

KPocBench: Benchmarking and Measuring Code Reasoning of Language Models in Linux Kernel

Ziyi Guo¹, Kexin Pei², Yan Chen¹

¹Northwestern University, ²University of Chicago

Correspondence: n7l8m4@u.northwestern.edu

Abstract

Large Language Model (LLM) agents increasingly act by generating, executing, and revising code, making code reasoning essential for understanding program state, diagnosing failures, and repairing behavior. Existing evaluations often measure end-to-end software engineering outcomes, obscuring which reasoning capabilities limit agents in complex, execution-grounded systems. We study this problem through Proof-of-Concept (PoC) generation, where an agent synthesizes an executable trigger that reproduces a reported crash or sanitizer failure. We instantiate this task in the Linux kernel, whose scale, statefulness, pointer-heavy value flow, memory layout, and concurrency stress long-horizon code reasoning. We decompose kernel PoC generation into four capabilities: state-machine reasoning, dynamic memory layout reasoning, value-flow reasoning, and concurrency reasoning. We build KPocInfra to emit branch-level execution telemetry and curate KPocBench, a benchmark of 181 reproducible Linux kernel KASAN cases from Syzbot. On KPocBench, state-machine and value-flow reasoning explain most failures, while capability-scaffolded evidence improves the best success rate from 50% to 61% and moves 29 unsolved attempts deeper along vulnerable execution paths. These results position kernel PoC generation as a practical stress test for LLM code reasoning in large-scale systems software.

1 Introduction

Code reasoning has emerged as a key capability of modern large language models (LLMs). Beyond its immediate relevance to software engineering, code reasoning is also becoming a prerequisite for agentic AI: code increasingly represents the action space through which agents invoke tools, manipulate environments, plan multi-step behavior, and repair their own failures (Ning et al., 2026).

An agent that cannot reason about the code it executes or generates is limited in its ability to diagnose unexpected outcomes, revise plans, and safely interact with complex software systems. In practice, automated code reasoning also provides a foundation for building trustworthy software (Liu et al., 2024b), supporting broad software reliability and security applications, including automated program repair (Liu et al., 2023; Van Tonder and Le Goues, 2018; Jain et al., 2023; Nong et al., 2024; Li et al., 2025, 2026), fuzz testing (Zhu et al., 2025a; Xia et al., 2024), vulnerability detection (Li et al., 2024; Zibaeirad and Vieira, 2024), and formal verification (Yang et al., 2026, 2025a,b).

Evaluating this code reasoning capability, however, requires more than measuring whether an LLM agent completes an end-to-end software development task, e.g., whether an issue has been fixed (Jimenez et al., 2024; Hariharan et al., 2025). To better probe how models reason about code, a growing body of work has proposed benchmarks that target code reasoning capabilities more directly (Gu et al., 2024; Xie et al., 2025; Liu et al., 2024a; Chen et al., 2025; La Malfa et al., 2025; Beger and Dutta, 2025; Wei et al., 2025). However, many of these benchmarks still exercise only local code behavior or simplified analysis tasks. For example, small-scale static-analysis settings can test whether a model recognizes a particular vulnerability pattern, but they often abstract away the scale, execution environment, and cross-component interactions that dominate real system-scale workloads. Other recent benchmarks raise the difficulty by evaluating code reasoning capabilities on real-world repo-level tasks (Roy et al., 2025; Kopel; Liu et al., 2025). Among these tasks, Proof-of-Concept (PoC) generation is particularly appealing: given a known vulnerability or bug report, the agent must synthesize a small executable trigger, such as a user input to the target system, that reproduces the reported crash or sanitizer fail-

ure. This task requires substantially longer-horizon planning, codebase exploration, trigger construction, and iterative use of execution feedback (Zhao et al., 2025; Wang et al., 2025; Zhu et al., 2025b; Fang et al., 2024; Heelan, 2025). Unfortunately, existing PoC-generation evaluations focus primarily on user-space programs and web applications. In these settings, the input-to-crash chain is often shorter, the execution environment is more observable, and the trigger can often be constructed without deeply exercising system state, dynamic memory layout, indirect value flows, or concurrency. As a result, they provide limited visibility into the reasoning demands that become critical in large-scale systems software, where long-horizon state tracking, sparse execution feedback, pointer-heavy value propagation, and concurrent interleavings can combinatorially expand the space of possible executions.

Our approach. In this paper, we instantiate PoC generation in the Linux kernel and use *PoC generation from kernel bug reports* as a stress test for long-horizon code reasoning in large-scale systems software. In the Linux kernel, such a PoC is typically a user-space program that issues a sequence of system calls (syscalls) to drive the kernel into the vulnerable state. This combination is important: PoC generation supplies the iterative reasoning task, while the Linux kernel supplies the scale, statefulness, and limited observability that make the task difficult. Starting from a bug report, an agent must interpret the reported failure, localize the relevant code paths, infer the triggering conditions, synthesize an executable program, and refine failed attempts using execution feedback. At the same time, success is unambiguously verifiable by reproducing the reported crash or sanitizer report.

We target the Linux kernel because it amplifies reasoning requirements that are less explicit in the user-space and web-application settings studied in prior PoC work. First, its sheer *scale and complexity*: roughly 40 million lines of C across hundreds of subsystems, far exceeding any current model’s context window and forcing the agent to retrieve and reason about code on demand. Second, its *statefulness*: triggering a kernel bug typically requires a precise, ordered syscall sequence that drives the subsystem through multiple intermediate states before the final vulnerable operation can take effect. Third, its *limited observability*: a bug report usually records only a post-mortem stack trace, and

a failed PoC often produces little or no diagnostic output, leaving the agent uncertain about where execution diverged from the intended path. Finally, kernel vulnerabilities often require reasoning about dynamic memory layout, indirect value flows, and concurrency, which quickly expands the space of possible executions.

Our key insight is that the combination of the task and the target system is what makes the setting challenging. PoC generation on simpler software can often avoid deep reasoning about state, memory layout, value flow, or concurrency, while kernel-focused tasks without PoC generation do not exercise the same long-horizon planning and self-correction loop. Kernel PoC generation thus provides a setting in which code reasoning failures are practically meaningful, experimentally observable, and security-impactful.

Technical design. As these challenges are often entangled, a single aggregate success rate cannot reveal which reasoning capability is limiting current agents. For example, a kernel use-after-free may require the agent to drive the subsystem into the right state, shape allocator behavior, propagate the correct value through pointer aliases, and win a race, but a failed PoC gives no indication which step broke. To make such failures observable, we decompose kernel PoC generation into four reasoning capabilities: *state-machine reasoning*, *dynamic memory layout reasoning*, *value-flow reasoning*, and *concurrency reasoning*, each grounded in concrete kernel-level failure modes (§2). We then develop KPocInfra, which executes each candidate PoC in QEMU on a kernel instrumented at vulnerability-relevant branches. After each iteration, KPocInfra emits structured execution telemetry, including reached branches, guard outcomes, and runtime values of selected expressions. By selectively exposing this telemetry, we can intervene in the agent’s reasoning loop and disentangle how specific reasoning capabilities contribute to task success or failure, using branch-granularity evidence rather than a binary pass/fail outcome. We open source the Dataset and related kPocInfra in <https://github.com/OrangeRoland/kPocInfra>.

Results. Building on this infrastructure, we curate KPocBench, a benchmark of 181 reproducible Linux kernel cases reported by Kernel Address Sanitizer (KASAN) and drawn from Syzbot (Google, 2017), spanning ten bug types

and including 57 cases (31%) that require race conditions to trigger. We evaluate two agent harnesses with three model configurations: Claude Code with Claude Opus-4.6 and DeepSeek V4-Pro, and OpenAI Codex with GPT-5.4. Before feeding telemetry back to the agent, we first use KPocInfra as an observation tool to monitor where each candidate PoC stops making progress. This analysis reveals a stable ordering of capability bottlenecks: state-machine reasoning is dominant, accounting for 42–59% of failures, while value-flow reasoning is the second consistent barrier, accounting for 31–37%. Concurrency and memory-layout reasoning account for smaller, model-specific shares. This ordering is stable across model families and agent harnesses, suggesting that it reflects the structure of kernel PoC generation rather than the weakness of a single agent.

We then use telemetry-guided feedback as a controlled intervention that approximates stronger reasoning capabilities by externalizing branch-level runtime evidence the agent would otherwise need to infer. This intervention consistently improves performance: the best configuration solves 61% of cases, compared with 50% under raw-log feedback, and reduces the number of iterations in 52% of commonly solved cases. Even when cases remain unsolved, telemetry-guided feedback moves the generated PoC to a strictly deeper divergence point in 29 cases, suggesting that stronger reasoning capabilities, approximated by externalized runtime evidence, help agents advance further along the vulnerable execution path. Together, the observation and intervention results show that the same capabilities that explain failure also predict where improving code reasoning can advance kernel PoC generation.

2 Reasoning Capabilities for Kernel PoC Generation

We identify four reasoning capabilities that together span the cognitive demands of Linux kernel PoC generation. For each, we give a working definition, the sub-dimensions we measure, and how its failure manifests in execution traces. This taxonomy organizes both the design of KPocInfra (§2.5), which determines what evidence sources observe which capability, and the failure attribution analysis in §3.

2.1 State-Machine Reasoning

A vulnerable kernel path is rarely reachable from a single syscall. Typically, the agent must drive the relevant subsystem through an *ordered sequence of state transitions*, each established by a syscall whose effects modify internal kernel objects, data structures, and control flags. Reaching the vulnerable state requires every transition to complete and leave the kernel in exactly the configuration the next transition expects. A single missing or misordered step leaves the kernel in an unreachable configuration, even though the final triggering call itself may be correct. We decompose this capability into two sub-dimensions.

Pre-conditions. Identifying and establishing the initial conditions that must hold before any vulnerable operation can begin. This includes configuring objects in the right modes, selecting the right flags, and constructing inputs such as filesystem images or network packets that satisfy kernel-internal validation. Failure produces PoCs that trigger early-stage errors or are silently misdirected to a different code path.

State-transfer. Driving the kernel through the *complete* intermediate sequence connecting initial setup to the final vulnerable operation. An agent that knows the start and end states but not the transitions between them produces incomplete or misordered syscall sequences that stall in an intermediate, unreachable configuration.

2.2 Dynamic Memory Layout Reasoning

A large fraction of KASAN-detected bugs ([Linux Kernel Documentation, 2026a](#)) depend on specific relationships among dynamically allocated kernel objects. Unlike user-space memory bugs, these arise inside a highly structured allocator composed of slab caches, per-CPU freelists, and size-class pools whose behavior is invisible at the syscall interface. The agent must reason along two axes.

Spatial axis. Sizes, alignments, and relative positions of objects in memory. Out-of-bounds bugs are typically gated on a specific relationship between an allocated object’s size and an offset derived from user input. The agent must identify the slab cache holding the object, its allocation granularity, and the input value that produces the required overflow distance, often passing through allocator size rounding and kernel validation.

Temporal axis. Object lifecycles, including allocation, free, and reuse. Use-after-free triggering requires that, at the moment of the second access, the freed memory has been reclaimed and repopulated with an object of the right type. Reliable triggering frequently demands active *heap shaping* (Chen and Xing, 2019; Wu et al., 2018): draining freelists, controlling recycle order, and placing replacement objects adjacent to or coincident with the freed region. None of this appears in the bug report; it must be inferred from the SLUB allocator’s reclaim policy and the object types involved.

2.3 Value-Flow Reasoning

A user-controlled value at the syscall interface must propagate through many layers of kernel code. It is copied via `copy_from_user`, stored into an inode or socket buffer, retrieved through a VFS or networking abstraction, and finally tested at a guard that decides whether the vulnerable branch is taken. Value-flow reasoning is the agent’s ability to perform a partial symbolic evaluation of this chain across function boundaries to predict which input values steer execution toward the vulnerable path. In kernel code the chain routinely spans ten or more functions across multiple files, a setting where static dependency benchmarks also report sharp degradation on multi-step trace generation and long dependency chains (Xie et al., 2025).

Failure manifests insidiously: the agent invokes the *correct* syscall on the *correct* subsystem with parameter values that happen to take a different control-flow branch, with no error output to guide correction. Value-flow reasoning also subsumes *value-range analysis*: understanding not just what value a variable holds but what range of values satisfies a predicate. This is critical for boundary-condition bugs such as integer overflows and allocation limits.

2.4 Concurrency Reasoning

A substantial fraction of kernel vulnerabilities are race conditions that manifest only under specific thread interleavings. We evaluate concurrency reasoning along two sub-dimensions.

Concurrency identification. Determining whether a bug requires a concurrent setup at all. Bug reports rarely state this explicitly; the agent must infer it from indirect signals such as multiple threads in the crash stack, the KASAN report type, or the structure of the involved code paths.

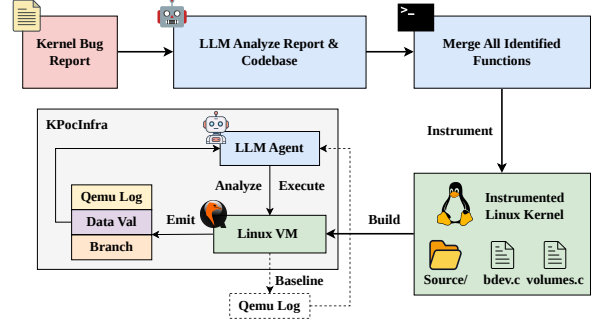


Figure 1: Overall workflow of KPocInfra.

Failures take two forms. The first is a sequential PoC for a race-required bug, which reaches the right code but never opens the race window. The second is unnecessary threading for a sequential bug, which introduces synchronization overhead and spurious failures.

Race-window targeting. Given correct identification, actually constructing a PoC that hits the window. Race windows are typically narrowed by spinlocks, RCU grace periods, refcounts, or atomic operations to a handful of instructions (Lee et al., 2021), so spawning two threads is rarely sufficient. An effective concurrent PoC must orchestrate the precise interleaving via synchronization primitives, memory pressure, or careful pre-race state setup. The required effort is asymmetric across bug types: refcount-underflow UAFs offer wide windows, whereas unlocked concurrent modification of complex data structures may have windows as narrow as a single store. Agents that apply generic threading patterns without adapting to the specific race structure fail at this dimension.

2.5 Capability-Attributable Infrastructure: KPocInfra

Figure 1 illustrates the overall workflow.

Overview. To evaluate LLM capabilities across the four reasoning dimensions described above, we build KPocInfra, a benchmark and evaluation infrastructure that *operationalizes* these capabilities by compiling them into the execution environment as structured runtime signals. Rather than expecting the agent to recover state-machine, memory-layout, value-flow, and concurrency evidence purely from static source inspection, KPocInfra instruments vulnerability-relevant kernel branches, drives iterative PoC generation inside a QEMU virtual machine, and returns per-iteration execution

evidence that externalizes the exact signals each capability would otherwise have to infer.

Our evaluation of each reasoning dimension is grounded in three complementary sources of evidence. First, the *branch instrumentation traces* collected during PoC execution record, for each instrumented branch, how many times it was reached, whether its condition was satisfied, and the concrete runtime value of the condition expression. These traces are produced at every iteration, forming a per-session record of how the generated PoC’s kernel coverage evolved over time. Second, the *LLM agent reasoning logs* capture the full sequence of hypotheses, code revisions, and diagnostic interpretations the agent produced across all iterations, providing direct evidence of how the agent responded to the scaffolded signals and whether its reasoning remained coherent and adaptive over the session. Third, the *original PoC and its corresponding instrumentation trace* serve as a reference baseline: by comparing the branch coverage and expression values produced by the original PoC against those produced by the generated PoC, we can precisely characterize how far the generated PoC diverged from the ground-truth execution path and at which specific kernel location the divergence first occurred.

Together, these three sources enable assessment of each reasoning dimension without relying on a single signal. State-machine reasoning failures are identified when coverage stalls at a subsystem boundary across many iterations, confirmed by reading the instrumented kernel source to establish what pre-condition or state transition was not satisfied. Value-flow failures are localized to specific branch points where the generated trace records a different expression value than the original, revealing which user-controlled input was incorrectly propagated through the kernel. Dynamic memory layout failures are evidenced by size or pointer values at allocation-related branches that diverge from the original trace, indicating that the agent did not reason correctly about the kernel’s object layout requirements. Concurrency reasoning failures are assessed by cross-referencing the Syzkaller reproducer metadata as a ground-truth concurrency label, the agent’s generated threading model, and the presence or absence of race-specific branch activations in the instrumentation traces.

3 Evaluations and Experiment

3.1 Dataset Construction

We construct our evaluation dataset from real-world Linux kernel vulnerability cases reported through Syzbot (Google, 2017, 2015), focusing exclusively on cases detected by the Kernel Address Sanitizer (KASAN) (Linux Kernel Documentation, 2026a). Starting from the full set of publicly available Syzbot KASAN reports, we filter for cases that include a fuzzer-generated C reproducing script, which serves as the ground-truth PoC for validation purposes.

Because Syzbot cases span a wide range of kernel versions and toolchain configurations, directly reusing the original reproducing scripts in a unified environment is not always possible. To ensure consistency, we reproduce each candidate case by compiling the corresponding kernel version, executing the fuzzer-generated C reproducing script, and verifying that the execution successfully triggers the same bug type and crash location as described in the original Syzbot report. Cases that fail to reproduce under our environment due to kernel version incompatibilities, missing configuration options, or toolchain differences are excluded from the final dataset. After this filtering and validation process, we retain a total of 181 reproducible cases.

Bug type distribution. Table 3 (Appendix) summarizes the distribution of vulnerability types across the 181 cases. Use-after-free vulnerabilities constitute the largest category (60 cases, 33.15%), followed by slab-out-of-bounds (49 cases, 27.07%), slab-use-after-free (26 cases, 14.36%), and null-pointer-dereference (25 cases, 13.81%). The remaining categories, including vmalloc-out-of-bounds, double-free, global-out-of-bounds, stack-out-of-bounds, out-of-bounds, and user-memory-access, collectively account for 21 cases (11.60%).

Race condition distribution. Among the 181 cases, 57 (31%) require a race condition to trigger the vulnerability, while the remaining 124 (69%) are sequential bugs that do not involve concurrency. Table 4 (Appendix) shows the breakdown of race-condition cases by bug type. Use-after-free vulnerabilities dominate the race-required subset with 29 cases, followed by slab-use-after-free (13 cases) and slab-out-of-bounds and vmalloc-out-of-bounds (5 cases each). This distribution reflects the fact that many use-after-free bugs in the kernel arise

from unsynchronized concurrent access to shared objects, making concurrency reasoning a critical capability for a substantial portion of the dataset.

3.2 Experimental Setup

Agent configuration. For each of the 181 cases, we launch an LLM agent instance with access to two sources of information: (1) the original Syzbot bug report, which includes the KASAN error description, stack trace, and crash metadata; and (2) the kernel source code corresponding to the kernel version in which the bug was reported. The agent is not provided with the ground-truth reproducing script or any hints derived from it.

The agent operates in an iterative loop. In each iteration, the agent first reads and analyzes the bug report to understand the vulnerability type, the affected subsystem, and the crash-site call chain. It then navigates the kernel source code to inspect relevant functions, data structures, and control-flow paths. Based on this analysis, the agent writes a C-language PoC program and submits it for execution.

Execution environment. All PoC executions are performed on a bare metal machine Intel(R) Xeon(R) Platinum 8358 CPU 2.6GHz; 100 G RAM, QEMU version 6.2.0 (Bellard and The QEMU Project, 2003) virtual machine running the corresponding instrumented kernel build. Each Qemu memory is 2G and SMP is 2. We use a unified QEMU configuration across all cases to ensure that differences in PoC success or failure are attributable to the generated code rather than to environmental variation. The VM is reset to a clean snapshot before each PoC execution to eliminate state contamination across iterations.

Feedback modes. We evaluate two feedback configurations to understand the impact of runtime observability on PoC generation quality.

In *baseline mode*, the agent receives only the raw QEMU console log after each PoC execution. This log contains kernel messages, KASAN reports (if triggered), and any error output produced by the PoC program. The agent must infer the cause of failure from these unstructured signals alone.

In *instrumented mode*, the agent receives the QEMU console log together with the structured instrumentation feedback produced by KPocInfra. This feedback includes the triggering status of each instrumented branch (whether it was reached and whether its condition was satisfied), the concrete

runtime values of up to three recorded meta-data expressions per branch, and the mapping from each instrument to its corresponding source-code location. This additional information allows the agent to precisely identify which branches along the vulnerability path were reached, where execution diverged from the expected path, and what runtime values were observed at each decision point.

Under both modes, the agent is given the same iteration budget and time limit per case. If the agent successfully triggers the target bug (as determined by matching the KASAN report type and crash location against the original Syzbot report), the case is marked as solved and the loop terminates. If the budget is exhausted without triggering the bug, the case is marked as failed.

3.3 Results

Treating instrumented feedback as a controlled probe over the four capabilities, we find a robust ordering of reasoning bottlenecks that persists across all three agents and both feedback modes (Table 2): state-machine reasoning dominates failures (42–59%), followed by value-flow reasoning (31–37%), with concurrency (10–16%) and memory-layout (0–9%) as smaller, model-specific gaps. Likelihood ratio test, chi square=0.08, df=2, p=0.96. The capability-attributed gains corroborate this ordering: scaffolding layout evidence yields the largest per-bug-type improvements on memory-layout-sensitive categories (Figure 2), and on 29 still-unsolved cases the agent nonetheless reaches a strictly deeper divergence point.

This is notable that kPocInfra is an experimental try, instead of direct and deterministic "causally gates". To be precise about what we do provide, the telemetry channels in KPocInfra are a capability aligned observability scaffold. They approximate the measurement of each reasoning capability by externalizing the runtime evidence an agent would otherwise have to infer, and we use them as a controlled intervention rather than as an independent causal signal.

We acknowledge there can be better ways or implementation. The telemetry we currently implement, branch reachability and expression values, is only one instantiation of such a scaffold and does not exhaust the observable signals. Other channels that we have not implemented, for example allocator level or scheduling level telemetry, could enrich the measurement of memory layout and concurrency reasoning and would allow finer

Table 1: Overall PoC generation success rates by agent and feedback mode.

Agent	Mode	Success	Total	Rate
Claude Opus-4.6	Baseline	92	181	50%
Claude Opus-4.6	Instrumented	111	181	61%
OpenAI GPT-5.4	Baseline	67	181	37%
OpenAI GPT-5.4	Instrumented	88	181	49%
Deepseek V4-Pro [1M]	Baseline	85	181	47%
Deepseek V4-Pro [1M]	Instrumented	108	181	59%

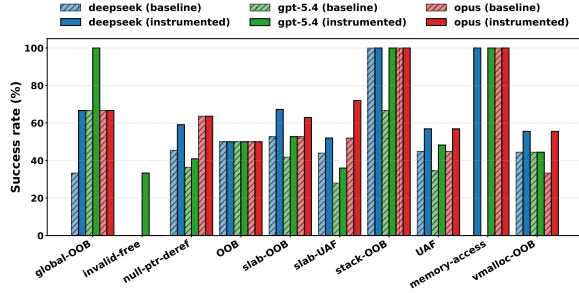


Figure 2: Per-bug-type PoC generation success rates for Deepseek-V4-Pro, OpenAI GPT-5.4, and Claude Opus-4.6 under baseline (raw QEMU log) and instrumented (KPocInfra feedback) modes. Categories with small sample sizes (out-of-bounds, user-memory-access, stack-out-of-bounds, global-out-of-bounds; see Table 3) exhibit high variance and should be interpreted with caution.

attribution.

Overall generation success rates. Table 1 presents the overall PoC generation success rates across three LLM agents (Claude Opus-4.6 (Anthropic, 2025), OpenAI GPT-5.4 (OpenAI, 2025), Deepseek V4-Pro (DeepSeek-AI, 2025)) under both baseline and instrumented feedback modes. Among the three agents, Claude Opus-4.6 achieves the highest success rate, solving 92 out of 181 cases (50%) in baseline mode and 111 out of 181 cases (61%) in instrumented mode.

Success rates by bug type. Figure 2 breaks down success rates by vulnerability type. The largest gains concentrate on memory-layout-sensitive categories, such as vmalloc-out-of-bounds (+22%) and slab-use-after-free (+19%) for Claude Opus-4.6, consistent with concrete runtime values closing the layout reasoning loop at allocation and access decision points. No single agent dominates across all bug types, indicating that aggregate metrics obscure where each agent’s reasoning capability is strongest.

Iteration efficiency. Beyond success rate, we examine whether instrumented feedback accelerates PoC generation by reducing the number of iterations required to reach a successful trigger. To isolate the effect of feedback on iterative refinement, we exclude 33 cases in which the instrumented agent succeeded on the first iteration, as these cases reflect correct initial generation rather than feedback-driven improvement.

Among the remaining cases, the instrumented agent required fewer iterations than the baseline agent in 77 cases (52%), indicating that structured branch coverage and runtime value feedback enables the agent to diagnose and correct failures more efficiently. This finding confirms that KPocInfra’s instrumentation not only expands the set of solvable cases but also reduces the computational cost of reaching a solution for cases that both modes eventually solve.

Capability gap analysis in failure cases.

KPocInfra’s instrumented mode externalizes the four reasoning capabilities as runtime signals, so comparing baseline and instrumented modes can be read as an ablation of which capabilities the agent must perform unaided from source alone. Failures that disappear under instrumentation indicate reasoning that the agent struggled to recover statically but can exploit once scaffolded by runtime evidence; failures that persist indicate gaps not closed by the current scaffolding.

To localize these gaps, we analyze every unsuccessful case across all three agents and both modes, attributing each failure to the primary reasoning capability that broke first. For each case, we inspect the agent’s reasoning log, the final generated PoC, and the instrumentation trace when available, and locate the earliest point at which the agent’s behavior diverged from the correct path. Table 2 summarizes the resulting distribution along with the subsystem breakdown within each capability.

State-machine reasoning is consistently the dominant failure mode. Across every (model, mode) configuration in Table 2, state-machine reasoning accounts for the largest share of failures, ranging from 42% to 59%. This confirms that driving the kernel through the ordered sequence of syscalls and intermediate states required to reach a vulnerable path is the single hardest reasoning challenge in kernel PoC generation, substantially more so than tracing individual values or shaping memory layouts. The pattern is robust across models of dif-

Table 2: Capability gap distribution across failure cases. Columns are (model, setup); rows report overall failure rate and capability-gap shares. Percentages are reported within each setup (baseline vs. instrumented).

	Claude Opus 4.6		GPT-5.4		DeepSeek V4 Pro	
	Base.	Instr.	Base.	Instr.	Base.	Instr.
Failed (%)	49	39	63	51	53	40
Capability share (%)						
State-Machine	49	42	59	51	53	49
Value-Flow	36	33	31	37	34	32
Concurrency	12	16	10	11	10	12
Memory-Layout	2	9	-	2	2	7

ferent families, suggesting that this gap is a property of the task itself rather than of any particular agent. When state-machine reasoning is scaffolded by branch-level reachability and guard outcomes, its failure share drops for Claude Opus-4.6 (49% $\rightarrow$ 42%) and OpenAI GPT-5.4 (59% $\rightarrow$ 51%). This drop directly measures state-machine reasoning that the agent fails to perform from source alone but can successfully exploit once externalized. A substantial gap nevertheless remains.

Value-flow reasoning is the second consistent bottleneck. Value-flow failures account for 31–37% of remaining failures across all configurations. Unlike state-machine, scaffolding value-flow signals does not consistently shrink its failure share, for OpenAI GPT-5.4 it rises from 31% to 37%. This reflects a compositional effect: externalizing state-machine evidence removes shallower failures and allows the agent to progress further into the kernel, where the remaining failures concentrate on deeper value-flow chains, such as tracing a user-controlled value through ten or more function boundaries.

Concurrency and memory-layout failures are comparatively rare. Concurrency accounts for 10–16% of failures, and memory-layout for 0–9%. Memory-layout’s particularly low share partly reflects that layout-sensitive cases are disproportionately solved rather than failed (consistent with the per-bug-type gains on slab- and vmalloc-out-of-bounds in Figure 2), leaving few such cases in the failure pool.

Subsystem distribution of failures. Tables 5–8 in the Appendix report the subsystem distribution within each capability gap, revealing where in the kernel each capability is hardest to exercise. Two patterns stand out. First, the filesystem (FS) sub-

system dominates both state-machine failures (40–67% across configurations) and value-flow failures (47–74%), reflecting the deep state hierarchies and long user-input-to-guard chains characteristic of VFS and individual filesystem implementations. Second, memory-layout failures concentrate almost entirely in the networking (Net) and device (Dev) subsystems, where complex object lifecycles around sockets, sk_buffs, and device-bound allocations make heap shaping particularly hard to reason about from source alone. These patterns suggest that targeted improvements to filesystem-aware state reasoning would yield disproportionately large gains across both dominant capability gaps.

Implications. State-machine reasoning is the highest-impact target for future agent work: a moderate improvement there would translate into the largest success-rate gain, after which value-flow reasoning becomes the next frontier.

Capability improvement analysis. Beyond the aggregate counts above, we ask whether instrumented feedback produces measurable progress even on cases that remain unsolved. For each case that fails under both modes, we compare the earliest divergence point in the baseline trace against that in the instrumented trace, using the depth of the divergence along the vulnerability-relevant call chain as a proxy for how far the generated PoC advanced toward the trigger condition. We find that in 29 such cases, the instrumented PoC diverges at a strictly deeper point than the baseline PoC, meaning the improved capability signal let the agent drive the kernel further into the code path even when it ultimately failed to reach the bug. This indicates that the benefit of instrumented feedback is not fully captured by the binary success-or-failure

verdict: instrumentation advances the frontier of what the agent can reason about in real kernel code, and the remaining gap is a matter of *how far* that frontier still falls short of the trigger, not of whether the agent made any progress at all.

4 Conclusion and Future Work

We presented a systematic capability-attributed study of LLM-driven Linux kernel PoC generation. Using KPocBench and capability-attributed evaluation, we identified a robust ordering of reasoning bottlenecks across three frontier LLM agents: state-machine reasoning dominates, followed by value-flow, with concurrency and memory-layout as smaller, model-specific gaps. This ordering identifies state-machine reasoning as the most impactful target for future agent work, a prioritization that aggregate success rates alone cannot provide.

Several directions extend this study. First, our capability taxonomy and instrumentation can be extended to other kernel sanitizers, including KCSAN for data races and KMSAN for uninitialized memory, broadening coverage beyond KASAN-detected memory safety. Second, future agents could learn to actively query the instrumentation infrastructure for the specific signals their current hypothesis requires, turning capability scaffolding into an interactive reasoning protocol.

Limitations

Our study has several limitations. First, our benchmark focuses exclusively on KASAN-detected memory-safety bugs from Syzbot. Other classes of kernel vulnerabilities, such as KCSAN-detected data races ([Linux Kernel Documentation, 2026b](#)), KMSAN-detected uninitialized-memory bugs ([Linux Kernel Documentation, 2026c](#)), locking bugs, and logic flaws without an associated sanitizer, are out of scope, and our capability attribution may not generalize directly to them. Second, we evaluate three frontier LLM agents (Claude Opus-4.6, OpenAI GPT-5.4, and Deepseek V4-Pro); while we observe consistent patterns across these three model families, results on smaller open-source models or on future models with substantially different training regimes may differ. Also, most of the results are collected on Feb. 2026, so as the time goes by, the capability of models could be different. Third, our capability-gap attribution involves manual inspection of agent reasoning logs and instrumentation traces, which introduces some

subjectivity; we mitigate this by anchoring each attribution to concrete branch-level evidence wherever available, but a fully automated attribution pipeline remains future work. Fourth, all experiments are run under a fixed iteration and time budget; the relative ordering of capability gaps could shift under substantially larger budgets, particularly for value-flow and concurrency reasoning which often require deeper exploration. Finally, our PoCs target already-disclosed and patched bugs, and we do not study PoC generation for fix-validation (regression) or for zero-day vulnerabilities, that being said, there could be potential memorized bugs or leaked information that causing data contamination.

Ethical Considerations

This work studies LLM agents’ ability to generate Proof-of-Concept programs for Linux kernel vulnerabilities. All 181 cases in KPocBench correspond to bugs that have been publicly disclosed through Syzbot and patched upstream; we do not study zero-day vulnerabilities or embargoed disclosures. The PoCs reproduce sanitizer-detected crashes (KASAN reports) rather than weaponized exploits achieving privilege escalation or arbitrary code execution, and the capabilities measured by our benchmark apply equally to defensive workflows such as patch regression testing and root-cause analysis. We will release KPocBench and KPocInfra under a research-only license consistent with Syzbot’s existing public-disclosure practice.

On the other hand, we use the AI assistant (LLMs) to polish the grammar and wording in Section 1 and Section 3 (written by human), instead of directly generating content.

References

- Anthropic. 2025. [Claude opus 4: Model card](#).
- Claas Beger and Saikat Dutta. 2025. Coconut: Structural code understanding does not fall out of a tree. In *2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*, pages 128–136. IEEE.
- Fabrice Bellard and The QEMU Project. 2003. [QEMU: A generic and open source machine emulator and virtualizer](#). Accessed 2026.
- Junkai Chen, Zhiyuan Pan, Xing Hu, Zhenhao Li, Ge Li, and Xin Xia. 2025. Reasoning runtime behavior of a program with llm: How far are we? In *2025*

- IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 1869–1881. IEEE.
- Yueqi Chen and Xinyu Xing. 2019. SLAKE: Facilitating slab manipulation for exploiting vulnerabilities in the Linux kernel. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
- DeepSeek-AI. 2025. DeepSeek-V3.2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*.
- Richard Fang, Rohan Bindu, Akul Gupta, and Daniel Kang. 2024. LLM agents can autonomously exploit one-day vulnerabilities. *arXiv preprint arXiv:2404.08144*.
- Google. 2015. [syzkaller: an unsupervised coverage-guided kernel fuzzer](#). Accessed 2026.
- Google. 2017. [syzbot: continuous fuzzing infrastructure for the Linux kernel](#). Accessed 2026.
- Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I Wang. 2024. Cruxeval: A benchmark for code reasoning, understanding and execution. *arXiv preprint arXiv:2401.03065*.
- Kaivalya Hariharan, Uzay Girit, Atticus Wang, and Jacob Andreas. 2025. Breakpoint: Scalable evaluation of system-level reasoning in LLM code agents. *arXiv preprint arXiv:2506.00172*.
- Sean Heelan. 2025. [Anamnesis: Automatic exploit generation with LLMs](#).
- Naman Jain, Shubham Gandhi, Atharv Sonwane, Aditya Kanade, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2023. Staticfixer: From static analysis to static repair. *arXiv preprint arXiv:2307.12465*.
- Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. 2024. SWE-bench: Can language models resolve real-world GitHub issues? In *The Twelfth International Conference on Learning Representations (ICLR)*.
- Rob Kopel. Execution-eval: Can language models execute real-world code?
- Emanuele La Malfa, Christoph Weinhuber, Orazio Torre, Fangru Lin, X Angelo Huang, Samuele Marro, Anthony Cohn, Nigel Shadbolt, and Michael Wooldridge. 2025. Code simulation as a proxy for high-order tasks in large language models. *arXiv preprint arXiv:2502.03568*.
- Yoochan Lee and 1 others. 2021. ExpRace: Exploiting kernel races through raising interrupts. In *USENIX Security Symposium*.
- Weichen Li, Albert Jan, Baishakhi Ray, Junfeng Yang, Chengzhi Mao, and Kexin Pei. 2025. Editlord: Learning code transformation rules for code editing. *arXiv preprint arXiv:2504.15284*.
- Weichen Li, Jiamin Song, Bogdan Alexandru Stoica, Arav Dhoot, Gabriel Ryan, Shengyu Fu, and Kexin Pei. 2026. Semrep: Generative code representation learning with code transformations. *arXiv preprint arXiv:2603.13640*.
- Ziyang Li, Saikat Dutta, and Mayur Naik. 2024. IRIS: LLM-assisted static analysis for detecting security vulnerabilities.
- Linux Kernel Documentation. 2026a. [The kernel address sanitizer \(KASAN\)](#). Accessed 2026.
- Linux Kernel Documentation. 2026b. [The kernel concurrency sanitizer \(KCSAN\)](#). Accessed 2026.
- Linux Kernel Documentation. 2026c. [The kernel memory sanitizer \(KMSAN\)](#). Accessed 2026.
- Changshu Liu, Alireza Ghazanfari, Yang Chen, and Reyhaneh Jabbarvand. 2025. Evaluating code reasoning abilities of large language models under real-world settings. *arXiv preprint arXiv:2512.14917*.
- Changshu Liu, Shizhuo Dylan Zhang, Ali Reza Ibrahimzada, and Reyhaneh Jabbarvand. 2024a. Codemind: A framework to challenge large language models for code reasoning. *arXiv preprint arXiv:2402.09664*.
- Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. 2024b. Large language model-based agents for software engineering: A survey. *ACM Transactions on Software Engineering and Methodology*.
- Yu Liu, Sergey Mechtaev, Pavle Subotić, and Abhik Roychoudhury. 2023. Program repair guided by datalog-defined static analysis. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1216–1228.
- Xuying Ning, Katherine Tieu, Dongqi Fu, Tianxin Wei, Zihao Li, Yuanchen Bei, Jiaru Zou, Mengting Ai, Zhining Liu, Ting-Wei Li, and 1 others. 2026. Code as agent harness. *arXiv preprint arXiv:2605.18747*.
- Yu Nong and 1 others. 2024. A case study of LLM for automated vulnerability repair: Assessing impact of reasoning and patch validation feedback. *arXiv preprint arXiv:2405.15690*.
- OpenAI. 2025. [GPT-5 system card](#).
- Monoshi Kumar Roy, Simin Chen, Benjamin Steenhoek, Jinjun Peng, Gail Kaiser, Baishakhi Ray, and Wei Le. 2025. Codesense: a real-world benchmark and dataset for code semantic reasoning. *arXiv preprint arXiv:2506.00750*.

- Rijnard Van Tonder and Claire Le Goues. 2018. Static automated program repair for heap properties. In *Proceedings of the 40th International Conference on Software Engineering*, pages 151–162.
- Zhun Wang, Tianneng Shi, Jingxuan He, Matthew Cai, Jialin Zhang, and Dawn Song. 2025. CyberGym: Evaluating AI agents’ real-world cybersecurity capabilities at scale. *arXiv preprint arXiv:2506.02548*.
- Anjiang Wei, Jiannan Cao, Ran Li, Hongyu Chen, Yuhui Zhang, Ziheng Wang, Yuan Liu, Thiago SFX Teixeira, Diyi Yang, Ke Wang, and 1 others. 2025. Equibench: Benchmarking large language models’ understanding of program semantics via equivalence checking. In *NeurIPS 2025 Fourth Workshop on Deep Learning for Code*.
- Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. 2018. FUZE: Towards facilitating exploit generation for kernel use-after-free vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 781–797.
- Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*.
- Danning Xie, Mingwei Zheng, Xuwei Liu, Jiannan Wang, Chengpeng Wang, Lin Tan, and Xiangyu Zhang. 2025. CORE: Benchmarking LLMs’ code reasoning capabilities through static analysis tasks. In *Advances in Neural Information Processing Systems (NeurIPS) Track on Datasets and Benchmarks*. ArXiv:2507.05269.
- Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Jianan Yao, Weidong Cui, Yeyun Gong, Chris Hawblitzel, Shuvendu Lahiri, Jacob R Lorch, Shuai Lu, and 1 others. 2025a. Autoverus: Automated proof generation for rust code. *Proceedings of the ACM on Programming Languages*, 9(OOPSLA2):3454–3482.
- Chenyuan Yang, Natalie Neamtu, Chris Hawblitzel, Jacob R Lorch, and Shan Lu. 2025b. Verusage: A study of agent-based verification for rust systems. *arXiv preprint arXiv:2512.18436*.
- Jun Yang, Yuechun Sun, Yi Wu, Rodrigo Caridad, Yongwei Yuan, Jianan Yao, Shan Lu, and Kexin Pei. 2026. Exverus: Verus proof repair via counterexample reasoning. *arXiv preprint arXiv:2603.25810*.
- Mengyao Zhao, Kaixuan Li, Lyuye Zhang, Wenjing Dang, Chenggong Ding, Sen Chen, and Zheli Liu. 2025. A systematic study on generating web vulnerability proof-of-concepts using large language models. *arXiv preprint arXiv:2510.10148*.
- Jie Zhu, Chihao Shen, Ziyang Li, Jiahao Yu, Yizheng Chen, and Kexin Pei. 2025a. Locus: Agentic predicate synthesis for directed fuzzing. *arXiv preprint arXiv:2508.21302*.
- Yuxuan Zhu and 1 others. 2025b. CVE-bench: A benchmark for AI agents’ ability to exploit real-world web application vulnerabilities. *arXiv preprint arXiv:2503.17332*.
- Arastoo Zibaeirad and Marco Vieira. 2024. VulnLLMEval: A framework for evaluating large language models in software vulnerability detection and patching. *arXiv preprint arXiv:2409.10756*.

A Appendix

A.1 KPocInfra

KPocInfra is a runtime awareness infrastructure for PoC generation tasks of Linux kernel vulnerabilities. The key design is to provide automatically observable runtime feedback to the generator (here indicates to LLM agent). In order to provide such capability in infrastructure level, we implement and integrate several critical functionalities to achieve that, including source-code layer instrumentation, Kernel driver injection, and kernel memory dump.

Inputs. As described in §2.5, the core target of KPocInfra is to generate the PoC for Linux kernel vulnerabilities based on the bug report. So the basic input is only the relevant vulnerability report of a certain bug. The report needs to contains two critical items: 1) the (potential) type of vulnerability; 2) the (potential) stack trace or call chain of the vulnerability. The vulnerability report is not necessarily generated from the dynamic execution, but can also be derive from the static analyzer.

Source-code Layer Instrumentation. Guided by the provided vulnerability report, KPocInfra initiates its workflow by systematically extracting the set of functions manifested within the runtime stack trace or the associated call chain. Operating in parallel, KPocInfra deploys LLM-based agents to perform an in-depth semantic analysis of the bug report while autonomously navigating the kernel codebase to inspect pertinent source files and header definitions. The primary objective of this dual-channel information acquisition strategy is to facilitate a more holistic and comprehensive identification of critical functions, thereby ensuring high-fidelity coverage during the subsequent instrumentation phase. This design choice is necessitated by the fact that stack traces in conventional bug reports are inherently coupled with the immediate, "post-mortem" context of the vulnerability’s triggering point. While such localized information offers an efficient and precise means of pinpointing

the crash site, it frequently leads to the inadvertent omission of precursor functions—those that were executed earlier in the control flow but had already returned before the system state was captured. Consequently, KPocInfra transcends the inherent limitations of static log data; it empowers the LLM to conduct a global, codebase-level retrieval and exploration, leveraging the model’s sophisticated reasoning capabilities and pre-trained security knowledge to uncover latent critical functions that are essential for reconstructing the full vulnerability lifecycle but remain absent from the final crash trace.

Kernel Driver Injection. In KPocInfra, we design and implement the kernel layer driver injection to make the whole kernel level vulnerability runtime context capturing can be a plug-in module. To achieve it, instead of directly modifying Linux kernel code in a large-scale, we implement a customized Linux kernel module named GateTracer. To enable or disable the GateTracer, KPocInfra also introduce a kernel config option CONFIG_GATE_TRACE.

To effectively record the kernel runtime information, every branch KPocInfra insert a GATE_TRACE_BEFORE(id, expr_1, expr_2, expr_3) and a GATE_TRACE_AFTER(id). Technically, each instrument has its own id as the unique identifier. On the one hand, for the before gate, there are three slots for it to record the valuable meta data in a branch. The reason is to expose the meta data which might have an impact on the vulnerability trigger process. The before gate contains the information both in control flow and data flow: 1) whether this position is reached? 2) what’s the value of meta data when trying to trigger the vulnerability? On the other hand, for after gate, we only need to record whether the control flow entering such in-branch codes, since the value of meta data already been recorded by the before gate.

As we shown in List 1, the original branch is transform and wrapped by the BEFORE instrument and AFTER instrument, the first parameter of the instrument macro is the id, the following three parameters represent three different expressions or variables that LLM recognize it’s valuable to record. Each time when a certain instrument is reached or triggered the parameters are converted to the unsigned long type, then written to the memory space we reserved. The List 2 shows the implementation of the two macros that

```

1 @@ -899,15 +900,25 @@ struct bdev_handle
   ↪ *bdev_open_by_path(const char *path,
   ↪ blk_mode_t mode,
2     int error;
3
4     error = lookup_bdev(path, &dev);
5     if (error)
6         return ERR_PTR(error);
7     {
8         GATE_TRACE_BEFORE(0,
   ↪ (unsigned long)(error), 0, 0);
9         if (error)
10            {
11                GATE_TRACE_AFTER(0);
12                return
   ↪ ERR_PTR(error);
13            }
14    }
15
16    handle = bdev_open_by_dev(dev,
   ↪ mode, holder, hops);
17    if (!IS_ERR(handle) && (mode &
   ↪ BLK_OPEN_WRITE) &&
18    {
19        GATE_TRACE_BEFORE(1,
   ↪ (unsigned long)(IS_ERR(handle)),
   ↪ (unsigned long)(mode &
   ↪ BLK_OPEN_WRITE), 0);
20        if (!IS_ERR(handle) &&
   ↪ (mode & BLK_OPEN_WRITE) &&
21        bdev_read_only(handle->bdev)) {
22            GATE_TRACE_AFTER(1);
23            bdev_release(handle);
24            return ERR_PTR(-EACCES);
25        }
26    }
27
28    return handle;
29 }

```

Listing 1: The example of GateTracer instrumentation.

KPocInfra used in instrumentation stage. Except from the id and three meta data value parameters, each macro also implicitly record the __gate_trace_loc, which refers to the value of the RIP register when kernel execution reach the instrument. Such operation benefits the reverse mapping from the position of instrument to the position in Linux kernel source code (the pair of file:line).

When the kernel is launching, the GateTracer initialized itself by setting all the instrument data to zero, and create the related DebugFS file. To avoid the disturb to the Linux kernel dynamic memory layout, KPocInfra put all the instruments in global static area (BSS section).

During the kernel’s booting process, it might already start to trigger the instrumented functions, which could result in the unexpected instrumenta-

```

1 #define GATE_TRACE_BEFORE(id, expr_1,
  ↳ expr_2, expr_3) do {
2   __label__ __gate_trace_loc;
3   __gate_trace_loc:
4   gate_trace_record_before_multi(
5     (id),
6     (unsigned long)&__gate_trace_loc,
7     ↳ /* RIP capture */
8     (unsigned long)(expr_1),
9     (unsigned long)(expr_2),
10    (unsigned long)(expr_3),
11    0, /* cond3 unused */
12    3); /* num_conds = 3 */
13 } while (0)
14 #define GATE_TRACE_AFTER(id) do {
15   __label__ __gate_trace_loc;
16   __gate_trace_loc:
17   gate_trace_record_after(
18     (id),
19     (unsigned long)&__gate_trace_loc);
20 } while (0)

```

Listing 2: The implementation of two instrumentation macros.

tion records. To avoid this problem, KPocInfra also introduce three control commands that can be send from DebugFS from the userspace interface in `/sys/kernel/debug/gate_trace/control`. The command "reset" will clears all counters; "enable" will make all instrumentation macros now ACTIVE; "disable" will stop all the counters.

Dump Linux Kernel Memory. KPocInfra records the kernel runtime execution information in guest’s kernel layer. To extract such information in host, it will dump the guest’s kernel memory when the generated PoC finished running. To do so, after booting succeed, and the initialization finished, the GateTracer will do address announcement by printing out the kernel memory address that contains all the instruments. At this moment, the kernel QEMU monitor will detect it and record the memory address. Once PoC is finished, there are two different status: Crash or No Crash. The QEMU monitor will directly use the memsave to dump the memory as it records from address announcement previously. The dumped memory segment is raw bytes, KPocInfra will unpack and utilize `llvm-addr2line` to convert and map the raw address bytes to actual location in Linux kernel source code, also the each instrument’s meta data information and triggering information, then save as the readable files.

Table 3: Bug type across the 181 evaluation cases.

Bug Type	Count	%
use-after-free	60	33.15%
slab-out-of-bounds	49	27.07%
slab-use-after-free	26	14.36%
null-ptr-deref	25	13.81%
vmalloc-out-of-bounds	9	4.97%
double-free	3	1.66%
global-out-of-bounds	3	1.66%
stack-out-of-bounds	3	1.66%
out-of-bounds	2	1.10%
user-memory-access	1	0.55%

Table 4: Race condition cases by bug type (57 out of 181 total).

Bug Type	Race Cases
use-after-free	29
slab-use-after-free	13
slab-out-of-bounds	5
vmalloc-out-of-bounds	5
null-ptr-deref	4
double-free	1

A.2 Subsystem breakdown within capability gaps

A.3 Use of LLM Disclosure

AI writing assistants (LLMs) were used for language polishing and grammar/wording refinement of the manuscript only in Section 1 and Section 3. No AI tool was used to generate research ideas, design experiments, or write code beyond standard IDE autocomplete. All technical content, analysis, and conclusions are the authors’ own. This use falls under the ACL Policy on AI Writing Assistance categories (a) "Assistance purely with the language of the paper" and (b) "Short-form input assistance."

A.4 More models

we conducted an experiment about the GLM 5.2 model under the same settings. As for GLM 5.2 model, for BASELINE MODE, it succeed 86 cases (47.5%); and for INSTRUMENTED MODE, GLM 5.2 succeed 110 cases (60.7%). In Baseline failures (95 in total), State-Machine is 55%, Value-Flow is 36%, Concurrency is 9%, Memory-Layout is 0%. In instrumented failures (71 in total), State-Machine is 49%, Value-Flow is 41%, Concurrency

Table 5: Subsystem share within *State-Machine* failures (%). Percentages are reported within each setup (baseline vs. instrumented).

	Claude Opus 4.6		GPT-5.4		DeepSeek V4 Pro	
	Base. Instr.		Base. Instr.		Base. Instr.	
FS	40	54	55	64	53	67
Dev	31	29	18	21	18	17
Net	17	11	24	13	18	8
Mem	5	-	-	-	4	3
USB	5	7	4	2	-	-
Knl	2	-	-	-	8	6

Table 6: Subsystem share within *Value-Flow* failures (%). Percentages are reported within each setup (baseline vs. instrumented).

	Claude Opus 4.6		GPT-5.4		DeepSeek V4 Pro	
	Base. Instr.		Base. Instr.		Base. Instr.	
FS	74	73	62	47	70	61
Dev	16	23	17	18	15	17
Net	3	5	10	21	12	13
Knl	3	-	3	6	-	-
Mem	-	-	3	3	3	9
USB	-	-	-	3	-	-
Other	-	-	3	3	-	-
BPF	3	-	-	-	-	-

is 6%, Memory-Layout is 4%. These results line up closely with our main findings in two ways. First, the gain from instrumented feedback is 13.3 points (47.5% to 60.8%), which falls in the same range as the other three models (roughly 11 to 13 points), showing that the benefit of telemetry is not specific to the models we originally evaluated. Second, the ordering of capability bottlenecks is identical, with state machine reasoning dominant (55% and 49%), value flow second (36% and 41%), and concurrency and memory layout smaller. This consistency on an additional model strengthens our claim that the observed ordering reflects the structure of the kernel PoC generation task rather than the weakness of any single LLM model.

Table 7: Subsystem share within *Concurrency* failures (%). Percentages are reported within each setup (baseline vs. instrumented).

	Claude Opus 4.6		GPT-5.4		DeepSeek V4 Pro	
	Base.	Instr.	Base.	Instr.	Base.	Instr.
FS	30	55	-	20	60	67
Dev	20	-	22	10	20	22
Net	20	27	22	30	-	-
KnI	30	18	44	30	10	11
Mem	-	-	11	10	10	-

Table 8: Subsystem share within *Memory-Layout* failures (%). Percentages are reported within each setup (baseline vs. instrumented).

	Claude Opus 4.6		GPT-5.4		DeepSeek V4 Pro	
	Base.	Instr.	Base.	Instr.	Base.	Instr.
FS	-	50	-	-	-	-
Dev	-	17	-	-	100	60
Net	100	33	-	100	-	40