ChainDelta: Automatic Patch-Based Exploit Generation for Ethereum with Fuzzing Agents

MINGXI YE[†], Sun Yat-sen University, China
YUHONG NAN[*†], Sun Yat-sen University, China
ZHIJIE ZHONG[†], Sun Yat-sen University, China
JIANZHONG SU[†], Sun Yat-sen University, China
XINGWEI LIN, Zhejiang University, China
PEILIN ZHENG[†], Sun Yat-sen University, China
ZIBIN ZHENG[†], Sun Yat-sen University, China

Given the critical nature of Ethereum, exploiting 1-day vulnerabilities that are patched but not yet widely deployed is essential. Meanwhile, Automatic Patch-based Exploit Generation (APEG) is a promising technique for this, as it helps developers understand root causes, verify fixes in downstream forks, and detect incomplete patches. However, existing exploit generation tools can not work well for vulnerabilities on Ethereum due to three key unique challenges: (1) navigating complex and cross-language exploit paths hidden within patches, (2) synthesizing complicated and stateful environment configurations, and (3) handling non-deterministic inconsistencies between blockchain nodes that lead to false alarms.

To address these challenges, we introduce *ChainDelta*, a novel fuzzing agent framework driven by Large Language Models to automatically generate exploits based on Ethereum security patches. *ChainDelta* consists of three core modules: a directed fuzzer utilizes call graph analysis to guide testing towards vulnerable code based on the patch information; an agent-based environment fuzzer acts as an expert to automatically set up the necessary blockchain states to trigger vulnerabilities; and finally, a state-aware sanitizer performs differential analysis while monitoring the blockchain transient state to distinguish true inconsistencies from benign non-determinism.

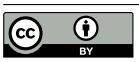
We evaluate *ChainDelta* on a diverse benchmark with real-world patches, covering a wide range of types such as data racing and denial-of-service. *ChainDelta* successfully generated exploits with a 64% success rate and only a 15.8% false positive rate. An ablation study confirms the contribution of each module to the overall performance. To demonstrate its practical impacts, we conducted a real-world auditing campaign on top of *ChainDelta*, leading to the discovery of four previously undisclosed vulnerabilities with bug bounties.

CCS Concepts: • **Security and privacy → Software and application security**.

Additional Key Words and Phrases: Blockchain Security, Exploit Generation, Directed Fuzzing, Large Language Model Agents

[*]Corresponding author.
[†]Also with Guangdong Engineering Technology Research Center of Blockchain.

Authors' Contact Information: Mingxi Ye, Sun Yat-sen University, China, yemx6@mail2.sysu.edu.cn; Yuhong Nan, Sun Yat-sen University, China, nanyh@mail.sysu.edu.cn; Zhijie Zhong, Sun Yat-sen University, China, zhongzhj3@mail2.sysu.edu.cn; Jianzhong Su, Sun Yat-sen University, China, sujzh3@mail2.sysu.edu.cn; Xingwei Lin, Zhejiang University, China, xwlin.roy@zju.edu.cn; Peilin Zheng, Sun Yat-sen University, China, zhengpl3@mail2.sysu.edu.cn; Zibin Zheng, Sun Yat-sen University, China, zhzibin@mail.sysu.edu.cn.



ACM 2994-970X/2026/7-ARTFSE150
https://doi.org/10.1145/3808157

ACM Reference Format:
Mingxi Ye, Yuhong Nan, Zhijie Zhong, Jianzhong Su, Xingwei Lin, Peilin Zheng, and Zibin Zheng. 2026. ChainDelta: Automatic Patch-Based Exploit Generation for Ethereum with Fuzzing Agents. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE150 (July 2026), 23 pages. https://doi.org/10.1145/3808157

1 Introduction

Blockchain security requires both proactive vulnerability detection and automated exploit generation to verify fixes, especially in high-value ecosystems like Ethereum. However, core software like `go-ethereum` [4] evolves rapidly, often leading to complex or incomplete patches. As many projects (e.g., Binance Smart Chain [3]) fork Ethereum, security fixes must be accurately propagated downstream. Failure to verify these patches with concrete exploits leaves the entire ecosystem vulnerable to 1-day attacks.

Exploit generation in blockchain systems goes beyond merely finding a bug. As shown in Figure 1, triggering an integer truncation bug in the `transfer` function requires a specific exploit environment, not just malicious input. An adversary must first prepare the environment by deploying smart contracts (i.e., the `Exploit` and the `WETH` contract). Once configured, a malicious payload navigates the cross-language path between the blockchain implementation and the smart contracts to trigger the bug. Only then can the malicious payload be propagated to the vulnerable function (i.e., the `transfer` function) and eventually cause unlimited creation of cryptocurrencies in the `WETH` contract.

The manual effort and deep expertise required for exploit generation motivate the need for automated solutions. One such solution is Automatic Patch-based Exploit Generation (APEG) [10], a technique that analyzes the differences between a vulnerable and a patched version of a program to automatically synthesize an exploit. Applying APEG to blockchain systems offers three benefits. First, it provides developers with a concrete proof-of-concept [46], helping to understand root causes and impacts of vulnerabilities. Second, it gives forking projects a reliable way to verify that a security patch is correctly integrated [72]. Finally, the generated exploit can also serve to proactively find similar 1-day vulnerabilities in other projects across the ecosystem before they are widely exploited by an adversary [70].

Challenges. Despite significant effort having been invested in blockchain security, existing methodologies are not designed for automated exploit generation from patches. Current research mainly focuses on proactive vulnerability detection [57, 65, 69, 71], prioritizing broad code coverage over the precise reachability required for specific patches. Conversely, traditional APEG tools [7, 10, 74] target standalone executable binaries that consume one-off inputs, failing to account for

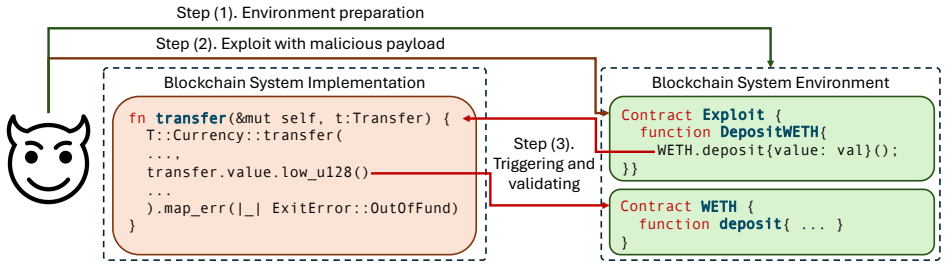


Fig. 1. An example exploit targeting an integer truncation vulnerability in blockchain infrastructure. The exploit requires three steps: (1) preparing the exploit environment by deploying the prerequisite `Exploit` and `WETH` contracts; (2) navigating to the vulnerable code via a crafted payload based on the identified malicious call trace; (3) validating the exploit by triggering the unlimited creation of cryptocurrencies (i.e., `WETH`).

the complex execution environment of a blockchain node. In particular, we identify three key challenges for generating valid exploits from blockchain system patches:

- **Navigating cross-language paths.** Existing techniques, such as directed fuzzing, focus on intra-language structural reachability. However, Ethereum vulnerabilities often span the blockchain implementation and Ethereum Virtual Machine (EVM) bytecode, linked by cross-language semantics. Reaching a patched location necessitates specific contract interactions. Simply extending call graphs is insufficient, as tools lack the reasoning capabilities to connect low-level contract execution with high-level node logic.
- **Configuring the complex environment.** Existing APEG tools synthesize payloads for target processes, assuming static or mocking environments. In blockchain systems, triggering vulnerabilities requires a specifically crafted environment, such as a multi-node testnet for reproducing consensus bugs or precise smart contract deployment sequences. Lacking exploit environment synthesis, current tools cannot satisfy the complex preconditions required to trigger blockchain vulnerabilities.
- **Exploit validation.** Conventional sanitizers focus on detecting immediate memory corruption or output divergence in deterministic settings. Differential analysis between a vulnerable and patched program can reveal execution inconsistencies. However, non-deterministic factors inherent in blockchains (e.g., transaction ordering) often cause legitimate inconsistencies. Without state-aware invariants, existing tools struggle to distinguish genuine exploits from benign execution differences.

These challenges require semantic reasoning rather than just syntactic exploration. For instance, satisfying cross-language checks requires understanding smart contract logic. Large Language Models (LLMs) are uniquely suited for this task, as they can translate these high-level semantic hints into concrete execution strategies.

Our work. In this paper, we present *ChainDelta*, a novel fuzzing agent framework driven by Large Language Model (LLM) to automatically generate exploits for blockchain system patches. To address the unique challenges of the blockchain execution model, *ChainDelta* is consisted of three key modules: (1) a directed patch fuzzing module that dynamically identifies reachable code paths and gathers runtime execution feedback; (2) an agent-based environment fuzzing module that leverages the feedback to synthesize complex blockchain states; and (3) a state-aware sanitizer that performs differential analysis while accounting for blockchain state to distinguish vulnerabilities from benign behaviors.

The directed patch fuzzing module serves as the framework foundation, focusing on reaching targeted code locations identified in a patch. This module utilizes call graph analysis to map execution paths and employs directed fuzzing strategies to guide the search. Throughout the process, it shares runtime feedback (e.g., code coverage and execution traces) with the environment fuzzer. This feedback is essential for identifying execution bottlenecks, particularly paths spanning the underlying blockchain implementation and EVM bytecode. As the primary execution engine, this module provides the visibility required to track exploit progress across multiple execution layers.

The agent-based environment fuzzing module employs a multiple agent framework to synthesize exploit environments and navigate cross-language paths based on collected feedback. The architecture decouples tasks to prevent context overflow. An observation agent analyzes runtime feedback to identify bottlenecks (e.g., complex cross-language checks). An execution agent reasons about these obstacles, manipulating the blockchain environment to satisfy exploit preconditions. A corpus agent translates semantic environment configurations into valid seeds for the corpus of the

directed fuzzer. This allows *ChainDelta* to actively assist the directed fuzzer in breaking through cross-language challenges.

The state-aware sanitizer module acts as an oracle to validate exploits while minimizing false positives. Traditional differential analysis often fails in blockchain settings due to non-deterministic factors. Our module addresses this by monitoring transient blockchain state variables during differential execution. By correlating behavioral inconsistencies with these state-dependent factors, the sanitizer distinguishes genuine exploit from legitimate execution variations inherent to the consensus model.

We conducted a comprehensive evaluation of *ChainDelta* using a representative benchmark of real-world blockchain system patches, detailed in Section 6. We evaluate *ChainDelta* in terms of efficacy and efficiency. *ChainDelta* successfully generated exploits for Ethereum patches, achieving a success rate of 64% with a false positive rate of only 15.8%. Furthermore, we performed an ablation study to show the contribution of each module. To further validate the effectiveness in real-world scenarios, we perform real-world auditing and present an in-depth case study on an incomplete patch. Our efforts revealed four undisclosed vulnerabilities with three being confirmed, and we received $2,000 bug bounty.

The contributions of this paper are summarized as follows:

- We design and implement *ChainDelta*, a novel fuzzing agent framework with a dual fuzzer architecture. The design coordinates directed patch fuzzing and agent-based environment fuzzing to efficiently generate exploits.
- We propose a novel state-aware sanitizer for the differential analysis of blockchain systems. By monitoring transient state variables, our technique accurately distinguishes exploitable vulnerabilities from benign behaviors.
- We conduct a comprehensive evaluation on real-world scenarios, demonstrating the efficacy of *ChainDelta* in exploiting patches. We further show that these exploits can be used to discover undisclosed vulnerabilities, along with a case study of exploiting real-world patches.

The rest of this paper is organized as follows: Section 2 provides necessary background knowledge. Section 3 presents a motivating example to demonstrate the key idea of *ChainDelta*. Section 4 elaborates on the detailed designs of *ChainDelta*. Section 5 clarifies implementation details. Section 6 presents the experimental setup and evaluation of *ChainDelta*. Section 7 discusses related work, and Section 8 concludes the paper.

2 Background

2.1 Patch-Based Exploit Generation

Exploit generation [8] is the process of crafting a specific input that triggers a known vulnerability to demonstrate its impact, which can range from a simple program crash to arbitrary code execution. A security patch serves as a powerful hint for this process [10], as it precisely pinpoints a vulnerable location and reveals the conditions that were changed to fix it. This makes patch analysis a highly efficient starting point compared to blind testing.

Previous work in automatic patch-based exploit generation (APEG), such as that by Brumley et al. [10], relied heavily on static analysis. They compare the vulnerable and patched versions of a program to symbolically reason about path differences and synthesize an exploit. However, these methods often struggle with the complexity and path explosion inherent in real-world applications [70].

To overcome these limitations, modern approaches now integrate patch analysis with directed fuzzing [68]. Instead of attempting to formally solve for an exploit, the patch analysis is used to generate guidance for a fuzzer. Tools like `SemFuzz` [73] and `Magneto` [74] use this guidance to focus

the fuzzer's mutations toward the specific code locations identified in the patch, combining the scalability of fuzzing with the precision of patch analysis.

2.2 Fuzzing and Sanitizers

Fuzzing is an automated software testing technique that randomly feeds invalid or unexpected inputs to a target program, in order to uncover bugs [51]. A fuzzer typically consists of three core modules, namely a generator that creates test cases, an executor that runs them, and an oracle that detects failures.

Input generation strategies can be divided into mutation-based or generation-based. Mutation-based fuzzers (e.g., `AFL` [60]) operate in a feedback loop. These fuzzers modify an initial set of seeds and use code coverage to guide mutations toward interesting program patches. In contrast, generation-based fuzzers create inputs from scratch using a formal grammar or model of the expected input structure [15], which is effective for testing highly structured targets like network protocols.

The oracle is critical for identifying bugs. For memory safety vulnerabilities, sanitizers like AddressSanitizer (i.e., `ASan`) [62] detect errors like buffer overflows at runtime. For uncovering functional bugs, differential testing [58] finds bugs by flagging inconsistencies in the output of two or more implementations of the same specification. Additionally, metamorphic testing [11] verifies program correctness by checking for violations of expected relationships between different but related inputs and outputs.

3 Motivation

This section investigates a motivating example of a real-world vulnerability [1] that yielded one of the biggest bug bounties, to illustrate the challenges of generating exploits for blockchain patches.

3.1 Motivating Example

Figure 2 presents a real-world vulnerability that highlights the challenges of generating exploits for blockchain system patches. This patched vulnerability in Figure 2(a) is an integer truncation bug in a native token transfer function. Triggering it requires setting the environment in a specific state by deploying smart contracts, as shown in Figure 2(b).

The root cause of the vulnerability lies in the token transfer logic. The function in Figure 2(a) takes a transfer amount as a `u256` integer and passes it to an internal function (i.e., `T::Currency::transfer`), which expects a `u128` integer. This down-casting was assumed to be safe because the total cryptocurrency supply is less than `u128::MAX`, preventing any adversary from directly transferring a larger amount. However, the developers overlooked a critical corner case. A smart contract is not bound by this same supply limit and can invoke the function with an arbitrarily large value, as shown in the `Exploit` contract (Figure 2(b), Line 17).

The exploit path requires a complex interaction across multiple languages, including the blockchain implementation in Figure 2(a) and the two contracts in Figure 2(b). An adversary calls the `depositWETH` function (Figure 2(b), Line 15), providing a small amount of the native token (e.g., one wei [67]) as `msg.value`. However, the actual passed number, namely the `val` parameter, is very large (e.g., $2^{128} + 1$ wei). Then, the actual transferred value is truncated from `u256` to `u128`, resulting in only sending one unit of the native token while receiving $2^{128} + 1$ units of the wrapped token, effectively creating tokens from nothing.

3.2 Challenges

The motivating example highlights three key challenges in analyzing and exploiting vulnerabilities from patches in blockchain systems.

```
fn transfer(&mut self, transfer: Transfer) -> Result<(), ExitError> {
    let source = T::AddressMapping::into_account_id(t.source);
    let target = T::AddressMapping::into_account_id(t.target);
@@ -667,7 +676,10 @@
    T::Currency::transfer(
        &source,
        &target,
-       transfer.value.low_u128().unique_saturated_into(),
+       transfer
+           .value
+           .try_into()
+           .map_err(|_| ExitError::OutOfFund)?,
        ExistenceRequirement::AllowDeath,
    ).map_err(|_| ExitError::OutOfFund)
}
```

(a) A patch for a native token transfer function. The highlighted integer truncation from u256 to u128 is the root cause of the vulnerability, which can be exploited to mint an arbitrary number of cryptocurrencies.

```
contract WETH {
    function deposit() public payable {
        balanceOf[msg.sender] += msg.value;
        Deposit(msg.sender, msg.value);
    }
    function withdraw(uint wad) public {
        require(balanceOf[msg.sender] >= wad);
        balanceOf[msg.sender] -= wad;
        msg.sender.transfer(wad);
        Withdrawal(msg.sender, wad);
    }
}

contract Exploit {
    function depositWETH() payable external {
        uint256 val = msg.value + (1 << 128);
        WETH.deposit{value: val}();
    }
    function withdrawETH(uint256 amount) external {
        WETH.withdraw(amount);
    }
}
```

(b) Smart contracts required for vulnerability exploiting, which serve as the entry point of the exploiting path to trigger the vulnerable blockchain implementation. The `WETH` contract wraps the native token, and the `Exploit` contract provides the entry point to call the vulnerable transfer function with a malicious payload.

Fig. 2. A real-world integer truncation vulnerability leading to significant financial loss. Exploiting this vulnerability is challenging because it requires complicated exploitation paths across multiple languages. It requires: (1) deploying contracts to set up necessary blockchain states; (2) a complex sequence of smart contract invocations to trigger the vulnerable code; and (3) observing the inconsistent state change in the contract storage to confirm the exploit.

First, Ethereum clients employ a execution model where control flow transitions between the host implementation and the guest EVM bytecode. These layers interact via transactions rather than static function calls. Existing directed fuzzers target structural reachability within a single language. In the example, the vulnerability is not reachable via a simple RPC call without a sequence of contract deployments and invocations. Furthermore, the root cause (Figure 2(a), Line 8) is semantically subtle and easily overlooked within a large patch [2], making exploit path discovery difficult.

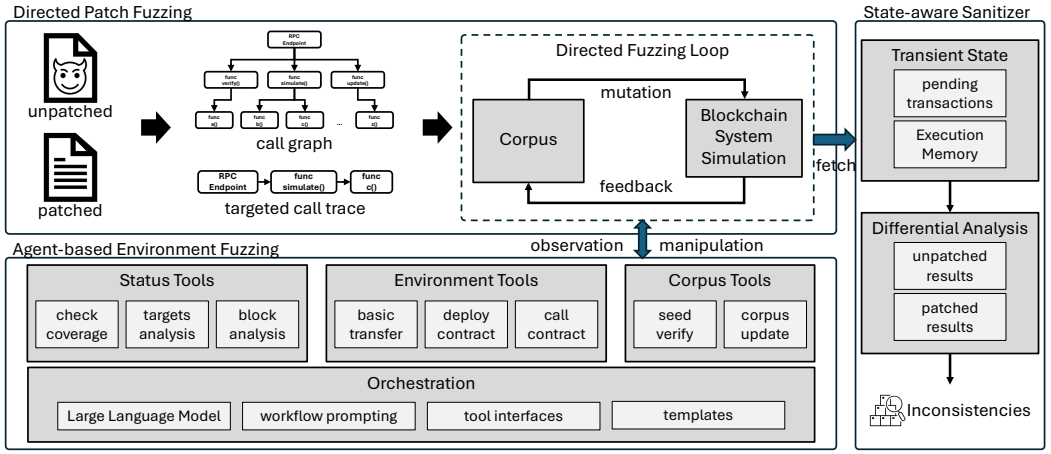


Fig. 3. Overview of the *ChainDelta* framework, illustrating its three integrated modules designed to generate exploits for blockchain system patches. The directed patch fuzzing module applies directed fuzzing strategies towards vulnerable code paths using call graph analysis. The agent-based environment fuzzing module dynamically sets up a complex blockchain environment and smart contract interactions. The state-aware sanitizer performs differential analysis, leveraging transient blockchain state to distinguish true vulnerabilities from benign inconsistencies.

Second, blockchain vulnerabilities are often gated by cumulative state changes. The preconditions of this exploit demand a specific blockchain environment that cannot be trivially created. Traditional APEG tools typically focus on synthesizing localized payloads for stateless processes. The vulnerability is only exposed after deploying the smart contracts (i.e., `WETH` and `Exploit` in Figure 2(b)) to interact with the token transfer logic. Reproducing this scenario requires domain-specific knowledge of how wrapped tokens and contract interactions work. This need for a complex setup goes beyond the capabilities of traditional fuzzing techniques that primarily focus on stateless inputs or simple API sequences. Since the infinite permutations of blockchain states render exhaustive search infeasible, a multi-agent architecture is necessary to prune the state space using real-time fuzzer feedback.

Finally, blockchains are inherently non-deterministic due to the consensus mechanism. Thus, defining a precise oracle to detect this exploit is non-trivial. Standard differential testing frameworks typically focus on strict output equivalence, assuming any divergence indicates a bug. For this specific vulnerability, the oracle must check for an illicit change in the `WETH` token balance, which is a highly specific and non-generalizable condition. A more generic approach, such as differential testing between the vulnerable and patched versions of blockchain systems, is prone to false alarms. Benign inconsistencies can easily arise from non-deterministic factors inherent in blockchain networks (e.g., transaction ordering or block timing), making it difficult to isolate the true vulnerable symptoms without a state-aware sanitizer.

4 Detailed Design of *ChainDelta*

4.1 Overview

Figure 3 provides an overview of *ChainDelta*, our fuzzing agent framework designed to generate exploits for patched vulnerabilities in Ethereum. *ChainDelta* includes three key modules, each addressing a specific challenge outlined in the previous section.

First, the directed patch fuzzing module applies call graph analysis on the blockchain system codebase and patches. This process pinpoints the precise call traces required to reach and exercise the modified code locations from externally exposed interfaces (e.g., RPC endpoints). However, guiding fuzzing effectively exploring blockchain environments is still complex due to the cross-language execution contexts and specific on-chain state preconditions. Thus, this module focuses on general directed fuzzing strategies for blockchain system implementations.

Then, the agent-based environment fuzzing module complements the directed patch fuzzing module. It acts as an intelligent expert, leveraging LLM to orchestrate a complex environment setup based on the key idea of environment fuzzing. This agent utilizes a suite of tools to observe the patch fuzzing status, deploy and interact with smart contracts, manipulate blockchain states, and dynamically update the fuzzing corpus. This cooperative approach allows *ChainDelta* to overcome state-dependent obstacles and guide the fuzzer toward exploring deeper states.

Finally, the state-aware sanitizer module functions as the oracle for *ChainDelta*. It conducts differential analysis by comparing the behaviors and outputs of the vulnerable and patched versions of blockchain systems. Note that this analysis is enhanced by fetching and incorporating runtime transient state information (e.g., mempool contents and pending transactions) to filter out false alarms. This state awareness allows the sanitizer to accurately filter out benign inconsistencies that arise from the non-deterministic nature of blockchain networks, ensuring that true vulnerabilities are reported.

4.2 Directed Patch Fuzzing

The primary goal of this module is to focus the fuzzing efforts on the code locations affected by a security patch. It consists of two main components, namely a call graph analysis that identifies target locations and a directed fuzzing loop that executes the test. The fuzzing loop is built on top of a mutation-based fuzzing architecture, designed with specific interfaces. Those extra interfaces, as indicated with comments in Algorithm 1, allow other modules (e.g., the LLM agent and the sanitizer) to observe and control the fuzzing process.

Call graph analysis. To generate guidance for the fuzzing loop, we extract targeted call traces within the blockchain infrastructure (i.e., client implementation), namely the sequences of function calls that can reach a patched code location from a publicly exposed interface. This process involves the identification of both the destination and the source, taking the full source code of both the vulnerable and patched versions as input.

We identify the patched code locations by building call graphs at the basic block level for both the vulnerable and the patched versions of the blockchain system. By comparing these graphs, we pinpoint the functions and basic blocks that were modified without requiring manual patch identification.

We find potential entry points based on the insight that different blockchain clients share common interface patterns for external communication, due to the requirements of the decentralized nature of blockchain networks. Our analysis applies a set of heuristics based on function signatures to identify publicly exposed functions that serve as entry points, such as RPC handlers and peer-to-peer message processors, which often have a specific function signature.

With both the entry point and patched code locations identified, we can traverse the call graph to automatically extract the relevant call traces that guide the fuzzing loop.

Directed fuzzing loop. The core fuzzing logic is shown in Algorithm 1. While based on a traditional mutation-based architecture, our fuzzing loop is extended with several key procedures to enable external control. The key idea is inspired by how a human expert might manually guide a

Algorithm 1 The fuzzing loop of *ChainDelta*.

Input: The initial corpus, S; The set of targeted call trace, T
Shared Memory: The dynamic corpus, S_s; The transient states, S_t; Execution requests, R

```
1:  for reach timeout do
2:      for s in S do
3:          S_m = MUTATE(s)
4:          for s_m in S_m do
5:              EXECUTE(R)                    ▷ Observe status and manipulate fuzzing environment
6:              is_interesting ← EXECUTE(s_m, T)
7:              UPDATE(S_t)                   ▷ Update transient states for external references
8:              if is_interesting is true then
9:                  S ← s_m
10:             end if
11:             SCANCORPUS(S_s)               ▷ Allow inserting seeds to the corpus externally
12:         end for
13:     end for
14: end for
```

fuzzer when hunting vulnerabilities. The whole procedure is divided into two parts according to its inputs, namely the initial fuzzing inputs and shared memory interfaces.

The loop starts with an initial seed corpus and the set of targeted call traces from the analysis phase. These traces are used to prioritize seed inputs. A test case is considered interesting and saved for future mutation if it increases basic block coverage within any of the targeted call traces or achieves distinct coverage within the targeted call traces in the vulnerable and patched version of the blockchain system.

The fuzzer exposed three customized data structures in shared memory, allowing other modules to interact with them dynamically (Algorithm 1, Lines 5, 7, and 11). The corresponding steps are highlighted with comments in Algorithm 1. The dynamic corpus allows the LLM agent to inject new, strategically crafted seed inputs that are relevant to the current on-chain environment. The transient state is logged after each execution with runtime information, making it available to the state-aware sanitizer for analysis. The execution request queue allows the agent to issue specific commands, such as observing current fuzzing status or manipulating blockchain environments. The `Execute` primitive is defined with two distinct signatures. Line 5 handles the environment manipulation phase, processing the execution request queue (i.e., R) issued by the environment agent. Line 6 handles the vulnerability fuzzing phase, executing the mutated seed (i.e., s_m) under the guidance of targeted call traces (i.e., T).

4.3 Agent-Based Environment Fuzzing

To navigate the complex state space of the Ethereum ecosystem and bridge the cross-language semantic gap, *ChainDelta* applies a multi-agent framework. Unlike single-agent architectures, which suffer from reasoning degradation when processing the massive context of a global blockchain state, our decoupled design enables specialized reasoning loops. As shown in Figure 4, these agents coordinate in a loop to transform execution feedback into valid exploit seeds.

- The observation agent is responsible for the semantic interpretation of directed fuzzer progress. It monitors runtime feedback, specifically targeting execution paths blocked at cross-language boundaries or state-dependent guard conditions. With source code access, the agent analyzes execution traces to hypothesize missing environmental dependencies.

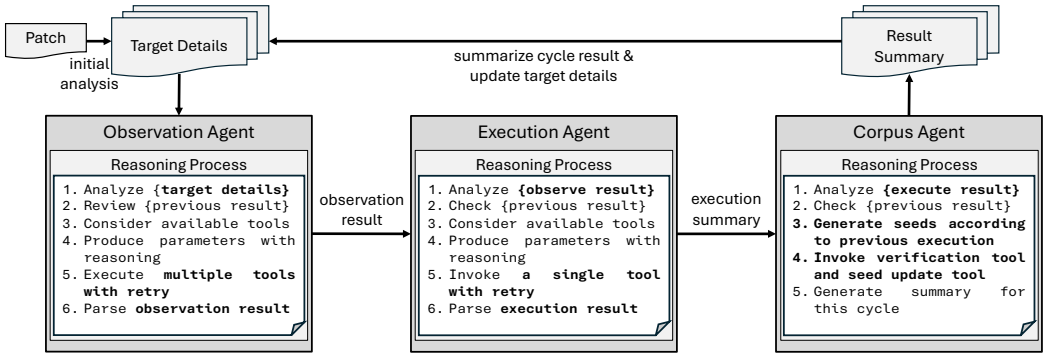


Fig. 4. *ChainDelta* employs a multi-agent framework to decouple tasks in environment fuzzing. The observation agent takes patch details and previous results as input, identifying semantic bottlenecks from runtime feedback. The execution agent reasons about the feedback to manipulate the blockchain environment. The corpus agent translates these high-level states into fuzzer seeds. This architecture performs in a loop, enabling complex exploit precondition configuration and cross-language execution navigation.

- Upon receiving the observation report, the execution agent synthesizes the necessary global state configuration. It reasons about the dependencies and generates a sequence of transactions (e.g., deploying auxiliary contracts or minting tokens) to construct the precise exploit environment.
- The corpus agent translates the environment configurations into valid seeds for the directed patch fuzzer. By providing inputs that satisfy complex environmental checks, the agent enables the directed fuzzer to bypass cross-language challenges statically unreachable via traditional bit-level mutation.

The necessity of this multi-agent design lies in the vast blockchain state space and the limited context window of LLMs. By decoupling independent tasks, *ChainDelta* preserves reasoning precision and prevents context overflow. This coordination ensures the framework does not merely perform random state manipulation, but executes a targeted search within the blockchain state space to unblock the directed patch fuzzer.

Tools. The observation agent is equipped with tools for information collection. These tools allow it to query the status of the directed patch fuzzing, inspect the state of the simulated blockchain environment, and extract semantic details from the security patch itself. To construct a comprehensive snapshot of the current context in each cycle, the agent is permitted to invoke multiple tools multiple times.

The execution agent is equipped with tools designed to manipulate the fuzzing environment. These include tools for direct blockchain interaction (e.g., deploying smart contracts or sending transactions) and for modifying configuration. To ensure that state changes are incremental, the fuzzer operates within a unified context, resetting only upon a predefined timeout. To ensure their effects can be clearly attributed, the executor is restricted to invoking only a single tool per cycle, though it may retry if an error occurs.

Finally, the corpus agent has tools focused on quality control. It uses a syntax verification tool to ensure the validity of generated seed inputs and a corpus deduplication tool to prevent redundant seeds from being added to the corpus.

Prompting. We guide our agents using a structured prompting strategy, as illustrated in Figure 5. A global system prompt (Figure 5(a)) is shared among all agents. This prompt establishes the

System Prompt

System: You are a blockchain security expert and fuzzing specialist working with ChainDelta, an advanced differential fuzzer for Ethereum clients.

Your Mission: ChainDelta is actively fuzzing to find discrepancies between two versions of a blockchain client. Your critical role is to:
1. **Understand the patch and potential bugs**: Analyze what changed and what vulnerabilities might exist.
2. **Set up proper fuzzing environments**: Create the blockchain state necessary to trigger bugs.
3. **Generate effective seeds**: Provide inputs that help the fuzzer explore vulnerable code paths.

Your Fuzzing Targets: {endpoint details}

Your Workflow: You operate in cycles, each consisting of:
1. **Observation**: Analyze current blockchain state and identify setup needs.
2. **Execution**: Create blocks, deploy contracts, send transactions to build the environment.
3. **Seed Update**: Generate new fuzzing inputs based on findings.

(a) The shared system prompt, providing all agents with their global mission, targets, and workflow.

Observation Prompt

Current Stage: You need to observe and analyze the blockchain targets to understand what environment setup is needed for effective fuzzing.
Previous Context: {previous_step_output}.

Available Tools: {tools}.
Tool names: {tool_names}.

Your Task:
1. Consider the patch and vulnerability type from the system context
2. Identify what blockchain state might be needed to trigger the bug
3. Perform observation to helps you understand current state and setup needs. Note that you can invoke tools multiple times before success.
4. Analyze results for environment setup needs and identify what's missing for bug triggering

Required Output: your response MUST end with this metadata block
```
{example}
```

(b) An example of a role-specific task prompt for the observation agent, detailing its per-cycle objective, available tools, and output requirements.

Fig. 5. An example of our structured prompting strategy. The system prompt provides all agents with their shared mission, fuzzing targets, and overall workflow. The observation prompt is an example of an agent-specific task prompt, detailing its current objective, available tools, and required output format.

collective mission, high-level workflow, and fuzzing targets across all agents. Building on this foundation, each agent receives a dynamic task prompt (e.g., Figure 5(b)) in every operational cycle. This prompt is tailored to the agent's specific role and provides all necessary operational context. The prompt includes recent history, a list of available tools, a concrete objective, and a required output format guided by few-shot prompting. To enhance robustness, the prompt also includes an error-handling directive that instructs the agent to retry a failed action with the error context. This prompting design provides a long-term strategic mission while enabling flexible and context-aware actions in each step.

4.4 State-Aware Sanitizer

Our sanitizer identifies vulnerabilities by performing a differential comparison between the vulnerable and patched versions of the blockchain systems after executing the same test case. The comparison targets across two dimensions, including internal state and external behavior. We perform a differential state comparison to find illicit changes. In the meantime, we detect behavioral inconsistencies by comparing blockchain system outputs (e.g., RPC responses) for the same test case. However, standard differential analysis is often hindered by the inherent non-determinism of blockchain consensus. To overcome this, the sanitizer employs the state-aware sanitizer shown in Algorithm 2, performing a semantic diff rather than a raw execution trace comparison.

Algorithm 2 State-Aware Sanitizer

Require: Traces E_{vuln}, E_{patch}, Global State S_{global}
Ensure: Verdict $V \in \{\text{Bug}, \text{Benign}\}$
1: $\Delta_{raw} \leftarrow \text{DiffState}(E_{vuln}, E_{patch})$
2: **if** $\Delta_{raw} = \emptyset$ **then**
3: **return** Benign
4: **end if**
5: // Identify if differences are transient (not yet committed to a block)
6: $\Delta_{permanent} \leftarrow \text{FilterCommitted}(\Delta_{raw}, S_{global})$
7: **if** $\Delta_{permanent} = \emptyset$ **then**
8: **return** Benign ▷ Transient state divergence detected
9: **end if**
10: **return** Bug

As formalized in Algorithm 2, the sanitizer utilizes state-aware logic to address blockchain non-determinism. It takes the global blockchain state S_{global} to evaluate the persistence of observed divergences. The FilterCommitted function distinguishes between transient noise (e.g., mempool transactions or temporary execution memory) and permanent state changes committed to a block. A divergence is flagged as a bug only if it results in a persistent discrepancy in the permanent state (i.e., S_{global}). Given identical inputs, such permanent divergence indicates a departure from expected consensus behavior, identifying the vulnerability without manual validation. This mechanism ensures accuracy and reproducibility by filtering benign inconsistencies.

If the final divergence can be attributed to an initial benign difference in the transient state, the finding is classified as non-malicious and discarded, as the divergences are caused by the non-deterministic factors of blockchain networks. While this state-aware filtering solves the accuracy problem, comparing the entire multi-gigabyte blockchain state is still computationally prohibitive. To address this performance bottleneck, we implement a tracer that logs only states accessed during execution. The sanitizer then compares only this small subset of touched states, drastically reducing overhead while maintaining accuracy.

5 Implementation

The implementation of *ChainDelta* consists of a multi-language architecture where each component is built with what best suits its task.

First, we developed a customized instrumentor in Go to support coverage collection, call graph analysis, and runtime state tracing. We chose to focus on blockchain systems written in Go initially, as it is the most widely used language for implementing major blockchain systems. The instrumentor injects extra code, which is designed to output runtime data to a unified shared memory structure, ensuring that the information is accessible to other modules regardless of their implementation language.

The foundation of our framework (i.e., the directed patch fuzzing and state-aware sanitizer) is implemented in Rust using the LibAFL [43] framework. We chose this stack for its high performance and the flexibility in constructing customized fuzzing loops and oracles. After instrumentation, each target blockchain system is packaged into a Docker container. This approach guarantees a consistent and reproducible testing environment that can be easily managed and orchestrated across the different modules of our system.

Finally, the agent-based environment fuzzer is built in Python using the LangChain framework [50]. This flexible framework allows for the easy development of our LLM-driven agents. Our

implementation includes the customized tools, designed prompts, state management, and workflow control logic necessary to orchestrate the pipeline of the multi-agent system.

6 Evaluation

To evaluate the effectiveness and practicality of *ChainDelta*, we conducted a comprehensive set of experiments using a collected benchmark. Our evaluation is structured around the following four research questions:

RQ1: How effective is *ChainDelta* at generating exploits for the diverse real-world vulnerabilities in our benchmark?
RQ2: To what extent does the agent-based environment fuzzer improve the exploration of state-dependent conditions required to trigger vulnerabilities?
RQ3: How effective is the state-aware sanitizer at reducing false alarms by correctly distinguishing benign inconsistencies and true vulnerabilities?
RQ4: Can the exploit generated by *ChainDelta* for known patches be used to discover undisclosed vulnerabilities in other projects?

To answer these questions, we first measure *ChainDelta*'s overall performance on our benchmark. We then conduct a detailed ablation study to isolate the contributions of its key modules. Finally, we conduct real-world auditing and present a case study to demonstrate its practical impact.

6.1 Experiment Setup

Benchmark construction. To construct a realistic benchmark for evaluating *ChainDelta*, we systematically collected real-world vulnerability patches from the dominant Ethereum client and a major fork (i.e., `go-ethereum` [4] and `op-geth` [41]). We focus on end-to-end exploitable vulnerabilities, as they require traversing the whole network stack, rather than simple unit-test mocking.
We employed a three-stage filtering pipeline to ensure the benchmark's representativeness and reproducibility. First, we aggregated all GitHub issues in `go-ethereum` and `op-geth` since 2024. Second, we filtered for issues labeled as bugs and linked to corresponding patches. Third, three authors with extensive experience in blockchain security research and CTF competitions manually analyzed each patch to verify its exploitability via public interfaces.

Benchmark diversity. Table 1 details the final benchmark, containing 25 real-world vulnerabilities with patches. While smaller than typical smart contract datasets, it is important to note that exploitable vulnerabilities in blockchain infrastructure are rarer. The benchmark is designed to be diverse and challenging to thoroughly evaluate the capabilities of *ChainDelta*. The vulnerabilities span a wide range of types, which include common software flaws like denial-of-service and data races. It also includes blockchain-specific issues such as incorrect virtual machine execution results and consensus bugs. Furthermore, the preconditions for triggering these vulnerabilities are also varied, requiring complex environment setups that involve manipulating the core blockchain status, deploying specific on-chain contracts, or even inducing abnormal network conditions like a chain reorganization. This diversity ensures a comprehensive test of *ChainDelta* performance to handle all three core challenges, namely navigating difficult code locations, configuring complex environments, and performing accurate sanitization.

Execution environments. All experiments were conducted on a server equipped with an Intel Xeon Gold 5218R CPU (@ 2.10GHz) and 512GB of RAM, running Ubuntu 20.04 LTS. The agent-based modules utilized the Gemini 2.5 Pro API [44]. All instrumented blockchain clients were packaged as Docker images to ensure a consistent and reproducible execution environment. For each vulnerability in our benchmark, we ran *ChainDelta* for a total of 60-minute timeout, with the directed patch fuzzer and the agent-based environment fuzzer operating concurrently.

Table 1. Our benchmark with 25 real-world blockchain system vulnerability patches, detailing each bug's root cause, symptom, and the specific blockchain environment required for its exploitation.

ID	Descriptions	Requirements
B1	Data race conditions when multiple goroutines simultaneously trace data, causing inconsistent trace results. [17]	Available block data for tracing.
B2	Improper state handling during chain reorganization, causing block indexing results to be inconsistent. [18]	Standard chain with frequent block reorganization.
B3	Tracers don't store interrupt reasons, returning incorrect trace data instead of errors. [19]	Heavy transaction load for trace interruption.
B4	No validation on parameter array size, allowing DoS by causing expensive sorting operations. [20]	Active block production allowing fee history queries.
B5	Gas estimator doesn't include blob gas in balance computation, returning incorrect result. [21]	EIP-4844 blob transactions with low-balance accounts.
B6	Gas estimator doesn't properly cap allowed gas usage, leaving a DoS attack vector. [22]	Standard chain with available RPC.
B7	Incorrect block context handling before tracing, causing incorrect opcode behaviors. [23]	Deployed contracts with the `BASEFEE` opcodes.
B8	Backward compatibility issue when tracing legacy transactions with an outdated format. [24]	Standard chain with unsupported legacy transactions.
B9	Incorrect connection handling for RPC client subscription, with no error handling. [26]	Standard network with clients for subscription.
B10	Incorrect configuration of the gas cap for execution, leaving a DoS attack vector. [27]	Standard chain with gas cap set as zero.
B11	Reactive block-sealing logic and potential deadlock, causing transactions to remain stuck in a pending state until a manual trigger occurs. [25]	Period zero mode with concurrent transaction submission.
B12	Incorrect path handling for the JWT secret flag can break Engine API authentication, preventing CL/EL communication. [28]	Execution of the Geth blsync tool with tilde-prefixed paths.
B13	Use of a minimal timestamp increment in simulation logic, causing transactions with time-dependent contract constraints to unexpectedly revert. [29]	JSON-RPC `eth_simulateV1` API calls with time-sensitive smart contract logic.
B14	EIP-712 typed-data hashing incorrectly encodes nested bytes arrays, leading to invalid signature verification. [30]	Incorrect recursive encoding of nested bytes arrays.
B15	Goroutine leak in the transaction indexer, causing the node to become unresponsive or stop syncing. [33]	High-traffic environment with frequent RPC queries.
B16	Redundant inclusion of EIP-7702 authority addresses in access list generation, causing inaccurate gas estimation results for RPC callers. [32]	Transactions utilizing EIP-7702 authorizations.
B17	A regression where raw blobGasUsed was returned instead of being divided by the block limit, breaking gas estimation tools. [31]	An EIP-4844-enabled chain with the queried window contains no blobs.
B18	Log indexing may return stale or duplicated results after chain rewinds, leading to incorrect event queries. [34]	Chain rewinds followed by log or filter queries.
B19	Block overrides in tracers are not correctly applied, returning incorrect trace results. [35]	Transactions for the trace APIs with block override parameters.
B20	`eth_getBlockReceipts` for pending blocks can return incomplete or inconsistent receipts, confusing clients. [36]	Receipts for the `pending` block with block production are ongoing.
B21	A performance bottleneck between block generation speed and log indexing speed, causing a deadlock scenario. [37]	Simulated backend with high block production speed and transactions.
B22	Pending transaction subscriptions may not terminate on backend errors, leaking goroutines and connections. [38]	WebSocket subscription to pending transactions under backend errors.
B23	A race condition where the TCP connection closes before the websocket close frame is sent, producing a different error than what the test previously expected. [39]	WebSocket clients receiving large payloads close to the read limit.
B24	Filter subscription uninstall is not triggered on errors, leaving dangling filters and resource leaks. [40]	RPC filter subscriptions encountering errors/timeouts.
B25	Fee history RPC can compute NaN when there are zero blobs, causing RPC failures. [61]	Nodes with blob fee history enabled and a window containing zero blobs.

6.2 Effectiveness

To answer RQ1, we evaluated the effectiveness of *ChainDelta* against state-of-the-art blockchain security tools (i.e., `EtherDiffer` [48] and `hive` [16]) and general-purpose fuzzers (i.e., `AFL++` [42] and `PatchFuzz` [63]). `EtherDiffer` is a blockchain differential fuzzer aligned with our patched-based analysis scenario. `hive` is the official integration testing suite by the Ethereum Foundation. Given the limitation of available tools for blockchain infrastructure security, we include `AFL++` and `PatchFuzz` as representative general-purpose fuzzer. We exclude other blockchain security tools, such as `BlockScope` [72], which focuses on migrating known bugs via code clone detection. Similarly, we omit smart contract tools (e.g., `Teether` [49] and `ItyFuzz` [64]), as they cannot target the underlying blockchain infrastructure.

Table 2. Effectiveness of *ChainDelta* compared against baselines on our benchmark. For each tool, we list the total number of inconsistencies found (i.e., reported) and the number of those verified as true positives (i.e., confirmed). Results for `EtherDiffer` are omitted, as it failed on all cases.

ID	***ChainDelta***		`hive`		`AFL++`		`PatchFuzz`	
	Reported	Confirmed	Reported	Confirmed	Reported	Confirmed	Reported	Confirmed
B1	1	1	403	1	7	0	4	0
B2	315	119	568	0	8	0	6	0
B3	442	0	425	1	7	0	7	0
B4	174,072	174,072	489	0	15	0	5	0
B5	1	1	406	0	12	1	7	1
B6	67,049	67,049	606	0	5	0	4	0
B7	8,272	8,272	581	0	8	0	7	0
B8	0	0	603	0	5	0	4	0
B9	0	0	602	0	6	0	7	0
B10	8,503	8,503	364	0	4	0	4	0
B11	1,865	1,865	922	0	7	0	4	0
B12	0	0	1,195	0	0	0	0	0
B13	20	9	1,139	2	0	0	0	0
B14	4	0	1,116	1	0	0	0	0
B15	0	0	72	0	6	0	5	0
B16	626	626	1.069	0	9	1	5	1
B17	47,395	242	1,112	0	335	323	22	18
B18	38,644	0	42	0	5	0	4	0
B19	0	0	33	1	6	0	5	0
B20	188	188	45	0	7	0	6	0
B21	481	481	32	1	9	0	4	0
B22	2,160	2,160	22	2	4	0	4	0
B23	0	0	22	1	9	0	4	0
B24	2,245	2,245	22	0	1	0	4	0
B25	2	2	0	0	17	0	4	0

Efficacy. The results of our comparison are summarized in Table 2. For each benchmark case, *ChainDelta* was provided with the security patch including code differences; `EtherDiffer` was provided with the vulnerable and the patched version as fuzzing targets; `hive` was run on the vulnerable version. To ensure a fair comparison, we manually developed harnesses mocking static blockchain network environments and implemented a differential analyzer to compare vulnerable and patched implementation results. Each tool was run for a 60-minute timeout. We define a true positive for *ChainDelta* as the successful generation of a working exploit. For others, a true positive is defined as reported inconsistencies that successfully identify the underlying vulnerability.

ChainDelta successfully generated exploits for 16 out of 25 vulnerabilities, achieving an 64% success rate. In contrast, `EtherDiffer` failed to identify any of these vulnerabilities. Other baselines (i.e., `hive`, `AFL++`, and `PatchFuzz`) detected only a limited number. It is convincing that *ChainDelta* outperforms state-of-the-art tools.

False positives. Our analysis of *ChainDelta* results revealed interesting cases. For B2, *ChainDelta* generated an exploit that worked on the patched version of the client, not the vulnerable one. This reveals that the patch was incomplete, and a potential attack vector remained. This demonstrates *ChainDelta* not just for verifying known vulnerabilities but for discovering flaws in the patches themselves. We also observed that benign inconsistencies were related to the state of pending

blocks or RPC response messages. While our state-aware sanitizer is designed to handle this, it represents a fundamental trade-off between detecting vulnerabilities that are related to pending blocks (e.g., B2) and avoiding false positives from benign non-determinism. In contrast, baselines suffer from high false alarms due to their inability to filter false alarms caused by transient states.

True negatives. *ChainDelta* failed to generate exploits for six cases, due to limitations in its environment interaction and oracle capabilities. We consider these outliers that highlight areas for future work. For example, B8 required deploying an unsupported legacy transaction type to the network. This action was outside the scope of the current agent implementation, which is limited to interacting with the client via standard endpoints. Meanwhile, the symptom of B9 is only observable with a client-side request timeout. We deliberately did not implement timing-based oracles, as they are notoriously prone to false positives caused by test environment fluctuations, making them unreliable for our automated framework. Note that `hive` is capable of identifying some vulnerabilities omitted by *ChainDelta*, as its detection relies on manually designed test suites. However, due to the high cost of manual effort, `hive` lacks scalability.

6.3 Ablation Study: Agent-Based Environment Fuzzing

To quantify the contribution of the agent-based environment fuzzing module, we created an ablated variant of our system, where this module was disabled. In this configuration, only the directed patch fuzzer operates, without any extra manipulation for on-chain environment setup. We then executed this variant on our entire benchmark using the same 60-minute timeout per case.

Impact on effectiveness. The results in the middle column of Table 3 demonstrate that the agent module is critical for overall effectiveness. While the ablated variant was able to trigger inconsistencies in 14 cases, only four were true positives. This represents a dramatic drop in the success rate from 64% to 16% and an increase in the false positive rate from 15.8% to 71.4%.

The analysis shows that without the agent, the directed fuzzer is unable to perform the necessary on-chain setup (e.g., deploying specific contracts) to create the preconditions for the more complex vulnerabilities. Instead, it gets stuck exploring shallower parts of the state space. This not only prevents it from reaching deep bugs but also causes it to repeatedly trigger benign inconsistencies in simple states (e.g., the false positives in B10), which the full *ChainDelta* framework avoids by having the agent guide the fuzzer toward more interesting states.

Performance overhead. The results in the middle column of Table 3 also reveal a clear performance trade-off. For the simple vulnerabilities that both systems could find (i.e., B4, B6, B7), the ablated variant found them much faster, with a Time to First Exploit (TTF) of under 10 seconds. The full *ChainDelta* framework required between one and 20 minutes for the same bugs.

This overhead is expected and stems from the extra inter-module communication (e.g., writing to shared memory). However, we argue this overhead is an acceptable and necessary trade-off. While it slows the detection of simple bugs, this capability is precisely what enables *ChainDelta* to solve the complex vulnerabilities that require deep state manipulation.

6.4 Ablation Study: State-Aware Sanitizer

To answer RQ3, we created the second ablated variant to isolate the contribution of our state-aware sanitizer. In this variant, we disabled the transient state filtering mechanism. The sanitizer was reconfigured to operate as a simple behavioral oracle, reporting any divergence in the blockchain system outputs (e.g., RPC responses) as a potential vulnerability. We then ran this variant on our benchmark under the same 60-minute timeout.

Impact on false positives. The results, shown in the rightmost columns of Table 2, highlight the critical role of the state-aware filter. Without it, the ablated variant was overwhelmed by an

Table 3. Ablation study on top of our benchmark, where we compare the performance of *ChainDelta* to its two ablated variants in terms of efficacy and efficiency.

ID	*ChainDelta*		*ChainDelta* without Agent-based Environment Fuzzing		*ChainDelta* without State-aware Sanitizer	
	Reported/TPs	Time	Reported/TPs	Time	Reported/TPs	Time
B1	1/1	**27min8s**	-/-	-	222/1	43min25s
B2	315/119	**11min18s**	-/-	-	1,583/0	-
B3	442/0	-	-/-	-	493/0	-
B4	174,072/174,072	44s	177,441/177,441	< 10s	222,342/222,342	45s
B5	1/1	**3min56s**	-/-	-	-/-	-
B6	67,049/67,049	5min47s	55,397/55397	< 10s	76,509/382	5min18s
B7	8,272/8,272	51s	6,426/6426	< 10s	8,408/8,408	52s
B8	-/-	-	-/-	-	31/0	-
B9	-/-	-	4/0	-	57/0	-
B10	8,503/8,503	**43s**	14/0	-	8,318/7,910	47s
B11	1,865/1,865	**6m41s**	1/0	-	34258/0	-
B12	-/-	-	-/-	-	143/0	-
B13	20/9	**4m37s**	1/0	-	152/0	-
B14	4/0	-	-/-	-	92/0	-
B15	-/-	-	1/0	-	171/0	-
B16	626/626	3m	1/0	-	6,391/64	1m6s
B17	47,395/242	**11s**	-/-	-	219/5	5m19s
B18	38,644/0	-	-/-	-	713,135/0	-
B19	-/-	-	1/0	-	91/0	-
B20	188/188	10m1s	1/0	-	3,694/133	49s
B21	481/481	**8m2s**	-/-	-	177/0	-
B22	2,160/2,160	5m1s	1/1	15m	3,961/72	30s
B23	-/-	-	1/0	-	140/0	-
B24	2,245/2,245	**10s**	-/-	-	40,799/0	-
B25	2/2	< **10s**	1/0	-	330/0	-
Success Rate Improvement			**4**×		**1.8**×	

Note that we did not create an ablated variant called *ChainDelta* without directed patch fuzzing, as this module acts as the foundation of the framework, where two other modules relied on it.

explosion of false positives, reporting thousands of inconsistencies across 22 of the 25 benchmark cases. The large volume of these alarms made manual verification infeasible. To analyze the results, we had to implement an automated verification script that scanned the outputs for predefined patterns corresponding to the known vulnerabilities. The variant confirms an extremely low true positive rate. This demonstrates that a naive differential oracle is insufficient and that state-aware filtering is essential for precision.

Performance overhead. The results also reveal a expected inefficiency. One might expect the simpler oracle to be faster, but the TTF was comparable to the full *ChainDelta* framework, and even slower in complex cases like B1. This is due to the spamming of the corpus. While the full *ChainDelta* incurs a direct cost from state tracing and analysis, the ablated variant suffers from a massive indirect overhead. The constant stream of false alarms spams the corpus in the directed patch fuzzer and can misguide the system into pursuing useless exploration paths based on benign inconsistencies. This shows that the direct overhead of our sanitizer is a worthwhile cost, as it prevents the greater performance degradation caused by an unmanageable number of false alarms.

6.5 Real-World Auditing

To answer RQ4, we investigated whether the exploits generated by *ChainDelta* for known patches could be used to find propagated vulnerabilities across EVM ecosystem. Our methodology focuses on the insight that many blockchain clients share standardized communication protocols (e.g., Ethereum JSON-RPC) and core execution logic.

Specifically, we selected target systems from leading bug bounty platforms (e.g., Immunefi), focusing on EVM-compatible chains. Then, we applied the generated exploits from collected security patches. As these exploits target standard RPC and API interfaces, they are inherently portable to other Ethereum compatible implementations. While a patch identifies the modified code, it fails to reveal the specific sequence of state transitions and RPC payloads required to trigger the bug. *ChainDelta* bridges this gap by automatically synthesizing these end-to-end exploits. We applied these exploits to all available node implementation by manually checking their documentation. In cases of slight interface divergence, minimal manual effort was required to align the API payload with the target's specific specification.

This process led to the discovery of four previously undisclosed vulnerabilities in four blockchain implementations, including Reth, BSC, Celo Network, and AntChain. These vulnerabilities were primarily denial-of-service vectors caused by resource exhaustion, triggered by insufficient access control on specific API payloads. Three of these vulnerabilities were confirmed by the respective development teams, and we were awarded a total of $2,000 in bug bounties for our responsible disclosure. The fourth vulnerability, a denial-of-service (DoS) vector via an RPC endpoint, was acknowledged by the developers but not classified as a security flaw, as they argued that DoS prevention is the responsibility of node operators.

This experiment underscores the practical value of *ChainDelta* for ecosystem-wide security. The widespread forking and code reuse in the blockchain space mean that a vulnerability in one project is often a strong indicator of similar flaws in others. By automatically generating a concrete exploit for a patched vulnerability, *ChainDelta* provides a clear picture of the attack. Security teams can then use this to proactively find and fix propagated vulnerabilities across the entire ecosystem, turning a single patch analysis into a much broader security improvement.

A case study. In addition to finding propagated vulnerabilities, our experiments also uncovered an incomplete patch in benchmark case B2. This case study provides an in-depth example of *ChainDelta* capability to rigorously verify security fixes, further addressing RQ4.

The original vulnerability, detailed in Figure 6, was a race condition. During a chain reorganization, transaction lookup operations could read inconsistent data while the underlying state was being modified. The corresponding fix, shown in a simplified form in Figure 6, was to introduce a read-write mutex. The reorganization process would acquire an exclusive write lock, while lookup functions, like `GetTransactionLookup`, were patched to acquire a read lock (`RLock` on Line 24), forcing them to wait until the modification was complete.

However, when we ran *ChainDelta* on this benchmark, it consistently triggered numerous panic errors in the patched version of the client, not the original vulnerable one. Our root cause analysis revealed that while the developers had correctly protected the primary `GetTransactionLookup` function, several other code paths also accessed the same sensitive data structures but did not acquire the new lock. The agent-driven exploration by *ChainDelta* was able to craft inputs that invoked these unprotected paths concurrently with a reorganization, triggering a new data race that the patch was supposed to prevent.

This case study highlights a critical challenge in manual security patching. That is, ensuring a fix covers all possible execution paths that can reach a vulnerable state. It demonstrates that automated exploit generation is uniquely positioned to validate such fixes. While a human code

```
--- a/core/blockchain.go
+++ b/core/blockchain.go
@@ func (bc *BlockChain) reorg(...) error {
      ...
-     // Reset the tx lookup cache...
-     bc.txLookupCache.Purge()
+     // Acquire the tx-lookup lock before mutation.
+     bc.txLookupLock.Lock()
      ...
      if err := indexesBatch.Write(); err != nil {
           log.Crit("Failed to delete useless indexes", "err", err)
      }
+     // Reset the tx lookup cache...
+     bc.txLookupCache.Purge()
+
+     // Release the tx-lookup lock after mutation.
+     bc.txLookupLock.Unlock()
      ...
}
--- a/core/blockchain_reader.go
+++ b/core/blockchain_reader.go
@@ func (bc *BlockChain) GetTransactionLookup(...) {
      ...
+    bc.txLookupLock.RLock()
+    defer bc.txLookupLock.RUnlock()
+
      // Short circuit if the txlookup already in the cache...
      if item, exist := bc.txLookupCache.Get(hash); exist {
           return item.lookup, item.transaction, nil
      ...
}
```

Fig. 6. Simplified patch from go-ethereum pull request #29343, which was intended to fix a race condition.

review might focus only on the intended changes, a tool like *ChainDelta* relentlessly explores the system's actual behavior, uncovering subtle and overlooked attack vectors. By finding this incomplete patch, *ChainDelta* proved its value not just in verifying known bugs, but in ensuring the correctness and security of the fixes themselves.

7 Related Work

Blockchain system security. Recent studies have been proposed for detecting blockchain system vulnerabilities. Most of these works focus on detecting specific vulnerabilities among blockchain modules, such as transaction pool denial-of-service [53, 66, 69], consensus bugs [56, 71], and RPC bugs [48, 52]. `DETER` [53] and `MPFuzz` [66] exploit the denial-of-service weakness of Ethereum clients' transaction pool by sending low-cost transactions that evict benign ones. `Tyr` [13], `LOKI` [56], and `Fluffy` [71] proposed to generate blockchain transactions and peer-to-peer messages to trigger consensus bugs in blockchain systems. `EtherDiffer` [48] detects blockchain RPC bugs through differential testing across diverse blockchain clients. It generates a set of RPC requests for these clients and checks for inconsistencies in their responses. These works primarily focus on detecting a set of pre-defined bugs or vulnerabilities and have limited capability to validate security patches for general vulnerabilities. The most closely related work is `BlockScope` [72], which leverages code similarity to detect propagated but unpatched vulnerabilities in forked blockchain projects. However, the effectiveness of code similarity metrics can be limited if a patch is complex or subtly integrated into the code.

Exploit generation. Automated exploit generation has been extensively studied for various software systems. Brumley et al. [10] first proposed the automated patch-based exploit generation problem against software patch distribution schemes, as well as the exploit generation technique based on pre-patch and post-patch analysis. The follow-up works studied the exploit generation for web applications [5, 6, 47], Android framework [55], and Linux kernels [12, 54, 68], etc.. Among these works, fuzzing has been a widely applied approach for exploit generation [14, 45, 54]. `SemFuzz` [73] and `VulScope` [14] leverage patch and vulnerability-related text, such as CVE reports, to guide fuzzers to explore vulnerable code statements and trigger vulnerabilities. `FUZE` [68] proposed a hybrid approach based on kernel fuzzing and symbolic execution to evaluate Linux system calls for kernel use-after-free vulnerability. `Magneto` [74] proposed an LLM-enhanced directed fuzzer to generate multi-step exploits for vulnerable dependent libraries based on static analysis of the call chains. The general-purpose automated exploit generation technique cannot be easily extended to blockchain systems, as specific stateful environments are required to trigger vulnerabilities.

Directed and Environment Fuzzing. Directed fuzzing guides execution toward specific targets, such as patched code. AFLGo [9] introduced distance-based energy scheduling to reach targets efficiently. More recently, PatchFuzz [63] integrated patch analysis with hybrid fuzzing to improve the exploration of patched code. In parallel, Program Environment Fuzzing [59] extended fuzzing to capture interactions with the operating system environment (e.g., files) for triggering environment-dependent bugs. While these general-purpose techniques are effective for standard software, they lack the domain-specific context required for blockchain systems.

8 Conclusion

In this paper, we introduced *ChainDelta*, a novel fuzzing agent framework designed to tackle the challenge of patch-based exploit generation for Ethereum. Automatically generating exploits for patched vulnerabilities is critical for mitigating 1-day threats, yet existing techniques fail in effectively addressing blockchain system complexity. *ChainDelta* overcomes these challenges through the integration of three core modules: a directed patch fuzzer that navigates complex code paths; an agent-based environment fuzzer that automates the intricate blockchain environment setup required to trigger bugs; and a state-aware sanitizer that accurately filters out benign behaviors to eliminate false positives.

Our evaluation demonstrates the effectiveness of *ChainDelta*. On a benchmark of real-world vulnerabilities, *ChainDelta* outperformed existing state-of-the-art fuzzers. Ablation studies confirmed that each module is crucial to the system's success, and our case study showed *ChainDelta*'s ability to uncover subtle flaws, such as incomplete patches. We demonstrated the real-world impact of our approach by using the exploits generated by *ChainDelta* to discover four previously unknown vulnerabilities in other blockchain projects, for which we received $2,000 in bug bounties.

9 Data Availability

This research complies with the open science policy. We have made the artifact of *ChainDelta* available online to facilitate future research at: https://github.com/MingxiYe/ChainDelta-Artifact. More details can be found in the README file.

Acknowledgments

This work was supported in part by the National Natural Science Foundation of China (No. 624B2139, No. 62572497, No. 62302538), NSFC-RGC Collaborative Research (No. 62461160332), the General Program of the Natural Science Foundation of Guangdong Province, China (No. 2025A1515010279), and Guangdong Zhujiang Talent Program (No. 2023QN10X561).

References

[1] [n. d.]. Could Wrapped Tokens Like WETH Be (forced) Insolvent? https://pwning.mirror.xyz/RFNTSouIIlHVNmTNDThUVb1obIeN5c1LAiQuN9Ve-ok

[2] 2023. Pull Request #1081: Introduce precompile v2 utils & minor refactoring. https://github.com/AstarNetwork/Astar/pull/1081. Accessed: 2025-09-04.

[3] 2025. bnb-chain/bsc. https://github.com/bnb-chain/bsc original-date: 2020-05-21T06:44:14Z.

[4] 2025. ethereum/go-ethereum. https://github.com/ethereum/go-ethereum original-date: 2013-12-26T13:05:46Z.

[5] Abeer Alhuzali, Birhanu Eshete, Rigel Gjomemo, and VN Venkatakrishnan. 2016. Chainsaw: Chained automated workflow-based exploit generation. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 641–652. doi:10.1145/2976749.2978380

[6] Abeer Alhuzali, Rigel Gjomemo, Birhanu Eshete, and VN Venkatakrishnan. 2018. {NAVEX}: Precise and scalable exploit generation for dynamic web applications. In *27th USENIX Security Symposium (USENIX Security 18)*. 377–392.

[7] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. 2011. AEG: Automatic Exploit Generation. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*. The Internet Society. https://www.ndss-symposium.org/ndss2011/aeg-automatic-exploit-generation

[8] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J Schwartz, Maverick Woo, and David Brumley. 2014. Automatic exploit generation. *Commun. ACM* 57, 2 (2014), 74–84. doi:10.1145/2560217.2560219

[9] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 2329–2344. doi:10.1145/3133956.3134020

[10] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. 2008. Automatic patch-based exploit generation is possible: Techniques and implications. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*. IEEE, 143–157.

[11] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. 2020. Metamorphic testing: a new approach for generating next test cases. *arXiv preprint arXiv:2002.12543* (2020).

[12] Weiteng Chen, Xiaochen Zou, Guoren Li, and Zhiyun Qian. 2020. {KOOBE}: Towards facilitating exploit generation of kernel {Out-Of-Bounds} write vulnerabilities. In *29th USENIX security symposium (USENIX security 20)*. 1093–1110.

[13] Yuanliang Chen, Fuchen Ma, Yuanhang Zhou, Yu Jiang, Ting Chen, and Jiaguang Sun. 2023. Tyr: Finding consensus failure bugs in blockchain system with behaviour divergent model. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2517–2532.

[14] Jiarun Dai, Yuan Zhang, Hailong Xu, Haiming Lyu, Zicheng Wu, Xinyu Xing, and Min Yang. 2021. Facilitating vulnerability assessment through poc migration. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 3300–3317. doi:10.1145/3460120.3484594

[15] Martin Eberlein, Yannic Noller, Thomas Vogel, and Lars Grunske. 2020. Evolutionary grammar-based fuzzing. In *International Symposium on Search Based Software Engineering*. Springer, 105–120.

[16] Ethereum. 2024. Hive: Ethereum end-to-end test harness. https://github.com/ethereum/hive.

[17] Ethereum Foundation. 2024. Pull Request #29238: go-ethereum. https://github.com/ethereum/go-ethereum/pull/29238. Accessed: 2025-09-04.

[18] Ethereum Foundation. 2024. Pull Request #29343: go-ethereum. https://github.com/ethereum/go-ethereum/pull/29343. Accessed: 2025-09-04.

[19] Ethereum Foundation. 2024. Pull Request #29623: go-ethereum. https://github.com/ethereum/go-ethereum/pull/29623. Accessed: 2025-09-04.

[20] Ethereum Foundation. 2024. Pull Request #29644: go-ethereum. https://github.com/ethereum/go-ethereum/pull/29644. Accessed: 2025-09-04.

[21] Ethereum Foundation. 2024. Pull Request #29703: go-ethereum. https://github.com/ethereum/go-ethereum/pull/29703. Accessed: 2025-09-04.

[22] Ethereum Foundation. 2024. Pull Request #29738: go-ethereum. https://github.com/ethereum/go-ethereum/pull/29738. Accessed: 2025-09-04.

[23] Ethereum Foundation. 2024. Pull Request #29811: go-ethereum. https://github.com/ethereum/go-ethereum/pull/29811. Accessed: 2025-09-04.

[24] Ethereum Foundation. 2024. Pull Request #29824: go-ethereum. https://github.com/ethereum/go-ethereum/pull/29824. Accessed: 2025-09-04.

[25] Ethereum Foundation. 2024. Pull Request #30264: go-ethereum. https://github.com/ethereum/go-ethereum/pull/30264. Accessed: 2026-02-03.

[26] Ethereum Foundation. 2024. Pull Request #30318: go-ethereum. https://github.com/ethereum/go-ethereum/pull/30318. Accessed: 2025-09-04.

[27] Ethereum Foundation. 2024. Pull Request #30474: go-ethereum. https://github.com/ethereum/go-ethereum/pull/30474. Accessed: 2025-09-04.
[28] Ethereum Foundation. 2024. Pull Request #30729: go-ethereum. https://github.com/ethereum/go-ethereum/pull/30729. Accessed: 2026-02-03.
[29] Ethereum Foundation. 2025. Pull Request #30981: go-ethereum. https://github.com/ethereum/go-ethereum/pull/30981. Accessed: 2026-02-03.
[30] Ethereum Foundation. 2025. Pull Request #31049: go-ethereum. https://github.com/ethereum/go-ethereum/pull/31049. Accessed: 2026-02-03.
[31] Ethereum Foundation. 2025. Pull Request #31246: go-ethereum. https://github.com/ethereum/go-ethereum/pull/31246. Accessed: 2026-02-03.
[32] Ethereum Foundation. 2025. Pull Request #31336: go-ethereum. https://github.com/ethereum/go-ethereum/pull/31336. Accessed: 2026-02-03.
[33] Ethereum Foundation. 2025. Pull Request #31752: go-ethereum. https://github.com/ethereum/go-ethereum/pull/31752. Accessed: 2026-02-03.
[34] Ethereum Foundation. 2025. Pull Request #31934: go-ethereum. https://github.com/ethereum/go-ethereum/pull/31934. Accessed: 2026-02-03.
[35] Ethereum Foundation. 2025. Pull Request #32183: go-ethereum. https://github.com/ethereum/go-ethereum/pull/32183. Accessed: 2026-02-03.
[36] Ethereum Foundation. 2025. Pull Request #32461: go-ethereum. https://github.com/ethereum/go-ethereum/pull/32461. Accessed: 2026-02-03.
[37] Ethereum Foundation. 2025. Pull Request #32594: go-ethereum. https://github.com/ethereum/go-ethereum/pull/32594. Accessed: 2026-02-03.
[38] Ethereum Foundation. 2025. Pull Request #32794: go-ethereum. https://github.com/ethereum/go-ethereum/pull/32794. Accessed: 2026-02-03.
[39] Ethereum Foundation. 2025. Pull Request #32889: go-ethereum. https://github.com/ethereum/go-ethereum/pull/32889. Accessed: 2026-02-03.
[40] Ethereum Foundation. 2025. Pull Request #32894: go-ethereum. https://github.com/ethereum/go-ethereum/pull/32894. Accessed: 2026-02-03.
[41] Ethereum-Optimism. [n. d.]. GitHub - ethereum-optimism/op-geth. https://github.com/ethereum-optimism/op-geth
[42] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association.
[43] Andrea Fioraldi, Dominik Christian Maier, Dongjia Zhang, and Davide Balzarotti. 2022. Libafl: A framework to build modular and reusable fuzzers. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 1051–1065. doi:10.1145/3548606.3560602
[44] Google AI for Developers. 2025. Gemini models. https://ai.google.dev/gemini-api/docs/models. Accessed: 9-September-2025.
[45] Sean Heelan, Tom Melham, and Daniel Kroening. 2019. Gollum: Modular and greybox exploit generation for heap overflows in interpreters. In *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*. 1689–1706. doi:10.1145/3319535.3354224
[46] Shih-Kun Huang, Min-Hsiang Huang, Po-Yen Huang, Han-Lin Lu, and Chung-Wei Lai. 2014. Software crash analysis for automatic exploit generation on binary programs. *IEEE Transactions on Reliability* 63, 1 (2014), 270–289. doi:10.1109/TR.2014.2299198
[47] Shih-Kun Huang, Han-Lin Lu, Wai-Meng Leong, and Huan Liu. 2013. Craxweb: Automatic web application testing and attack generation. In *2013 IEEE 7th International Conference on Software Security and Reliability*. IEEE, 208–217. doi:10.1109/SERE.2013.26
[48] Shinhae Kim and Sungjae Hwang. 2023. Etherdiffer: Differential testing on rpc services of ethereum nodes. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1333–1344. doi:10.1145/3611643.3616251
[49] Johannes Krupp and Christian Rossow. 2018. {teEther}: Gnawing at ethereum to automatically exploit smart contracts. In *27th USENIX security symposium (USENIX Security 18)*. 1317–1333.
[50] LangChain. 2025. LangChain. https://www.langchain.com/. Accessed: 9-September-2025.
[51] Jun Li, Bodong Zhao, and Chao Zhang. 2018. Fuzzing: a survey. *Cybersecurity* 1, 1 (2018), 6. doi:10.1186/s42400-018-0002-y
[52] Kai Li, Jiaqi Chen, Xianghong Liu, Yuzhe Richard Tang, XiaoFeng Wang, and Xiapu Luo. 2021. As Strong As Its Weakest Link: How to Break Blockchain DApps at RPC Service. In *NDSS*.
[53] Kai Li, Yibo Wang, and Yuzhe Tang. 2021. Deter: Denial of ethereum txpool services. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 1645–1667. doi:10.1145/3460120.3485369

[54] Zhenpeng Lin, Yueqi Chen, Yuhang Wu, Dongliang Mu, Chensheng Yu, Xinyu Xing, and Kang Li. 2022. Grebe: Unveiling exploitation potential for linux kernel bugs. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2078–2095. doi:10.1109/SP46214.2022.9833683

[55] Lannan Luo, Qiang Zeng, Chen Cao, Kai Chen, Jian Liu, Limin Liu, Neng Gao, Min Yang, Xinyu Xing, and Peng Liu. 2019. Tainting-assisted and context-migrated symbolic execution of Android framework for vulnerability discovery and exploit generation. *IEEE Transactions on Mobile Computing* 19, 12 (2019), 2946–2964. doi:10.1109/TMC.2019.2936561

[56] Fuchen Ma, Yuanliang Chen, Meng Ren, Yuanhang Zhou, Yu Jiang, Ting Chen, Huizhong Li, and Jiaguang Sun. 2023. LOKI: State-Aware Fuzzing Framework for the Implementation of Blockchain Consensus Protocols.. In *NDSS*.

[57] Fuchen Ma, Yuanliang Chen, Yuanhang Zhou, Jingxuan Sun, Zhuo Su, Yu Jiang, Jiaguang Sun, and Huizhong Li. 2023. Phoenix: Detect and locate resilience issues in blockchain via context-sensitive chaos. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 1182–1196. doi:10.1145/3576915.3623071

[58] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[59] Ruijie Meng, Gregory J Duck, and Abhik Roychoudhury. 2024. Program environment fuzzing. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 720–734. doi:10.1145/3658644.3690229

[60] Michał Zalewski. 2017. Technical "Whitepaper" for AFL. https://lcamtuf.coredump.cx/afl/technical_details.txt. Accessed: 2025-09-04.

[61] Optimism. 2025. Pull Request #543: op-geth. https://github.com/ethereum-optimism/op-geth/pull/543. Accessed: 2026-02-03.

[62] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX Association, Boston, MA, 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[63] Letian Sha, Luheng Zhang, Yifei Huang, Yan Lin, Fu Xiao, and Jiaye Pan. 2025. PatchFuzz: an Efficient Way to Incorporate Patching with Hybrid Fuzzing. *IEEE Transactions on Dependable and Secure Computing* (2025). doi:10.1109/TDSC.2025.3600181

[64] 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. doi:10.1145/3597926.3598059

[65] 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 on ACM SIGSAC Conference on Computer and Communications Security*. 2577–2590. doi:10.1145/3658644.3690256

[66] Yibo Wang, Yuzhe Tang, Kai Li, Wanning Ding, and Zhihua Yang. 2024. Understanding Ethereum Mempool Security under Asymmetric {DoS} by Symbolized Stateful Fuzzing. In *33rd USENIX Security Symposium (USENIX Security 24)*. 4747–4764.

[67] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.

[68] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. 2018. {FUZE}: Towards facilitating exploit generation for kernel {Use-After-Free} vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)*. 781–797.

[69] Aviv Yaish, Kaihua Qin, Liyi Zhou, Aviv Zohar, and Arthur Gervais. 2024. Speculative {Denial-of-Service} Attacks In Ethereum. In *33rd USENIX security symposium (USENIX Security 24)*. 3531–3548.

[70] Songtao Yang, Yubo He, Kaixiang Chen, Zheyu Ma, Xiapu Luo, Yong Xie, Jianjun Chen, and Chao Zhang. 2023. 1dfuzz: Reproduce 1-day vulnerabilities with directed differential fuzzing. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 867–879. doi:10.1145/3597926.3598102

[71] Youngseok Yang, Taesoo Kim, and Byung-Gon Chun. 2021. Finding consensus bugs in ethereum via multi-transaction differential fuzzing. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. 349–365.

[72] Xiao Yi, Yuzhou Fang, Daoyuan Wu, and Lingxiao Jiang. 2022. BlockScope: Detecting and investigating propagated vulnerabilities in forked blockchain projects. *arXiv preprint arXiv:2208.00205* (2022).

[73] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. 2017. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 2139–2154. doi:10.1145/3133956.3134085

[74] Zhuotong Zhou, Yongzhuo Yang, Susheng Wu, Yiheng Huang, Bihuan Chen, and Xin Peng. 2024. Magneto: A Step-Wise Approach to Exploit Vulnerabilities in Dependent Libraries via LLM-Empowered Directed Fuzzing. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1633–1644. doi:10.1145/3691620.3695531

Received 2026-02-24; accepted 2026-03-24