

Evidence-Grounded Logic Vulnerability Detection for Smart Contracts via Runtime Context Reasoning

Guang Gong
Tsinghua University
Beijing, China
360 Vulnerability Research Institute
Beijing, China
higongguang@gmail.com

Yucong Jiang
360 Vulnerability Research Institute
Beijing, China
jiangyucong@360.cn

Rong Jian
360 Vulnerability Research Institute
Beijing, China
jianr1@outlook.com

Xiaotao Feng
360 Vulnerability Research Institute
Beijing, China
escofeng@gmail.com

Zhihan Wei
360 Vulnerability Research Institute
Beijing, China
01chenqing@gmail.com

Qi Li
Tsinghua University
Beijing, China
qli01@tsinghua.edu.cn

Ke Xu✉
Tsinghua University
Beijing, China
xuke@tsinghua.edu.cn

Abstract

Smart contracts are foundational to DeFi, yet detecting business-logic vulnerabilities, i.e., execution is syntactically correct but violates a contract's intended behavior, remains difficult. Most recent LLM-based approaches still follow a largely static, source-code–centric auditing paradigm. In practice, however, smart contracts are inherently stateful: many logic flaws emerge only under specific contract states and across particular state transitions. As a result, code-only auditing can produce false positives and miss true vulnerabilities. To address this gap, we present RcAudit, a system for detecting smart-contract logic vulnerabilities via runtime-context–enhanced LLM reasoning. RcAudit constructs a Contract Logic Knowledge Base that captures contract behavior from multiple perspectives to support LLM analysis. It then applies a state- and intent-aware fuzzing strategy to generate logic-critical runtime contexts, and performs LLM reasoning over selected contexts that exhibit potential intent–behavior inconsistencies to pinpoint vulnerabilities. We evaluate RcAudit on three real-world datasets. RcAudit identifies 74 of 132 known logic vulnerabilities with high precision (96.9%), achieving 117.6% higher detection than the strongest baseline. Importantly, RcAudit further uncovers 22 zero-day vulnerabilities in real-world projects.

CCS Concepts

• **Security and privacy → Distributed systems security**; **Vulnerability scanners**; *Domain-specific security and privacy architectures.*



This work is licensed under a Creative Commons Attribution 4.0 International License.
ASE '26, Munich, Germany
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3837471

Keywords

Smart Contract, Vulnerability Detection, Logic Vulnerability, Fuzzing, Ethereum, Large Language Model

ACM Reference Format:
Guang Gong, Yucong Jiang, Rong Jian, Xiaotao Feng, Zhihan Wei, Qi Li, and Ke Xu. 2026. Evidence-Grounded Logic Vulnerability Detection for Smart Contracts via Runtime Context Reasoning. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3832783.3837471

1 Introduction

Smart contracts [47] serve as the foundational infrastructure for trustless automation in decentralized finance (DeFi) [24], securing billions of dollars in total value [15]. Despite their critical role, business logic vulnerabilities [37, 40, 46] have emerged as the primary security threat [5], characterized by syntactically correct executions that nonetheless violate the contract's intended economic or functional design. In 2025, business logic vulnerabilities accounted for over 85% of losses from contract vulnerability exploits—approximately $464M out of $525M [2]. Because these vulnerabilities are context-dependent and intertwined with specific application semantics, traditional automated analysis tools[9, 14, 19, 21, 23, 39, 41, 44] —which primarily focus on generic patterns like reentrancy or integer overflows—often struggle to distinguish between legitimate operations and those that contravene contract-specific intentions.

Recent studies have increasingly utilized Large Language Models (LLMs) to bridge the semantic gap in logic vulnerability detection by leveraging their advanced reasoning capabilities [12, 22, 37, 45]. These methods typically rely on direct querying of code artifacts[6] or using predefined prompt templates[37] to check for specific vulnerability patterns. In practice, owing to the stateful nature of smart contracts, many logic flaws emerge only under specific contract

states and across particular state transitions. Without access to concrete state evolution, static LLM inspection is inherently constrained, making it difficult to detect complex state-dependent logic vulnerabilities. For example, an auditor cannot easily determine a contract's status—such as its total liquidity or a user's debt—by looking at the source code alone.

As shown in Figure 1, our key insight is to shift the analysis target from static source code to concrete runtime contexts, namely executed transactions and their resulting contract states. By reasoning over observed state transitions, this approach enables accurate identification of logic vulnerabilities that manifest only under specific contract states.

Based on this insight, we develop RcAudit, a novel logic vulnerability detection system that grounds LLM reasoning in concrete runtime contexts. Instead of inferring flaws indirectly from code structure, RcAudit analyzes logic violations as they emerge during actual execution. RcAudit is organized around three tightly integrated stages. First, it constructs a Contract Logic Knowledge Base by analyzing source code structure and, where available, supplementing it with developer documentation to capture contract behavior. Second, for effective runtime context generation, RcAudit employs a state- and intent-aware fuzzing mechanism that systematically drives the contract into logic-critical states where violations of business intent may emerge. Finally, RcAudit performs logic reasoning grounded in runtime contexts. It identifies inconsistencies by analyzing executions that violate inferred contract logic through an auditing process conducted from two distinct perspectives.

Our contributions are summarized as follows:

- We present RcAudit, which integrates runtime execution evidence with LLM reasoning for logic vulnerability auditing. By directly observing state transitions, RcAudit mitigates the hallucinations and false positives that commonly afflict existing LLM-based tools.
- We design a framework that integrates contract fuzzing with a reasoning mechanism that evaluates the contract from two distinct perspectives. This synergy ensures deep exploration of states where the program logic is vulnerable, while ensuring precision by validating executions against global business intentions.
- Our evaluation shows that RcAudit outperforms the strongest baseline by 117.6% while maintaining high precision (96.9%) on the evaluated datasets. It further demonstrates practical utility by identifying 22 previously undisclosed vulnerabilities in real-world auditing projects.

2 Background

Smart Contract Fuzzing. Fuzzing is a widely adopted automated testing technique that explores program behaviors through randomized input generation and mutation. In smart contracts, fuzzing operates over sequences of transactions that invoke contract functions and drive state transitions [7, 43]. Since contract vulnerabilities rarely manifest as explicit crashes, fuzzers rely on *vulnerability oracles*—predicates that identify states that are abnormal. Existing smart contract fuzzers primarily employ predefined oracles targeting common vulnerability patterns, such as reentrancy and arithmetic errors [7, 25]. While some tools support programmable

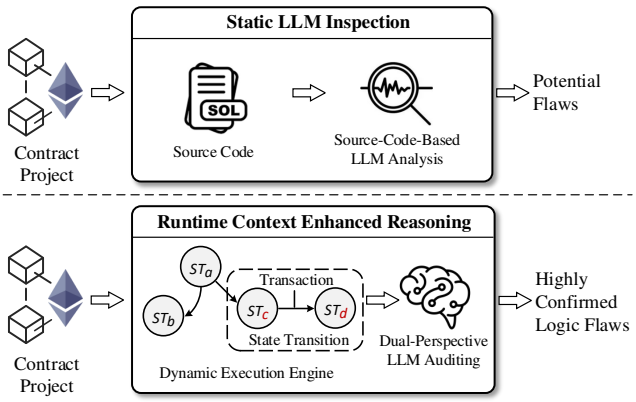


Figure 1: Comparison between static LLM-based auditing and RcAudit. While existing methods rely on static analysis to identify potential flaws, RcAudit leverages a dynamic execution engine to capture concrete state transitions, providing runtime-context-enhanced reasoning to confirm complex business logic vulnerabilities.

oracles to capture contract-specific properties [10, 20, 34], constructing such oracles requires deep domain expertise and significant manual effort, limiting their practicality at scale. Moreover, fuzzing guidance is typically based on low-level execution feedback, including branch coverage or storage access patterns [34, 43]. While effective for structural exploration, these signals provide little insight into the business intent of contract logic, making it difficult to reach or identify subtle logic flaws that depend on specific states.

Smart Contract Invariants. Invariants encode fundamental properties that a smart contract is expected to preserve during execution and are widely used as correctness or security oracles. Their effectiveness in exposing logic inconsistencies has motivated extensive work on automated invariant inference. Prior approaches mainly fall into two categories. Data-driven methods infer invariants from historical transactions [28, 35], while techniques based on static analysis, often enhanced with LLMs, synthesize invariants directly from source code [26, 27, 40]. These inferred invariants are then applied to fuzzing or verification as definitive correctness properties. However, existing techniques face inherent limitations. Data-driven methods are ineffective for contracts with limited execution history, while static and LLM-based approaches may overlook simple yet critical properties or generate overly abstract invariants.

3 Motivation

Logic vulnerabilities in smart contracts arise from mismatches between intended business intent and actual implementation. Unlike syntactic bugs, such vulnerabilities often evade static analyzers and template-based detectors, as they require specific, state-dependent transaction sequences to manifest. To illustrate the limitations of existing approaches, we examine a real-world exploit on the Ethereum blockchain.

3.1 Motivating Example

Figure 2 presents a simplified version of OTSeaStaking, a real-world staking contract that was exploited due to a vulnerability arising from logically inconsistent state updates across multiple functions. This vulnerability is representative of a broad class of

```
1  contract OTSeaStaking {
2    struct Deposit {
3      uint32 rewardEpoch;
4      uint88 amount; }
5
6    mapping(address => Deposit[]) public deposits;
7    uint32 public currentEpoch;
8
9    function stake(uint88 amount) external {
10     require(amount > 0);
11     ... // deposit transfer logic
12     deposits[msg.sender].push(Deposit(currentEpoch + 1,
           amount)); }
13
14   function withdraw(uint256 index) external {
15     Deposit storage deposit = deposits[msg.sender][index];
16     require(deposit.rewardEpoch != 0, "Already withdrawn");
17     uint256 amount = deposit.amount;
18     // Mark as withdrawn but keep the deposit
19     deposit.rewardEpoch = 0;
20     ... /* transfer amount to sender */ }
21
22   function claim(uint256 index) external {
23     Deposit storage deposit = deposits[msg.sender][index];
24     // statically calculate reward based on currentEpoch and
           epoch in calculateRewards
25     uint256 rewards = calculateRewards(msg.sender, index);
26     deposit.rewardEpoch = currentEpoch;
27     ... /* transfer amount to sender */ }
```

Figure 2: A simplified version of the vulnerable OTSeaStaking smart contract. The flaw enables a double-withdrawal attack due to an inconsistent state update in the `claim()` function.

logic flaws in DeFi protocols, where a state variable implicitly encodes multiple business roles and is modified along different execution paths without a global consistency check.

The contract maintains user deposits using a `Deposit` structure with a `rewardEpoch` storage variable. In practice, this single variable implicitly serves two distinct roles: 1). It determines whether a deposit is eligible for reward claims. 2). And acts as a withdrawal flag: after a successful `withdraw()`, the contract sets `rewardEpoch = 0` to prevent further withdrawals. However, the `claim()` function—designed solely to distribute accrued rewards—inadvertently overwrites this flag by resetting `rewardEpoch` to current epoch. Therefore, an attacker can exploit this by executing the following sequence:

(1) `stake()` - creates a deposit (line 9);
(2) `withdraw()` - succeeds and sets `rewardEpoch = 0` (line 14);
(3) `claim()` - resets `rewardEpoch` to a non-zero value (line 22);
(4) `withdraw()` again - succeeds, because the condition check `deposit.rewardEpoch != 0` now passes (line 16).

This exploit reveals a fundamental logic flaw: the same storage variable is updated under incompatible assumptions across different execution paths, enabling state transitions that violate the intended business rule.

3.2 The Limitation of Existing Approaches

Traditional static analyzers [3, 14] primarily reason over syntax and control flow, making them effective for detecting pattern-based vulnerabilities. However, logic vulnerabilities often require contract-specific semantic properties and cross-function state reasoning that are difficult to infer and scale in practice. For example, the `OTSeaStaking` flaw violates the intended role of `rewardEpoch`, while every individual execution remains valid, making it difficult for existing analyzers to identify without manually specified properties.

Recent LLM-based auditing systems [22, 37, 44] attempt to detect logic flaws via natural language reasoning, yet they typically analyze contracts in isolation without runtime evidence. In the `OTSeaStaking` case, the vulnerability depends on a precise state transition sequence (`stake` → `withdraw` → `claim`). Lacking visibility into concrete state evolution, LLMs may hallucinate incorrect state outcomes or misinterpret the actual value updates of variables like `rewardEpoch`. Furthermore, the `claim()` function is dominated by complex reward calculations. This excessive implementation context submerges the critical business logic violation—the unintended reset of the withdrawal flag—within the details.

3.3 Research Challenges

The inability of existing static analysis and LLM-based paradigms to reliably detect logic vulnerabilities points to three fundamental research challenges:

Challenge 1: Requirement for Global Business Intention Awareness. Logic vulnerabilities arise from deviations between high-level business rules and their low-level implementation. While such rules are often implicitly encoded in source code or accompanying documentation, they cannot be directly inferred through syntactic or control-flow analysis alone. Identifying these flaws therefore requires reasoning over both code structure and natural-language descriptions to derive meaningful, contract-specific correctness properties.

Challenge 2: Complexity of Inferring State-Dependent Exploitation Conditions. Logic vulnerabilities typically manifest only under specific internal states that are reached through non-trivial sequences of cross-function interactions. These exploitation conditions depend on precise state transitions accumulated over time, rather than on isolated function executions. Reasoning about such dynamic, state-dependent behaviors is inherently challenging for tools that solely examine static source code, especially when the underlying state space is large and sparse, making reachability analysis difficult in practice.

Challenge 3: Difficulty in Identifying Critical Logic amid Implementation Details. Core business logic is often interwoven with extensive implementation details, such as access control, safety checks, and auxiliary bookkeeping, causing logic-relevant state updates to be submerged within a large volume of incidental operations. As a result, pattern-based analyzers—lacking business intent—cannot distinguish critical accounting behaviors from protective code, while LLM-based approaches are easily distracted by excessive low-level details, making it difficult to isolate and reason about the true business logic.

4 Design of RcAudit

RcAudit generates runtime contexts guided by high-level business logic to detect mismatches between intended and actual behavior. As shown in Figure 3, it turns intent into execution evidence and validates behavior against that evidence. To generate diverse runtime contexts, RcAudit employs a fuzzing-based exploration framework [7, 20, 34, 43] augmented with intent-aware guidance, and

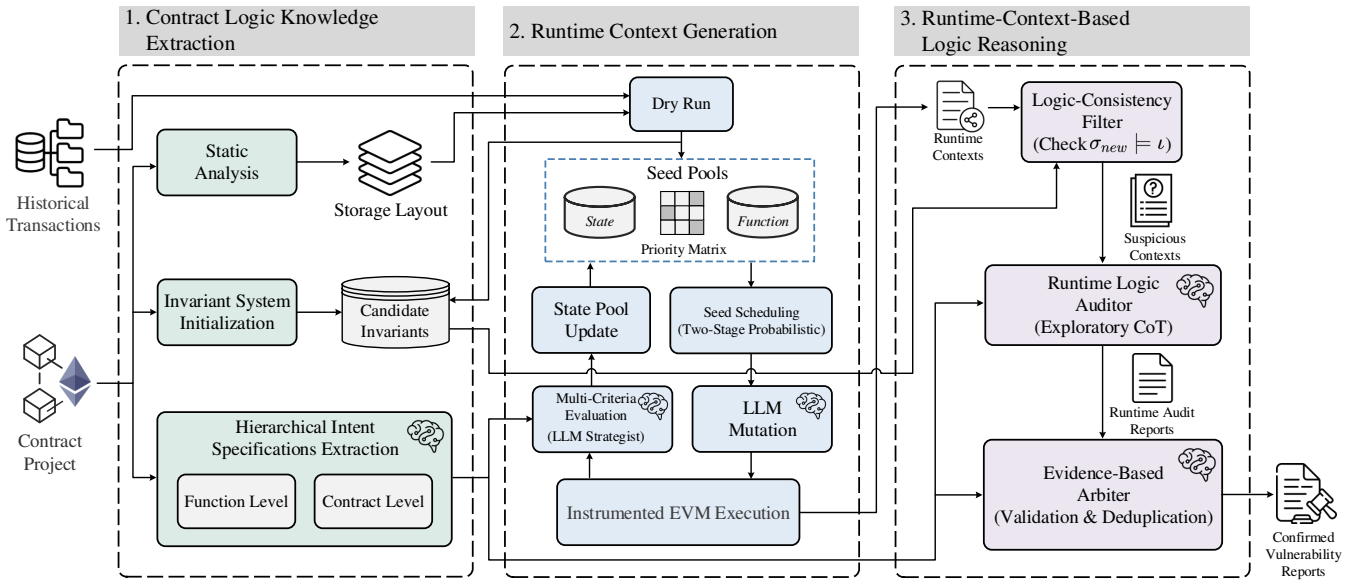


Figure 3: RcAudit's Workflow

applies LLM-based reasoning to validate potential intent–behavior inconsistencies.

RcAudit realizes this runtime-centric design by (i) constructing an explicit global logic knowledge base to support LLM reasoning (Section 4.1), (ii) generating runtime contexts that expose logic-critical states through state- and intent-aware fuzzing (Section 4.2), and (iii) reasoning over contexts that may contain logical inconsistencies (Section 4.3).

4.1 Contract Logic Knowledge Extraction

The workflow starts with Contract Logic Knowledge Extraction, where RcAudit organizes source code and available documentation into a structured knowledge base used by subsequent runtime guidance and reasoning.

This knowledge base contains three components: (i) a storage-semantic view that maps raw slots back to source-level variables, (ii) a dynamically maintained candidate invariant set used for runtime context filtering, and (iii) hierarchical intent specifications that summarize business logic at function and contract levels.

4.1.1 Storage Layout and Candidate Invariant Initialization. To ground runtime evidence in source-level semantics, RcAudit first recovers storage layouts using static analysis techniques commonly adopted in smart contract analyzers [7, 14, 25]. The recovered slot-to-variable mapping (including structured fields such as mappings) is then used to maintain parsed states for runtime context construction and downstream LLM reasoning.

RcAudit then initializes a candidate invariant set for runtime context filtering. Existing studies have shown that lightweight invariants derived from program structure and execution history capture common semantic properties of smart contracts [27, 28, 35, 40]. Building on these observations, RcAudit instantiate an invariant system tailored for fuzzing-based exploration, where invariants serve as runtime signals for identifying semantically interesting state transitions. Specifically, RcAudit maintains two categories of invariants: (1) contract-level invariants, which describe relations expected to hold across different function executions; and (2) function-level invariants, which capture function-specific pre/post-state relations. Each invariant is represented as a relational predicate over expressions extracted from contract states, transaction inputs, constants, and derived ghost expressions.

Algorithm 1 summarizes the initialization procedure. Starting from source-derived variables and transaction features, it constructs a unified expression space and generates type-compatible relational predicates, then augments them with constant and small-set constraints inferred from initial traces.

Algorithm 1: Invariant System Initialization

Require: Contract source code C
Ensure: Set of candidate invariants $\mathcal{I}$
$\mathcal{S} \leftarrow \text{StateVariables}(C)$
$\mathcal{T} \leftarrow \bigcup_{f \in \text{Functions}(C)} \text{TxVars}(f)$
$\mathcal{C} \leftarrow \text{Constants}(C)$
$\mathcal{G}_0 \leftarrow \text{ComposedExprsFromCFG}(C)$ {e.g., `balances[msg.sender]`}
$\mathcal{G} \leftarrow \mathcal{G}_0 \cup \{\texttt{Old}(g) \mid g \in \mathcal{G}_0\}$
$\mathcal{G} \leftarrow \mathcal{G} \cup \{\texttt{MappingValSum}(m) \mid m \text{ is a summable mapping in } C\}$
$\mathcal{G} \leftarrow \mathcal{G} \cup \{\texttt{ArrSum}(a) \mid a \text{ is a summable array in } C\}$ // (similarly extend for struct-field sums)
$\mathcal{E} \leftarrow \mathcal{S} \cup \mathcal{T} \cup \mathcal{C} \cup \mathcal{G}$
$\mathcal{I} \leftarrow \{e_1 \ \texttt{op} \ e_2 \mid (e_1, e_2) \in \mathcal{E} \times \mathcal{E}, \ \texttt{op} \in \{==, \leq, \geq\}, \ \texttt{TypeCompatible}(e_1, e_2, \texttt{op})\}$
for each $e \in \mathcal{E}$ **do**
  $C_e \leftarrow \text{InferConstants}(e)$ {from initial traces}
  $\mathcal{I} \leftarrow \mathcal{I} \cup \{e == c \mid c \in C_e\}$
  if $|C_e| <= 10$ **then**
    $\mathcal{I} \leftarrow \mathcal{I} \cup \{e \in C_e\}$
  end if
end for
return $\mathcal{I}$

Prompt Template: Function-Level Intent Distillation

Role: You are a smart contract architect analyzing DeFi contract logic.
Input Context:
- {Source_Code}: Target function code with internal callees and relevant data structures.
- {Call_Graph}: Direct and transitive call relationships.
- {Behavior}: Read/write sets and access modifiers.

CoT Instructions: Analyze the function and distill its **business intent**. In particular:
(1) **Business Role**: The purpose the function serves within the contract.
(2) **Execution Logic**: Key control-flow and state-modifying steps.
(3) **Operational Constraints**: Preconditions, access rules, and assumptions.

Output Format: Structured Markdown summary describing the function-level business intent.

Prompt Template: Contract-Level Intent Synthesis

Role: You are a smart contract architect synthesizing global contract logic.
Input Context:
- {Func_Intents}: Function-level business intent summaries.
- {Doc_Specs}: Optional developer-provided documentation.

CoT Instructions: Synthesize a **contract-level business intent model**. Specifically:
(1) **Core Variables**: State variables central to business logic.
(2) **Business Phases**: Distinct phases in the contract's business lifecycle, and how function invocations transition the contract between these phases.
(3) **Permission Structure**: Roles and access constraints governing phase transitions and variable updates.

Output Format: Structured Markdown report describing the contract-level business intent.

Figure 4: Prompt templates for business intent distillation. Function-level intents are extracted (left) and aggregated into a unified contract-level business intent model (right).

Formally, the expression space is $\mathcal{E} = \mathcal{S} \cup \mathcal{T} \cup \mathcal{C} \cup \mathcal{G}$, where $\mathcal{S}$ denotes storage variables, $\mathcal{T}$ transaction variables (e.g., `msg.sender`), $\mathcal{C}$ constants, and $\mathcal{G}$ derived ghost expressions (e.g., `Old`$(\cdot)$ and aggregate mapping summaries). Candidate invariants are generated by pairing type-compatible expressions with operators in $\{==, \leq, \geq, \in\}$. We adopt a lightweight invariant language to balance expressiveness and runtime efficiency. While more complex relational predicates can describe additional program behaviors, they provide limited additional value for runtime context filtering under fuzzing and introduce extra synthesis and checking overhead. The selected templates capture commonly occurring business-logic relations, such as authorization, accounting consistency, and cross-state transitions, while providing informative runtime signals with efficient incremental evaluation.

4.1.2 Hierarchical Intent Specifications Extraction. RcAudit adopts a two-stage context distillation process that incrementally extracts contract business intent into a structured intent specification.

Stage 1: Function-Level Intent Distillation. For each external function, RcAudit aggregates its relevant source code, call relations, and statically extracted behavioral features. Guided by the prompt shown in the left panel of Figure 4, the LLM summarizes this information into a concise function-level specification that captures the function's business intent, execution flow, and access constraints.

Stage 2: Contract-Level Intent Synthesis. The resulting function-level specifications, optionally augmented with developer documentation, are then combined to form a contract-level business intent specification. Using the prompt in the right panel of Figure 4, the LLM synthesizes a global view of the contract by identifying core state variables, business phases, and role-based permission structures. This contract-level intent serves as a foundation for subsequent fuzzing guidance and logic reasoning.

4.2 Runtime Context Generation

The Runtime Context Generation stage uses the extracted intent to guide the discovery of concrete evidence during execution. RcAudit employs a state- and intent-aware fuzzing strategy to drive the exploration to logic-critical states.

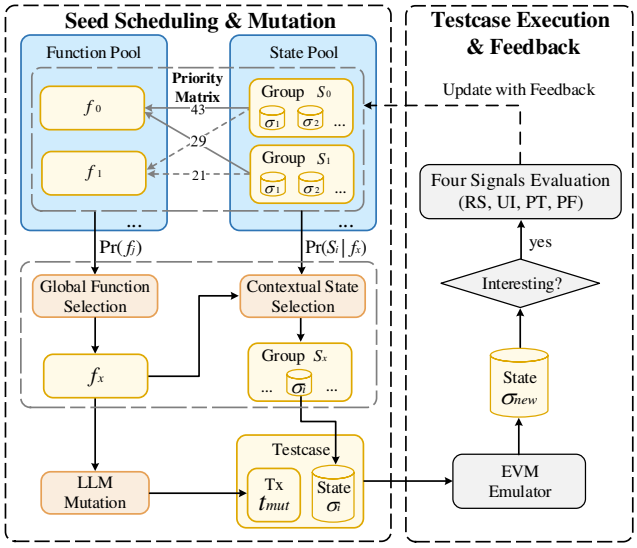


Figure 5: Runtime Context Generation of RcAudit

4.2.1 Dry Run & Initialization. The fuzzing process begins with a Dry Run phase, where historical on-chain transactions of the target contract are replayed in an instrumented EVM emulator. The resulting execution traces are used to bootstrap the initial seed corpus.

Seed Pool Initialization. As shown in Figure 5, RcAudit maintains two pools: a Function Pool, containing all externally callable functions, and a State Pool, which stores parsed storage snapshots as concrete contract states. We organize collected states into groups $\mathcal{S}_i$ according to the function that generated them. This grouping reflects a key observation: executions triggered by the same function tend to induce similar storage mutation patterns and correspond to related semantic contexts. For example, states generated by the `withdraw` function typically exhibit consistent updates to user balances, forming a coherent semantic cluster. By using functions as a proxy for behavioral equivalence, RcAudit clusters states into coarse-grained semantic classes, enabling efficient reuse of analysis results across similar contexts. During the dry run, each collected state is inserted into its corresponding state group.

State Evaluation & Priority Initialization. RcAudit maintains a priority matrix $\mathcal{A}$, where each entry $A_{i,j}$ quantifies the priority of invoking function f_j under state group $\mathcal{S}_i$. By sampling states from the initial corpus and evaluating their logical relevance to the target functions, RcAudit constructs an intent-aware priority distribution that enables the fuzzer to prioritize logic-sensitive execution paths during the early stages of exploration.

To control profiling overhead, the first state in each group is always evaluated, while subsequent states are sampled with probability c (default 0.01), reducing LLM invocation cost without affecting vulnerability reasoning. This sparse sampling strategy, coupled with an asynchronous priority update mechanism, ensures that the latency of LLM reasoning does not become a bottleneck for the execution of the fuzzer. For a selected state σ_k, RcAudit leverages the *State- and Intent-Aware Priority Scoring* prompt (Figure 6, left panel) to guide the LLM's reasoning. By grounding the model in the parsed storage snapshot and extracted business intent, RcAudit

Prompt Template: State- and Intent-Aware Priority Scoring

Role: You are a senior adversarial fuzzing strategist.

Input Context:
- {Runtime_Context}: Parsed storage snapshot and transaction history.
- {Business_Intent}: Contract- and function-level intent specifications.

CoT Instructions: Evaluate which functions should be prioritized for fuzzing in the current state. Score each function from 0 to 100 based on:
(1) **Rationality Score**: Plausibility given current state and caller.
(2) **Unintended Impact**: Likelihood of violating invariants.
(3) **Phase Transition**: Potential to advance business logic.
(4) **Precondition Fulfillment**: Ability to set up future calls.

Output Format: JSON mapping each function to a four-dimensional priority score.

Prompt Template: Logic-Driven Transaction Mutation

Role: You are an expert in smart contract interactions.

Input Context:
- {Target_Func}: Intent specification of the scheduled function.
- {Runtime_Context}: Current state variables and sender candidates.
- {Invariants}: Invariants currently holding.

CoT Instructions: Generate a concrete transaction to execute the target function. In particular:
(1) **Boundary Probing**: Explore edge and extreme values.
(2) **Invariant Targeting**: Craft inputs to falsify specific invariants.
(3) **Logic Advancement**: Satisfy constraints for state transitions.

Output Format: JSON containing encoded calldata and generation rationale.

Figure 6: Prompt templates used in the fuzzing loop. The scoring prompt (left) prioritizes functions based on runtime context and business intent, while the mutation prompt (right) generates concrete transactions to drive execution.

tasks it with evaluating each candidate function f_j specifically from the perspective of the current contract state.

The evaluation is grounded in four complementary signals:

- **Rationality Score (RS)** reflects whether invoking f_j is plausible under the current state and caller context;
- **Unintended Impact (UI)** estimates the likelihood that the invocation may trigger abnormal or violate established invariants;
- **Phase Transition (PT)** captures the potential of f_j to advance the contract's business logic;
- **Preconditions Fulfillment (PF)** measures whether the call helps satisfy conditions required for subsequent transactions of high impact.

The LLM assigns each signal a numerical score in the range $[0, 100]$.

RcAudit then synthesizes these scores across four dimensions using Algorithm 2 to derive an individual state-function priority $v_{k,j}$. The aggregation process prioritizes functions that are both applicable to the logic and relevant to security, while incorporating state-transition potential and precondition value. For each state group $\mathcal{S}_i$, the corresponding group-level priority $A_{i,j}$ is initialized as the average of the scores obtained from all profiled states within the group. The scoring and aggregation are designed to stabilize noisy LLM outputs and extract consistent guidance through repeated evaluation, making the approach robust to specific parameter choices.

4.2.2 Seed Scheduling & Mutation. To explore the state space efficiently, RcAudit employs a scheduling strategy with two stages that relies on the priority matrix $\mathcal{A}$ to select seeds. This is followed by a mutation process driven by the contract logic, as illustrated in Figure 5.

Intent-Aware Seed Scheduling. Seed selection is guided by the priority matrix $\mathcal{A}$. To ensure non-zero exploration probability for state groups not observed during the dry run, Laplace smoothing is applied to all priority scores.

In the first stage, RcAudit computes an aggregated priority $T_j = \sum_{i=1}^{n}(A_{i,j}+\alpha)$ for each function across all n state groups, where α is the smoothing constant (default $\alpha = 10$). A target function f_x

Algorithm 2: State Evaluation Score Aggregation

Require: • Context of a single state σ_k (belonging to group $\mathcal{S}_i$),
• For each function f_j ($j = 1, \ldots, m$), LLM-provided scores: $(RS_{k,j}, UI_{k,j}, PT_{k,j}, PF_{k,j}) \in [0, 100]$,
• Thresholds $\theta_{RS}, \theta_{UI} > 0$, and boost factor $\alpha_{boost} \geq 0$ (*default:* $\theta_{RS} = 60$, $\theta_{UI} = 50$, $\alpha_{boost} = 0.3$).
Ensure: Individual state score $v_{k,j}$ for each function f_j.

for each function f_j in the function pool **do**
  $RS_{\text{adj}} \leftarrow RS_{k,j}$
  if $RS_{k,j} > \theta_{RS}$ **and** $UI_{k,j} > \theta_{UI}$ **then**
    $RS_{\text{adj}} \leftarrow RS_{k,j} + \alpha_{boost} \cdot (UI_{k,j} - \theta_{UI})$
  end if
  $v_{k,j} \leftarrow \dfrac{RS_{\text{adj}} + PT_{k,j} + PF_{k,j}}{3}$
end for
return $\{v_{k,j}\}_{j=1}^{m}$

is then sampled with probability:

$$\Pr(f_j) = \frac{T_j}{\sum_{k=1}^{m} T_k}. \tag{1}$$

In the second stage, conditioned on the selected function f_x, RcAudit samples a state group $\mathcal{S}_i$ according to its function-specific priority:

$$\Pr(\mathcal{S}_i \mid f_x) = \frac{A_{i,x} + \alpha}{\sum_{k=1}^{n}(A_{k,x} + \alpha)}. \tag{2}$$

A concrete state σ is then uniformly drawn from $\mathcal{S}_i$ to serve as the base state for mutation.

Context-Aware Transaction Mutation. Given a selected function f_x and base state σ, RcAudit queries the LLM to synthesize transaction candidates using the provided runtime context. As shown in Figure 6 (right panel), the prompt provides the current state σ, the business intent specifications of the target function, and the invariants currently holding. The LLM is instructed to generate inputs that probe boundary conditions, drive state transitions, or attempt to violate existing invariants, enabling logic-driven transaction synthesis.

The generated transactions are designed to exercise semantically meaningful program behaviors. For example, when targeting a `withdraw(amount)` function, the LLM consults the parsed storage snapshot to generate amounts informed by the user's current balance, making the resulting transactions more likely to satisfy prerequisite conditions while exercising semantically meaningful behaviors.

4.2.3 Testcase Execution. Each generated test case is executed from the base state σ_{old} that contains a parsed storage snapshot. During execution, the instrumented EVM emulator synchronizes low-level storage slot updates with their corresponding variables in the snapshot. A human-readable post-state σ_{new} is materialized when the execution completes successfully.

State Pool Updating & Runtime Context Recording. Upon successful execution, RcAudit examines the resulting post-state to determine whether it should be retained. Following the feedback-driven criterion adopted in ItyFuzz [34], post-states that involve writes to previously accessed storage slots are considered interesting and incorporated into the state pool.

For each incorporated state, RcAudit records the corresponding *runtime context*, consisting of the pre-state σ_{old}, the executed transaction, and the post-state σ_{new}. This contextual information is preserved for subsequent analysis and forwarded to the *Runtime-Context-Based Logic Reasoning* stage.

Dynamic Priority Refinement. When a new interesting post-state σ_{new} is discovered, RcAudit identifies its corresponding state group $\mathcal{S}_i$ and, with probability c (Section 4.2.1), evaluates the state using the same LLM-based scoring procedure as in initialization, producing an individual score $v_{\text{new},j}$ for each candidate function. The resulting score is incorporated into the group-level priority $A_{i,j}$ via a time-decaying weighted average:

$$A_{i,j} \leftarrow \beta \cdot A_{i,j} + (1-\beta) \cdot v_{\text{new},j}, \tag{3}$$

where $\beta = e^{-\lambda \Delta t}$ controls the influence of historical observations. This time-decay mechanism attenuates the impact of uncertainty in individual LLM-assigned scores while enabling $A_{i,j}$ to capture the evolving distribution of states within each group. We set λ such that priorities exhibit a 5-minute half-life, balancing stability with timely incorporation of newly observed states without affecting the underlying detection logic.

4.3 Runtime-Context-Based Logic Reasoning

In the final stage, RcAudit turns runtime contexts into confirmed reports through structured logic reasoning. Most contexts are benign, and only a small fraction contradict inferred intent or violate invariants. To isolate and validate these anomalies, RcAudit adopts a three-step reasoning pipeline: (1) *Invariant Set Maintenance & Context Filtering* to isolate suspicious runtime contexts; (2) *Invariant-Violation-Oriented Execution Analysis* to reason about potential logic flaws; and (3) *Evidence-First Adjudication* to conservatively validate findings against the inferred business intent.

4.3.1 Invariant Set Maintenance & Context Filtering. The first step serves as a high-throughput runtime filter that identifies distinct semantic state transitions for downstream analysis. RcAudit maintains the candidate invariant set $\mathcal{I}$ initialized in Section 4.1.1. Different from invariants used as correctness specifications, these candidates represent tentative behavioral properties inferred from the explored executions. During fuzzing, newly discovered executions may invalidate such properties as the exploration space expands. Since fuzzing often revisits similar execution regions, retaining an already falsified invariant provides limited additional information. Therefore, for each incoming state σ_{new}, invariants that are violated by a valid execution are removed from $\mathcal{I}$, and the corresponding runtime context is flagged for further analysis:

$$\sigma_{\text{new}} \not\models \iota \;\Rightarrow\; \mathcal{I} \leftarrow \mathcal{I} \setminus \{\iota\},\ \text{flag}(\sigma_{\text{new}}). \tag{4}$$

This adaptive pruning mechanism reduces redundant runtime evidence while allowing the remaining invariant set to reflect unexplored behaviors. For fuzzing-oriented exploration, the efficiency gains from reducing redundant monitoring outweigh the potential loss of observing repeated violations of the same behavioral property, allowing RcAudit to focus runtime analysis on diverse semantic transitions.

Prompt Template: Runtime Logic Auditor

Role: You are a security auditor specializing in smart contract logic.
Input Context:
- `{Violated_Inv}`: Falsified invariants and violation history.
- `{Trace}`: Transaction sequence and state diffs.
- `{Business_Intent}`: Contract- and function-level business intent.

CoT Instructions: Analyze the invariant violation to identify potential logic flaws. Specifically:
(1) **Interpret**: Reconstruct the execution path leading to the violation.
(2) **Hypothesize**: Determine whether the violation indicates a logic flaw.
(3) **Assess**: Estimate severity and exploitability.

Output Format: JSON with vulnerability description, reproduction steps, and severity scores.

Prompt Template: Evidence-Based Arbiter

Role: You are a conservative security reviewer validating audit findings.
Input Context:
- `{Candidates}`: Vulnerability hypotheses from the Auditor.
- `{Business_Intent}`: Contract- and function-level business intent.

CoT Instructions: Validate reported vulnerabilities against intended business logic. In particular:
(1) **Validate**: Reject behavior consistent with intent.
(2) **Deduplicate**: Merge equivalent root causes.
(3) **Re-score**: Calibrate severity using verified evidence.

Output Format: Final consolidated and validated JSON report.

Figure 7: Prompt templates for the two-perspective runtime reasoning stage. The Runtime Logic Auditor (left) explores invariant violations, while the Evidence-Based Arbiter (right) conservatively validates findings.

4.3.2 Invariant-Violation-Oriented Execution Analysis. For each flagged context, the *Runtime Logic Auditor* analyzes the violation to pinpoint logical inconsistencies. As illustrated in the left panel of Figure 7, the Auditor is provided with the violated invariant, the triggering transaction sequence with associated state diffs, and the inferred business intent at both the contract and function levels. Grounded in this runtime context, the LLM is instructed to reason specifically about the violated property rather than performing unconstrained bug hunting.

Guided by the prompt, the Auditor reconstructs the execution path leading to the violation and examines how the observed state changes diverge from the expected business intent. When the violation cannot be explained as a benign edge case or intended behavior, the Auditor formulates a vulnerability hypothesis, including a concrete description of the issue, reproduction steps, and an initial assessment of severity and exploitability.

4.3.3 Evidence-First Adjudication. While the Auditor is designed to be sensitive to potential logic flaws, RcAudit employs an *Evidence-Based Arbiter* to conservatively validate reported findings and control false positives. As shown in the right panel of Figure 7, the Arbiter cross-validates the auditor's hypotheses against the business intent, which serves as the authoritative reference.

The Arbiter discards hypotheses whose observed behaviors, despite violating certain invariants, remain consistent with the intended business logic (e.g., exceptional but explicitly supported control paths). It further merges reports that stem from the same underlying root cause and re-calibrates severity scores based solely on verified runtime evidence. This design decouples exploration from adjudication. By balancing inquisitive reasoning with conservative validation, RcAudit detects subtle, state-dependent inconsistencies without sacrificing the precision needed for real-world audits.

5 Evaluation

This section presents a comprehensive evaluation of RcAudit. We address the following research questions through various experiments:

- **RQ1: Effectiveness of State- and Intent-Aware Fuzzing**: How effective is RcAudit's state- and intent-aware fuzzing strategy at improving exploration effectiveness and reaching logic-critical runtime contexts?
- **RQ2: Analysis of Runtime-Context-Based Logic Reasoning**: How does RcAudit transform runtime evidence into validated vulnerability findings?
- **RQ3: Effectiveness of Logic Vulnerability Detection**: How effectively does RcAudit detect logic vulnerabilities compared to state-of-the-art smart contract analysis tools?
- **RQ4: Zero-Day Vulnerability Discovery**: Can RcAudit discover previously unknown vulnerabilities in real-world smart contracts?

Experimental Environment. All experiments were conducted on a 12-core Intel i7-12700 machine running Ubuntu 24.04 LTS.

Tool Implementation. We implemented RcAudit with about 15K lines of Rust (runtime context generation) and 3K lines of Python (LLM orchestration), with integrated Slither [14] and Aderyn [11] for static analysis. RcAudit reuses ItyFuzz [34]'s native execution engine and fuzzing infrastructure. We implement a customized scheduling module together with execution instrumentation for runtime state parsing, invariant set maintenance, and runtime context recording. LLM-synthesized transactions are incorporated into the fuzzing campaign as additional seeds for subsequent exploration. The LLM-based components operate asynchronously with respect to execution: state evaluation for priority refinement and transaction candidate synthesis are performed in background processes, and their results are incorporated into the fuzzing campaign when available.

Detailed model comparisons across pipeline roles, including effectiveness–latency–cost trade-offs and a hyperparameter sensitivity check, are provided in the Supplementary Material. To balance reasoning quality, latency, and cost, we use GPT-5 [30] for intent extraction and adjudication, DeepSeek-V3.1-Terminus [13] for fuzz guidance, and Claude Haiku 3.5 [1] for execution-trace reasoning. Each experiment is repeated three times and we report medians.

Table 1: Datasets for Evaluation

#	Source	Projects	Contracts
Dataset 1	DeFiHackLabs [38]	132	1,082
Dataset 2	Top Market-Cap Contracts	18	185
Dataset 3	Real-World Auditing Projects	13	296

Datasets. Table 1 summarizes the three datasets. Dataset 1 comprises 132 projects (1,082 contracts) associated with logic-flaw incidents from DeFiHackLabs. We apply minimal filtering to logic-flaw-tagged entries, excluding only those without public source code or with contracts too old for reliable reproduction under current toolchains. Dataset 2 contains 18 high-TVL audited projects (185 contracts), split into small (≤1,000 LoC, 8 projects) and large (>1,000 LoC, 10 projects) groups. Dataset 3 contains 13 ongoing audit/bounty projects (296 contracts) from Code4rena [8], Cantina [4], and Sherlock [32]. We manually deduplicated projects and contracts across all sets.

5.1 RQ1: Effectiveness of State- and Intent-Aware Fuzzing

To evaluate the effectiveness of state- and intent-aware fuzzing, we conduct a controlled ablation study that isolates two guidance mechanisms: *intent-aware seed scheduling* and *context-aware transaction mutation*. From Dataset 1, we construct an ablation subset consisting of eight representative projects that span diverse logic-flaw families.[1] For each project, we manually identify one or more logic-critical invariants whose violation corresponds to the known exploit condition.

We compare four configurations: (1) **Full**, with both intent-aware scheduling and context-aware transaction mutation enabled; (2) **w/o Intent Sched.**, which replaces the intent-aware seed scheduling with the original power scheduling strategy used in ItyFuzz while retaining context-aware transaction mutation. (3) **w/o Context Mut.**, which disables the incorporation of LLM-synthesized transactions while retaining intent-aware scheduling; and (4) **w/o Guidance**, which disables both guidance mechanisms and serves as the LLM-free baseline.

All variants are executed for 20 minutes per project under identical settings. We evaluate each configuration using following metrics: the fraction of projects in which at least one logic-critical invariant is violated during fuzzing (Target Hit Rate); the mean time until first violation of a logic-critical invariant (Time-to-Target); the mean final branch coverage (Cov.); and the median number of runtime contexts recorded for invariant violations detected during fuzzing, including but not limited to logic-critical ones (Viol. Ctx.).

Table 2: Component ablation results on the ablation subset.

Variant	Target Hit Rate	Time-to-Target (s)	Cov.	Viol. Ctx.
Full	100.0%	271.1	46.58%	86
w/o Intent Sched.	100.0%	324.9	45.27%	102
w/o Context Mut.	75.0%	489.2	44.17%	70
w/o Guidance	37.5%	875.2	42.84%	66

Table 2 summarizes the results. The full system successfully triggers the violation of logic-critical invariants in all projects, whereas the unguided baseline triggers only 37.5%. Compared with the unguided baseline, RcAudit also reduces the mean time-to-target from 875 s to 271 s. Moreover, the guided configurations achieve higher branch coverage and record more invariant-violating runtime contexts for downstream reasoning.

The two guidance mechanisms contribute in complementary ways. Removing context-aware mutation reduces the target hit rate to 75% and increases the mean time-to-target to 489 s, indicating that context-aware transaction mutation is important for reaching logic-critical runtime contexts. In contrast, removing intent-aware scheduling preserves the target hit rate but increases the time-to-target from 271 s to 325 s. Although it records more invariant-violating contexts (102 vs. 86), these additional contexts do not lead to faster target hits, suggesting that intent-aware scheduling improves the quality of explored runtime contexts. Together, enabling both mechanisms achieves the best overall performance, combining higher target hit rate with more efficient exploration.

[1] Detailed per-project statistics can be found in the Supplementary Material.

Execution Instrumentation Cost Analysis. In state- and intent-aware fuzzing, both guidance mechanisms rely on runtime information collected by execution instrumentation. The corresponding LLM-based reasoning components process this information asynchronously. Therefore, the execution throughput is primarily determined by the instrumentation overhead. To quantify the performance impact of execution instrumentation, we compare the instrumented backend with an otherwise identical backend without instrumentation enabled under the same workload. On the ablation subset, the instrumented backend executes 1,239 executions per second on average, compared with 7,761 executions per second for the backend without instrumentation, corresponding to a 6.3× slowdown. Despite the overhead, the resulting throughput remains sufficient for practical fuzzing while supporting semantic guidance that provides significant effectiveness gains.

Summary for RQ1: RcAudit's state- and intent-aware fuzzing substantially improves exploration effectiveness, with scheduling and mutation providing complementary benefits.

5.2 RQ2: Analysis of Runtime-Context-Based Logic Reasoning

This research question analyzes how RcAudit transforms runtime contexts collected during fuzzing into validated vulnerability findings through invariant-based context filtering and evidence-based logic reasoning. The analysis is conducted on the same ablation subset used in RQ1.

Invariant Filtering Analysis. We first quantify the evolution of the candidate invariant set during exploration together with the runtime contexts collected before and after invariant-based filtering. As shown in Table 3, RcAudit instantiates a median of 21,029 candidate invariants per project. During exploration, these candidates are progressively eliminated as newly observed executions contradict them. Although eliminating invalidated invariants may suppress subsequent violations of the same property, our later analysis of the retained contexts shows that they still provide sufficient evidence for downstream reasoning while containing substantial redundant observations due to the repetitive nature of fuzzing exploration. Because many instantiated invariants share common state variables or expression components, a single runtime state can invalidate multiple candidates simultaneously. A median of 4,479 candidates remain active throughout the fuzzing campaign. The runtime context generation stage records a median of 1,966 distinct runtime contexts per project, of which 86 invariant-violating contexts are retained for downstream reasoning.

Table 3: Invariant filtering statistics on the ablation subset.

Metric	Median
Candidate invariants instantiated	21,029
Candidate invariants still holding at run end	4,479
Distinct runtime contexts generated	1,966
Invariant-violating runtime contexts retained	86

To characterize the types of properties captured by the filtering stage, we tag each falsified candidate invariant instance with the expression features it contains: an `Old(·)` *reference* to a pre-transaction value, an *aggregate sum* over a mapping or array, and a *variable-to-constant relation* (e.g., $x == c$). The features are not mutually exclusive. For example, an invariant comparing an `Old(·)` value against an aggregate sum contributes to both counts.

Table 4: Expression features among 112,725 falsified candidate invariant instances.

Expression feature	Count (%)
`Old(·)` reference	79,860 (70.8%)
Aggregate sum	74,572 (66.2%)
Variable-to-constant relation	3,801 (3.4%)

Table 4 shows that the violated evidence is dominated by cross-state and accounting relations: 70.8% of falsified candidate instances reference pre-transaction state and 66.2% involve mapping/array aggregates, while only 3.4% are variable-to-constant relations. These cross-state and accounting properties frequently co-occur, capturing transitions in which an updated state variable becomes inconsistent with its previous value or an aggregate balance, which are common patterns of business logic flaws.

Representative Case. *AzukiDAO* is a token claim contract that records whether a user has already claimed rewards, but fails to check this marker before processing subsequent claims. During exploration, RcAudit maintains function-level invariants from observed executions. For the claim function, one inferred invariant captures that a claim is only processed when the user's claim marker is unset: `Old(claimed)[user] == false`. This invariant remains consistent with previously observed claim executions until a replayed claim is accepted after the marker has already been set, violating the inferred transition property. The filtering stage retains the corresponding runtime context for downstream reasoning, which identifies the missing claim status check as the root cause.

Evidence-Based Logic Reasoning Analysis. We next analyze how the Runtime Logic Auditor and Evidence-Based Arbiter transform the retained runtime contexts into validated findings. Across all projects in the ablation subset, invariant-based filtering retains 696 invariant-violating runtime contexts in total. From these contexts, the Runtime Logic Auditor generates 21 vulnerability hypotheses. After Evidence-Based Arbiter adjudication, 8 hypotheses are retained as confirmed findings, while the remaining 13 are consolidated or rejected.

Among the 21 hypotheses, 20 are duplicate descriptions of the retained findings generated from different violating contexts and are consolidated during adjudication. This indicates that, under the current filtering strategy, the retained contexts are still dominated by repeated observations of the same logic issues. The remaining rejected hypothesis originates from a fuzzing-induced state anomaly caused by an unrealistic input condition. In this case, the auditor interprets an execution snapshot containing an abnormally inflated reward balance as a potential accounting flaw. The arbiter, however, verifies the corresponding transaction evidence and determines that the reported state results from an unrealistic mutation condition rather than a state reachable under normal contract execution, rejecting the hypothesis as an unsupported interpretation.

Summary for RQ2: Invariant-based context filtering effectively isolates semantically meaningful runtime contexts, while evidence-based logic reasoning consolidates duplicated hypotheses and validates findings.

5.3 RQ3: Effectiveness of Logic Vulnerability Detection

We evaluate the effectiveness of RcAudit by comparing it against GPTScan [37], SmartInv [40], PromFuzz [26], and ItyFuzz [34]. For fairness, we align key evaluation settings across baselines: fuzzing-based methods are run with a unified 20-minute budget per target to match RcAudit. Since GPTScan and PromFuzz rely on general-purpose large language models for analysis, we configure both with the same foundation model (GPT-5) used in RcAudit for key reasoning roles, enabling a backbone-controlled comparison. Preliminary experiments show that newer-generation models such as GPT-5 consistently outperform earlier versions (e.g., GPT-4) within these pipelines; thus, we adopt GPT-5 to ensure a stronger and fairer comparison. Other relevant tools, such as PropertyGPT [27], were excluded due to the unavailability of their source code. For RcAudit, historical on-chain transactions used in the dry-run phase do not include any post-exploit information.

To comprehensively evaluate RcAudit, we conduct three complementary evaluations targeting different aspects of detection effectiveness. Dataset 1 is used for (1) exploit root cause rediscovery and (2) overall vulnerability detection quality, while Dataset 2 evaluates false positive behavior on extensively audited high-TVL projects. All reported vulnerabilities were first independently assessed by three frontier LLMs. Disagreements were manually adjudicated, and final labels were verified by two authors.

Exploit Root Cause Rediscovery. For exploit rediscovery, each reported vulnerability is compared against the documented exploit root cause. A report is considered a match if both describe the same underlying logical flaw. As summarized in Table 5, RcAudit achieves the highest exploit root cause recall (56.1%), successfully rediscovering 74 of the 132 historical incidents, outperforming SmartInv (34/132), GPTScan (9/132), PromFuzz (3/132), and ItyFuzz (3/132).

Table 5: Exploit root cause rediscovery on Dataset 1.

Metric	RcAudit	SmartInv	GPTScan	PromFuzz	ItyFuzz
Root cause matches	74	34	9	3	3
Missed incidents (FN)	58	98	123	129	129
Recall	56.1%	25.8%	6.8%	2.3%	2.3%

We analyze the 58 incidents that remain undetected in Dataset 1. These misses are dominated by *cross-contract interactions*, where violations arise from interdependent state changes across multiple contracts rather than within a single contract's local state. This challenge is not unique to RcAudit: the baselines exhibit the same blind spot, and a small number of misses also stem from the intrinsic difficulty of reaching exploit-triggering conditions through fuzzing or from conservative filtering in final adjudication.

Overall Vulnerability Detection. We next evaluate overall vulnerability detection quality on Dataset 1. Unlike exploit rediscovery, which considers only historical exploit root cause matches, this evaluation counts every independently validated logical vulnerability as a true positive. A finding is labeled as a TP if the reported behavior reveals a reproducible logical inconsistency supported by runtime evidence and confirmed during manual validation. Therefore, multiple TPs may be identified for the same contract.

PromFuzz and ItyFuzz are excluded from this experiment because their fuzzing methodologies are not directly comparable for this precision-focused setting. The left half of Table 6 reports the resulting true positives (TPs), false positives (FPs), and precision values.

Table 6: Precision on Dataset 1 and false positive evaluation on Dataset 2.

Tool	Dataset 1 (Precision)			Dataset 2 (FP)		
	TP	FP	Precision[1]	SG[2]	LG[3]	FPs
RcAudit	253	8	96.9%	0	0	0
GPTScan	51	284	15.2%	0	0	0
SmartInv	140	118	54.3%	6	16	22

[1] Precision: $(TP/(TP+FP)) \times 100\%$.
[2] SG: Small Group in Dataset 2. [3] LG: Large Group in Dataset 2.

RcAudit reports the largest number of genuine findings (253) while retaining high precision (96.9%) on Dataset 1, demonstrating that the integrated runtime-evidence pipeline scales to a substantially broader incident benchmark. SmartInv produces 118 false alarms (precision 54.3%), showing that static, code-only invariant synthesis still struggles to distinguish benign deviations from real logic flaws. Even with the upgraded GPT-5 backbone, GPTScan yields 51 true positives with 15.2% precision, indicating that pure code-centric reasoning remains insufficient without dynamic runtime evidence.

False Positive Evaluation on Benign Projects. To further assess precision, we evaluated the tools on Dataset 2, which consists of high-market-capitalization projects that have undergone extensive professional audits and are expected to contain few exploitable logic flaws. This dataset is therefore well suited to characterizing false-positive behavior. PromFuzz and ItyFuzz were again excluded from this experiment since their fuzzing methodologies are not directly comparable for this task.

As shown in the right half of Table 6, RcAudit and GPTScan produce no false positives across all high-TVL contracts in Dataset 2, while SmartInv still reports 22 spurious alerts. Although Dataset 1 exposes a non-zero FP tail for RcAudit, its precision remains substantially higher than both baselines, and its zero-FP behavior on Dataset 2 indicates that the residual errors are concentrated in ambiguous incident contexts rather than benign production protocols in general. Taken together, results on both datasets indicate that RcAudit's integration of runtime invariant evaluation provides the concrete execution evidence needed to distinguish true logic violations from benign state changes far more reliably than code-only approaches.

Summary for RQ3: On Dataset 1, RcAudit rediscovered 74/132 historical exploit root causes and identified 253 independently validated vulnerabilities with 96.9% precision. On Dataset 2, RcAudit produced no false positives on extensively audited high-TVL projects.

5.4 RQ4: Zero-Day Vulnerability Discovery

To evaluate RcAudit's effectiveness in discovering previously unknown vulnerabilities, we applied it to Dataset 3, which consists of production-level DeFi protocols undergoing security audits, providing a realistic setting for assessing zero-day detection capability.

```
contract TokenRewards {
  mapping(address => uint) public excluded; // User debt
  mapping(address => uint) public realized; // User gains
  bool public paused;

  function setPaused(bool _p) external { paused = _p; }

  function deposit(uint amount) external {
     _distribute(msg.sender);     // 1. Payout pending
     _resetExcluded(msg.sender);  // 2. Reset debt
   }
  function _distribute(address user) internal {
    if (paused) return; // Correctly checks pause
    if (pending > 0) realized[user] += pending; }

  function _resetExcluded(address user) internal {
    // Vulnerability: Updates debt regardless of pause
    excluded[user] = ...; } }
```

Figure 8: Simplified logic of the `TokenRewards` vulnerability. When `paused` is true, the contract skips reward distribution but still updates the user's debt, causing permanent reward loss.

Across these projects, RcAudit discovered 22 previously unknown vulnerabilities, including 8 high-severity and 11 medium-severity logic flaws. All identified issues were responsibly disclosed, externally confirmed by the corresponding project teams and audit platforms, and subsequently mitigated. Most issues violate invariants in staking, liquidity, or reward distribution logic. We highlight one representative case below.

Case Study: TokenRewards. Figure 8 illustrates a high-severity logic flaw in a reward distribution contract. The flaw manifests only under a specific transaction sequence. When rewards are paused, `_distribute` correctly skips reward payout, but the subsequent call to `_resetExcluded` still updates the user's debt, permanently discarding accrued rewards.

Triggering the flaw requires first accumulating pending rewards while the contract is active, then pausing reward distribution before invoking another `deposit`. Existing logic-vulnerability detectors miss this flaw because the pause check itself appears correct, while the inconsistent update of `excluded` is observable only across transaction states. RcAudit first inferred that successful `deposit` executions consistently occurred when `paused == false`. After the fuzzer explored a paused `deposit`, the violated invariant triggered runtime-context analysis. Comparing pre/post-state snapshots, the LLM identified that `excluded` was updated while the contract was paused, revealing the root cause of the funds loss.

Summary for RQ4: RcAudit uncovers 22 zero-day vulnerabilities in audited production protocols, including subtle state-dependent flaws.

6 Discussion

Cross-Contract Reasoning and Invariant Scope. RcAudit currently prioritizes *intra-contract* reasoning for precision and scalability, so vulnerabilities spanning multiple interacting contracts can remain challenging. In addition, the lightweight invariant language is designed for efficient online monitoring during fuzzing, which may not capture more complex behavioral relations. Extending the invariant system with cross-contract dependencies and richer relational templates could further broaden the reasoning scope.

Dependence on Execution Coverage. Like other dynamic analysis approaches, RcAudit is fundamentally constrained by execution coverage: state mutations that cannot be reached during fuzzing cannot contribute to invariant construction or runtime reasoning. Combining fuzzing with symbolic execution or lightweight formal guidance could further improve deep-state exploration and expand the range of analyzable behaviors.

7 Related Work

Smart Contract Fuzzing. A large body of work improves smart-contract fuzzing through better feedback signals and seed scheduling. Representative examples include snapshot-based execution for efficient stateful exploration (ItyFuzz [34]), static-analysis-guided prioritization (Vulseye [25]), and hybrid static–dynamic guidance [7, 29]. Other systems [10, 20, 31, 43] further explore diverse mutation and scheduling strategies. Despite these advances, most fuzzers are still driven by low-level execution signals or predefined vulnerability oracles, which limits their ability to capture intent-level semantics and expose subtle, state-dependent logic flaws.

LLM-Assisted Smart Contract Security Analysis. Recent LLM-based analyzers [22, 37, 42, 44] reason over source code with prompts and auxiliary static information. While effective at extracting semantic cues, these methods remain largely code-centric and weakly grounded in concrete runtime behavior. LLMs have also been introduced into fuzzing pipelines for seed generation and prioritization [16, 33, 36]. Our approach is complementary to this line of work: we focus on state-aware online guidance during runtime-context generation (Section 4.2), which can directly incorporate stronger LLM seed-generation modules from prior studies.

Smart Contract Invariant Detection. Another line of work studies automatic invariant extraction. Some methods infer invariants from historical on-chain traces [28, 35], whereas others synthesize invariants from source code using LLMs [26, 27, 40]. Unlike approaches that treat mined invariants as fixed specifications, RcAudit uses invariants as runtime signals. This design enables effective logic-vulnerability detection without requiring complete or perfectly accurate specifications.

8 Conclusion

In this paper we presented RcAudit, a novel system for detecting smart contract logic vulnerabilities via runtime context–enhanced LLM reasoning. RcAudit employs state- and intent-aware fuzzing together with dynamically inferred invariants to generate and isolate logic-critical runtime contexts, enabling vulnerability reasoning grounded in concrete execution evidence. Our evaluation shows that RcAudit substantially outperforms existing approaches in detecting real-world logic vulnerabilities with high precision, and successfully uncovers 22 previously unknown vulnerabilities in real-world audits.

Acknowledgments

This work was supported in part by the National Natural Science Foundation of China under Grant No. 62425201.

Data Availability Statement

To support reproducibility, we archive RcAudit in DOI-backed repositories: the implementation and orchestration scripts are available as a software artifact [18], and the benchmark inputs and results are available as a data artifact [17].

References

[1] Anthropic. 2025. Claude Haiku. https://www.anthropic.com/claude/haiku.

[2] Beosin. 2025. 2025 Global Web3 Security Report. https://www.beosin.com/resources/2025_Global_Web3_Security_Report.pdf.

[3] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. 2020. Ethainter: a smart contract security analyzer for composite vulnerabilities. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 454–469. https://doi.org/10.1145/3385412.3385990

[4] Cantina. 2025. Cantina. https://cantina.xyz/.

[5] Certik. 2025. Hack3d: The Web3 Security Quarterly Report - Q2 + H1 2025. https://www.certik.com/resources/blog/hack3d-the-web3-security-quarterly-report-q2-h1-2025.

[6] Chong Chen, Jianzhong Su, Jiachi Chen, Yanlin Wang, Tingting Bi, Jianxing Yu, Yanli Wang, Xingwei Lin, Ting Chen, and Zibin Zheng. 2025. When chatgpt meets smart contract vulnerability detection: How far are we? *ACM Transactions on Software Engineering and Methodology* 34, 4 (2025), 1–30. https://doi.org/10.1145/3702973

[7] Jaeseung Choi, Doyeon Kim, Soomin Kim, Gustavo Grieco, Alex Groce, and Sang Kil Cha. 2021. Smartian: Enhancing smart contract fuzzing with static and dynamic data-flow analyses. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 227–239. https://doi.org/10.1109/ASE51524.2021.9678888

[8] Code4rena. 2025. Code4rena. https://code4rena.com/.

[9] ConsenSys. 2024. Mythril. https://github.com/ConsenSys/mythril.

[10] Crytic. 2025. Medusa. https://github.com/crytic/medusa.

[11] Cyfrin. 2025. Aderyn. https://github.com/Cyfrin/aderyn.

[12] Isaac David, Liyi Zhou, Kaihua Qin, Dawn Song, Lorenzo Cavallaro, and Arthur Gervais. 2023. Do you still need a manual smart contract audit? *arXiv preprint arXiv:2306.12338* (2023).

[13] DeepSeek. 2025. DeepSeek V3.1 Terminus. https://api-docs.deepseek.com/news/news250922.

[14] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: a static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 8–15. https://doi.org/10.1109/WETSEB.2019.00008

[15] Forbes. 2025. Cryptocurrency prices, market cap and charts - digital assets. https://www.forbes.com/digital-assets/crypto-prices.

[16] Keke Gai, Haochen Liang, Jing Yu, Liehuang Zhu, and Dusit Niyato. 2026. LLAMA: Multi-Feedback Smart Contract Fuzzing Framework With LLM-Guided Seed Generation. *IEEE Transactions on Information Forensics and Security* 21 (2026), 3281–3296. https://doi.org/10.1109/TIFS.2026.3666859

[17] Guang Gong, Yucong Jiang, Rong Jian, Xiaotao Feng, Zhihan Wei, Qi Li, and Ke Xu. 2026. *RcAudit: Benchmark Inputs and Evaluation Results*. https://doi.org/10.5281/zenodo.19345454

[18] Guang Gong, Yucong Jiang, Rong Jian, Xiaotao Feng, Zhihan Wei, Qi Li, and Ke Xu. 2026. *RcAudit: Implementation and Orchestration Scripts*. https://doi.org/10.5281/zenodo.19345906

[19] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–27. https://doi.org/10.1145/3276486

[20] Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. 2020. Echidna: effective, usable, and fast fuzzing for smart contracts. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*. 557–560. https://doi.org/10.1145/3395363.3404366

[21] Ákos Hajdu and Dejan Jovanović. 2019. solc-verify: A modular verifier for solidity smart contracts. In *Working conference on verified software: theories, tools, and experiments*. Springer, 161–179. https://doi.org/10.1007/978-3-030-41600-3_11

[22] Sihao Hu, Tiansheng Huang, Fatih İlhan, Selim Furkan Tekin, and Ling Liu. 2023. Large language model-powered smart contract vulnerability detection: New perspectives. In *2023 5th IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*. IEEE, 297–306. https://doi.org/10.1109/tps-isa58951.2023.00044

[23] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. Zeus: analyzing safety of smart contracts. In *Ndss*. 1–12. https://doi.org/10.14722/ndss.2018.23082

[24] Junsang Kim and Seyong Kim. 2021. A survey of decentralized finance (defi) based on blockchain. *Journal of the Korea Society of Computer and Information* 26, 3 (2021), 59–67. https://doi.org/10.9708/jksci.2021.26.03.059

[25] Ruichao Liang, Jing Chen, Cong Wu, Kun He, Yueming Wu, Ruochen Cao, Ruiying Du, Ziming Zhao, and Yang Liu. 2025. Vulseye: Detect smart contract vulnerabilities via stateful directed graybox fuzzing. *IEEE Transactions on Information Forensics and Security* (2025). https://doi.org/10.1109/TIFS.2025.3537827

[26] Xingshuang Lin, Qinge Xie, Binbin Zhao, Yuan Tian, Saman Zonouz, Na Ruan, Jiliang Li, Raheem Beyah, and Shouling Ji. 2025. PromFuzz: Leveraging LLM-Driven and Bug-Oriented Composite Analysis for Detecting Functional Bugs in Smart Contracts. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Seoul, Korea, Republic of). IEEE Press, 1044–1056. https://doi.org/10.1109/ASE63991.2025.00091

[27] Ye Liu, Yue Xue, Daoyuan Wu, Yuqiang Sun, Yi Li, Miaolei Shi, and Yang Liu. 2025. PropertyGPT: LLM-driven Formal Verification of Smart Contracts through Retrieval-Augmented Property Generation. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. The Internet Society. https://doi.org/10.14722/ndss.2025.241357

[28] Ye Liu, Chengxuan Zhang, and Yi Li. 2025. Automated invariant generation for solidity smart contracts. *IEEE Transactions on Dependable and Secure Computing* (2025). https://doi.org/10.1109/TDSC.2025.3592705

[29] Tai D Nguyen, Long H Pham, Jun Sun, Yun Lin, and Quang Tran Minh. 2020. sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of the ACM/IEEE 42nd international conference on software engineering*. 778–788. https://doi.org/10.1145/3377811.3380334

[30] OpenAI. 2025. GPT-5. https://openai.com/gpt-5/.

[31] Peng Qian, Hanjie Wu, Zeren Du, Turan Vural, Dazhong Rong, Zheng Cao, Lun Zhang, Yanbin Wang, Jianhai Chen, and Qinming He. 2024. Mufuzz: Sequence-aware mutation and seed mask guidance for blockchain smart contract fuzzing. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 1972–1985. https://doi.org/10.1109/ICDE60146.2024.00158

[32] Sherlock. 2025. Sherlock. https://sherlock.xyz/.

[33] Chaofan Shou, Jing Liu, Doudou Lu, and Koushik Sen. 2024. Llm4fuzz: Guided fuzzing of smart contracts with large language models. *arXiv preprint arXiv:2401.11108* (2024).

[34] Chaofan Shou, Shangyin Tan, and Koushik Sen. 2023. Ityfuzz: Snapshot-based fuzzer for smart contract. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 322–333. https://doi.org/10.1145/3597926.3598059

[35] Jianzhong Su, Jiachi Chen, Zhiyuan Fang, Xingwei Lin, Yutian Tang, and Zibin Zheng. 2025. SmartOracle: Generating Smart Contract Oracle via Fine-Grained Invariant Detection. *IEEE Transactions on Software Engineering* (2025). https://doi.org/10.1109/TSE.2025.3534567

[36] Jiaze Sun, Zhiqiang Yin, Hengshan Zhang, Xiang Chen, and Wei Zheng. 2025. Adversarial generation method for smart contract fuzz testing seeds guided by chain-based LLM. *Automated Software Engineering* 32, 1 (2025), 12. https://doi.org/10.1007/s10515-024-00483-4

[37] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Haijun Wang, Zhengzi Xu, Xiaofei Xie, and Yang Liu. 2024. 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*. 1–13. https://doi.org/10.1145/3597503.3639117

[38] SunWeb3Sec. 2025. DeFiHackLabs. https://github.com/SunWeb3Sec/DeFiHackLabs.

[39] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. 2018. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 67–82. https://doi.org/10.1145/3243734.3243780

[40] Sally Junsong Wang, Kexin Pei, and Junfeng Yang. 2024. Smartinv: Multimodal learning for smart contract invariant inference. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2217–2235. https://doi.org/10.1109/SP54263.2024.00126

[41] Yuepeng Wang, Shuvendu K Lahiri, Shuo Chen, Rong Pan, Isil Dillig, Cody Born, Immad Naseer, and Kostas Ferles. 2019. Formal verification of workflow policies for smart contracts in azure blockchain. In *Working Conference on Verified Software: Theories, Tools, and Experiments*. Springer, 87–106. https://doi.org/10.1007/978-3-030-41600-3_7

[42] Zhiyuan Wei, Jing Sun, Yuqiang Sun, Ye Liu, Daoyuan Wu, Zijian Zhang, Xianhao Zhang, Meng Li, Yang Liu, Chunmiao Li, Mingchao Wan, Jin Dong, and Liehuang Zhu. 2025. Advanced Smart Contract Vulnerability Detection via LLM-Powered Multi-Agent Systems. *IEEE Transactions on Software Engineering* 51, 10 (2025), 2830–2846. https://doi.org/10.1109/TSE.2025.3597319

[43] Valentin Wüstholz and Maria Christakis. 2020. Harvey: A greybox fuzzer for smart contracts. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1398–1409. https://doi.org/10.1145/3368089.3417064

[44] Lei Yu, Zhirong Huang, Hang Yuan, Shiqi Cheng, Li Yang, Fengjun Zhang, Chenjie Shen, Jiajia Ma, Jingyuan Zhang, Junyi Lu, et al. 2025. Smart-LLaMA-DPO: Reinforced Large Language Model for Explainable Smart Contract Vulnerability Detection. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 182–205. https://doi.org/10.1145/3728878

[45] Hongxiang Zhang, Yuyang Rong, Yifeng He, and Hao Chen. 2026. Llama-fuzz: Large language model enhanced greybox fuzzing. In *Proceedings of the 7th ACM/IEEE International Conference on Automation of Software Test*. 34–45. https://doi.org/10.1145/3793654.3793756

[46] Zhuo Zhang, Brian Zhang, Wen Xu, and Zhiqiang Lin. 2023. Demystifying exploitable bugs in smart contracts. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 615–627. https://doi.org/10.1109/ICSE48619.2023.00061

[47] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Weili Chen, Xiangping Chen, Jian Weng, and Muhammad Imran. 2020. An overview on smart contracts: Challenges, advances and platforms. *Future Generation Computer Systems* 105 (2020), 475–491. https://doi.org/10.1016/J.FUTURE.2019.12.019

Received 2026-03-26; accepted 2026-06-18