Exploiting Ethereum Rollback Semantics: Profit-Driven Attack Synthesis and Off-Chain Misinterpretation Testing

YIXUAN LIU, Nanyang Technological University, Singapore
XINLEI LI, Nanyang Technological University, Singapore
YI LI, Nanyang Technological University, Singapore

The Ethereum Virtual Machine (EVM) enforces atomic execution through rollback, reverting all state changes when execution fails. While necessary for correctness, rollback semantics introduce a distinct attack surface affecting both on-chain execution and off-chain infrastructures. On-chain, attackers can use conditional failure to filter executions, committing only profitable outcomes while rolling back unprofitable attempts. Off-chain, systems such as explorers, token trackers, and RPC providers may misinterpret aborted executions as successful, leading to inconsistent records or unintended transfers. Existing tools largely treat rollback as an execution endpoint, providing limited support for profit-driven attack synthesis or off-chain misinterpretation testing. To address this gap, we formalize two rollback attack models and develop RollGain, a unified framework for rollback-aware analysis. For on-chain attacks, RollGain models contract structure and value flows, validates candidate executions symbolically, and replays on a forked chain to rank profitability. For off-chain testing, RollGain conducts call tree analysis on 3.08 billion Ethereum transactions to characterize rollback patterns and exercises rollback-inducing execution vectors against external services. On our evaluation datasets, RollGain achieves 95.3% recall with zero false positives, and uncovers 20 rollback misinterpretation vulnerabilities across 18 off-chain systems, of which 18 have been confirmed, 16 fixed, and 5 assigned CVE identifiers.

CCS Concepts: • **Security and privacy** → **Web application security**; **Distributed systems security**.

Additional Key Words and Phrases: Ethereum Transactions, Rollback Attacks, Attack Synthesis, Off-Chain Inconsistencies

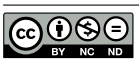
ACM Reference Format:
Yixuan Liu, Xinlei Li, and Yi Li. 2026. Exploiting Ethereum Rollback Semantics: Profit-Driven Attack Synthesis and Off-Chain Misinterpretation Testing. *Proc. ACM Softw. Eng.* 3, ISSTA, Article ISSTA155 (October 2026), 22 pages. https://doi.org/10.1145/3832246

1 Introduction

Blockchain platforms provide a shared execution environment for smart contracts, enabling decentralized applications (DApps) to manage assets and coordinate interactions without trusted intermediaries [4, 25]. The EVM has emerged as a dominant execution standard and is adopted by many widely deployed blockchains. Smart contracts running on the EVM implement application logic that controls asset transfers and state updates, while off-chain infrastructures, such as explorers, wallets, analytics services, and RPC providers, monitor and interpret contract execution to present user-facing state and functionality.

A fundamental property of the EVM execution model is *atomicity* [45]. When execution fails due to an explicit revert or an exceptional halt, all state changes performed during that execution

Authors' Contact Information: Yixuan Liu, liuy0255@e.ntu.edu.sg, Nanyang Technological University, Singapore, Singapore; Xinlei Li, xinlei003@e.ntu.edu.sg, Nanyang Technological University, Singapore, Singapore; Yi Li, yi_li@ntu.edu.sg, Nanyang Technological University, Singapore, Singapore.



ACM 2994-970X/2026/10-ARTISSTA155
https://doi.org/10.1145/3832246

are rolled back. This rollback mechanism is essential for preserving consistency and preventing partial updates. However, rollback is not merely a failure-handling detail but a core execution semantic that directly shapes transaction behavior and the interpretation of execution effects by both on-chain logic and off-chain infrastructures.

Prior work has examined rollback-related behaviors from task-specific perspectives. Some studies focus on how developers use `revert` statements in contract logic [18]. Others leverage rollback effects in specific attack strategies, such as manipulating randomness or denial-of-service attacks [12, 17, 29]. More recently, rollback semantics have been explored in cross-layer settings, for example by exploiting rollback behavior on layer-2 systems to affect bridge behavior on layer-1 [36]. Despite these efforts, existing work remains limited in two aspects. First, no approach incorporates rollback into profit-oriented attack synthesis or explicitly models loss-avoidance strategies. Second, the impact of rolled-back transactions on off-chain infrastructures has not been systematically analyzed.

To address these gaps, we formalize two transaction rollback attack models and develop RollGain, a unified framework for rollback-aware analysis across on-chain and off-chain settings. On-chain, RollGain models profit-oriented rollback attacks in which adversaries deliberately induce rollback to filter candidate executions, using graph-based representations of contract structure and asset flows to generate and validate call sequences against a rollback-aware profit guard. Off-chain, RollGain models rollback misinterpretation attacks and provides a testing module that exercises diverse rollback-inducing execution vectors, including explicit reverts and exceptional halts, and evaluates how external infrastructures interpret aborted and partially rolled-back executions based on execution traces. Together, RollGain enables end-to-end reasoning about rollback-driven attacks, from on-chain exploit synthesis to their downstream impact on off-chain systems.

Our results demonstrate both the feasibility and the real-world impact of rollback-based attacks. On-chain, RollGain synthesizes concrete exploits showing how selective rollback can avoid persistent state losses while committing only outcomes that satisfy the modeled profit guard, revealing vulnerabilities that are overlooked by conventional analysis approaches. Off-chain, our empirical study uncovers widespread rollback misinterpretation issues in deployed infrastructures, confirming that executions which fully or partially roll back state changes can lead to inconsistent or misleading off-chain behavior. Collectively, these findings indicate that rollback semantics constitute an underestimated but exploitable attack surface.

Specifically, we investigate two categories of rollback exploitation: (1) *On-chain*: How can rollback semantics be exploited to synthesize profit-driven attacks? (2) *Off-chain*: How do off-chain services misinterpret rollback transactions?

In summary, we make the following contributions in this paper:

- **Rollback Attack Models.** We formalize two rollback attack models that capture how rollback semantics can be exploited in on-chain and off-chain settings, covering profit-oriented loss-avoidance attacks and rollback misinterpretation attacks against off-chain infrastructures.
- **On-Chain Attack Synthesis.** We design and implement RollGain's on-chain module, which models contract structure and value flows using graph-based representations (CCDG and RTTG) to synthesize profit-driven exploits, pruning infeasible prefixes through symbolic validation. On benchmark datasets, RollGain achieves 95.3% recall with zero false positives.
- **Off-Chain Misinterpretation Testing.** We conduct an empirical study on 3.08 billion Ethereum transactions using call tree analysis to characterize rollback patterns, and develop an off-chain testing module that exercises rollback-inducing execution vectors.
- **Real-World Impact and Mitigation.** We apply RollGain to deployed infrastructures and uncover 20 rollback misinterpretation vulnerabilities across 18 real-world services, of which

18 have been confirmed, 16 fixed, and 5 assigned CVE identifiers. We further summarize mitigation guidelines to help off-chain systems correctly handle rollback semantics.

2 Background and Threat Model

2.1 Ethereum Transactions

In Ethereum, a transaction is a cryptographically signed message from an external account that specifies fields including nonce, gas parameters, recipient, value, and calldata [45]. Transactions are the fundamental unit of computation and state transition: they may transfer native token, invoke a smart contract function, or deploy a new contract. The EVM executes each transaction atomically, committing all state changes if execution succeeds or restoring the prior state if it fails. In both cases, the transaction is recorded on-chain and the sender is charged for the gas consumed. Each transaction also produces a receipt with metadata such as gas usage and a status flag [14]. While the global state only reflects the final results of successful transactions, intermediate effects *within those transactions*, such as internal calls (subcalls) and internal value transfers, are not represented directly in the state and must be inspected via execution traces.

2.2 Smart Contract Execution

A smart contract is a program deployed on Ethereum that executes automatically when invoked by a transaction [4]. During execution, a smart contract may invoke other contracts through internal calls (also known as message calls or subcalls), each creating a new call frame with its own execution context. Internal calls are initiated via EVM call and creation opcodes (e.g., `CALL`, `DELEGATECALL`, `CREATE`). These calls form a call tree, where each frame may succeed or revert independently. Beyond native token transfers, EVM-compatible blockchains support a variety of token standards. For example, the ERC-20 standard [42] specifies fungible token interfaces (`transfer()`, `transferFrom()`) that emit `Transfer` events, which are commonly used by off-chain services to track token movements.

2.3 Transaction Rollback Mechanism

Transaction-level atomicity in the EVM is enforced by the rollback mechanism, which operates at the granularity of individual call frames. During execution, each external call executes in its own frame. If a call frame does not complete successfully, all state changes produced by that frame are discarded and the prior state is restored. A rollback may be triggered by an explicit revert (via `revert` or `require` [3]) or by an exceptional halt, such as an invalid operation or resource exhaustion. If the caller handles the error (for example, using `try/catch`), the rollback can be confined to the failing subcall without aborting the entire transaction. Accordingly, we distinguish two scenarios: **full rollback**, where the root frame reverts and the transaction fails, and **partial rollback**, where one or more inner frames revert while the root frame succeeds, resulting in mixed committed and reverted calls within the same execution trace. In both cases, reverted operations leave no persistent on-chain state changes but remain visible in execution traces [18].

2.4 Off-Chain Monitoring Services

Off-chain services play a central role in connecting on-chain activities with external systems and applications. They commonly serve two purposes: presenting blockchain execution results to users and triggering automated actions based on observed execution outcomes. One category includes blockchain explorers, token trackers, and analytics platforms, which collect execution data and present it to end users in a structured form. Another category includes Decentralized Finance (DeFi) relays, such as cross-chain bridges and centralized exchanges, which rely on execution

data to trigger token transfers or other state-dependent actions [30]. Because the committed state records only final outcomes (e.g., updated balances and storage values), these services must analyze execution traces and surviving events to reconstruct token flows and interactions. If rollback effects are not correctly distinguished from successfully committed effects, off-chain systems may misreport token flows or act on invalid data, creating inconsistencies that mislead users or even trigger unintended financial operations.

2.5 Threat Model

We assume the adversary has ordinary Ethereum user privileges, including deploying contracts, invoking functions, and submitting transactions. The attacker does not break cryptographic primitives, alter consensus rules, or control transaction ordering, but can strategically exploit rollback semantics. In the *on-chain setting*, the adversary deploys a contract that executes candidate attack sequences and uses rollback as a filter: sequences that fail to produce positive gross asset gain revert, while those that pass this condition complete and commit their state changes. In the *off-chain setting*, the adversary submits transactions with operations that roll back but remain visible in execution traces used by off-chain services, potentially causing these systems to misreport token flows or trigger invalid follow-up transfers. We assume flash loans may be used when available, but do not assume MEV ordering advantages, oracle manipulation, cross-transaction state control, or chain reorganization.

3 On-Chain Rollback Attack Synthesis

This section presents the on-chain module of RollGain. We first formalize the attack model, then design a profit-directed synthesis pipeline that extracts value-transfer paths from contract models, generates candidate call sequences guided by path reachability and state dependencies, and validates them through symbolic execution and fork replay.

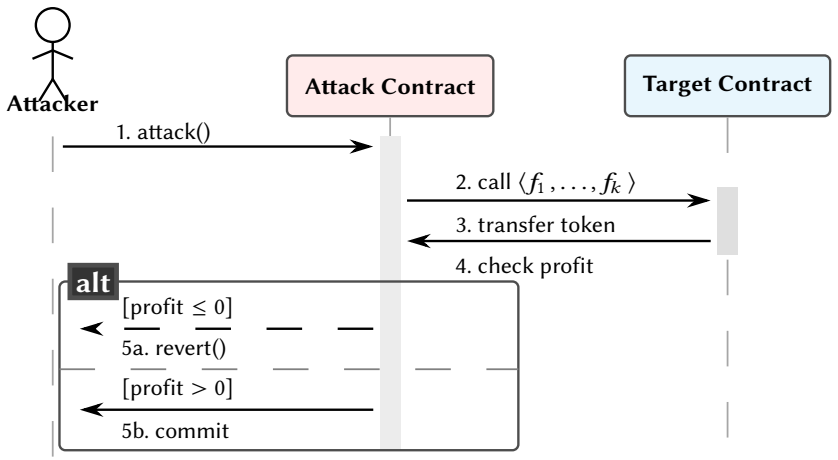


Fig. 1. On-chain rollback attack sequence.

3.1 Attack Model

Following the threat model in Section 2.5, the attacker deploys a contract that executes candidate call sequences against target contracts and uses rollback as a profit filter, as illustrated in Figure 1.

The **attack goal** is to find a sequence with positive gross asset gain: $\text{profit}_{\text{gross}} = \sum_{t \in \mathcal{T}} \delta_t \cdot p_t$, where $\mathcal{T} = \{\text{Native}\} \cup \mathcal{T}_{20}$, $\mathcal{T}_{20}$ denotes the set of ERC-20 token contracts whose balances are modified during execution, δ_t is the end-of-transaction net balance change of asset t measured at the attacker-controlled address, and p_t is the native-token-denominated price of asset t ($p_{\text{Native}} = 1$). Gas costs are outside this synthesis objective; consequently, RollGain does not claim positive realized net profit.

3.2 Overview

Figure 2 shows an overview of RollGain. RollGain takes as input the target contract source code and an optional initial state (user-provided or fetched from the chain). The pipeline consists of four stages: *contract modeling*, *attack sequence generation*, *symbolic validation*, and *fork testing*. In *contract modeling*, RollGain builds Comprehensive Contract Dependency Graph (CCDG) and Role-based Token Transfer Graph (RTTG) from the compiled intermediate representation (SlithIR) [8]. In *attack sequence generation*, RollGain identifies profit edges on the RTTG and enumerates candidate call sequences with iterative deepening; results from symbolic solving provide feedback to prune later candidates. In *symbolic validation*, each candidate undergoes bounded symbolic execution; interval analysis removes infeasible ranges before solving. In *fork testing*, accepted sequences are replayed on a local fork under the given block context, and the exploits are ranked by their measured profit.

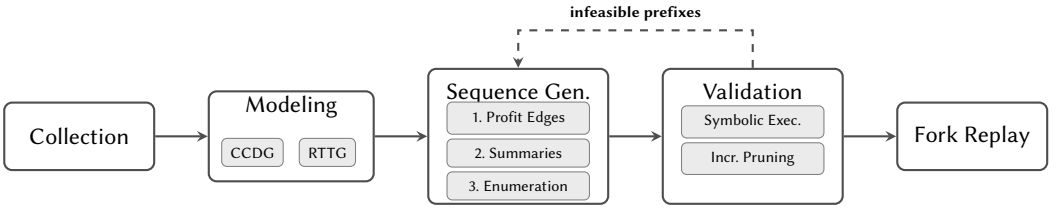


Fig. 2. Overview of the RollGain pipeline.

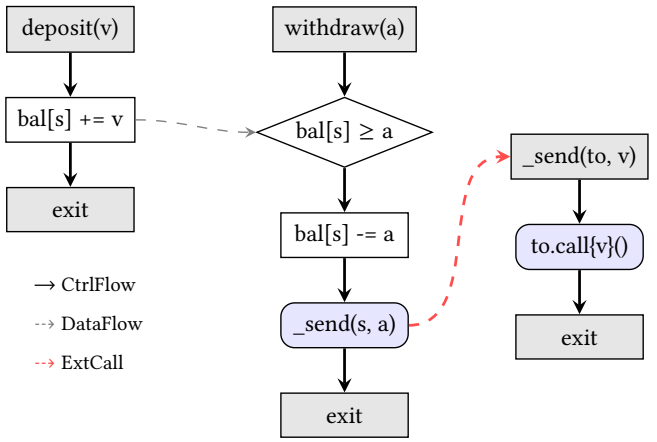
3.3 Contract Modeling

We model contract code into two complementary graph views, where the CCDG provides a unified representation of control, data, and call structure, and the RTTG abstracts token movement between roles under explicit execution paths and state conditions.

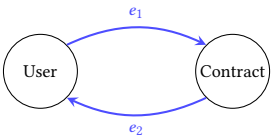
3.3.1 CCDG. A **CCDG** is a labeled directed graph that captures *control flow*, *data dependencies*, and *cross-contract invocations* in a smart contract execution. We write $G_{\text{CCDG}} = (V, E)$, where V is the set of program points and E is the set of dependency edges.

Nodes. Each node in V corresponds to exactly one *SlithIR* element. We partition $V = V_{\text{Entry}} \cup V_{\text{Exit}} \cup V_{\text{Expr}} \cup V_{\text{Call}} \cup V_{\text{Ctrl}}$, where V_{Entry} includes one entry node per *public or external function*, V_{Exit} includes one exit node per function, V_{Expr} includes *expressions and state operations*, V_{Call} includes all *call sites* (internal or external, including those that may trigger token transfers), and V_{Ctrl} includes *control-flow decision points*.

Edges. We use three labeled relations such that $E = E_{\text{CtrlFlow}} \cup E_{\text{DataFlow}} \cup E_{\text{ExtCall}}$. Edges in E_{CtrlFlow} represent *control-flow transitions* between nodes, edges in E_{DataFlow} represent *data dependence*, and edges in E_{ExtCall} model *invocation from a caller to the entry of the callee*. This unified graph supports *backward slicing* from any instruction and *path-sensitive summarization*.



(a) CCDG



	tok	amt	ψ	$\mathcal{D}$	path
e_1	ETH	v	true	∅	deposit
e_2	ETH	a	bal[s]≥a	{bal}	withdraw

(b) RTTG

Fig. 3. CCDG and RTTG for a simplified vault contract.

3.3.2 RTTG. The **RTTG** is a labeled directed multigraph that captures token movements between *abstract roles*. Each transfer edge is associated with an explicit execution path and its state conditions. We write $G_{\text{RTTG}} = (R, E_{\text{rt}})$, where R is the set of roles and E_{rt} is the set of transfer edges.

Endpoints and attributes. We define endpoint functions src, dst : $E_{\text{rt}} \to R$, where for any edge e, $\text{src}(e)$ is the sender role and $\text{dst}(e)$ is the receiver role; multiple distinct edges may share the same ordered pair $(\text{src}(e), \text{dst}(e))$. Each transfer edge $e \in E_{\text{rt}}$ is annotated with five attributes: $tok(e)$ denotes the token type (native token or an ERC-20 contract address), $amt(e)$ is the concrete or symbolic transfer amount, $\psi(e)$ is the execution condition that conjoins branch predicates (e.g., `require` checks) and storage-derived constraints along the control-flow path to the transfer site, $\mathcal{D}(e)$ is the set of state variables that influence $\psi(e)$, $amt(e)$, or the endpoint roles $\text{src}(e)/\text{dst}(e)$, and $path(e)$ is the CCDG path that triggers the transfer. Each role $r \in R$ denotes either a concrete address (for example, a user or owner) or a symbolic role defined by state predicates. We write $r_{\text{att}} \in R$ for the attacker-controlled address. Storage-dependent endpoints remain symbolic during graph construction; full-sequence validation constrains them with path and state conditions, and fork replay checks the concrete assignment. Unrepresentable or unresolved storage expressions remain unsupported. Figure 3 illustrates both representations for a simplified vault contract: the CCDG captures control flow, data dependencies, and external calls across `deposit()`, `withdraw()`, and the internal `_send()` function, while the corresponding RTTG abstracts these into role-based token transfers with their associated conditions.

RTTG Construction. We target explicit native-token and ERC-20 transfer patterns in the analyzed source code. As shown in Algorithm 1, we iterate over public and external functions in the CCDG (Line 2) and enumerate bounded intraprocedural paths with loop unrolling up to a depth limit ℓ (Line 3). For each path, we detect native-token or ERC-20 transfers (Line 4). When a path contains transfers (Line 5), we summarize them into a single RTTG edge using $\text{AggregatePath}(p, T)$ (Line 6). This function performs backward slicing at each transfer site to extract the sender and receiver roles, token type, transfer amount, execution condition ψ (the conjunction of branch predicates up to the site), and dependency set $\mathcal{D}$ (state variables used by ψ, amt, or endpoint roles). When the path contains multiple transfers of the same token between overlapping roles, they are aggregated if *stable* (no intervening write to $\mathcal{D}$ between consecutive transfers): amounts are summed, endpoint

Algorithm 1: RTTG Construction

Input: $G_{\text{CCDG}} = (V, E)$, loop bound ℓ
Output: $G_{\text{RTTG}} = (R, E_{\text{rt}})$

1 $R \leftarrow \varnothing$; $E_{\text{rt}} \leftarrow \varnothing$;
2 **foreach** function g in public/external functions **do**
3 **foreach** path $p \in \text{Paths}_{\leq \ell}(\text{Entry}(g), \text{Exit}(g))$ **do**
4 $T \leftarrow \text{Transfers}(p)$;
5 **if** $T \neq \varnothing$ **then**
6 $(r_s, r_r, tok, a, \psi, \mathcal{D}) \leftarrow \text{AggregatePath}(p, T)$;
7 $R \leftarrow R \cup \{r_s, r_r\}$;
8 $E_{\text{rt}} \leftarrow E_{\text{rt}} \cup \{(r_s, r_r, tok, a, \psi, \mathcal{D}, p)\}$;
9 **return** (R, E_{rt});

roles are refined to their intersection, and dependency sets are merged. Otherwise, AggregatePath returns the attributes of the dominant transfer (the one with the largest amount or the last on the path). The extracted roles are added to R (Line 7), and the resulting edge—annotated with the originating path p—is added to E_{rt} (Line 8). Note that $\psi(e)$ captures only prefix constraints up to the transfer site; suffix feasibility is deferred to symbolic validation (Section 3.5).

Expressiveness Compared to Prior Graphs. *CCDG* integrates control, data, and inter-procedural edges in a single typed graph aligned with Solidity semantics (e.g., `msg.sender`, `msg.value`, external calls), whereas control-flow graphs (CFGs), program dependence graphs (PDGs), and call graphs each isolate only one of these relations. This unified view enables cross-function, path-sensitive slicing and explicit update-chain reasoning. *RTTG* attaches execution conditions, dependency sets, and the originating CCDG path to each transfer edge, whereas transfer or event-flow graphs typically capture only endpoints. This distinction allows differentiating transfers with identical endpoints but different conditions or state-dependent amounts, supporting profit-directed sequence synthesis and feasibility checking.

3.4 Attack Sequence Generation

We generate candidate call sequences from the RTTG in three steps: extract profit edges, build function summaries, and run iterative-deepening enumeration. Symbolic validation (Section 3.5) verifies these candidates and feeds back infeasible prefixes to prune the search.

3.4.1 Profit Edge Extraction. We enumerate transfer edges on G_{RTTG} that, if triggered within one transaction, increase the attacker's balance. An edge e is a *profit edge* when: (1) $amt(e) > 0$; (2) $\text{dst}(e)$ can become r_{att}, either directly or through a feasible update chain; and (3) $\text{src}(e)$ cannot become r_{att} in any feasible execution, i.e., $\text{src}(e)$ is either a concrete non-attacker address or a symbolic role with no feasible update chain to r_{att}. An update chain for a state variable u is a feasible CCDG path from an attacker-callable entry point to a write site that assigns $u := r_{\text{att}}$; feasibility requires the conjunction of branch predicates along the path to be satisfiable.

3.4.2 Function Summaries. Let $\mathcal{F}$ be the set of attacker-callable functions (i.e., public or external functions). For each $f \in \mathcal{F}$, we summarize the effect of f on state using the control-flow and data-flow in G_{CCDG}.

Variables. Let $\mathcal{V}_f$ be all state variables that f may read or write, together with any state variable that appears in a branch condition or in the dataflow to a transfer amount on a path from $\text{Entry}(f)$ to $\text{Exit}(f)$. Environment inputs (i.e., `msg.sender`, `msg.value`, `block.timestamp`) are tracked separately as symbolic terms.

Algorithm 2: Iterative-deepening enumeration

Input: G_{RTTG}, G_{CCDG}; fixed summaries $\{\text{Pre}_f, \text{Post}_f\}_{f \in \mathcal{F}}$
Output: Validated sequence-edge pairs Σ

1 $\Sigma \leftarrow \varnothing$; $E \leftarrow \{e \in E_{\text{rt}} \mid \text{SAT}(\psi(e))\}$;
2 **foreach** $e \in E$ **do**
3 choose $d_{\max}$, $T_{\max}$; init S_0; $\text{h}_0 \leftarrow \text{SAT}(\psi(e), \text{S}_0)$;
4 **for** $d \leftarrow 1$ **to** $d_{\max}$ **do**
5 stack $\leftarrow \{(\langle\,\rangle, \text{S}_0, \text{h}_0)\}$; visited $\leftarrow \varnothing$;
6 **while** stack $\neq \varnothing$ **and** within $T_{\max}$ **do**
7 $(\sigma, \text{S}, \text{h}) \leftarrow \text{POP}(stack)$;
8 **if** visited $(\sigma, \text{S}, \text{h})$ **then** continue;
9 visited $\leftarrow$ visited $\cup \{(\sigma, \text{S}, \text{h})\}$;
10 **if** h **and** $|\sigma| \geq 1$ **then**
11 **if** VALIDATE$(\sigma, e) =$ SAT **then** $\Sigma \leftarrow \Sigma \cup \{(\sigma, e)\}$;
12 **if** $|\sigma| = d$ **then** continue;
13 **foreach** $f \in \mathcal{F}$ **do**
14 $\text{S}' \leftarrow \text{Compose}(\text{S}, \text{Pre}_f, \text{Post}_f)$; // memoized
15 **if** $\text{S}' \neq \bot$ **then**
16 $\text{h}' \leftarrow \text{h} \vee \text{SAT}(\psi(e), \text{S}')$;
17 PUSH$(\sigma\langle f \rangle, \text{S}', \text{h}')$
18 **return** Σ

Variable preconditions. For each $v \in \mathcal{V}_f$, define $\text{Pre}_f(v) = (I_v^{\text{pre}}, \gamma_v)$. Here I_v^{pre} is the type-aware interval of v before calling f, with bounds derived from the variable's declared type (e.g., $[0, 2^{256}-1]$ for `uint256`). The guard γ_v is the disjunction over all paths from $\text{Entry}(f)$ that read or write v, where each path contributes the conjunction of its branch predicates.
Variable postconditions. For each $v \in \mathcal{V}_f$, define $\text{Post}_f(v) = (I_v^{\text{post}}, w_v)$. Here I_v^{post} is the type-aware interval of v after f returns, and w_v is the assignment expression if v is written on any path through f; otherwise $w_v = \varnothing$. If f does not write v, then $I_v^{\text{post}} = I_v^{\text{pre}}$. We write Pre_f and Post_f to denote the collection of all per-variable summaries for f.

3.4.3 Iterative-Deepening Enumeration. As shown in Algorithm 2, given G_{RTTG} and the function summaries with G_{CCDG}, we synthesize executable call sequences that trigger a profit edge. The search uses bounded depth-first traversal, increasing the depth limit across iterations.

For profit edges where $\text{dst}(e)$ requires an update chain, we first conjoin the chain's path condition to $\psi(e)$. Pre-filtering then selects edges by the augmented $\psi(e)$: $E = \{e \in E_{\text{rt}} \mid \text{SAT}(\psi(e))\}$ (Line 1). This discards structurally unsatisfiable edges before considering concrete state; the state-aware check $\text{SAT}(\psi(e), \text{S})$ is performed during enumeration.

For each target edge $e \in E$ (Line 2), we initialize the search state (Line 3) and iterate over increasing depth bounds (Line 4). The initial store summary S_0 maps each tracked state variable to its type-default interval (e.g., $[0, 2^{256}-1]$ for `uint256`) and has no recorded writes; the initial flag $\text{h}_0 = \text{SAT}(\psi(e), \text{S}_0)$ indicates whether the profit edge is already satisfiable before any call. At each depth d, we reset the stack and visited set (Line 5) and perform depth-first search to find a call sequence $\sigma = \langle f_1, \ldots, f_k \rangle$ whose composed pre/post conditions make e executable in one transaction (Lines 6–17).
Summary-level composition. Given a prefix $\sigma = \langle f_1, \ldots, f_k \rangle$, we keep a store summary S_k that maps each tracked state variable to its current type-aware interval and last assignment expression.

Extending the prefix is incremental (Line 14): $S_{k+1} = \text{Compose}(S_k, \text{Pre}_{f_{k+1}}, \text{Post}_{f_{k+1}})$, which checks γ_v and I_v^{pre} of f_{k+1} against S_k, then applies last-write-wins and updates each variable's interval according to $\text{Post}_{f_{k+1}}$:

$$I_v^{(k+1)} := \begin{cases} I_{v,f_{k+1}}^{\text{post}} & \text{if } w_{v,f_{k+1}} \neq \varnothing, \\ I_v^{(k)} & \text{otherwise,} \end{cases}$$

where $I_{v,f_{k+1}}^{\text{post}}$ and $w_{v,f_{k+1}}$ are the postcondition interval and assignment from the summary of f_{k+1}. When f writes v, the interval is replaced; otherwise it is unchanged. We record w_v when present, and propagate environment variables (e.g., `msg.sender`, `msg.value`) as symbolic terms across function boundaries.

The Compose function returns $\bot$ if applying $\text{Pre}_{f_{k+1}}$ and $\text{Post}_{f_{k+1}}$ to S_k yields an inconsistent store (e.g., empty interval, out-of-bounds, or cross-variable conflict); otherwise it returns the updated store. Preconditions are checked incrementally at each call, so feasibility is maintained without re-evaluating the final state.

Screening and validation. For prefix $\sigma = \langle f_1, \ldots, f_k \rangle$, the historical screening flag h records whether $\psi(e)$ was summary-satisfiable at this or an earlier prefix. We compute $\mathsf{h}' \leftarrow \mathsf{h} \vee \text{SAT}(\psi(e), S')$ (Line 16) without assuming monotonic intervals: later writes may replace, narrow, or widen them. This flag only gates validation; the full ordered sequence is checked path-precisely, rejecting stale summary hits.

When h is true and the sequence contains at least one call ($|\sigma| \geq 1$, Line 10), we invoke the symbolic validator for a path-precise check (Line 11); the SMT solver returns SAT, UNSAT, or UNKNOWN, and we report the sequence only if SAT. The $|\sigma| \geq 1$ guard excludes the degenerate case where profit is trivially available at deployment without any attacker action. Pruning feedback is described in Section 3.5.

By default, we do not stop after the first hit: we continue searching until the depth bound or time budget $T_{\max}$ is reached, as other branches may yield larger objective values. When configured for fast confirmation, the search terminates upon finding the first valid sequence. Fork testing computes the final values (Section 3.6).

Search policy. For each candidate function $f \in \mathcal{F}$ (Line 13), we attempt to extend the current prefix. We prioritize calls that act on variables in $\mathcal{D}(e)$, since state preconditioning (e.g., satisfying execution conditions or triggering overflow/underflow in amount computations) may be required before the transfer edge can yield positive gain. We do not exclude other calls; they remain eligible if Pre_f is compatible with S_k. If Compose succeeds ($S' \neq \bot$), we update the satisfiability flag and push the extended state onto the stack (Lines 15–17). We memoize Compose results (Line 14) and skip visited (σ, S, h) tuples (Line 8) to avoid redundant computation; checking the full tuple ensures correctness when the same sequence is reached via different search paths.

Termination and output. At each depth d, the inner search terminates when the stack is empty or the time budget $T_{\max}$ expires (Line 6). Sequences reaching the current depth bound are not extended further (Line 12) but remain in the output if they pass validation. The outer loop (Line 4) increases d until $d_{\max}$ is reached, ensuring shorter sequences are found before longer ones. The output contains the validated sequences discovered during the search.

Validation feedback. Algorithm 2 shows a simplified view where VALIDATE is called inline. In practice, candidates are validated in batches at each depth, with pruning feedback detailed in Section 3.5.

3.5 Symbolic Validation

RTTG construction collects only prefix constraints to reduce constraint size and speed candidate generation. In this stage, we verify full-path feasibility: the entire execution from contract deployment through the call sequence must be satisfiable, including suffix conditions (e.g., later `require` checks, external-call outcomes) that were deferred during RTTG construction.

3.5.1 Harness Generation. For each candidate sequence $\sigma = \langle f_1, \ldots, f_k \rangle$, we generate two Solidity test harnesses. The *executability harness*: (1) deploys the target contract with symbolic constructor arguments, (2) executes the call sequence with symbolic function parameters and `msg.value`, and (3) asserts that at least one call in the sequence must fail. The *profit harness* shares the same setup but additionally records pre and post execution balances and asserts that the attacker's balance does not increase. The latter turns profit finding into a counterexample search.

3.5.2 Symbolic Execution. Blockchain environment variables are modeled symbolically within type-aware intervals derived from the target block context. For `block.timestamp` and `block.number`, we use intervals starting from the target block's values; when contract behavior depends on parity or specific thresholds, we generate test variants accordingly. The initial balance of the target contract is also parameterized to cover both zero and non-zero cases.

For proxies, we merge an explorer-resolved verified implementation into the analysis input; this is not a historical reconstruction of every upgrade. Because `CALL` uses callee storage while `DELEGATECALL` uses caller storage, unresolved delegate targets and storage aliasing across arbitrary call chains may be missed.

External calls are modeled abstractly without inlining arbitrary callee code. Each call is represented by a symbolic success flag: if the call fails and the failure is uncaught, the path reverts; if the caller handles failure (e.g., `try`/`catch` or low-level call return checks), both success and failure branches are explored. Reentrancy is modeled only when a reentrant pattern is identified in the contract and the corresponding re-entry is explicitly represented in the attack sequence σ as an attacker-callable function invocation.

To solve hash-related constraints, we maintain a lookup table of known input-output pairs for `keccak256`-like elements. For constraints outside this table, if the SMT query fails due to the unsolved hash term while all other constraints are satisfiable, we mark the path as high-likelihood and return the hash constraint for manual investigation or further resolution.

We leverage bounded symbolic execution over EVM bytecode to explore execution paths. The engine maintains symbolic state for storage slots, memory, and stack, and forks execution at each branch point. Arithmetic is modeled with fixed-width bit-vectors under Solidity's checked semantics; loops are unrolled up to a configurable bound. When a path reaches the assertion, the accumulated path constraints are submitted to an SMT solver. For the executability harness, a sequence is deemed executable if any path yields SAT (i.e., all calls complete without revert). For the profit harness, a sequence satisfies the profit guard if any path yields SAT (i.e., the balance-increase assertion is violated), and the resulting model provides concrete values for constructor arguments, function parameters, `msg.value`, and environment variables.

3.5.3 Incremental Pruning. As noted in Section 3.4, validation results feed back to prune the candidate space. Iterative deepening naturally produces candidates in increasing length order; we validate candidates at each depth in batches before proceeding to the next, running both harnesses described above for each candidate. Only the executability test result drives pruning: if this test returns UNSAT (i.e., no execution path reaches completion), we record the prefix and skip all longer sequences sharing the same prefix in subsequent batches, as they inherit the infeasibility. A sequence that is executable but fails the profit test (profit test UNSAT, executability test SAT) does

not trigger pruning, as an extension may still reach a qualifying edge. An UNKNOWN result does not trigger pruning. This feedback loop reduces both validation and solving overhead.

3.6 Fork Testing and Profit Ranking

Symbolic validation verifies path feasibility and produces concrete parameters under an abstract execution model. Fork testing complements this by replaying validated sequences on a local fork that reproduces the exact chain state at the target block height, instantiating the attack model and profit formula defined in Section 3.1.

3.6.1 Attack Contract Generation and Profit Computation. For each validated sequence with concrete parameters, we generate a deployable attack contract whose structure mirrors the validation harness and implements *profit-guarded rollback*: it reverts if $\text{profit}_{\text{gross}} \leq 0$, preventing persistent state changes when the guard fails. The objective value is computed by valuing native token directly and querying on-chain price oracles at the forked block height for ERC-20 tokens to obtain p_t; tokens without a price source are excluded.

3.6.2 Execution and Ranking. We replay each attack contract on the fork and mark a sequence *successful* if its transaction commits. Successful sequences are ranked by descending $\text{profit}_{\text{gross}}$; ties are broken by preferring sequences with fewer calls. The final report includes the generated attack contract (PoC), execution traces, and the computed objective value.

4 Off-Chain Rollback Misinterpretation

Rollback semantics not only affect on-chain exploitability but can also create inconsistencies in off-chain infrastructures. This section formalizes such misinterpretation and presents RollGain's testing framework for detecting it.

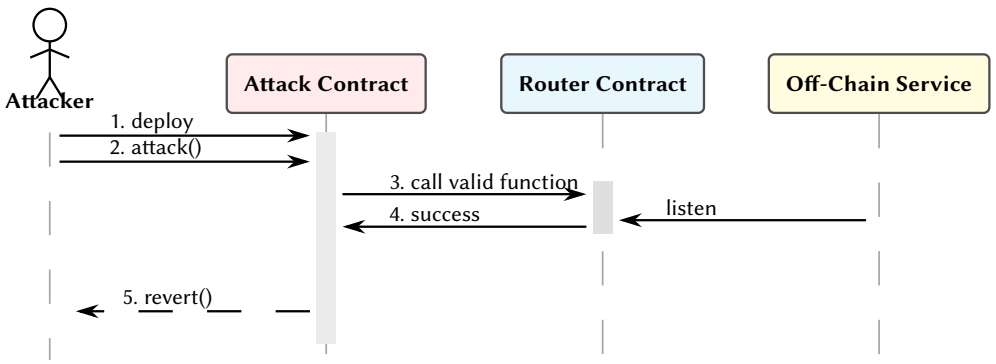


Fig. 4. Off-chain rollback attack sequence.

4.1 Attack Model

The **attack goal** is to cause off-chain services to misinterpret rollback transactions, creating inconsistencies between on-chain finality and off-chain state. As shown in Figure 4, the attacker deploys a contract and invokes a router function while an off-chain service is listening. The transaction then rolls back, so that no state changes persist on-chain, while off-chain observers may be misled. Based on the rollback definitions in Section 2.3, we distinguish two attack vectors:
Full rollback. The adversary issues a transaction that performs state-changing operations and triggers a full rollback through an explicit revert or an exceptional halt (e.g., out-of-gas). As a result,

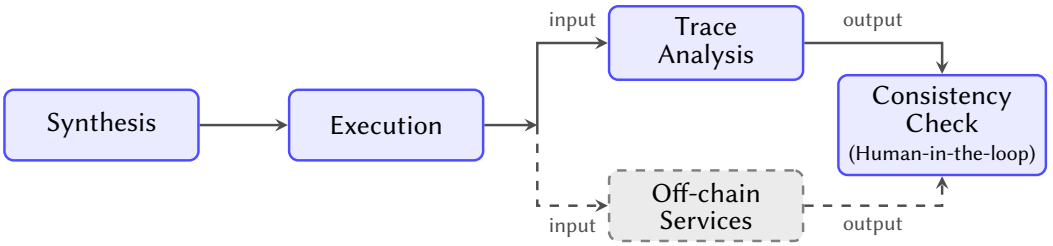


Fig. 5. Overview of the off-chain testing module.

no state changes persist on-chain. However, off-chain systems may mishandle this scenario by (1) recording pending operations as successful without reconciling them with the final transaction status, or (2) caching intermediate states observed during execution that are later discarded.

Partial rollback. In a single transaction, the root call succeeds while some internal calls roll back. The vulnerability arises when off-chain infrastructures process execution traces as flat, linear sequences and fail to preserve call nesting and rollback semantics, causing rolled-back internal calls to be misclassified as successful.

4.2 Testing Module Design

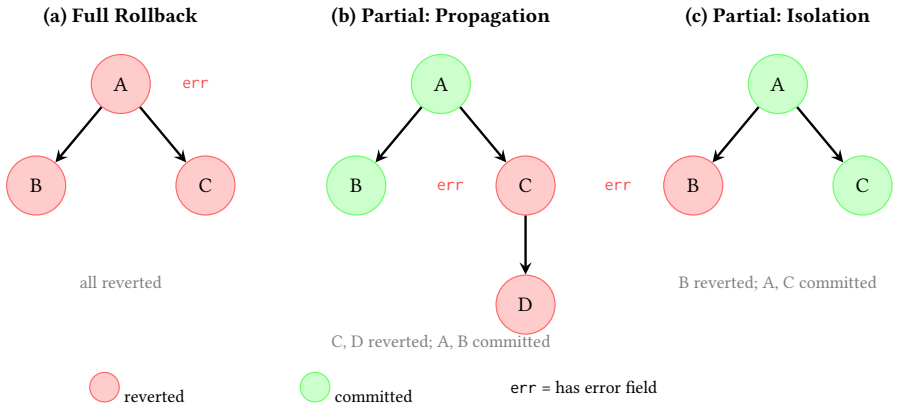


Fig. 6. Call tree scenarios: (a) full rollback, (b)(c) partial rollback demonstrating propagation and isolation.

Figure 5 shows an overview of the testing module. We first generate harness contracts that trigger rollback behaviors with optional value transfers in *Synthesis*. In *execution*, we deploy the harnesses on a local fork or test network and invoke test variants that cover full and partial rollback executions, both with and without value transfers. The resulting transactions are fed into two parallel paths. The first path is *trace analysis*, we retrieve and parse nested execution traces from the EVM and apply revert-state propagation to determine which operations are ultimately committed. In parallel, *off-chain services* process the same transactions and produce their own interpretations. Finally, a *consistency check* compares the off-chain outputs against the trace analysis results and reports discrepancies to assist manual confirmation of potential vulnerabilities.

4.2.1 Call Tree Analysis. To inform the design of targeted rollback-testing harnesses, we first conduct an empirical study to understand how rollback behaviors manifest in real execution

Table 1. Harness contract variants for rollback testing.

Category	Variant	Trigger	Structure	Expected Behavior
Full Rollback	revert	`revert()`	Any	All state changes reverted
	require	`require(false)`	Any	All state changes reverted
	assert	`assert(false)`	Any	Panic error, all reverted
	invalid	`invalid` opcode	Any	Gas consumed, all reverted
	out-of-gas	Infinite loop	Any	Gas exhausted, all reverted
Partial Rollback	try-catch	`try-catch` block	Single child	Inner failure caught
	low-level	`call` returns false	Single child	Inner call has error
	nested	Nested call chain	Depth 3+	Subtree reverted
	mixed	Multiple targets	2+ children	Some succeed, some fail

traces. We analyze Ethereum mainnet transactions from block 1 to block 23,749,999, covering 3,077,026,269 transactions. For each transaction, we retrieve the execution trace and convert it into a call tree structure, where each node represents an internal call annotated with its error status, value transferred, and child calls. From an execution-structure perspective, this analysis reveals two canonical rollback patterns, as illustrated in Figure 6.

Full rollback (Figure 6a) occurs when the root call fails, causing all descendant calls to be rolled back regardless of their individual success status. **Partial rollback** (Figure 6b,c) arises when the root call succeeds but some internal calls fail, and can be divided into two patterns. *Propagation* (Figure 6b) occurs when a failed call causes its parent to revert, allowing the error to bubble up through the call chain until it is caught by a handler or reaches the root. In contrast, *Isolation* (Figure 6c) occurs when the parent catches the failure via `try`/`catch` or low-level `call`, confining rollback to the affected subtree. Propagation introduces greater implementation challenges, as services must trace upward through potentially deep call chains to locate the catch boundary, whereas Isolation provides an explicit and immediate boundary at the parent level.

To establish ground truth for subsequent evaluation, we recursively propagate revert status from parent to child. A call is marked as reverted if it either reports an error or has any reverted ancestor. This partitions all calls into committed and reverted sets, which serve as the reference for comparison against off-chain outputs.

4.2.2 Harness Contract Design. Based on the call tree analysis, we design harness contracts that systematically cover rollback structures and error-propagation behaviors observed in real execution traces, as summarized in Table 1. The design aims to capture canonical rollback patterns that may lead to off-chain misinterpretation, rather than exhaustively enumerating all possible executions.

For **full rollback**, we include five trigger variants corresponding to distinct EVM-level failure modes, including explicit reverts and exceptional halts. Although these triggers differ in how the rollback is initiated, they all cause the root call frame to revert, resulting in the entire transaction being rolled back. These variants allow us to test whether off-chain services correctly discard intermediate effects when the transaction ultimately fails.

For **partial rollback**, we vary both the error-handling mechanism and call structure to reflect the propagation and isolation behaviors identified in the call tree analysis. Specifically, we consider error handling via `try`/`catch` and low-level `call`, together with different call structures, including single-child calls, nested call chains, and multiple sibling calls. These variants exercise scenarios in which some internal calls revert while others remain committed within the same execution trace.

Table 2. Quantitative comparison (left) and ablation study results (right). In the ablation table, *Detected* is identified vulnerable contracts, *Val. Calls* denotes generated sequences, and *Time* is total analysis time.

(a) Quantitative comparison

Tool	TP	FP	Prec.	Recall	F1
RollGain	**41**	**0**	**100.0%**	**95.3%**	**0.98**
Slither	32	11	74.4%	74.4%	0.74
Mythril	29	8	78.4%	67.4%	0.72
Oyente	26	12	68.4%	60.5%	0.64
Securify2	4	0	100.0%	9.3%	0.17
ConFuzzius	2	0	100.0%	4.7%	0.09

(b) Ablation study

Configuration	Detected	Val. Calls	Time
RollGain (Full)	**38/43**	**180**	54 min
w/o RTTG	32/43	561	92 min
Random Baseline	30/43	1026	113 min

Across both rollback categories, we instantiate harnesses at increasing call depths to assess whether off-chain services preserve call nesting, propagate revert semantics, and avoid misinterpreting deeper events or value movements. Each variant may also include native token transfers at selected call sites to test whether services distinguish committed transfers from those in reverted frames.

5 Evaluation

We structure our evaluation around five research questions:

- **RQ1:** How does RollGain compare with existing tools?
- **RQ2:** Can RollGain synthesize executable attack sequences?
- **RQ3:** What is the contribution of each component in RollGain?
- **RQ4:** What are the dominant rollback patterns on Ethereum mainnet?
- **RQ5:** Do real-world off-chain services correctly handle rollback transactions?

Implementation. RollGain uses Foundry [26] for Solidity compilation and fork testing, SlithIR [8] for CCDG/RTTG construction, and Halmos [27] for symbolic execution. The other components are implemented in about 16.5k lines of Python, with 1.2k lines of Solidity for off-chain testing.
Experimental Setup. Our experiments were conducted on Apple M4 Max (16 cores), 128 GB RAM, running macOS 15. The search depth is 10, the maximum sequences is 200, the Halmos timeout is 600 s, and the process timeout is 3600 s. Attacker addresses are funded with 1 ETH for testing.
Datasets. We use two datasets for on-chain evaluation. For recall evaluation, we adopt dataset from Wei et al. [43], which contains 472 contracts covering vulnerabilities including reentrancy, arithmetic errors, unchecked send, unsafe delegatecall, transaction ordering dependence, time manipulation, bad randomness, authorization via `tx.origin`, unsafe suicidal, and gasless send. The contracts are collected from SWC Registry [34], SmartBugs [7], Not So Smart Contracts [40], prior empirical studies [9, 31], and blogs, with duplicates removed. Two auditors independently labeled every contract for profit exploitability under our threat model, excluding phishing, denial-of-service, and isolated single-function defects without a profitable sequence. Agreement before adjudication was high (Cohen's $\kappa = 0.97$); disagreements were cross-validated and resolved through discussion, yielding **43** contracts. For false positive evaluation, we use **14** honeypot contracts [39].

5.1 RQ1: Quantitative Comparison

We compare RollGain with eight tools: static analyzers (Slither [8], Securify2 [41]), symbolic executors (Mythril [32], Oyente [21], Manticore [22], Osiris [38]), fuzzers (ConFuzzius [37]), and linters (Solhint [28]). We measure *Precision* as TP / (TP + FP), *Recall* as TP / 43 on the vulnerable contract dataset, and *F1* as the harmonic mean.

Results. Table 2 reports the quantitative comparison. For baseline tools, a true positive denotes a *profitable vulnerability*, whereas for RollGain it requires a *concrete and executable attack sequence* satisfying the on-chain objective. RollGain achieves **0** false positives and **95.3%** recall (F1=0.98) by validating candidates through attack synthesis, filtering honeypot contracts that exhibit suspicious patterns but admit no such sequence. In contrast, tools with higher recall such as Slither, Mythril, and Oyente produce 8–12 false positives on honeypots, as they report vulnerabilities without checking economic realizability, while tools with zero false positives (Securify2 and ConFuzzius) suffer from very low recall (4.7–9.3%) due to overly restrictive detection rules. Manticore, Osiris, and Solhint reported no positives on our dataset (TP=0, FP=0). Overall, synthesis-based validation improves precision without sacrificing recall relative to existing approaches. The two false negatives are due to unsupported features discussed in Section 5.6.

5.2 RQ2: Synthesis Capability

We evaluate RollGain's ability to synthesize executable attack sequences. Existing tools in RQ1 focus on vulnerability detection and do not support attack contract generation.

Results. RollGain produces fork-replayable PoCs for **38** of the 41 contracts detected in RQ1 (**92.68%**). The three remaining PoC-generation failures are due to limitations of the symbolic execution backend, including unsupported syntax patterns (`selfdestruct`) or solver timeout during harness validation. Among the 38 successful cases, **28** require no environment-specific setup (e.g., `block.timestamp`) and are directly reproducible. The remaining **10** depend on transaction ordering or caller-specific effects, where exploitability varies with the relative order of transactions or participating accounts; our rollback-based validation prevents persistent state losses on unsuccessful attempts and enables systematic exploration of such behaviors.

5.3 RQ3: Ablation Study

We evaluate component contributions using three configurations: **Full RollGain**, with CCDG-guided search and RTTG-based profit edge prioritization; **w/o RTTG**, which uses CCDG for reachability but randomizes the search order among profit edges; and a **Random Baseline**, which generates sequences without CCDG or RTTG guidance. For each configuration, we apply a 10-minute timeout per contract and stop validation as soon as the first executable PoC satisfying the objective is synthesized.

Results. Table 2 summarizes the results. The full RollGain achieves 88.4% recall (38/43) with 180 validation calls in 54 min. Removing RTTG guidance reduces recall to 74.4% (32/43, −14%) and increases validation calls by 3.1× (561 calls), as the search explores more paths before finding a qualifying PoC. The random baseline achieves only 69.8% recall (30/43) with 5.7× more validation calls (1026 calls), demonstrating that both CCDG-based reachability analysis and RTTG-based ordering are essential for effective attack synthesis. Notably, without RTTG guidance, **6** contracts (about **16%** of those detected by the full system) time out.

5.4 RQ4: Prevalence and Patterns

Building on the call tree analysis methodology described in Section 4.2, we quantify rollback prevalence across the full Ethereum mainnet history (block 1 to 23,749,999, totaling 3.08 billion transactions) using data from XBlock-ETH [49] and Dune Analytics [6]. We classify transactions by receipt status: *Full Rollback* (receipt status = 0), *Partial Rollback* (receipt status = 1 with failed internal calls), or *Unknown* (pre-Byzantium transactions lacking receipt status per EIP-658 [14]). Partial rollback is further divided into *Propagation* (parent also fails) and *Isolation* (parent succeeds via try/catch).

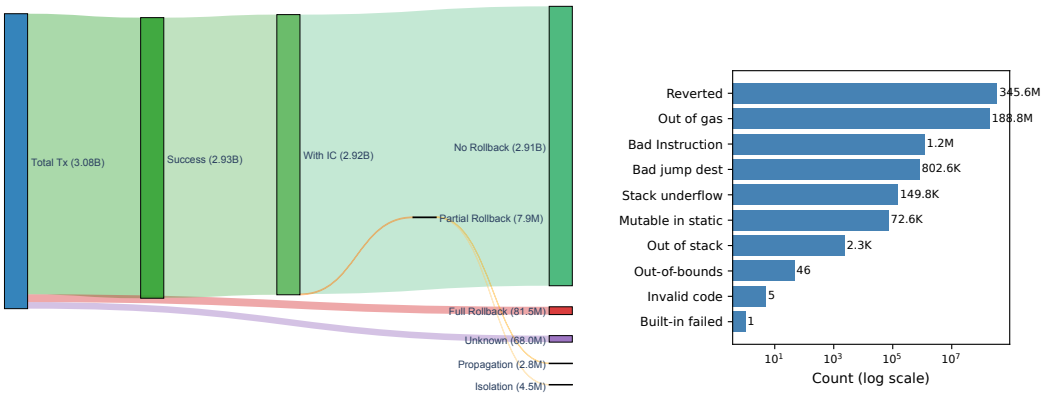


Fig. 7. Left: transaction rollback distribution. Right: internal call failure reasons.

Figure 7 shows the distribution, where absolute counts are rounded for readability and percentages are computed from exact values. Of 3.08B transactions, **95.14% succeed** (2.93B), **2.65% fully rollback** (81.5M), and 2.21% are unknown (68.0M). Among 2.92B successful transactions with internal calls, **7.9M (0.27%) show partial rollback**, split into **Isolation** (4.5M, 56.9%) and **Propagation** (2.8M, 36.0%), with the remainder root-level failures. For internal call failures (536.6M; Figure 7 right), **explicit `revert` accounts for 64.4%** (345.6M) and **out-of-gas for 35.2%** (188.8M).

Finding 1: Partial rollback requires call-tree traversal to detect. Receipt status identifies full rollback, whereas **partial rollback is invisible without reconstructing call hierarchy**. Off-chain services that process traces linearly may therefore misclassify it.

Finding 2: Propagation is harder to handle than Isolation. Propagation poses greater implementation challenges: services must recursively trace revert origins and mark entire subtrees as rolled back. Isolation has clear try/catch boundaries, while Propagation requires unbounded upward traversal.

Finding 3: Revert and gas exhaustion dominate failure causes. Together, **explicit `revert` and out-of-gas failures** account for 99.6% of internal call failures; other causes (invalid opcode, stack errors) are negligible (<0.4%). These two mechanisms represent the primary ways contracts trigger rollback, whether intentionally (validation, access control) or through resource exhaustion.

5.5 RQ5: Off-Chain Testing

Building on the harness design in Section 4.2.2 and guided by the prevalence findings in Section 5.4, we apply RollGain's off-chain testing module to real-world services.

Dataset. We selected off-chain services from Alchemy's curated developer tools list [1], supplemented with search engine queries and open-source projects from GitHub, covering token trackers, RPC providers, and blockchain explorers.

Methodology. We instantiate the harness variants described in Table 1, covering full rollback triggers (e.g., `revert`, `invalid`, gas exhaustion) and partial rollback patterns (e.g., `try/catch`, low-level `call`, nested structures). Each variant optionally includes native token or ERC-20 transfers. Following the testing pipeline in Section 4.2, we deploy harnesses and execute test transactions on a network accessible to the target service (local fork or public testnet, depending on service requirements). Ground truth is derived via revert-state propagation, recursively marking a call as rolled back if it reports an error or has any rolled-back ancestor. In parallel, the target service processes the same transactions via its API. A vulnerability is confirmed when the output of the

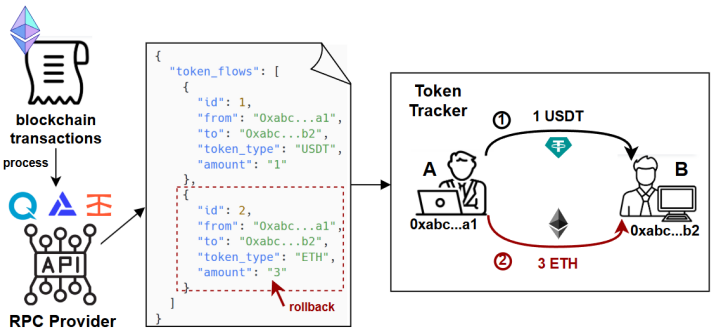


Fig. 8. Token-flow misinterpretation in partial rollback.

target service diverges from our ground truth, such as reporting rolled-back calls as successful or displaying rolled-back transfers as confirmed.

Results. We discovered 20 rollback misinterpretation vulnerabilities across 18 off-chain services. Specific failure modes include: (1) *Token trackers* (9 services, 9 issues) displayed native token transfers from partial-rollback transactions as confirmed movements, affecting unexpected top-up or user portfolio displays; (2) *Explorers* (7 services, 8 issues) misclassified rolled-back calls as successful; one explorer even labeled a full-rollback transaction (receipt status = 0) as successful; (3) *RPC providers* (2 services, 3 issues) returned internal call results without propagating ancestor failure status, causing downstream applications to act on invalid data. We responsibly disclosed all vulnerabilities; 18 have been confirmed, 16 fixed, and 5 assigned CVE identifiers. Due to ongoing remediation or commercial sensitivity, we do not disclose specific affected commercial services.

Finding: Missing error fields in rolled-back calls. A key insight from our testing is that child calls may lack an `error` field even when their effects are rolled back. Under propagation, a child may complete successfully yet have its state discarded when an ancestor later fails. We observed that some RPC providers and explorers annotate the error field only at the ancestor node, leaving descendants unmarked. **Although we do not classify this behavior as a vulnerability or report it**, these services are widely used by downstream applications, and the missing annotations pose a latent risk: **downstream consumers that check only the `error` field of individual calls will misinterpret rolled-back operations as successful**.

Case Study. Figure 8 illustrates a representative vulnerability in a token tracker relying on the token-flow API of an RPC provider. The affected platform, used for fund tracking with over 10,000 users, rendered committed and rolled-back transfers identically, causing nonexistent ETH movements from partial-rollback transactions to propagate to user dashboards and compliance monitoring systems. After our disclosure, the RPC provider patched the API to filter rolled-back transfers.

5.6 Threats to Validity

Internal. For on-chain analysis, bounded path enumeration may miss paths beyond the configured depth, and dynamic dispatch via proxies can obscure transfer sites; we mitigate this using configurable depth limits and fork-based validation. We analyze explorer-resolved verified proxy implementations, but unavailable or dynamic delegate targets and arbitrary call-chain storage aliasing may cause false negatives, and fork replay cannot recover modeling omissions. Additionally, Halmos does not support `selfdestruct`, and RollGain can not handle dynamic arrays, which

may cause false negatives on contracts relying on these features; we plan to address these in future work. For prevalence analysis, our classification relies on Ethereum archive node traces; we exclude transactions with malformed or missing trace data and cross-validate with receipt status. For off-chain testing, test vectors cover the dominant failure patterns identified in our prevalence study, accounting for 99.6% of real-world failures.

External. On-chain datasets are representative but not exhaustive; we reuse datasets from prior work for comparability. Prevalence data covers Ethereum mainnet up to a fixed block height, with scripts provided for replication on newer data. Off-chain services were tested at fixed versions; all vulnerabilities were responsibly disclosed and fixes verified.

6 Mitigation

This section provides mitigation for contract and off-chain services based on our findings.

On-Chain Mitigation. Because rollback-based attacks exploit the combination of conditional revert logic and exploitable value flows, smart contract developers can reduce exposure by following the checks-effects-interactions pattern [24] to limit information available to adversarial callees, applying reentrancy guards to constrain call interleaving within a transaction, and using commit-reveal schemes for contracts involving randomness or competitive outcomes to prevent rollback-based outcome selection. For contracts where users can influence state and withdraw value, time-locked withdrawals can limit the ability to atomically test and profit within a single transaction.

Off-Chain Mitigation. The key to preventing rollback misinterpretation is to process intermediate data with awareness of transaction finality. When processing internal calls from execution traces, off-chain services should first check the top-level transaction receipt status, then recursively propagate revert states from parent calls to all descendants in the call tree, since a child call may complete without error but still have its effects discarded when an ancestor reverts. To maintain compatibility, systems can introduce status annotations in APIs and user interfaces while keeping existing fields unchanged, clearly marking reverted transfers to avoid presenting them as confirmed operations. Bridges and financial transfer services should refrain from acting on pending or intermediate execution results and instead confirm outcomes against receipts before taking action, thereby preventing premature releases or misallocation of funds.

7 Related Work

Smart Contract Vulnerability Detection. Smart contract vulnerability detection has been extensively studied using static, dynamic, and hybrid analyses. Early analyzers such as Oyente [21], ZEUS [15], and Slither [8] focus on common vulnerabilities, including reentrancy, arithmetic errors, and unsafe calls. Pattern- and logic-based systems such as Securify encode compliance and violation properties over dependency graphs [41]. Dynamic and hybrid approaches broaden coverage through symbolic execution frameworks (e.g., Mythril, Manticore [22, 23]), fuzzing techniques (e.g., ContractFuzzer, Echidna [10, 13]), and hybrid fuzzers such as ConFuzzius [37]. Several studies also examine revert-related behaviors: Liu et al. characterize developers' use of transaction-reverting statements [18], SmartState detects state-reverting vulnerabilities via fine-grained state-dependency analysis [17], and other work studies rollback-based randomness attacks on Ethereum and EOSIO [12, 29]. Despite these advances, existing tools largely treat rollback as an execution endpoint rather than a controllable attack primitive, and do not model how rollback semantics propagate into off-chain infrastructures where reverted operations may be misinterpreted.

Smart Contract Attack Synthesis. Beyond detection, prior work explores automated exploit generation. Symbolic execution has been applied to search multi-transaction vulnerability sequences [35], while template-based approaches generate attacker contracts from ABI information but struggle with complex control flow or cross-contract behavior [2]. Snapshot-based fuzzing

improves scalability by reusing intermediate states; for example, ItyFuzz synthesizes concrete exploits efficiently without executing long transaction sequences [33]. Recent efforts target economic attacks: FlashSyn synthesizes flash-loan exploits using counterexample-guided approximation [5], other techniques generate profitable swap sequences against automated market makers [11], and profit-oriented fuzzing systems such as VERITE integrate exploitability and profit maximization into the testing loop [16]. While these works demonstrate the effectiveness of automated attack synthesis, none integrates rollback as a filter for profit-oriented synthesis. DeFiRanger reconstructs high-level financial actions from already executed Ethereum transactions and detects price-manipulation attacks, rather than generating attacker contracts [46]. The closest synthesis framework, Foray, lifts a DeFi protocol into a token-flow graph over a predefined Abstract Financial Language and completes reachable attack sketches with domain-specific constraints; its synthesis requires attack-goal and initial-state specifications and may require mappings from abstract actions to concrete functions [44]. In contrast, RollGain automatically constructs CCDG/RTTG from the target's compiled IR, retains path conditions and state dependencies, and performs rollback-aware search with infeasible-prefix feedback and profit-guarded symbolic and fork validation.

Off-Chain and Cross-Chain Security. Off-chain infrastructures, including explorers, token trackers, RPC providers, and cross-chain bridges, play a critical role in interpreting and reacting to blockchain activity. Prior studies analyze their security implications in DeFi and CeFi ecosystems and investigate attacks on cross-chain bridges [19, 30, 47, 48]. For example, XScope detects bridge attacks by analyzing inter-chain event flows [47], and recent work demonstrates how rollback semantics can be exploited in cross-layer settings, such as double-spending attacks on Arbitrum that affect L1 bridge state [36]. Transaction risk analysis has also been explored to assess on-chain activity [20]. However, these studies primarily focus on committed transactions and do not address how off-chain services may misinterpret aborted executions as successful when processing traces containing reverted operations. Our work fills this gap by systematically evaluating how explorers, token trackers, and RPC providers handle rollback transactions, uncovering misinterpretation vulnerabilities that lead to inconsistent records or unintended actions.

8 Conclusion

This paper formalizes on-chain and off-chain rollback attack models and presents RollGain, a unified framework for profit-driven attack synthesis and off-chain misinterpretation testing. RollGain achieves 95.3% recall with zero false positives by reverting unprofitable sequences, and uncovers 20 vulnerabilities across 18 real-world off-chain services (including explorers, token trackers, and RPC providers), with 18 confirmed, 16 fixed, and 5 assigned CVEs. Our results show that rollback semantics form an underestimated attack surface, and we provide practical mitigation guidance for both contract and off-chain service development.

Data Availability

Our replication package is available online at https://github.com/yxsec/rollback.

Acknowledgments

This research is supported by the Singapore Ministry of Education Academic Research Fund Tier 2 (T2EP20224-0003), Academic Research Fund Tier 1 (RG12/23), and the Nanyang Technological University Centre for Computational Technologies in Finance (NTU-CCTF). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of MOE and NTU-CCTF.

References

[1] Alchemy Insights, Inc. 2026. Web3 Developer Tools — dApps on Alchemy. https://www.alchemy.com/dapps/top/developer-tools. Accessed: 2026-01-25.

[2] Ignacio Ballesteros, Clara Benac-Earle, Luis Eduardo Bueso de Barrio, Lars-Åke Fredlund, Ángel Herranz, and Julio Mariño. 2022. Automatic generation of attacker contracts in solidity. In *4th International Workshop on Formal Methods for Blockchains (FMBC 2022)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 3–1. doi:10.4230/OASIcs.FMBC.2022.3

[3] Alex Beregszaszi and Nikolai Mushegian. 2017. *EIP-140: REVERT instruction.* Technical Report 140. Ethereum Improvement Proposals. Accessed: 2026-01-25. https://eips.ethereum.org/EIPS/eip-140

[4] Vitalik Buterin. 2014. Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform. https://ethereum.org/whitepaper/. White paper.

[5] Zhiyang Chen, Sidi Mohamed Beillahi, and Fan Long. 2024. Flashsyn: Flash loan attack synthesis via counter example driven approximation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. 1–13. doi:10.1145/3597503.3639190

[6] Dune. 2026. *Dune.* Accessed: 2026-01-25. https://dune.com/

[7] Thomas Durieux, João F Ferreira, Rui Abreu, and Pedro Cruz. 2020. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International conference on software engineering (ICSE)*. 530–541. doi:10.1145/3377811.3380364

[8] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: A Static Analysis Framework for Smart Contracts. In *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 8–15. doi:10.1109/WETSEB.2019.00008

[9] Asem Ghaleb and Karthik Pattabiraman. 2020. How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 415–427. doi:10.1145/3395363.3397385

[10] 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 (ISSTA)*. 557–560. doi:10.1145/3395363.3404366

[11] Sujin Han, Jinseo Kim, Sung-Ju Lee, and Insu Yun. 2025. Automated Attack Synthesis for Constant Product Market Makers. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025). doi:10.1145/3728872

[12] Ningyu He, Ruiyi Zhang, Haoyu Wang, Lei Wu, Xiapu Luo, Yao Guo, Ting Yu, and Xuxian Jiang. 2021. {EOSAFE}: security analysis of {EOSIO} smart contracts. In *30th USENIX security symposium (USENIX Security 21)*. 1271–1288.

[13] Bo Jiang, Ye Liu, and Wing Kwong Chan. 2018. ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*. 259–269. doi:10.1145/3238147.3238177

[14] Nick Johnson. 2017. *EIP-658: Transaction status code.* Technical Report. Ethereum Improvement Proposals. Accessed: 2026-01-25. https://eips.ethereum.org/EIPS/eip-658

[15] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. In *Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 1–12. doi:10.14722/ndss.2018.23082

[16] Ziqiao Kong, Cen Zhang, Maoyi Xie, Ming Hu, Yue Xue, Ye Liu, Haijun Wang, and Yang Liu. 2025. Smart Contract Fuzzing Towards Profitable Vulnerabilities. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 153–175. doi:10.1145/3715720

[17] Zeqin Liao, Sicheng Hao, Yuhong Nan, and Zibin Zheng. 2023. Smartstate: Detecting state-reverting vulnerabilities in smart contracts via fine-grained state-dependency analysis. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 980–991. doi:10.1145/3597926.3598111

[18] Lu Liu, Lili Wei, Wuqi Zhang, Ming Wen, Yepang Liu, and Shing-Chi Cheung. 2021. Characterizing Transaction-Reverting Statements in Ethereum Smart Contracts. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 630–641. doi:10.1109/ASE51524.2021.9678597

[19] Yixuan Liu, Yuxin Dong, Ye Liu, Xiapu Luo, and Yi Li. 2025. Phantom Events: Demystifying the Issues of Log Forgery in Blockchain. *arXiv preprint arXiv:2502.13513* (2025).

[20] Yixuan Liu, Xinlei Li, and Yi Li. 2025. DeepTx: Real-Time Transaction Risk Analysis via Multi-Modal Features and LLM Reasoning. In *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. doi:10.1109/ASE63991.2025.00366

[21] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security (CCS)*. 254–269. doi:10.1145/2976749.2978309

[22] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1186–1189.

doi:10.1109/ASE.2019.00133
[23] Bernhard Mueller. 2018. Smashing ethereum smart contracts for fun and real profit. *HITB SECCONF Amsterdam* 9, 54 (2018), 4–17.
[24] Dominik Muhs. 2026. Smart Contract Security Field Guide. https://scsfg.io/. Accessed: 2026-01-26.
[25] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system.
[26] Paradigm. 2026. Foundry: A blazing fast, portable and modular toolkit for Ethereum application development. https://getfoundry.sh/. Accessed: 2026-01-25.
[27] Daejun Park, Yi Seo, and Seonghyun Kim. 2025. Halmos: A symbolic testing tool for EVM smart contracts. https://github.com/a16z/halmos. Accessed: 2026-01-25.
[28] Protofire. 2025. Solhint: An open source project for linting Solidity code. https://github.com/protofire/solhint. Accessed: 2026-01-25.
[29] Peng Qian, Jianting He, Lingling Lu, Siwei Wu, Zhipeng Lu, Lei Wu, Yajin Zhou, and Qinming He. 2023. Demystifying random number in ethereum smart contract: Taxonomy, vulnerability identification, and attack detection. *IEEE Transactions on Software Engineering* 49, 7 (2023), 3793–3810. doi:10.1109/TSE.2023.3271417
[30] Kaihua Qin, Liyi Zhou, Yaroslav Afonin, Ludovico Lazzaretti, and Arthur Gervais. 2021. CeFi vs. DeFi–Comparing Centralized to Decentralized Finance. *arXiv preprint arXiv:2106.08157* (2021).
[31] Meng Ren, Zijing Yin, Fuchen Ma, Zhenyang Xu, Yu Jiang, Chengnian Sun, Huizhong Li, and Yan Cai. 2021. Empirical evaluation of smart contract testing: What is the best choice?. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 566–579. doi:10.1145/3460319.3464837
[32] Nipun Sharma and Swati Sharma. 2022. A survey of Mythril, a smart contract security analysis tool for EVM bytecode. *Indian Journal of Natural Sciences* 13, 75 (2022), 51003–51010.
[33] Chaofan Shou, Shangyin Tan, and Koushik Sen. 2023. ItyFuzz: Snapshot-Based Fuzzer for Smart Contracts. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 322–333. Accessed: 2026-01-18. doi:10.1145/3597926.3598059
[34] SmartContractSecurity. 2020. Smart Contract Weakness Classification Registry (SWC). https://swcregistry.io/. Accessed: 2026-01-25.
[35] Sunbeom So, Seongjoon Hong, and Hakjoo Oh. 2021. {SmarTest}: Effectively hunting vulnerable transaction sequences in smart contracts through language {Model-Guided} symbolic execution. In *30th USENIX Security Symposium (USENIX Security 21)*. 1361–1378.
[36] Zhiyuan Sun, Zihao Li, Xinghao Peng, Xiapu Luo, Muhui Jiang, Hao Zhou, and Yinqian Zhang. 2024. DoubleUp Roll: Double-spending in Arbitrum by Rolling It Back. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2577–2590. doi:10.1145/3658644.3690256
[37] Christof Ferreira Torres, Antonio Ken Iannillo, Arthur Gervais, and Radu State. 2021. CONFUZZIUS: A Data Dependency-Aware Hybrid Fuzzer for Smart Contracts. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 103–119. doi:10.1109/EuroSP51992.2021.00018
[38] Christof Ferreira Torres, Julian Schütte, and Radu State. 2018. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th annual computer security applications conference (ACSAC)*. 664–676. doi:10.1145/3274694.3274737
[39] Christof Ferreira Torres, Mathis Steichen, and Radu State. 2019. The Art of The Scam: Demystifying Honeypots in Ethereum Smart Contracts. In *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA, 1591–1607.
[40] Trail of Bits. 2023. Not So Smart Contracts. https://github.com/crytic/not-so-smart-contracts. Accessed: 2026-01-25.
[41] 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 (CCS)*. 67–82. doi:10.1145/3243734.3243780
[42] Fabian Vogelsteller and Vitalik Buterin. 2015. EIP-20: Token Standard. https://eips.ethereum.org/EIPS/eip-20. Accessed: 2026-01-25.
[43] Zhiyuan Wei, Jing Sun, Zijian Zhang, Xianhao Zhang, Meng Li, and Liehuang Zhu. 2023. A comparative evaluation of automated analysis tools for solidity smart contracts. *arXiv preprint arXiv:2310.20212* (2023).
[44] Hongbo Wen, Hanzhi Liu, Jiaxin Song, Yanju Chen, Wenbo Guo, and Yu Feng. 2024. FORAY: Towards Effective Attack Synthesis against Deep Logical Vulnerabilities in DeFi Protocols. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. Association for Computing Machinery, New York, NY, USA, 1001–1015. doi:10.1145/3658644.3690293
[45] Gavin Wood. 2014. Ethereum: A Secure Decentralised Generalised Transaction Ledger. https://ethereum.github.io/yellowpaper/paper.pdf. Ethereum Yellow Paper.
[46] Siwei Wu, Zhou Yu, Dabao Wang, Yajin Zhou, Lei Wu, Haoyu Wang, and Xingliang Yuan. 2024. DeFiRanger: Detecting DeFi Price Manipulation Attacks. *IEEE Transactions on Dependable and Secure Computing* 21, 4 (2024), 4147–4161. doi:10.1109/TDSC.2023.3346888

[47] Jiashuo Zhang, Jianbo Gao, Yue Li, Ziming Chen, Zhi Guan, and Zhong Chen. 2022. Xscope: Hunting for cross-chain bridge attacks. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1–4. doi:10.1145/3551349.3559520

[48] Mengya Zhang, Xiaokuan Zhang, Yinqian Zhang, and Zhiqiang Lin. 2024. Security of Cross-chain Bridges: Attack Surfaces, Defenses, and Open Problems. In *Proceedings of the 27th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2024)*. Association for Computing Machinery, 298–316. doi:10.1145/3678890.3678894

[49] Peilin Zheng, Zibin Zheng, Jiajing Wu, and Hong-Ning Dai. 2020. XBlock-ETH: Extracting and Exploring Blockchain Data From Ethereum. *IEEE Open Journal of the Computer Society* 1 (2020), 95–106. doi:10.1109/ojcs.2020.2990458

Received 2026-01-30; accepted 2026-04-16