpoint root causes. We use the developer patch as the definition of the correct fix location for these cases. While tools such as LoopRepair incorporate dynamic information, for instance, constraints derived from symbolic execution, they face significant scalability and generality challenges. In our evaluation, the KLEE symbolic execution engine used by LoopRepair failed to execute on all 24 targets, producing no patches. From these observations, we believe our system needs to use lightweight dynamic analysis that avoids the overhead of formal symbolic execution while still grounding the repair process in runtime evidence.

Throughout our experiment, we also observe that the PatchAgent's reasoning frequently surfaces the right location, but lengthy call chains, unrelated source files, and tangential snippets accumulate alongside it. Under this noise, the LLMs drift from the correct fix sites and edits elsewhere. In our study, 9 out of 24 bugs follow this drift pattern: the LLM initially examines the correct line, but the surrounding context pulls it away. In another 6, it stays on the correct line but repeatedly produces patches that either fail to compile or fail to mitigate the crash, exhausting its repair budget. These 15 cases suggest that even when the correct location is identified, ineffective context filtering remains a critical barrier to successful repair.

In summary, these two observations motivate Kumushi's design:

(1) *Static analysis alone cannot pinpoint the root cause.*

(2) *Unfiltered context drowns out the correct fix site.*

3 Kumushi: Root-Cause-Driven Patch Generation

Motivated by these observations, we propose Kumushi, a dynamic-diversified, root-cause-focused, agentic patch generation framework that jointly addresses fault localization accuracy and patch quality. As shown in Figure 1, Kumushi accepts three inputs (a sanitizer report, the original proof-of-concept (PoC), and the project codebase) and operates through three tightly coupled stages. First, *Diversified Fault Localization* constructs a failure-relevant candidate pool. It combines crash-stack extraction from the sanitizer report, dynamic function-call traces from fuzzer-generated crash variants of the original PoC, and static dependency analysis. Second, *Evidence-Weighted FoI Ranking* consolidates the pool into a focused top-20 set of Functions of Interest. It scores each candidate against crash-class-conditioned evidence tags, then applies a diversity-aware rerank so the patch agent receives distinct hypotheses across source files. Finally, *Agentic Patch Generation* consumes the ranked FoIs as context. The patch agent iteratively explores the codebase and synthesizes candidate patches, while an automated verifier checks each edit against oracles. The agent emits a patch only when all three checks pass. The remainder of this section details each stage.

3.1 Stage 1: Diversified Fault Localization

Stage 1 collects a broad pool of functions likely correlated with the vulnerability's root cause. Stage 2 then consolidates, scores, and filters this pool into the focused top-20 FoI set that guides patch generation. Motivated by the localization limitations identified in section 2, Kumushi addresses both the coverage gap of static-only approaches and the scalability limitations of heavyweight dynamic analysis through a two-step collection process.

Dynamic FoI Collection via Fuzzing. Rather than reasoning from a single crashing execution, Kumushi first diversifies the original PoC using AFL++ [11] (v4.40c) in crash exploration mode with a 12-hour time budget per bug. In crash exploration mode, the fuzzer takes the original crashing input as a seed and mutates it to generate new inputs, retaining only those that still crash the target. Aurora [5] observed that crash variants derived from the same seed tend to trigger the same underlying bug via different execution paths. The resulting family of variant inputs therefore exposes more root-cause-correlated functions than any single trace alone. Crash exploration may occasionally produce inputs that trigger a different bug. Stage 2's evidence-weighted scoring down-weights functions that appear only in a minority of traces, mitigating such outliers.

For each variant in the diversified PoC family, Kumushi instruments the target binary with function-entry hooks and replays the variant under the instrumented binary. Every function executed on the path to the crash is emitted as a candidate FoI and enters the pool. Kumushi additionally parses the ASan report of the original PoC and extracts every function named on the crash stack as a candidate FoI. Prior study found that the fix for roughly 60% of cases patched a function on the crash stack [10]. The union of all per-variant execution traces and the crash-stack functions forms the dynamic candidate pool.

Static FoI Augmentation via Dependency Analysis. Dynamic tracing captures functions that actually execute but may miss functions that influence the crash site through data dependencies not exercised by the available inputs. To augment FoI coverage, Kumushi applies CodeQL static dependency analysis [19], but deliberately restricts it to the subset of functions that appear in the original PoC's ASan report. Applying static analysis to all dynamic candidates would re-introduce the over-approximation problem, exploding the candidate pool with functions only loosely related to the crash. By anchoring static analysis to the ASan report subset, Kumushi obtains the precision benefits of crash-stack grounding while still recovering candidates that dynamic tracing alone would miss.

Concretely, CodeQL's intra-procedural dataflow query (`DataFlow::localFlow`) identifies variables whose values flow into the crash-line variables within the crash site function. A second CodeQL query enumerates all functions that read or write variables of matching name and access type. Whole-program dataflow is out of scope at our evaluation scale.

Beyond the local dataflow analysis, we take up to ten call-stack functions from the sanitizer report as *anchors* and perform a bounded bidirectional breadth-first traversal of the call graph: upstream to callers and downstream to callees. For spatial and temporal bugs we include allocation frames; for temporal bugs we also include free frames. For each discovered function f, we track three signals: the number of anchors from which f is reachable, $|\text{anchors}(f)|$; the number of call edges landing on f, $\text{edge}_{\text{hits}}(f)$; and the shortest distance from any anchor to f, $\min_{\text{depth}}(f)$. A function reached by many anchors, via many edges, at shallow depth is more likely to be implicated in the crash. We rank candidates lexicographically by $\langle -|\text{anchors}|,\ -\text{edge}_{\text{hits}},\ \min_{\text{depth}} \rangle$ and retain the top 300 to bound pool growth. If too few candidates are returned, we widen the search by adding more call-stack, allocation-stack, and free-stack functions and merge the results.

After both steps, Kumushi merges the dynamically diversified candidates and the statically augmented candidates into a single pool of crash-correlated FoIs. Stage 2 then consolidates, scores, and ranks this pool.

3.2 Stage 2: Evidence-Weighted FoI Ranking

Stage 2 takes the merged candidate pool from Stage 1 and produces a focused, ranked top-20 FoI set that serves as the repair context for the patch agent. Each candidate in the pool carries one or more source tags indicating which collection step contributed it. The tags mark a function as appearing on the crash stack, in a per-variant dynamic execution trace, on an allocation or free stack from the sanitizer, at a sanitizer-reported object-origin site, or in the static variable-dependency results. A candidate contributed by multiple steps carries multiple tags, and the scoring step below uses this overlap as its primary signal.

We first consolidate the pool, discarding functions that cannot plausibly be the defect site: fuzzing harnesses and entry wrappers (e.g., `LLVMFuzzerTestOneInput`), standard library primitives (`memcpy`, `malloc`, and similar), and candidates rooted in test or fuzzer directories. Duplicates contributed by multiple steps are

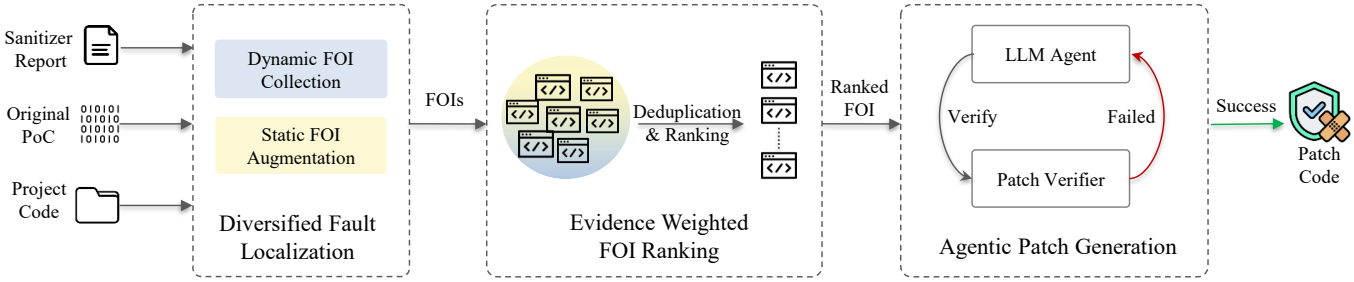

Figure 1: Overview of Kumushi. Given a sanitizer report, crash input, and project codebase, the Diversified Fault Localization stage produces a pool of Functions of Interest (FoIs); the Evidence-Weighted FoI Ranking stage consolidates and ranks them into a top-20 list; and the Agentic Patch Generation stage iteratively produces and verifies a patch, looping back with additional information on each failure until the patch is validated.

fused into a single record whose tag set is the union of the originals'. We then score each surviving candidate. Tag weights depend on the crash class. For temporal errors such as use-after-free, the free stack carries the greatest weight; for contract violations such as null dereference, the crash stack carries it instead. To prevent correlated sources from being counted multiple times, we first partition tags into evidence families.

For instance, free-stack, allocation-stack, and object-origin share one family because all three describe the same allocation event. Within a family f, Kumushi aggregates the active tag weights using an OWA-style operator [55]:

$$s_f = \min\Big(c_f,\ w_{(1)} + \alpha \sum_{i \geq 2} w_{(i)}\Big), \tag{1}$$

where $w_{(1)} \geq w_{(2)} \geq \ldots$ are the bug-class weights of the active tags in that family, $\alpha < 1$ discounts additional same-family tags to reflect correlation rather than independence, and c_f limits the maximum contribution of family f.

Intuitively, the strongest tag in a family determines the primary contribution, while the remaining tags provide only attenuated reinforcement, since tags from the same family often witness the same underlying fact. The cap c_f encodes the maximum confidence that the family can contribute in principle. For example, stack-trace evidence, which places the function directly on the crashing control flow, contributes more than call-trace evidence, which shows only that the function executed.

We then combine the family-level scores with a noisy-OR [43]:

$$s = 1 - \prod_f (1 - s_f), \tag{2}$$

under the standard assumption that evidence families provide independent support for defectiveness. Example 1 illustrates the resulting behavior. A single strong evidence family can substantially raise the final score, while weak or missing families do not pull down the contribution of stronger ones. The two operators serve complementary purposes. The OWA-style aggregation controls redundancy within a family, where multiple tags may be correlated observations of the same event. The noisy-OR combines support across families, where each family represents a distinct source by which a defect may manifest.

Given a score per candidate, the ranker produces the final list in two passes. We first rank by raw score, then apply a file-level diversity rerank to the tail. The rerank prefers candidates from source files not yet selected, so the patch agent receives distinct hypotheses rather than near-duplicates from the same file. We then truncate the list to match the patch agent's context budget.

Example 1. *Consider three use-after-free FoIs drawn from the same tag vocabulary (*stack-trace 0.85, free-stack 0.90, alloc-stack 0.65, object-origin 0.55*; α=0.25, c_{crash}=0.97, c_{alloc}=0.95*). By tag-family taxonomy,* stack-trace *is a crash-stack tag, while* free-stack, alloc-stack, *and* object-origin *are allocation-family tags. Each FoI fires its own subset.*

(1) `do_close` *fires* stack-trace *(crash family) and both* free-stack *and* alloc-stack *(allocation family). It scores* **0.9925***:*

- *allocation family:* $s_{alloc} = \min(0.95,\ 0.90 + 0.25 \cdot 0.65) = 0.95$
- *crash-stack family:* $s_{crash} = \min(0.97,\ 0.85) = 0.85$
- *noisy-OR:* $s = 1 - (1 - 0.85)(1 - 0.95) = 0.9925$

(2) A FoI witnessed only by allocation-family tags — with any combination of free-stack, alloc-stack, *and* object-origin *active — cannot exceed* $\mathbf{c}_{alloc} = \mathbf{0.95}$*, since no crash-family term enters the noisy-OR.*

(3) A FoI tagged only by stack-trace *on the crash stack scores* **0.85**.

The ordering $0.85 < 0.95 < 0.9925$ *makes the design choice visible: no amount of within-family evidence, even at saturation, overtakes a candidate witnessed across two independent families. Kumushi ranks breadth of evidence above depth.*

3.3 Stage 3: Agentic Patch Generation

The patch agent is an LLM agent driven by a structured prompt and a small tool set. The prompt supplies the sanitizer report, the top-20 FoIs together with their source code, an explore-then-edit workflow, and fix-quality guidelines for patching vulnerabilities. We equip the agent with read-only tools to navigate the clang-indexed codebase, a string-substitution tool to edit source files, and a verifier to validate patches. Every run operates on a freshly copied, git-baselined working directory so that the agent can explore the codebase and edit files without affecting the original codebase.

After each edit, the patch verifier applies three progressively stricter validation checks: compiling the patched project under AddressSanitizer, replaying the original crashing PoC, and re-executing the project's existing test suite. If any check fails, the verifier returns the failure output to the agent to guide its next exploration or edit. The agent then can refine its hypothesis and try again. Until the first time the patch passes all three checks, the run terminates and the agent emits the patch. The agent runtime caps each run at 150 turns to prevent infinite loops.

4 Evaluation

We design experiments to study three research questions:

- **RQ1**: How do LLM-based patching agents compare under dynamic oracle-based evaluation?
- **RQ2**: How do root-cause-driven and general-purpose LLM-based patching agents differ in patch comprehensiveness, and what repair strategies and failure modes characterize each?
- **RQ3**: What factors drive patch quality differences between root-cause-driven and general-purpose LLM-based patching agents, and what do they reveal about the strengths and limitations of root-cause-anchored patching?

4.1 Experiment Setup

Tools. We compare Kumushi against three state-of-the-art techniques that represent the spectrum of LLM-based patch generation.

- **PatchAgent** [57] relies on the LLM alone for agentic fault localization and patch synthesis, iteratively refining candidate patches against the original PoC and developer test suite until the repair budget is exhausted. It uses no dynamic analysis beyond oracle replay validations.
- **LoopRepair** [56] augments the LLM-based repair loop with symbolic execution: it runs KLEE on the original PoC's execution path to extract path constraints and taint traces that predict patch locations before delegating patch synthesis to an LLM. We configured it to use the same PoC-replay and test-suite oracle as PatchAgent for validation.
- **Codex** [37] is a general-purpose coding agent. We configure it with the original PoC, the developer test suite, and the project codebase, and prompt it to perform agentic patch generation, letting it autonomously decide how to explore the codebase and produce a patch. Unlike Kumushi, it receives no pre-computed fault-localization context and no fix-quality guidelines.

To ensure a fair comparison, all techniques use gpt-5.2-2025-12-11 [38], the latest model while we ran the experiment, as the underlying large language model backend.

Dataset. We evaluate on the 178-bug benchmark introduced by Yu et al. [57], which spans 30 open-source C/C++ projects ranging from 5k to 1M source lines of code (SLOC). Each bug ships with an AddressSanitizer (ASan) crash report, a crash input, and a script to run test suites. We group ASan crash labels into five classes based on the ASan report label; Table 1 gives the class definitions. Appendix B lists the full set of targets.

Table 1: Overview of the 178-bug benchmark.

Size (SLOC)	#Proj.	Spatial	UAF	NPD	Num	Other
Small (<50 k)	10	25	6	2	1	5
Medium (50 k–250 k)	13	40	6	5	4	7
Large (>250 k)	7	52	11	11	1	2
Total	**30**	**117**	**23**	**18**	**6**	**14**

Spatial: heap-, stack-, or global-buffer overflow, or heap out-of-bound read.
UAF: use-after-free, double-free, and invalid-free.
NPD: null-pointer dereference.
Num: divide-by-zero, integer overflow, and invalid bit-shift.
Other: sanitizer-unclassified crashes including unknown-read/write, SEGV-on-unknown-address, and API-contract violations.

Prompt. The prompt for Kumushi's patching agent includes the root-cause analysis summary, the sanitizer report, the overall workflow, and the patching guidelines. Appendix C shows the complete prompt.

Token usage. Table 2 reports per-bug mean and median token consumption and estimated cost for each system. All four systems use GPT-5.2 under their default repair budgets.

Table 2: Token usage and estimated cost (using GPT-5.2).

	Mean		Median	
System	**Tokens**	**Cost**	**Tokens**	**Cost**
PatchAgent	44,899	$0.16	43,255	$0.16
LoopRepair	83,936	$0.30	50,046	$0.18
Codex	239,848	$0.86	130,830	$0.47
Kumushi	1,120,538	$4.02	517,992	$1.86

4.2 Root Cause Assessment Metric

To rigorously measure the performance of each tool and compare the generated patches comprehensively, we introduce a two-tier patch quality metric that combines the scalability of automated oracles with the precision of human assessment.

Our two-tier metric design considers both evaluation precision and scalability. Automated oracles such as PoC replay, test suites, and fuzzing are cheap to run at scale but passing them does not prove the underlying defect has been fixed [24], as a patch may suppress the observed failure while leaving the root cause unaddressed. Human review can determine whether a patch fixes the root cause, but applying it to every generated patch is prohibitively expensive, taking 16 minutes per patch [47]. Our metric addresses this tension by running automated oracles first to eliminate failed patches, then forwarding only the surviving plausible patches to human raters who assess whether the root cause has been resolved. Since the oracle stage requires only the patch, the crash report, and the program code base, our metric works for patches from any repair system.

4.2.1 Tier 1: Dynamic Oracle Validation. Three oracles cover the most common checks for validating LLM-generated patches [24]: *PoC replay* confirms that a reported crash no longer triggers. The *developer test suite* checks that documented behavior still holds.

AFL++ crash exploration, seeded from the PoC and run for 12 hours on the project's harness, detects patches that silence the reported crash but leave the defect reachable through mutated inputs.

We categorize each generated patch into one of three outcomes under progressively stricter dynamic oracles:

- **No Patch**: the patch fails to compile, or mitigate the original PoC crash, or pass the developer's existing test suite.
- **Partial Patch**: the patch passes compilation, PoC replay, and the developer test suite, but does not suppress all variant crashes discovered through 12 hours of AFL++ crash-exploration fuzzing against the project's own harness.
- **Plausible Patch**: the patch passes every dynamic check above, including suppression of all AFL++-discovered variant crashes within the 12-hour exploration budget.

4.2.2 Tier 2: Structured expert assessment. We forward patches that pass all three dynamic oracles to human raters, who evaluate each patch along two orthogonal axes and then, for Kumushi and Codex pairs, apply a pairwise comparison rubric. The primary axis captures whether the patch addresses the underlying defect:

- **Root-Cause Fix**: the patch directly repairs the underlying defect identified from the crash report and PoC, eliminating the faulty invariant at its source.
- **Symptom Fix**: the patch prevents or reduces the crash without addressing the underlying defect, leaving related execution paths potentially vulnerable.

The secondary axis is a boolean flag, `has_unrelated_changes`, set whenever the diff contains modifications orthogonal to the root cause. We track this flag because LLM-based tools occasionally edit code beyond the scope of the root cause: reformatting, refactoring adjacent functions, or making drive-by changes. These edits inflate apparent productivity, add review burden, and risk regressions. Surfacing such edits as a separate signal lets us distinguish a clean root-cause fix from a correct fix bundled with unwanted changes.

We recruited eight raters (graduate students with research experience in software security and with more than 1 year of vulnerability research experience). For each bug, three raters first reconstruct the runtime environment in Docker to run the PoC, then identify the root cause from the crash report and observed PoC behavior. Both Kumushi and Codex generated patches against identical Docker environments; only the patch-generation agent differed, controlling for confounders in the reproduction setup. We consult the original developer patch as a reference only and do not treat it as ground truth, since upstream fixes are themselves frequently incomplete or symptom-level [26, 51].

To mitigate source bias, we anonymize Codex and Kumushi outputs as `tool_a` and `tool_b`; the mapping is disclosed only after labeling is complete. Labeling proceeds in independent rounds: each rater labels every patch in isolation without access to the others' judgments, and raters rotate across batches of bugs across rounds.

In each round, the rater assigns a primary label (root-cause fix or symptom fix) and writes a justification that references the identified root cause. After a dedicated scan for out-of-scope edits, the rater sets the `has_unrelated_changes` flag. Raters record every flagged edit in a shared spreadsheet, along with their rationales. Only after both patches have been labeled in isolation raters apply the pairwise comparison rubric. Correctness takes priority: a root-cause fix outranks a symptom fix. When both patches repair the root cause, the rater breaks ties first by *semantic precision*, preferring the patch that targets the faulty invariant without broadening scope. The number of unrelated changes serves as the second tiebreaker. When neither patch reaches the root cause in our evaluation, we fall back to comprehensiveness, preferring the patch that covers more affected execution paths, edge cases, and related failure modes. Raters label clear dominance cases directly, reserve the rubric for close calls, and log all rationales for post-hoc audit. After three rounds, we use majority vote to determine the final label and provide inter-rater reliability statistics in Table 7.

Pairwise Comparison Rubric. The rubric is lexicographic on:

(1) **Correctness.** When one patch repairs the root cause and the other only suppresses the reported crash, prefer the root-cause fix. When both patches repair the root cause, prefer the one with stricter semantic alignment to the defect, which is the patch that targets the faulty invariant without broadening scope. Unrelated changes weigh against a patch under`has_unrelated_changes`; when counts are close, raters exercise judgment rather than apply a fixed threshold.
(2) **Comprehensiveness.** When neither patch repairs the root cause, prefer the symptom fix that covers more of the relevant execution paths, edge cases, and related failure modes stemming from the same defect. Unrelated changes again weigh against the patch under the same judgment-based weighting.

4.3 RQ1: Patching Capability

To answer RQ1, we run Kumushi against three baselines on 178 C/C++ vulnerabilities under three dynamic oracles: PoC replay, developer test suite, and 12h of AFL++ variant-crash exploration per bug. Table 3 reports the full outcome distribution. We classify each patch as No Patch, Partial Patch, or Plausible Patch. Kumushi substantially outperforms PatchAgent and LoopRepair, confirming the effectiveness of root-cause-driven localization. Kumushi and Codex, however, are statistically indistinguishable under all three oracles, revealing a ceiling on what oracle-based metrics can discriminate at the frontier.

Table 3: Patch quality across 178 C/C++ vulnerabilities, judged by dynamic oracles.

Tool	No Patch	Partial Patch	Plausible Patch
PatchAgent	13.5% (24)	14.6% (26)	71.9% (128)
LoopRepair	97.2% (173)	0.6% (1)	2.2% (4)
Codex	2.2% (4)	11.8% (21)	86.0% (153)
Kumushi	1.7% (3)	12.9% (23)	85.4% (152)

Per-tool breakdown. PatchAgent produces no patch on 13.5% of bugs and only partial patches on another 14.6%, for a combined failure rate of 28.1%. The partial-patch cases suppress the reported crash but fail under AFL++ variants; the no-patch cases concentrate on bugs where static context retrieval returned code irrelevant to the defect.

LoopRepair fails on 97.2% of bugs. Its symbolic-execution component (KLEE) compiled on 68 of the 178 targets and produced

usable output on only 5, leaving LoopRepair without localization input on the rest of the benchmark. Beyond this implementation limitation, symbolic execution is more expensive than the dynamic tracing Kumushi uses. Even on targets where KLEE succeeds, its per-target cost cannot scale to the diversified multi-input tracing Kumushi performs across the AFL++ variant family.

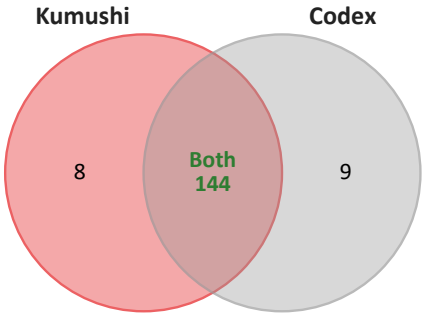

Figure 2: Comparison of Kumushi and Codex on plausible patch classification. The two systems overlap on 144 bugs; Kumushi uniquely patches 8 and Codex uniquely patches 9.

Kumushi and Codex produce nearly identical outcome distributions under all three oracles (Table 3). Among the 161 bugs where either tool produced a plausible patch, only 144 are shared: Kumushi patches 8 bugs that Codex misses, and Codex patches 9 that Kumushi misses (Figure 2). The aggregate counts are indistinguishable, but the tools fix different sets of bugs. The indistinguishability of Kumushi and Codex under all three oracles is not evidence of equivalent patch quality; it is evidence that oracle-based metrics cannot separate frontier tools at this level. RQ2 applies tier-2 human assessment to examine what the oracles cannot see.

4.4 RQ2: Patch Comprehensiveness

To answer RQ2, we examine whether Kumushi and Codex differ in the quality of the plausible patches they produce, a dimension that the three oracle metrics in RQ1 cannot capture. Three raters independently label all 305 plausible patches (152 from Kumushi and 153 from Codex) along two axes following the tier-2 protocol in Section 4.2.2: a root-cause vs. symptom judgment and a boolean `has_unrelated_changes` flag. We then characterize the repair strategies each tool employs through open coding, yielding eight root-cause fix strategies and nine symptom-fix failure strategies.

Tier-2 labeling reveals that Kumushi produces more root-cause fixes and fewer symptom fixes than Codex, and that the two tools diverge in the kind of repair they produce when they fall short of the root cause.

Aggregated results. Across 305 labeled patches, Kumushi produces more root-cause fixes than Codex (84.9% vs. 77.8%; 129 vs. 119, Table 4), indicating that Kumushi generates more comprehensive and correct patches than Codex overall. The root-cause fix gap falls short of $p < 0.05$ at this sample size; the unpaired aggregate comparison has limited statistical power, and the per-bug paired analysis in RQ3 is better suited to detect the difference.

Kumushi also produces more patches containing unrelated edits than Codex (28.3% vs. 21.6%; 43 vs. 33, Table 4). This higher rate does not signal lower patch quality from Kumushi. Within each tool, the unrelated-edit rate is consistent across root-cause fixes and symptom fixes (29.5% vs. 21.7% for Kumushi; 22.7% vs. 17.6% for Codex), with neither within-tool difference reaching statistical significance ($p > 0.05$ for both). Unrelated edits therefore reflect a stable property of each tool's editing behavior rather than a signal of patch incorrectness, and the higher unrelated-edit rate in Kumushi does not account for its root-cause fix advantage over Codex.

Table 4: Patch quality and unrelated change rate per tool.

Tool	Patch Quality	Count	With Unrelated	Without Unrelated
Codex	root-cause fix	119	27	92
	symptom fix	34	6	28
	total	**153**	**33**	**120**
Kumushi	root-cause fix	129	38	91
	symptom fix	23	5	18
	total	**152**	**43**	**109**

Repair strategy analysis. We next group each patch by its primary repair strategy through open coding, yielding eight root-cause fix strategies and nine symptom-fix strategies (full definitions in Appendix A). Table 5 and Table 6 present the strategy name and per-tool counts for root-cause fixes and symptom fixes, respectively.

Among each tool's root-cause fixes (Table 5), the per-tool category distributions are broadly similar: both lean on bound checks (35% for Codex, 33% for Kumushi) and API-contract enforcement. The one notable difference is invariant validation, where Kumushi produces three times as many fixes as Codex (12 vs. 4). This is consistent with Kumushi's diversified fault localization more often surfacing the upstream producer where the invalid input enters the system, rather than anchoring at the downstream consumer where the crash surfaces.

Table 5: Root-cause fix strategies ($n = 248$ after audit).

Category	Codex	Kumushi	Total
Bound check	42	43	85
API contract	21	21	42
Size arithmetic	20	19	39
Control-flow fix	13	12	25
Ownership repair	11	12	23
Invariant validation	4	12	16
State sync	7	7	14
Type representation	1	3	4
Total	119	129	248

Among symptom fixes (Table 6), the two tools diverge more sharply. Codex produces nearly twice as many crash-site guards as Kumushi (13 vs. 7) and is alone in producing trigger blocks (3

vs. 0); both are narrow patches that silence the observed failure without enforcing the general invariant. KUMUSHI's symptom fixes spread more evenly across categories with no single failure mode dominating, suggesting that when KUMUSHI does not reach the root cause it does so for more varied reasons rather than consistently drifting to the crash site.

Together, the aggregate gap and the categorical asymmetries agree. KUMUSHI more often repairs the defect at its source, while CODEX more often patches at the crash site. RQ3 confirms this difference with a paired within-bug comparison and illustrates its qualitative character through case studies.

Table 6: Symptom-fix strategies ($n = 57$ after audit).

Category	CODEX	KUMUSHI	Total
Crash-site guard	13	7	20
Downstream guard	4	5	9
Value masking	4	4	8
Corrupt-state tolerance	4	4	8
Trigger block	3	0	3
Incomplete coverage	2	1	3
Lifetime masking	2	1	3
Wrong location	1	1	2
Ineffective guard	1	0	1
Total	34	23	57

Inter-rater reliability. For each label we report three statistics in Table 7: Fleiss's κ [12], the raw observed agreement (mean of the three pairwise rater-agreement rates), and the prevalence of the positive class. We compute the patch quality and unrelated-change labels over the 161-bug union and the pairwise winner label over the 144-bug intersection where both tools produced a plausible patch. We treat abstentions as their own category.

On *patch quality*, raw agreement is 0.81/0.86 (CODEX/KUMUSHI) at a positive-class (root-cause fix) prevalence of 0.75/0.83, giving $\kappa = 0.57/0.62$. The relatively low κ reflects prevalence compression rather than rater disagreement [7]: when the positive class dominates, chance agreement is high and κ deflates even when raters largely agree. The root-cause vs. symptom-fix label, assigned after raters first identified each bug's root cause, is reproducible across rounds. On *unrelated change*, raw agreement is 0.84/0.74 at a near-balanced prevalence of 0.25/0.32, giving $\kappa = 0.64/0.47$. The moderate κ for KUMUSHI unrelated changes reflects the inherent subjectivity in judging whether an edit is within or outside the scope of the defect, rather than systematic rater disagreement.

Manual labeling surfaces two classes of differences the oracles in RQ1 could not detect: an aggregate gap on root-cause-fix and unrelated-edit rates, and categorical asymmetries in the kinds of repairs each tool produces. We characterize each below. RQ3 addresses whether the aggregate gap is statistically significant, using a paired within-bug comparison that has more statistical power than comparing the total counts at this sample size. RQ2 surfaces what differs between the two tools. Whether those differences translate into a per-bug preference, and which tool that preference favors, is the question RQ3 answers.

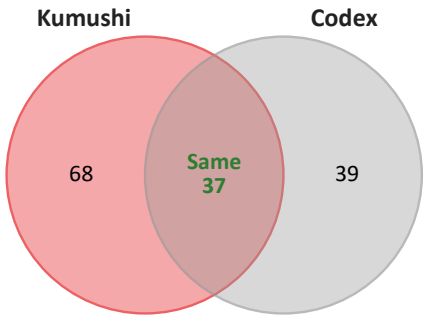

Figure 3: Human preference between KUMUSHI and CODEX plausible patches. Raters prefer KUMUSHI for 68 patches and CODEX for 39, with 37 rated equivalent.

Table 7: Inter-rater agreement on RQ2 per-patch labels and RQ3 pairwise winner label. *Prev. (+)*: positive-class prevalence among non-missing ratings. *Agree*: mean pairwise observed agreement. κ: Fleiss's chance-corrected agreement.

Label (positive class)	Tool	Prev. (+)	Agree	κ
Patch quality (root cause)	CODEX	0.75	0.81	0.57
	KUMUSHI	0.83	0.86	0.62
Unrelated change (`True`)	CODEX	0.25	0.84	0.64
	KUMUSHI	0.32	0.74	0.47
Winner (pairwise)	—	—	0.68	0.54

4.5 RQ3: Pairwise Patch Quality Analysis

To answer RQ3, three raters apply the pairwise comparison rubric (Section 4.2.2) to the 144-bug intersection where both KUMUSHI and CODEX produced a plausible patch, ranking correctness first and comprehensiveness second. This paired design removes bug-difficulty noise and gives the test enough statistical power to confirm the quality gap that RQ2 could only suggest directionally.

Figure 3 summarizes the outcome: KUMUSHI is preferred on 68 (47%), CODEX on 39 (27%), and the two are tied on 37 (26%). Among the 107 decisive bugs where raters reached a clear preference, KUMUSHI wins 63.6% of comparisons; a sign test gives $p = 0.0065$ with a 95% confidence interval of (54.4%, 72.7%) for KUMUSHI's win-rate, confirming that the directional gap observed in RQ2 is statistically significant after controlling for bug difficulty. Inter-rater agreement on the winner label is moderate ($\kappa = 0.54$, Table 7), so the pairwise judgment is reproducible across raters and not driven by a single rater's call.

In the majority of bugs where KUMUSHI is preferred, its diversified fault localization surfaces the producer-side defect alongside the ASan-named crash site, allowing KUMUSHI to repair the invariant rather than guard the symptom; Cases A and B illustrate this pattern. Case C illustrates a different dynamic: a CODEX-wins case in which the decisive evidence emerged only after synthesizing auxiliary analysis code at repair time, a capability outside KUMUSHI's design.

4.5.1 Case A: libxml2 Cursor Underflow Across Halt Boundary. We use Case A as an example to show that Codex misses the root cause location because of missing dynamic information.

The bad state originates in `xmlHaltParser` at `parserInternals.c:289`; on a failed buffer grow, it assigns the parser cursor to a one-byte global, `ctxt->input->cur = BAD_CAST""`, leaving `base == cur == end` on a one-byte string. The mutation that violates the post-halt invariant occurs in `xmlParseNCNameComplex` (`parser.c:3434`); when its lookahead returns `c == 0` after a halt, it computes `ctxt->input->cur -= l` with `l > cur - base`, underflowing below the global. ASan reports the subsequent read at `xmlParseQName` (`parser.c:8867`), where the caller dereferences the underflowed cursor while testing for `':'`. The producer site, the underflow site, and the crash site each sit in a distinct function.

```
// Producer (parserInternals.c:289):
//   xmlHaltParser leaves cur on a 1-byte
//   global -- base == cur == end
  ctxt->input->cur  = BAD_CAST"";
  ctxt->input->base = ctxt->input->cur;
  ctxt->input->end  = ctxt->input->cur;

// Underflow site (xmlParseNCNameComplex):
  if (c == 0) {

// Codex (SF-4, corrupt_state_tolerance):
+   if (ctxt->instate == XML_PARSER_EOF)
+     return(NULL);
    ctxt->input->cur -= l;
    GROW;
+   if (ctxt->instate == XML_PARSER_EOF) {
+     ctxt->input->cur = ctxt->input->base;
+     return(NULL);
+   }

// Kumushi (RCF-1, bound_check):
+   ptrdiff_t back = l;
+   if (ctxt->input->cur
+       - ctxt->input->base < back)
+     back = ctxt->input->cur
+          - ctxt->input->base;
+   ctxt->input->cur -= back;
    GROW;
    if (ctxt->instate == XML_PARSER_EOF)
      return(NULL);
    ctxt->input->cur += back;
    c = CUR_CHAR(l);
  }
```

Listing 1: Both tools patch `parser.c:3434`; the asymmetry is in mechanism. Codex guards the EOF-state transitions around the underflow site (SF-4 `corrupt_state_tolerance`); Kumushi clamps the cursor arithmetic so the subtraction cannot underflow (RCF-1 bound_check).

Kumushi's Stage-2 ranking surfaced `xmlHaltParser` as a top FoI on the object-origin tag, with the line `ctxt->input->cur = BAD_CAST""` pre-loaded into first-turn context. With the post-halt cursor state visible, the agent recognized the invariant `cur >= base` and patched the underflow site by clamping the subtraction at the cursor extent with `back = min(l, cur - base)` (RCF-1 bound_check, Listing 1). The clamp enforces `cur >= base` at the subtraction site, so the underflow cannot occur whether or not the parser's `instate` has transitioned to EOF.

Codex's exploration anchored at the ASan-named consumer and walked the call chain through `xmlParseNCName` into `xmlParseNCNameComplex`. Codex grep'd the codebase for `input->cur -=`, found one occurrence at `parser.c:3434`, and read that as proof the bug lived only there; it never opened `xmlHaltParser`, even though the ASan report named the producer file as hosting the underflowed buffer. Without the halt-state fact, Codex framed the bug as an EOF-transition flaw and patched the same line with `instate` guards around `GROW`, including a post-`GROW` reset of `cur` to `base` that re-aligns the cursor after the corrupt state has already been created rather than preventing its creation (SF-4 `corrupt_state_tolerance`, Listing 1). The guards fire only when `instate` has reached EOF, so any halt path that reaches the subtraction before that transition does so unguarded.

Case A shows that two agents can patch the same line under different invariants, with Stage-2 evidence deciding which invariant gets encoded. This sharpens the localization bottleneck named in observation (1) of section 2.3. Without dynamic information about the upstream producer, the Codex reaches the patch site but cannot recover the invariant that the producer violates.

4.5.2 Case B: libsndfile IMA-ADPCM Block Decode. We use Case B as an example to show that Codex reaches the upstream invariant in its reasoning, then drops it after a context-expansion step pulls in irrelevant code.

The defect is a parser-boundary trust violation in the WAV/W64 IMA-ADPCM reader. The nibble-unpack loop in `wavlike_ima_decode_block` assumes its payload is an exact multiple of $4 \times$ `channels` bytes (eight nibbles per channel per iteration). A malformed `fmt` chunk advertising an inconsistent block-alignment makes the cached block geometry, sample count, and decode loop disagree, so the loop walks past the sample buffer. The producer is `ima_reader_init`, whose WAV/W64 switch arm already validates `samplesperblock` against a derived count but does not check that `blocksize - 4 * channels` is a positive multiple of `4 * channels`.

```
// Nibble-unpack: layout invariant
while (blockindx < blocksize) {
  for (chan ...) {
    for (k = 0; k < 4; k++)
      samples[indx] = ...;  // OOB
} }

// Codex (SF-1, crash_site_guard):
+ if (indxstart >= total) break;
  for (chan ...) {
    for (k = 0; k < 4; k++)
+     if (indx < total)
        samples[indx] = ...;
  }

// Kumushi (RCF-6, invariant_validation):
+ if (blockalign < 4 * channels ||
+     (blockalign - 4 * channels)
+      % (4 * channels) != 0)
+   return SFE_WAV_BAD_BLOCKALIGN;
```

Listing 2: Codex states the upstream invariant but does not apply it, guarding writes at the crash site instead (SF-1 `crash_site_guard`). Kumushi encodes the same invariant at the producer (RCF-6 `invariant_validation`).

Kumushi's Stage-2 ranking placed `ima_reader_init` at the top of the FoI list, above the ASan-named `wavlike_ima_decode_block`,

contributed by CodeQL's intra-procedural dataflow tag and the dynamic call trace; the surrounding switch arm and the existing `samplesperblock` guard were pre-loaded into first-turn context. The agent extended the same arm with the missing divisibility precondition, returning `SFE_WAV_BAD_BLOCKALIGN` when the input violates the nibble-layout invariant (RCF-6 `invariant_validation`, Listing 2).

Codex's trace identifies the same invariant and then drops it. After reading the decode loop and the init function, Codex stated the upstream check explicitly, that `(blockalign - 4*channels)` should be divisible by `(4*channels)`. After a call-graph sweep on `decode_block` broadened context with the encoder, the read-block helper, and the seek paths, Codex dropped the divisibility precondition in favor of in-loop bounds clamping. Only the floor check `blocksize < 4 * channels` survived into the emitted patch, which guards each indexed write at the consumer site rather than rejecting the malformed geometry upstream (SF-1 `crash_site_guard`, Listing 2).

Case B instantiates observation (2) of section 2.3. When unfiltered context expands during exploration, even a correctly identified upstream check can be displaced by off-path code patterns. Kumushi's Stage-2 ranking places the upstream initializer at the agent's first read, so the call-graph drift that displaces the right invariant in Codex's exploration never has the chance to occur.

4.5.3 Case C: mruby Heap Buffer Overflow. We use Case C as an example to show that a defect that lies outside Kumushi's reach. The root cause resides in the compiler's bytecode generator, separated from the crash by thousands of frames in the call graph. This places it outside the scope of Kumushi's static analysis and renders it indistinguishable from normal behavior under dynamic analysis.

The proof of concept `~(:regex or 1*1)` triggers a heap-buffer-overflow at `vm.c:1120`, where the VM indexes a one-entry literal pool with `b = 1`. The crash site, however, is not the defect site. The underlying bug is in `gen_uniop` (`codegen.c:1137`), the only unary peephole rewrite path that lacks a `no_peephole()` guard. As Listing 3 shows, the failing case rewrites `OP_LOADI_1` (2 bytes) into `OP_LOADINEG` (3 bytes) after an `OP_JMPIF` target has already been patched. The jump target therefore remains at the old byte offset and lands in the middle of the rewritten instruction, so the VM decodes operand byte `0x02` as a fresh opcode and eventually reads past the literal pool.

```
before rewrite:
  2: OP_JMPIF    r3, +8    // target = 10
  8: OP_LOADI_1  r3        // 2 bytes
 10: <next instruction>
after gen_uniop folds ~1 to -2:
  2: OP_JMPIF    r3, +8    // still targets 10
  8: OP_LOADINEG r3, 2     // 3 bytes
 10: 0x02                  // jump lands in operand byte
```

Listing 3: Bytecode drift after peephole rewrite.

`gen_uniop` runs during compilation, hundreds of frames before `vm.c:1120`, and the bytecode buffer that links compiler to VM is opaque to static dataflow. Kumushi's evidence sources score candidates by uniqueness in the trace and connectivity through dataflow tags, but no dataflow edge leads back to the rewriter

```
static mrb_bool
gen_uniop(codegen_scope *s, mrb_sym sym, uint16_t dst)
{
  if (no_peephole(s)) return FALSE;
  struct mrb_insn_data data = mrb_last_insn(s);
  ...
}
```

Listing 4: Minimal guard for unary peephole folding.

and bytecode-generation routines appear throughout the trace, so neither signal surfaces it. Kumushi therefore cannot localize the root cause and emits a patch that validates the decoded literal-pool index `b` before the VM reads `pool[b]` (SF-1 `crash_site_guard`). The patch suppresses the observed crash but does not repair the violated compiler invariant.

The Codex run, by contrast, generated a bytecode disassembler during repair, exposed the one-byte drift in Listing 3, and traced the problem back to a missing `no_peephole()` condition on the unary folding path. Its patch inserts the guard at the caller-side fold site in `codegen.c`, blocking the same unguarded fold path as the ground-truth fix, which places the check inside `gen_uniop` (RCF-4 `control_flow_fix`, Listing 4).

Case C demonstrates that crash-anchored ranking misses defects whose decisive evidence emerges only at repair time. Codex reached the rewriter because a general-purpose coding agent can construct a custom program-analysis tool when its reasoning requires one. Kumushi's current Stage-2 evidence sources have no edge to follow back across the encoded-bytecode boundary, so this class of bugs is unreachable by the present pipeline.

Takeaway. In summary, the three cases identify the factors that drive the quality gap. Kumushi's evidence-weighted ranking surfaces upstream producers that the ASan stack does not name (Case A) and places them at the agent's first read, preempting the call-graph drift that displaces the right invariant during Codex's exploration (Case B). Both moves push patches toward root-cause categories (RCF-1 `bound_check`, RCF-6 `invariant_validation`) rather than crash-site or downstream guards. Root-cause-anchored patching's strength comes from grounding the agent in evidence the LLM cannot recover by reading code alone. Its limitation appears when the decisive evidence sits behind a boundary that no static or dynamic source can cross (Case C), in which case a general-purpose agent that synthesizes its own analysis tool can outperform it.

5 Related Work

Automated Program Repair. Automated program repair (APR) has been extensively studied for generating patches that transform defective or vulnerable programs into correct or secure ones. Existing APR techniques can be broadly categorized into search-based, template-based, semantic-based, and learning-based approaches.

Search-based repair. Search-based approaches formulate patch generation as a search problem over a space of program variants [17, 21]. GenProg [21], for example, generates candidate variants through mutation and crossover and evaluates them against test suites. Fix2Fit [17] further incorporates fuzzing to refine the search space and guide the generation of crash-avoiding patches.

Template-based repair. Template-based approaches rely on predefined patch patterns or repair templates to produce candidate fixes [15, 29, 33, 46, 52]. BovInspector [15] uses safe API templates to repair buffer-overflow vulnerabilities. RegexScalpel [33] employs regular-expression templates to mitigate ReDoS vulnerabilities. VFix [52] uses two sets of repair templates to address null-pointer-dereference vulnerabilities.

Semantic-based repair. Semantic-based approaches exploit program semantics, vulnerability properties, or inferred constraints to synthesize patches [16, 18, 23, 25, 44, 48, 60]. SAVER [23] constructs constraints from variable states and control-/data-flow graphs to generate repairs. ExtractFix [18] derives crash-free constraints and uses them to synthesize candidate patches. VulnFix [60] infers program invariants and leverages them to guide vulnerability repair.

Learning-based repair. More recently, learning-based APR techniques have used neural models and large language models to translate vulnerable code into repaired code [8, 13, 14, 22, 62]. VulRepair [14], for instance, fine-tunes CodeT5 [49] for vulnerability repair. Other work has explored prompting strategies that improve the ability of LLMs to generate security patches [30, 34, 42].

Root Cause Analysis. Root cause analysis aims to identify the program locations and execution conditions responsible for a failure or vulnerability. Prior work has shown that combining static and dynamic analysis can improve the localization of root causes and provide useful guidance for patch generation [52]. Existing techniques diagnose bugs and vulnerabilities using statistical, trace-based, and semantic information.

Statistical root cause analysis. Statistical techniques rank candidate program locations according to their correlation with failing executions [1, 6, 28, 41]. AURORA [6] compares traces from crashing and non-crashing inputs and uses statistical analysis to identify likely root-cause locations. Such techniques are effective when sufficient execution evidence is available, but their accuracy depends on the quality and diversity of the collected traces and may degrade when failures are triggered by complex semantic conditions.

Semantic root cause analysis. Semantic techniques use program reasoning to explain how a proof-of-concept input reaches a vulnerable state [53, 54]. ARCUS [54], for example, builds on symbolic execution with angr [45] to trace a vulnerability and identify its root cause. Compared with statistical techniques, semantic approaches can provide more precise explanations of vulnerability-triggering conditions, but they often depend on heavyweight program analyses and may face scalability challenges on complex programs.

6 Discussion

Oracle-Based Evaluation Is No Longer Sufficient. The automated vulnerability repair community has long acknowledged that replay-based oracles are an imperfect proxy for patch quality[24]. A patch may suppress the observed failure while leaving the underlying defect intact. The community recognized this limitation but tolerated it. When the field's best tools produced meaningfully different oracle-passing rates, oracle-based evaluation could still discriminate between them. PatchAgent, LoopRepair, and earlier prompt-based approaches produce meaningfully different plausible patch rates (71.9%, 2.2%, respectively), and oracle metrics alone are sufficient to rank them.

We find that oracle-based evaluation no longer separates the latest LLM-based agents. Our evaluation shows that Kumushi, a specialized root-cause-driven agent, and Codex, a general-purpose coding agent, are statistically indistinguishable under all three oracles, yet expert assessment reveals a significant quality gap in favor of Kumushi ($p = 0.0065$). This is not an artifact of our benchmark or our tools. It reflects a structural shift in agent capability. General-purpose coding agents now reliably produce oracle-passing patches across a wide range of vulnerability types, saturating the metric the community has long relied on.

The implication for the APR community is direct. As coding agents continue to improve, oracle-passing rate will fail to separate tools that produce root-cause fixes from those that produce symptom patches. Therefore, they require evaluation methods capable of assessing *what* a patch fixes, not *whether* the observed crash is suppressed. The two-tier metric introduced in this work is one step in that direction, pairing automated oracles with structured expert assessment of patch comprehensiveness. We hope future work can build automated proxies for expert judgment that can flag symptom patches before they are counted as successes.

Limitations of Crash-Anchored Localization. Kumushi's root-cause-driven design assumes the defect site is reachable from the crash site through static or dynamic evidence. This assumption holds for the majority of bugs in our benchmark, where the producer, the underflow site, and the crash site are connected by dataflow edges or execution traces that Stage 1 can collect and Stage 2 can rank. Case C shows where it breaks down. When the defect site is separated from the crash by an opaque boundary, such as an encoded bytecode buffer or an inter-process channel, no static dataflow edge and no dynamic trace leads back to it. Evidence-weighted ranking cannot surface what no evidence source can reach.

This is a structural limit of crash-anchored localization, not an implementation gap specific to Kumushi. Any approach that grounds localization in crash-site evidence will face the same issue. Case C also illustrates what can fill the gap. A general-purpose agent that synthesizes its own analysis tools at repair time can cross boundaries that pre-computed evidence sources cannot. The tradeoff is that such agents operate without the focused context that evidence-weighted ranking provides, making them more susceptible to the drift pattern that motivates Kumushi's design in the first place.

Two additional scope constraints bound the current prototype. First, the quality of Stage 1's candidate pool depends on how many distinct crash paths the fuzzer finds within the time budget. Bugs with narrow trigger conditions may not diversify enough to expose root-cause-correlated functions beyond the crash stack. Second, the pipeline currently assumes sanitizer instrumentation and fuzzing infrastructure, which limits direct applicability to languages and ecosystems where such tooling is unavailable. We leave extending Kumushi to them to future work.

Toward Automated Patch Quality Validation. Manual expert assessment, as applied in our tier-2 evaluation, does not scale to the patch volumes that LLM-based repair pipelines now produce. Our evaluation required approximately 16 minutes per patch [47], and even with eight raters working in parallel, coverage was bounded

by the 144-bug intersection where both tools produced a plausible patch. As repair systems improve and benchmarks grow, this bottleneck will become more acute.

Our taxonomy of symptom-fix strategies in Table 6 suggests a path forward. The nine failure modes we identify are not arbitrary. Crash-site guards, downstream guards, and trigger blocks share a structural signature in which the patch modifies code at or near the ASan-reported crash site without touching any function that the evidence-weighted ranking would have flagged as a root-cause candidate. Value masking and corrupt-state tolerance leave the producer intact while suppressing its observable effect. These structural fingerprints are in principle detectable without human review. An automated checker can compare the patch's edit locations against the FoI ranking, determine whether the modified code touches a producer or only a consumer, and flag patches whose diffs are confined to the crash stack. We hope future work uses this taxonomy as a foundation for automated validators that can triage symptom patches before they reach a human reviewer, closing the scalability gap that tier-2 assessment currently cannot bridge.

7 Conclusion

This work introduces Kumushi, a vulnerability patching system that addresses two limitations of existing LLM-based tools: (1) static-only fault localization that fails to pinpoint root causes, and (2) unfiltered context that drives agents toward crash-site suppression instead of root-cause repair. In particular, Kumushi uses two innovative techniques: diversified fault localization and evidence-weighted context ranking. Based on our evaluation of patching 178 C/C++ vulnerabilities, Kumushi generates plausible patches for over 90% of vulnerabilities under the replay-based oracle and produces more high-quality patches than commercial general-purpose coding agents under blind human expert assessment, which outperforms the state-of-the-art automatic root cause repair techniques in both academia and industry.

References

[1] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. 2009. A practical evaluation of spectrum-based fault localization. *J. Syst. Softw.* 82 (2009), 1780–1792.

[2] anthropic. 2025. Effective context engineering for AI agents. https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents. (2025). Accessed: 2026-04-20.

[3] Anthropic. 2026. Claude Code. https://github.com/anthropics/claude-code. (2026). Accessed: 2026-03-31.

[4] Anthropic. 2026. Project Glasswing. https://www.anthropic.com/glasswing. (2026).

[5] Tim Blazytko, Moritz Schlögel, Cornelius Aschermann, Ali Abbasi, Joel Frank, Simon Wörner, and Thorsten Holz. 2020. AURORA: statistical crash analysis for automated root cause explanation. In *Proceedings of the 29th USENIX Conference on Security Symposium (SEC'20)*. USENIX Association, USA, Article 14, 18 pages.

[6] Tim Blazytko, Moritz Schlögel, Cornelius Aschermann, Ali Reza Abbasi, Joel Cameron Frank, Simon Wörner, and Thorsten Holz. 2020. AURORA: Statistical Crash Analysis for Automated Root Cause Explanation. In *USENIX Security Symposium*.

[7] Ted Byrt, Janet Bishop, and John B Carlin. 1993. Bias, prevalence and kappa. *Journal of clinical epidemiology* 46, 5 (1993), 423–429.

[8] Zimin Chen, Steve Kommrusch, and Monperrus Martin. 2022. Neural Transfer Learning for Repairing Security Vulnerabilities in C Code. *IEEE Transactions on Software Engineering* 49 (2022), 147–165.

[9] Jianlei Chi, YunHuan Qu, Ting Liu, Qinghua Zheng, and Heng Yin. 2022. SeqTrans: Automatic Vulnerability Fix Via Sequence to Sequence Learning. *IEEE Transactions on Software Engineering* 49 (2022), 564–585.

[10] Tejinder Dhaliwal, Foutse Khomh, and Ying Zou. 2011. Classifying field crash reports for fixing bugs: A case study of Mozilla Firefox. In *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance (ICSM '11)*. IEEE Computer Society, USA, 333–342. https://doi.org/10.1109/ICSM.2011.6080800

[11] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: combining incremental steps of fuzzing research. In *Proceedings of the 14th USENIX Conference on Offensive Technologies (WOOT'20)*. USENIX Association, USA, Article 10, 1 pages.

[12] Joseph L Fleiss. 1971. Measuring nominal scale agreement among many raters. *Psychological bulletin* 76, 5 (1971), 378.

[13] Michael Fu, Van-Anh Nguyen, Chakkrit Kla Tantithamthavorn, Dinh Q. Phung, and Trung Le. 2024. Vision Transformer Inspired Automated Vulnerability Repair. *ACM Transactions on Software Engineering and Methodology* 33 (2024), 1 – 29.

[14] Michael Fu, Chakkrit Kla Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Q. Phung. 2022. VulRepair: a T5-based automated software vulnerability repair. *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2022).

[15] Fengjuan Gao, Linzhang Wang, and Xuandong Li. 2016. BovInspector: Automatic inspection and repair of buffer overflow vulnerabilities. *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2016), 786–791.

[16] Qing Gao, Yingfei Xiong, Yaqing Mi, Lu Zhang, Weikun Yang, Zhao fa Zhou, Bing Xie, and Hong Mei. 2015. Safe Memory-Leak -Fixing for C Programs. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering* 1 (2015), 459–470.

[17] Xiang Gao, Sergey Mechtaev, and Abhik Roychoudhury. 2019. Crash-avoiding program repair. *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (2019).

[18] Xiang Gao, Bo Wang, Gregory J Duck, Ruyi Ji, Yingfei Xiong, and Abhik Roychoudhury. 2021. Beyond tests: Program vulnerability repair via crash constraint extraction. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 2 (2021), 1–27.

[19] Github. 2026. CodeQL. https://codeql.github.com/. (2026). Accessed: 2026-03-31.

[20] google. 2026. Google OSS-Fuzz. https://github.com/google/oss-fuzz. (2026). Accessed: 2026-04-20.

[21] Claire Le Goues, Thanhvu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38 (2012), 54–72.

[22] Jacob A. Harer, Onur Ozdemir, Tomo Lazovich, Christopher P. Reale, Rebecca L. Russell, Louis Y. Kim, and Peter Chin. 2018. Learning to Repair Software Vulnerabilities with Generative Adversarial Networks. *The Thirty-Second Annual Conference on Neural Information Processing Systems (NIPS)* (2018).

[23] Seongjoon Hong, Junhee Lee, Jeongsoo Lee, and Hakjoo Oh. 2020. SAVER: Scalable, Precise, and Safe Memory-Error Repair. *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)* (2020), 271–283.

[24] Yiwei Hu, Zhen Li, Kedie Shu, Sheng Guan, Deqing Zou, Shouhuai Xu, Bin Yuan, and Hai Jin. 2025. SoK: Automated Vulnerability Repair: Methods, Tools, and Assessments. In *USENIX Security Symposium*.

[25] Zhen Huang, David Lie, Gang Tan, and Trent Jaeger. 2019. Using Safety Properties to Generate Vulnerability Patches. *2019 IEEE Symposium on Security and Privacy (SP)* (2019), 539–554.

[26] Zhi Yu Jiang, Shuitao Gan, Adrián Herrera, Flavio Toffalini, Lucio Romerio, Chaojing Tang, Manuel Egele, Chao Zhang, and Mathias Payer. 2022. Evocatio: Conjuring Bug Capabilities from a Single PoC. *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (2022).

[27] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? *The Twelfth International Conference on Learning Representations* abs/2310.06770 (2024).

[28] James A. Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering* (2005).

[29] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. *2013 35th International Conference on Software Engineering (ICSE)* (2013), 802–811.

[30] Youngjoon Kim, Sunguk Shin, Hyoungshick Kim, and J. Yoon. 2025. Logs In, Patches Out: Automated Vulnerability Repair via Tree-of-Thought LLM Analysis. In *USENIX Security Symposium*.

[31] Hwiwon Lee, Ziqi Zhang, Hanxiao Lu, and Lingming Zhang. 2025. SEC-bench: Automated Benchmarking of LLM Agents on Real-World Software Security Tasks. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*. https://sec-bench.github.io/index.html

[32] Frank H. Li and Vern Paxson. 2017. A Large-Scale Empirical Study of Security Patches. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017).

[33] Yeting Li, Yecheng Sun, Zhiwu Xu, Jialun Cao, Yuekang Li, Rongchen Li, Haiming Chen, Shing Chi Cheung, Yang Liu, and Yang Xiao. 2022. RegexScalpel: Regular Expression Denial of Service (ReDoS) Defense by Localize-and-Fix. In *USENIX Security Symposium*.

[34] Yu Nong, Haoran Yang, Long Cheng, Hongxin Hu, and Haipeng Cai. 2025. APPATCH: Automated Adaptive Prompting Large Language Models for Real-World Software Vulnerability Patching. In *USENIX Security Symposium*.
[35] NVD. 2026. CVE-2022-1286. https://nvd.nist.gov/vuln/detail/cve-2022-1286. (2026). Accessed: 2026-03-31.
[36] nvd. 2026. national vulnerability database. https://nvd.nist.gov/. (2026). Accessed: 2026-04-20.
[37] OpenAI. 2025. Codex. https://github.com/openai/codex. (2025). Accessed: 2026-03-31.
[38] OpenAI. 2026. Introducing GPT-5.2. https://openai.com/index/introducing-gpt-5-2/. (2026). Accessed: 2026-03-31.
[39] OpenHands. 2026. OpenHands. https://github.com/OpenHands/OpenHands. (2026). Accessed: 2026-03-31.
[40] oss-fuzz. 2022. hunspell:affdicfuzzer: Heap-buffer-overflow in AffixMgr::cpdpat_check. https://issues.oss-fuzz.com/issues/42518895. (2022). Accessed: 2026-03-31.
[41] Younggi Park, Hwiwon Lee, Jinho Jung, Hyungjoon Koo, and Huy Kang Kim. 2024. Benzene: A Practical Root Cause Analysis System with an Under-Constrained State Mutation. *2024 IEEE Symposium on Security and Privacy (SP)* (2024), 1865–1883.
[42] Hammond A. Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2023. Examining Zero-Shot Vulnerability Repair with Large Language Models. *2023 IEEE Symposium on Security and Privacy (SP)* (2023), 2339–2356.
[43] Judea Pearl. 2014. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Elsevier.
[44] Ridwan Shariffdeen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. 2021. Concolic program repair. *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (2021).
[45] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*.
[46] Sooel Son, Kathryn S. McKinley, and Vitaly Shmatikov. 2013. Fix Me Up: Repairing Access-Control Bugs in Web Applications. In *Network and Distributed System Security Symposium*.
[47] Yida Tao, Jindae Kim, Sunghun Kim, and Chang Xu. 2014. Automatically generated patches as debugging aids: a human study. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2014).
[48] Rijnard van Tonder and Claire Le Goues. 2018. Static Automated Program Repair for Heap Properties. *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)* (2018), 151–162.
[49] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. *ArXiv* abs/2109.00859 (2021).
[50] Zhun Wang, Tianneng Shi, Jingxuan He, Matthew Cai, Jialin Zhang, and Dawn Song. 2026. CyberGym: Evaluating AI Agents' Real-World Cybersecurity Capabilities at Scale. *The Fourteenth International Conference on Learning Representations* (2026).
[51] Haolai Wei, Liwei Chen, Zhijie Zhang, Gang Shi, and Dan Meng. 2024. Sleuth: A Switchable Dual-Mode Fuzzer to Investigate Bug Impacts Following a Single PoC. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 730–742. https://doi.org/10.1145/3650212.3680316
[52] Xuezheng Xu, Yulei Sui, Hua Yan, and Jingling Xue. 2019. VFix: Value-Flow-Guided Precise Program Repair for Null Pointer Dereferences. *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)* (2019), 512–523.
[53] Carter Yagemann, Simon Pak Ho Chung, Brendan Saltaformaggio, and Wenke Lee. 2021. Automated Bug Hunting With Data-Driven Symbolic Root Cause Analysis. *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (2021).
[54] Carter Yagemann, Matthew Pruett, Simon Pak Ho Chung, Kennon Bittick, Brendan Saltaformaggio, and Wenke Lee. 2021. ARCUS: Symbolic Root Cause Analysis of Exploits in Production Systems. In *USENIX Security Symposium*.
[55] R.R. Yager. 1988. On ordered weighted averaging aggregation operators in multicriteria decisionmaking. *IEEE Transactions on Systems, Man, and Cybernetics* 18, 1 (1988), 183–190. https://doi.org/10.1109/21.87068
[56] Zhenlei Ye, Xiaobing Sun, Sicong Cao, Lili Bo, and Bin Li. 2026. Well Begun is Half Done: Location-Aware and Trace-Guided Iterative Automated Vulnerability Repair. *Proceedings of the IEEE/ACM 48th International Conference on Software Engineering* (2026).
[57] Zheng Yu, Ziyi Guo, Yuhang Wu, Jiahao Yu, Meng Xu, Dongliang Mu, Yan Chen, and Xinyu Xing. 2025. PATCHAGENT: A Practical Program Repair Agent Mimicking Human Expertise. In *USENIX Security Symposium*.
[58] Zheng Yu, Wenxuan Shi, Xin Sun, Zheyun Feng, Meng Xu, and Xinyu Xing. 2026. Patch Validation in Automated Vulnerability Repair. https://arxiv.org/pdf/2603.06858
[59] Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Reddy Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. 2026. Agentic Context Engineering: Evolving Contexts for Self-Improving Language Models. In *Proceedings of the Fourteenth International Conference on Learning Representations (ICLR)*.
[60] Yuntong Zhang, Xiang Gao, Gregory J. Duck, and Abhik Roychoudhury. 2022. Program vulnerability repair via inductive inference. *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (2022).
[61] Yuntong Zhang, Jiawei Wang, Dominic Berzin, Martin Mirchev, and Abhik Roychoudhury. 2026. Fixing Security Vulnerabilities with Agentic AI in OSS-Fuzz. In *Proceedings of the 48th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '26)*. ACM, Rio de Janeiro, Brazil.
[62] Xin Zhou, Kisub Kim, Bowen Xu, Donggyun Han, and David Lo. 2024. Out of Sight, Out of Mind: Better Automatic Vulnerability Repair by Broadening Input Ranges and Sources. *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)* (2024), 1071–1083.
[63] Aleksandr Zverianskii, Ashley Zhang, Jacob Clyne, Antía Garcia, Fazl Barez, and Shriyash Upadhyay. 2026. Code Review Bench. (2026). https://github.com/withmartian/code-review-benchmark

A Repair Strategies Definition

The definition of root cause fix strategies is in Table 8. The definition of symptom fix strategies is in Table 9.

B Benchmark

Table 10 shows all the bugs in our dataset.

Mechanism	Definition
bound_check	Enforces the correct index, cursor, remaining-length, or object extent before access (sizes, lengths, indices, offsets vs. capacities).
api_contract	Enforces the intended boundary contract for nullable returns, EOF/error propagation, output parameters, string termination, or memory API semantics.
size_arithmetic	Repairs overflow, underflow, divide-by-zero, capacity accounting, or exact bytes-needed calculations before the result is trusted.
control_flow_fix	Repairs the predicate, iteration, recursion, parser transition, or error-path branch that produces the bad state.
ownership_repair	Repairs the ownership or lifetime rule that creates stale, double-freed, or dangling state (UAF, double-free, freed-pointer reuse).
invariant_validation	Rejects or normalises invalid input at the producer/parse boundary before it becomes trusted internal state (non-zero divisor, plausible count, valid enum).
state_sync	Keeps derived state, cached metadata, counters, lengths, or helper-visible buffer extents synchronised after mutation.
type_representation	Repairs signedness, integer width, truncation, or producer/consumer type-domain agreement.

Table 8: Root-cause fix Strategies

Failure mode	Definition
crash_site_guard	Immediate guard around the observed dereference, division, shift, copy, or read while the upstream invalid state remains intact.
downstream_guard	Patches a caller or consumer while leaving the producer or shared helper defect intact, so equivalent callers still trigger the same bug.
value_masking	Hides the bad value by clamping, replacing, padding, inserting a dummy object, or forcing a sentinel/default.
corrupt_state_tolerance	Detects or tolerates corrupted state after it has already been created instead of preventing creation.
trigger_block	Blocks one observed input shape (the proof-of-concept) without enforcing the general invariant.
incomplete_coverage	Fixes one crash location while leaving another path for the same root cause unpatched.
lifetime_masking	Pins, retains, or avoids freeing an object to suppress a lifetime symptom while leaving the ownership transition wrong.
wrong_location	Modifies code that is not on the causal path, or addresses a different bug than the one reported.
ineffective_guard	Adds a check that cannot fire, checks the wrong condition, or enforces a nearby but non-causal invariant.

Table 9: Symptom-fix Strategies

Table 10: Benchmark

Project	Commit	Bug Type	Project	Commit	Bug Type	Project	Commit	Bug Type
assimp	0422dff	heap_buffer_overflow	gpac	a6b6408	heap_out_of_bound	md4c	3478ec6	heap_buffer_overflow_a
assimp	2d44861	unknown_write	gpac	a8bc2c8	heap_out_of_bound	md4c	3478ec6	heap_buffer_overflow_b
assimp	565539b	heap_buffer_overflow	gpac	b1042c3	null_pointer_deref	md4c	db9ab41	heap_buffer_overflow
c-blosc	01df770	heap_buffer_overflow	gpac	b6b6360	null_pointer_deref	md4c	db9ab41	unknown_read
c-blosc	41f3a2e	heap_buffer_overflow	gpac	bb9ee4c	heap_out_of_bound	mruby	0ed3fcf	heap_out_of_bound
c-blosc2	38b23d5	negative_size_param	gpac	be23476	heap_out_of_bound	mruby	4c196db	heap_out_of_bound
c-blosc2	4f6d42a	invalid_free	gpac	ca1b48f	heap_out_of_bound	mruby	55b5261	null_pointer_deref
c-blosc2	6fc4790	heap_buffer_overflow_b	gpac	cc95b16	heap_out_of_bound	mruby	8aec568	use_after_free
c-blosc2	81c2fcd	heap_buffer_overflow_b	gpac	d2de8b5	stack_out_of_bound	mruby	af5acf3	use_after_free
c-blosc2	81c2fcd	heap_buffer_overflow_c	gpac	de7f3a8	heap_out_of_bound	mruby	b4168c9	use_after_free
c-blosc2	aebf2b9	heap_buffer_overflow	gpac	ebedc7a	use_after_free	mruby	bdc244e	null_pointer_deref
c-blosc2	cb15f1b	unknown_read	gpac	fc9e290	heap_out_of_bound	mruby	bf5bbf0	use_after_free
binutils	515f23e	divide_by_zero	h3	f581626	global_buffer_overflow	mruby	c30e6eb	heap_out_of_bound
binutils	c48935d	heap_buffer_overflow	hoextdown	933f9da	heap_buffer_overflow	mruby	d1f1b4e	null_pointe_deref
coreutils	658529a	heap_buffer_overflow	hostap	703c2b6	heap_buffer_overflow	openexr	115e42e	heap_buffer_overflow_a
coreutils	68c5eec	negative_size_param	hostap	8112131	unknown_read	openexr	115e42e	heap_buffer_overflow_b
coreutils	8d34b45	memcpy_param-overlap	hostap	a6ed414	heap_buffer_overflow_a	openexr	7c40603	stack_buffer_overflow
coreutils	ca99c52	global_buffer_overflow	hostap	a6ed414	heap_buffer_overflow_b	radare2	0927ed3	heap_out_of_bound
jasper	3c55b39	heap_buffer_overflow	htslib	dd6f0b7	unknown_read	radare2	0be8f25	heap_out_of_bound
jasper	b9be3d9	integer_overflow	hunspell	1c1f34f	heap_buffer_overflow	radare2	108dc76	heap_out_of_bound
libjpeg-turbo	208d927	heap_buffer_overflow	hunspell	473241e	heap_buffer_overflow	radare2	27fe803	null_pointer_deref
libjpeg-turbo	3212005	null_ptr_dereference	hunspell	6291cac	heap_buffer_overflow_a	radare2	4823451	heap_out_of_bound
libjpeg-turbo	4f24016	stack_buffer_overflow	hunspell	6291cac	heap_buffer_overflow_b	radare2	4b22fc5	null_pointer_deref
libjpeg-turbo	ae8cdf5	heap_buffer_overflow	hunspell	6291cac	stack_buffer_overflow_a	radare2	515e592	heap_out_of_bound
libtiff	0ba5d88	heap_buffer_overflow	hunspell	6291cac	stack_buffer_overflow_b	radare2	5c0bde8	null_pointer_deref
libtiff	2c00d31	divide_by_zero	hunspell	74b08bf	heap_buffer_overflow_a	radare2	72ffc02	null_pointer_deref
libtiff	3144e57	invalid_shift	hunspell	74b08bf	heap_buffer_overflow_b	radare2	7cfd367	use_after_free
libtiff	6a984bf	heap_buffer_overflow	hunspell	74b08bf	heap_double_free	radare2	95b648f	heap_out_of_bound
libtiff	891b1b9	heap_buffer_overflow	hunspell	82b9212	heap_use_after_free	radare2	9650e3c	use_after_free
libtiff	8b6e80f	heap_buffer_overflow	hunspell	ddec95b	heap_buffer_overflow	radare2	9bcc98f	heap_out_of_bound
libtiff	9a72a69	divide_by_zero	irssi	afcb483	heap_use_after_free	radare2	a16cb20	heap_out_of_bound
libtiff	acb5bb9	heap_buffer_overflow	irssi	b472570	heap_buffer_overflow_a	radare2	a58b8d4	heap_out_of_bound
libtiff	c421b99	heap_buffer_overflow	irssi	b472570	heap_buffer_overflow_b	radare2	cf780fd	use_after_free
libtiff	e596d4e	divide-by-zero	krb5	d864d74	heap_buffer_overflow	radare2	d026503	heap_out_of_bound
libtiff	f3069a5	heap_buffer_overflow	libmobi	1297ee0	heap_out_of_bound	radare2	d17a7bd	use_after_free
libxml2	362b322	null_ptr_dereference	libmobi	4b60805	heap_out_of_bound	radare2	d22d160	use_after_free
libxml2	4ea74a4	heap_buffer_overflow	libmobi	afa8ce1	stack_out_of_bound	radare2	eca58ce	heap_out_of_bound
libxml2	8f30bdf	heap_buffer_overflow	libmobi	eafc415	null_pointer_deref	sleuthkit	34f995d	heap_buffer_overflow_a
libxml2	cbb2716	heap_buffer_overflow	libmobi	eb4a262	null_pointer_deref	sleuthkit	34f995d	heap_buffer_overflow_b
libxml2	db07dd6	heap_buffer_overflow	libplist	491a3ac	heap_buffer_overflow_a	sleuthkit	38a13f9	heap_buffer_overflow
gpac	0b29a41	use_after_free	libplist	491a3ac	heap_buffer_overflow_b	sleuthkit	82d254b	heap_buffer_overflow
gpac	112767e	heap_out_of_bound	libplist	491a3ac	heap_use_after_free	sleuthkit	d9b19e1	heap_buffer_overflow
gpac	1b77837	heap_out_of_bound	libsndfile	2b4cc4b	heap_buffer_overflow	wasm3	139076a	use_after_free
gpac	3ffe59c	heap_out_of_bound	libsndfile	4819cad	heap_buffer_overflow	wasm3-harness	0124fd5	heap_buffer_overflow
gpac	4607052	double_free	libsndfile	932aead	negative_size_param	wasm3-harness	355285d	global_buffer_overflow
gpac	4925c40	use_after_free	libsndfile	b706e62	heap_buffer_overflow	wasm3-harness	4f0b769	heap_use_after_free
gpac	49cb88a	heap_out_of_bound	libsndfile	fe49327	heap_buffer_overflow	wasm3-harness	4f0b769	unknown_read
gpac	4c77303	heap_out_of_bound	libtpms	e563166	stack_buffer_overflow	wasm3-harness	4f0b769	unknown_write
gpac	50a60b0	null_pointer_deref	libxml2	20f5c73	global_buffer_overflow	wasm3-harness	bc32ee0	segv_on_unknown_address
gpac	514a3af	stack_out_of_bound	libxml2	5f4ec41	global_buffer_overflow	wasm3-harness	bc32ee0	unknown_read
gpac	6f28c4c	null_pointer_deref	libxml2	7fbd454	global_buffer_overflow	yasm	84be2ee	heap_out_of_bound
gpac	78f5269	heap_out_of_bound	libxml2	9ef2a9a	global_buffer_overflow_a	yasm	9defefa	heap_out_of_bound
gpac	7a6f636	stack_out_of_bound	libxml2	9ef2a9a	global_buffer_overflow_b	yasm	ecb47f1	null_pointer_deref
gpac	7e2cb01	use_after_free	libxml2	b167c73	global_buffer_overflow_a	zstd	0a96d00	heap_buffer_overflow
gpac	7e2e92f	use_after_free	libxml2	b167c73	global_buffer_overflow_b	zstd	2fabd37	global_buffer_overflow
gpac	7edc40f	null_pointer_deref	libxml2	b167c73	global_buffer_overflow_c	zstd	3cac061	heap_buffer_overflow
gpac	89a80ca	heap_out_of_bound	libxml2	b167c73	global_buffer_overflow_d	zstd	6f40571	unknown_read
gpac	8db20cb	heap_out_of_bound	libxml2	b167c73	global_buffer_overflow_e	zstd	d68aa19	heap_buffer_overflow
gpac	8f3088b	null_pointer_deref	lz4	7654a5a	heap_buffer_overflow			
gpac	94cf5b1	stack_out_of_bound	lz4	9d20cd5	invalid_free			

C LLM System Prompt

This appendix reproduces verbatim the system prompt sent to the LLM patch agent. The two placeholders `{crash_report}` and `{rca_summary}` are substituted at run time with the sanitizer report and the ranked Root Cause Analysis summary, respectively.

```
You are an expert C/C++ security researcher and patch developer. Your job is to analyze a vulnerability in a C/C++ project, understand its root cause, and develop a minimal, correct patch that fixes the vulnerability without breaking existing functionality.

## Crash Report (ASan/Sanitizer Output)

{crash_report}

## Root Cause Analysis Summary

{rca_summary}

## How the RCA Pipeline Works

The Root Cause Analysis (RCA) results you see were produced by a multi-stage pipeline that combines static analysis and dynamic tracing to rank candidate functions by their likelihood of containing the bug:

1. **Stack Trace Analysis** (source: STACK_TRACE) -- Parses the ASAN/sanitizer crash report to extract the exact crash location and call stack. These are the highest-confidence candidates -- they are directly on the crash path.

2. **Dynamic Call Tracing** (source: CALL_TRACE) -- Instruments the binary and replays the crashing input to record every function called at runtime. Functions are ranked by their distance from the crash site (closer = more relevant) and call frequency.

3. **Static Variable Dependencies** (source: VAR_DEP) -- Performs static data-flow analysis to find functions that handle data flowing toward the crash location. These may reveal upstream allocation, sizing, or validation functions that are the true root cause.

4. **Fuzzing Coverage Analysis** (source: AURORA) -- Uses fuzzer corpus and crash coverage to score functions by how strongly their coverage correlates with crashes.

Each FOI (Function of Interest) cluster groups related functions from these analyses. Clusters are ranked by a combination of how many independent analyses flagged them and the priority of those sources (STACK_TRACE is highest priority; AURORA varies by score -- check each cluster's reasoning for its confidence level; then VAR_DEP, then CALL_TRACE). A function flagged by multiple analyses ranks higher than one flagged by a single high-priority source. The top-ranked clusters are the most likely to contain the root cause, but the actual buggy code may be in a caller, a callee, or a related helper -- not necessarily the crash site itself.

## Workflow

### Phase 1: Deep Exploration (do NOT edit code yet)
Before making any edits, you MUST thoroughly understand the vulnerability by examining the relationship between the RCA candidates and the bug:

1. **Study the RCA clusters**: The most relevant clusters are shown above with full source code -- review them first. Use `get_rca_results(index)` to view source for any additional clusters listed compactly. Pay attention to the source of each cluster -- clusters flagged by multiple analyses deserve extra attention.

2. **Understand the crash mechanism**: From the crash report, identify the bug type (buffer overflow, use-after-free, null dereference, etc.). Identify the exact memory operation that fails and what value/pointer is invalid.

3. **Trace the call chain**: Use `view_function` to read each function in the crash stack -- it shows the function source, its callees, and global variables. Use `read_source_file` to see surrounding context -- macros, struct definitions, buffer allocation sites, size computations. Look ABOVE the crash function for where the faulty data originates. Also inspect CALLEES of crash-stack functions -- the bug may be in a utility function (byte processing, buffer management, character handling) that the crash-site function calls.

4. **Explore related code**: Use `search_functions` to find related functions by name pattern -- it also searches source file text when the function index has no match, so it can locate macros, struct definitions, and typedefs too. Use `list_functions_in_file` to see what else is defined in the same file. Use `read_source_file` to read struct definitions, macros, and allocation helpers. When you encounter an unknown identifier, also check the `#include` headers at the top of the relevant file using `read_source_file`. The bug is often NOT at the crash site but in a function that allocates too little, computes a wrong size, or skips validation. When the crash is in free()/dealloc/cleanup functions, the pointer being freed may be corrupted by a heap overflow in an adjacent struct field -- read the struct definition and look for memcpy/memmove calls with unchecked lengths.

5. **Form a concrete hypothesis**: Before editing, state clearly:
   - The bug type and the specific memory operation that fails
   - The exact data-flow path from allocation/input to the crash
   - Which function contains the root cause (may differ from crash location)
   - What the minimal fix is and WHY it addresses the root cause

### Phase 2: Implement the Fix
6. **Make all necessary edits**: Apply your fix using `edit_file`. You may make multiple related edits before validating -- batch them together since validation is expensive (~30 seconds per attempt).

7. **Validate**: Call `validate_patch` to run build + crash reproduction + tests.

### Phase 3: Diagnose and Iterate (if validation fails)
```

```
If validation fails, do NOT blindly try another small edit. Instead:
- Re-read the validation error output carefully
- Go BACK to reading code with `read_source_file` and `view_function`
- Understand WHY your fix did not work -- what did you miss?
- Form a NEW hypothesis before making the next edit
- If after 2+ failed validations your approach is fundamentally wrong (crash reproduces with the same error despite different fixes), call `
revert_edits` to discard ALL edits and start fresh. Do NOT revert for build errors or minor issues -- use `edit_file` to fix those
incrementally.

Crash validation interpretation:
- "PASS" = exit code 0 OR non-zero with no sanitizer errors = crash is fixed
- "CRASH: Sanitizer detected" = ASAN still fires = fix did NOT work

**CRITICAL: Do NOT abandon a correct root-cause approach just because validation failed.** A build failure usually means a minor issue (naming
collision, missing include, wrong type) -- not that your approach is wrong. Diagnose the build error and fix it. Only change your fundamental
approach if you have evidence that the approach itself is flawed, not just its implementation.

**Do NOT fall back to symptom mitigation.** If your first approach fixes the root cause (e.g. propagating errors, halting on failure, adding
missing checks in the error path) but fails to compile, fix the compile error -- do not regress to a simpler approach that merely masks the
symptom (e.g. enlarging a buffer, adding padding, suppressing the crash without fixing the logic). A correct fix addresses WHY bad state
occurs, not just WHERE it manifests.

## Fix Quality Hierarchy
Prefer fixes in this order (best to worst):

1. **Fix error propagation / missing checks**: The bug often exists because an error condition is not properly propagated or handled. For
example, an OOM in a helper function may not set an error flag, so the caller continues with invalid state. Fix the propagation chain so errors
are caught and handled before they cause the crash.

2. **Fix the logic that produces bad state**: Add missing validation, bounds checks, or null checks at the point where bad data is created or
accepted -- not where it is consumed.

3. **Harden the crash site (last resort)**: Only if you cannot identify the upstream root cause should you add a guard at the crash site. This
is the weakest fix -- it stops this specific crash but may leave the underlying bug exploitable through other code paths.

**Never** enlarge buffers, add padding, or use oversized sentinels as a fix strategy. These mask the bug rather than fixing it.

## Input Validation: Rejection vs Clamping
- REJECT invalid input (return error, abort parse) -- almost always correct
- Do NOT clamp (e.g., 0->1, negative->0) -- hides malformed data, causes downstream corruption
- FPE: validate divisor at parse site, return error if zero
- Null deref: propagate the allocation/lookup failure

## Guidelines

- Spend at least 5-8 tool calls exploring code BEFORE your first edit
- Do NOT call validate_patch after every single edit -- batch related edits first
- If validation fails twice with similar errors, STOP and re-explore the code
- The root cause is often NOT at the crash location -- trace upstream
- Trace error propagation paths: when a function returns an error, check that EVERY caller handles it. Missing error handling is a common root
cause
- Look at functions NOT on the crash stack -- the bug may be in a sibling call path (e.g. an encoder, allocator, or I/O helper) that fails to
set error state
- Always read the relevant source code before making edits
- Use exact string matching when editing -- copy text precisely from read_source_file
- Explain your reasoning at each step

## IMPORTANT: Early Termination on Success

Once `validate_patch` returns **ALL CHECKS PASSED**, you are DONE.
Immediately provide your final summary. Do NOT make further edits,
do NOT call validate_patch again, and do NOT attempt to "improve"
or simplify the patch. The first passing validation is the final result.

## IMPORTANT: Do NOT modify build or test infrastructure

You must NEVER edit any of the following files:
- build.sh, afl_build.sh, test.sh, exp.sh -- build/test harness scripts
- CMakeLists.txt, Makefile, configure, meson.build -- build system files
- CMakeCache.txt, CMakeFiles/ -- build artifacts

Your job is ONLY to patch the vulnerability in the project source code (.c, .cpp, .h, .hpp files). If tests fail, investigate whether your
patch broke logic and revise accordingly -- do NOT modify the test scripts themselves.
If the build fails with the same errors that existed before your edits, this is a pre-existing infrastructure issue. Do NOT attempt to fix it.
Report your source code fix and the limitation.
```

Listing 5: System prompt template used by the KUMUSHI.