

Towards Better Linux Kernel Fault Localization: Leveraging Contrastive Reasoning and Hierarchical Context Analysis

Haichi Wang
wanghaichi@tju.edu.cn
School of Computer
Software, Tianjin
University
Tianjin, China

Ruiguo Yu
rgyu@tju.edu.cn
School of Computer
Science and Technology,
Information and Network
Center, Tianjin University
Tianjin, China

Yesong Pang
pangyesong_0503@tju.edu.cn
School of Computer
Software, Tianjin
University
Tianjin, China

Yingquan Zhao
zhaoyingquan@tju.edu.cn
School of Cybersecurity,
Tianjin University
Tianjin, China

Junjie Chen*
junjiechen@tju.edu.cn
School of Computer
Software, Tianjin
University
Tianjin, China

Jiajun Jiang
jiangjiajun@tju.edu.cn
School of Computer
Software, Tianjin
University
Tianjin, China

Zan Wang
wangzan@tju.edu.cn
School of Computer
Software, School of
Artificial Intelligence,
Tianjin University
Tianjin, China

Abstract

Debugging the Linux kernel remains a formidable challenge due to its vast codebase, complex architecture, and low-level programming intricacies. Effective fault localization (FL) is thus essential for efficient kernel debugging and maintenance. While existing FL techniques (both traditional and LLM-based) have shown promise in general-purpose software, they are ill-suited for the kernel context. In particular, recent LLM-based techniques often treat bug reports and source code as plain text, lacking deep integration of kernel-specific knowledge, which limits their ability to identify root causes and achieve fine-grained localization.

We present CoHiKER, a novel LLM-based FL technique tailored to the Linux kernel. CoHiKER introduces two key innovations: (1) contrastive reasoning, which identifies root causes by analyzing the behavioral divergence between carefully mutated passing and failing test cases, and (2) hierarchical context analysis, which systematically narrows the localization scope from files to methods by integrating crash reports, syscall semantics, inter-file dependencies, and kernel-specific features. Unlike prior techniques that rely on static understanding and full-code input, CoHiKER decomposes the localization task and enables structured LLM prompting to reason semantically over meaningful contexts.

We evaluate CoHiKER on an extended Linux kernel bug dataset against five state-of-the-art baselines. CoHiKER consistently outperforms all competitors, improving *Top-1* localization accuracy by up to 26.07% at the file level and 56.85% at the method level over state-of-the-art LLM-based baselines, while achieving up to

8.84× and 28.9× reductions in token consumption, respectively. Furthermore, CoHiKER demonstrates strong generalizability on the non-kernel dataset, with comparable gains (15.5% and 5.3% in *Top-1* at file and method levels). These results demonstrate the promise of CoHiKER as a practical and accurate solution for kernel FL.

CCS Concepts

• **Software and its engineering** → **Software testing and debugging**.

Keywords

Fault Localization, Linux Kernel, Large Language Model

ACM Reference Format:

Haichi Wang, Ruiguo Yu, Yesong Pang, Yingquan Zhao, Junjie Chen, Jiajun Jiang, and Zan Wang. 2026. Towards Better Linux Kernel Fault Localization: Leveraging Contrastive Reasoning and Hierarchical Context Analysis. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26)*, April 12–18, 2026, Rio de Janeiro, Brazil. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3744916.3787792>

1 Introduction

The Linux kernel, a core component of Linux-based operating systems, manages critical resources across diverse platforms. Over decades of continuous development, it has evolved into one of the largest software systems, exceeding 40 million lines of code [1]. Maintaining this massive codebase is challenging, and despite extensive community contributions, bugs persist, often impacting system stability and security. While researches have focused on detecting kernel bugs, particularly through advanced fuzzing techniques [2–4], developers still devote substantial effort to localizing root causes before fixing [5]. However, Kernel fault localization is inherently more complex than in general software due to: (1) The pronounced gap between bug symptoms and root causes, driven by asynchronous execution and cross-subsystem coupling that delay and obscure bug manifestations. (2) The massive and highly

*Junjie Chen is the corresponding author.



This work is licensed under a Creative Commons Attribution 4.0 International License. *ICSE '26, Rio de Janeiro, Brazil*

© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2025-3/2026/04
<https://doi.org/10.1145/3744916.3787792>

interdependent codebase, comprising numerous functionally diverse subsystems and far exceeding other software in scale (e.g., the already sophisticated GCC compiler has only about 15 million lines of code [6]). (3) The frequent loss of diagnostic information, resulting from system-wide failures inherent to kernel bugs that erase runtime states and hinder the collection of crucial runtime data (e.g., code coverage).

Fault localization (FL) techniques identify code locations responsible for software failures, aiding developers in diagnosing and resolving bugs more efficiently [7]. In general software, FL techniques fall into three categories: spectrum-based (SBFL) [8, 9], mutation-based (MBFL) [10, 11], and information retrieval-based (IRFL) [12–14]. However, the unique characteristics of the Linux kernel significantly hinder the effectiveness of these traditional techniques. SBFL depends on runtime coverage, often unavailable in kernel crashes [15]; MBFL becomes computationally infeasible due to costly recompilation per mutation; IRFL, which rely on lexical or semantic similarity between bug reports and source code, struggles to cope with the kernel’s low-level semantics and indirect relationship between bug descriptions and faulty code [5].

Large Language Models (LLMs) have recently shown promise for FL in general software [16–18]. LLM-based techniques [19–22] leverage LLMs to interpret bug reports and reason over code semantics to identify likely fault locations. However, their effectiveness remains limited, as existing techniques (e.g., Agentless [19]) rely on static reasoning that naively combine heterogeneous inputs (e.g., bug reports, crash traces, and source code) into uniform representations. Such static and simplified reasoning is inadequate for kernel fault localization because it: (1) Neglects the distinctive nature of kernel bugs, where the gap between bug symptoms and root causes (e.g., a file-system bug triggers a crash in memory management through shared states, with the faulty file absent from the bug report) makes reasoning solely over static information insufficient. (2) Overlooks the massive scale and complex dependencies in kernel, where even partial code contexts can exceed the input capacity of LLMs. To improve FL for the Linux kernel, LinuxFL+[20] was recently proposed as the first LLM-based technique tailored to this domain. However, LinuxFL+ incorporates only limited kernel-specific knowledge (such as incorporating filenames extracted from bug-related emails to expand suspicious file sets) and does not explicitly address the limitations of static reasoning and scalability issues that persist in prior LLM-based techniques. Therefore, a more effective and kernel-aware FL technique is still needed.

To address these limitations, we propose **CoHiKER** (Contrastive Reasoning and Hierarchical Context Analysis for Kernel Fault Localization), a novel LLM-based FL technique tailored to the Linux kernel. CoHiKER introduces two key innovations: (1) **Contrastive Reasoning**: To address the limitation of static reasoning, CoHiKER performs contrastive reasoning to bridge the gap between bug symptoms and root causes in kernel fault localization. In particular, CoHiKER mutates failing tests to generate passing variants and leverages LLMs to reason over their execution results differences (i.e., pass or fail). By identifying causal differences between test behaviors, it isolates fault-relevant kernel features and narrows the root cause search space. This alleviates the coverage loss caused by kernel crashes, and mitigates the reliance of SBFL[8, 9] and MBFL[10, 11] on code coverage or costly recompilation after kernel

code mutations. (2) **Hierarchical Context Analysis**: To cope with the kernel’s large codebase under LLM input constraints, CoHiKER employs a hierarchical analysis framework that integrates context at both file and method granularity, enabling scalable reasoning and progressive refinement for more precise localization. At the file level, it identifies Top-K suspicious files by analyzing textual cues from bug reports and inter-file dependencies. At the method level, it analyzes internal code structures, such as call hierarchies and parameter flows, to rank suspicious methods within those files. Finally, contrastive reasoning results (i.e., explanations of how the bug is triggered) are integrated across both levels to refine the fault ranking by aligning kernel semantics with root cause hypotheses. Together, these components enable CoHiKER to overcome the limitations of prior techniques, offering a kernel-specific, causally grounded, and context-aware framework for more accurate and interpretable FL in complex kernel environments.

To evaluate CoHiKER, we extended an existing kernel FL dataset [23] to 210 verified instances, comparing it against two categories of baselines: (1) traditional FL techniques, including the file-level IR-based technique PathIdea [14], the method-level IR-based technique FineLocator [13] and the method-level trace-based technique CrashLocator [24]; and (2) state-of-the-art LLM-based techniques, namely Agentless [19], LinuxFL+ [20] (extended from SWE-Agent [16]) and SoapFL [25] (for non-kernel context only). As only LinuxFL+ was designed for the Linux kernel, we adapted the other techniques to ensure compatibility with the kernel context. Results show CoHiKER achieves significantly higher FL accuracy. For example, the traditional PathIdea achieves only 6.7% Top-1 accuracy at the file level, while CoHiKER achieves 55.24%. The traditional FineLocator cannot identify any faulty method as Top-1, while CoHiKER achieves 32.86% Top-1 accuracy at the method level. Compared to LLM-based baselines (Agentless and LinuxFL+), CoHiKER improves Top-1 accuracy by 4.5%–26.70% at the file level and 32.71%–56.85% at the method level.

CoHiKER also demonstrates superior efficiency, reducing token consumption by $4.46\times$ – $8.84\times$ at file level and $21.92\times$ – $28.90\times$ at method level compared to baselines. Furthermore, CoHiKER shows strong generalizability on non-kernel systems (SWE-bench-lite [26]), achieving improvements over state-of-the-art baselines (e.g., 15.5% and 5.3% in Top@1 at file and method levels) that align with its gains in the kernel domains. These results confirm CoHiKER’s consistent superiority across diverse systems, highlighting the broad applicability of its core methodologies.

To sum up, this work makes the following major contributions:

- We propose CoHiKER, a novel LLM-based FL technique tailored to the Linux kernel, combining contrastive reasoning and hierarchical context analysis to address the unique challenges in complex kernel scenarios.
- We introduce a contrastive reasoning component that leverages LLMs to compare failing and passing test cases, enabling the identification of root causes that are tightly linked to kernel-level bug manifestations.
- We design a hierarchical context analysis framework, which discriminatively incorporates file-level and method-level bug signals while optimizing for LLM input efficiency and contextual relevance.

```

1 r0 = syz_open_procfs$pagemap(0x0 ①-> -1, &(0x7f0140))
2 ioctl$PAGEMAP_SCAN(r0, 0xc0606610,
  &(0x60 ②-> 0x0, ..., &(0x7f/0x1000 ③-> 0x2000)=nil,
  &(0x7f20/0x2000)=nil, 0x0, 0x0})
Reasoning root causes:
... (1) a valid pagemap file is opened through fs/proc/base.c
with proper process, and (2) a PAGEMAP_SCAN is issued with
scan size (0x60), triggering pagemap scan operation. ...
Syscall analysis:
open_procfs$pagemap: fs/proc/base.c
ioctl$PAGEMAP_SCAN: fs/proc/task_mmu.c
LLM inference: fs/proc/generic.c, mm/page_alloc.c

(a) Test Case
1 ...
2 Register Values: ...
3 Call Trace:
4 unmap_single_vma+0x194/0x2b0 mm/memory.c:1699 //file1
5 ...
6 exit_mmap+0x1ad/0xa60 mm/mmap.c:3308 //file2
7 __mmap+0x12a/0x4d0 kernel/fork.c:1349 //file3
8 ...
9 exit_mm kernel/exit.c:567 //file4
10 ...
11 do_syscall_x64 arch/x86/entry/common.c:51 //file5
12 ...

```

(b) Crash Trace

Figure 1: Bug Report: WARNING in unmap_page_range [27]

- We conduct extensive experiments on an extended kernel FL dataset, showing that CoHiKER achieves superior localization accuracy and significantly reduces token consumption compared to state-of-the-art LLM-based baselines.

2 An Illustrative Example

We use a real-world example [27] to show CoHiKER’s kernel bug localization process. Figure 1a shows a simplified test case that triggers a crash: it opens a file via `syz_open_procfs$pagemap` (Line 1), then performs a `PAGEMAP_SCAN` operation (Line 2). A bug in the kernel’s memory management subsystem causes this operation to crash the system (*ref.* Figure 2). Figure 1b shows the corresponding crash trace (omitting repetitive frames). While the trace captures the symptom, it does not clearly indicate the root cause, complicating localization given the semantic gap between the syscall-level test case and the underlying kernel code.

Specifically, the kernel crash precludes code coverage collection, making coverage-based techniques (e.g., SBFL) inapplicable. Furthermore, the crash trace shows symptoms in the `mm/` and `kernel/` subsystems, but the root cause actually lies in the functionally distinct `proc/` subsystem. This semantic discrepancy between the crash context and the actual faulty location hinders IR-based methods from identifying correct files. While recent LLM-based methods attempt to bridge this gap by incorporating semantically relevant code, they often struggle with the significant functional misalignment between bug reports and the faulty code. Consequently, they either fail to include the desired faulty code or encounter a massive search space. For instance, LinuxFL+ still includes 3371 files in its refined space for this example, making it challenging to precisely localize the fault due to the limited LLM input capacity. Hence, more effective kernel fault localization techniques are urgently needed.

To address these limitations of existing approaches, CoHiKER begins with contrastive reasoning to uncover the root cause of the failure. This is done by comparing how small changes to the test

case affect the observed execution results. Specifically, CoHiKER leverages the LLM (i.e., DEEPSEEK-V3[28]) to generate mutated variants (three in this example) of the failing test case (Figure 1(a)) by slightly altering each parameter (highlighted in red). The LLM is guided to modify only interface-related arguments (e.g., flags, sizes) while preserving the syscall sequence, keeping each mutant semantically close to the original test. Instead of treating each test in isolation, the LLM compares execution results across passing and failing tests and reasons about their implications. Specifically, the first two mutations (① and ②) prevent the test failure while the third (③) does not. Using this contrast, along with an analysis of the passing/failing test cases and the syscall definition (explaining the functionality of each parameter), the LLM infers the root cause: faulty handling of valid input combinations in the kernel’s memory management logic, specifically for a `PAGEMAP_SCAN` operation using a file like `0x0` and a scan size of `0x60`. The LLM’s reasoning output in Figure 1a demonstrates that by focusing on how test cases reveal or suppress faulty kernel behavior, CoHiKER enables deep, LLM-driven causal reasoning that isolates the true root causes of bugs.

However, root cause inference alone is insufficient for actionable localization given the massive codebase in the kernel, such as still 3371 files left even by using the state-of-the-art LinuxFL+ for space refinement as aforementioned, making the fine-grained method-level fault localization extremely hard. To make the process scalable and precise, CoHiKER applies hierarchical localization: first identifying a small set of suspicious files, then drilling down to individual methods. At the file level, CoHiKER parses the crash trace (Figure 1b) to extract five candidate files (marked in blue) and analyzes system calls (e.g., `ioctl$PAGEMAP_SCAN`) in the test case to infer more relevant files (e.g., `fs/proc/task_mmu.c`). The objective for this process is to create a manageable search space that includes the faulty files. It starts with the stack trace because the actual faulty files, even though not explicitly listed, must have dependencies with some file presented in the trace. Specifically, CoHiKER leverages the LLM to infer additional dependent files based on test case semantics and crash context. Finally, it ranks candidate files by aligning them with the contrastive reasoning output, which substantially narrows the search space. In this case, as the bug involves faulty handling of `PAGEMAP_SCAN`, CoHiKER forms a focused set of 18 files closely related to the `proc/` subsystem and correctly prioritizes `fs/proc/task_mmu.c`, as shown in Figure 2.

CoHiKER then conducts method-level analysis within the set of suspicious files, as file-level granularity is often too coarse for practical debugging. The file `fs/proc/task_mmu.c` alone contains 96 methods (many omitted in Figure 2 for brevity). As a consequence, it still results in 1,034 methods to analyze in this case even if only considering the top 10 suspicious files from the file-level results like existing approaches [20], which remains a substantial search space. To further narrow the search space, CoHiKER prunes candidates with low likelihood of being faulty, using a defect prediction model trained on historical kernel bugs, which results in 81 methods left as suspicious. For each remaining method, it prompts the LLM to summarize its functionality based on the method’s signature and body. It then ranks methods by matching summaries with the inferred root cause. In this example, `pagemap_scan_pmd_entry` is ranked top-1, as its functionality closely matches the inferred root cause.

```

1 // in fs/proc/task_mmu.c
2 pagemap_scan_pmd_entry(...) {                //method1
3     ...
4 - if (!p->vec_out) {
5 + if ((p->arg.flags&PM_SCAN_WP_MATCHING)&&!p->vec_out) {
6     /* Fast path for performing exclusive WP */
7     ...
8 }
9 proc_map_release(...) { ... }                //method2
10 smaps_rollup_open(...) { ... }               //method3
11 clear_refs_write(...) { ... }               //method4
12 ...

```

Figure 2: Suspicious File: fs/proc/task_mmu.c

This diagnosis is confirmed by the official patch, which modifies this method directly (Lines 4–6, Figure 2).

To sum up, CoHiKER enables a more accurate root cause inference of the fault and a more precise search space refinement at both file and method levels by integrating the contrastive reasoning and hierarchical context analysis, addressing key limitations of prior techniques in the complex landscape of Linux kernel debugging.

3 Technique

In this paper, we propose **CoHiKER**, a framework for Linux kernel fault localization that combines *contrastive reasoning* for root cause identification with *hierarchical context analysis* for efficient diagnostic signal utilization within LLM input constraints.

Figure 3 illustrates CoHiKER’s three main stages: **First**, it leverages LLMs to identify precise root causes by analyzing both failing and intentionally generated passing test cases. **Second**, it locates suspicious files via file-level fault localization (FL@F, stage ②). Specifically, CoHiKER first identifies suspicious files using bug reports and system call (i.e., syscall) analysis via LLM-based context enrichment. The final candidates undergo LLM analysis to rank the Top-K most suspicious files by aligning contrastive reasoning results. **Third**, it locates suspicious methods through method-level fault localization (FL@M, stage ③). Specifically, given suspicious files by FL@F, CoHiKER combines a fault prediction model and LLMs to first identify the most probable methods, which are then analyzed by the LLM to rank and select the Top-K most suspicious methods via aligning the contrastive reasoning results.

3.1 Contrastive Reasoning

As outlined in Section 1, precise root-cause analysis is essential for effective fault localization. To this end, CoHiKER introduces a contrastive reasoning component that identifies bug-related root causes by analyzing test cases along with their execution outcomes, operating in two stages: contrastive mutants generation and root cause reasoning.

CoHiKER begins by leveraging mutation analysis [10] to generate contrastive mutants. Traditionally used to assess test suite quality, mutation analysis has also been applied to detect bugs in compilers and other systems[29–31]. In CoHiKER, however, mutation analysis serves a different goal: to identify bug-relevant elements in test cases by observing which mutations cause the failure to disappear (i.e., pass) or persist (i.e., fail). Given complex syscall parameter constraints, traditional handcrafted mutation operators struggle

to preserve semantic correctness or guide mutations toward passing behavior. Hence, CoHiKER’s LLM-based mutator leverages semantic understanding and contextual reasoning to generate minimally constraint-aware test variants. The prompt (Figure 4) used in CoHiKER consists of three components: a role-defining system message, a specific task description, and an input-output format. Following the common LLM prompting practice [16], CoHiKER adopts a Chain-of-Thought (CoT) prompting strategy to encourage step-by-step reasoning.

For a failing test case, the LLM is prompted to (1) analyze how the bug is triggered, (2) identify potential fault-inducing system call and its parameters, and (3) suggest minimal modifications to these parameters (e.g., file descriptors, pointers) that may prevent the bug from being triggered. CoHiKER then generates mutants with small syscall parameter changes. This design choice is motivated by two reasons: (1) Coarse-grained mutations (e.g., modifying entire syscall instructions) often break data dependencies (e.g., variable references), making it harder to isolate fault-relevant behavior. (2) Fine-grained parameter-level mutations yield more precise causal signals, as parameters typically map to specific functionalities in kernel code and require fewer semantic checks. Each generated mutant is executed to determine whether the original bug persists. This process continues until a sufficient contrastive set is obtained (three passing mutants along with the original failing test by default, balancing LLM input constraints with reasoning quality in follow-up root cause reasoning).

With these contrastive mutants, CoHiKER infers the bug’s root cause by comparing semantic differences between the original test case and its contrastive variants. This step also leverages LLMs, using a dedicated reasoning prompt shown in Figure 5. The reasoning process unfolds in three stages: (1) The LLM interprets the original test to infer exercised kernel subsystems (e.g., memory management, file systems). (2) It then examines passing mutants to understand how specific input modifications may prevent the bug, thereby identifying the functionalities or subsystems affected by those changes. (3) Finally, the LLM compares this analysis with the failing test to isolate the key differences that consistently correlate with bug manifestation. By triangulating these insights, CoHiKER infers a precise and interpretable root cause that reflects the kernel-level logic responsible for the failure. This contrastive reasoning component allows CoHiKER to move beyond surface-level pattern matching and instead derive causal explanations grounded in system behavior.

3.2 File-Level Fault Localization

While contrastive reasoning infers plausible root causes, locating these within the kernel’s complex codebase still remains challenging. To address this, CoHiKER introduces a hierarchical context analysis framework that performs fault localization at both the file and method levels. By leveraging the distinct semantics and granularity of each level, this design improves information utilization while narrowing the search space for fault localization. Specifically, CoHiKER begins by identifying suspicious files, which then form the basis for finer-grained localization at the method level.

At the file level, CoHiKER begins by analyzing the crash report, which typically reveals the dynamic execution path taken during the

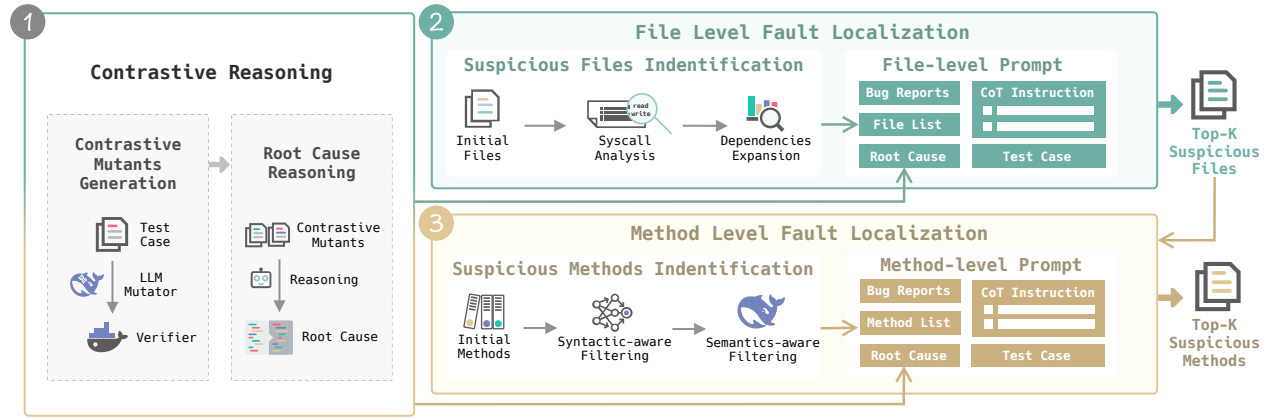


Figure 3: Overview of CoHiKER

System: You are an AI assistant for program analysis. Given a bug report and a test case, your task is to generate 10 minimally modified mutants of the test case that avoid triggering the original bug. Follow the steps below carefully.

1. Read the bug report and understand the bug
2. Analyze each syscall in the test case
3. For each of the 10 mutants:
 - (1) Select one syscall and apply a small, valid change to one of its parameters to the original test case.
 - (2) Ensure the mutant avoids triggering the bug while preserving semantic validity.

Input: <TEST_CASE> <BUG_REPORT> <SYSCALL_DEFINITION>
Output: 1. Provide 10 mutants with minimally mutation
 2. Keep all syscall parameters visible in the output
 3. Ensure mutants are meaningful and bug-avoiding

Figure 4: Prompt for Test Case Mutation

System: You are an AI assistant for program analysis. Given the test case along with its mutants, infer the root cause of the fault. Follow the steps below carefully

1. Understanding the bug's root cause, including involved kernel subsystem and kernel resource lifecycle phase under test.
2. Analyze successful mutants to determine how they prevent the bug, what lifecycle or kernel functionalities are affected.
3. Analyze failed mutants, focus features still trigger the bug.
4. Compare successful and failed mutations to infer the critical operations necessary for bug manifestation and infer faults at multiple granularities.
5. Summarize the root cause of the fault, including what conditions or sequences are required for the bug to manifest.

Input: <TEST_CASE> <PASSING_MUTANT> <FAILING_MUTANT>
Output: Provide the analysis of the root cause in JSON format.

Figure 5: Prompt for Contrastive Reasoning

failure. Kernel files referenced in the crash trace are treated as initial candidates, as they are likely involved in the observed malfunction. However, crash traces often provide only partial clues: asynchronous faults, delayed state corruptions, or silent failures may prevent key faulty files from appearing in the trace. To address this limitation, CoHiKER complements the crash report with a deeper analysis of the failing test case to identify additional fault-relevant files that may not surface in the trace. Specifically, it expands candidates from two perspectives: (1) **System Call Analysis:** CoHiKER leverages LLMs to identify syscalls in the test case and map them to semantically related kernel files, where static analysis is often unreliable and fails under heavy macro expansion and conditional compilation [32]. For instance, a file-open syscall may span file-system and memory-management in ways obscured by macro expansion,

causing static analysis to miss these links, whereas LLM can bridge the gap by leveraging semantic cues from naming, documentation, and usage contexts. (2) **Dependency Expansion:** CoHiKER further uses LLMs to infer semantic and functional dependencies among kernel files, capturing cross-module relations that static analysis often misses [4]. For example, opening a file requires not only `fs/open.c`, but also permission checks in `security/`, forming a dispersed dependency that static analysis struggles to identify but remains semantically apparent to an LLM based on the file context.

After these analysis, CoHiKER compiles a comprehensive list of suspicious files that serves as the foundation for subsequent method-level localization. This final set includes: (1) files extracted from the crash report, (2) files mapped from syscalls in the test case, and (3) LLM-inferred dependency-linked files. A concrete illustration is provided in Section 2, where the final suspicious set includes files from the crash trace (Figure 1(b)), syscall mappings (Figure 1(a)), and semantically related dependencies. To ensure completeness, CoHiKER adopts a conservative strategy that prioritizes expanding the suspicious-file set over early pruning. This is crucial, as file-level results directly influence method-level analysis, i.e., any missing files at this stage could compromise the overall localization accuracy. By capturing a broader yet semantically grounded file set, CoHiKER lays a robust foundation for fine-grained localization.

Compared to scanning the entire kernel codebase, CoHiKER's file-level localization significantly narrows the search space, improving efficiency and fitting LLM input constraints. To further prioritize the most relevant files, CoHiKER ranks the suspicious candidates using the contrastive reasoning results described in Section 3.1. Specifically, CoHiKER employs a dedicated file-level localization prompt, illustrated in Figure 6. Following best practices in recent LLM-based techniques, the prompt defines a clear expert role for the model and adopts a CoT prompting strategy to decompose the task into interpretable reasoning steps. The CoT process begins by instructing the LLM to analyze the bug report, description of inter-file dependency, and the inferred root cause (Steps 1–2). Next, the LLM is prompted to summarize the functionality and distinguishing features of each suspicious file. Based on this analysis, it ranks the files by estimating their likelihood of being fault-relevant, considering alignment with the root cause and the bug report's contextual clues (Steps 3–5). The resulting Top-K

System: You are an expert in software fault localization. Your task is to localize the buggy file in the Linux kernel.

1. Understand the bug report and test case.
2. Understand the root cause analysis.
3. Think about what subsystems are involved in failure context, search in the candidate files which are related to them.
4. Think about the bug category the root cause points to, e.g., memory, concurrency, etc., and prioritize corresponding files that are responsible for involved modules.
5. Rank top <K> Files along with justification based on step3-4.

Input: <BUG_REPORT> <TEST_CASE> <CANDIDATE_LIST> <ROOT_CAUSE>

Output: Provide the analysis of each step in JSON format

Figure 6: Prompt for File-level Fault Localization

Table 1: Defect Prediction Features of Kernel Method

Category	Feature	Description
Structural Complexity	Parameters	Parameter numbers of the given method.
	Cyclomatic[33]	Linearly independent paths in the given method.
	Halstead Effort[34]	Workload metric based on operator statistics.
Call Relationship	Max X Depth	Maximum depth of X construct in the method.
	Call Depth	Maximum local call depth of the given method.
	In-Degree	Methods that call the given method.
Code Quality	Out-Degree	Method calls in the given method.
	Change Time	Time since last modification before versioning.
	Global Variables	Global variables used in the given method.
	Static Variables	Static variables used in the given method.

suspicious files form the input for method-level localization stage, ensuring finer-grained analysis builds on a focused candidate set.

3.3 Method-Level Fault Localization

While file-level localization significantly reduces the search space, each kernel file still contains about 55 methods on average, making method-level localization computationally demanding. To further refine the search space, CoHiKER applies method-level filtering by exploiting method-specific characteristics to pinpoint likely faulty methods within suspicious files. This filtering consists of two complementary strategies: (1) syntactic-aware filtering and (2) semantics-aware filtering. Syntactic-aware filtering examines the static code structure to exclude methods exhibiting simple or low-risk patterns, effectively pruning unlikely candidates. Semantics-aware filtering then leverages information from bug reports and file-level rankings to evaluate the functional relevance of the remaining methods, enabling more precise candidate selection. The following sections provide a detailed explanation of each filtering strategy.

Syntactic-Aware Filtering. CoHiKER employs the widely-used multilayer perceptron (MLP) [35, 36] to train a lightweight multi-task defect prediction model to identify and retain highly suspicious methods. Using defect prediction models for candidate filtering is a well-established practice in fault localization research [37]. Following this practice, we collect historical kernel defect data and extract method-level features to train a model specialized for kernel methods. Table 1 summarizes the selected features, including their categories, names, and descriptions. These features have been widely adopted and validated in prior defect prediction studies [33, 38–40]. Specifically, CoHiKER constructs a labeled dataset comprising historically faulty and non-faulty kernel methods, each represented by a feature vector based on the metrics detailed in Table 1. We then train a multi-task multilayer perceptron model [35, 36] to distinguish faulty methods from non-faulty ones. Applying this model to the methods within the Top-K suspicious files identified by FL@F,

each method’s feature vector is scored for defect likelihood. CoHiKER filters methods based on these scores, retaining only those with high likelihood (i.e., the top 80%) for further semantic analysis.

Semantics-Aware Filtering. The second filtering stage refines the set of suspicious methods from a functional semantics perspective. For each candidate method, CoHiKER prompts an LLM to generate a concise summary describing the method’s functionality, based on its name and signature. These summaries provide semantically meaningful representations that go beyond syntactic structure and are particularly useful for assessing the method’s relevance to the observed bug. CoHiKER then evaluates each method summary in the context of the bug report and the file-level localization results. The bug report are typically used as semantic anchors by providing critical keywords, error descriptions, and symptoms, etc. The file-level ranking provides a prior suspicion score, reflecting the likelihood that a file contains the faulty method. To balance precision and efficiency, CoHiKER applies a proportion-based candidate selection strategy, guided by file rankings and informed by empirical distributions from prior studies [20]. Specifically, CoHiKER allocates more method candidates to higher-ranked files based on their estimated suspiciousness, where 60% of the candidate methods from the Top-1-3 file, 30% from files ranked 4–5, and the remaining 10% to the other files. If prior files lack sufficient methods, the missing candidates will be supplemented from subsequent files.

CoHiKER then ranks the filtered suspicious methods by integrating the contrastive reasoning results described in Section 3.1. To enable effective method-level fault localization, CoHiKER designs a specialized prompt, as illustrated in Figure 7. While the prompt structure follows the same CoT reasoning framework used at the file level, it is tailored to capture fine-grained method-level semantics and their alignment with the inferred root cause. Specifically, the prompt guides the LLM through three stages: The LLM is first instructed to analyze the bug report and contrastive root cause explanation, as well as the LLM-generated summary of each method’s functionality. For each method, the LLM is asked to evaluate whether its functionality logically aligns with the identified root cause. This step emphasizes semantic reasoning over structural patterns and requires the LLM to infer whether the method could plausibly implement or affect the faulty behavior. Based on this alignment, CoHiKER instructs the LLM to assign a suspicion score to each method and output the Top-K most likely faulty methods.

This method-level ranking mechanism enables CoHiKER to connect high-level fault descriptions derived from test behavior to specific code-level implementations. By explicitly reasoning over the alignment between root causes and method semantics, CoHiKER ensures precise, interpretable, and actionable fault localization.

4 Evaluation

To systematically assess the effectiveness of our fault localization technique, we formulate five research questions:

- **RQ1 (File-Level Localization):** How effective is CoHiKER in identifying faulty source files within the Linux kernel?
- **RQ2 (Method-Level Localization):** How effective is CoHiKER in pinpointing the exact faulty methods within the Linux kernel?
- **RQ3 (Efficiency Analysis):** How efficient is CoHiKER compared with baselines, in terms of runtime and token cost?

System: You are an expert in software fault localization. Your task is to localize the buggy method in the Linux kernel.

1. Understand the bug report and test case.
2. Understand the root cause analysis.
3. Think about the direct observable failure (e.g., null dereference, panic), and search in the candidate methods that are responsible for these operation (e.g., incorrect pointer handling, resource allocation).
4. Trace symbolic call stack, reason about when do the bug happen in kernel lifecycle (e.g. initialization, runtime execution, teardown), and search for the corresponding candidate methods.
5. Think about what kernel function trigger the failure from the root cause, search for the candidates with same functions.
6. Rank top <K> methods along with justification based on step3-5.

Input: <BUG_REPORT> <TEST_CASE> <CANDIDATE_LIST> <ROOT_CAUSE>

Output: Provide the analysis of each step in JSON format

Figure 7: Prompt for Method-level Fault Localization

- **RQ4 (Ablation Study):** What are the effectiveness and costs of individual components in CoHiKER?
- **RQ5 (Generalizability to non-kernel systems):** How does CoHiKER perform on other (non-kernel) systems ?

4.1 Experimental Setup

4.1.1 Dataset. To answer RQ1-RQ4, we use a dataset extended from KGYM [23], a publicly kernel FL dataset used in previous work [41]. Specifically, it originally consists of 279 Linux kernel bugs with the associated bug reports, test cases, patches, and reproduction instructions.

Due to platform differences (local Ubuntu 20.04/QEMU 4.2.1 vs. Google Cloud), we reproduced 88 of the original bugs. To ensure representativeness and reliability of the evaluation results, we extended the dataset using KGYM’s pipeline, collecting 684 additional bugs from Syzbot [41] and reproducing 122, yielding 210 total bugs for our final evaluation. The details of these bugs are presented in Table 2, where we report the total number of bugs (Total) and the number of bugs that we successfully reproduced (Repro).

In particular, while manually checking the quality and correctness of the datasets, we found that some bug reports may explicitly reference the faulty files or methods, which can significantly reduce the difficulty of fault localization. To enable an in-depth analysis of our approach’s effectiveness, we further split these bugs into two sub-groups, one of which contains bugs whose bug reports explicitly mentioned the genuine faulty file or faulty method while the other group does not. The details are shown in Table 2. To ease the presentation, in the following, we use D_F^{MN} to represent the set of bugs whose bug reports explicitly mention (MN) the faulty files; D_F^{UN} to represent the set of bugs whose bug reports do not mention (UN) the faulty files; D_M^{MN} to represent the set of bugs whose bug reports explicitly mention the faulty methods; D_M^{UN} to represent the set of bugs whose bug reports do not mention the faulty methods.

To answer RQ5, we evaluate whether CoHiKER remains effective on non-kernel systems, since its core ideas (contrastive reasoning and hierarchical context analysis) are general and applicable to other software systems. We therefore experiment on SWE-bench-lite [26], a widely used dataset for fault localization on non-kernel systems. In particular, SWE-bench-lite contains 300 bugs from 11 popular open-source projects (e.g., Django, sympy, matplotlib), each paired with bug reports, bug-triggering test cases, repair patches, and instructions for reproduction, which is similar to our kernel

Table 2: Details of the Used Datasets

Source	Total	Repro	MN / UN (file)	MN / UN (method)
Kgym	279	88	79/9	50/38
Syzbot	684	122	83/40	61/61
Summary	963	210	161/49	111/99

Table 3: Details of the Compared Baselines

Techniques	Kernel Specific	Kernel Dataset (RQ1-RQ4)	Non-kernel Dataset (RQ5)	FL level	
				FL@F	FL@M
Pathidea	✗	✓	✗	✓	✗
FineLocator	✗	✓	✗	✗	✓
Crashlocator	✗	✓	✗	✗	✓
LinuxFL+	✓	✓	✗	✓	✓
Agentless	✗	✓	✓	✓	✓
SWE-Agent	✗	✗	✓	✓	✓
SoapFL	✗	✗	✓	✓	✓

setup, but additionally includes coverage information that is often unavailable for kernel bugs.

4.1.2 Baselines. To comprehensively evaluate the effectiveness of CoHiKER, we compare it with seven fault localization techniques across both kernel and non-kernel scenarios. Table 3 summarizes all compared techniques, where the Kernel-Specific column indicates whether the technique is designed for kernel fault localization, while the remaining two columns describe the datasets and the localization granularities used in our experiments (determined by the design of each technique).

To answer RQ1-RQ4, we compare CoHiKER against the former five fault localization techniques in Table 3. Among these techniques, LinuxFL+ [20] is the latest and the only technique designed for kernel bugs, whereas the others target general-purpose software. Specifically, Crashlocator [24] performs localization by analyzing crash stack traces, Pathidea [14] and FineLocator [13] are two representative IR-based FL techniques, while Agentless [19] is the state-of-the-art LLM-based FL technique that represents the latest advance of LLMs in the FL domain [19]. We did not evaluate SWE-Agent and SoapFL in RQ1-RQ4, because SWE-Agent serves as the underlying agent framework of LinuxFL+, which is already evaluated, whereas SoapFL requires coverage information that is not always available in kernel.

To answer RQ5, we compare CoHiKER against the latter three localization techniques in Table 3. Here, we did not evaluate all techniques, since their effectiveness on non-kernel systems has been extensively studied [25, 42]. Therefore, we select the state-of-the-art Agentless and the kernel-specific technique LinuxFL+ as representatives. However, since LinuxFL+ builds on SWE-Agent by adding kernel-specific information (e.g., mailing-list context) that is unavailable in the non-kernel setting, we use SWE-Agent instead. Moreover, we additionally compare against SoapFL, which identifies suspicious methods through coverage analysis of failing tests. We include it here because coverage has never been used for kernel fault localization, making it a meaningful baseline here.

Note that we exclude traditional techniques (e.g., SBFL [8, 9], MBFL [10, 11]) because: (1) They cannot be effectively applied to kernel bugs (as discussed in Section 1), and (2) for non-kernel systems,

existing LLM-based techniques (e.g., Agentless) have been shown to outperform traditional approaches. For all the compared techniques, we used their default setting and adhered to their original design, as shown in the last two columns of Table 3. For techniques not originally designed for kernel bugs, we apply the necessary adaptations, which will be detailed in Section 4.1.4.

4.1.3 Metrics. To measure the performance of different techniques in localizing bugs, we adopted three metrics that are commonly used in existing FL studies [8, 43, 44]. The details of them are presented as follows:

- *Top@n*: The ratio of kernel bugs whose faulty file/method appears in the top- n ranked candidates (for $n = 1, 3, 5, 10$).
- *Mean Reciprocal Rank (MRR)* [43]: The average reciprocal rank of the ground-truth faulty file/method across all kernel bugs, where higher values indicate better ranking performance.
- *Mean First Rank (MFR)* [45]: Given the best rank of the faulty file/method (if involving multiple ones) for a kernel bug, MFR is the average value of the best ranks across all faulty files/methods.

4.1.4 Implementation and Configuration. We use Python for implementation, with Deepseek-V3 [28] as the core LLM backend. The output limit K is set to 10, aligning with developer preferences [46]. The LLM temperature is 0 to ensure reproducibility.

For syntactic-aware filtering, we follow prior work and train a lightweight defect prediction model, i.e., multilayer perceptron (MLP) model, to filter low-risk methods. Specifically, we collect 1,000 additional defective kernel bugs from the Syzbot project (not in our dataset) and train a lightweight multi-task learning defect prediction model using a MLP architecture. It takes a 10-dimensional vector as input, where each dimension represents a normalized value of a structural feature (detailed in Table 1). Training is conducted over 200 epochs with Adam ($lr = 1e - 3$). We then filter the bottom 20% of candidates. For the semantic-aware filtering, the number of total candidate methods is constrained by the LLM’s context length. In our setting, a 16k-token context budget allows summarizing approximately 80 methods per bug, using the proportion-based selection strategy from Section 3.

To adapt CoHiKER to SWE-bench-lite, we introduce three modifications. First, in contrastive reasoning, we mutate function calls in the failing tests (analogous to syscalls in the kernel). To ensure the validity of mutations, we use static analysis to extract function signatures and provide them as constraints. In cases where function calls may be absent, we follow prior work (DIWI [47]) to perform minimal-expression mutations, such as modifying operators. Second, in syntactic-aware filtering, we retrain the defect prediction model using 1,000 failing tests from SWE-bench[26]. Finally, in all the LLM-based components, we simply replace kernel-specific context with project-specific context while retaining original design.

We reuse LinuxFL+’s public implementation with SWEAgent [16], and faithfully reproduce other baselines by closely following the methodologies described in their respective publications. Note that Crashlocator was only evaluated on a subset of our dataset, i.e., 123 out of 210 bugs, due to environment incompatibility issues as the Linux kernel bugs in the dataset span hundreds of kernel versions. For SoapFL, we directly reuse its publicly released implementation.

We have released all our implementations for replication at [48].

Table 4: File-level Fault Localization Result (RQ1)

Data	Technique	<i>Top@1</i>	<i>Top@3</i>	<i>Top@5</i>	<i>Top@10</i>	<i>MRR</i>	<i>MFR</i>
D^{all}	Pathidea	6.7%	13.4%	15.78%	34.92%	0.11	8.06
	Agentless	52.86%	70%	75.24%	81.43%	0.63	3.66
	LinuxFL+	43.6%	63.81%	70.95%	72.86%	0.55	4.34
	CoHiKER	55.24%	77.62%	84.29%	89.05%	0.67	2.98
D_F^{MN}	Pathidea	8.75%	15.63%	18.75%	42.5%	0.14	7.93
	Agentless	68.52%	89.51%	95.68%	98.15%	0.80	1.79
	LinuxFL+	55.56%	77.78%	84.57%	85.8%	0.67	2.98
	CoHiKER	69.75%	92.59%	98.15%	98.77%	0.81	1.63
D_F^{UN}	Pathidea	0	6.12%	6.12%	10.2%	0.03	10.41
	Agentless	0	4.08%	6.12%	26.53%	0.05	9.84
	LinuxFL+	4.08%	16.33%	24.49%	30.61%	0.12	8.80
	CoHiKER	6.12%	26.53%	36.73%	55.1%	0.20	7.39

Table 5: Method-level Fault Localization Result (RQ2)

Data	Technique	<i>Top@1</i>	<i>Top@3</i>	<i>Top@5</i>	<i>Top@10</i>	<i>MRR</i>	<i>MFR</i>
D^{all}	Agentless	20.95%	37.14%	43.81%	54.29%	0.31	6.94
	LinuxFL+	24.76%	43.33%	48.1%	53.81%	0.35	6.49
	CoHiKER	32.86%	52.86%	58.57%	65.71%	0.43	5.61
D_M^{MN}	Agentless	36.94%	65.77%	75.68%	93.69%	0.54	3.94
	LinuxFL+	45.95%	74.77%	81.89%	89.19%	0.62	3.36
	CoHiKER	57.66%	84.68%	89.19%	93.69%	0.71	2.74
D_M^{UN}	Agentless	3.03%	5.05%	8.08%	10.1%	0.05	10.35
	LinuxFL+	1.1%	8.08%	10.1%	14.14%	0.05	10.03
	CoHiKER	5.05%	17.17%	24.24%	34.34%	0.12	8.87

4.2 Results and Analysis

4.2.1 RQ1: File-Level FL Effectiveness. The results of RQ1 are shown in Table 4. Across D^{all} , LLM-based techniques (CoHiKER, Agentless, and LinuxFL+) substantially outperform the IR-based technique Pathidea, achieving improvements of 551% to 724% in *Top@1*. Among them, CoHiKER consistently delivers the best results. Compared to Agentless, CoHiKER improves *Top@1–Top@10* by 4.50%–12.03%; against LinuxFL+, by 18.08%–26.07%. In terms of ranking metrics, CoHiKER achieves a *MRR* of 0.67 and a *MFR* of 2.98, outperforming both Agentless (*MRR*: 0.63, *MFR*: 3.66) and LinuxFL+ (*MRR*: 0.55, *MFR*: 4.34). These results demonstrate CoHiKER’s effectiveness, setting a new state-of-the-art.

To evaluate explicit fault evidence’s impact, we test on D_F^{MN} and D_F^{UN} . On D_F^{MN} (with faulty files mentioned), all methods perform better, and CoHiKER performs best by leveraging these direct clues. This aligns with expectations, as LLMs can leverage report mentions or stack traces to directly pinpoint likely fault locations. On D_F^{UN} (without faulty files), performance drops for all techniques. Yet, CoHiKER still leads, surpassing LinuxFL+ by 50% in *Top@1* and both baselines by 62.46%–550.25% in *Top@3* and 49.98%–500.16% in *Top@5*. Notably, Agentless fails completely in *Top@1* due to its reliance on surface-level report-code correlations, which fails without explicit mentions. This contrast underscores CoHiKER’s strength in reasoning with indirect evidence.

Statistical analysis confirms CoHiKER’s significance: Friedman test [49] shows significant differences ($\chi^2 = 23.512$, $p < 0.001$), with post-hoc Wilcoxon tests revealing CoHiKER outperforms Agentless ($W = 584.0$, $p = 0.0002$) and LinuxFL+ ($W = 864.5$, $p < 0.001$). These findings confirm that CoHiKER provides statistically significant improvements in file-level FL effectiveness.

Table 6: Token Cost for LLM-based Techniques (RQ3)

FL level	Token cost per case. (Input / Output / Total)		
	LinuxFL+	Agentless	CoHiKER
FL@F	94.3K / 3.61K / 97.9K	0.32M / 2.72K / 0.32M	10.5K / 0.57K / 11.07K
FL@M	0.11M / 4.01K / 0.11M	0.54M / 4.4K / 0.54M	20K / 4.64K / 24.63K

4.2.2 RQ2: Method-Level FL Effectiveness. Table 5 shows RQ2 results. Finelocator and Crashlocator perform poorly (0% and 0.85% *Top@5*), so we omit them from further analysis.

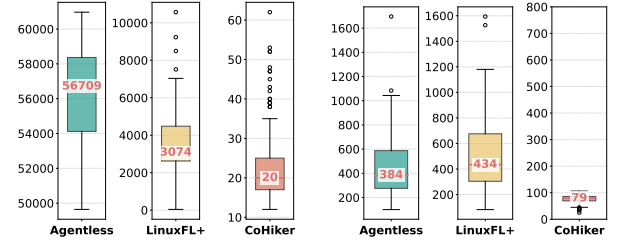
As shown in Table 5, CoHiKER consistently outperforms LLM-based baselines in method-level fault localization, improving *Top@1–Top@10* by 21.04%–56.85% and achieving $MRR=0.43$, $MFR=5.61$ compared to baselines ($MRR: 0.31–0.35$, $MFR: 6.49–6.94$). Though method-level performance is lower than file-level due to finer granularity, CoHiKER maintains a clear lead, indicating its strong capability in effectively leveraging contextual information and reasoning about the root causes of kernel faults.

On D_M^{MN} (with faulty methods explicit), CoHiKER leads across all metrics, improving *Top@1* by 25.48%–56.09%, *Top@3* by 13.25%–28.75%, and *Top@5* by 8.91%–17.85%. On the more challenging D_M^{UN} , CoHiKER still achieves a 66.67%–359.09% gain in *Top@1* (5.05% absolute), with notable improvements in *Top@3–Top@10* (112.50%–240.00%). It ranks the correct method in *Top@10* for 34.34% of cases. This result demonstrates the robustness and reasoning capability of our contrastive reasoning-enhanced LLM approach in handling incomplete or indirect fault evidence.

Finally, a Friedman test reveals a statistically significant difference in method-level performance across the evaluated approaches ($\chi^2 = 43.24$, $p < 0.001$). Follow-up Wilcoxon signed-rank tests confirm that CoHiKER significantly outperforms Agentless ($W = 1077.5$, $p < 0.001$) and LinuxFL+ ($W = 965.0$, $p < 0.001$), indicating our approach’s effectiveness in advancing method-level FL for the Linux kernel.

4.2.3 RQ3: Efficiency Analysis. Columns 1-7 of Table 7 show runtime overhead for CoHiKER and baselines. The Total column reports the overall runtime, and the CR and HCA columns report the time spent by CoHiKER in Contrastive Reasoning (CR) and Hierarchical Context Analysis (HCA), respectively. Since FL@F does not involve method-level localization, the HCA for FL@F includes only the file-level localization time. Based on this table, we can find that CoHiKER is substantially more efficient than the baselines at both FL@F and FL@M. At the more time-consuming FL@M, CoHiKER requires only 332s per case, and achieves speedups of 4.05× and 1.7× over LinuxFL+ and Agentless, respectively. Moreover, the FL@M is significantly more time-consuming than FL@F, indicating that finer-grained analysis incurs higher overhead.

While LLM-based techniques demonstrate promising results compared to traditional approaches, they often incur substantial token usage. To assess this overhead, we analyzed token consumption of different LLM-based techniques. As shown in Table 6, CoHiKER achieves the most efficient token usage. In comparison, LinuxFL+ consumes 8.84× more tokens at FL@F and 4.46× more at FL@M. Agentless incurs the highest overhead, using 28.9× and 21.92× more tokens at FL@F and FL@M, respectively. This efficiency stems from CoHiKER’s candidate identification module, which effectively



(a) Candidates size at FL@F (b) Candidates size at FL@M
Figure 8: Candidates Size of LLM-based Techniques

narrows the candidate set before LLM invocation. We further analyzed the size of the candidate set sizes across LLM-based baselines. The results are shown in Figures 8a and 8b.

From the two figures, it’s clear that CoHiKER examines the smallest candidate set at both FL@F and FL@M (without sacrificing accuracy). Especially in FL@F, CoHiKER achieves a median of only 20 candidates, compared to 56,709 (2,835× more) for Agentless and 3,074 (153× more) for LinuxFL+. This drastic reduction in input size lowers token usage and enables the LLM to focus on relevant contextual information rather than processing excessive noise. Specifically, CoHiKER (using Deepseek-V3) incurs a cost of approximately \$0.0256 and \$0.0771 per case at the file and method levels, respectively, which is acceptable for practical usage.

4.2.4 RQ4: Ablation Study. RQ4 evaluates the contributions of CoHiKER’s components and LLM reasoning through ablation studies from two perspectives. First, from the perspective of components, we designed two variants. $CoHiKER_{w/o}^{CR}$ removes CR and infers potential root causes using only the bug report and failing tests. $CoHiKER_{w/o}^{HCA}$ removes HCA (i.e., Syscall Analysis and Dependencies Expansion in FL@F, and Syntactic/Semantics Filtering in FL@M), while retaining the two stage localization order (file then method). Second, from the perspective of LLM reasoning capability, there are three main stages that rely on LLM. Thus, we ablate each of these stages and design three variants: $CoHiKER_{w/o}^{MU}$, $CoHiKER_{w/o}^{FI}$, and $CoHiKER_{w/o}^{SF}$. Specifically, $CoHiKER_{w/o}^{MU}$ removes test-case mutation in CR and replaces it with the syntax mutations in Syzkaller [50]. $CoHiKER_{w/o}^{FI}$ removes Syscall Analysis and Dependency Expansion in FL@F of HCA, and replaces it with a TF-IDF-based IR model [51] to retrieve files lexically related to the syscall, combined with static dependency analysis for file expansion. $CoHiKER_{w/o}^{SF}$ removes Semantic-aware Filtering in FL@M of HCA, and replaces it with the same TF-IDF model to retrieve methods most textually similar to the bug report. The motivation for designing these variants is to replace LLM-driven semantic reasoning with syntax-based alternatives, thus assessing the advantage of LLMs. We then compare the results of these two classes of ablations.

Contributions of each component: Table 8 (Rows 2-4, 7-9) presents the results of CoHiKER and its two component ablation variants at FL@F and FL@M. It’s evident that removing any component of CoHiKER leads to a decline in localization performance across all metrics. However, the impact of removing CR is larger than that of removing HCA, indicating that inaccurate root-cause reasoning directly undermines localization accuracy. To better understand the source of the performance difference, we further analyze which

Table 7: Runtime Cost Comparison between Different Techniques (RQ3&4)

FL Level	CoHiKER			Baselines			Ablation mutants of CoHiKER				
	CR	HCA	Total	Pathidea	LinuxFL+	Agentless	CoHiKER ^{CR} _{w/o}	CoHiKER ^{HCA} _{w/o}	CoHiKER ^{MU} _{w/o}	CoHiKER ^{FI} _{w/o}	CoHiKER ^{SF} _{w/o}
FL@F	138.5s	10.4s	148.9s	195.8s	1312.0s	549.6s	10.4s (- 93.0%)	144.7s (- 2.8%)	234.4s (+ 57.4%)	149.4s (+ 0.3%)	148.9s (- 0.0%)
FL@M	138.5s	193.5s	332s	-	1345.5s	566.6s	193.5s (- 41.7%)	160.67s (-51.6%)	417.5s (+ 25.8%)	332.5s (+0.2%)	331.2s (-0.2%)

Table 8: Contribution of each Components (RQ4)

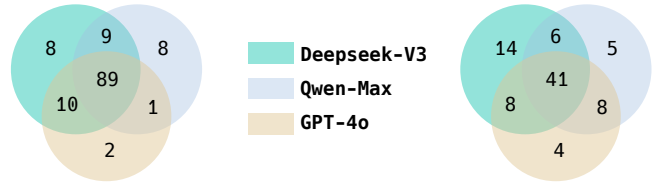
FL Level	Technique	Top@1	Top@3	Top@5	Top@10	MRR	MFR
FL@F	CoHiKER	55.24%	77.62%	84.29%	89.05%	0.67	2.98
	CoHiKER ^{CR}	54.29%	73.33%	77.76%	81.9%	0.64	3.51
	CoHiKER ^{HCA} _{w/o}	47.14%	69.05%	78.57%	86.67%	0.60	3.49
FL@M	CoHiKER ^{MU} _{w/o}	53.33%	73.33%	81.43%	87.14%	0.65	3.31
	CoHiKER ^{FI} _{w/o}	53.81%	73.33%	76.67%	78.10%	0.64	3.66
	CoHiKER ^{SF} _{w/o}	53.81%	73.33%	76.67%	78.10%	0.64	3.66
FL@M	CoHiKER	32.86%	52.86%	58.57%	65.71%	0.43	5.61
	CoHiKER ^{CR}	26.79%	44.98%	50.24%	58.37%	0.37	6.63
	CoHiKER ^{HCA} _{w/o}	27.54%	47.55%	53.43%	65.20%	0.39	6.05
	CoHiKER ^{MU} _{w/o}	28.09%	47.14%	53.81%	62.38%	0.41	5.99
	CoHiKER ^{FI} _{w/o}	29.07%	42.11%	49.28%	58.85%	0.38	6.11
	CoHiKER ^{SF} _{w/o}	25.48%	41.15%	46.89%	54.67%	0.34	6.68

bugs are affected by ablating different components. Following existing work [52], we classify all bugs in the dataset into three categories: concurrency bugs (e.g., deadlocks), semantic bugs (e.g., exception handling), and memory bugs (e.g., null-pointer dereference). We refer readers to [52] for finer-grained definitions. We then analyze how MRR varies across these categories for each variant, as MRR captures a technique’s ability to rank the true faults highly. The results show that removing CR (i.e., CoHiKER^{CR}_{w/o}) reduces MRR by 26.51%, 20.99%, and 17.97% for concurrency, semantic, and memory bugs, respectively. Removing HCA (i.e., CoHiKER^{HCA}_{w/o}) yields smaller drops of 7.02%, 20.67%, and 2.29% for the same categories. Based on this, we can find that CR has a substantial impact across all bug types, highlighting the importance of identifying the root cause. In contrast, HCA has a limited effect on concurrency and memory bugs, which often produce clear crash traces, whereas semantic bugs tend to be more subtle and harder to map to related files. However, these issues can be mitigated by combining the two components of CoHiKER.

Contributions of LLM Reasoning: Table 8 (Rows 5-6, 10-12) presents the result of three CoHiKER variants at FL@F and FL@M, where FL@F does not include CoHiKER^{SF}_{w/o} since it does not involve method-level localization. The results show that removing any stage that uses LLM leads to a drop in localization performance. The impact is larger at FL@M than at FL@F, indicating that finer-grained localization requires stronger reasoning capability. Across stages, the overall trends are similar, but removing the Semantic-aware Filtering (CoHiKER^{SF}_{w/o}) results in a slightly larger drop, e.g., at FL@M, the Top@3 decreases by 11.71% for CoHiKER^{SF}_{w/o}, compared with drops of 5.72% and 10.75% for CoHiKER^{MU}_{w/o} and CoHiKER^{FI}_{w/o}. This indicates that, at FL@M, semantic similarity matching and filtering based on the LLM-inferred root cause play an important role in localization. All these findings suggest that although the LLM-based stages can be replaced by traditional techniques (e.g., syntax-based mutation), LLMs still provide additional gains, particularly for finer-grained (e.g., method-level)

Table 9: Ablation Study on LLM Backends (RQ4)

FL level	LLMs	Top@1	Top@3	Top@5	Top@10	MRR
FL@F	Deepseek-V3	55.24%	77.62%	84.29%	89.05%	0.67
	Qwen-Max	50.95%	71.73%	77.14%	84.29%	0.60
	GPT-4o	48.57%	72.38%	80.00%	83.33%	0.62
FL@M	Deepseek-V3	32.86%	52.86%	58.57%	65.71%	0.43
	Qwen-Max	28.57%	42.38%	52.38%	59.05%	0.38
	GPT-4o	29.05%	39.52%	50%	58.1%	0.37

**(a) FL@F(Top@1)****(b) ML@F(Top@1)****Figure 9: Overlap Analysis with Different LLM Backends**

localization. Moreover, this motivates a deeper examination of how the LLM backend contributes to CoHiKER’s overall effectiveness. Thus, besides Deepseek-V3, we choose other two widely used LLMs, i.e., GPT-4o [53] and Qwen-Max [54], as the comparative LLM backend. Table 9 reports the results of different LLM backends at FL@F and FL@M, and Figure 9 shows their overlap in correctly locating bugs under Top@1. Taken together, the results indicate that: (1) Different LLM backends induce some performance variation, but the overall effectiveness still surpasses existing techniques, which is largely attributed to CoHiKER’s design, and (2) each backend captures different bugs, suggesting that multiple LLM backends can complement each other in practice.

Beyond evaluating effectiveness, we also analyzed the efficiency of all five ablated variants to understand accuracy-runtime trade-offs (Columns 8-12 of Table 7). CoHiKER^{MU}_{w/o} and CoHiKER^{FI}_{w/o} increase runtime, indicating that replacing LLM not only reduces accuracy but also incurs additional overhead, such as generating many invalid mutations. CoHiKER^{SF}_{w/o} similarly brings little runtime benefit. In contrast, fully removing CR (CoHiKER^{CR}_{w/o}) and HCA (CoHiKER^{HCA}_{w/o}) reduces time cost. Together with the results in Table 8, we suggest striking a practical balance between these two variants and CoHiKER in practice.

4.2.5 RQ5: Generalizability on non-kernel systems. Table 10 reports the FL@F and FL@M results of CoHiKER and the baselines on the dataset of non-kernel systems. CoHiKER still achieves the best localization performance at both levels, demonstrating the strength and generality of its design. Another interesting observation is that, compared with the results in Table 8, the localization accuracy on non-kernel systems is much higher at both the file and method levels, indicating that kernel bugs are inherently more difficult

Table 10: Fault Localization Result on Non-kernel Systems

FL-Level	Technique	<i>Top@1</i>	<i>Top@3</i>	<i>Top@5</i>	<i>Top@10</i>	<i>MRR</i>	<i>MFR</i>
FL@F	Agentless	62.04%	78.83%	83.21%	86.86%	0.715	2.91
	SWE-Agent	56.72%	69.40%	73.13%	76.86%	0.637	3.32
	SoapFL	63.49%	80.15%	81.34%	81.34%	0.716	3.11
	CoHiKER	73.36%	89.41%	94.16%	95.62%	0.821	1.89
FL@M	Agentless	51.62%	58.544%	61.38%	62.20%	0.557	5.01
	SWE-Agent	45.52%	64.18%	68.66%	73.13%	0.554	4.4
	SoapFL	49.80%	61.13%	65.18%	67.61%	0.585	4.46
	CoHiKER	54.37%	71.89%	77.37%	84.30%	0.643	3.41

to localize. This highlights the inherent difficulty of kernel fault localization and underscores the significance of our work.

5 Discussion

Like other LLM-based localization methods, CoHiKER may also be affected by data leakage if the studied kernel bugs were used during model pre-training. To mitigate this, we evaluated 21 bug reports dated after March 24, 2025, the release date of the Deepseek-V3 model used in CoHiKER. The results show that CoHiKER performs similarly on these unseen bugs, indicating that data leakage has minimal impact. In the future, we will explore additional ways to reduce data leakage and further deepen kernel FL, such as examining performance differences among baselines and improving localization granularity (e.g., toward line-level).

6 Related Work

6.1 Non-LLM-based Fault Localization

Typical FL techniques include spectrum-based (SBFL), mutation-based (MBFL), information retrieval-based (IRFL), and hybrid techniques. SBFL is one of the most widely studied FL techniques [8, 9], which are broadly applicable to software debugging tasks [55, 56]. SBFL calculates suspiciousness scores for each program element according to the test coverage discrepancy between failing and passing runs via a predefined formula, e.g., Ochiai [55], Dstar[56] and Tarantula[57]. Similar to SBFL, MBFL leverages the concept of mutation testing [10] to generate program variants and scores them based on coverage differences [58, 59]. However, both SBFL and MBFL require runtime coverage data, which is often unavailable in the context of Linux kernel failures. The reason is that execution traces in the kernel are held in the volatile memory, which will be lost upon a system crash [60]. IRFL ranks suspicious locations by textual similarity between bug reports and code [12–14, 61]. For example, Finelocator [13] improves method-level localization by augmenting short method representations with neighbor features for similarity computation. BoostNSift [61] improves fault localization performance by enhancing the quality of bug report titles for more accurate relevance measurement. The most recent Pathidea [14] incorporates the static call graph to aid the construction of the failing execution path, which further improves the representation of textual features from bug reports. Although IRFL techniques can effectively leverage textual information, they often struggle with the complexity of kernel code and the sparsity of symbolic information (e.g., function names) in kernel stack traces. This makes it difficult to accurately match bug reports with relevant code entities, especially in large codebases like the Linux kernel.

6.2 LLM-based Fault Localization

LLMs’ success has boosted their adoption in fault localization. Wu et al. [21] prompt LLMs with code and error logs, and instruct the model to identify the buggy lines, while LLMAO [22] trains lightweight bidirectional adapters on top of LLMs to produce a suspiciousness score for each line. FuseFL [62] further feeds the spectrum information and test execution results to LLMs for identifying suspicious code. These approaches, however, are designed for small code snippets but not suitable for large codebases like the Linux kernel due to their reliance on inputting the source code.

To address LLM input length limitations, AutoFL [42] uses function calling capability to inspect covered methods and pinpoint buggy methods. Similarly, FlexFL [63] first collects SBFL/IRFL candidates, and then utilizes LLM agents to refine the results by analyzing code structures and bug reports. SoapFL [25] leverages LLM agents to iteratively narrow down the search space by analyzing test cases’ dynamically execution paths. As a consequence, they are also inapplicable to localize Linux kernel bugs due to the shared limitations, e.g., unavailable code coverage and sparse symbolic information. The most recent Agentless [19] addresses the above limitations through a recursive fault localization strategy, which identifies suspicious files by iteratively invoking the LLM for search space reduction, making it extremely costly (*ref.* Table 6). LinuxFL+[20], latest research designed for kernel faults, uses LLM agents to identify suspicious files and re-ranks them through incorporating contextual features of code.

Compared to these existing approaches, CoHiKER is specially designed to address the challenges in Linux kernel fault localization. Specifically, it incorporates a contrastive reasoning component for fine-grained root cause identification and search space reduction. It has been proved to be effective in our comprehensive evaluation.

7 Conclusion

In this work, we present CoHiKER, a novel LLM-based fault localization technique tailored for the Linux kernel. By introducing a contrastive reasoning component and a hierarchical feature integration framework, CoHiKER addresses key limitations of prior approaches, i.e., *lack of precise root cause analysis* and *ineffective context utilization*. Our design enables more accurate and efficient fault localization by aligning LLM reasoning with kernel-specific characteristics at both file and method levels. Through a comprehensive evaluation on 210 real-world kernel bugs, CoHiKER outperforms both traditional and state-of-the-art LLM-based baselines in localization accuracy, while significantly reducing token consumption. Specifically, compared to LLM-based baselines CoHiKER improves *Top@1* by up to 26.70% at the file level and 56.85% at the method level, and reduces LLM token usage by up to 8.84× and 28.9×, respectively. Furthermore, CoHiKER generalizes well to the non-kernel dataset, confirming its applicability across diverse systems.

Acknowledgments

This work is supported by National Key Research and Development Program of China (No. 2024YFB4506300), and National Natural Science Foundation of China (Nos. 62322208, 12411530122, 6257234, 62202324 and 62472310).

References

- [1] StackScale. 2024. Linux Kernel Surpasses 40 Million Lines of Code. <https://www.stackoverflow.com/blog/linux-kernel-surpasses-40-million-lines-code>. Accessed: 2025-07-18.
- [2] Hao Sun, Yuheng Shen, Cong Wang, Jianzhong Liu, Yu Jiang, Ting Chen, and Aiguo Cui. 2021. Healer: Relation learning guided kernel fuzzing. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 344–358.
- [3] Bodong Zhao, Zheming Li, Shisong Qin, Zheyu Ma, Ming Yuan, Wenyu Zhu, Zhihong Tian, and Chao Zhang. 2022. {StateFuzz}: System {Call-Based} {State-Aware} linux driver fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*. 3273–3289.
- [4] Chenyuan Yang, Zijie Zhao, and Lingming Zhang. 2025. Kernelgpt: Enhanced kernel fuzzing via large language models. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 560–573.
- [5] Haichi Wang, Ruiguo Yu, Dong Wang, Yiheng Du, Yingquan Zhao, Junjie Chen, and Zan Wang. 2025. An empirical study of test case prioritization on the Linux Kernel. *Automated Software Engineering* 32, 2 (2025), 49.
- [6] Karine Even-Mendoza, Arindam Sharma, Alastair F. Donaldson, and Cristian Cadar. 2023. GrayC: Greybox Fuzzing of Compilers and Analysers for C. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17–21, 2023*, René Just and Gordon Fraser (Eds.). ACM, 1219–1231.
- [7] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740.
- [8] Daming Zou, Jingjing Liang, Yingfei Xiong, Michael D Ernst, and Lu Zhang. 2019. An empirical study of fault localization families and their combinations. *IEEE Transactions on Software Engineering* 47, 2 (2019), 332–347.
- [9] Ming Wen, Junjie Chen, Yongqiang Tian, Rongxin Wu, Dan Hao, Shi Han, and Shing-Chi Cheung. 2019. Historical spectrum based fault localization. *IEEE Transactions on Software Engineering* 47, 11 (2019), 2348–2368.
- [10] Yue Jia and Mark Harman. 2010. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering* 37, 5 (2010), 649–678.
- [11] James H Andrews, Lionel C Briand, Yvan Labiche, and Akbar Siami Namin. 2006. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering* 32, 8 (2006), 608–624.
- [12] Xin Xia and David Lo. 2023. Information Retrieval-Based Techniques for Software Fault Localization. *Handbook of Software Fault Localization: Foundations and Advances* (2023), 365–391.
- [13] Wen Zhang, Ziqiang Li, Qing Wang, and Juan Li. 2019. FineLocator: A novel approach to method-level fine-grained bug localization by query expansion. *Information and Software Technology* 110 (2019), 121–135.
- [14] An Ran Chen, Tse-Hsun Chen, and Shaowei Wang. 2021. Pathidea: Improving information retrieval-based bug localization by re-constructing execution paths using logs. *IEEE Transactions on Software Engineering* 48, 8 (2021), 2905–2919.
- [15] Sanan Hasanov, Stefan Nagy, and Paul Gazzillo. 2024. A Little Goes a Long Way: Tuning Configuration Selection for Continuous Kernel Fuzzing. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 521–533.
- [16] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems* 37 (2024), 50528–50652.
- [17] Yihao Qin, Shangwen Wang, Yiling Lou, Jinhao Dong, Kaixin Wang, Xiaoling Li, and Xiaoguang Mao. 2024. Agentfl: Scaling llm-based fault localization to project-level context. *arXiv preprint arXiv:2403.16362* (2024).
- [18] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury. 2024. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2024)*, M. Christakis and M. Pradel (Eds.). ACM, Vienna, Austria, 1592–1604.
- [19] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489* (2024).
- [20] Zhenhao Zhou, Zhuochen Huang, Yike He, Chong Wang, Jiajun Wang, Yijian Wu, Xin Peng, and Yiling Lou. 2025. Benchmarking and Enhancing LLM Agents in Localizing Linux Kernel Bugs. *arXiv preprint arXiv:2505.19489* (2025).
- [21] Yonghao Wu, Zheng Li, Jie M Zhang, Mike Papadakis, Mark Harman, and Yong Liu. 2023. Large language models in fault localisation. *arXiv preprint arXiv:2308.15276* (2023).
- [22] Aidan ZH Yang, Claire Le Goues, Ruben Martins, and Vincent Hellendoorn. 2024. Large language models for test-free fault localization. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12.
- [23] Alex Mathai, Chenxi Huang, Petros Maniatis, Aleksandr Nogikh, Franjo Ivančić, Junfeng Yang, and Baishakhi Ray. 2024. Kgym: A platform and dataset to benchmark large language models on linux kernel crash resolution. *Advances in Neural Information Processing Systems* 37 (2024), 78053–78078.
- [24] Rongxin Wu, Hongyu Zhang, Shing-Chi Cheung, and Sunghun Kim. 2014. Crashlocator: Locating crashing faults based on crash stacks. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 204–214.
- [25] Yihao Qin, Shangwen Wang, Yiling Lou, Jinhao Dong, Kaixin Wang, Xiaoling Li, and Xiaoguang Mao. 2025. SoapFL: A Standard Operating Procedure for LLM-based Method-Level Fault Localization. *IEEE Transactions on Software Engineering* (2025).
- [26] 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*. <https://openreview.net/forum?id=VTF8yNQm66>
- [27] Syzbot. 2025. Kernel Crash: WARNING in *unmap_page_range*. <https://syzkaller.appspot.com/bug?extid=7ca4b2719dc742b8d0a4>. Accessed: 2025-07-18.
- [28] DeepSeek. 2025. DeepSeek-v3.
- [29] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. 2020. Deep learning library testing via effective model generation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 788–799.
- [30] Yingquan Zhao, Zan Wang, Junjie Chen, Mengdi Liu, Mingyuan Wu, Yuqun Zhang, and Lingming Zhang. 2022. History-driven test program synthesis for JVM testing. In *Proceedings of the 44th International Conference on Software Engineering*. 1133–1144.
- [31] Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C Desmarais. 2024. Effective test generation using pre-trained large language models and mutation testing. *Information and Software Technology* 171 (2024), 107468.
- [32] Reinhard Tartler, Christian Dietrich, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2014. Static analysis of variability in system software: The 90,000# ifdefs issue. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. 421–432.
- [33] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. 2006. Mining metrics to predict component failures. In *Proceedings of the 28th International Conference on Software Engineering*. 452–461.
- [34] Maurice H Halstead. 1977. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc.
- [35] Frank Rosenblatt. 1958. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review* 65, 6 (1958), 386.
- [36] Tong Sun, Yao Shao, Xiaoxiao Li, Liang Zhang, Yabo Yang, and Jie Zhou. 2020. Learning Sparse Sharing Architectures for Multiple Tasks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 8936–8943.
- [37] Jeongju Sohn and Shin Yoo. 2017. Flucss: Using code and change metrics to improve fault localization. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 273–283.
- [38] Ritu Kapur and Balwinder Sodhi. 2018. Estimating defectiveness of source code: A predictive model using github content. *arXiv preprint arXiv:1803.07764* (2018).
- [39] Luca Pascarella, Fabio Palomba, and Alberto Bacchelli. 2019. Fine-grained just-in-time defect prediction. *Journal of Systems and Software* 150 (2019), 22–36.
- [40] Mijung Kim, Jaechang Nam, Jongwook Yeon, and Sunghun Kim. 2015. REMI: Defect prediction for efficient API testing. (2015), 990–993.
- [41] 2025. Syzbot. <https://syzkaller.appspot.com>.
- [42] Sungmin Kang, Gabin An, and Shin Yoo. 2024. A quantitative and qualitative evaluation of LLM-based explainable fault localization. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1424–1446.
- [43] Ellen M Voorhees et al. 1999. The trec-8 question answering track report.. In *Trec*, Vol. 99. 77–82.
- [44] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. 2019. DeepFL: Integrating Multiple Fault Diagnosis Dimensions for Deep Fault Localization. (2019), 169–180.
- [45] Y. Li, S. Wang, and T. Nguyen. 2021. Fault localization with code coverage representation learning. (2021), 661–673.
- [46] Chris Parnin and Alessandro Orso. 2011. Are automated debugging techniques actually helping programmers?. In *Proceedings of the 2011 international symposium on software testing and analysis*. 199–209.
- [47] Junjie Chen, Jiaqi Han, Peiyi Sun, Lingming Zhang, Dan Hao, and Lu Zhang. 2019. Compiler bug isolation via effective witness test program generation. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 223–234.
- [48] anonymity. 2025. Public repository for CoHiker. <https://doi.org/10.5281/zenodo.17569818>.
- [49] Janez Demšar. 2006. Statistical Comparisons of Classifiers over Multiple Data Sets. *Journal of Machine Learning Research* 7 (2006), 1–30.
- [50] Google. 2015. syzkaller: an unsupervised coverage-guided kernel fuzzer. <https://github.com/google/syzkaller>. Accessed: 2025-11-09.
- [51] Gerard Salton and Christopher Buckley. 1988. Term-weighting approaches in automatic text retrieval. *Information processing & management* 24, 5 (1988), 513–523.
- [52] Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and Chengxiang Zhai. 2014. Bug characteristics in open source software. *Empirical software*

- engineering* 19, 6 (2014), 1665–1705.
- [53] OpenAI. 2024. Hello GPT-4o.
 - [54] qwenlm. 2024. Qwen-Max.
 - [55] Rui Abreu, Peter Zoetewij, and Arjan JC Van Gemund. 2007. On the accuracy of spectrum-based fault localization. In *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE, 89–98.
 - [56] W Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. 2013. The DStar method for effective software fault localization. *IEEE Transactions on Reliability* 63, 1 (2013), 290–308.
 - [57] James A Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. 273–282.
 - [58] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. 2014. Ask the mutants: Mutating faulty programs for fault localization. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 153–162.
 - [59] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: mutation-based fault localization. *Software Testing, Verification and Reliability* 25, 5-7 (2015), 605–628.
 - [60] Tegawendé F Bissyandé, Laurent Réveillère, Julia L Lawall, and Gilles Muller. 2012. Diagnosys: automatic generation of a debugging interface to the linux kernel. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 60–69.
 - [61] Abdul Razzaq, Jim Buckley, James Vincent Patten, Muslim Chochlov, and Ashish Rajendra Sai. 2021. BoostNSift: A query boosting and code sifting technique for method level bug localization. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 81–91.
 - [62] Ratnadira Widyasari, Jia Wei Ang, Truong Giang Nguyen, Neil Sharma, and David Lo. 2024. Demystifying Faulty Code with LLM: Step-by-Step Reasoning for Explainable Fault Localization. *arXiv preprint arXiv:2403.10507* (2024).
 - [63] Chuyang Xu, Zhongxin Liu, Xiaoxue Ren, Gehao Zhang, Ming Liang, and David Lo. 2025. Flexfl: Flexible and effective fault localization with open-source large language models. *IEEE Transactions on Software Engineering* (2025).