FAULTSEEKER: LLM-Empowered Framework for Blockchain Transaction Fault Localization

Kairan Sun*, Zhengzi Xu†§, Kaixuan Li*, Lyuye Zhang*, Yuqiang Sun*, Liwei Tan‡, and Yang Liu*
* Nanyang Technological University, Singapore
† Imperial College London, Imperial Global Singapore, Singapore
‡ MetaTrust Labs, Singapore

***Abstract*—Web3 applications, particularly decentralized finance (DeFi) protocols, have grown rapidly with over $100 billion locked in smart contracts, attracting sophisticated attacks causing billions in losses. When attack occur, security analysts need to perform fault localization to identify vulnerable functions and understand attack vectors. This critical process currently requires an average of 16.7 analyst hours per incident due to complex blockchain execution models, rapidly evolving protocol interactions, and multi-contract attack patterns that exceed existing analytical capabilities. Despite its critical importance, blockchain fault localization has received limited attention due to fundamental challenges requiring semantic understanding of economic models and protocol-specific logic. Existing blockchain-specific tools target only single vulnerability types, while the only comprehensive solution, DAppFL, relies on machine learning model that may miss sophisticated exploits and lacks interpretability in results. Recent advances in large language models (LLMs) demonstrate remarkable code comprehension capabilities, but existing applications focus on proactive vulnerability detection with minimal exploration of post-incident fault localization.**

We present FAULTSEEKER, an LLM-empowered framework for blockchain transaction fault localization. Our two-stage architecture combines transaction-level forensics for strategic scoping with coordinated specialist agents for sustained reasoning. This design provides long-term memory management via orchestrator agents and specialized attention allocation through coordinated workers, enabling comprehensive analysis across complex multi-contract transactions without context loss. We evaluate FAULTSEEKER on a compiled dataset of 115 real-world malicious transactions with expert-validated annotations spanning diverse attack patterns and complexity levels. Results demonstrate that FAULTSEEKER significantly outperforms existing approaches, including DAppFL and leading native LLMs (GPT-4o, Claude 3.7 Sonnet, DeepSeek R1), while maintaining practical efficiency (4.4-8.6 minutes) and cost-effectiveness ($1.55-$4.53 per transaction).

***Index Terms*—Smart Contract Security, Blockchain Transaction Analysis, Fault Localization**

I. INTRODUCTION

Web3 applications, particularly decentralized finance (DeFi) applications, have grown exponentially in both complexity and economic significance, with over $100 billion locked in smart contracts as of April 2025 [3]. Accompanying this growth is a sharp increase in sophisticated attacks, resulting in billions of dollars in losses [32]. When such attack incidents occur, security analysts need to dissect malicious transactions, identify exploited vulnerabilities, and understand the attack vectors. This post-incident analysis is vital not only to contain financial damage but also to prevent vulnerability propagation, preserve market stability, and maintain user trust.

Despite its critical role, post-incident analysis of smart contract attacks remains a labor-intensive process. Fault localization, in particular, often takes analysts hours or even days to trace the root cause. Recent research confirms this inefficiency, showing that analysts spend an average of 16.7 hours manually localizing faults in smart contract incidents [41]. This inefficiency comes from several persistent challenges in smart contract domain. First, fault localization demands extensive domain knowledge of blockchain economic models and protocol-specific logic, posing a steep learning curve for less experienced analysts. Second, the DeFi ecosystem evolves rapidly, introducing new protocols, composability patterns, and novel attack strategies at an unprecedented pace. Keeping up with this dynamism is difficult, especially under the time pressure of active incident response. Third, many attacks span multiple protocols and contracts, producing intricate execution chains. Reconstructing these flows across system boundaries is mentally taxing and significantly delays mitigation. Together, these challenges highlight an urgent need for tools that can automate, or at least substantially accelerate, the fault localization of smart contract exploits.

While automated fault localization has been actively studied in traditional software analysis [23], [24], [26], [30], limited attempts have been made in the smart contract domain as the challenges discussed above pose significant barriers to automation. Existing approaches exploring this task tend to only target on single vulnerability types such as Reentrancy [39], [44]. The current state-of-the-art tool, DAppFL [41], represents a step forward by integrating fund flow analysis with cryptocurrency information in a learning-based model. However, DAppFL still faces key limitations affecting fault localization effectiveness. It relies on fund flow graph construction with cryptocurrency information integration, which may introduce inaccuracies that propagate as noise into the learning-based model. Additionally, fund flow captures financial outcomes but cannot reveal the exploitation logic needed to distinguish the faults behind the outcomes. Moreover, DAppFL provides ranked functions without explanatory evidence, hindering security analysts who need to validate findings. These limitations indicate a gap between current automated tools and the analytical requirements for blockchain exploit investigation.

Recent breakthroughs in large language models (LLMs)

§Zhengzi Xu is the corresponding author (z.xu@imperial.ac.uk).

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00017

have demonstrated remarkable capabilities in code comprehension [22], [28] and vulnerability detection [12], [27]. In the smart contract domain, existing LLM-based approaches [34], [45] have primarily focused on vulnerability detections through static code analysis, achieving notable success in identifying potential security flaws before deployment. However, little exploration has been conducted on applying LLMs to post-incident analysis such as fault localization, despite their capabilities to address current approach limitations. LLMs possess code comprehension capabilities that enable the analysis of transaction execution traces to understand why involved function calls can exploit potential vulnerabilities, while fund flow representations only capture the resulting token transfers. Additionally, LLMs provide explanatory evidence for their findings rather than unexplained rankings, addressing the interpretability gap in current automated approaches.

In this paper, we present FAULTSEEKER, an LLM-empowered framework that addresses the challenges of blockchain transaction fault localization through progressive understanding and coordinated reasoning. Our approach is motivated by the observation that while LLMs demonstrate strong code comprehension capabilities, their direct application to complex blockchain transactions faces critical limitations including memory constraints and attention degradation across lengthy multi-contract execution sequences. Inspired by memory and attention mechanisms in cognitive science [29], FAULTSEEKER employs a two-stage framework combining transaction-level forensics for strategic scoping with task-driven function analysis through coordinated specialist agents. This design provides long-term memory management via orchestrator agents that maintain global context across extended reasoning chains, and specialized attention allocation through coordinated workers that focus analytical capacity on specific problem aspects. By mimicking how human expert perform fault localization task, FAULTSEEKER maintains high-level situation awareness while investigating technical details.

We conduct comprehensive evaluation on a dataset of 115 real-world malicious transactions spanning diverse attack patterns and complexity levels with expert-validated ground truth annotations. Our experimental results demonstrate that FAULTSEEKER significantly outperforms existing state-of-the-art approaches, including DAppFL [41] and leading native LLMs (GPT-4o [6], Claude 3.7 Sonnet [2], DeepSeek R1 [17]), across all complexity scenarios with substantial performance margins. We conduct ablation studies to validate the necessity of each component in the framework, with a case study demonstrating the practical application of FAULTSEEKER. The main contributions of this work include:

- **Extended Benchmark Dataset.** We extend the DAppFL dataset to 115 real-world malicious transactions with fault annotations and complexity-based classifications.
- **Fault Localization Framework.** We propose FAULTSEEKER, a multi-agent framework for automated fault localization in blockchain transactions. The source code is available in our repository [8].
- **Native LLM Limitation Analysis.** We provide empirical analysis on the performance of leading LLMs and fundamental challenges LLMs face when performing blockchain transaction fault localization tasks .
- **Comprehensive Framework Evaluation.** We demonstrate the effectiveness of FAULTSEEKER through systematic comparison with state-of-the-art tools and ablation studies quantifying the contribution of each component.

II. BACKGROUND

A. DeFi Security Incident Analysis and Response

When exploit occurs, current industry practice relies heavily on specialized security teams manually inspecting transaction traces, which is a labor-intensive process with the increasing complexity of modern exploits that leverage multiple protocols, flash loans, and price manipulations to extract value. As discussed in existing work [41], the fault localization consumes an average of 16.7 analyst hours per incident, presenting the primary bottleneck in the response workflow. Regarding adversarial inputs or obfuscated transactions during fault localization, successful attacks require effective execution on the blockchain, making extensive obfuscation counterproductive since it could compromise attack functionality. The analyzed transaction traces therefore represent execution paths that achieved their malicious objectives. While attackers may attempt to mislead analysis through variable naming or code structure obfuscation, the core execution patterns remain observable in transaction traces.

B. Transaction Analysis in Blockchain Systems

Transaction analysis in blockchain systems examines execution traces, state changes, and fund flows, with current tools like Phalcon [13] and Tenderly [37] providing trace reconstruction but limited semantic interpretation of transaction intent or vulnerability identification. Complex DeFi interactions often generate thousands of low-level operations across multiple contracts, making manual analysis time-consuming and error-prone. These limitations necessitate automated approaches for effective fault localization in blockchain transactions.

C. Multi-Agent Systems

An *agent* is an autonomous entity that can perceive the environment and process information, and act to achieve specific objectives [40]. *Multi-agent systems* consist of multiple interacting agents that coordinate to solve complex problems beyond individual capabilities [38]. Multi-agent architectures enable division of labor through specialized agents focused on specific problem aspects while maintaining coordination through communication protocols. This approach has demonstrated effectiveness in various code-related tasks [18], [19], [35], motivating our exploration of coordinated specialist agents for blockchain transaction fault localization, where complex analytical requirements exceed individual model capabilities.

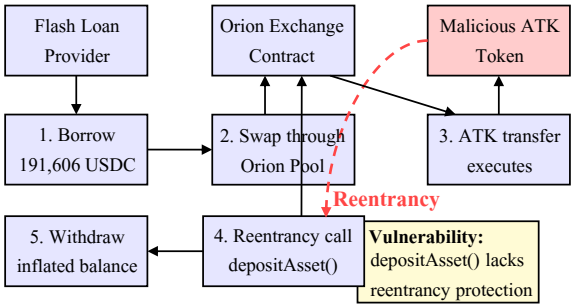

Fig. 1: Orion Protocol reentrancy attack flow.

D. Motivating Example

To illustrate the limitations of existing fault localization approaches and motivate our framework, we use the Orion Protocol exploit [16] as a motivating example, which occurred in February 2023 and resulted in a $3 million loss.

Incident Description. Figure 1 illustrates the Orion Protocol exploit involving a reentrancy attack. The attacker created a malicious token (ATK) and obtained a flash loan. The attacker then initiated a swap through `swapThroughOrionPool()` using the ATK token. During swap execution, the ATK token's `transfer()` function reentered the protocol by calling the unprotected `depositAsset()` function. This allowed the attacker to deposit the flash-borrowed funds and exploit the balance calculation logic to extract profits.

Why Existing Tools Fail. DAppFL applies heterogeneous graph neural networks (HGNNs) to learn node features and rank function suspiciousness, but incorrectly identifies `swap` functions as most suspicious while missing the vulnerable `depositAsset()` function entirely. This potentially occurs because the learning-based model may prioritize functions with fund flow patterns, causing token transfer operations to receive higher suspiciousness scores. Additionally, DAppFL provides ranked results without explanatory evidence, requiring human experts to manually understand attack flows and validate findings. This defeats the purpose of automated fault localization by forcing the same time-intensive analysis that such tools aim to eliminate.

Our Approach. Our approach identifies the vulnerable `depositAsset()` function by analyzing transaction execution traces rather than static graph features. LLMs enable the understanding of cross-contract interactions and context-dependent vulnerabilities through code comprehension and semantic reasoning. Our approach analyzes exploit logic to identify suspicious patterns, then validates findings through source code examination. This improves fault localization accuracy and provides interpretable explanations, addressing the limitations demonstrated by existing approaches. We have detailed how our approach performs the fault localization task in this case in the later evaluation (Section IV-H).

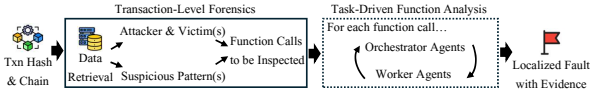

Fig. 2: Overview of our approach.

III. Approach

A. Overview

As illustrated in Figure 2, our approach introduces a two-stage pipeline to complete fault localization tasks: transaction-level forensics and task-driven function analysis. This approach operates on the principle of progressive understanding [21]. It starts with broad transaction characteristics and iteratively refines its analysis toward precise fault localization, mimicking how human experts perform fault localization tasks.

Transaction-level forensics is inspired by the analysis scoping process performed by human experts. When analyzing complex transactions, human experts often begin by explicitly deciding which parts of the transaction to investigate deeply based on initial observations, rather than attempting to analyze all components with equal depth. Our forensic stage leverages this insight by first identifying key addresses in the transaction through LLM-driven reasoning applied to basic transaction summaries, distinguishing potential attackers, victims, and intermediaries by their behavioral patterns. In addition, our approach detects suspicious interaction patterns that are frequently associated with exploits, such as repeated function calls involving token transfers, unusual sequencing of operations, or transactions designed to borrow large amounts of capital. By recognizing these characteristic patterns, the system generates a prioritized list of function calls for further investigation. These functions are likely to contain the entry points that trigger the targeted vulnerability.

With the narrowed scope for further investigation, our approach proceeds to the second stage, task-driven function analysis. This stage implements a multi-agent architecture consisting of two types of agents: orchestrator agents and specialist worker agents. Orchestrators maintain global context, manage the analysis workflow, and decompose complex investigations into specific analytical tasks. They continuously track analysis progress and adaptively maintain a task tree based on emerging insights. Workers work together to complete the execution of a specific task including task generation, data retrieval, and task execution. This design mirrors the cognitive processes of human experts, where a high-level situation awareness is maintained while diving into specific technical details.

B. Transaction-Level Forensics

The workflow of transaction-level forensics is presented in Figure 3. During this stage, multi-dimensional transaction data will first collected, after which strategic analysis scoping proceeds with the support of LLM-driven reasoning and a precompiled checklist of suspicious patterns.

1) Basic Information Collection

To enable access to complementary views of a transaction, our approach begins by collecting three distinct categories of transaction information. These information categories align with those provided by Phalcon Explorer [4], a popular transaction analysis platform widely used by human experts.

Transaction Information. We retrieve basic transaction information through standard blockchain API calls [7], including execution status, block number, timestamp, sender address, and recipient address. This basic contextual information establishes the temporal and network positioning of a transaction. They can serve as an effective first filter for distinguishing normal transactions from potentially malicious ones and identifying suspicious activity patterns [14], [31].

Token Flow. Token flow includes transfer amounts, source and destination addresses, and token types during transaction execution. This financial data helps the identification of potential victims (addresses with unexpected losses) and attackers (addresses with disproportionate gains). We collect token transfers from blockchain explorers like Etherscan, which provide transfer records through accessible APIs [7].

Invocation Flow. Invocation flow represents the hierarchical sequence of function calls within a transaction, including parameters, return values, and call depths. This execution trace enables understanding of exploit logic through detailed call patterns and inter-contract interactions. We reconstruct invocation flow using the Foundry framework [5], which can generate complete execution traces by replaying the specified transaction hash with blockchain RPC url endpoints (remote procedure calls to provide access to the historical state data).

2) Analysis Scoping

With the collected information, our approach proceeds to strategic analysis scoping through a two-step process.

First, we identify potential attackers and victims by examining the relationship between transaction initiators and addresses with significant balance changes. This initial actor identification establishes a preliminary attack hypothesis that narrows the investigative focus to relevant transaction components. To understand attack strategies and prioritize investigation efforts, we leverage LLM to identify operational roles (e.g., flash loan providers and lending protocols) of involved addresses. This role identification enables rapid attack type inference. For example, flash loan presence suggests potential price manipulations, while lending protocol involvement indicates potential liquidation or collateral attacks. It also guides investigation prioritization by focusing on newer protocols and complex interactions over established stablecoins and mature protocols. The prompt used is as follows:

> You are a smart contract security expert excels at analyzing malicious transactions. Given an address and its function calls, identify possible operational roles of the address (e.g., flashloan, lending, stablecoin). Return possible roles in a list. If uncertain, return [unknown] only. Do not include explanations or any other text.

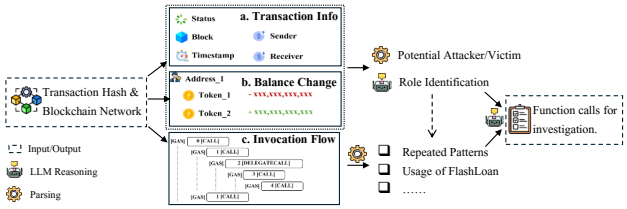

Fig. 3: Workflow of transaction-level forensics.

Second, we evaluate the transaction trace against a precompiled checklist of suspicious patterns developed through systematic analysis of attack transactions in our dataset. This process involved manually reviewing each attack transaction to identify the initial function calls that enable access to fault and summarize patterns based on execution characteristics and positioning within the call hierarchy. To avoid bias, we invite two security experts independently analyze the transactions and resolve disagreements through discussion. Note that this is a one-time effort since the patterns generalize across attack types and can be applied automatically to new transactions. The resulting checklist includes: (1) repeated function calls at the same call depth, indicating iteration-based exploits; (2) nested repeated calls between contracts, characteristic of reentrancy vulnerabilities; (3) function calls immediately following large capital acquisition through flashloans, typical of price manipulation attacks; and (4) function calls related to token transfer or state write with suspicious parameters. These patterns are translated to code-based rules available in our code repository [8], serving as relaxed scoping to identify function calls for further analysis. These function calls are then ranked by LLMs based on contextual information including matched suspicious patterns, function parameters, call frequencies, and execution positioning. LLMs prioritize functions most likely to contain exploited faults for subsequent analysis.

C. Task-Driven Function Analysis

After narrowing the scope through transaction-level forensics, our approach proceeds to task-driven function analysis with a multi-agent architecture. As depicted in Figure 4, the proposed architecture consists of five LLM agents operating on structured function call information summaries. This multi-agent design addresses the challenge that analyzing transactions with numerous child calls and diverse execution parameters can overwhelm a single agent, making it difficult to maintain focus and trace reasoning processes. These agents are organized into two complementary categories: knowledge orchestration agents that maintain global analytical context, and function call investigation agents that perform specialized tasks. This design enables systematic fault localization while preserving both analytical depth and investigative efficiency.

1) Function Call Information Summary

Given a function call to be investigated, our approach structures information into three categories essential for transaction analysis as follows.

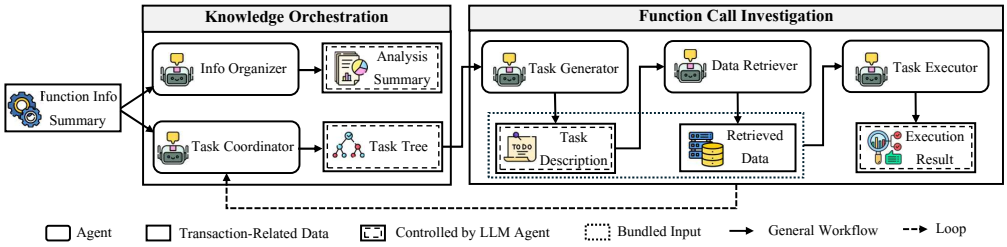

Fig. 4: Workflow of task-driven function analysis.

Execution metadata provides execution information for fault localization analysis and is extracted from invocation flow data. It includes: 1) Gas consumption shows computational costs that may reveal resource-intensive operations. 2) Call type classification implies distinct state operations. Specifically, *delegatecall* modifies the state of the caller while *staticcall* enforces read-only operations that cannot modify state, and *call* modifies the state of callee. 3) Caller address helps to establishes the origin of function call. 4) Call depth indicates the position of the function call in the invocation flow. This determines whether the current function is a child, sibling, or parent relative to previously analyzed functions, maintaining proper context during progressive analysis.

Function behavior data are derived from invocation flow, including: 1) concrete runtime inputs of the function call, exposing potential exploits through validation bypass or anomalous values, and 2) child call summaries that include function names, call types, occurrence frequencies, and hierarchy positions. Complete child call information with runtime inputs, nested calls, and source code creates excessive data volume that overwhelms LLMs, so we extract detailed information only when required for further investigation.

Code-level details include source code retrieved through available tool [1], enabling correlation between runtime behavior and actual implementation logic for fault localization. This provides code implementation-level evidence when available to validate identified suspected faults.

These information are structured into JSON format and provided to LLM agents through prompts as input data.

2) Knowledge Orchestration

Within our multi-agent architecture, knowledge orchestration agents maintain high-level situation awareness throughout the in-depth function call analysis process. They are designed to integrate findings into a coherent analytical model (information organizer) while ensuring systematic investigation progress (task coordinator). These agents operate at the global context level, managing the overall investigation strategy while specialist workers perform targeted analytical tasks.

The primary task of the information organizer is to maintain an evolving understanding of the transaction through continuous knowledge integration. Initially, it develops a high-level overview based on transaction-level forensics results. As in-depth function investigations proceed, this agent continuously integrates new investigation results to refine and update its understanding of the overall attack scenario. The agent identifies connections between different function behaviors that collectively constitute the exploit mechanism. Additionally, it tracks functions where faults may be located, maintaining a prioritized list of fault candidates based on accumulated evidence from ongoing investigations. This evolving understanding enables the system to recognize complex attack patterns that span multiple function interactions.

```
1. Basic Function Call Info Gathering - [completed]
    1.1 Summary of appearance in the transaction - [completed]
    1.2 Summary of the relation with other addresses - [completed]
    1.3 Retrieve the source code of the function call - [completed]
2. Detailed Analysis of Function Calls - [to-do]
    2.1 Analyze `74a97af6` Function Call - [to-do]
        2.1.1 Analyze `74a97af6` call with gas 128649 and params - [to-do]
            2.1.1.1 Analyze balance changes related to `74a97af6` - [to-do]
            2.1.1.2 Examine the logic for potential vulnerabilities- [to-do]
    2.2 Analyze High Appearance Count Children Calls - [to-do]
        2.2.1 Investigate `fallback` function at <Address> - [to-do]
        2.2.2 Investigate `swapExactAmountIn` at <Address> - [to-do]
```

Fig. 5: Example of task tree maintained by task coordinator.

The task coordinator manages the investigation workflow by maintaining and dynamically updating a hierarchical task tree structure. Figure 5 presents an example of the task tree generated by the task coordinator, where the initial task (i.e., Basic Function Call Info Gathering) is predefined to establish the expected task structure. The task tree tracks the status of all tasks as completed or to-do, organizing them in a hierarchical structure where high-level analytical objectives are decomposed into specific investigative subtasks. This hierarchical organization ensures systematic progression from broad function call characterization to detailed function call analysis. The result of the predefined initial task refers to the collected function call information summary (Section III-C1). With this summary, the coordinator generates an initial task tree based on function characteristics, prioritizing tasks according to their potential for fault localization. When new investigation results are provided by function call investigation agents, the task coordinator updates the task tree to reflect new priorities and exploration paths. This dynamic task management enables systematic coverage of potential vulnerability scenarios while avoiding unnecessary analysis of irrelevant function behaviors.

By coupling comprehensive knowledge integration with adaptive task prioritization, we ensure that the specialized worker agents operate within a coherent analytical framework, focusing their efforts on specific function behavior.

3) Function Call Investigation

The function call investigation component performs detailed analysis of specific function calls through three specialized worker agents. The task generator selects prioritized subtasks from the task tree and generates detailed task descriptions. The data retriever interprets these task descriptions and retrieves essential data required for task completion. The task executor performs the specified analysis on the retrieved data following the task description. This coordinated workflow enables systematic function call investigation with the goal of fault localization while maintaining clear separation of responsibilities among worker agents.

The task generator initiates each investigation cycle by selecting a high-priority task from the task tree maintained by the task coordinator. For the selected task, the generator generates a detailed task description that specifies the precise analytical objective, such as identifying parameter validation weaknesses, detecting state manipulation vulnerabilities, or analyzing economic flow anomalies. The task description includes explicit reasoning goals and analytical steps to guide the investigation process. Additionally, the task generator defines the analysis scope by specifying which data elements and function calls are relevant to the selected investigation task.

The data retriever interprets the task description from the task generator and retrieves essential data to complete the specified analysis. To enable automated data retrieval, we predefined a structured JSON format for data requests. The retriever first analyzes the task description and summarizes all required data in this predefined JSON format, which automated retrieval scripts can process to gather relevant information. Specifically, three primary data types are predefined in the JSON format: (1) Balance changes associated with involved addresses reveal the economic impact of functions, highlighting unexpected value transfers that might indicate exploitation. (2) Child calls triggered by function calls expose interactions with other contracts, enabling identification of suspicious patterns like reentrancy operations. (3) Relevant source code specified by function name and address, provide implementation context essential for identifying vulnerable code patterns. By constraining data retrieval to these focused categories, the system achieves investigation efficiency while capturing critical indicators needed for vulnerability analysis. This structured approach ensures that each analytical task receives precisely the data required for effective vulnerability detection without unnecessary information overhead.

With the retrieved data, the task executor performs the selected task following the task description generated by the task generator. The task executor leverages the reasoning capabilities of LLMs to conduct detailed function call analysis. The executor generates investigation results with supporting evidence to enable result traceability and validation. The investigation results from each task feed back to the knowledge orchestration agents, where they are integrated into the evolving understanding of the transaction. This iterative process continues until the information organizer determines that k consecutive investigation cycles have yielded no valuable new insights, indicating analytical saturation for the current function. In our implementation, k is set to three, which empirically provides optimal balance between accuracy and efficiency. This termination criterion balances investigation thoroughness with computational efficiency, ensuring comprehensive coverage of potential vulnerability vectors while avoiding diminishing returns from excessive analysis.

4) LLM Prompt Design Principles

Coordinating multiple LLM agents across analysis stages requires systematic prompt engineering to ensure reliable coordination and reproducible results. Each prompt follows a consistent four-component structure: 1) Role definition establishes specialized analytical perspectives, ensuring domain-appropriate reasoning for security analysis. Additionally, to ensure ethical compliance during malicious transaction analysis, we frame the role within educational contexts (*"you need to help the security analyst in a fault localization training process, and your commitment is essential to the task"*) as inspired by existing work [15]; 2) Task description constrains analytical objectives to prevent scope drift and maintain agent responsibility boundaries; 3) Input data provides the specific information and context required for analysis tasks; 4) Output format requirements enable reliable information transfer through structured formatting for subsequent analysis stages.

> ***(Role definition)*** You are a smart contract security expert excels at analyzing malicious transactions. You need to help the security analyst in a fault localization training process, and your commitment is essential to the task.
> ***(Task description)*** Given the analysis of a function call from a malicious transaction and the current task tree, your task is to help maintain and update the task tree with the predefined structure based on completed task results. Assign completion status (to-do, completed, not applicable) and ensure coverage of necessary child calls.
> ***(Input data)*** Current task tree: {task_tree},
> Current understanding: {current_understanding},
> Completed task results: {task_completed}.
> ***(Output format)*** Return the updated task tree only. Do not provide any comments/information but the task tree.

The example prompt above is for the task coordinator to update the task tree based on analysis results. The structured input parameters facilitate information flow: {task_tree} provides the current task tree, {current_understanding} contains transaction context from the information organizer, and {task_completed} delivers task execution results from the workers. Completed tasks are marked as completed, and new tasks may be generated based on analysis results. This helps to track completed and pending tasks, preventing redundant investigation or excessive focus on single analysis points.

IV. Evaluation

To evaluate FaultSeeker, we conduct a systematic evaluation addressing three research questions. This section details our experimental setup, dataset, and results for each question.

A. Research Questions

In this section, we show the performance of FAULTSEEKER and its comparison with existing approaches. We aim to answer the following research questions:

RQ1: How does the performance of FAULTSEEKER compare with that of existing SOTA?

RQ2: How does the performance of FAULTSEEKER compare with that of native LLM models?

RQ3: How effectively does each component contribute to the overall performance of FAULTSEEKER?

RQ4: What are the efficiency and cost characteristics of FAULTSEEKER in real-world scenario?

RQ5: What is the real-world applicability of FAULTSEEKER within professional security analysis workflows?

B. Implementation Details

The prototype of FAULTSEEKER is built with Python, comprising over 10k lines of code across 28 files. The implementation leverages GPT-4o [6] model by OpenAI for all LLM-based reasoning tasks. The choice to evaluate using a single model within our framework is because we focus on architectural effectiveness rather than comparative model performance. By maintaining consistent model selection across all agents, we isolate the contribution of our framework innovations from variations in underlying model capabilities. The code implementation can be found in our repository [8].

C. Dataset Composition

Our dataset comprises 115 malicious transactions collected from reported Web3 security incidents before February 2025. The dataset construction process begins with collecting 84 transactions from DAppFL [41] and 381 from DeFiHackLabs [10], with 9 overlappings identified and removed to 456 unique transactions. Three key selection criteria applies to ensure evaluation quality and reproducibility. First, all involved smart contracts are implemented in Solidity or Vyper, the two most popular contract programming languages (all satisfy). Second, all transactions are recorded on Ethereum or BNB Chain, the two most popular DApp platforms (369 transactions pass). Third, each transaction includes expert-validated fault annotations that serve as ground truth for evaluation. These annotations are established through consensus among blockchain security experts (115 transactions pass). Detailed statistics can be found on our webpage [8].

To further assess the performance of our approach, we classify transaction complexity based on gas usage. Higher gas usage often implies multiple cross-contract interactions and substantial state manipulations, indicating greater localization challenges. We adopt established gas usage ranges [11], [36] to categorize into three complexity levels: moderate, complex, and extreme (exceptionally complex). As summarized in Table I, most malicious transactions fall into the complex and extreme categories, reflecting that most DeFi exploits involve multi-contract interactions and substantial state modifications.

TABLE I: Transaction complexity distribution of dataset.

Complexity Level	Count	Avg Function Calls	Avg Addresses	Avg Max Depth
Moderate	14	7	3	3
Complex	51	83	9	8
Extreme	50	1,935	32	11
Total	**115**	**879**	**18**	**9**

Note: Total row shows sum of counts and dataset-wide averages for other metrics.

D. Comparative Analysis with SOTA (RQ1)

Experimental Setup. To answer **RQ1**, we compare FAULTSEEKER against DAppFL [41], the current state-of-the-art (SOTA) tool for blockchain transaction fault localization. The selection of DAppFL as our sole baseline is justified by the current state of blockchain fault localization research. Our systematic literature review identified DAppFL as the only tool that addresses cross-contract, multi-vulnerability fault localization in blockchain transactions. Existing alternatives fall into two categories with fundamental limitations: (1) blockchain-specific tools that target only single vulnerability types such as Reentrancy [39], [44], unsuitable for comprehensive multi-vulnerability transaction analysis; (2) general fault localization methods that lack essential blockchain domain knowledge (e.g., gas mechanisms and state dependencies) for smart contract analysis, making direct adaptation impractical for comprehensive evaluation.

Result Analysis. Table II presents the comparative performance results between FAULTSEEKER and DAppFL, measured by recall at top-1, top-3, and top-5 ranking positions.

FAULTSEEKER achieves 50.43% overall top-1 recall compared to 6.09% for DAppFL, demonstrating improved fault localization effectiveness. This performance advantage extends to expanded rankings, with FAULTSEEKER reaching 73.91% at top-3 and 76.52% at top-5, while DAppFL achieves 12.17% and 14.78% respectively. Both approaches exhibit similar ranking improvement characteristics: substantial gains from top-1 to top-3, followed by minimal increases from top-3 to top-5. FAULTSEEKER improves by 23.48% from top-1 to top-3, then by 2.61% to top-5. DAppFL shows 6.08% and 2.61% gains respectively. The substantial improvement from top-1 to top-3 for both tools indicates that actual faults often rank second or third, reflecting the challenge of precisely identifying actual vulnerabilities from suspicious benign functions. This necessitates interpretable evidence to support ranking decisions for practical security analysis workflows.

Additionally, FAULTSEEKER demonstrates consistent performance across complexity levels, maintaining effectiveness from moderate cases (71.43%) through complex cases (58.82%) to extreme cases (36.00%). In contrast, DAppFL shows the best performance in complex cases (13.73%), while achieving limited success in moderate and extreme cases. This suggests the learning models may overfit to specific attack patterns encountered during training, resulting in reduced effectiveness on moderate cases that lack sufficient complexity features and extreme cases beyond the learned feature space.

To provide comprehensive evaluation, we analyze failure

cases where FAULTSEEKER cannot accurately localize faults. These cases share a common characteristic: they require analysis capabilities extending beyond transaction execution to bytecode-level understanding. For example, storage collision attacks exploiting low-level EVM mechanisms present challenges because they arise from unintended storage layout interactions rather than logical function behavior, requiring deep understanding of compiler-level implementation details beyond transaction execution semantics. However, our framework remains effective for the majority of practical attack scenarios encountered in real-world blockchain exploits.

Finding 1: FAULTSEEKER consistently outperforms DAppFL across complexity levels and ranking positions, with overall top-1 recall of 50.43% versus 6.09%.

We further partition the dataset based on source code availability because this factor affects the analytical capabilities of different approaches. While DAppFL fails to operate in such scenarios due to its reliance on source code access for fault predictions, FAULTSEEKER successfully localizes faults in all 52 cases where source code is unavailable. This is because DAppFL employs a learning-based approach that requires source code for all identified contracts to extract features and generate fault predictions through its trained model. In contrast, FAULTSEEKER handles this by leveraging transaction traces, analyzing behavioral evidence such as unusual parameter values and call frequency without requiring implementation details. When source code is available, it serves as supplementary validation to enhance confidence in the analysis results. In fact, without access to code implementation details, attackers tend to exploit publicly visible contract interfaces, resulting in more explicit behavioral evidence for analysis.

Finding 2: FAULTSEEKER can handle cases when source code unavailable, providing fault localization capabilities where traditional approaches fail.

E. Comparative Analysis with Native LLMs (RQ2)

Experimental Setup. To answer **RQ2**, we compare performance of FAULTSEEKER with three leading LLMs: GPT-4o [6], Claude 3.7 Sonnet [2], and DeepSeek R1 [17]. These models represent leading general-purpose LLMs known for strong reasoning performance. Given the substantial expert time required for manual iterative analysis with native LLMs, we employ stratified sampling to select 30 transactions (10 per complexity level) to balance evaluation comprehensiveness with practical constraints. Each LLM receives identical transaction data as described in Section III-B1 along with standardized prompts specifying the task and output format. Each LLM needs to identify the top 5 most likely fault locations per transaction. A prediction is successful if the ground-truth fault location appears within the top-5 list, providing tolerance for fault localization uncertainty. All experimental materials are available in our repository [8].

TABLE II: Top-N performance comparison with DAppFL.

Complexity Level	Recall@1 (%)		Recall@3 (%)		Recall@5 (%)	
	FaultSeeker	DAppFL	FaultSeeker	DAppFL	FaultSeeker	DAppFL
Moderate (14)	71.43	0.00	85.71	7.14	85.71	7.14
Complex (51)	58.82	13.73	82.35	19.61	86.27	21.57
Extreme (50)	36.00	0.00	62.00	6.00	64.00	10.00
Overall (115)	**50.43**	**6.09**	**73.91**	**12.17**	**76.52**	**14.78**

Note: Values represent percentage of successful fault localizations at top-1, top-3, and top-5 rankings respectively.

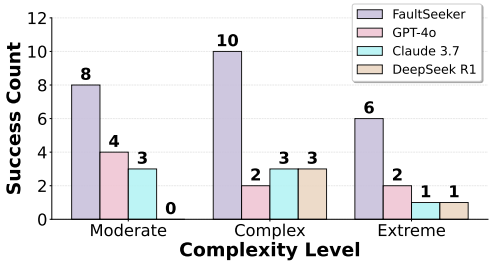

Fig. 6: Performance comparison across native LLMs.

Result Analysis. Figure 6 presents the comparative performance results between FAULTSEEKER and the three native LLM models across complexity levels. The results reveal that FAULTSEEKER achieves superior performance across all complexity categories, demonstrating resilience to increasing analytical complexity. Among the native LLMs, GPT-4o performs best but degrades substantially with complexity, Claude 3.7 maintains consistent limited performance before sharp decline with extreme cases, while DeepSeek R1 shows the most constrained results, failing entirely in moderate cases.

The performance decline observed across all native LLMs can be attributed to fundamental limitations in their memory and attention mechanisms when processing lengthy, multi-contract transaction sequences. Traditional LLMs operate with fixed context windows and attention patterns that struggle to maintain coherent reasoning across extended analytical chains required for complex blockchain transactions. In contrast, our multi-agent framework addresses these limitations through persistent memory management via orchestrator agents and specialized attention allocation through coordinated worker agents, enabling sustained analytical focus without losing contextual information as complexity increases.

Finding 3: FAULTSEEKER consistently outperforms all native LLMs across complexity levels, demonstrating superior resilience while LLMs show significant performance degradation with increased complexity.

Furthermore, during our analysis, we observe that native LLMs repeatedly misidentify certain functions as faulty across different models. Figure 7 summarizes the frequency of function names flagged across all models, showing that LLMs consistently generate false positives for: `withdraw` (48 times), `transfer` (40 times), and `approve` (33 times). These functions are correctly implemented but frequently misclassified as faulty. This suggests that LLMs exhibit learned bias

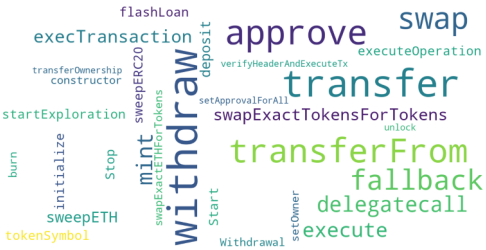

Fig. 7: Functions frequently identified as vulnerable by LLMs.

toward identifying functions related to asset operation without considering their usage in the transaction trace or actual implementation. This demonstrates that native LLMs rely on surface-level pattern matching rather than semantic analysis of transaction flows and code implementation. This limitation validates the necessity of our multi-agent framework, which enables investigation beyond function name patterns through systematic analysis of transaction execution flows.

Finding 4: Native LLMs frequently produce false positives for functions like `withdraw`, `transfer`, and `approve` despite correct implementation, indicating reliance on surface-level patterns over semantic analysis.

F. Ablation Study (RQ3)

Experimental Setup. To answer **RQ3**, we conduct ablation study with the following four variants:

- **Variant 1 (No Transaction-Level Forensics):** Skips forensic scoping and treats all outermost function calls as vulnerability entry points for analysis.
- **Variant 2 (No Information Organizer Agent):** Removes knowledge integration, with worker agents operating independently without global context integration.
- **Variant 3 (No Task Coordinator Agent):** Excludes task tree record, with task generation proceeding without awareness of completed analysis steps.
- **Variant 4 (No Specialized Worker Agents):** Replaces specialized workers with a single monolithic agent, evaluating the benefits of distributed analytical labor.

These variants isolate four critical design decisions: strategic scoping (Variant 1), knowledge integration (Variant 2), task coordination (Variant 3), and specialized workers (Variant 4). Variant 1 removes the entire transaction-level forensics stage because all its components collectively serve one objective of narrowing the investigation scope before function analysis.

Result Analysis. Table III presents the evaluation results for each variant compared to the complete FAULTSEEKER. The results reveal distinct patterns of component criticality, with transaction-level forensics (Variant 1) affecting result the most. Variant 1 removes the transaction-level forensics stage, demonstrates catastrophic performance degradation with a 97.8% drop in successful localizations. This dramatic decline underscores that strategic scoping is not merely an optimization but a critical prerequisite for accurate fault identification.

TABLE III: Ablation study results across variants.

Variant	Moderate	Complex	Extreme	Total	Drop
Complete FAULTSEEKER	10	42	41	93	–
Variant 1: No Forensics	0	2	0	2	-97.8%
Variant 2: No Organizer	7	20	11	38	-59.1%
Variant 3: No Coordinator	2	23	7	32	-65.6%
Variant 4: No Specialization	1	15	5	21	-77.4%

Note: Drop refers to performance drop percentage, calculated as (variant total - complete total) / complete total.

Without proper scoping, the framework becomes overwhelmed by the unfiltered search space when forced to analyze all function calls without prioritization.

The specialized workers (Variant 4) prove essential for maintaining the effectiveness of analysis, showing a substantial 77.4% performance reduction when replaced by monolithic execution. This validates the division of analytical labor principle [33], demonstrating that coordinated specialist agents significantly outperform sequential task execution approaches.

The task coordinator agent (Variant 3) contributes meaningfully to systematic analysis, with its removal resulting in a 65.6% performance drop, though the framework retains moderate effectiveness in complex scenarios. Interestingly, this variant performs competitively in complex cases, suggesting that while coordination enhances overall efficiency, the underlying analytical capabilities remain partially functional without centralized workflow management.

The information organizer agent (Variant 2) exhibits the smallest individual impact with a 59.1% performance reduction. This indicates that while centralized knowledge management provides valuable system coherence, individual workers retain considerable autonomous analytical capacity. The performance of the variant in complex transactions suggests that distributed reasoning can partially compensate for the absence of global context integration, though the overall system benefits significantly from coordinated information management.

These findings collectively demonstrate that our framework is carefully designed where each component contributes meaningfully to fault localization effectiveness, with transaction-level forensics serving as the foundational enabler and specialized agents providing the analytical depth necessary for comprehensive vulnerability detection.

Finding 5: All components contribute to fault localization effectiveness, with transaction-level forensics serving as the foundational enabler.

G. Efficiency and Cost Analysis (RQ4)

Experimental Setup. To answer **RQ4**, we conduct efficiency and cost analysis across complexity levels, measuring execution time and token usage for both stages separately. Time measurements capture end-to-end execution from function call to return. Cost analysis tracks token usage through recorded complete conversations between users and LLMs, calculating input prompts and output responses for stage 1 and each agent in Stage 2. Cost estimation applies recorded token usage

TABLE IV: Details of execution time across complexity levels.

Complexity Level	Stage 1 *(sec)*	Stage 2 *(sec)*	Total *(sec)*	Cost *(USD)*
Moderate	2.72	262.77	265.49	$1.55
Complex	7.87	446.53	454.40	$3.04
Extreme	33.58	484.40	517.98	$4.53

with Equation (1), where C is the total cost, n is the number of function calls, T_{in} and T_{out} are average input and output tokens per call, and $P_{in} = \$40$ and $P_{out} = \$80$ are the pricing rates per million tokens for GPT-4o [9].

$$C = n \times \left(\frac{T_{in}}{10^6} \times P_{in} + \frac{T_{out}}{10^6} \times P_{out} \right) \quad (1)$$

Result Analysis. Table IV presents the execution time and cost breakdown across complexity levels, revealing practical efficiency characteristics suitable for real-world deployment. Our framework demonstrates reasonable execution times ranging from 265 seconds (4.4 minutes) for moderate cases to 518 seconds (8.6 minutes) for extreme complexity transactions. These execution times compare favorably to manual expert analysis, which typically requires 30-60 minutes for moderate cases and several hours or days for complex multi-contract exploits [41], representing a significant efficiency gain while maintaining comprehensive analytical depth. Stage 1 exhibits minimal overhead, while Stage 2 dominates execution time, reflecting the intensive multi-agent coordination required for thorough vulnerability investigation.

Cost analysis reveals a linear scaling pattern from $1.55 for moderate to $4.53 for extreme complexity cases, demonstrating the economic feasibility. The cost scaling reflects the complexity of blockchain fault localization, where even human experts require multiple investigative attempts to identify correct exploit paths through intricate transaction flows.

We also conduct in-depth analysis with the detailed breakdown of token consumption patterns across specialist agents during task-driven function analysis. Retrieval agents consume the highest input tokens in average (10,178), reflecting the extensive data gathering required for comprehensive vulnerability investigation. Task selector agents demonstrate substantial computational demands with both high input (8,126) and output (4,123) token consumption, reflecting the resource-intensive nature of adaptive task prioritization in multi-agent coordination. The detailed statistics for the token usage breakdown can be found in our repository [8].

Finding 6: FAULTSEEKER completes fault localization within 4.4-8.6 minutes at $1.55-$4.53 per transaction, which is suitable for routine security assessments.

H. Real-World Applicability (RQ5)

Experimental Setup. To answer **RQ5**, we conduct an in-depth case study using the Orion Protocol exploit transaction from Section II-D to demonstrate the practical relevance and real-world applicability of FAULTSEEKER. We apply the complete two-stage pipeline with GPT-4o [6] model by OpenAI. The analysis documents intermediate results, decision points, and analytical insights at each stage to showcase progressive understanding capabilities and evaluate practical applicability within professional security analysis workflows.

Result Analysis. During transaction-level forensics, FAULTSEEKER evaluates the transaction trace against precompiled patterns as detailed in Section III-B2. These patterns include nested repeated calls characteristic of reentrancy vulnerabilities and operations following flash loans typically involved in price manipulation attacks. The evaluation identifies three suspicious function calls: `swapThroughOrionPool()`, `withdraw()`, and `0x4729ed3c()`. These represent functions where fault likely reside deeper within their call hierarchies. This scoping approach focuses investigation efforts on suspicious interaction patterns within complex transactions.

With these identified function calls for further inspection, the task-driven function analysis proceeds to investigate each suspicious function to locate actual fault within the call hierarchies. During the analysis of `swapThroughOrionPool()`, the highest-priority from the forensics stage. As shown in Listing 1, the orchestrator decomposes investigation into a task tree that maintains throughout analysis. Workers execute specialized tasks, with critical findings emerging from task 1.9. The analysis reveals `depositAsset` appears 6 times as a child call under `swapThroughOrionPool`, indicating unusual execution patterns. Workers retrieve and examine `depositAsset` source code, confirming the absence of reentrancy protection. The orchestrator synthesizes findings to identify the fault: `depositAsset` lacks reentrancy protection and becomes exploitable when invoked during swap execution through malicious token callbacks.

```
High Appearance Count in Child Calls:
— depositAsset::0xb55.....45AA: 6 calls

Task Tree for swapThroughOrionPool Analysis:
1.9 Identify high frequency child calls
   1.9.1 Analyze depositAsset function call
      1.9.1.1 Investigate depositAsset logic
      1.9.1.2 Assess asset handling for security flaws
```

Listing 1: Intermediate result during the task-driven analysis of function call `swapThroughOrionPool`.

This case study illustrates the real-world applicability of FAULTSEEKER for fault localization in blockchain exploit incident analysis. By providing interpretable evidence with intermediate results, FAULTSEEKER aims to assist security experts in transaction analysis while supporting confident decision-making in professional security analysis.

Finding 7: FAULTSEEKER provides interpretable analysis that enables security teams to validate findings and understand the analysis process.

V. DISCUSSION

A. Threats to Validity

Internal Validity. The primary threat involves ground truth establishment for fault annotations, where subjectivity may

introduce labeling inconsistencies. We mitigate this by selecting transactions with expert analysis from reliable security platforms and research works, validated through consensus review by three independent smart contract security experts.

External Validity. Our evaluation focuses on historical attack incidents from specific blockchain networks, which may not fully represent rapidly evolving smart contract vulnerabilities. We address this by including recent cases up to February 2025 and ensuring diverse representation across attack complexity levels and blockchain platforms. Additionally, the limited sample size when comparing with native LLMs (Section IV-E), while justified by methodological constraints, may affect the generalizability of the results. We try to mitigate this limitation through stratified sampling with equal representation across complexity levels.

B. Discussion and Future Work

In this section, we discuss possible directions for future research. First, our approach targets EVM-compatible blockchains as they represent the dominant market with mature toolsets [1], [5]. Adapting to non-EVM blockchains presents unique challenges due to architectural differences. For example, EVM blockchains maintain global state accessible by any contract during execution while blockchains like Solana do not maintain persistent contract state on chain and require transactions to pre-declare all data they will access.

Second, FAULTSEEKER currently provides interpretable evidence and intermediate results to support human analyst review, but enhanced human-AI collaboration represents a promising direction. The orchestrator-worker architecture could be extended to support interactive capabilities where analysts guide task prioritization and provide corrective feedback during analysis. Such collaboration would leverage human domain expertise to complement automated analysis capabilities.

Third, we opt for prompt-based adaptation rather than fine-tuning LLMs on blockchain-specific tasks. This design choice prioritizes deployment cost and adaptability to rapidly evolving attack patterns through prompt updates rather than costly model retraining cycles. However, domain-specific fine-tuning could potentially improve performance and represents a valuable direction for future research.

VI. Related Work

A. Fault Localization

Fault localization in traditional software engineering has explored diverse approaches including crash deduplication through fault signatures [20], learning-based model [42], and control flow tracing [43]. However, blockchain fault localization presents unique challenges by involving blockchain-specific features such as gas consumption and economic models beyond code-level defects.

Existing blockchain fault localization tools are limited and tend to target single vulnerability types like reentrancy attacks [39], [44]. DAppFL [41] integrates execution flow analysis with token tracing using machine learning but relies on statistical pattern recognition, requires source code, and produces results that lack interpretability. Recent concurrent work Malo [46] employs graph encoders for fault localization with LLM for interpretation, but the fault localization results remain limited by the encoder model. Our approach differs by integrating transaction trace analysis directly into fault localization, analyzing execution sequences and semantic dependencies throughout the entire fault identification process rather than using structural graph relationships.

B. LLM-Based Security Analysis

Recent research has explored LLMs for smart contract security, primarily focusing on proactive vulnerability detection through static code analysis. GPTScan [34] combines GPT with program analysis for vulnerability detection while PropertyGPT [25] for formal verification, and SmartLLMSentry [45] for comprehensive security analysis. These approaches demonstrate the effectiveness of LLMs in understanding contract semantics and identifying vulnerabilities before deployment. However, little exploration has been conducted on post-incident fault localization, which requires understanding transaction-level behavior rather than static code patterns. Our work addresses this gap through a multi-agent framework that enables sustained LLM reasoning across complex multi-contract execution flows.

VII. Conclusion

This paper presents FAULTSEEKER, a multi-agent framework for blockchain fault localization through transaction-level forensics and coordinated function analysis. The two-stage architecture enables sustained reasoning across multi-contract interactions through persistent context management and specialized agent coordination. Evaluation on 115 real-world malicious transactions shows improved performance over existing approaches including DAppFL and native LLMs, while maintaining practical efficiency (4.4-8.6 minutes) and cost-effectiveness ($1.55-$4.53). These results demonstrate the applicability of FAULTSEEKER for automated fault localization in blockchain security analysis.

VIII. Data Availability

The evaluation dataset, experimental results, and framework implementation details are available in our repository [8].

Acknowledgment

This research is supported by the National Research Foundation, Singapore, and DSO National Laboratories under the AI Singapore Programme (AISG Award No: AISG4-GC-2023-008-1B); by the National Research Foundation Singapore and the Cyber Security Agency under the National Cybersecurity R&D Programme (NCRP25-P04-TAICeN); and by the Prime Minister's Office, Singapore under the Campus for Research Excellence and Technological Enterprise (CREATE) Programme. Any opinions, findings and conclusions, or recommendations expressed in these materials are those of the author(s) and do not reflect the views of the National Research Foundation, Singapore, Cyber Security Agency of Singapore, Singapore.

REFERENCES

[1] "5hawnxu/getsourcecode," 2025, [Online; accessed 2025-05-22]. [Online]. Available: https://github.com/5hawnXu/getSourceCode

[2] "Claude sonnet 3.7 \ anthropic," 2025, [Online; accessed 2025-05-23]. [Online]. Available: https://www.anthropic.com/news/claude-3-7-sonnet

[3] "Defillama - defi dashboard," 2025, [Online; accessed 2025-04-28]. [Online]. Available: https://defillama.com/

[4] "Dive into transaction to act wisely - blocksec phalcon explorer," 2025, [Online; accessed 2025-05-21]. [Online]. Available: https://blocksec.com/explorer

[5] "foundry-rs/foundry: Foundry is a blazing fast, portable and modular toolkit for ethereum application development written in rust." 2025, [Online; accessed 2025-05-21]. [Online]. Available: https://github.com/foundry-rs/foundry

[6] "Hello gpt-4o — openai," 2025, [Online; accessed 2025-05-23]. [Online]. Available: https://openai.com/index/hello-gpt-4o/

[7] "Introduction — etherscan," 3 2025, [Online; accessed 2025-05-21]. [Online]. Available: https://docs.etherscan.io/

[8] "kairanskrr/faultseeker," 2025, [Online; accessed 2025-10-02]. [Online]. Available: https://github.com/kairanskrr/FaultSeeker

[9] "Pricing — openai," 2025, [Online; accessed 2025-05-30]. [Online]. Available: https://openai.com/api/pricing/

[10] "Sunweb3sec/defihacklabs: Reproduce defi hacked incidents using foundry." 2025, [Online; accessed 2025-05-22]. [Online]. Available: https://github.com/SunWeb3Sec/DeFiHackLabs

[11] Admin, "Understanding gas consumption: Analyzing highly paid operations on ethereum and other blockchains - bitquery," 8 2024, [Online; accessed 2025-05-25]. [Online]. Available: https://bitquery.io/blog/gas-consumption-analysis-ethereum-blockchains

[12] V. Akuthota, R. Kasula, S. T. Sumona, M. Mohiuddin, M. T. Reza, and M. M. Rahman, "Vulnerability detection and monitoring using llm," in *2023 IEEE 9th International Women in Engineering (WIE) Conference on Electrical and Computer Engineering (WIECON-ECE)*. IEEE, 2023, pp. 309–314.

[13] BlockSec, "Blocksec phalcon security platform," 2025, accessed: 2025-05-31. [Online]. Available: https://blocksec.com/phalcon/security

[14] T. Chen, Z. Li, Y. Zhu, J. Chen, X. Luo, J. C.-S. Lui, X. Lin, and X. Zhang, "Understanding ethereum via graph analysis," *ACM Transactions on Internet Technology (TOIT)*, vol. 20, no. 2, pp. 1–32, 2020.

[15] G. Deng, Y. Liu, V. Mayoral-Vilches, P. Liu, Y. Li, Y. Xu, T. Zhang, Y. Liu, M. Pinzger, and S. Rass, "{PentestGPT}: Evaluating and harnessing large language models for automated penetration testing," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 847–864.

[16] etherscan.io, "Ethereum transaction hash: 0xa6f63fcb6b... — etherscan," 2025, [Online; accessed 2025-09-06]. [Online]. Available: https://etherscan.io/tx/0xa6f63fcb6bec8818864d96a5b1bb19e8bd85ee37b2cc916412e720988440b2aa

[17] D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi *et al.*, "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning," *arXiv preprint arXiv:2501.12948*, 2025.

[18] T. Guo, X. Chen, Y. Wang, R. Chang, S. Pei, N. V. Chawla, O. Wiest, and X. Zhang, "Large language model based multi-agents: A survey of progress and challenges," *arXiv preprint arXiv:2402.01680*, 2024.

[19] M. A. Islam, M. E. Ali, and M. R. Parvez, "Mapcoder: Multi-agent code generation for competitive problem solving," *arXiv preprint arXiv:2405.11403*, 2024.

[20] A. Kallingal Joshy and W. Le, "Fuzzeraid: Grouping fuzzed crashes based on fault signatures," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3551349.3556959

[21] R. W. Lawler, "The progressive construction of mind," *Cognitive Science*, vol. 5, no. 1, pp. 1–30, 1981.

[22] T. Lehtinen, C. Koutcheme, and A. Hellas, "Let's ask ai about their programs: Exploring chatgpt's answers to program comprehension questions," in *Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training*, 2024, pp. 221–232.

[23] X. Li, W. Li, Y. Zhang, and L. Zhang, "Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization," in *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*, 2019, pp. 169–180.

[24] Y. Li, S. Wang, and T. Nguyen, "Fault localization with code coverage representation learning," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 661–673.

[25] Y. Liu, Y. Xue, D. Wu, Y. Sun, Y. Li, M. Shi, and Y. Liu, "Propertygpt: Llm-driven formal verification of smart contracts through retrieval-augmented property generation," *arXiv preprint arXiv:2405.02580*, 2024.

[26] Y. Lou, Q. Zhu, J. Dong, X. Li, Z. Sun, D. Hao, L. Zhang, and L. Zhang, "Boosting coverage-based fault localization via graph-based representation learning," in *Proceedings of the 29th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2021, pp. 664–676.

[27] G. Lu, X. Ju, X. Chen, W. Pei, and Z. Cai, "Grace: Empowering llm-based software vulnerability detection with graph structure and in-context learning," *Journal of Systems and Software*, vol. 212, p. 112031, 2024.

[28] D. Nam, A. Macvean, V. Hellendoorn, B. Vasilescu, and B. Myers, "Using an llm to help with code understanding," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[29] K. Oberauer, "Working memory and attention–a conceptual analysis and review," *Journal of cognition*, vol. 2, no. 1, p. 36, 2019.

[30] R. Ren, Y. Wang, F. Liu, Z. Li, G. Tyson, T. Miao, and G. Xie, "Grace: Interpretable root cause analysis by graph convolutional network for microservices," in *2023 IEEE/ACM 31st International Symposium on Quality of Service (IWQoS)*. IEEE, 2023, pp. 1–4.

[31] M. Rodler, W. Li, G. O. Karame, and L. Davi, "Sereum: Protecting existing smart contracts against re-entrancy attacks," *arXiv preprint arXiv:1812.05934*, 2018.

[32] C. Sendner, L. Petzi, J. Stang, and A. Dmitrienko, "Vulnerability scanners for ethereum smart contracts: A large-scale study," *arXiv preprint arXiv:2312.16533*, 2023.

[33] A. Strauss, "Work and the division of labor," *Sociological quarterly*, vol. 26, no. 1, pp. 1–19, 1985.

[34] Y. Sun, D. Wu, Y. Xue, H. Liu, H. Wang, Z. Xu, X. Xie, and Y. Liu, "Gptscan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[35] W. Tao, Y. Zhou, Y. Wang, W. Zhang, H. Zhang, and Y. Cheng, "Magis: Llm-based multi-agent framework for github issue resolution," *Advances in Neural Information Processing Systems*, vol. 37, pp. 51 963–51 993, 2024.

[36] C. Team, "What are gas limits? understanding ethereum transaction fees (2025 guide)," 4 2025, [Online; accessed 2025-05-25]. [Online]. Available: https://www.coinsdo.com/en/blog/what-are-gas-limits

[37] Tenderly, "Tenderly: Full-stack web3 infrastructure platform," 2025, accessed: 2025-05-31. [Online]. Available: https://tenderly.co/

[38] Q. Wang, Z. Wang, Y. Su, H. Tong, and Y. Song, "Rethinking the bounds of llm reasoning: Are multi-agent discussions the key?" *arXiv preprint arXiv:2402.18272*, 2024.

[39] Z. Wang, J. Chen, Y. Wang, Y. Zhang, W. Zhang, and Z. Zheng, "Efficiently detecting reentrancy vulnerabilities in complex smart contracts," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 161–181, 2024.

[40] M. Wooldridge and N. R. Jennings, "Intelligent agents: Theory and practice," *The knowledge engineering review*, vol. 10, no. 2, pp. 115–152, 1995.

[41] Z. Wu, J. Wu, H. Zhang, Z. Li, J. Chen, Z. Zheng, Q. Xia, G. Fan, and Y. Zhen, "Dappfl: Just-in-time fault localization for decentralized applications in web3," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 137–148. [Online]. Available: https://doi-org.remotexs.ntu.edu.sg/10.1145/3650212.3652116

[42] D. Xu, D. Tang, Y. Chen, X. Wang, K. Chen, H. Tang, and L. Li, "Racing on the negative force: Efficient vulnerability Root-Cause analysis through reinforcement learning on counterexamples," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 4229–4246. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/xu-dandan

[43] C. Yagemann, M. Pruett, S. P. Chung, K. Bittick, B. Saltaformaggio, and W. Lee, "ARCUS: Symbolic root cause analysis of exploits in production systems," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 1989–2006. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/yagemann

[44] S. Yang, J. Chen, M. Huang, Z. Zheng, and Y. Huang, "Uncover the premeditated attacks: Detecting exploitable reentrancy vulnerabilities by identifying attacker contracts," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–12.

[45] O. Zaazaa and H. El Bakkali, "Smartllmsentry: A comprehensive llm based smart contract vulnerability detection framework," *Journal of Metaverse*, vol. 4, no. 2, pp. 126–137, 2024.

[46] H. Zhang, J. Wu, Z. Wu, Z. Chen, D. Lin, J. Chen, Y. Zhou, and Z. Zheng, "Malo in the code jungle: Explainable fault localization for decentralized applications," *IEEE Transactions on Software Engineering*, 2025.