CRASH2POC: AGENTIC ROOT CAUSE ANALYSIS AND REPRODUCER SYNTHESIS FOR LINUX KERNEL BUGS

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language model (LLM)-driven agents have shown considerable promise in software engineering. However, their reliability remains limited in tasks that require jointly interpreting noisy execution reports and searching through large codebases. Reproducing Linux kernel vulnerabilities exemplifies this challenge: an agent must extract critical evidence from crash logs, locate the relevant code in the correct kernel revision, derive a well-grounded root-cause explanation, and generate an executable proof-of-concept (PoC) reproducer. We present **Crash2PoC**, a tool-augmented agentic framework that incorporates execution feedback to improve the reliability of kernel vulnerability reproduction. Crash2PoC decomposes the workflow into three stages: agent-driven crash localization, static-analysis-guided evidence enrichment, and agent-reasoning-supported root-cause PoC generation. We evaluate Crash2PoC on real-world Linux kernel vulnerabilities detected by sanitizers. Our results show that integrating static analysis, dynamic execution feedback, and agent-based reasoning improves the success rate of generating validated vulnerability reproducers.

1 INTRODUCTION

Large language models (LLMs) are increasingly used as code agents that navigate repositories, modify source files, and execute programs to solve software engineering tasks (Jimenez et al., 2024; Yang et al., 2024). As these agents move beyond isolated functions to large software systems, their reliability depends on understanding interactions among components, program state, and execution environments. However, producing plausible code does not establish its correctness: more comprehensive testing can expose defects in LLM-generated programs that pass the original tests (Liu et al., 2023). This gap makes executable evidence increasingly important for assessing both generated code and the reasoning used to produce it. Bug-reproducing tests offer such evidence by demonstrating a concrete failure and providing a basis for debugging and repair validation (Mündler et al., 2024; Cheng et al., 2025).

In this work, we study *crash-guided vulnerability reproduction* in the Linux kernel, a demanding instance of generating executable evidence for failures in large codebases. Given a sanitizer crash report, its accompanying execution log, and the corresponding kernel revision, the task is to synthesize a standalone C proof-of-concept (PoC) program that reproduces the reported failure in the matched environment. A successful reproducer must invoke the appropriate user-space interfaces, establish the required kernel state, and satisfy the conditions that expose the target vulnerability. Compilation alone, or a crash unrelated to the reported failure, is insufficient. The central problem is therefore to connect incomplete runtime observations to the program semantics and concrete actions needed for target-specific reproduction, without relying on an existing reproducer or a security patch as generation input.

Despite progress in agentic program synthesis, making this connection remains difficult. Studies of security agents show that generating working PoCs remains challenging, while recent kernel reproduction work identifies incorrect root-cause reasoning and insufficient source-code investigation as important failure modes (Lee et al., 2025; Pu et al., 2026a). A crash report records where an error becomes observable, but the operations that created the invalid state may lie in other functions or earlier executions. For example, reproducing a use-after-free can require establishing an object, releasing it through one path, and subsequently accessing it through another; inspecting only the final

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

dereference does not reveal the complete triggering sequence. An agent that overlooks these dependencies may produce a buildable program that never reaches the required state, or repeatedly revise inputs around an incorrect explanation. These limitations motivate explicit support for recovering and checking the program context on which PoC generation depends.

Our intuition is that reliable reproduction requires a chain of evidence connecting the observed crash, the relevant program dependencies, and the generated test. The crash report and revision-matched source can first establish checkable facts about where the failure manifests. Static analysis can then expose candidate calling and data-flow relationships around that location, while LLM reasoning assesses which candidates are compatible with the failure mechanism described by the report. The resulting context guides the generator toward the interfaces and ordering constraints needed to construct a reproducer. Finally, execution in the matched environment determines whether a candidate actually exposes the target failure. This division of responsibilities uses the agent to interpret and synthesize, while external checks constrain the evidence and validate the resulting program.

Turning this intuition into an effective workflow raises two challenges. **First, how can an agent establish trustworthy crash facts from noisy reports and a large, evolving codebase?** Reports contain multiple stack frames and memory-related values with different meanings; moreover, a source location is meaningful only in the revision that produced the crash. Incorrectly interpreting these observations can misdirect every subsequent generation step, even when the resulting explanation sounds plausible. **Second, how can the agent recover the trigger-relevant context needed to turn a crash location into an executable reproducer?** A location alone omits resource prerequisites and cross-function dependencies, whereas static analysis can return candidate paths that are irrelevant to the observed failure or infeasible under its execution conditions. The workflow must therefore combine precise evidence grounding with selective structural guidance, and assess their value through both end-to-end reproduction effectiveness (RQ1) and component ablations (RQ2).

Based on these observations, we present CRASH2POC, a tool-augmented agentic framework for crash-guided Linux kernel vulnerability reproduction. CRASH2POC addresses the two challenges through a three-stage workflow that combines evidence-grounded crash localization, static-analysis-guided trace recovery, and PoC synthesis. It first grounds the reported failure in verifiable crash facts and revision-matched source code, then recovers candidate traces and filters them according to their consistency with the observed failure mechanism. The synthesis stage uses this context to generate a standalone C PoC, which is compiled and executed in a matched virtual machine for validation. This workflow connects crash evidence to concrete triggering actions while separating the agent's reproduction hypothesis from execution-based validation.

Overall, this paper makes the following contributions:

- We design an evidence-grounded crash localization mechanism that associates extracted crash facts with supporting report lines and validates their structure, literal evidence, and source-location consistency. This mechanism provides a verifiable source anchor for subsequent analysis and generation.
- We develop a static-analysis-guided trace recovery mechanism that combines candidate trace enumeration with LLM-based semantic filtering. Integrated into CRASH2POC, this mechanism supplies trigger-relevant program context to guide the synthesis of executable kernel vulnerability reproducers.
- We evaluate CRASH2POC on real-world syzbot cases against a general-purpose code agent, and conduct ablation studies to assess the contributions of verified crash localization and static trace recovery to end-to-end reproduction effectiveness.

2 BACKGROUND

This section introduces bug-reproducing tests and vulnerability PoCs, reviews LLM-based reproduction methods, and explains the kernel-specific evidence and program context required by CRASH2POC.

2.1 Code Agents and Bug-Reproducing Tests

Repository-level software engineering requires reasoning across functions, files, and dependencies. SWE-bench captures this setting through real repository issues, while SWE-agent demonstrates how interfaces for code navigation, editing, and execution support agentic issue resolution (Jimenez et al., 2024; Yang et al., 2024). These capabilities also enable agents to generate tests, but a test that executes successfully or increases coverage need not expose a particular defect. A *bug-reproducing test* instead aims to demonstrate the behavior described in a bug report; when a valid fix is available, a common evaluation criterion is that the test fails on the buggy version and passes on the repaired version.

LIBRO uses LLM prompting to generate reproduction tests from bug reports and applies post-processing to filter and rank candidates (Kang et al., 2023). BRT Agent extends agentic reproduction to an industrial codebase and studies how generated tests support automated program repair (Cheng et al., 2025). SWT-Bench evaluates issue reproduction and the use of generated tests to validate candidate fixes (Mündler et al., 2024). Together, these studies establish reproduction as a useful bridge between natural-language reports and executable checks. Kernel vulnerability reproduction poses an additional requirement: the generated program must establish system state that causes a specific low-level failure, rather than merely encode an assertion about a library's return value.

2.2 LLM-Based PoC and PoV Generation

A vulnerability PoC provides an executable witness that a security-relevant defect can be triggered under specified conditions. Related work also uses the term *proof of vulnerability* (PoV). Here, a PoC denotes a program that reproduces the target kernel failure; it need not demonstrate privilege escalation or a complete exploit chain. The generation task requires both identifying a route to the vulnerable operation and satisfying the conditions under which that operation fails.

Recent systems address these requirements through different forms of program guidance. PoCGen combines LLM interpretation of npm vulnerability reports with static taint paths, usage context, runtime validation, and iterative refinement (Simsek et al., 2025). FaultLine decomposes reproduction into reasoning about data flow from an accessible API to a vulnerable sink, reasoning about intervening branch conditions, and generating and repairing a PoV test using feedback (Nitin et al., 2025). Its program-analysis-inspired reasoning does not require language-specific static or dynamic analysis components. DrillAgent grounds PoV generation in concrete execution states, translating execution observations into source-level constraints to refine candidate inputs (Li et al., 2026a). Together, these systems establish program structure and execution feedback as complementary sources of guidance for reproduction.

Evaluation also depends on the available evidence and the acceptance criterion. SEC-bench provides reproducible environments for evaluating PoC generation and vulnerability patching (Lee et al., 2025). DrillAgent explicitly distinguishes target-matching PoVs from candidates that trigger other sanitizer failures (Li et al., 2026a). This distinction is essential for crash-guided reproduction: a compiler error, an environment failure, or a different crash cannot establish that the reported vulnerability has been reproduced.

2.3 Kernel Vulnerability Reproduction

The Linux kernel exposes stateful interfaces whose behavior depends on earlier operations, object lifetimes, configuration, and interactions among subsystems. Consequently, a reproducer often needs a sequence of system calls with consistent resource dependencies before it can exercise the vulnerable operation. SemFuzz uses information extracted from vulnerability descriptions and Linux Git logs to guide semantics-based generation of kernel PoCs (You et al., 2017a). SyzDirect uses static analysis to identify relevant system calls and argument constraints, then guides fuzzing toward target kernel locations (Tan et al., 2023). KernelGPT uses LLMs to infer and refine syscall specifications for syzkaller, improving the interface descriptions available to kernel fuzzing (Yang et al., 2025a). These methods illustrate the importance of interface semantics and prerequisite state, although their outputs and supervision differ from generating a standalone reproducer directly from crash evidence.

More recent work applies LLM agents directly to kernel reproduction. K-Repro takes a security patch as input and equips an agent with source navigation, virtual-machine interaction, and debugging capabilities to construct and test PoCs (Pu et al., 2026a). SyzHarness also starts from a patch, but uses an LLM to construct a parameterized fuzzing harness: the harness supplies prerequisite setup while coverage-guided fuzzing searches uncertain, bug-critical values (Li et al., 2026b). A patch can identify changed code and provide information about the intended correction. Our setting instead starts from the crash report, execution log, and vulnerable revision, requiring the workflow to establish its source anchor and recover relevant program context from those artifacts.

2.4 Crash Evidence and Root-Cause Context

Syzbot continuously fuzzes Linux kernels and publishes artifacts for reported failures (The Syzkaller Authors, 2026). For memory-safety violations, KASAN reports can provide the access type, faulting address, stack traces, and information about the affected memory region (The Linux Kernel Developers, 2026). These observations have distinct meanings. The address of an invalid access is not interchangeable with the base of a displayed memory-state row, and neither directly identifies a static source variable. Likewise, the function performing the invalid access need not be the function that originally created the erroneous state. Interpreting a report therefore requires distinguishing the *crash site*, where the failure manifests, from the *root cause* and the preceding operations needed to trigger it.

Existing diagnosis systems recover complementary forms of this missing context. E&V prompts an LLM to execute and verify analysis pseudocode, applying backward taint reasoning to kernel crash triage (Hao et al., 2023a). KernelRCA reconstructs contextual causality chains using selective tracing and program analysis, exposing bug-contributing instructions together with their calling contexts and data dependencies (Gu et al., 2026a). KernelDiag connects heterogeneous crash artifacts and source-level reasoning through evidence graphs for faulty-method localization and explanation (Wang et al., 2026a). Their primary objective is diagnosis. For PoC synthesis, diagnostic context must additionally support concrete user-space operations that recreate the relevant state. Similarly, a statically recovered trace remains a candidate explanation: its presence does not establish path feasibility, correct object-lifetime ordering, or successful reproduction.

2.5 Problem Setting and Design Rationale

For a case i, let $X_i = (R_i, L_i, C_i)$ denote the crash report, execution log, and target kernel commit. Let E_i be the corresponding execution environment, including the kernel configuration and runtime artifacts. The objective is to generate a standalone C program P_i whose execution in E_i reproduces the failure described by R_i. In this setting, the report, log, and revision-matched source provide generation evidence; an existing reference reproducer or fix is not a generation input. Acceptance requires execution evidence consistent with the target failure, rather than agreement with the generator's own explanation. Matching a coarse crash category alone is insufficient to establish a shared root cause.

This formulation motivates two complementary forms of guidance. *Evidence grounding* checks that extracted crash facts are supported by the original artifacts and refer to valid locations in the target revision. *Structural guidance* exposes candidate interprocedural relationships that the agent can examine when recovering trigger conditions. Neither substitutes for executing the generated program. Crash2PoC connects these stages so that synthesis proceeds from checked crash facts and selected program context, and the resulting PoC is assessed in the matched environment. RQ1 evaluates the effectiveness of this complete workflow; RQ2 examines whether verified localization and static trace recovery provide complementary benefits.

3 Implementation

3.1 Overview

Our method comprises three stages and combines static analysis, dynamic analysis, and the semantic reasoning capabilities of an LLM-based agent. Given the i-th syzbot case, we represent the input as

$$X_i = (R_i, L_i, C_i)\,, \tag{1}$$

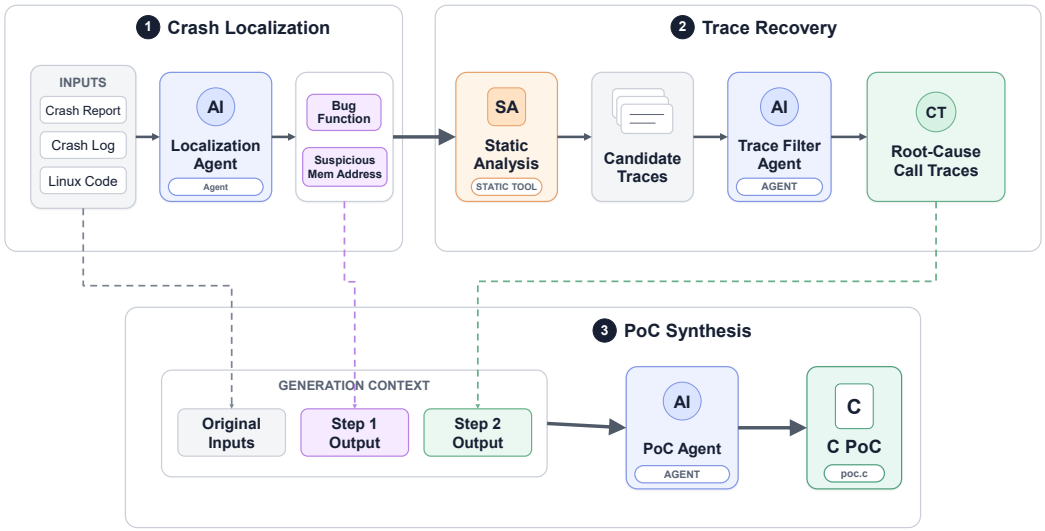


Figure 1: Overview of our three-stage pipeline for generating Linux kernel vulnerability proofs of concept.

where R_i is the crash report, L_i is the execution log, and C_i denotes the corresponding Linux kernel commit. The system then performs three stages: (1) evidence-grounded crash localization, (2) static reconstruction of candidate call traces and LLM-assisted filtering for traces that share the root cause of the original crash, and (3) agent-driven C proof-of-concept (PoC) synthesis followed by dynamic validation in the matched kernel environment.

The output of the complete pipeline is represented as

$$P_i = (F_i, Q_i, P_i, X_i)\,, \tag{2}$$

where F_i denotes the verified crash localization, Q_i denotes the static-analysis result when available, and P_i is the executable PoC. Our key design principle is to use the agent for semantic reasoning, source-code exploration, and program synthesis, while delegating acceptance decisions to deterministic parsers, schema constraints, source-level consistency checks, compilers, and virtual-machine execution. This hybrid design retains the flexibility of agentic exploration while using an explicit workflow to improve the stability, auditability, and reproducibility of the end-to-end process.

3.2 Input Normalization and Version Consistency

The pipeline takes the crash-report URL, execution-log URL, and the kernel commit associated with the bug as its primary inputs. For each case, the system fetches the exact target commit, verifies the checked-out `HEAD`, and materializes a temporary Linux worktree. All subsequent source-code retrieval, location validation, static analysis, and root-cause reasoning are performed against this revision. This revision-locked design prevents inconsistencies caused by kernel-version drift, including renamed functions, relocated source files, and changed line numbers. After processing a case, the expanded worktree is removed by default, whereas the Git object cache is retained. Later stages can therefore reconstruct the same revision without downloading it again.

3.3 Step 1: Evidence-Grounded Crash Localization

The objective of Step 1 is to transform an unstructured sanitizer report into three complementary and independently verifiable crash facts: the faulting access address (`fault_address`), the base address of the marked memory-state row (`memory_base`), and the static crash location (`crash_site`), which consists of a function name, a kernel-relative source path, and a line number.

Faulting access address. `fault_address` is the runtime virtual address that directly triggers the sanitizer report. In a KASAN report, it answers the question of which address was read or written by the invalid memory operation. This value can distinguish accesses to neighboring objects, different offsets within an object, or different executions of the same code path. However, a runtime address alone cannot be mapped directly to a static source location because it depends on the concrete execution and memory layout.

Memory-state row base. `memory_base` is the starting address of the memory-state row marked by the sanitizer. It is the display-row base of the hexadecimal memory dump rather than the exact address accessed by the faulty operation. For example, an invalid access to `0xffff888034bc4250` may be displayed within a row beginning at a lower, row-aligned address. Consequently, `memory_base` and `fault_address` have distinct semantics and are neither conflated nor required to have the same value.

Static crash site. `crash_site` associates the runtime failure with a static location in the target kernel revision. It contains three fields: `function`, which identifies the concrete subsystem function that performs the invalid access; `file`, which records the path relative to the Linux source root; and `line`, which records the source line reported in the crash stack trace. This location serves as the source-code anchor for subsequent static analysis and root-cause reasoning. The verifier checks that the target file exists in the specified kernel revision, that the function is defined in that file, and that the reported line lies within the function boundary.

Together, these fields describe (i) where the invalid access occurred in memory, (ii) how the surrounding memory state was represented by the sanitizer, and (iii) which static program location performed the access. They therefore establish an evidence chain from the runtime failure to the kernel source. For sanitizer reports that do not expose runtime memory addresses, such as some UBSAN reports, the two address fields and their paired evidence lines may be null; `crash_site` remains mandatory.

The system first applies a deterministic parser to obtain candidate addresses and stack frames. These candidates are treated only as untrusted hints and are never accepted directly. The agent then examines the raw report, execution log, and revision-matched kernel source to identify the subsystem function that performs the invalid access. In addition to the structured fields, the agent must return the complete raw evidence line supporting each extracted fact.

Each candidate output is evaluated by three deterministic checks:

1. **Structural validation.** The output must satisfy the predefined output schema, contain all required fields, and introduce no unsupported fields.
2. **Literal-evidence validation.** Every evidence line must occur verbatim in the original report or log. The addresses, function name, file path, and line number extracted from these lines must agree with the corresponding structured fields.
3. **Source-location validation.** The target file must exist in the specified kernel commit, the function must be defined in that file, and the reported line must lie within the function's source range.

If any check fails, the workflow collects the verifier errors and returns them to the agent for another localization attempt. The loop performs at most five attempts. Only an output that passes all three checks is accepted as F_i; otherwise, the case is recorded as producing no verified result. This mechanism converts agent-generated hypotheses into auditable and traceable program facts.

3.4 Step 2: Root-Cause Trace Recovery

Step 1 produces two types of evidence: the function in which the bug manifests and a set of suspicious memory addresses extracted from the crash. Step 2 uses this evidence to recover call traces that can reach the same underlying vulnerability. As shown in Figure 1, this step consists of taint-based trace enumeration followed by LLM-assisted filtering.

Taint-based trace enumeration. Let $\mathcal{A} = \{a_1, \ldots, a_m\}$ denote the suspicious memory addresses identified in Step 1, and let $\mathcal{K}$ denote the Linux kernel codebase. We configure our static analyzer

to treat every address in $\mathcal{A}$ as a taint source. The analyzer propagates the taint through the kernel codebase according to its interprocedural transfer rules and records the call paths along which the tainted memory can reach bug-relevant operations. We retain all discovered paths at this stage to favor recall over precision. Formally, the analyzer produces a candidate trace set

$$\mathcal{T}_{\text{cand}} = \bigcup_{a_i \in \mathcal{A}} \text{Trace}(\mathcal{K}, a_i). \tag{3}$$

Because multiple kernel paths may access the same memory location, this process often produces many candidate traces. Some traces only share a memory address with the original crash and do not correspond to the same root cause.

LLM-assisted trace filtering. We use an LLM to distinguish root-cause-relevant traces from spurious static analysis results. For each candidate trace, the LLM receives the original crash report and crash log, the bug function identified in Step 1, and the functions and call relationships contained in the trace. The filtering decision is based on root-cause consistency rather than function-name or address overlap alone. In particular, the LLM examines whether the trace reaches the vulnerable operation under a compatible execution context and whether the propagated memory is involved in the same failure mechanism described by the crash evidence.

The filter returns a subset

$$\mathcal{T}_{\text{root}} = \{\tau \in \mathcal{T}_{\text{cand}} \mid F_{\text{LLM}}(\tau, R, L, f^*) = 1\}, \tag{4}$$

where R and L denote the crash report and crash log, respectively, and f^* is the bug function localized in Step 1. The resulting set $\mathcal{T}_{\text{root}}$ contains call traces that are consistent with the root cause of the original crash and serves as the main structural guidance for PoC generation.

3.5 Step 3: PoC Synthesis

Step 3 synthesizes a C proof of concept by jointly using the original crash inputs and the intermediate results from the previous two steps. Specifically, the generation context contains the crash report, crash log, and Linux codebase, together with the localized bug function and suspicious memory addresses from Step 1 and the root-cause-consistent call traces from Step 2. We denote this context as

$$\mathcal{C} = (R, L, \mathcal{K}, f^*, \mathcal{A}, \mathcal{T}_{\text{root}}). \tag{5}$$

An LLM-based PoC agent consumes $\mathcal{C}$ and generates a standalone C program. The original crash report and log describe the expected failure, while the Step 1 output anchors generation to the vulnerable kernel location. The filtered traces from Step 2 provide the call-sequence structure needed to reach the same root cause. Together, these inputs reduce the need for the generator to rediscover the complete triggering path from the raw crash evidence alone.

During generation, the agent uses the retained traces to identify the operations and ordering constraints required to reach the vulnerable code. It then translates this information into concrete user-space actions and constructs the corresponding C program. The generation process can be summarized as

$$P = G_{\text{PoC}}(\mathcal{C}), \tag{6}$$

where P is the generated program. The final output of the pipeline is a standalone `poc.c` intended to trigger the same root cause as the original kernel crash.

4 Evaluation

4.1 Dataset Preparation

We construct our evaluation dataset from bug reports in the *fixed* archive of syzbot. Using fixed bugs reduces the possibility that the collected reports correspond to transient failures or unresolved infrastructure issues. Each dataset entry represents one syzbot bug record. We use the official C reproducer provided by syzbot as the ground-truth triggering program.

Table 1: PoC reproduction results. The methods were evaluated on different numbers of cases.

Approach	Evaluated	Successful	Success rate
MINI-SWE-AGENT	20	8	40.0%
Crash2Poc	337	264	78.3%

This work focuses on memory-safety violations reported by the Kernel Address Sanitizer (KASAN). Our collected snapshot contains 337 KASAN cases, covering **20 distinct KASAN crash categories**. We derive the category from the syzbot crash title by removing the crash-site suffix beginning with the standalone keyword `in`. For example, `KASAN: null-ptr-deref Write in sys_io_uring_register` is categorized as `KASAN: null-ptr-deref Write`. Read and write violations are treated as different categories, as are allocator-specific variants such as `slab-use-after-free` and `use-after-free`.

The kernel source tree is checked out at the exact commit recorded by syzbot, and the corresponding kernel configuration is retained to minimize differences between the original and reconstructed environments. The crash report provides the sanitizer diagnosis and stack trace, while the complete execution log preserves boot messages and events preceding the crash. The symbolized `vmlinux` image and matching source revision are used for source-level analysis and crash-site validation.

Consistency checks. Before including a case in the evaluation, we verify that its C reproducer, kernel revision, configuration, crash report, and execution log refer to the same syzbot record. We also check that the reported crash title is consistent with the assigned KASAN category. Cases are indexed by their syzbot identifiers to maintain traceability from the processed dataset back to the original reports.

4.2 Experimental setup.

For each case, we boot an isolated QEMU virtual machine using the case-specific kernel and disk images described in Section 4.1, and execute the generated C reproducer inside the guest. We use GLM-5.3 as the backbone model and compare our method with MINI-SWE-AGENT. To control for differences unrelated to the agent design, both methods use the same model, input artifacts, and per-case execution budget.

4.3 RQ1: How effective is Crash2Poc at generating vulnerability PoCs?

Motivation. Generating an executable program does not necessarily imply that it reproduces the intended vulnerability. RQ1 evaluates the end-to-end effectiveness of Crash2Poc in generating PoCs that trigger reported Linux kernel vulnerabilities. We also report the available results for MINI-SWE-AGENT, a general-purpose agent directly prompted to generate PoCs.

Evaluation Metrics. We use end-to-end PoC success rate as the primary metric, defined as the number of successfully reproduced vulnerabilities divided by the total number of evaluated vulnerabilities. A case is considered successful only if the generated PoC can be compiled and executed without manual modification and triggers the target vulnerability in the specified kernel environment. Compilation alone or an unrelated crash does not count as successful reproduction.

Results. Table 1 summarizes the available results. Overall, Crash2Poc successfully generates valid PoCs for 264 out of 337 vulnerabilities, achieving a success rate of 78.3%. The remaining 73 cases do not yield a successful reproduction. For MINI-SWE-AGENT, 8 out of 20 evaluated vulnerabilities are successfully reproduced, corresponding to a success rate of 40.0%. Because the two evaluations cover different numbers of vulnerabilities, these aggregate rates do not establish a controlled comparison on a common set of cases.

Answer to RQ1. Crash2Poc achieves an end-to-end PoC success rate of 78.3% on the 337 evaluated vulnerabilities. Quantifying its improvement over direct agent-based generation requires evaluating both methods on the same cases under matched experimental conditions.

Table 2: Ablation results for crash localization and static analysis. Success rate denotes the percentage of cases for which a verified PoC successfully triggers the target bug.

CONFIGURATION	SUCCESS RATE (%)
Crash2Poc	78.4
NoLocalization	45.2
NoStaticAnalysis	35.7

4.4 RQ2: Ablation Study

Motivation. Our system uses two analysis components before agent-driven root-cause reasoning and PoC synthesis. First, evidence-grounded crash localization extracts and verifies the concrete source location at which the invalid access occurs. Second, static analysis reconstructs the root-cause-relevant call chain to provide the agent with the program context connecting externally reachable entry points to the crash site. We conduct an ablation study to determine whether these components improve root-cause identification and end-to-end PoC generation, rather than merely adding preprocessing steps to the pipeline.

Evaluation Metrics. We compare the Crash2Poc pipeline with the following variants. All variants use GLM-5.3 as the base LLM model, prompts for the remaining stages, tool set, target kernel revision, and interaction-step budget.

- **Crash2Poc**: The complete pipeline uses the Localization Agent to identify the bug function and suspicious memory addresses, and the Static Analysis component to enumerate candidate traces. These traces are subsequently filtered to guide PoC synthesis.
- **NoLocalization**: This variant removes the Localization Agent, which identifies the bug function and suspicious memory addresses from the crash report, execution log, and kernel source code. This ablation evaluates the contribution of agent-based crash localization to end-to-end PoC generation.
- **NoStaticAnalysis**: This variant retains the Localization Agent but removes the Static Analysis component responsible for enumerating candidate traces. The subsequent reproduction workflow therefore operates without statically generated candidate traces. This ablation evaluates the contribution of static program analysis to end-to-end PoC generation.

Answer to RQ2. The complete pipeline achieves a PoC success rate of 78.3%, compared with 45.2% for NoLocalization and 35.7% for NoStaticAnalysis. Removing crash localization or static trace recovery reduces the success rate by 33.1 and 42.6 percentage points, respectively. These results suggest that both components contribute to end-to-end PoC generation effectiveness. The larger decrease associated with removing static analysis highlights the importance of providing the generator with trigger-relevant program context under the evaluated configuration.

5 Conclusion

In this paper, we presented Crash2Poc, a tool-augmented agentic framework for reproducing Linux kernel vulnerabilities from crash reports, execution logs, and revision-matched source code. Crash2Poc combines evidence-grounded crash localization with static-analysis-guided trace recovery to connect observed failures with the program context required for PoC synthesis. By validating extracted crash facts and filtering candidate traces according to the observed failure mechanism, the framework provides explicit guidance for generating standalone C reproducers. The generated programs are then compiled and executed in matched kernel environments to assess whether they reproduce the target failure. This design separates plausible agent-generated explanations from executable evidence, providing a structured approach to vulnerability reproduction in large codebases. Future work will investigate richer execution feedback for refining trigger conditions, particularly for failures involving complex object lifetimes and concurrency, and extend the framework to other large-scale software systems.

AI USE DISCLOSURE

We used Codex and Cursor as assistive tools during the research and writing process. Specifically, these tools were used to support research ideation and execution, including brainstorming and refining possible research directions, assisting with code implementation and debugging, and providing suggestions related to the experimental workflow. They were also used for information retrieval and discovery, such as identifying potentially relevant terminology, research directions, and related work. In addition, the tools assisted in drafting portions of the manuscript and improving the clarity, grammar, organization, and readability of the writing.

All outputs produced by these tools were treated as unverified suggestions rather than authoritative sources. The authors independently evaluated the AI-assisted research ideas and retained full control over all methodological and experimental decisions. AI-suggested literature and citations were checked manually against the original publications before inclusion; no AI-generated bibliographic information was relied upon without verification. AI-assisted code was reviewed, executed, debugged, and validated by the authors, and the reported experimental results were obtained and examined by the authors. All AI-assisted manuscript text was reviewed and revised to ensure that it accurately reflects the authors' intended meaning, methods, results, and conclusions.

Codex and Cursor were not treated as authors and were not used as substitutes for scholarly judgment or as sources of evidence. The authors take full responsibility for the originality, accuracy, integrity, and final content of this work, including all text, claims, citations, code, results, and other artifacts produced with the assistance of generative AI.

REFERENCES

Weiteng Chen, Xiaochen Zou, Guoren Li, and Zhiyun Qian. KOOBE: Towards facilitating exploit generation of kernel Out-Of-Bounds write vulnerabilities. In *29th USENIX Security Symposium (USENIX Security 20)*, pp. 1093–1110. USENIX Association, 2020. URL https://www.usenix.org/conference/usenixsecurity20/presentation/chen-weiteng.

Runxiang Cheng, Michele Tufano, Jürgen Cito, José Cambronero, Pat Rondon, Renyao Wei, Aaron Sun, and Satish Chandra. Agentic bug reproduction for effective automated program repair at Google. arXiv preprint arXiv:2502.01821, 2025. URL https://arxiv.org/abs/2502.01821.

Kangzheng Gu, Yifan Zhang, Yuan Zhang, and Min Yang. KernelRCA: Facilitating root cause analysis of memory corruptions in Linux kernel with contextual causality chain. In *35th USENIX Security Symposium*, pp. 791–810. USENIX Association, 2026a. URL https://www.usenix.org/conference/usenixsecurity26/presentation/gu-kangzheng.

Kangzheng Gu, Yifan Zhang, Yuan Zhang, and Min Yang. KernelRCA: Facilitating root cause analysis of memory corruptions in linux kernel with contextual causality chain. In *35th USENIX Security Symposium (USENIX Security 26)*, pp. 791–810. USENIX Association, 2026b. URL https://www.usenix.org/conference/usenixsecurity26/presentation/gu-kangzheng.

Yu Hao, Weiteng Chen, Ziqiao Zhou, and Weidong Cui. E&V: Prompting large language models to perform static analysis by pseudo-code execution and verification. arXiv preprint arXiv:2312.08477, 2023a. URL https://arxiv.org/abs/2312.08477.

Yu Hao, Weiteng Chen, Ziqiao Zhou, and Weidong Cui. E&v: Prompting large language models to perform static analysis by pseudo-code execution and verification. *arXiv preprint arXiv:2312.08477*, 2023b. URL https://arxiv.org/abs/2312.08477.

Junyoung Jang, Gwanhyun Lee, Hwiwon Lee, Kyuheon Kim, Jongseong Kim, Jinho Jung, and Lingming Zhang. Evaluating coding agents on kernel exploit generation. *arXiv preprint arXiv:2609.25591*, 2026. URL https://arxiv.org/abs/2609.25591.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2310.06770.

Sungmin Kang, Juyeon Yoon, and Shin Yoo. Large language models are few-shot testers: Exploring LLM-based general bug reproduction. In *IEEE/ACM International Conference on Software Engineering*, pp. 2312–2323, 2023. doi: 10.1109/ICSE48619.2023.00194. URL https://arxiv.org/abs/2209.11515.

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

Haoyu Li, Xijia Che, Yanhao Wang, Xiaojing Liao, and Luyi Xing. Execution-state-aware LLM reasoning for automated proof-of-vulnerability generation. arXiv preprint arXiv:2602.13574, 2026a. URL https://arxiv.org/abs/2602.13574.

Xingyu Li, Juefei Pu, Haonan Li, Arrdya Srivastav, Kareem Shehada, Srikanth V. Krishnamurthy, and Zhiyun Qian. SyzHarness: Patch-based kernel bug reproduction with LLM-synthesized fuzzing harnesses. arXiv preprint arXiv:2609.23889, 2026b. URL https://arxiv.org/abs/2609.23889.

Xingyu Li, Juefei Pu, Haonan Li, Arrdya Srivastav, Kareem Shehada, Srikanth V Krishnamurthy, and Zhiyun Qian. Syzharness: Patch-based kernel bug reproduction with llm-synthesized fuzzing harnesses. *arXiv preprint arXiv:2609.23889*, 2026c. URL https://arxiv.org/abs/2609.23889.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems*, 2023. URL https://arxiv.org/abs/2305.01210.

Niels Mündler, Mark Niklas Müller, Jingxuan He, and Martin Vechev. SWT-Bench: Testing and validating real-world bug-fixes with code agents. In *Advances in Neural Information Processing Systems*, 2024. URL https://arxiv.org/abs/2406.12952.

Vikram Nitin, Baishakhi Ray, and Roshanak Zilouchian Moghaddam. FaultLine: Automated proof-of-vulnerability generation using LLM agents. arXiv preprint arXiv:2507.15241, 2025. URL https://arxiv.org/abs/2507.15241.

Juefei Pu, Xingyu Li, Haonan Li, Zhengchuan Liang, Jonathan Cox, Yifan Wu, Kareem Shehada, Arrdya Srivastav, and Zhiyun Qian. Patch-to-PoC: A systematic study of agentic LLM systems for Linux kernel N-day reproduction. arXiv preprint arXiv:2602.07287, 2026a. URL https://arxiv.org/abs/2602.07287.

Juefei Pu, Xingyu Li, Zhengchuan Liang, Jonathan Cox, Yifan Wu, Kareem Shehada, Arrdya Srivastav, and Zhiyun Qian. Patch-to-poc: A systematic study of agentic llm systems for linux kernel n-day reproduction. *arXiv preprint arXiv:2602.07287*, 2026b. URL https://arxiv.org/abs/2602.07287.

Deniz Simsek, Aryaz Eghbali, and Michael Pradel. PoCGen: Generating proof-of-concept exploits for vulnerabilities in Npm packages. arXiv preprint arXiv:2506.04962, 2025. URL https://arxiv.org/abs/2506.04962. Version 4 revised in June 2026.

Xin Tan, Yuan Zhang, Jiadong Lu, Xin Xiong, Zhuang Liu, and Min Yang. SyzDirect: Directed greybox fuzzing for Linux kernel. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pp. 1630–1644, 2023. URL https://security.fudan.edu.cn/publications/content/2023-09-11-003/.

The Linux Kernel Developers. Kernel address sanitizer (KASAN). The Linux Kernel documentation, 2026. URL https://docs.kernel.org/dev-tools/kasan.html. Accessed September 26, 2026.

The Syzkaller Authors. Syzbot documentation. Syzkaller project documentation, 2026. URL https://github.com/google/syzkaller/blob/master/docs/syzbot.md. Accessed September 26, 2026.

Weijing Wang, Zan Wang, Dong Wang, Haichi Wang, and Junjie Chen. KernelDiag: Agent-based root cause diagnosis for kernel crashes. arXiv preprint arXiv:2607.17722, 2026a. URL https://arxiv.org/abs/2607.17722.

Weijing Wang, Zan Wang, Dong Wang, Haichi Wang, and Junjie Chen. KernelDiag: Agent-based root cause diagnosis for kernel crashes. *arXiv preprint arXiv:2607.17722*, 2026b. URL https://arxiv.org/abs/2607.17722.

Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. FUZE: Towards facilitating exploit generation for kernel Use-After-Free vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)*, pp. 781–797. USENIX Association, 2018. URL https://www.usenix.org/conference/usenixsecurity18/presentation/wu-wei.

Chenyuan Yang, Zijie Zhao, and Lingming Zhang. KernelGPT: Enhanced kernel fuzzing via large language models. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 560–573, 2025a. doi: 10.1145/3676641.3716022. URL https://doi.org/10.1145/3676641.3716022.

Chenyuan Yang, Zijie Zhao, and Lingming Zhang. KernelGPT: Enhanced kernel fuzzing via large language models. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 560–573. Association for Computing Machinery, 2025b. URL https://doi.org/10.1145/3676641.3716022.

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

Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. SemFuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pp. 2139–2154, 2017a. doi: 10.1145/3133956.3134085. URL https://doi.org/10.1145/3133956.3134085.

Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. SemFuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pp. 2139–2154. Association for Computing Machinery, 2017b. URL https://doi.org/10.1145/3133956.3134085.

A Related Work

A.1 Automated Kernel Vulnerability Reproduction

Automated reproduction of kernel vulnerabilities has traditionally relied on fuzzing, symbolic execution, and vulnerability-specific program analysis. SemFuzz extracts security-relevant information from CVE descriptions and Linux Git logs and uses the extracted semantics to guide the generation and mutation of kernel test cases (You et al., 2017b). Other systems focus on evaluating or amplifying the exploitability of particular classes of memory corruption, such as use-after-free vulnerabilities (Wu et al., 2018) and out-of-bounds write vulnerabilities (Chen et al., 2020). These approaches demonstrate that kernel PoC construction benefits from vulnerability semantics and multi-system-call reasoning, but they either depend on publicly available vulnerability descriptions or specialize in particular vulnerability and exploitability classes.

More recently, K-Repro and SyzHarness employ LLM agents, together with source-code exploration and runtime feedback, to generate PoCs for vulnerabilities with available security patches (Pu et al., 2026b; Li et al., 2026c). Both systems rely on an existing fix patch to describe the target vulnerability. Unlike patch-guided approaches, our method recovers trigger-relevant call traces directly from the crash report, execution log, and vulnerable kernel source revision, and uses them to guide PoC synthesis.

A.2 Kernel Crash Diagnosis and Root-Cause Analysis

A kernel crash report usually identifies where a failure manifests, but the reported access or top stack frame need not identify the operation that created the faulty state. Propagation-oriented approaches address this gap by reconstructing how an erroneous state is created and propagated to the observed failure. E&V and KernelRCA follow this direction using complementary techniques: E&V prompts an LLM to perform verified backward-taint reasoning, whereas KernelRCA uses selective tracing and program analysis to recover calling contexts and data dependencies (Hao et al., 2023b; Gu et al., 2026b). Another line of work integrates heterogeneous diagnostic evidence. KernelDiag employs specialized agents to connect crash reports, execution logs, system calls, kernel configurations, and source code, organizing the resulting dependencies into evidence graphs for root-cause localization and explanation (Wang et al., 2026b). These approaches primarily produce a blamed program location or a causal explanation. Our pipeline instead uses crash diagnosis as an intermediate step toward synthesizing an executable program that reaches and triggers the same root cause.

A.3 LLM-Assisted Kernel Analysis and Security Agents

Prior work has applied LLMs to several kernel-security tasks, including crash analysis, system-call specification generation, vulnerability reproduction, and agent-based exploit generation. Despite their different objectives, these studies share a common design pattern: LLM reasoning is coupled with external grounding and validation. Static-analysis and specification-generation approaches constrain LLM outputs through execution rules, structured evidence, or validation feedback (Hao et al., 2023b; Wang et al., 2026b; Yang et al., 2025b), whereas vulnerability reproduction and agent-evaluation settings rely on compilation, virtual-machine execution, coverage-guided search, or deterministic runtime verifiers (Pu et al., 2026b; Li et al., 2026c; Jang et al., 2026).

Our system follows this hybrid design while operating without a security patch. The agent is responsible for source-code exploration, semantic trace filtering, and PoC synthesis; structured validators ground crash localization in literal report evidence and revision-matched source code. Compilation checks whether a generated program is buildable, while execution in the matched virtual machine determines whether it reproduces the target failure.

B BASELINE COMPARISON RESULT

Table 3: Detailed one-round reproduction results for the 20 KASAN categories. The common `KASAN:` prefix is omitted.

CATEGORY	REPRO.	NOT REPRO.	UNABLE TO EXECUTE
`slab-use-after-free Read`	–	✓	–
`slab-out-of-bounds Read`	–	✓	–
`use-after-free Read`	–	✓	–
`slab-use-after-free Write`	–	✓	–
`slab-out-of-bounds Write`	✓	–	–
`null-ptr-deref Read`	–	✓	–
`stack-out-of-bounds Read`	–	✓	–
`null-ptr-deref Write`	✓	–	–
`global-out-of-bounds Read`	✓	–	–
`use-after-free Write`	–	✓	–
`vmalloc-out-of-bounds Read`	–	✓	–
`out-of-bounds Read`	✓	–	–
`vmalloc-out-of-bounds Write`	✓	–	–
`invalid-free`	✓	–	–
`wild-memory-access Read`	–	✓	–
`invalid-access Read`	✓	–	–
`invalid-access Write`	✓	–	–
`stack-out-of-bounds Write`	–	–	✓
`user-memory-access Write`	–	✓	–
`wild-memory-access Write`	–	–	✓