

Code Researcher: Deep Research Agent for Large Systems Code and Commit History

Ramneet Singh* Sathvik Joel* Abhav Mehrotra Nalin Wadhwa
 Ramakrishna B Bairi Aditya Kanade Nagarajan Natarajan
 Microsoft Research

{ramneet2001,ksjoe30,abhavm1,nalin.wadhwa02}@gmail.com
 {ram.bairi,kanadeaditya,nagarajan.natarajan}@microsoft.com

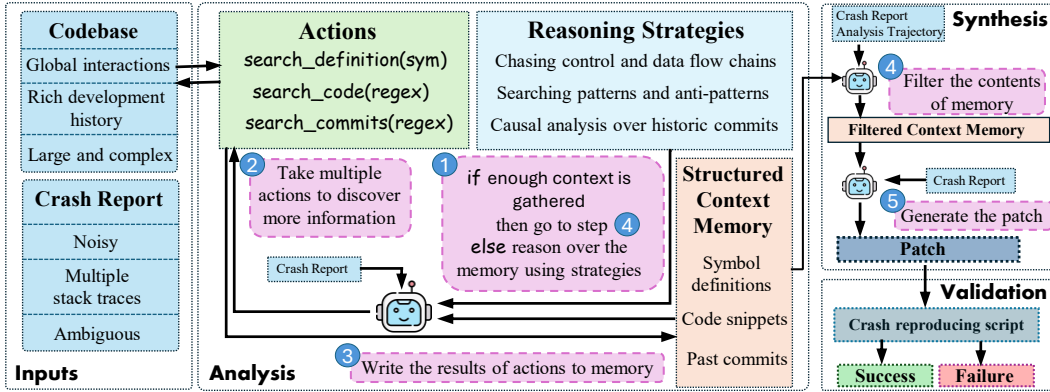


Figure 1: *Code Researcher* conducts deep research over code in three phases: (1) Starting with the codebase and crash report as input, the **Analysis** phase performs multi-step reasoning about semantics, patterns, and commit history of code. It gathers context in a structured memory. (2) The **Synthesis** phase filters the contents of the memory to keep relevant context and generates a patch. (3) The **Validation** phase uses external tools to validate the patch.

Abstract

Large Language Model (LLM)-based coding agents have shown promising results on coding benchmarks, but their effectiveness on systems code remains underexplored. Due to the size and complexities of systems code, making changes to a systems codebase is a daunting task, even for humans. It requires *researching* about many pieces of context, derived from the large codebase and its massive commit history, *before* making changes. Inspired by the recent progress on deep research agents, we design the first deep research agent for code, called *Code Researcher*, and apply it to the problem of generating patches for mitigating crashes reported in systems code. *Code Researcher* performs multi-step reasoning about semantics, patterns, and commit history of code to gather sufficient context. The context is stored in a structured memory which is used for synthesizing a patch. We evaluate *Code Researcher* on kBenchSyz [23], a benchmark of Linux kernel crashes, and show that it significantly outperforms strong baselines, achieving a crash-resolution rate of 58%, compared to 37.5% by SWE-agent [40]. On an average, *Code Researcher* explores 10 files in each trajectory whereas SWE-agent explores only 1.33 files, highlighting *Code Researcher*’s ability to deeply explore the codebase. Through another experiment on an open-source multimedia software, we show the generalizability of *Code Researcher*. Our experiments highlight the importance of global context gathering and multi-faceted reasoning for large codebases.

*Equal contribution

1 Introduction

Automating coding using Large Language Models (LLMs) and LLM-based agents has become a very active area of research. Popular benchmarks like LiveCodeBench [16] and SWE-bench [17] respectively test coding abilities on standalone competitive coding problems and GitHub issues over library or application code. Despite the demonstrated progress of LLM-based coding agents on these benchmarks, they are yet to scale to complex tasks over an important class of code, *systems code*.

Systems code powers critical and foundational software like file and operating systems, networking stacks, distributed cloud infrastructure and system utilities. Systems codebases have multiple dimensions of complexity. Firstly, they are *very large*, containing thousands of files and millions of lines of code. Secondly, systems code often interfaces directly with the hardware and is performance critical. This results in *complex low-level code* (involving pointer and bit manipulations, compile-time macros, etc.) in languages like C/C++, and *global interactions* between different parts of the codebase for concurrency, memory management, maintenance of data-structure invariants, etc. Finally, foundational systems codebases have *rich development histories* spanning years or even decades, containing contributions by hundreds or thousands of developers, which are important references on legacy design decisions and code changes.

Due to the size and complexities of systems code, making changes to a systems codebase is a daunting task, even for humans. Automating such changes requires a different type of agents, agents that can *research* about many pieces of context, derived automatically from the large codebase and its massive commit history, *before* making changes. Recently, *deep research* agents have been developed to solve complex, knowledge-intensive problems that require careful context gathering and multi-step reasoning, before synthesizing the answer. The agents and techniques have mostly focused on long-form document generation or complex question-answering over web contents [32, 30, 7, 31, 20, 38] and enterprise data [1, 25]. Inspired by these advances, we propose the first deep research agent for code, called *Code Researcher*, and apply it to the problem of generating patches for mitigating crashes reported in systems code.

As shown in Figure 1, *Code Researcher* works in three phases: (1) **Analysis**: Starting with the crash report and the codebase, this phase performs multi-step reasoning over semantics, patterns, and commit history of code. The “Reasoning Strategies” block shows the reasoning strategies used. Each reasoning step is followed by invocations of tools (labeled “Actions” in Figure 1) to gather context over the codebase and its commit history. The information gathered is stored in a *structured context memory*. When the agent is able to conclude that it has gathered sufficient context, it moves to the next phase. (2) **Synthesis**: The **Synthesis** phase uses the crash report, the context memory, and the reasoning trace of the **Analysis** phase to filter out irrelevant memory contents. Then, it identifies one or more buggy code snippets from memory, possibly spread across multiple files, and generates patches. (3) **Validation**: Finally, the **Validation** phase checks if the generated patches prevent the crash from occurring using external tools. A successful patch is presented to the user.

The Linux kernel [21] is a canonical example of a systems codebase with complex low-level code and massive size (75K files and 28M lines of code), and has rich development history. Mathai et al. [23] recently proposed a benchmark, called kBenchSyz, of 279 Linux kernel crashes detected by the Syzkaller fuzzer [12]. Through extensive experimentation on this challenging benchmark, we evaluate the effectiveness of *Code Researcher* and compare it to strong baselines. Most of the existing coding agents are geared towards resolving bugs in moderately-sized codebases given issue descriptions, as exemplified by the popular SWE-bench [17] benchmark. The issue descriptions are written by humans, wherein they explain the nature of the bug and which files are likely relevant. Coding agents [40, 36] are designed to take advantage of this and quickly navigate the repository to reach the buggy files. They do not expend much efforts in gathering codebase-wide context. In our setting, the bugs are described by stack traces which are devoid of natural language hints and typically contain a much larger number of files and functions than an issue description. Therefore, multi-step reasoning and context gathering becomes more important. Our experimental results bear a strong witness to this.

As a strong baseline, we customized SWE-agent [40], a SOTA open-source agent on SWE-bench, for kernel crash resolution. *Code Researcher* resolved 48% crashes compared to 31.5% of SWE-agent with a budget of 5 trajectories each and GPT-4o as the LLM. *Code Researcher* explored about 10 files per trajectory compared to a much smaller number 1.33 of files explored by SWE-agent. Further, in a

direct comparison on 90 bugs where both *Code Researcher* and SWE-agent edit all the ground-truth buggy files, the resolution rate of *Code Researcher* is 61.1% compared to 37.8% of SWE-agent. This clearly shows that *Code Researcher* is able to research and gather more useful context. Using o1 only for patch generation, *Code Researcher*’s performance improves to 58%, showing that well-researched context enables a reasoning model to improve the performance significantly.

Concurrent to our work, Mathai et al. [24] have proposed a specialized agent for resolving Linux kernel crashes. However, they perform evaluation in the *assisted setting* wherein (a) the agent is provided the ground-truth buggy files to edit, and (b) they build Linux-kernel specific tooling to scale. In contrast, we evaluate in the realistic, *unassisted setting* in which *Code Researcher* has to identify the buggy files by itself, using general search tools. Another factor that distinguishes our work is the use of commit history, which to the best of our knowledge, none of the existing coding agents do. Commit history is known to contain important information [18]. With an ablation study, we show that searching over commits plays an important role in the success of *Code Researcher*.

In addition to the thorough experimentation on kBenchSyz, we also experiment on an open-source multimedia software, FFmpeg [3]. *Code Researcher* was able to generate crash-preventing patches for 7/10 crash reports tested, establishing its generalizability.

In summary, we make the following main contributions:

- (1) We design the first deep research agent for code, *Code Researcher*, capable of handling large systems code and resolving crashes. Recognizing the importance of commit history in systems code, we equip the agent with a tool to efficiently search over commit histories.
- (2) We evaluate *Code Researcher* on the challenging kBenchSyz benchmark [23] and achieve a crash resolution rate of 58%, outperforming strong baselines. We also demonstrate generalizability of *Code Researcher* on a multimedia software, FFmpeg.
- (3) Through a comprehensive evaluation, we provide insights such as (i) how our deep research agent outperforms agents that do not focus on gathering relevant context, (ii) that this advantage persists even if the existing SOTA agent is given higher inference-time compute, and (iii) reasoning models improve performance significantly if given well-researched context.

2 Related work

LLM-powered software development subfield has produced several autonomous coding agents [40, 39, 36, 42, 35], predominantly evaluated on SWE-bench [17]. SWE-bench focuses on GitHub issues from small to medium-sized Python repositories. However, systems code, the focus of our work, presents unique challenges. We highlight and contrast key related work in this context.

Coding agents Agents like SWE-agent [40] or OpenHands [36] use a single ReAct-style [41] loop endowed with shell commands or specialized tools for file navigation and editing. However, agents like these do not use program structure to traverse the codebase (e.g., following data and control flow chains) and are not designed to reason about complex interactions and gather context. As a result, they tend to explore a small number of files per bug and make an edit, without gathering and reasoning over the full context of the bug (see Section 5.3). AutoCodeRover [42] uses tools based on program structure to traverse the codebase (albeit limited to Python code). It performs explicit localization of the functions/classes to edit using these tools, and those are later repaired. *Code Researcher* does not explicitly localize the functions to edit; instead it gathers relevant context for patch generation and decides what to edit in the Synthesis phase. *Code Researcher* is also the first agent to incorporate causal analysis over historical commits; this is critical to handling subtle bugs introduced by code evolution in long-lived systems codebases.

Deep research agents Deep research is a fast emerging subfield in agentic AI [25, 30, 7, 31], to tackle complex, knowledge-intensive tasks, that can take hours or days even for experts. Academic work so far has focussed on long-form document generation [5, 32], scientific literature review [38, 14], and complex question-answering [20, 37] based on the web corpus. The key challenges in deep research for such complex tasks include (a) intent disambiguation, (b) exploring multiple solution paths (breadth of exploration), (c) deep exploration (iterative tool interactions and reasoning), and (d) grounding (ensuring that the claims in the response are properly attributed). Most of the aforementioned challenges also apply to our setting. To the best of our knowledge, our work is the first to design and evaluate a deep research strategy for complex bug resolution in large codebases.

Most recently, OpenAI’s Deep Research model has been integrated with GitHub repos for report generation and QA over codebases [29]. However, (a) it does not support agentic tasks like bug fixing, and (b) there is no report on its effectiveness in real-world developer tasks.

Long context reasoning Support for increasing context length sizes in LLMs has been an active area of research [33, 15], opening up the possibility of feeding the entire repository into an LLM’s context and generating a patch. But there are a few complications. First, note that the Linux kernel has over 75K files and 28 Million lines of code. In contrast, state of the art models today (e.g., Gemini 2.5 Pro) support at most 2M tokens in the context window [8, 9], roughly corresponding to around 100K lines of code [8]. Second, long-context models do not robustly make use of the information in context. They often get “lost in the middle” [22], performing highest when relevant information occurs at the beginning or end of the input context, and significantly worse when they must access relevant information in the middle of long contexts. Li et al. [19] found that long-context LLMs struggle with processing long, context-rich sequences and reasoning over multiple pieces of information (which is important for any automated software development task).

Automated kernel bug detection and repair Prior work for detecting Linux kernel bugs includes various types of sanitizers, e.g., Kernel Address Sanitizer (KASAN) [10], and the Syzkaller kernel fuzzer [12], an unsupervised coverage-guided fuzzer that tries to find inputs on which the kernel crashes. *Code Researcher*, complementary to this, generates patches from crash reports. We use some traditional software engineering concepts like deviant pattern detection [4] and reachability analysis [26], but leverage LLMs to scale to large codebases (Section 3). As noted earlier, CrashFixer [24] targets Linux kernel crashes but assumes that buggy files are known *a priori*. This assumption is unrealistic for large codebases like the Linux kernel. In contrast, *Code Researcher* autonomously locates buggy files using general search tools.

3 Design of *Code Researcher*

Large systems codebases, owing to their critical nature, undergo strict code development and reviewing practices by expert developers. The bugs that still sneak in are subtle and involve violations of global invariants (e.g., a certain data structure should be accessed only after holding a specific lock) and coding conventions (e.g., use of a specific macro to allocate memory), and past changes that cause unintended side effects. To fix such bugs, an agent needs to gather sufficient context about the codebase and its commit history, before it can generate any hypotheses about the cause of a bug and attempt to fix it. With this insight, we design our deep research agent, *Code Researcher*. As shown in Figure 1, *Code Researcher* comprises of three phases: (1) Analysis, (2) Synthesis and (3) Validation. We discuss the design of these phases in details now. The detailed prompts for all these phases are provided in the supplementary material.

3.1 Analysis phase

The Analysis phase of *Code Researcher* is responsible for performing deep research to understand the cause of a reported crash. We equip this phase with (a) actions to efficiently search over the codebase and the commit history, (b) reasoning strategies for code, and (c) a structured context memory.

3.1.1 Actions to search over codebase and commit history

We support the following actions: (1) `search_definition(sym)`: To search for the definition(s) of the specified symbol, which can be the name of a function, struct, global constant, union or macro and so on. It can be optionally passed the file name to limit the search to a file. (2) `search_code(regex)`: To search the codebase for matches to the specified regular expression. This is a simple yet powerful tool, which can be used for searching for any coding pattern such as call to a function, dereferences to a pointer, assignment to a variable and so on. (3) `search_commits(regex)`: To search for matches to a regular expression over commit messages and diffs associated with the commits. The regular expression offers expressiveness, e.g., to search for occurrence of a term (“memory leak”) in the commit messages or coding patterns in code changes (diffs). In addition, the agent can invoke (4) `done` to indicate that it has finished the Analysis phase and (5) `close_definition(sym)`: To remove the definition of a symbol from the memory if the symbol is deemed irrelevant to the task.

3.1.2 Reasoning strategies for code

We ask the agent to explore the codebase to figure out the root cause of a crash and gather sufficient context to propose a fix. The crash reports consist of stack traces and additional information generated by diagnostic tools such as an address sanitizer [10] that detects memory corruption, a concurrency sanitizer [11] that detects data races or an undefined behavior sanitizer [13] that detects undefined behavior at runtime. We provide a brief description of these tools in the prompt to help the agent interpret the diagnostic information. We induce the following reasoning strategies through prompting to guide the exploration of the codebase and its commit history. As shown in Figure 1, each reasoning step is followed by one or more actions. Additionally, we present the agent with a simple scratchpad at each step (shown as a markdown list of strings in the prompt), where it can add any important discoveries about the bug that should be emphasized for future steps.

Chasing control and data flow chains The *control flow* [26] of a code snippet refers to the functions that are called and the branches in it, including conditional statements, loops, gotos and even conditional compilation macros. Given a crash report and some code, the agent is asked to reason about control flow to understand how execution flows between different functions and how it leads to the crash. Similarly, *data flow* [26] refers to how values of variables get passed to different functions and how one variable is used to define another. So the agent should also reason about how data flows in the code. As a result of this reasoning, the agent may invoke a `search_definition(sym)` action (optionally also specifying the file to search in) to search for the definition of `sym` if it suspects that `sym` may have something to do with the buggy behavior and needs more information about `sym` to confirm or dispel the suspicion. It can also use other actions as suitable, e.g., `search_code(x\s*)` to look for assignments to a variable named `x`, with `\s*` indicating zero or more whitespaces.

Searching for patterns and anti-patterns Traditional software engineering literature thinks of bugs as anomalies – patterns of code that are deviant [4]. It follows that, to diagnose and understand a bug, one can find certain patterns of frequent behavior in the repository and check if a given piece of code deviates from it. *Code Researcher* reasons about which behavior is common or “normal” as well as which code snippets look anomalous. It can then perform a `search_code(regex)` action to search for these patterns and anti-patterns using regular expressions. A classic case is checking a pointer for null value after allocation. If the agent notices a missing null check for `ptr`, it can perform `search_code(if\s*\s*(ptr==NULL\s*))` to search for null checks throughout the codebase on `ptr`. Similarly, it can perform `search_code(ptr\s*=\s*.\s*alloc\s*(.\s*))` to search for all allocations to `ptr` and actually verify whether other parts of the codebase typically perform a null check or not.

Causal analysis over historical commits An interesting and challenging aspect of a codebase that has been in development for a long time, as many foundational systems codebases have, is the rich history of commits. Because of continuous development, it is likely that a new bug has some past commits that can prove helpful in understanding or solving it. Indeed, developers often reference other commits when they come up with patches. *Code Researcher* reasons about how the codebase has evolved and how that evolution is related to the crash report. It can issue a `search_commits(regex)` action to search over past commit messages and diffs. For instance, the regular expression `handle->size | crypto_fun\((` matches commits that add or remove a `handle->size` access, or a call to `crypto_fun`.

Iterative process of deep research As shown in Figure 1, in each reasoning step, *Code Researcher* is asked to decide if it has acquired sufficient context to understand and solve the crash. If yes, it moves to the next phase of synthesizing the patch (Section 3.2). Initially, the context is empty and it starts its reasoning process by analyzing the contents of the stack trace and the diagnostic information provided as input. In each step, the agent gets to evaluate the context accrued so far, as a whole, in the light of all possible reasoning strategies it can use. Based on this, it can decide which lines of exploration to extend, and issue multiple search actions simultaneously.

3.1.3 Structured context memory

We maintain a structured context memory to keep a list of (action, result) pairs for every reasoning step. Examples of actions and their results are given in Appendix A. The contents of the memory are reviewed by the agent in each reasoning step.

3.2 Synthesis and Validation phases

The contents of memory and the reasoning trace of the **Analysis** phase are passed to the **Synthesis** phase, along with the crash report. The **Analysis** phase has the flexibility to follow multiple paths of inquiry simultaneously. It can thus end up collecting information that does not turn out to be relevant, which also happens when a human does research on some topic. The **Synthesis** phase first filters the memory and discards (action, result) pairs that are deemed irrelevant to the task of fixing the crash. The agent then uses the filtered information to generate a hypothesis about the nature of the bug and a potential remedy, and the corresponding patch. Finally, in the **Validation** phase, the patch is applied to the codebase, and the codebase is compiled. The user-space program that had originally caused a crash is run. If the crash is reproduced, the patch is rejected. If not, it is accepted.

4 Experimental setup

Benchmarks We use the kBenchSyz benchmark [23] of 279 past Linux kernel crashes found by the fuzzing tool Syzkaller [12]. Each instance in the benchmark consists of (1) a reproducer file, containing the user-space program, that triggers the crash, (2) the ground-truth commit that fixed the bug, i.e., the commit after which the kernel no longer crashes on the reproducer, and (3) the crash report at the parent commit of the fix commit (we run *Code Researcher* and other competing tools at this parent commit). The benchmark also has the kernel config used to compile the crashing kernel and the list of kernel subsystems involved in each bug. We validated the 279 instances (i.e., the reproducers and the ground-truth fixes), and ruled out 9 instances for which we could not run the kernel at the parent commit, 27 for which the kernel at the parent commit did not crash, and 43 where the kernel still crashed after applying the fix. So, for our experiments, we use the remaining **200 instances** that we successfully validated. For reproducibility, we use the crash reports generated during our validation instead of the crash reports originally present in kBenchSyz. To show generalizability, we also evaluated *Code Researcher* on 10 recent crashes reported for an open-source multimedia software, FFmpeg [3] (more details in Section 5.5).

Evaluation metrics We compute Pass@ k ($P@k$) defined as $P@k = 1$, if at least one of the k candidate patches generated by the tool prevents the crash, i.e., after applying the patch, the compiled kernel does not crash on the reproducer anymore, or $P@k = 0$ otherwise. We report (1) **Crash Resolution Rate (CRR)** which is average $P@k$, (2) average **recall**, i.e., the fraction of files modified in the ground-truth commit (*the ground-truth buggy files*) in the set of files edited by the agent, averaged over the k candidate patches, and (3) the percentage of candidate patches where **All**, **Any** or **None** of the ground-truth buggy files are edited. When a tool does not produce a patch (e.g., it runs out of LLM call budget), the set of edited files is assumed to be empty. All the metrics are averaged over the 200 instances in the benchmark.

LLMs, sampling parameters, and budget We employ **GPT-4o** (v. 2024-08-06) for *Code Researcher* and for the competing tools. We also experiment with **o1** (v. 2024-12-17) in the **Synthesis** phase of *Code Researcher*. All our experiments have a context length limit of 50K tokens. In the **Analysis** phase, we use a **temperature** of 0.6 and independently sample k trajectories. For the **Synthesis** phase, we sample with increasing temperatures (0, 0.3, 0.6) until the agent produces a correctly-formatted patch, with a maximum of 3 attempts (more sampling details in Appendix B). We allow *Code Researcher* and SWE-agent a budget of at most **max calls** LLM calls per trajectory.

Baselines We evaluate *Code Researcher* in the **unassisted setting** (i.e., the ground-truth buggy files that are part of the fix commits are not divulged to the tool) and compare it against the following baselines and state-of-the-art techniques:

- (1) **o1** [28] and **GPT-4o** [27] in the **assisted setting**, i.e., we directly give the ground-truth files that are part of the fix commits, truncated to the context length limit and the crash report as input. We prompt the model to generate a hypothesis about the root cause of the crash and a patch.
- (2) **o1** and **GPT-4o** in the **stack context setting**, where we give the contents of the files mentioned in the crash report (truncated to the context length limit) as input besides the crash report.
- (3) **SWE-agent** 1.0 [40], a SOTA coding agent on the SWE-bench benchmark, in the *unassisted* setting. For fairness, we added a Linux kernel-specific example trajectory and background about the kernel to its prompts. We sample k (for Pass@ k) SWE-agent trajectories independently using a

Table 1: Crash resolution rate (CRR) for different tools on the kBenchSyz benchmark (200 bugs). LLMs used by the agentic tools are in parentheses. *CrashFixer numbers are from [24], out of 279 bugs; results wrt to the 200 bugs (Section 4) will be updated when available.

Setting	Tool	Max calls	P@k	CRR (%)
Assisted	GPT-4o	1	P@5	36.00
	o1	1	P@5	51.00
	CrashFixer (Gemini 1.5 Pro-002)*	≥ 4	P@16	49.22*
Stack context	GPT-4o	1	P@5	29.50
	o1	1	P@5	40.00
Unassisted	SWE-agent (GPT-4o)	15	P@5	31.50
	Code Researcher (GPT-4o)	15	P@5	48.00
	Code Researcher (GPT-4o + o1)	15	P@5	58.00
Unassisted + Scaled	SWE-agent (GPT-4o)	30	P@5	32.00
	Code Researcher (GPT-4o)	30	P@5	47.50
	SWE-agent (GPT-4o)	15	P@10	37.50
	Code Researcher (GPT-4o)	15	P@10	54.00

temperature of 0.6.

(4) **CrashFixer** [24], state-of-the-art agent for Linux kernel crash resolution, in the *assisted* setting, as it requires the ground-truth files to generate patches. If a patch fails to build or crashes with the ground-truth reproducer, it iteratively refines it using the respective error messages.

(5) Additionally, we consider an **unassisted + test-time scaled setting**, where we increase the test-time compute for *Code Researcher* and SWE-agent along two axes, i.e., max calls and P@k.

Additional details The **detailed prompts** and necessary **configurations** for *Code Researcher* and the baselines will be made available in an updated version. The details about **crash reproduction setup** and **implementation of the search actions** are presented in Appendix B.

5 Experimental results

In this section, we present comprehensive evaluation results that show (a) the effectiveness of *Code Researcher* in helping resolve Linux kernel crashes compared to state-of-the-art coding agents and baselines, (b) the importance of context gathered by *Code Researcher*, and (c) the impact of historical commits on *Code Researcher*’s performance. We provide additional results in the Supplementary.

5.1 RQ1: How effective are different tools at resolving Linux kernel crashes?

Our main results are presented in Table 1, organized by setting, namely, assisted, stack context, unassisted, and unassisted + test-time scaled (Section 4).

The assisted setting establishes that given the contents of the ground-truth buggy files, LLMs like GPT-4o are quite capable of resolving crashes. Using a reasoning model like o1 significantly boosts this performance (from 36% to 51%). CrashFixer achieves 49.22% CRR using 4 parallel searches with tree depth of 4 and branching factor of 1 each (resulting in P@16) where each tree node employs multiple LLM calls (exact budget is not known, we estimate it as at least 4 calls) to arrive at the patch.

However, the assumption that an oracle can tell us exactly which files need to be edited is impractical. To show the gap between the assisted (idealistic) setting and the practical unassisted setting, we present results on the simple but effective stack context setting. Here, the models are given the contents of all the files mentioned in the crash report (truncated to fit the context length limit) along with the crash report as input. This is a strong baseline because all the ground-truth buggy files are present in the crash report for 74.50% bugs in our dataset. We find that o1 achieves a CRR of 40%, which is impressive, albeit a drop of 11% in absolute points from that of the assisted setting. Importantly, *Code Researcher* (GPT-4o) in the unassisted setting with 48% CRR, (a) significantly outperforms both GPT-4o and o1 models in the stack context setting, as well as SWE-agent (GPT-4o)

in the unassisted setting, (b) even improves on GPT-4o’s CRR in the assisted setting. Furthermore, using GPT-4o for the Analysis phase and o1 for the Synthesis phase, *Code Researcher* achieves the best CRR of 58% on the dataset. These results indicate that *Code Researcher*’s Analysis produces context that is much more effective than giving file contents based on the crash report, and is even better than directly giving all the contents of the files to be edited.

Finally, we show how test-time scaling, in terms of total inference budget (max calls \times num trajectories k), impacts performance. We find that doubling the max calls budget, i.e., making the trajectories of the agents longer, has a negligible effect on the CRR. Increasing the number of trajectories sampled, on the other hand, improves SWE-agent’s CRR to 37.50% and *Code Researcher* (GPT-4o)’s CRR to 54.00%.

5.2 RQ2: How well do the files edited by the tools match those modified in developer fixes?

Table 2: Average recall and All/Any/None percentages (metrics defined in Section 4) for the two agentic tools. LLMs used by the tools are in parentheses.

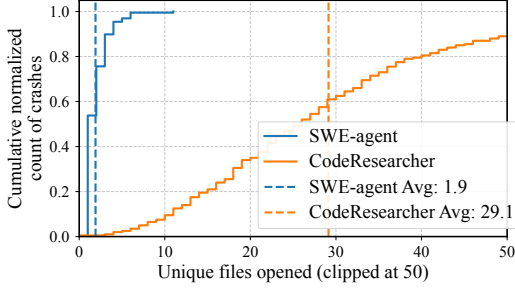
Setting	Tool	Max calls	P@k	Avg. Recall	All/Any/None (%)
Stack context	GPT-4o	1	P@5	0.45	42.5 /7.8/49.7
	o1	1	P@5	0.42	39.7/7.7/52.6
Unassisted	SWE-agent (GPT-4o)	15	P@5	0.37	35.1/5.6/59.3
	<i>Code Researcher</i> (GPT-4o)	15	P@5	0.51	48.2/7.8/44.0
	<i>Code Researcher</i> (GPT-4o + o1)	15	P@5	0.52	50.0 /7.6/42.4
Unassisted + Scaled	SWE-agent (GPT-4o)	30	P@5	0.40	37.9/6.4/55.7
	<i>Code Researcher</i> (GPT-4o)	30	P@5	0.52	49.5 /8.0/42.5
	SWE-agent (GPT-4o)	15	P@10	0.36	34.3/5.4/60.2
	<i>Code Researcher</i> (GPT-4o)	15	P@10	0.51	48.4/7.8/43.8

We now investigate whether the tools edited the same files as developers did. Since the ground-truth buggy files are already divulged to tools in the assisted setting, we focus on the stack context and unassisted settings. The results are in Table 2. We note that *Code Researcher* variants (both GPT-4o and GPT-4o + o1) have significantly higher average recall than GPT-4o and o1 in the stack context setting, as well as SWE-agent in the unassisted setting. In addition, *Code Researcher* (GPT-4o) edits all the ground-truth buggy files in 48.2% of the candidate patches, and at least one in an additional 7.8% of the patches, totaling 56%. These metrics are significantly better than that of all other tools in the stack context and unassisted settings. Finally, for *Code Researcher* and SWE-agent, scaling test-time compute (via increasing max calls or P@ k) preserves the degree of overlap between the edited files and the ground-truth buggy files.

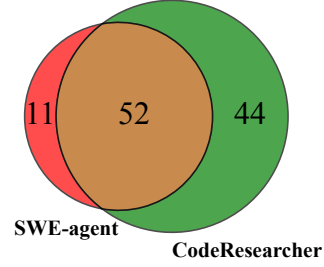
5.3 RQ3: How effective is context gathering for resolving Linux kernel crashes?

Recall from Table 1 that *Code Researcher* (GPT-4o) significantly outperforms GPT-4o in the assisted and stack context settings. This points to the usefulness of the context gathered by *Code Researcher*. On the other hand, SWE-agent (GPT-4o) in Table 1 also gathers context, but its performance is not on par. Below, we investigate this discrepancy along multiple axes:

1) *Code Researcher* gathers much more context than SWE-agent: Figure 2 (a) shows the distribution of the number of unique files read by *Code Researcher* and SWE-Agent (both using GPT-4o, P@5, 15 max calls) for each bug (i.e., all unique files read in the 5 trajectories). *Code Researcher* really performs deep research over the codebase, reading 29.13 unique files across 5 top-level directories on average for each bug. In stark contrast, SWE-agent reads only 1.91 files on average for each bug. When averaged by trajectory, *Code Researcher* explores 10 unique files compared to only 1.33 files explored by SWE-agent. One reason for this huge difference is that existing coding agents have been designed with respect to benchmarks like SWE-bench [17]. Tasks in these benchmarks do not require deep context gathering and reasoning, unlike tasks over complex systems codebases where deep exploration and reasoning is crucial (as is evident from our results).



(a) Files explored for each crash (summed over 5 trajectories).



(b) Overlap of crashes resolved.

Figure 2: SWE-agent vs *Code Researcher* (both at GPT-4o, P@5, 15 max calls)

2) *Code Researcher* has better overlap with developer-referenced context: We use LLM-as-judge to analyze the context gathered by *Code Researcher* (GPT-4o) and SWE-agent; and determine the overlap of the contexts with the context mentioned by the developer in the fix commit message (details in Appendix D). This context overlap is 54.18% (over candidate patches) for SWE-agent compared to 63.7% for *Code Researcher*. This suggests that *Code Researcher* does a much better job of identifying relevant context that the developer explicitly relied on when making the fix.

3) *Code Researcher* has a significantly higher CRR than SWE-agent when both the tools correctly identify all the ground-truth modified files: To isolate the impact of the gathered context, we consider the subset of 90 bugs, where both *Code Researcher* (GPT-4o) and SWE-agent edited *all* the ground-truth files in *at least one candidate patch* generated by each tool. Now, if both the tools are editing the ground-truth files and using the same model for patch generation (GPT-4o), we can attribute the success (or failure) of the tools on these bugs to the context gathered. We find that *Code Researcher* resolves $55/90 = 61.10\%$ of crashes in this subset, while SWE-agent resolves only $34/90 = 37.78\%$ (note that we discount crash-resolving patches from each tool that do not edit *all* the ground-truth files). This suggests that *Code Researcher*’s context is more relevant to the task of crash resolution than that of SWE-agent.

5.4 RQ4: How important are historical commits for resolving crashes in the Linux kernel?

Table 3: Effectiveness of access to historical commits.

Tool	Max Calls	P@k	CRR(%)	Avg. Recall	All/Any/None(%)
<i>Code Researcher</i> (GPT-4o)	15	P@5	48.00	0.51	48.2/7.8/44.0
w/o <code>search_commits</code> ¹	15	P@5	38.00	0.33	32.6/2.4/65.0

¹ We do this ablation only on the 96 bugs resolved by *Code Researcher* (GPT-4o, Pass@5, 15 max calls).

To the best of our knowledge, *Code Researcher* is the first agent to explicitly leverage the rich development history of codebases. To evaluate the importance of historical commits, we perform an ablation study, where we run *Code Researcher* without the `search_commits` action on the set of 96 bugs that were successfully resolved by *Code Researcher* (GPT-4o, Pass@5, 15 max calls). The results are in the second row of Table 3. We observe that removing the `search_commits` action leads to a 10% drop in the crash resolution rate. More importantly, both recall and the model’s ability to identify all or any of the ground-truth modified files decrease substantially. This highlights that the `search_commits` action plays a crucial role in guiding the agent toward relevant context and in correctly localizing the files to be fixed. Notably, for the example in Appendix C.1, we also observe that *Code Researcher* navigates to the *same* buggy commit that the developer identified as the source commit that originally introduced the bug being repaired.

5.5 RQ5: Does *Code Researcher* generalize to other systems codebases?

To demonstrate that *Code Researcher* generalizes with a little effort to other codebases, we experiment with the task of crash resolution in the FFmpeg [3] codebase. FFmpeg is a leading open-source multimedia framework, that supports ability to decode, encode, transcode, mux, demux, stream, filter and play all existing media formats. Since it needs to handle a wide range of formats, from very old to the cutting edge, low-level data manipulation is common in the codebase. FFmpeg has $\sim 4.8K$ files and $\sim 1.46M$ lines of code, primarily comprising C / C++ and also some handwritten assembly code for performance.

Dataset We use vulnerabilities discovered by the OSS-Fuzz service [6] that runs fuzzers on various open source projects and creates alerts for the bugs detected. We focus on security issues, which are assigned the top priority by OSS-Fuzz. These include heap-based buffer overflows, stack-based buffer overflows, use-after-frees, etc. We build a small dataset of 10 FFmpeg crashes, taking the 11 most recent crashes (as of May 14, 2025) that have been verified as fixed and skipping 1 that we could not validate.² We use the instructions recommended by OSS-Fuzz for building FFmpeg and testing whether a crash reproduces.³ The dataset contains the commit at which OSS-Fuzz found the crash, a reproducer file that triggered the crash, and the crash report that we generated by reproducing the crash (the crash reports found by OSS-Fuzz are not publicly visible). The dataset and the detailed prompts will be made available in an updated version.

Results To run *Code Researcher* on these crashes, we keep the same core prompts, adding a one-paragraph preamble about FFmpeg and replacing the few-shot examples for the Linux kernel with corresponding ones for FFmpeg.

Code Researcher, in the unassisted setting using GPT-4o for the Analysis phase and o1 for the Synthesis phase, with a max calls budget of 15, **resolves 7 out of the 10 crashes** in our dataset at Pass@1, i.e., we sample only one patch per crash. *Code Researcher* achieves an **average recall of 0.78, edits all the ground-truth modified files in 7 crashes and none of the ground-truth modified files in 2 crashes**.⁴ While this suggests that FFmpeg crashes are typically not as complex to resolve as Linux kernel crashes, our results show that *Code Researcher*’s techniques generalize easily and effectively to other systems codebases.

5.6 Qualitative evaluation

Even if a patch prevents the crash, that does not guarantee that it actually fixes the underlying issue. Any form of test-based evaluation (as is done even in other benchmarks like SWE-bench [17]) has the limitation that it cannot ensure preservation of functionality that the tests do not cover. Testing the full functionality of the kernel easily and reliably is a hard open research problem. While perusing the crash-preventing patches, we came across the following types of patches:

- (1) **Accurate** These patches correctly identify and fix the root cause of the crash in a manner that closely resembles the developer solution. For example, for a crash in the JFS filesystem (Listing 1, Appendix E), *Code Researcher* generated a patch equivalent to the developer’s solution.
- (2) **Overspecialized** These patches successfully prevent the crash but may be overspecialized. As shown in (Listing 2, Appendix E) for a crash in the Bluetooth HCI H5 driver, *Code Researcher* correctly identified that `hu->serdev` could be NULL. However, while the developer simply added a NULL check around the existing code, *Code Researcher*’s patch included additional error handling with a diagnostic message and explicit return values.
- (3) **Incomplete** These patches correctly identify the problem area and approach, but may not be complete. They provide significant debugging insights and could accelerate the path to a proper fix. For example, in the QRTR networking subsystem (Listing 3, Appendix E), *Code Researcher* correctly identifies a concurrency bug involving radix tree traversal without RCU protection and inserts the appropriate synchronization in one affected function, but not in others.
- (4) **Incorrect** These patches fail to address the root cause or may introduce new issues. In the QRTR

²https://issues.oss-fuzz.com/issues?q=project:ffmpeg%20type:vulnerability%20status:verified&s=modified_time:desc&p=1

³<https://google.github.io/oss-fuzz/advanced-topics/reproducing/>

⁴For 1 crash where we couldn’t find the ground-truth fix (and hence the ground-truth modified files), we exclude it from the calculation of recall and All/Any/None numbers.

networking subsystem (Listing 4, Appendix E), *Code Researcher* failed to infer the root cause of the issue which is an integer truncation problem when handling large u32 port numbers. *Code Researcher* instead added checks that reject ports with `port < 0`.

6 Conclusions, limitations, and future work

In this work, we take a concrete step forward in the fast emerging field of LLM-based coding agents, extending them to deep research scenarios arising in resolving complex issues in large systems codebases. We show that (a) we can exploit the rich history of development in the codebases (commits), and (b) design effective deep exploration strategies for gathering rich context often needed to root-cause and patch code crashes. We establish state-of-the-art results on the latest and challenging benchmark of Linux kernel crashes.

Our work currently targets the crash resolution problem, but there are other equally important problems faced by systems software such as slow response times, excessive resource usage and flakiness. It remains to be seen if our deep research strategy could generalize to these scenarios. Deep research for code is a new subfield of agentic AI. We intend to explore novel usecases and strategies beyond the ones presented in the paper.

References

- [1] Anthropic. Claude takes research to new places, 2025. URL <https://www.anthropic.com/news/research>.
- [2] ccache. ccache. URL <https://github.com/ccache/ccache>.
- [3] F. community. Ffmpeg, 2025. URL <https://ffmpeg.org/>.
- [4] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. *SIGOPS Oper. Syst. Rev.*, 35(5):57–72, Oct. 2001. ISSN 0163-5980. doi: 10.1145/502059.502041. URL <https://doi.org/10.1145/502059.502041>.
- [5] A. Godbole, N. Monath, S. Kim, A. S. Rawat, A. McCallum, and M. Zaheer. Analysis of plan-based retrieval for grounded text generation. *arXiv preprint arXiv:2408.10490*, 2024.
- [6] Google. Oss-fuzz | documentation for oss-fuzz. URL <https://google.github.io/oss-fuzz/>.
- [7] Google. Gemini deep research, 2025. URL <https://gemini.google/overview/deep-research>.
- [8] Google. Generative ai | documentation | long context. <https://cloud.google.com/vertex-ai/generative-ai/docs/long-context>, 2025. Accessed: 2025-03-13.
- [9] Google. Generative ai | documentation | gemini 2.5 pro. <https://cloud.google.com/vertex-ai/generative-ai/docs/models/gemini/2-5-pro>, 2025. Accessed: 2025-05-16.
- [10] Google. Kasan, 2025. URL <https://github.com/google/kernel-sanitizers/blob/master/KASAN.md>.
- [11] Google. Kcsan, 2025. URL <https://github.com/google/kernel-sanitizers/blob/master/KCSAN.md>.
- [12] Google. Syzkaller, 2025. URL <https://github.com/google/syzkaller/>.
- [13] Google. Ubsan, 2025. URL <https://github.com/google/kernel-sanitizers/blob/master/UBSAN.md>.
- [14] J. Gottweis, W.-H. Weng, A. Daryin, T. Tu, A. Palepu, P. Sirkovic, A. Myaskovsky, F. Weissenberger, K. Rong, R. Tanno, et al. Towards an ai co-scientist. *arXiv preprint arXiv:2502.18864*, 2025.

- [15] D. Guo, C. Xu, N. Duan, J. Yin, and J. McAuley. Longcoder: A long-range pre-trained language model for code completion. In *International Conference on Machine Learning*, pages 12098–12107. PMLR, 2023.
- [16] N. Jain, K. Han, A. Gu, W.-D. Li, F. Yan, T. Zhang, S. Wang, A. Solar-Lezama, K. Sen, and I. Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.
- [17] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. R. Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL <https://openreview.net/forum?id=VTF8yNQM66>.
- [18] H. Kagdi, M. L. Collard, and J. I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of software maintenance and evolution: Research and practice*, 19(2):77–131, 2007.
- [19] T. Li, G. Zhang, Q. D. Do, X. Yue, and W. Chen. Long-context llms struggle with long in-context learning. *arXiv preprint arXiv:2404.02060*, 2024.
- [20] X. Li, J. Jin, G. Dong, H. Qian, Y. Zhu, Y. Wu, J.-R. Wen, and Z. Dou. Webthinker: Empowering large reasoning models with deep research capability. *arXiv preprint arXiv:2504.21776*, 2025.
- [21] Linus Torvalds. Linux, 1991. URL <https://github.com/torvalds/linux>.
- [22] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024.
- [23] A. Mathai, C. Huang, P. Maniatis, A. Nogikh, F. Ivančić, J. Yang, and B. Ray. Kgym: A platform and dataset to benchmark large language models on linux kernel crash resolution. *Advances in Neural Information Processing Systems*, 37:78053–78078, 2024.
- [24] A. Mathai, C. Huang, S. Ma, J. Kim, H. Mitchell, A. Nogikh, P. Maniatis, F. Ivančić, J. Yang, and B. Ray. Crashfixer: A crash resolution agent for the linux kernel. *arXiv preprint arXiv:2504.20412*, 2025.
- [25] Microsoft. Introducing researcher and analyst in microsoft 365 copilot, 2025. URL <https://www.microsoft.com/en-us/microsoft-365/blog/2025/03/25/introducing-researcher-and-analyst-in-microsoft-365-copilot/>.
- [26] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of program analysis*. springer, 2015.
- [27] OpenAI. Introducing openai o1-preview, 2024. URL <https://openai.com/index/hello-gpt-4o/>.
- [28] OpenAI. Introducing openai o1-preview, 2024. URL <https://openai.com/index/introducing-openai-o1-preview/>.
- [29] OpenAI. Openai deep research integration with github, 2025. URL <https://help.openai.com/en/articles/11145903-connecting-github-to-chatgpt-deep-research>.
- [30] OpenAI. Introducing deep research, 2025. URL <https://openai.com/index/introducing-deep-research/>.
- [31] Perplexity. Introducing perplexity deep research, 2025. URL <https://www.perplexity.ai/de/hub/blog/introducing-perplexity-deep-research>.
- [32] Y. Shao, Y. Jiang, T. A. Kanell, P. Xu, O. Khattab, and M. S. Lam. Assisting in writing wikipedia-like articles from scratch with large language models. *arXiv preprint arXiv:2402.14207*, 2024.
- [33] G. Team, P. Georgiev, V. I. Lei, R. Burnell, L. Bai, A. Gulati, G. Tanzer, D. Vincent, Z. Pan, S. Wang, et al. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*, 2024.

- [34] universal-ctags. ctags. URL <https://github.com/universal-ctags/ctags>.
- [35] N. Wadhwa, A. Sonwane, D. Arora, A. Mehrotra, S. Utpala, R. B. Bairi, A. Kanade, and N. Natarajan. MASAI: Modular architecture for software-engineering AI agents. In *NeurIPS 2024 Workshop on Open-World Agents*, 2024. URL <https://openreview.net/forum?id=NSINt81LYB>.
- [36] X. Wang, B. Li, Y. Song, F. F. Xu, X. Tang, M. Zhuge, J. Pan, Y. Song, B. Li, J. Singh, H. H. Tran, F. Li, R. Ma, M. Zheng, B. Qian, Y. Shao, N. Muennighoff, Y. Zhang, B. Hui, J. Lin, R. Brennan, H. Peng, H. Ji, and G. Neubig. Openhands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*, 2025. URL <https://openreview.net/forum?id=0Jd3ayDDoF>.
- [37] J. Wu, J. Zhu, and Y. Liu. Agentic reasoning: Reasoning llms with tools for the deep research. *arXiv preprint arXiv:2502.04644*, 2025.
- [38] W. Wu, S. Huang, Y. Jiang, P. Xie, F. Huang, and H. Zhao. Unfolding the headline: Iterative self-questioning for news retrieval and timeline summarization. *arXiv preprint arXiv:2501.00888*, 2025.
- [39] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang. Agentless: Demystifying llm-based software engineering agents, 2024. URL <https://arxiv.org/abs/2407.01489>.
- [40] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press. Swe-agent: Agent-computer interfaces enable automated software engineering, 2024. URL <https://arxiv.org/abs/2405.15793>.
- [41] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.
- [42] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, page 1592–1604, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400706127. doi: 10.1145/3650212.3680384. URL <https://doi.org/10.1145/3650212.3680384>.

A Example of agent trajectory

In this section, we show an example **Analysis** trajectory generated by *Code Researcher* while solving a kernel crash from the dataset⁵. The truncated crash report that initiates the investigation of the example is shown in Figure 3.

The trajectory begins in Figure 4. *Code Researcher* identifies a warning in `smsusb_term_device` related to `__flush_work` and forms an initial hypothesis about workqueue synchronization issues. The agent then explores the codebase by examining the `__flush_work` implementation and analyzing the `smsusb_stop_streaming` function’s synchronous URB cancellation pattern in Figure 5, while also searching for relevant commit history to identify potential race conditions.

In Figures 6–8, *Code Researcher* traces prior fixes for related bugs in `smsusb_term_device`, inspects synchronization behavior in `cancel_work_sync` and `__cancel_work_timer`. It uncovers a critical misstep where work item initialization via `INIT_WORK` in `smsusb_onresponse` can be bypassed, resulting in `NULL work->func` pointers that explain the observed warning.

Finally, in Figure 9, *Code Researcher* confirms the root cause: though `smsusb_onresponse` correctly initializes work structures with `INIT_WORK`, it is not sufficient. If `smsusb_stop_streaming` is called before any URB completion occurs, the system attempts to cancel uninitialized work items, triggering warnings in `__flush_work` when it encounters `NULL` function pointers.

```
smsusb:smsusb_probe: board id=7, interface number 0
-----[ cut here ]-----
WARNING:
CPU: 0 PID: 9 at kernel/workqueue.c:3182 __flush_work+0x95c/0xbf0 kernel/workqueue.c:3182
Call Trace:
<TASK>
__cancel_work_timer+0x3e6/0x580 kernel/workqueue.c:3276
smsusb_stop_streaming drivers/media/usb/siano/smsusb.c:182 [inline]
smsusb_term_device+0xef/0x300 drivers/media/usb/siano/smsusb.c:344
smsusb_init_device+0xb60/0xd10 drivers/media/usb/siano/smsusb.c:419
smsusb_probe+0x5b1/0x10c0 drivers/media/usb/siano/smsusb.c:567
usb_probe_interface+0x307/0x930 drivers/usb/core/driver.c:396
...
worker_thread+0x687/0x1110 kernel/workqueue.c:2552
kthread+0x33a/0x430 kernel/kthread.c:379
ret_from_fork+0x1f/0x30 arch/x86/entry/entry_64.S:308
</TASK>
```

Figure 3: Crash report of the kernel crash example discussed in Appendix A.

⁵Bug in syzkaller: <https://syzkaller.appspot.com/bug?id=35dd38010e383ee37d27b39f6b50eb25b4c92dd2>

You are analyzing a kernel crash report. Your task is to explore the repository using available tools and gather enough context to completely understand all possible root causes of the crash and how to fix it.

STEP 0 of 9

BUG TITLE : WARNING in `smsusb_term_device`

CRASH REPORT:
{crash_report}

AVAILABLE ACTIONS (write one action per line with no formatting inside <actions> tags):

<actions>

`close_definition(filePath, symbolName, startLine)`

Close a definition that is not relevant to the bug (e.g. debug functions like `dump_stack`)

`search_definition(filePath, symbolName)`

Open the definition of a specified symbol, which can be a function name, a struct name, a constant, a union or a macro

`search_definition(symbolName)`

Open the definition of a symbol when you don't know its file path. Use this to search for the definition of symbols whose file you don't know.

`search_code("regex pattern")`

Search through all tracked files in the repository using `git grep` with regex pattern. Shows 2 context lines before and after each match, limited to 5 matches total.

done End exploration because you have gathered enough context to understand the bug and propose a fix. You must only take this action if you are satisfied with the justification you provided in the thoughts section. If unsure, continue exploring. You should also write your justification after this action inside <justification></justification> tags. (followed by justification in <justification> tags)

</actions>

You should examine the evidence and write your thoughts about the current state... Focus on gathering all the relevant context and understanding the bug step-by-step, through careful reasoning steps.

The crash report indicates a warning in the `'smsusb_term_device'` function. The warning is triggered during a call to `'__flush_work'` in `'kernel/workqueue.c'`.

...

The issue appears to be related to the handling of work items, possibly involving improper synchronization or misuse of workqueues. To understand the root cause, I need to examine the `'smsusb_term_device'` function and the `'smsusb_stop_streaming'` function to see how work items are managed and flushed. Additionally, I should explore how `'__flush_work'` is used and what could lead to the warning in this context.

<actions>

`search_definition("drivers/media/usb/siano/smsusb.c", "smsusb_term_device")`

`search_definition("drivers/media/usb/siano/smsusb.c", "smsusb_stop_streaming")`

`search_code("__flush_work")`

</actions>



The results of actions are added to memory

Figure 4: Code Researcher trajectory while solving the crash described in Figure 3: Step 0 of 9

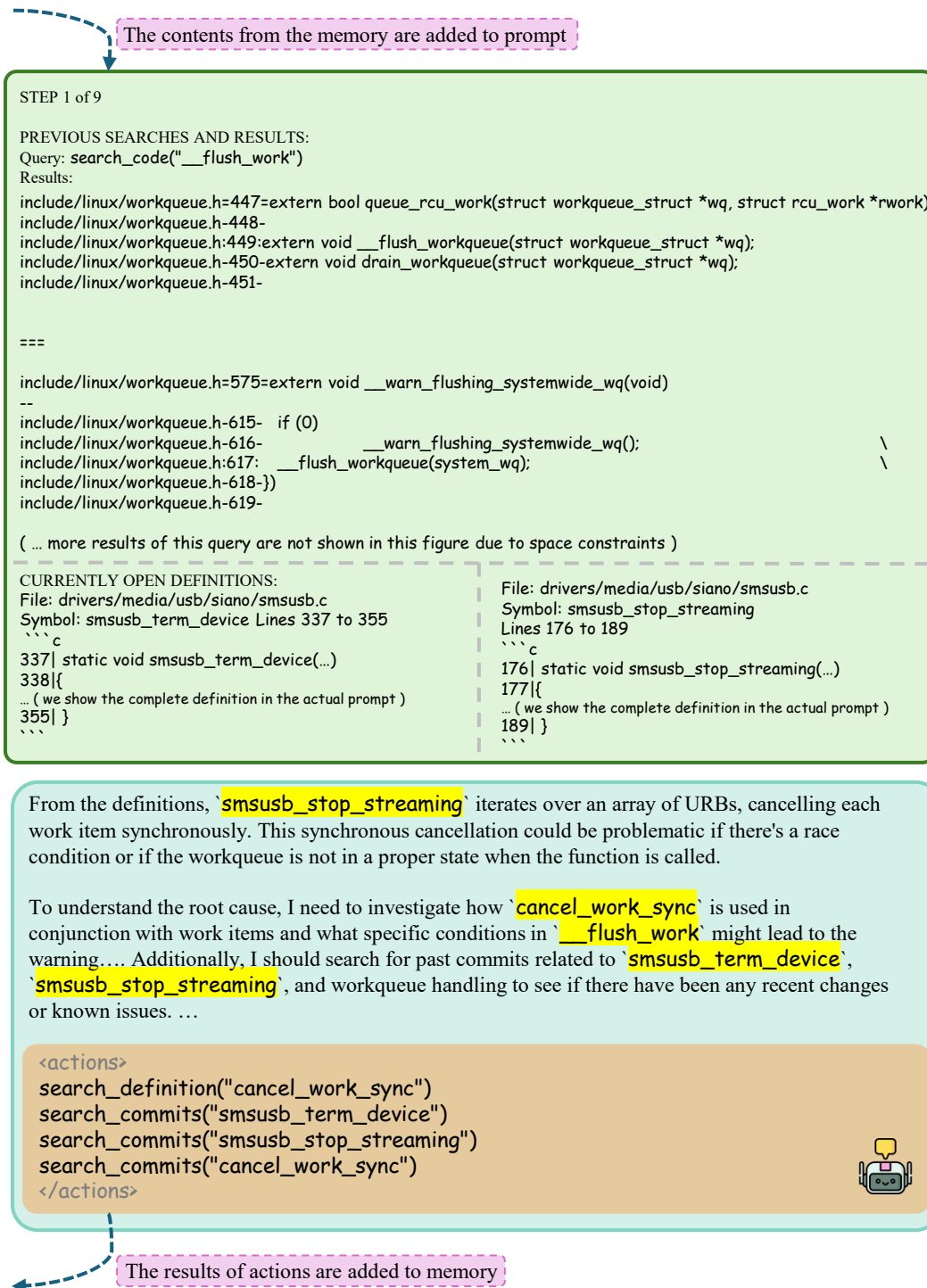


Figure 5: Code Researcher trajectory while solving the crash described in Figure 3: Step 1 of 9

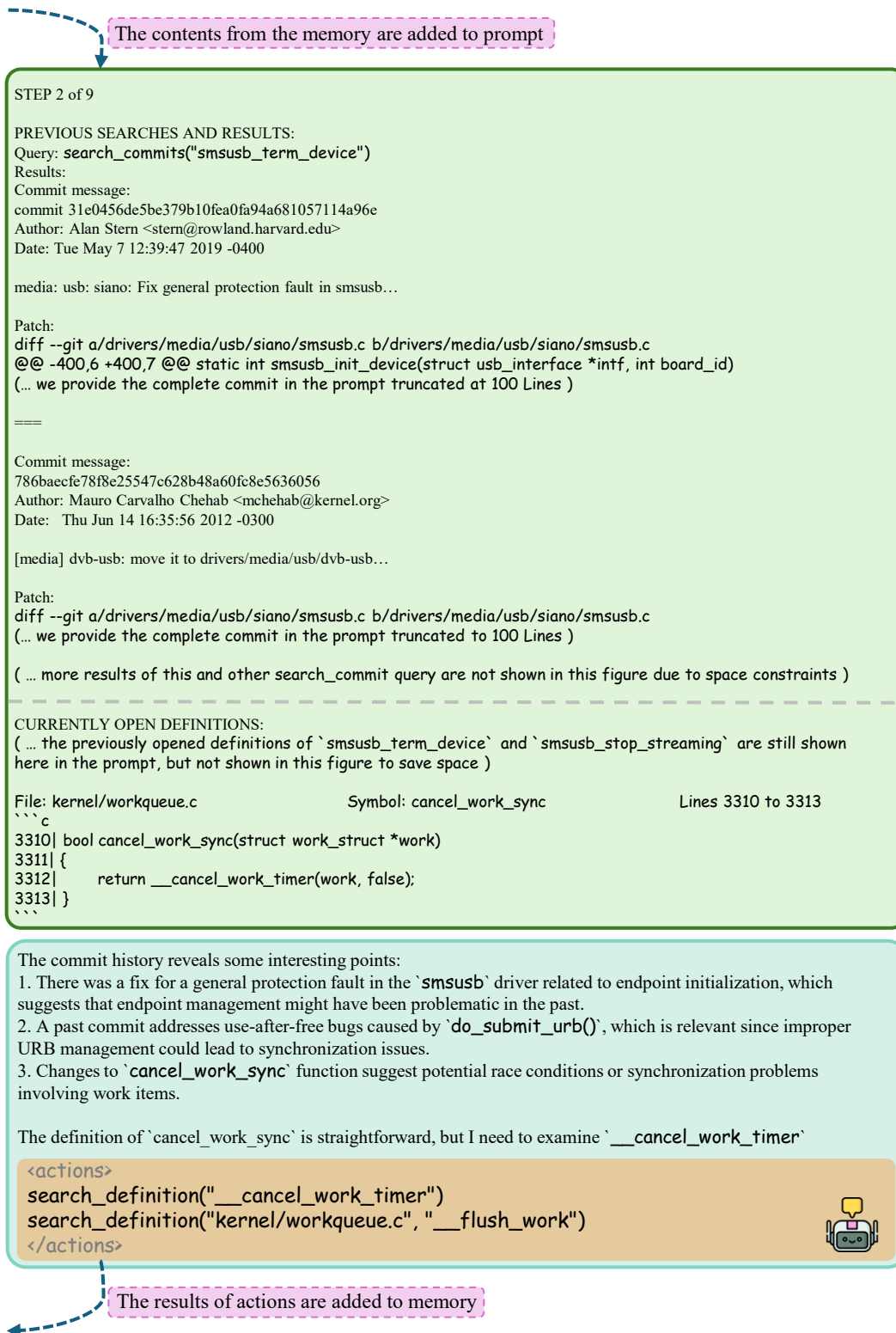


Figure 6: *Code Researcher* trajectory while solving the crash described in Figure 3: Step 2 of 9

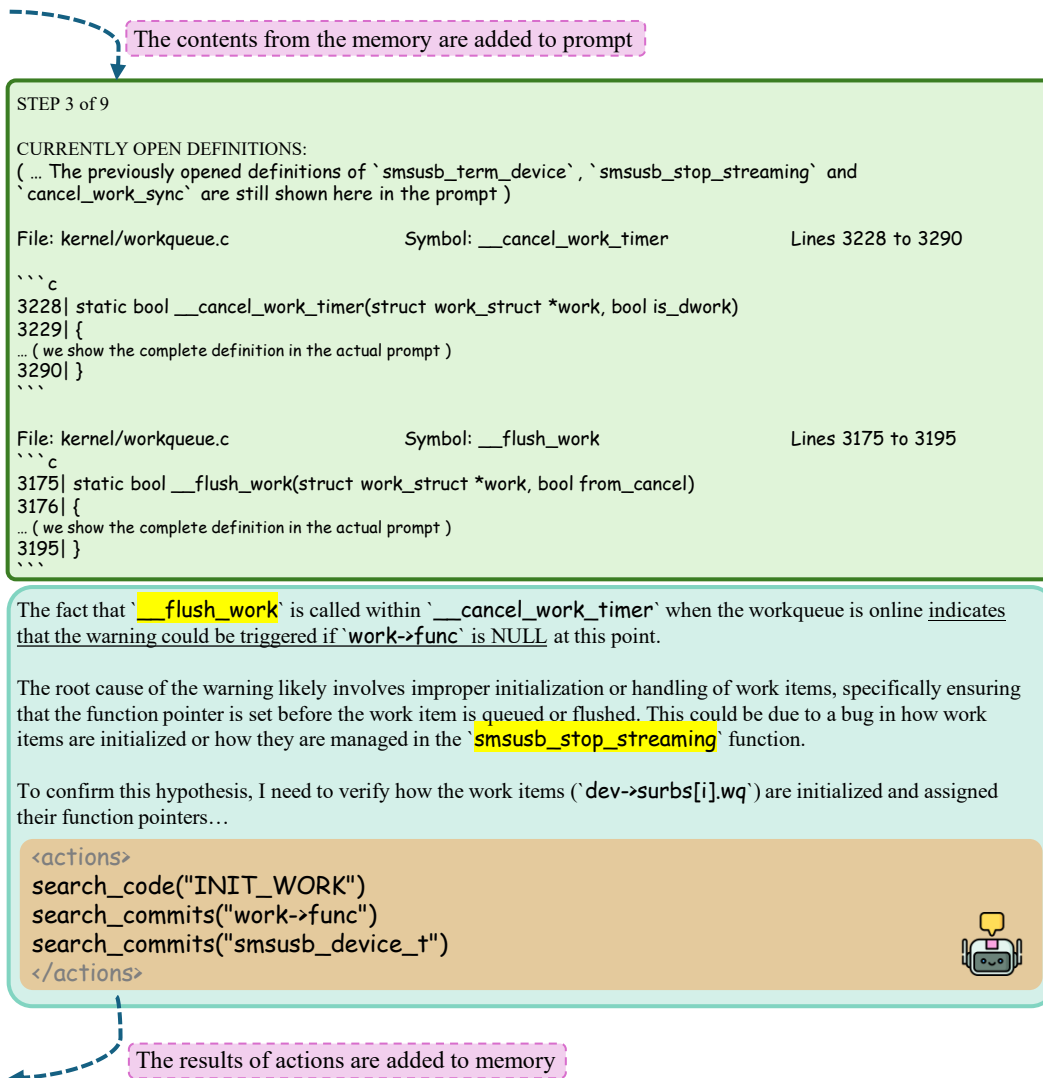


Figure 7: Code Researcher trajectory while solving the crash described in Figure 3: Step 3 of 9

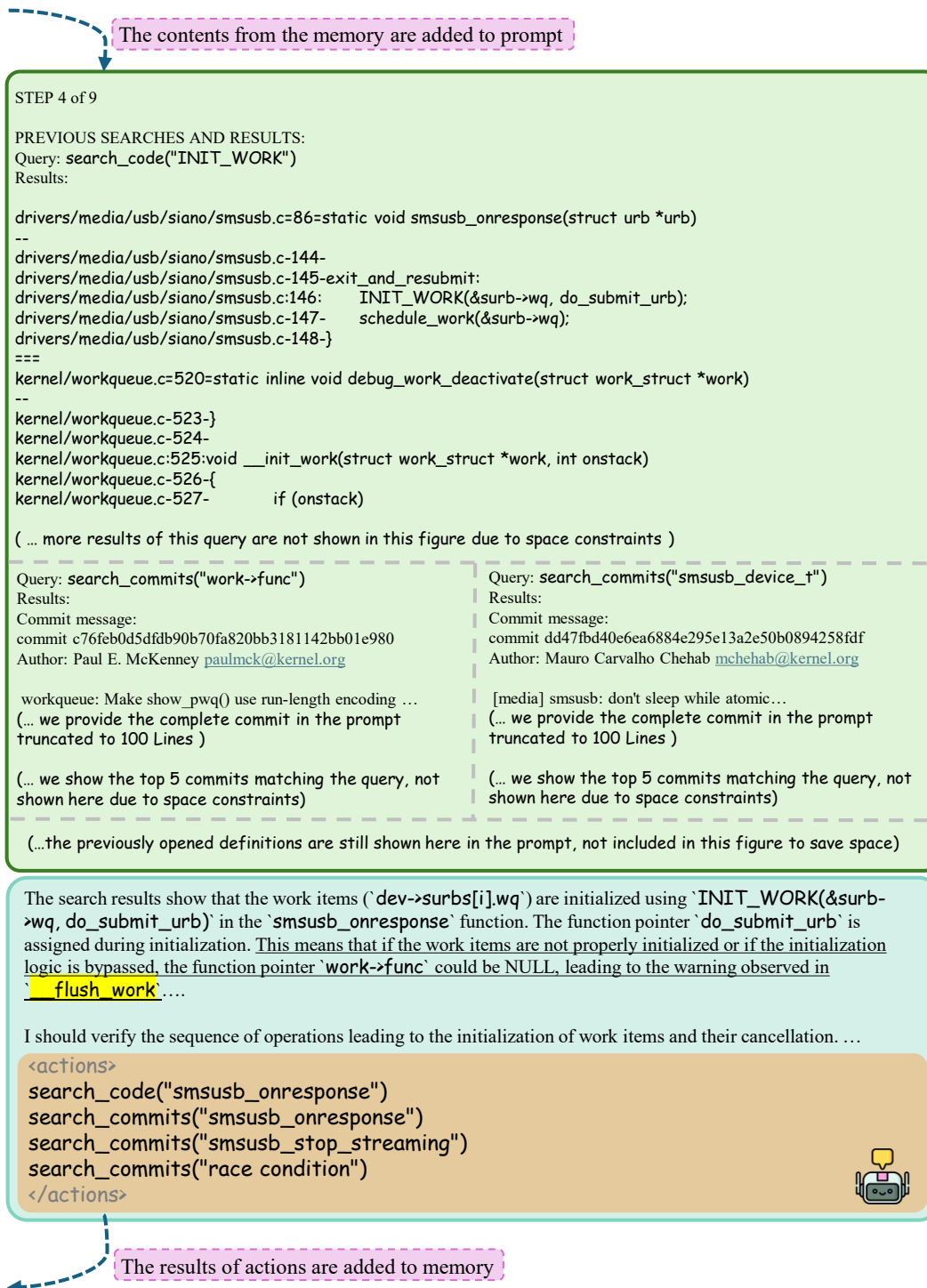


Figure 8: Code Researcher trajectory while solving the crash described in Figure 3: Step 4 of 9

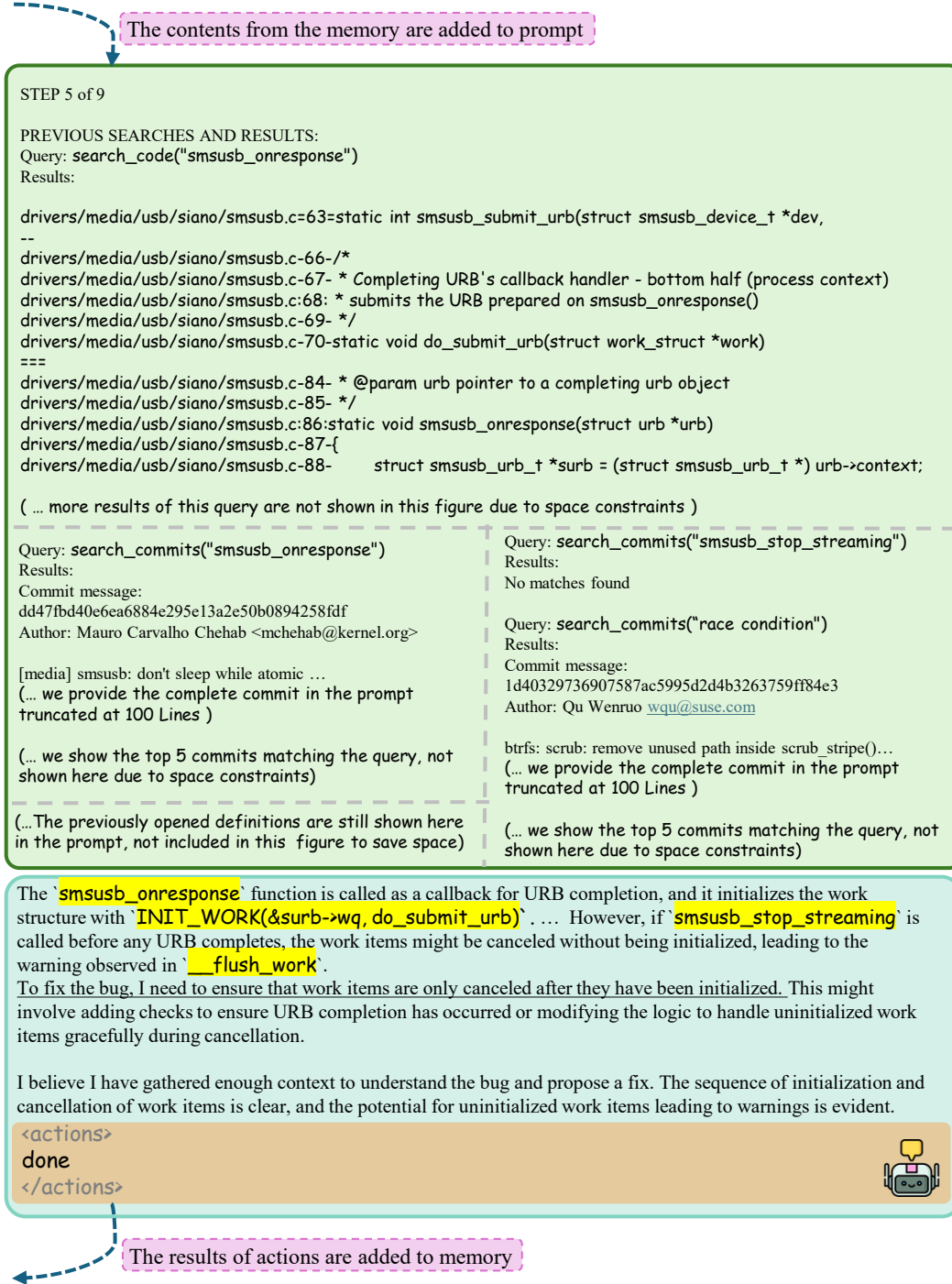


Figure 9: Code Researcher trajectory while solving the crash described in Figure 3: Step 5 of 9 Analysis phase

B Experimental setup: additional details

Dataset details We use the kBenchSyz dataset containing 279 instances from Mathai et al. [23]. The dataset is publicly available at <https://github.com/Alex-Mathai-98/kGym-Kernel-Playground> and is under an MIT License. The subset of 200 instances that we were able to reproduce will be made available in an updated version.

Sampling details In the Synthesis phase, we ask the agent to generate a hypothesis and patch in the following format. It has to write the hypothesis inside `<hypothesis>` tags and the patch inside `<patch>` tags. The content inside the `<patch>` tags is a list of `<symbol>` tags covering all the symbols whose definitions the agent wants to change in its patch. With each tag, the agent has to provide `file`, `name` and `start line` attributes and inside each tag, it has to rewrite the complete definition of the symbol (after making the desired changes). We use successively higher temperatures (0, 0.3, 0.6) until the agent gives a correctly formatted patch. For o1, since its API does not support a temperature parameter, we sample the desired number of patches by setting the `n` parameter (number of completions) in the OpenAI Chat Completions API.

Crash reproduction setup Our setup for building the Linux kernel and running it on reproducer files is built on top of the *kGym* platform (MIT Licensed, publicly available at <https://github.com/Alex-Mathai-98/kGym-Kernel-Gym>) [23] and has a couple of major modifications. First, while *kGym* runs only on the Google Cloud platform, our setup can run locally on any machine and uses cloud storage for preserving compiled kernels, crash reports, etc. Second, we use *ccache* [2] for caching build files generated during kernel compilation and our own logic for caching git checkouts.

kGym has a distributed setup featuring five workers - *kBuilder*, *kReproducer*, *kScheduler*, *kDashboard* and *kmq*. (1) *kBuilder* takes as input a source commit, a kernel config, and (optionally) a patch. It checks out the kernel at the source commit, applies the patch, compiles the kernel and uploads the build artifacts (kernel image, vmlinux binary, etc.) to cloud storage. (2) *kReproducer* takes as input the build artifacts and a reproducer file and runs the kernel on the reproducer while monitoring for crashes. To handle non-deterministic bugs, we launch 4 VMs in parallel, each of which runs the reproducer. Each VM further runs multiple processes where system calls can execute in parallel so concurrency bugs can also be reproduced. If any of these VMs crash within 10 minutes or if *kReproducer* loses connection to the VMs, we say that the kernel crashes on the reproducer. It then uploads the crash reports to cloud storage. (3) *kScheduler* serves an API where we can send reproduction jobs with the source commit, config, reproducer and (optionally) patch. It communicates with *kBuilder* and *kReproducer* through the message queue *kmq* and orchestrates the overall flow of build with *kBuilder* followed by reproduction with *kReproducer*. (4) Finally, *kDashboard* displays each job’s logs and results in a web UI.

Compute resources We setup 10 replicas of the distributed setup (containing 5 workers) described above. Each machine was equipped with an AMD EPYC 7V13 Processor running at 2.50 GHz, had 24 cores and 220 GB RAM. For one evaluation run on our dataset of 200 instances for any tool in the P@5 setting (i.e., for evaluating 1000 patches on whether they prevent a crash or not), we divided the instances among the 10 replicas, and the overall time ranged from 10 to 15 hours.

SWE-agent details We use SWE-agent [40] as one of our baselines. The codebase is publicly available at <https://github.com/SWE-agent/SWE-agent/tree/main> and is under the MIT License. We use version 1.0.1 of SWE-agent, and add a Linux kernel-specific example trajectory and background about the Linux kernel to its prompts. We will provide the complete configuration file (including all prompts and the example trajectory) in an updated version.

Implementation of the search actions We implement the `search_definition(sym)` action using the `ctags` [34] tool to generate (and read) an index file of language objects found in source files for programming languages. The index file is constructed once at the start of *Code Researcher*’s run, usually taking a few minutes for the Linux kernel codebase, and is used throughout the Analysis trajectory. Whenever we show a symbol definition in the prompt, for each line of code that is mentioned in the crash report, we additionally add an annotation (as a C-style comment at the end of the line) saying that this line is important. For `search_code(regex)`, we use the `git grep -E` command to search over all the tracked files in the codebase and show 2 lines of context before and after each matching line. Finally, for `search_commits(regex)`, we use the `git log -E -G` and `git log -E -grep` commands to search over historical commits matching in the code changes and commit messages, respectively. The message and patch of each relevant commit are returned as output, truncated to a maximum of 100 lines. Each action can return a maximum of 5 results. To make these searches over an extremely large repository faster, we progressively search over the files of the symbol definitions present in context memory, then those mentioned in the crash report, then

those in the kernel subsystems of the bug, and finally all the files in the codebase. This prioritization strategy allows us to use a timeout of 60 seconds for the `git log` commands (which usually take the longest time) while still getting relevant results in a large number of cases.

C Importance of causal analysis over historical commits

C.1 Illustrative example

Figures 10–12: Illustration of *Code Researcher* analyzing and repairing a real-world memory leak bug⁶ from the kBenchSyz dataset (the complete trajectory of *Code Researcher* is truncated and only the relevant parts are shown due to space constraints). Figure 10 shows the developer's original commit, including the fix and a "Fixes:" tag that references the buggy commit where the issue originated: commit `6679f4c5e5a6`—highlighted in yellow. This section is shown in the orange box. The developer's fix is available at the following link⁷. Figure 11 displays a subset of actions taken by *Code Researcher* in the Analysis phase, specifically several `search_commits` steps (green box), which reveal how the agent retrieves both the buggy commit and other related commits that involve memory management. These retrieved historical commits provide guidance in the Synthesis phase. Figure 12 presents the patch and analysis synthesized by *Code Researcher* (blue box) during the Synthesis phase, which correctly identifies the missing deallocation and inserts the appropriate `kfree(bt_const_extended)` call. Notably, *Code Researcher* successfully navigates to the same buggy commit identified by the developer's "Fixes:" tag, demonstrating its ability to infer causality and leverage prior commits to localize and fix bugs.

```

commit 50d34a0
author Marc Kleine-Budde
Fixes 6679f4c5e5a6 ("can: gs_usb: add extended bt_const feature")

...The information must be read with an USB control message. The memory for this message is allocated but
not free(fed) (in the non error case). This patch adds the missing free.

<patch>
static struct gs_can *gs_make_candev(unsigned int channel,
dev->data_bt_const.brp_inc = le32_to_cpu(bt_const_extended->dbrp_inc);
dev->can.data_bittiming_const = &dev->data_bt_const;
+
+ kfree(bt_const_extended);
+
}

SET_NETDEV_DEV(netdev, &intf->dev);
</patch>

```


 **Developer commit message and patch**

Figure 10: Developer commit message and patch.

⁶Bug in Syzbot dashboard: <https://syzkaller.appspot.com/bug?id=92a742e993c8b9e769f8502a0497c88c0afa78af>.

⁷Developer's fix commit: <https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=50d34a0d151dc7abdbbec781bd7f09f2b3cbf01a>.



Actions taken by Code Researcher


`search_commits("bt_const_extended|memory leak")`
 commit 6679f4c5e5a6
 Author: Peter Fink

can: gs_usb: add extended bt_const feature

... phase to reach maximum performance at higher speeds. ...

```

<patch>
static struct gs_can *gs_make_candev(unsigned int channel,
    struct net_device *netdev;
    int rc;
    struct gs_device_bt_const *bt_const;
+   struct gs_device_bt_const_extended *bt_const_extended; u32 feature;
    bt_const = kmalloc(sizeof(*bt_const), GFP_KERNEL);
</patch>

```


`search_commits("kfree|memory management")`
 commit 5cda3ee5
 Author: Marc Kleine-Budde
 Date: Sun Jun 4 14:03:42 2017 +0200

can: gs_usb: fix memory leak in gs_cmd_reset()

This patch adds the missing kfree() in `gs_cmd_reset()` to free the memory that is not used anymore after `usb_control_msg()`.

```

<patch>
static int gs_cmd_reset(struct gs_usb *gsusb, struct gs_can *gsdev)
    sizeof(*dm),
    1000);
+   kfree(dm);
+
    return rc;
}
</patch>

```

Figure 11: Code Researcher actions (search_commits in green box).

The crash report indicates a memory leak in the `gs_usb_probe` function... *The search results and past commits provide examples of similar memory management issues*... The patch *should add a kfree* call for `bt_const_extended` after its data has been used to populate the `data_bt_const` fields in the `gs_make_candev` function.

```
<patch>
static struct gs_can *gs_make_candev(unsigned int channel,
    dev->data_bt_const.brp_inc = le32_to_cpu(bt_const_extended->dbrp_inc);

    dev->can.data_bittiming_const = &dev->data_bt_const;
+
+    kfree(bt_const_extended); // Free the allocated memory for bt_const_extended
}

SET_NETDEV_DEV(netdev, &intf->dev);
</patch>
```



**Analysis and patch
by Code Researcher**

Figure 12: Code Researcher patch and analysis.

D LLM-as-Judge evaluation of overlap between developer commit and tool-gathered context

We use LLM-as-judge to analyze the context gathered by *Code Researcher* and SWE-agent to determine the overlap of context in their trajectory with the context mentioned by the developer in the ground-truth fix commit message. We first identify code symbols mentioned in the commit message for a given bug b , which we denote as s_b^* . Then for each candidate patch i , we find the overlap of s_b^* with the symbols whose definitions are seen in its trajectory. We denote this overlap by $s_{b,i}$. The prompts used to retrieve these results are included in the supplementary material. We define symbol ratio SR for each candidate patch as

$$SR_{b,i} = \frac{|s_{b,i}|}{|s_b^*|}.$$

We consider patch i to have overlapping symbol context with the developer commit if $SR_{b,i} \geq 0.33$. We label all candidate patches with this criterion. As mentioned in Section 5.3 (2), we find that SWE-agent has 54.01% overlapping symbol context patches, while *Code Researcher* has 63.7% overlapping symbol context patches. This indicates that *Code Researcher* is more effective at identifying relevant context.

Additionally, we also measure the impact of finding relevant context on the crash resolution rate (CRR) as:

$$P(\text{patch resolves crash} \mid \text{overlapping symbol context}) = 0.309,$$

$$P(\text{patch resolves crash} \mid \text{non-overlapping symbol context}) = 0.116.$$

This suggests that patches with overlapping symbol context have a significantly higher probability of resolving crashes than patches without.

In addition to symbols, we also identify commit IDs mentioned in the commit message for a given bug b which we denote as c_b^* . Then for each candidate patch i , we find the overlap of c_b^* with the commits retrieved in its trajectory, denoted as $c_{b,i}$. We note that c_b^* is typically a small number, with a maximum value of 3 in our dataset of 200 bugs. Therefore, instead of a ratio, we label patch i to have overlapping commit context when all the commits in c_b^* are present in $c_{b,i}$ (i.e., $\frac{|c_{b,i}|}{|c_b^*|} = 1$). We find that 30.8% of patches produced by *Code Researcher* have overlapping commit context (recall that SWE-agent does not search over commit IDs). Further, we find that overlapping commit context also has a positive impact on CRR:

$$P(\text{patch resolves crash} \mid \text{overlapping commit context}) = 0.315,$$

$$P(\text{patch resolves crash} \mid \text{non-overlapping commit context}) = 0.205.$$

Overall, these results bring out the utility of effective context retrieval.

E Qualitative evaluation and examples

Example A: jfs_dmap.c boundary check. Listing 1⁸ compares the developer’s ground-truth patch with the patch generated by *Code Researcher*. Both fixes add a lower-bound check on `bmp->db_agl2size` alongside the existing upper-bound check; the only difference is the ordering of the two disjuncts in the if condition, an immaterial variation in this case. This illustrates the class of *Accurate* patches.

Listing 1: Semantically equivalent patch produced by *Code Researcher* for the `jfs_dmap.c` crash.

```
1  --- a/fs/jfs/jfs_dmap.c                                /* developer */
2  +++ b/fs/jfs/jfs_dmap.c
3  @@ -193,7 +193,8 @@ int dbMount(struct inode *ipbmap)
4      bmp->db_agwidth = le32_to_cpu(dbmp_le->dn_agwidth);
5      bmp->db_agstart = le32_to_cpu(dbmp_le->dn_agstart);
6      bmp->db_agl2size = le32_to_cpu(dbmp_le->dn_agl2size);
7  -   if (bmp->db_agl2size > L2MAXL2SIZE - L2MAXAG) {
8  +   if (bmp->db_agl2size > L2MAXL2SIZE - L2MAXAG ||
9  +       bmp->db_agl2size < 0) {
10         err = -EINVAL;
11         goto err_release_metapage;
12     }
13
14  --- a/fs/jfs/jfs_dmap.c                                /* generated */
15  +++ b/fs/jfs/jfs_dmap.c
16  @@ -193,7 +193,7 @@ int dbMount(struct inode *ipbmap)
17      bmp->db_agwidth = le32_to_cpu(dbmp_le->dn_agwidth);
18      bmp->db_agstart = le32_to_cpu(dbmp_le->dn_agstart);
19      bmp->db_agl2size = le32_to_cpu(dbmp_le->dn_agl2size);
20  -   if (bmp->db_agl2size > L2MAXL2SIZE - L2MAXAG) {
21  +   if (bmp->db_agl2size < 0 || bmp->db_agl2size > L2MAXL2SIZE - L2MAXAG)
22  ↪ {
23         err = -EINVAL;
24         goto err_release_metapage;
25     }
```

Example B: hci_h5.c null-check addition. In Listing 2⁹, both the developer and *Code Researcher* address an unsafe access to `hu->serdev->dev`, but the generated patch goes beyond the developer’s fix. While the developer simply guards the power management calls with a null check, *Code Researcher* adds an else branch that logs an error and returns `-ENODEV`. This reflects a conservative design that prevents execution in the event of a null pointer, even though the surrounding kernel code may already guarantee that `hu->serdev` is non-null. Such overspecialization can be seen as a benign deviation: the patch is functionally correct and improves robustness, but at the risk of silently diverging from upstream assumptions. This illustrates the class of *overspecialized* patches.

Listing 2: Overspecialized patch generated for `hci_h5.c`.

```
1  --- a/drivers/bluetooth/hci_h5.c                      /* developer */
2  +++ b/drivers/bluetooth/hci_h5.c
3  @@ -587,9 +587,11 @@ static int h5_recv(...)
4      count -= processed;
5      }
6
7  -   pm_runtime_get(&hu->serdev->dev);
8  -   pm_runtime_mark_last_busy(&hu->serdev->dev);
9  -   pm_runtime_put_autosuspend(&hu->serdev->dev);
```

⁸Example A in Syzkaller: <https://syzkaller.appspot.com/bug?id=5eb8a5d29d77f8a364cf3270bf9625eb4d4ffc52>

⁹Example B in Syzkaller: <https://syzkaller.appspot.com/bug?id=1bc76a5779300075e3466a0e2abcd753df212779>

```

10 +   if (hu->serdev) {
11 +       pm_runtime_get(&hu->serdev->dev);
12 +       pm_runtime_mark_last_busy(&hu->serdev->dev);
13 +       pm_runtime_put_autosuspend(&hu->serdev->dev);
14 +   }
15
16     return 0;
17 }
18
19
20 --- a/drivers/bluetooth/hci_h5.c          /* generated */
21 +++ b/drivers/bluetooth/hci_h5.c
22 @@ -587,9 +587,14 @@ static int h5_recv(...)
23         count -= processed;
24     }
25
26 -   pm_runtime_get(&hu->serdev->dev);
27 -   pm_runtime_mark_last_busy(&hu->serdev->dev);
28 -   pm_runtime_put_autosuspend(&hu->serdev->dev);
29 +   if (hu->serdev) {
30 +       pm_runtime_get(&hu->serdev->dev);
31 +       pm_runtime_mark_last_busy(&hu->serdev->dev);
32 +       pm_runtime_put_autosuspend(&hu->serdev->dev);
33 +   } else {
34 +       bt_dev_err(hu->hdev, "serdev is not initialized");
35 +       return -ENODEV;
36 +   }
37
38     return 0;
39 }

```

Example C: ns.c RCU read lock insertion. In Listing 3¹⁰ both the developer and *Code Researcher* address the unsafe traversal of a radix tree without proper RCU synchronization. The developer applies a comprehensive fix, wrapping all relevant `radix_tree_for_each_slot` iterations with `rcu_read_lock()` and `rcu_read_unlock()` across multiple functions. In contrast, *Code Researcher* focuses only on the `ctrl_cmd_new_lookup()` function, inserting the necessary locking primitives in that scope alone. While this partial patch is not directly mergeable due to its incompleteness, it demonstrates an accurate understanding of the underlying concurrency issue and correctly applies the mitigation in the context it modifies. As such, it exemplifies the class of *incomplete* patches, offering concrete insight into the nature and location of the bug, and accelerating the path toward a complete and upstreamable fix.

Listing 3: Developer and plausible patches for ns.c.

```

1 --- a/net/qrtr/ns.c          /* developer */
2 +++ b/net/qrtr/ns.c
3 @@ -193,12 +193,13 @@ static int announce_servers(struct sockaddr_qrtr *sq)
4         struct qrtr_server *srv;
5         struct qrtr_node *node;
6         void __rcu **slot;
7 -       int ret;
8 +       int ret = 0;
9
10        node = node_get(qrtr_ns.local_node);
11        if (!node)
12            return 0;
13

```

¹⁰Example C in Syzkaller: <https://syzkaller.appspot.com/bug?id=07c9d71dc1a215b19c6a245c68f502bc57dbdb83>

```

14 +         rcu_read_lock();
15         /* Announce the list of servers registered in this node */
16         radix_tree_for_each_slot(slot, &node->servers, &iter, 0) {
17             srv = radix_tree_deref_slot(slot);
18 @@ -206,11 +207,14 @@ static int announce_servers(struct sockaddr_qrtr *sq)
19             ret = service_announce_new(sq, srv);
20             if (ret < 0) {
21                 pr_err("failed to announce new service\n");
22 -                 return ret;
23 +                 goto err_out;
24             }
25         }
26
27 -         return 0;
28 +err_out:
29 +         rcu_read_unlock();
30 +
31 +         return ret;
32     }
33
34     static struct qrtr_server *server_add(unsigned int service,
35 @@ -335,7 +339,7 @@ static int ctrl_cmd_bye(struct sockaddr_qrtr *from)
36         struct qrtr_node *node;
37         void __rcu **slot;
38         struct kvec iv;
39 -         int ret;
40 +         int ret = 0;
41
42         iv.iov_base = &pkt;
43         iv.iov_len = sizeof(pkt);
44 @@ -344,11 +348,13 @@ static int ctrl_cmd_bye(struct sockaddr_qrtr *from)
45         if (!node)
46             return 0;
47
48 +         rcu_read_lock();
49         /* Advertise removal of this client to all servers of remote node
50         ↪ */
51         radix_tree_for_each_slot(slot, &node->servers, &iter, 0) {
52             srv = radix_tree_deref_slot(slot);
53             server_del(node, srv->port);
54         }
55 +         rcu_read_unlock();
56
57         /* Advertise the removal of this client to all local servers */
58         local_node = node_get(qrtr_ns.local_node);
59 @@ -359,6 +365,7 @@ static int ctrl_cmd_bye(struct sockaddr_qrtr *from)
60         pkt.cmd = cpu_to_le32(QRTR_TYPE_BYE);
61         pkt.client.node = cpu_to_le32(from->sq_node);
62
63 +         rcu_read_lock();
64         radix_tree_for_each_slot(slot, &local_node->servers, &iter, 0) {
65             srv = radix_tree_deref_slot(slot);
66
67 @@ -372,11 +379,14 @@ static int ctrl_cmd_bye(struct sockaddr_qrtr *from)
68             ret = kernel_sendmsg(qrtr_ns.sock, &msg, &iv, 1,
69             ↪ sizeof(pkt));
70             if (ret < 0) {
71                 pr_err("failed to send bye cmd\n");
72 -                 return ret;

```

```

71 +                                goto err_out;
72         }
73     }
74
75 -    return 0;
76 +err_out:
77 +    rcu_read_unlock();
78 +
79 +    return ret;
80 }
81
82 static int ctrl_cmd_del_client(struct sockaddr_qrtr *from,
83 @@ -394,7 +404,7 @@ static int ctrl_cmd_del_client(struct sockaddr_qrtr
↪ *from,
84     struct list_head *li;
85     void __rcu **slot;
86     struct kvec iv;
87 -    int ret;
88 +    int ret = 0;
89
90     iv.iov_base = &pkt;
91     iv.iov_len = sizeof(pkt);
92 @@ -434,6 +444,7 @@ static int ctrl_cmd_del_client(struct sockaddr_qrtr
↪ *from,
93     pkt.client.node = cpu_to_le32(node_id);
94     pkt.client.port = cpu_to_le32(port);
95
96 +    rcu_read_lock();
97     radix_tree_for_each_slot(slot, &local_node->servers, &iter, 0) {
98         srv = radix_tree_deref_slot(slot);
99
100 @@ -447,11 +458,14 @@ static int ctrl_cmd_del_client(struct sockaddr_qrtr
↪ *from,
101         ret = kernel_sendmsg(qrtr_ns.sock, &msg, &iv, 1,
↪         sizeof(pkt));
102         if (ret < 0) {
103             pr_err("failed to send del client cmd\n");
104 -            return ret;
105 +            goto err_out;
106         }
107     }
108
109 -    return 0;
110 +err_out:
111 +    rcu_read_unlock();
112 +
113 +    return ret;
114 }
115
116 static int ctrl_cmd_new_server(struct sockaddr_qrtr *from,
117 @@ -554,6 +568,7 @@ static int ctrl_cmd_new_lookup(struct sockaddr_qrtr
↪ *from,
118     filter.service = service;
119     filter.instance = instance;
120
121 +    rcu_read_lock();
122     radix_tree_for_each_slot(node_slot, &nodes, &node_iter, 0) {
123         node = radix_tree_deref_slot(node_slot);
124

```



```

125 @@ -568,6 +583,7 @@ static int ctrl_cmd_new_lookup(struct sockaddr_qrtr
    ↪ *from,
126         lookup_notify(from, srv, true);
127     }
128 }
129 + rcu_read_unlock();
130
131 /* Empty notification, to indicate end of listing */
132 lookup_notify(from, NULL, true);
133
134
135 --- a/net/qrtr/ns.c                      /* generated */
136 +++ b/net/qrtr/ns.c
137 @@ -554,6 +554,7 @@ static int ctrl_cmd_new_lookup(struct sockaddr_qrtr
    ↪ *from,
138         filter.service = service;
139         filter.instance = instance;
140
141 + rcu_read_lock(); // Add RCU read lock
142     radix_tree_for_each_slot(node_slot, &nodes, &node_iter, 0) {
143         node = radix_tree_deref_slot(node_slot);
144
145 @@ -568,6 +569,7 @@ static int ctrl_cmd_new_lookup(struct sockaddr_qrtr
    ↪ *from,
146         lookup_notify(from, srv, true);
147     }
148 }
149 + rcu_read_unlock(); // Add RCU read unlock
150
151 /* Empty notification, to indicate end of listing */
152 lookup_notify(from, NULL, true);

```

Example D: qrtr.c port validation. In Listing 4¹¹, the developer correctly replaces `idr_alloc()` with `idr_alloc_u32()` to avoid integer truncation when handling large u32 port numbers. By contrast, *Code Researcher* adds defensive checks that reject ports with `port < 0`, both in `qrtr_port_assign` and `__qrtr_bind`. This patch reflects a misunderstanding of the root issue: `port` is an int, but the incoming value originates from `__u32 sq_port`, and special constants like `QRTR_PORT_CTRL` (defined as `0xffffffffeu`) are valid and widely used in the subsystem. Interpreting these as negative values and rejecting them causes legitimate use cases to fail. The patch not only fails to address the original warning in `idr_alloc()`, but also introduces a regression by breaking valid assignments. This is a clear instance of *incorrect patches*, where the fix appears misdiagnoses the problem and undermines correct behavior.

Listing 4: Developer and incorrect patches for `qrtr.c`.

```

1 --- a/net/qrtr/qrtr.c                      /* developer */
2 +++ b/net/qrtr/qrtr.c
3 @@ -692,23 +692,25 @@ static void qrtr_port_remove(struct qrtr_sock *ipc)
4     */
5     static int qrtr_port_assign(struct qrtr_sock *ipc, int *port)
6     {
7 +     u32 min_port;
8         int rc;
9
10        mutex_lock(&qrtr_port_lock);
11        if (!*port) {
12 -            rc = idr_alloc(&qrtr_ports, ipc,

```

¹¹Example D in Syzkaller: <https://syzkaller.appspot.com/bug?id=ca2299cf11b3e3d3d0f44ac479410a14eecbd326>

```

13 -             QRTR_MIN_EPH_SOCKET, QRTR_MAX_EPH_SOCKET +
14 ↪ 1,
15 -             GFP_ATOMIC);
16 -             if (rc >= 0)
17 -                 *port = rc;
18 +             min_port = QRTR_MIN_EPH_SOCKET;
19 +             rc = idr_alloc_u32(&qrtr_ports, ipc, &min_port,
20 ↪ QRTR_MAX_EPH_SOCKET, GFP_ATOMIC);
21 +             if (!rc)
22 +                 *port = min_port;
23 } else if (*port < QRTR_MIN_EPH_SOCKET && !capable(CAP_NET_ADMIN))
24 ↪ {
25     rc = -EACCES;
26 } else if (*port == QRTR_PORT_CTRL) {
27 -     rc = idr_alloc(&qrtr_ports, ipc, 0, 1, GFP_ATOMIC);
28 +     min_port = 0;
29 +     rc = idr_alloc_u32(&qrtr_ports, ipc, &min_port, 0,
30 ↪ GFP_ATOMIC);
31 } else {
32 -     rc = idr_alloc(&qrtr_ports, ipc, *port, *port + 1,
33 ↪ GFP_ATOMIC);
34 -     if (rc >= 0)
35 -         *port = rc;
36 +     min_port = *port;
37 +     rc = idr_alloc_u32(&qrtr_ports, ipc, &min_port, *port,
38 ↪ GFP_ATOMIC);
39 +     if (!rc)
40 +         *port = min_port;
41 }
42 mutex_unlock(&qrtr_port_lock);
43
44 --- a/net/qrtr/qrtr.c                /* generated */
45 +++ b/net/qrtr/qrtr.c
46 @@ -695,7 +695,9 @@ static int qrtr_port_assign(struct qrtr_sock *ipc, int
47 ↪ *port)
48     int rc;
49
50     mutex_lock(&qrtr_port_lock);
51 -     if (!*port) {
52 +     if (*port < 0) {
53 +         rc = -EINVAL;
54 +     } else if (!*port) {
55         rc = idr_alloc(&qrtr_ports, ipc,
56             QRTR_MIN_EPH_SOCKET, QRTR_MAX_EPH_SOCKET +
57             ↪ 1,
58             GFP_ATOMIC);
59 @@ -759,6 +761,9 @@ static int __qrtr_bind(struct socket *sock,
60     return 0;
61
62     port = addr->sq_port;
63 +     if (port < 0)
64 +         return -EINVAL;
65 +
66     rc = qrtr_port_assign(ipc, &port);
67     if (rc)
68         return rc;

```