BLOCKSEC

Security Testing

Report for MegaETH: MegaEVM, Stateless Validator & SALT

Date: December 15, 2025 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	MegaETH
Target	MegaEVM, Stateless Validator & SALT

Version History

Version	Date	Description
1.0	December 15, 2025	First release

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2025-12-16 14:11:53+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 Security Testing Overview

Information	Description
Type	Security Testing
Approach	Differential testing, fuzzing and manual verification

MegaETH [1] is an EVM-compatible blockchain designed to bring Web2-level real-time performance to the crypto world. Its goal is to push performance to the limits of modern hardware, narrowing the gap between blockchains and traditional cloud computing servers. MegaETH offers several distinguishing features, including high transaction throughput, abundant compute capacity, and, most notably, millisecond-level response times even under heavy load. This enables developers to build and compose highly demanding applications without practical performance constraints.

As an EVM-compatible Layer 2 (L2) solution, MegaETH has undergone extensive customization to maximize performance. These changes include the integration of a customized EVM implementation and the replacement of the state management component, design choices intended to improve operational efficiency and align the system with the platform's specific functional requirements.

The security testing was conducted over a six-week period, from October 11, 2025 to November 28, 2025, and focused on:

- **MegaETH EVM** [2] (hereafter `MegaEVM`), a specialized Ethereum Virtual Machine (EVM) implementation tailored to MegaETH's specifications. It is built on top of `revm` and `op-revm` with MegaETH-specific modifications and optimizations to support high-performance blockchain execution.
- **Small Authentication Large Trie** [3] (hereafter `SALT`), the core state management component of the MegaETH blockchain. It is a memory-efficient state trie data structure designed to replace the Merkle Patricia Trie (MPT) in blockchain systems. `SALT` provides authenticated key–value storage using IPA (Inner Product Argument) and Pedersen commitments. Unlike traditional MPT, which requires frequent disk I/O during state root updates, `SALT` is designed to keep all intermediate commitments in memory and eliminate random disk I/O.
- **Stateless Validator** [4], a Rust implementation of a stateless blockchain validator specifically designed for MegaEVM, Stateless Validator & SALT. This validator enables efficient block verification using cryptographic witness data from `SALT` instead of maintaining full blockchain state. The stateless approach eliminates the need for validators to run on high-end hardware comparable to sequencer nodes, making it practical to run validator nodes

[1] https://www.megaeth.com/research

[2] https://github.com/megaeth-labs/mega-evm

[3] https://github.com/megaeth-labs/salt

[4] https://github.com/megaeth-labs/stateless-validator

at scale.

The concrete scope, goals, and detailed description of this security testing, including the methodology and findings, will be presented in the chapters on `MegaEVM` and `Stateless Validator` (Chapter 2) and `SALT` (Chapter 3), respectively.

1.2 Disclaimer

The scope of this security testing is limited to the code mentioned in Section 1.1. This penetration test does not guarantee the discovery of all security issues in the target system, the assessment results do not guarantee the absence of any other security vulnerabilities. This report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. Because no single penetration test can be completely comprehensive, we always recommend conducting independent penetration tests and implementing a public bug bounty program to ensure the security of the network infrastructure.

1.3 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [5] and Common Weakness Enumeration [6]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

[5] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[6] https://cwe.mitre.org/

Furthermore, the status of a discovered item will fall into one of the following five categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Partially Fixed** The item has been confirmed and partially fixed by the client.
- **Fixed** The item has been confirmed and fixed by the client.

Chapter 2 Security Testing for MegaEVM and Stateless Validator

In this chapter, we first describe the targets, scope, and high-level methodology for the security testing of MegaEVM and the Stateless Validator in Section 2.1. Section 2.2 then presents a comprehensive differential testing framework designed to validate MegaEVM against the reference implementation op-revm (Optimism's Rust-based EVM). Since MegaEVM is based on Optimism's Isthmus hard fork, we use op-revm as the reference and replay blocks starting from the Isthmus activation as historical data. This testing ensures that, while MegaEVM optimizes performance and resource management, it still adheres strictly to the EVM semantic invariants required for security and compatibility.

Next, in Section 2.3, we describe our fuzzing methodology, which combines IR-based transaction fuzzing with low-level bytecode generation to explore edge cases and uncover potential security vulnerabilities, such as denial-of-service (DoS) conditions and execution crashes. Section 2.4 further evaluates the correctness of the Stateless Validator by replaying historical transactions through the Mega-Reth sequencer and verifying that the validator can reconstruct the canonical state using SALT witnesses.

2.1 Targets, Scope, and Methodology

2.1.1 MegaEVM Overview

The Ethereum Virtual Machine (EVM) serves as the execution engine for Ethereum. As blockchain scalability becomes paramount, high-performance EVM implementations are emerging to meet the demand for higher throughput and lower latency. MegaEVM is one such specialized implementation designed for the MegaETH network. It is built upon the foundations of op-revm (Optimism's EVM) but introduces significant architectural changes to support a high-performance, resource-efficient execution environment.

MegaEVM distinguishes itself from standard EVM implementations (i.e., Optimism's Isthmus fork) through its adoption of the Mini-Rex hardfork specification [1]. While the standard EVM relies on a single-dimensional gas metric to limit block execution and resource usage, Mini-Rex introduces a paradigm shift to support the MegaETH network's performance goals. Key differences include:

- **Multi-Dimensional Resource Accounting**. Unlike the single gas limit in Ethereum/Optimism Mainnet, MegaEVM decouples resource constraints into three independent dimensions: Compute Gas (for execution steps), Data Size (for bandwidth/storage), and Key-Value (KV) Updates (for state access). This prevents any single resource from becoming a bottleneck or security vector.
- **Dual Gas Model**. Costs are separated into *Compute Gas* and *Storage Gas*. Storage operations incur additional storage gas costs that scale dynamically based on state saturation

[1] https://github.com/megaeth-labs/mega-evm/blob/main/specs/MiniRex.md

(SALT bucket scaling), independent of computational costs. For example, an SSTORE operation transitioning a slot from zero to non-zero incurs a base cost of 20,000 gas in standard EVM. In MegaEVM, this triggers an additional 2,000,000 storage gas charge (multiplied by a bucket scaling factor), representing a 100x increase to accurately price state expansion. Similarly, LOG operations and calldata incur approximately 10x higher costs compared to standard execution to reflect their storage burden.

- **Extended Limits & Restrictions**. Mini-Rex significantly increases the maximum contract size to 512 KB (up from the standard 24 KB) to support complex applications. Conversely, it strictly prohibits the SELFDESTRUCT opcode to ensure state immutability and simplify state management.
- **Gas Detention Mechanism**. To handle volatile data access safely, MegaEVM implements a gas detention mechanism that temporarily withholds gas during specific operations, ensuring resource availability without halting execution unnecessarily.

Given these substantial deviations from the reference EVM specifications, it is critical to ensure the correctness and equivalence of MegaEVM in areas where behavior is intended to match. Differential testing [2] is a validation technique widely used in compiler and VM testing. It involves executing the same inputs (in this context, transactions and blocks) across multiple implementations of the same specification and comparing their outputs (state transitions, execution traces, and logs).

Scope Here we conducted one round of differential testing against the projects at the specific commit versions shown in the table below.

# of Round	Project	Commit Hash
1	mega-evm [3]	32ef93f61e6864f8053c6332871ab34e82f83056
	op-revm [4]	7d0545dbc8087b524ef34e707a944c7898705d10

Table 2.1: Scope of differential testing for the MegaEVM

2.1.2 Stateless Validator Overview

Stateless Validator is a validator for MegaETH that verifies blocks using SALT witnesses instead of maintaining full state. Witnesses supply authenticated account/storage proofs while contract bytecode is fetched on demand, yielding partial statelessness and small witness sizes. The validator is designed for horizontal scalability: independent workers claim blocks, execute them with MegaEVM, and advance the canonical chain if the computed state root matches the witness. This multi-engine approach complements the high-performance sequencer (Mega-Reth) and provides an independently implemented state transition function to reduce single-client risk. We aim to identify correctness issues in the Stateless Validator by replaying historical transactions and performing differential analysis between Stateless Validator and Mega-Reth.

Figure 2.2 illustrates the end-to-end workflow: Mega-Reth (sequencer) replays historical Optimism Mainnet transactions, the Witness Generator produces SALT witnesses from the re-

[2] https://en.wikipedia.org/wiki/Differential_testing

[3] https://github.com/megaeth-labs/mega-evm

[4] https://github.com/bluealloy/revm

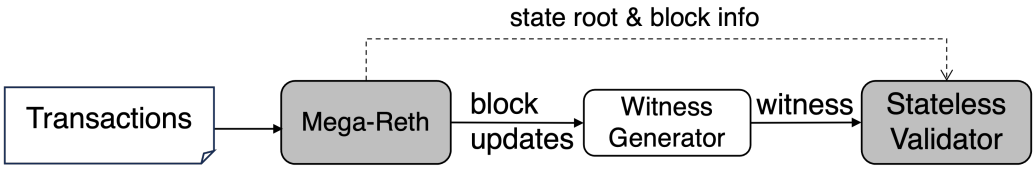

Figure 2.1: Workflow of the Stateless Validator.

sulting state, and the `Stateless Validator` consumes those witnesses to independently verify blocks and state roots.

The detailed workflow for the bootstrap of the `Stateless Validator` is as follows:

1. **Sequencer Execution**: `Mega-Reth` acts as the sequencer, executing transactions and updating the state. It produces blocks and the canonical state root.
2. **Witness Generation**: The Witness Generator asynchronously reads the new state from the sequencer and generates `SALT` witnesses. These witnesses contain the minimal necessary cryptographic proofs (account states, storage slots) required to execute the block without full state access.
3. **Stateless Validation**: The `Stateless Validator` receives the block (containing transactions) from the sequencer and the corresponding witness from the Witness Generator. It may also fetch contract bytecodes on-demand if not present in the witness.
4. **Verification**: The `Stateless Validator` re-executes the transactions using the witness data. It computes a new state root and compares it against the one provided by the sequencer. A match confirms the validity of the block and the state transition.

Scope Here we conducted two rounds of differential testing against the projects at the specific commit versions shown in the table below. Due to version constraints, the first round did not use `chain-ops` for environment setup. In the second round, we used `chain-ops` to build and start the environment, so `chain-ops` was only involved in the second round.

# of Round	Project	Commit Hash
1	`mega-reth` [5]	`78f00b880b4fd5c518416512ae445c4dec13c0a2`
	`stateless-validator` [6]	`f239b511b0305dae792276014e344f8b0fa9a824`
2	`mega-reth` [5]	`23cb905f49628506e44d658b673a0ec2a1c63f5a`
	`stateless-validator` [6]	`cb02b0d25ced6c7540b60445b611c7461dae4efb`
	`chain-ops` [7]	`c22af78db3e8e49271b0180f5fac7620ae426dff`

Table 2.2: Scope of differential testing for the `Stateless Validator`

2.1.3 Methodology and Goals

Our evaluation strategy employs three complementary tracks: *differential testing of `MegaEVM`* against the reference implementation, *fuzzing-based testing of `MegaEVM`* to uncover security vulnerabilities, and *replay-based validation of the `Stateless Validator`* to ensure the correctness of its verification functionality.

[5] https://github.com/megaeth-labs/mega-reth

[6] https://github.com/megaeth-labs/stateless-validator

[7] https://github.com/megaeth-labs/chain-ops

2.2 Differential Testing of MegaEVM

Our differential testing system is designed to automate the detection of execution discrepancies between the system under test (i.e., MegaEVM) and a reference implementation (i.e., op-revm). The system consists of three core components and they are the *Input Generator*, the *Dual-execution Dispatcher*, and the *Post-state Analyzer*.

2.2.1 System Design

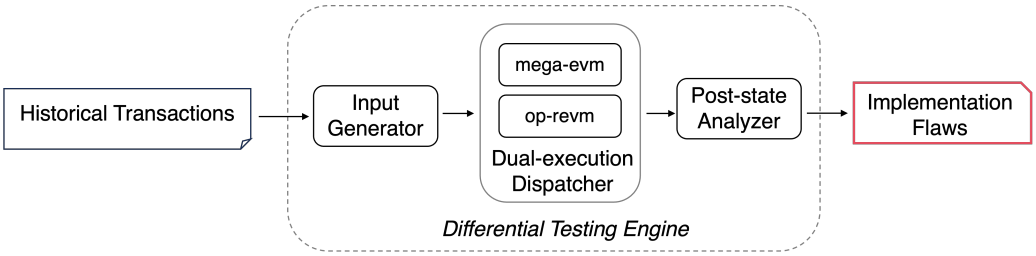

Figure 2.2: Differential testing framework.

Figure 2.2 illustrates our differential testing framework. The differential testing engine takes historical transactions as input and reports any implementation flaws it identifies:

1. **Input Generator**. This module fetches historical transactions and blocks from Optimism Mainnet and feeds them into the engine as test inputs.
2. **Dual-execution Dispatcher**. This module aims to dispatch the inputs (i.e., historical transactions) to the reference node and the target node. Furthermore, it also aims to collect the execution traces from both nodes.
   - *Reference node*. The op-revm implementation serves as the ground truth for standard Optimism EVM behavior. We instrument it to emit fine-grained execution traces, including program counter (PC) steps, stack contents, memory states, and storage modifications.
   - *Target node*. The wrapped mega-evm binary running the MegaEVM implementation. It is configured to output execution traces that are compatible with those of the reference node.
3. **Post-state analyzer**. This module contains the differential oracle, which is the core comparison engine. After execution, the oracle aligns the traces from both backends and performs a step-by-step comparison. In particular, it checks:
   - *State consistency*: differences in account balances, nonces, and storage slot values.
   - *Execution flow*: divergences in the PC sequence, indicating disagreements in control-flow logic.
   - *Opcode behavior*: discrepancies in stack outputs or memory mutations produced by individual opcodes.

Handling Architectural Differences A significant challenge in differentially testing MegaEVM is its intentional deviation from standard gas accounting. Since MegaEVM uses a multi-dimensional gas model, naive comparison of gas_remaining would yield false positives. Our methodology addresses this by:

- Excluding Gas Comparison: We explicitly filter out gas-related fields from the differential oracle unless testing for specific compute-gas equivalence.
- Gas Limit Adjustment: For executions on `MegaEVM`, we specifically increase the transaction `gas_limit` by a factor of 100 to accommodate the highest gas-cost opcodes (i.e., the 100x cost for `SSTORE`). The reference `op-revm` executions retain the original gas limits. This asymmetry ensures that `MegaEVM` transactions can complete execution despite the stricter pricing model, allowing us to compare the final state against the reference without false positives from early Out-of-Gas errors.

Instrumentation To facilitate deep inspection, we utilize a modified evm runner that supports selective debugging. We apply patches to the reference `op-revm` implementation to enable an efficient tracing mode (`SelectiveDebug`). This mode captures critical state changes (`CALL`s, `STORE`s, log emissions) without the massive overhead of full instruction-level tracing, allowing for high-throughput testing of millions of blocks.

2.2.2 Evaluation and Findings

To assess the robustness and correctness of `MegaEVM`, we conducted an extensive evaluation using our differential testing framework.

Evaluation Setup All experiments were conducted on a workstation equipped with an Intel Core i9-13900K CPU and 128GB of RAM, running Ubuntu 24.04 LTS.

Dataset Our evaluation dataset comprises 1,000,000 blocks (containing 24,190,506 transactions) replayed from the Optimism Mainnet, starting from block height 135,589,412 (post-Isthmus hardfork). This dataset ensures coverage of modern EVM features and complex DeFi interactions.

2.2.2.1 Overall Results

We analyzed the execution of the 24.19 million replayed transactions. Our differential testing framework identified 7,411,419 transactions (approx. 30.6%) that exhibited discrepancies primarily attributed to the specific design choices of `MegaEVM` (e.g., MiniRex gas model). The discrepancies were categorized by the type of divergence observed by the Oracle:

- Differential PC: 7,232,753 (Control flow divergence)
- Differential Record Counts: 97,181 (Difference in number of call records)
- Differential Storage: 72,086 (Storage value mismatches)
- Differential EVM Output: 8,951 (Return data mismatches)
- Differential Rejection Status: 448 (Transaction rejection status mismatch)

Impact of Volatile Gas Design `MegaEVM`'s Volatile Data Access Control (triggering *Compute Gas* limits when accessing frequently contended data such as block environment variables and beneficiary) is another significant source of divergence. Since accessing volatile data triggers gas limiting mechanisms that do not exist in standard EVM, they can lead to transaction failures or altered execution paths.

To isolate this impact, we conducted a comparative test by toggling the volatile gas design features on the same 1,000,000 block dataset. Our analysis revealed:

- 7,411,419 transactions were affected by the general Gas Model design (as mentioned above).
- 1,204,888 transactions were specifically affected by the Volatile Data Gas Limit design.
- 200,581 transactions were affected by both designs simultaneously.

This demonstrates that while the general pricing model causes the majority of discrepancies, the volatile data restrictions also play a substantial role in execution divergence.

2.2.2.2 Root Cause Analysis

We further classified these discrepancies based on the root cause, specifically relating to the Dual Gas Model and resource accounting. We will illustrate the details in the following sections.

Category	Count	Percentage	Description
Fixed Call Gas	7,063,170	92.6%	Cross contract invocations using hardcoded gas limits (e.g., `callgas: 2300`) are insufficient for storage costs and other high-cost opcodes.
Rated Gas	142,127	4.7%	Gas calculated dynamically (e.g., code-gasleft() - overhead) can be insufficient for a specific call frame execution.
Gas Tainted	114,744	1.55%	The result of the `GAS` opcode influenced the data flow or control flow, causing execution path divergence (e.g., a cross contract invocation can fail in `MegaEVM` but succeed in `op-revm`).
Sufficient Gas	51,234	0.69%	Transactions that are successfully completed in `MegaEVM` due to the 100x gas limit increase, whereas they might have failed or behaved differently in `op-revm`.
Self Destruct	31,350	0.42%	Differences arising from the disabled `SELFDESTRUCT` opcode in `MegaEVM`.
Sufficient Gas but Failed	4,978	0.07%	Call frames had sufficient gas to execute (unlike in `op-revm`), but the execution still failed due to other constraints or logic.
Bomb Opcode Revert	3,773	0.05%	Reverts triggered by excessive use of "bomb" opcodes (e.g., `LOG`) which exhausted the gas limit despite the increase.
Balance Tainted	48	< 0.001%	Divergence caused by the initial sender balance difference (due to different gas limits), where the balance value influenced control or data flow.

In total, we have **nine** notes.

- Note: 9

The details are given below by examining specific transactions that illustrate the primary causes of divergence.

ID	Severity	Description	Category	Status
2.2-1	-	Fixed Gas Call Failure	Note	-
2.2-2	-	Rated Gas	Note	-
2.2-3	-	Gas Tainted	Note	-
2.2-4	-	Sufficient Gas	Note	-
2.2-5	-	Self Destruct	Note	-
2.2-6	-	Sufficient Gas still Failed	Note	-
2.2-7	-	Bomb Opcode	Note	-
2.2-8	-	Storage Tainted by Sender Balance	Note	-
2.2-9	-	Volatile Data Gas Limit	Note	-

2.2.2.3 Fixed Gas Call Failure

Description Cross contract invocations using hardcoded gas limits (e.g., `callgas: 2300`) are insufficient for storage costs and other high-cost opcodes in `MegaEVM`. This occurs because the `MegaEVM` imposes significantly higher storage gas costs, causing transactions that would succeed in standard EVM to fail when using fixed gas limits for cross-contract calls.

The following case demonstrates the transaction out-of-gas caused by the *Fixed Call Gas*, using Optimism Mainnet transaction [8]. Specifically,, the contract `ICLFactory` attempts to fetch a swap fee using a static call with a hardcoded gas limit of 200,000.

```
function getSwapFee(address pool) external view override returns (uint24) {
  if (swapFeeModule != address(0)) {
    // Fixed gas limit of 200,000 passed to static call
    (bool success, bytes memory data) = swapFeeModule.excessivelySafeStaticCall(
      200_000, 32, abi.encodeWithSelector(IFeeModule.getFee.selector, pool)
    );
    if (success) {
      uint24 fee = abi.decode(data, (uint24));
      if (fee <= 100_000) {
        return fee;
      }
    }
  }
  // Fallback logic if call fails
  return tickSpacingToFee[CLPool(pool).tickSpacing()];
}
```

Listing 2.1: ICLFactory

For `op-revm`, the 200,000 gas is sufficient for the `getFee` execution. The `success` returns `true`, and the custom fee is returned. However, in `MegaEVM`, the `getFee` logic involves storage reads (`SSTORE`/`SLOAD` equivalents) which are priced significantly higher in the Dual Gas Model (*Storage Gas*). The 200,000 gas limit is exhausted rapidly by the storage gas component. The cross contract invocation fails (`success` becomes `false`). This highlights the challenge of *Fixed Gas* patterns in a storage-gas-heavy environment: existing contracts with hardcoded limits may fail to execute read-heavy logic even if the transaction's total gas limit is raised (as we did in testing), because the internal sub-call limit remains fixed.

[8] 0x25b88d0dc42b357d4cad0d4c96a5936587ae4d0b4c79feacfb97b8a798ac661b

2.2.2.4 Rated Gas

Description Gas calculated dynamically (e.g., `gasleft() - overhead`) can be insufficient for a specific call frame execution in `MegaEVM`. This divergence arises from differences in gas allocation mechanisms between Mini-Rex (98/100 of remaining gas) and Optimism (63/64 of remaining gas), particularly in deep call stacks where the cumulative effect becomes significant.

The following case demonstrates the transaction out-of-gas caused by the *Rated Gas*, using Optimism Mainnet transaction [9].

```
 1 STATICCALL depth=4 →Proxy(0xd147) . 0x24e6be0e
 2 STATICCALL depth=4 →Proxy(0xb32c) . getUserReserveData
 3 STATICCALL depth=4 →Proxy(0xb32c) . getUserReserveData
 4    DELEGATECALL depth=5 →Impl(0xbfefca...) . getUserReserveData
 5
 6 STATICCALL depth=4 →Proxy(0xb32c) . getReserveConfigurationData
 7 STATICCALL depth=4 →Proxy(0xd147) . tokenConvert
 8 STATICCALL depth=4 →USDC . balanceOf
 9    DELEGATECALL depth=5 →FiatTokenV2_2 . balanceOf
10
11 STATICCALL depth=4 →WETH . balanceOf
12 STATICCALL depth=4 →Proxy(0xd147) . 0xe47670ec
13    DELEGATECALL depth=5 →Impl(0x8e68b4...) . 0xe47670ec
14       STATICCALL depth=6 →WETH . decimals
15       STATICCALL depth=6 →USDC . decimals
16          DELEGATECALL depth=7 →FiatTokenV2_2 . decimals
17
18 STATICCALL depth=6 →Aggregator(0x13e3) . latestAnswer
19 STATICCALL depth=7 →Aggregator(0x02f5) . latestAnswer
20 STATICCALL depth=6 →Aggregator(0x16a9) . latestAnswer
```

Listing 2.2: Invocation Trace

This transaction involves a deep call stack, as illustrated in the call depth trace above. According to the Mini-Rex specification, a cross contract invocation's gas limit is set to 98/100 of the remaining gas, whereas Optimism sets it to 63/64. With excessive call depth, the invocation frame in `MegaEVM` receives insufficient gas to execute compared to Optimism, leading to an inconsistency.

2.2.2.5 Gas Tainted

Description The result of the `GAS` opcode influenced the data flow or control flow, causing execution path divergence. In `MegaEVM`, gas values differ significantly from standard EVM due to the customized gas model, and when contracts use `gasleft()` or `GAS` opcode results in conditional logic or return values, this can lead to different execution outcomes (e.g., a cross contract invocation can fail in `MegaEVM` but succeed in `op-revm`).

[9] 0x001d810d64e811982ed836555859c27f6db8739846c0caf5a1520d38ed092cf4

The following case demonstrates the transaction out-of-gas caused by the *Gas Tainted*, using Optimism Mainnet transaction [10].

```
function swap(SwapExecutionParams calldata execution)
    external
    payable
    returns (uint256 returnAmount, uint256 gasUsed)
{
  uint256 gasBefore = gasleft();
... // bunch of operations
  unchecked {
    gasUsed = gasBefore - gasleft();
  }
}
```

Listing 2.3: KyberSwap: Meta Aggregation Router v2

In the `KyberSwap: Meta Aggregation Router v2` contract, the function calculates the gas used, returns it as a value, and ultimately affects the transaction's output.

2.2.2.6 Sufficient Gas

Description Transactions that are successfully completed in `MegaEVM` due to the 100x gas limit increase (applied in our testing to accommodate storage gas costs), whereas they might have failed or behaved differently in `op-revm`. This creates an inconsistency where transactions that would fail on Optimism Mainnet due to insufficient gas succeed on `MegaEVM` with the increased limit.

The following case demonstrates the transaction out-of-gas caused by the *Sufficient Gas*, using Optimism Mainnet transaction [11].

```
  NftFarmStrategy.exit()
  1 → CALL
     0xe0148a2f48a6f737fae04580c3bc6332db5f2474.multicall()
  .. // bunch of similar call
```

Listing 2.4: Invocation Trace

This transaction originally did not provide enough gas to execute on Optimism Mainnet. However, since we increased the gas limit by 100x for `MegaEVM`, the transaction executed successfully. The inconsistency arises because the transaction succeeds on `MegaEVM` (due to the increased limit) but fails on Optimism Mainnet (due to insufficient gas).

2.2.2.7 Self Destruct

Description Differences arising from the disabled `SELFDESTRUCT` opcode in `MegaEVM`. While Optimism allows `SELFDESTRUCT` operations, `MegaEVM` rejects them as Mini-Rex does not support this opcode to ensure state immutability and simplify state management. Transactions attempting to use `SELFDESTRUCT` will fail in `MegaEVM` but succeed in `op-revm`.

[10] 0x02759858330b451a740ce92b2e3aeea199af118743730bb380114981f6d123a4

[11] 0x0005e953ce9640f26b12f2777af60f6469128c41da2b3de830745b80e3806f9e

This case demonstrates the transaction out-of-gas caused by the *Self Destruct*, using Optimism Mainnet transaction [12].

```
function powerDown() external {
    require(msg.sender == _original, "XEN Proxy: unauthorized");
    selfdestruct(payable(address(0)));
}
```

Listing 2.5: 0xaf18644083151cf57f914cccc23c42a1892c218e

The transaction shows a call to the `powerDown` function, which executes the `SELFDESTRUCT` opcode. While Optimism permits `SELFDESTRUCT`, `MegaEVM` rejects this operation because MiniRex does not support the opcode.

2.2.2.8 Sufficient Gas still Failed

Description Call frames had sufficient gas to execute (unlike in `op-revm`), but the execution still failed due to other constraints or logic. Even with the 100x gas limit increase applied in testing, some transactions fail in `MegaEVM` due to business logic constraints, revert conditions, or other non-gas-related failures, creating divergence from the reference implementation.

The following case demonstrates the transaction out-of-gas caused by the *Sufficient Gas still Failed*, using Optimism Mainnet transaction [13].

```
[SENDER] 0xd526cf...8e3f
|
|-- 1. CALL (141,988 gas) REVERTED: execution reverted
|      to: 0xe8d5d017f08e...0x88315635
|
|   |-- 1.1 STATICCALL (9,750 gas) REVERTED
|   |      to: USDC.balanceOf
|   |      account = 0xf6b1258a5b8dd72e00...9e0cc0e5aab7
|   |    ↩ returned: 5,086,204,850 (but reverted upstream)
|
|   |-- 1.2 DELEGATECALL (2,553 gas) REVERTED
|   |      to: FiatTokenV2_2.balanceOf
|   |      same account = 0xf6b1258a5b8dd72e00...9e0cc0e5aab7
|   | ↩      returned same value: 5,086,204,850
|
|   |-- 1.3 CALL (96,321 gas) REVERTED: execution reverted
|   |      to: UniswapV3Pool.swap
|   |      params:
|   |          - recipient = 0…xf6b1258a
|   |          - zeroForOne = false
|   |          - amountSpecified = 483,508,302,323,933,184
|   |          - sqrtPriceLimitX96 = 1,461,446,703,485,210,103,287,...
|   |          - data = ""
|   |
|   |   |-- 1.3.1 CALL (14,083 gas)
... // bunch of reverted calls
```

[12] 0x00122debf10f48dd55dec8b3481fdf44fde66e664e55895551345c15008672d9

[13] 0x028d0a37894e3a1a7229b4bc96a68bcf6790e6efee706dd097a157a66472a9e7

Listing 2.6: Invocation Trace

Since the MegaEVM gas limit was increased by 100x, the first cross contract invocation succeeded, whereas it failed on Optimism Mainnet. However, a subsequent cross contract invocation failed, causing the transaction to revert nonetheless.

2.2.2.9 Bomb Opcode

Description Reverts triggered by excessive use of "bomb" opcodes (e.g., LOG) which exhausted the gas limit despite the increase. Operations imposing storage burdens are significantly more expensive on MegaEVM due to the Dual Gas Model, and even with the 100x gas limit increase, transactions that emit large logs or perform other storage-heavy operations can still exhaust the gas limit and revert due to Mini-Rex's bomb opcode protection mechanism.

The following case demonstrates the transaction out-of-gas caused by the *Bomb Opcode*, using Optimism Mainnet transaction [14].

```
emit 0xA0Cc33Dd6f4819D473226257792AFe230EC3c67f.
    0x4e41ee13e03cd5e0446487b524fdc48af6acf26c074dacdbdfb6b574b42c8146(
        0x00000000000000000000000000000000000000000000000000000000000000a1
        0000000000000000000000000000000000000000000000000000000000000002
        0000000000000000000000000000000000000000000000000000000000000014
        00000000000000000000000e71bdfe1df69284f00ee185cf0d95d0c7680c0d4
        000000000000000000000000000000000000000000000000b957244a3
    )
```

The transaction emits a log with large data. Since operations imposing storage burdens (e.g., LOG) are significantly more expensive on MegaEVM, the transaction reverted due to the "bomb opcode" protection mechanism in Mini-Rex.

2.2.2.10 Storage Tainted by Sender Balance

Description Divergence caused by the initial sender balance difference (due to different gas limits), where the balance value influenced control or data flow. In our testing setup, the 100x gas limit increase for MegaEVM transactions results in a larger gas reservation from the sender's balance, which can affect contract logic that inspects account balances, leading to different execution paths and storage updates compared to the reference implementation.

The following case study demonstrates how the gas_limit itself can influence execution flow and storage updates, using Optimism Mainnet transaction [15]. The BeefyRevenueBridge contract uses the sender's ETH balance to determine how much to transfer or bridge.

```
1   function _sendCowllectorFunds() private {
2     if (cowllector.sendFunds) {
3       // Checks off-chain balance of cowllector plus token balance
4       uint256 amountOnHand = address(cowllector.cowllector).balance + IERC20(native).balanceOf(
            cowllector.cowllector);
```

[14] 0x1b74b4be0e936d7e2f3baf462a057cdee440ce34c6f3d7b8cb282d3c8946a1aa

[15] 0x19ff8c9047cfe34aa6e4d9f43f95fc0e34d424cb8ae4d94d5b776f46bd3b0e1d

```
5       if (amountOnHand < cowllector.amountCowllectorNeeds) {
6         uint256 thisBal = _balanceOfNative();
7         // Logic depends on 'amountOnHand', which is derived from address balance
8         uint256 amountToSend = cowllector.amountCowllectorNeeds - amountOnHand;
9         amountToSend = amountToSend > thisBal ? thisBal : amountToSend;
10        IERC20(native).safeTransfer(cowllector.cowllector, amountToSend);
11        emit CowllectorRefill(amountToSend);
12      }
13    }
14  }
```

Listing 2.7: BeefyRevenueBridge

As we increased the `gas_limit` for `MegaEVM` transactions by 100x to accommodate storage gas, the sender's initial balance (in the simulation) is decremented by `gas_limit * gas_price` at the start of the transaction (or strictly speaking, the max cost is reserved). If the contract logic relies on `address(sender).balance` (or similar balance checks of the executing account), the available balance will appear significantly lower in `MegaEVM` due to the massive gas reservation. In this specific transaction, the logic calculating `amountOnHand` or `thisBal` (if related to the sender or a contract funded by the sender) computes a different `amountToSend`. This leads to different transfer amounts and storage updates compared to the reference execution, resulting in a *Balance Tainted* divergence. This is a subtle artifact of the testing methodology (altering gas limits) interacting with application logic that inspects balances.

2.2.2.11 Volatile Data Gas Limit

Description Transactions fail when accessing volatile opcodes (e.g., `TIMESTAMP`, `BLOCKHASH`) followed by storage operations. In `MegaEVM`'s Volatile Gas Design, accessing volatile opcodes triggers a gas limit cap at the volatile gas limit. If the transaction subsequently attempts a storage write operation (e.g., `SSTORE`), the capped gas limit may be insufficient, causing the transaction to revert even though it would succeed in standard EVM.

This case demonstrates the transaction out-of-gas caused by the *Volatile Data Gas Limit*, using Optimism Mainnet transaction [16]. The contract first accesses a volatile opcode (`timestamp`) and then performs a storage write operation. When the `timestamp` opcode is triggered, the transaction's gas limit is capped at the volatile gas limit, which is insufficient for the subsequent `SSTORE` opcode.

```
1  function _distUpgrading(uint256 _user, uint256 _level) private {
2    bool isSuper;
3    if (msg.sender == owner && (block.timestamp - startTime) < 13 hours)
4      isSuper = true;
5    uint256 upline = userInfo[_user].upline;
6    uint256 referrer = userInfo[_user].referrer;
7    uint256 _RankPayout = levels[_level];
8    //bool UP_Found = false;
9
10   _RankPayout = (_level <= 2)
```

[16] 0x01b1edc0cd5c228971a08cba917e39de535cc153c087e7fc6f4c4d2c695d8ffc

```
11      ? (_RankPayout * 90) / 100
12      : (_RankPayout * 45) / 100;
13
14    if (_level > 2) {
15      if (
16        userInfo[referrer].level > _level &&
17        userInfo[referrer].directTeam >= directRequired
18      ) {
19        payable(userInfo[referrer].account).transfer(_RankPayout);
20        userInfo[referrer].totalIncome += _RankPayout;
21        userInfo[referrer].levelIncome += _RankPayout;
22        userInfo[referrer].income[_level] += _RankPayout;
23        incomeInfo[referrer].push(
24          Income(_user, 25, _RankPayout, block.timestamp)
25        );
26      } else {
27        for (uint256 i = 0; i < royalty.length; i++) {
28          if (!isSuper)
29            royalty[i] += (_RankPayout * royaltyPercent[i]) / 100;
30        }
31      }
32    }
```

Listing 2.8: _distUpgrading

2.3 Fuzzing-based Testing of MegaEVM

To address corner cases not covered by differential testing, our fuzzing effort employs two complementary strategies: transaction-level fuzzing driven by an intermediate representation (IR) and low-level opcode fuzzing. Both are coupled with strict oracle checks to catch safety violations. The workflow follows a classic coverage-guided architecture with specific instrumentation for MegaEVM:

- **Seed Corpus**. Historical Optimism Mainnet transactions supply realistic calldata layouts and control-flow shapes. A separate generator produces well-formed raw bytecode programs to stress interpreter edge cases.
- **Lifter and Mutators**. The IR lifter extracts reachable contract bytecode into basic blocks and edges, enabling semantics-aware mutations (block recombination, opcode tweaks) while transaction fields are mutated independently (gas, value, access list, storage preimages).
- **Scheduler**. A coverage- and cost-aware queue prioritizes seeds that open new edges in MegaEVM code or approach oracle thresholds (long runtimes), ensuring both breadth and depth of exploration.
- **Executor**. An instrumented MegaEVM instance runs each mutated input, emitting lightweight block/edge coverage and resource usage metrics without perturbing gas semantics.
- **Oracle and Triage**. Panics/asserts, >10 GB RSS, and >10 s wall-time failures are flagged. Passing seeds are deduplicated by coverage/signature and fed back into the queue for further mutation.

2.3.1 System Design

IR-based Transaction Fuzzing Figure 2.3 illustrates the IR-based fuzzer: reachable contract bytecode is lifted into basic blocks and edges, control-flow- and opcode-aware mutations are applied, and mutated transactions plus valid bytecode are executed on `MegaEVM`. Coverage and oracle outcomes guide seed prioritization and minimization so the corpus expands interpreter coverage while preserving program validity.

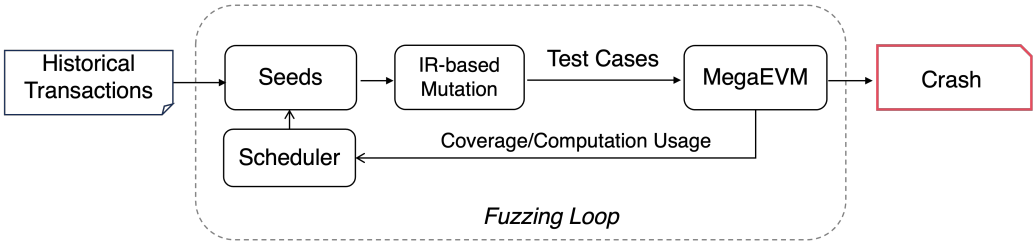

Figure 2.3: IR-based transaction fuzzing framework.

An Intermediate Representation (IR) is a structured form of reachable contract bytecode (basic blocks, edges, stack effects). It does not encode transaction fields. Using the IR avoids blind byte-level mutation that often produces invalid programs. Instead, we mutate semantics-aware units (blocks, calls, storage accesses) while preserving stack correctness and valid jump targets. The IR also stabilizes coverage measurement because equivalent bytecode variants map to the same logical blocks, improving deduplication.

We bootstrap the corpus with historical Optimism Mainnet transactions to ensure realistic calldata shapes and control-flow patterns. Each seed pairs the original transaction fields with the IR of the reachable contract bytecode. Transaction fields stay in their native format and are mutated separately. The IR keeps bytecode mutations valid:

- *Transaction-level Mutations*: Weighted mutation for `gas_limit`, `gas_price`, value, and access lists, and perturb storage pre-images and account balances to exercise corner cases in gas accounting and state access.
- *Control-flow Mutations* (IR-guided): Recombine IR basic blocks, splice call sequences, and insert error-handling branches to stress divergent execution paths.
- *Opcode-level Mutations* (IR-guided): Within the lifted bytecode, mutate jump destinations, stack manipulation instructions, and memory operations while maintaining stack soundness, preventing trivial invalid-program rejections.

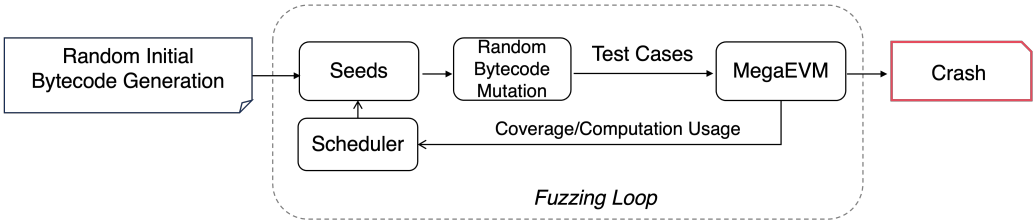

Figure 2.4: Low-level bytecode fuzzing framework.

Low-level Bytecode Fuzzing Figure 2.4 illustrates the bytecode fuzzer. Specifically, it skips IR lifting and instead generates stack- and jump-safe bytecode directly, emphasizing deep

stacks, dense memory and storage accesses, and unusual control flow. Note the input generation are guided by the coverage and computation usage without depending on realistic transaction shapes.

To probe interpreter correctness beyond real-world transaction shapes, we also generate low-level bytecode programs without IR scaffolding. Programs are built by sampling opcodes under constraints that ensure the basic correctness (stack depth limits, bounded code size, guarded jump targets). This method can exercise the correctness of the basic implementation of each opcode in `MegaEVM`.

Oracle Design We instrument `MegaEVM` with crash and computation usage oracles to detect correctness issues:

- *Panic/Assertion Failures*. Rust panics or debug assertions indicating safety violations.
- *Out-of-Memory*. Resident memory usage exceeding 10 GB, signaling unreasonable memory usage.
- *CPU Timeout*: Single-execution wall time exceeding 10 seconds, indicating potential denial-of-service (DoS) loops or gas mis-accounting.

Guidance and Scheduling Seed selection is guided by edge coverage and execution cost:

- *Coverage-guided Queueing*. It prioritizes seeds that increase block and edge coverage, maximizing semantic exploration of interpreter code paths.
- *Cost-aware Scheduling*. It favors seeds that approach the CPU-time oracle threshold, helping surface performance pathologies such as quadratic memory growth or pathological CALL fan-out.

2.3.2 Evaluation and Findings

Setup We executed both fuzzing strategy (IR-based transaction fuzzer and bytecode fuzzer) on a workstation with an Intel Core i9-13900K CPU and 128 GB RAM running Ubuntu 24.04 LTS. Each fuzzing strategy was allocated 8 CPU cores for 12 hours.

`MegaEVM` was built from commit 32ef93f61e6864f8053c6332871ab34e82f83056, with instrumentation added to emit lightweight coverage signals and oracle events.

ID	Severity	Description	Category	Status
2.3-1	High	DoS Attack via Unbounded `tstore` Operations	Security Issue	Confirmed
2.3-2	High	DoS Attack via Cheap KZG Calculation	Security Issue	Confirmed
2.3-3	High	EVM Crash due to Incorrect Gas Cost Implementation in KZG Precompile	Security Issue	Fixed
2.3-4	High	DoS Risk Through High-Cost Opcode Looping	Security Issue	Confirmed

Table 2.3: Summary of fuzzing findings for `MegaEVM`.

2.3.2.1 Overall Results

The campaign produced 108 unique timeout-triggering seeds, all captured by the 10-second CPU oracle. We reached 66.17% code coverage with the IR-based transaction fuzzer and 63.15% code coverage with the bytecode fuzzer. Additionally, we observed one out-of-memory case that exhausted more than 20 GB of host memory within a single transaction configured with a 10 billion gas limit. Notable characteristics of the timeout-triggering seeds include infinite loops coupled with intensive stack and memory operations.

In total, we found **four** security issues, as shown in Table 2.3. The details of these issues are presented in the sections below.

2.3.2.2 DoS Attack via Unbounded `tstore` Operations

Status Confirmed

Description There is no gas limitation or revision for the `tload/tstore` instructions, which can cause DoS issues for both stateless validators and the sequencer. An attacker can construct malicious transactions that abuse the 10B gas limit to enlarge the in-memory `TransientStorage` mapping, leading to excessive memory usage (>20GB) and slow computation time (>45s for a single transaction execution). Specifically, the `tstore` instruction defined in `reth` uses the following type for `TransientStorage`.

```
/// Structure used for EIP-1153 transient storage
pub type TransientStorage = HashMap<(Address, StorageKey), StorageValue>;
```

Listing 2.9: crates/state/src/types.rs

This mapping has the same memory consumption as normal K/V updates, but there is no limitation in the `MegaEVM`'s MiniRex spec.

When `tstore` is called with a new key, the mapping is directly updated.

```
#[inline]
pub fn tstore(&mut self, address: Address, key: StorageKey, new: StorageValue) {
    let had_value = if new.is_zero() {
        // if the new value is zero, remove entry from transient storage.
        // if the previous value was some insert it inside journal.
        // If it is none nothing should be inserted.
        self.transient_storage.remove(&(address, key))
    } else {
        // insert values
        let previous_value = self
            .transient_storage
            .insert((address, key), new)
            .unwrap_or_default();

        // check if previous value is same
        if previous_value != new {
            // if it is different, insert previous values inside journal.
            Some(previous_value)
        } else {
            None
```

```
21         }
22     };
23 }
```

Listing 2.10: crates/context/src/journal/inner.rs

However, the `self.transient_storage.insert((address, key), new)` operation is memory-intensive but only costs `100` gas (i.e., `gas::WARM_STORAGE_READ_COST`).

```
1 /// EIP-1153: Transient storage opcodes
2 /// Store value to transient storage
3 pub fn tstore<WIRE: InterpreterTypes, H: Host + ?Sized>(context: InstructionContext<'_, H, WIRE>)
      {
4     check!(context.interpreter, CANCUN);
5     require_non_staticcall!(context.interpreter);
6     gas!(context.interpreter, gas::WARM_STORAGE_READ_COST);
7
8     popn!([index, value], context.interpreter);
9
10    context
11        .host
12        .tstore(context.interpreter.input.target_address(), index, value);
13 }
```

Listing 2.11: crates/interpreter/src/instructions/host.rs

Therefore, an attacker can construct a transaction filled with `tstore` operations to different keys to expand the TransientStorage mapping. Since `MegaEVM` typically has a 10 billion gas limit for a single transaction, there can be around 100,000,000 `tstore` operations and 100,000,000 keys in the mapping, leading to **over 20 GB memory consumption** in our evaluation:

Each `HashMap<(Address, StorageKey), StorageValue>` entry consumes approximately 108 bytes:

- Key: 52 bytes (Address: 20 bytes + StorageKey: 32 bytes)
- Value: 32 bytes (StorageValue)
- HashMap overhead: 24 bytes

Hence, the base memory usage is 100,000,000 entries × 108 bytes ≈ 10 GB. Considering the HashMap load factor and fragmentation, the actual consumption can be significantly larger.

Feedback from the project We temporarily set the Transaction Compute Gas Limit to 200 million to mitigate the DoS risk.

2.3.2.3 DoS Attack via Cheap KZG Calculation

Status Confirmed

Description The KZG precompile only costs `100_000 + 50_000` for a single round calculation. An attacker can forge a transaction filled with KZG calls to DoS the execution layer.

```
1 /// Gas cost for the KZG point evaluation precompile.
2 pub const GAS_COST: u64 = 100_000;
3
4 /// KZG point evaluation precompile. This is the modified version of the original precompile
```

```
/// with a custom gas cost.
pub const KZG_POINT_EVALUATION: PrecompileWithAddress = PrecompileWithAddress(
    revm::precompile::kzg_point_evaluation::ADDRESS,
    |input, gas_limit| {
        let mut output = revm::precompile::kzg_point_evaluation::run(input, gas_limit)?;
        output.gas_used = GAS_COST;
        Ok(output)
    },
);
```

Listing 2.12: crates/mega-evm/src/precompiles.rs

```
/// Gas cost of the KZG point evaluation precompile.
pub const GAS_COST: u64 = 50_000;
```

Listing 2.13: revm-precompile/src/kzg_point_evaluation.rs

Feedback from the project We temporarily set the Transaction Compute Gas Limit to 200 million to mitigate the DoS risk.

2.3.2.4 EVM Crash due to Incorrect Gas Cost Implementation in KZG Precompile

Status Fixed in Commit d62345f

Description The KZG precompile is wrapped by a custom gas cost function that sets the output gas used without checking the remaining gas.

```
/// Gas cost for the KZG point evaluation precompile.
pub const GAS_COST: u64 = 100_000;

/// KZG point evaluation precompile. This is the modified version of the original precompile
/// with a custom gas cost.
pub const KZG_POINT_EVALUATION: PrecompileWithAddress = PrecompileWithAddress(
    revm::precompile::kzg_point_evaluation::ADDRESS,
    |input, gas_limit| {
        let mut output = revm::precompile::kzg_point_evaluation::run(input, gas_limit)?;
        output.gas_used = GAS_COST;
        Ok(output)
    },
);
```

Listing 2.14: crates/mega-evm/src/precompiles.rs

`alloy-evm` asserts that the output `gas_used` must not exceed the gas limit when a precompile is called successfully.

```
match precompile_result {
    Ok(output) => {
        let underflow = result.gas.record_cost(output.gas_used);
        assert!(underflow, "Gas underflow is not possible");
        result.result = InstructionResult::Return;
        result.output = output.bytes;
    }
}
```

Listing 2.15: src/precompiles.rs

Therefore, if a KZG precompile is called successfully but the remaining gas is less than 100,000, the assertion fails, causing the EVM to crash.

2.3.2.5 DoS Risk Through High-Cost Opcode Looping

Status Confirmed

Description `MegaEVM` enforces a 10-billion gas limit per transaction. To understand the worst-case execution cost under this limit, we conducted stress tests by constructing tight loops for each opcode, ensuring stack-correct execution (i.e., adding required `PUSH/POP` instructions when necessary). For each opcode, we maximized the number of iterations until the full gas budget (10 billion) was consumed.

Through this systematic evaluation, we observed that several opcodes can drive CPU execution time beyond 10 seconds on our test machine. The following six opcodes were the most expensive and can consume a lot of CPU computing power.

Opcode	CPU Time (s)
BN128ADD	87.540
TSTORE	83.990
KECCAK256	49.150
BLAKE2F	16.140
IDENTITY	13.850
SHA256	12.270

Table 2.4: CPU execution time for high-cost opcodes under 10 billion gas limit

Feedback from the project We temporarily set the Transaction Compute Gas Limit to 200 million to mitigate the DoS risk.

2.4 Replay-based Validation of `Stateless Validator`

We replay historical Optimism Mainnet transactions on `Mega-Reth` (sequencer role), generate `SALT` witnesses with the witness generator, and feed those witnesses to the `Stateless Validator`. Agreement between the validator's reconstructed state root and the sequencer's state root indicates a successful stateless validation.

1. **Genesis Construction**. We build a genesis from the 6085441 transactions in 250k blocks starting at height 135,589,412 of `op-revm`. After that, we combine the repo-provided block environment with an alloc derived from fetched storage and remove empty accounts to avoid database errors, where an account is empty if: (a) balance and nonce are zero/none and storage is absent, or (b) balance and nonce are zero/none and all storage values are zero. In the end, we generate the genesis file with 6,820,217 account records.
2. **Sequencer Replay** (`Mega-Reth`). We run `Mega-Reth` with the constructed genesis and replay Optimism Mainnet transactions to produce blocks/state.

3. **Witness Generation**. We copy the sequencer datadir to a witness directory and run the `megaeth-witness` binary to emit `SALT` witnesses for each block.
4. **Stateless Validation**. Finally, we run the `Stateless Validator` against the generated witnesses and blocks, comparing computed state roots with the sequencer outputs.

In the following, we describe two rounds of testing, conducted without and with `chain-ops`, respectively.

2.4.1 Evaluation and Findings of Round #1

Setup We utilized `Mega-Reth` (commit 78f00b880b4fd5c518416512ae445c4dec13c0a2) and the `Stateless Validator` (commit f239b511b0305dae792276014e344f8b0fa9a824) as the test targets. `Mega-Reth` was launched as a local sequencer with a 6-second block time. The data directory was subsequently duplicated for the witness generator, which was started and connected to the sequencer at `http://localhost:8545` on master port 16800. This configuration enabled witness production for each block.

2.4.1.1 Overall Results

`Stateless Validator` fails to fetch block updates. When sending Optimism Mainnet transactions to the sequencer and running the `Stateless Validator` with aforementioned setup, the block validation of stateless validator failed with missing block updates:

```
INFO Updating finalized block to BlockNumHash { block_number: 187, block_hash: 0
    x12715ad842253045582746c0ca865a9418a22fe9ec36139ed615fcc6d2042b74 }
2025-11-17T06:40:45.808443Z ERROR Failed to scan and process block witnesses: No block updates
    found for block(107) hash: 0xf58f27a7a2e7b51a7859c68075835b07cf0a482574c330955c9c75ff5d3e5b0d
    in remote dev
```

Listing 2.16: Error Message

This indicates the validator could not locate witness updates for block 107 in the remote datastore, preventing end-to-end validation under this round.

2.4.2 Evaluation and Findings of Round #2

Setup As shown in Table 2.2, we employed `Mega-Reth` (commit 23cb905f4), `chain-ops` (commit c22af78d), and the `Stateless Validator` (commit cb02b0d). The entire environment—comprising the sequencer, witness generator, and validator—was orchestrated using the chain-ops invoke start-op-stack workflow. For validation, Optimism Mainnet transactions spanning the same 250,000-block window (starting at height 135,589,412) were replayed through the sequencer endpoint.

2.4.2.1 Overall Results

The `Stateless Validator` successfully validated the generated blocks against the `SALT` witnesses, confirming state-root agreement with `Mega-Reth` under this round.

Chapter 3 Security Testing for SALT

In this chapter, we first describe the targets, scope, and high-level methodology for the security testing of the SALT in Section 3.1. Overall, our evaluation of SALT consists of three complementary test tracks.

Specifically, Section 3.2 presents robustness testing by replaying actual mainnet transactions to validate SALT's behavior under real-world workloads. Section 3.3 then introduces our stateless fuzzing approach, which treats SALT as an isolated module and focuses on comprehensive coverage of its public API surface. Section 3.4 builds on the state snapshots captured during the replay phase to perform stateful fuzzing from realistic initial states derived from blockchain history.

3.1 Targets, Scope, and Methodology

3.1.1 SALT Overview

Small Authentication Large Trie (SALT) is an authenticated key-value store designed to serve as the state management layer for blockchain systems, specifically developed for the MegaETH blockchain. Traditional blockchain implementations use Merkle Patricia Tries (MPT) for state management, which suffer from significant performance bottlenecks: they require frequent random disk I/O operations during state root updates and maintain large, sparse tree structures that consume excessive memory. SALT addresses these limitations through a novel two-tier architecture that combines a static, shallow main trie with dynamic bucket-based storage. The system uses cryptographic vector commitments based on Inner Product Arguments (IPA) with Pedersen commitments over the Banderwagon elliptic curve.

SALT is implemented as a Rust workspace comprising three crates that form a layered architecture, with each layer building upon the cryptographic primitives provided by lower layers.

- **SALT Core Layer**. The SALT Crate (`salt/`) and its implementation is organized into three logical sub-layers:
  - **State Management Layer** (`state/`). This layer manages the actual key-value data using Strongly History-Independent (SHI) hash tables. The `EphemeralSaltState` module provides an in-memory state view that implements SHI hash table operations (insertion, deletion, lookup) and translates between plain keys (used by applications) and SALT's internal bucket-slot addressing scheme. The SHI hash table algorithm uses linear probing with a canonical ordering invariant to ensure that the same set of key-value pairs always produces the same deterministic layout, regardless of insertion order. The layer also tracks state changes via StateUpdates for batch processing.
  - **Authentication Layer** (`trie/`). This layer computes and maintains cryptographic commitments for the entire state using IPA vector commitments. The `StateRoot` module manages the two-tier trie structure: a static 4-level, 256-ary main trie with approximately 16.8 million leaf nodes, where each leaf represents a bucket commitment. When buckets grow beyond 256 slots, they expand into dynamic subtrees (up

to 5 additional levels). The layer leverages the homomorphic properties of IPA commitments to perform incremental updates, recomputing only the commitments along changed paths rather than rebuilding the entire tree. The `TrieUpdates` module tracks commitment changes across all trie levels for batch persistence.

- **Proof Layer** (`proof/`). This layer generates and verifies cryptographic witnesses for state verification. The `Witness` module provides a high-level interface for creating proofs of key existence or non-existence. The `SaltWitness` module handles the low-level proof structure with authenticated state subsets.

- **Cryptographic Layer**. This layer is composed of two components:
  - **Banderwagon** (`banderwagon/`). This module provides elliptic curve operations over the Bandersnatch curve, specifically using the Banderwagon prime subgroup construction that eliminates low-order points and exceptional points. The crate implements core cryptographic primitives including group element operations (Element), scalar field arithmetic (Fr), and optimized multi-scalar multiplication (MSM) routines.
  - **IPA Multipoint** (`ipa-multipoint/`). Building on the Banderwagon curve operations, this module implements a vector commitment scheme using Inner Product Arguments. This module enables the homomorphic commitment updates that are central to SALT's incremental state root computation.

Scope Here we conducted the security testing against the project at the specific commit versions shown in the table below.

Project	Commit Hash
salt [1]	319b50c13f586cfc2492c858a0f3b617153eabf3

Table 3.1: Scope of security testing for the SALT

3.1.2 Methodology and Goals

To comprehensively validate the correctness and security of SALT, we designed a multi-faceted testing approach that addresses both robustness under real-world conditions and systematic exploration of the implementation's behavior space through fuzzing.

Since MegaETH is an Optimism Stack-based Layer 2 blockchain, we conducted robustness testing by replaying actual mainnet transactions to validate SALT's behavior under real-world workloads. Beyond *replay testing*, we implemented comprehensive fuzzing harnesses to systematically explore SALT's behavior space and discover potential bugs, or security vulnerabilities. Our fuzzing strategy is divided into two complementary approaches, each targeting different aspects of SALT's implementation.

- *Stateless Fuzzing*. This approach treats SALT as an isolated module and focuses on comprehensive coverage of its public API surface. Starting from an empty initial state, the stateless fuzzer generates arbitrary sequences of operations including key-value insertions, deletions, and lookups, etc. The fuzzer explores various input dimensions: key and value sizes (from empty to maximum allowed lengths), key distributions (sequential, random, clustered), operation sequences (patterns of insertions and deletions that

[1] https://github.com/megaeth-labs/salt

stress the SHI hash table algorithm), and proof generation scenarios (membership and non-membership proofs for various key patterns). The primary goal of stateless fuzzing is breadth of coverage—ensuring that all public APIs are tested with a wide variety of inputs, including boundary conditions and unexpected input combinations.

- *Stateful Fuzzing*. Building upon the state snapshots captured during the replay testing phase, stateful fuzzing operates from realistic initial states derived from actual blockchain history. The fuzzer randomly selects a continuous range of blocks from the replayed data and loads the corresponding state as the starting point for fuzzing. From this realistic baseline, it generates mutated inputs that are informed by the existing state. This approach is specifically designed to target bucket expansion and contraction scenarios. As buckets grow beyond capacity thresholds (256, 512 slots, etc.) and shrink, the bucket subtree structure dynamically changes, creating complex interactions in the authentication layer. Stateful fuzzing systematically explores these transitions to test the stability and correctness of commitment computations across structural changes, and verifying that each module maintains its invariants during rehashing operations.

3.2 Replay Testing

We selected 200,000 consecutive blocks from the Optimism mainnet following the Isthmus upgrade and integrated `SALT` into a custom replay program that processes each block's transactions sequentially, applying state updates to `SALT`, computing state roots and generating block witnesses.

This testing approach serves multiple purposes:

- It validates whether `SALT` can correctly handle the full spectrum of Ethereum state operations as defined;
- It checks whether the SHI hash table algorithm correctly manages hash collisions and maintains its ordering invariants under realistic key distributions;
- It verifies whether bucket resizing logic functions properly when buckets reach capacity during organic growth;
- It verifies whether bucket resizing logic functions properly when buckets reach capacity during organic growth;
- It confirms that the incremental commitment updates produce consistent state roots across block boundaries.

3.2.1 System Design

Replay testing serves as a critical validation mechanism for assessing the robustness of the MegaETH `SALT` component and its associated storage backend (MegaETH `rust-rocksdb`). The testing methodology involves re-executing historical blockchain data to verify that the system maintains correctness and stability under real-world workloads.

The replay process was conducted in two phases to isolate and test different system components:

- Phase 1: `SALT` Comoponent Validation.

- Target: MegaETH `SALT` component
- Scope: 200K blocks post-Isthmus upgrade
- Persistence: Block state and state updates dumped to disk concurrently
- Objective: Verify `SALT`'s robustness in state transitions, trie updates, and witness generation

- Phase 2: Storage Backend Integration.
  - Target: MegaETH `rust-rocksdb` integration
  - Scope: 10K blocks post-Isthmus upgrade
  - Persistence: Full persistence layer with MegaETH's customized RocksDB serially
  - Objective: Validate robustness of persistent storage operations and database interaction patterns

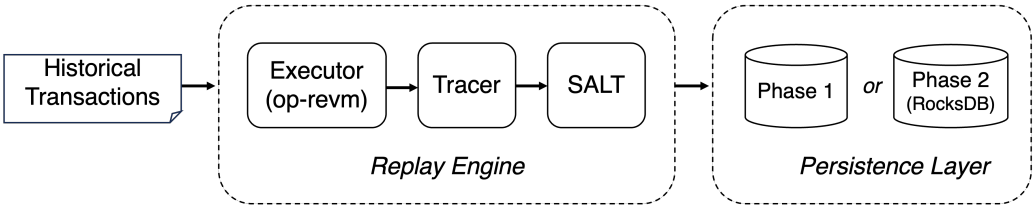

Figure 3.1: `SALT` Replay Framework.

Figure 3.1 gives the framework of the proposed replay framework. It uses a layered design that keeps execution, state tracking, and persistence separate. This design allows accurate re-execution of past blocks and collects complete block state transition data.

3.2.2 Evalution and Findings

Setup Our replay testing was executed on a machine equipped with a 12th Gen Intel(R) Core(TM) i7-12700 processor, 32 GB of RAM, and a ST2000DM008-2FR1 disk with 2 TB of storage. In phase 1, to accelerate the replay of 200,000 blocks, the workload was distributed across 8 concurrent instances. In phase 2, we used a single-instance sequential execution to isolate storage-layer behavior without concurrency-related complexity.

3.2.2.1 Overall Results

The overall results of the two phases are summarized as follows:

- **Phase 1: `SALT` Component Validation**. The replay completed without crashes or memory errors, and execution times aligned with expectations. All 200,000 blocks were successfully processed through the `SALT` component, producing state updates, trie commitments, and witness data without critical failures or unexpected behavior. These results indicate that `SALT` demonstrates strong robustness under production workloads.
- **Phase 2: Storage Backend Integration**. The integration test with MegaETH `rust-rocksdb` completed successfully without any unexpected issues. All database operations—including writes, reads, and batch commits—executed correctly, and post-test integrity checks confirmed data consistency. These results indicate that the MegaETH `rust-rocksdb` integration handles production-level write throughput, maintains data integrity, and delivers the expected performance characteristics.

3.3 Stateless Fuzzing

This approach treats SALT as an isolated component and focuses on comprehensive coverage of its public API surface. Starting from an empty initial state, the *stateless fuzzer* generates arbitrary sequences of operations, including key–value insertions, deletions, and lookups. It explores multiple input dimensions: key and value sizes (from empty to maximum allowed lengths), key distributions (sequential, random, clustered), operation sequences (insertion and deletion patterns that stress the SHI hash table algorithm), and proof generation scenarios (membership and non-membership proofs for diverse key patterns). The primary goal of stateless fuzzing is breadth of coverage, ensuring that all public APIs are exercised with a wide range of inputs, including boundary conditions and unexpected input combinations.

3.3.1 System Design

To comprehensively fuzz the `State`, `Trie`, and `Proof` modules, we use libFuzzer [2] as the fuzzing engine and design targeted harnesses, oracles, and mutation strategies for each module.

3.3.1.1 Harness

A fuzz harness is a specially crafted program or wrapper that repeatedly feeds generated or mutated inputs to a target function, library, or API, while linking necessary dependencies, in order to discover crashes and bugs. In this section, we introduce our harness design for the three SALT modules.

`State` and `Trie` Modules For the `State` and `Trie` modules, we designed three distinct scenarios based on the semantics of their APIs and test them separately. The fuzzing harness uses a three-scenario architecture, where each fuzzing iteration randomly selects one testing strategy: *SALT key operations*, *state root consistency*, or *two-phase commit*. LibFuzzer generates raw bytes that are deserialized into structured operations (for example, account insertions, storage updates, and storage deletions). These are then converted into key–value pairs and fed into SALT's state and trie APIs in orchestrated sequences designed to verify specific correctness properties.

- **SALT Key Operations**. SALT's architecture involves a multi-layered address translation scheme where plain keys are transformed through bucket ID computation (via hashing), slot ID resolution (via *SHI probing*), and ultimately mapped to internal `SaltKey` representations that encode bucket and slot coordinates, with corresponding `SaltValue` structures that embed both the original plain key and its associated value. Users interact with this system through three distinct lookup APIs: the function `find()` performs the complete SHI hash table search and returns both the internal `SaltKey` and the full `SaltValue`; the function `plain_value()` provides a convenience wrapper that extracts and returns only the value bytes; and the function `value()` enables direct slot access using an already-resolved `SaltKey`. Despite these APIs traversing different code paths, utilizing different

[2] https://llvm.org/docs/LibFuzzer.html

internal representations, and serving different use cases, they must all maintain a consistent view of the key-value mapping, as any inconsistency would indicate that implementation details of the internal addressing scheme are corrupting the data structure's integrity.

Thus we design this testing scenario to evaluate the consistency. The initial state construction follows a batch initialization approach. The fuzzer generates more than 1,000 random key-value pairs. These pairs are first committed to the state layer via the function `EphemeralSaltState::update_fin()`. The resulting state updates are then applied to the trie layer via `StateRoot::update_fin()`. This sequential two-phase process creates the initial state for consistency testing. After that , it performs checks the data consistency in the following ways:

1. `salt_value.value() == plain_value` confirms that `find()` and `plain_value()` return identical value bytes.
2. `salt_value.key() == plain_key` verifies that the `SaltValue` correctly embeds the original key without corruption.

- **State Root Consistency**. `SALT` involves complex operations like key insertions, deletions, bucket expansions, and rehashing. These operations create intricate state transitions that must preserve the fundamental invariant that incrementally-maintained state roots match ground-truth full rebuilds. `SALT`'s `StateRoot` module provides two distinct computation paths for state root derivation. For the first path, the function `update_fin()` performs incremental trie updates. In this approach, changes are propagated upward through the trie hierarchy, leveraging cached intermediate nodes and delta-based commitment recalculation. For the second path, the function `rebuild()` reconstructs all commitments from scratch by traversing the entire state storage. It is important that these two independent computation paths produce an identical state root after any sequence of operations.

  Thus, we design this testing scenario to verify that the identical state root is produced in both cases. In this scenario, the harness generates batches of diffrent operations, then executing a series of verifications for each batch: first applying regular key-value operations through the function `update_fin()` and persisting the state updates, then explicitly invoking the function `set_nonce()` to force bucket rehashing, then computing the incremental state root via the function `StateRoot::update_fin()`, and finally executing the function `rebuild()` to verify the identical state root. This verification is performed after every single batch rather than only at the end.
- **Two-phase Commit**. There are two distinct API patterns exposed for applying updates. Specifically, the first is a two-phase incremental interface: `EphemeralSaltState::update()` or `StateRoot::update()` may be called multiple times to accumulate key–value modifications in memory, after which a finalization call processes all pending updates. At the state layer, `canonicalize()` handles bucket reconciliation, while at the trie layer, `finalize()` performs trie node recomputations.

  The second is a single-phase atomic interface: `update_fin()` immediately performs both the data insertion and the complete finalization logic in one operation. Because of this dual-API design, there is a strict correctness requirement that both execution paths must produce identical final states, even though they traverse different code paths.

Thus we design this testing scenario to evaluate the correctness. In this scenario, the fuzzer harness generates more than 1000 randomized operation sequences and executes both the incremental path (repeated calls for the function `update()` followed by a call for the function canonicalize() or `finalize()`) and the atomic path (a single call for the function `update_fin()`) on identical input data, then verifies identical final states. The harness checks correctness at three levels:

1. *Per-key value comparison* using `plain_value()` to ensure individual key-value pairs match.
2. *State root hash comparison* between the two code paths to verify cryptographic commitments align.
3. *State root validation* using `StateRoot::rebuild()` to confirm that the incremental result matches a full reconstruction from storage.

Proof Module `SALT`'s `Proof` module implements the cryptographic proof designed for verification. This module creates a cryptographic witness containing proofs for existing keys and exclusion proofs for non-existing keys. The function `Witness::create()` takes three inputs: a list of plain keys to look up, a set of insert, delete, or update operations, and a store reference to access the state and trie data. The fundamental correctness requirements are that `Witness.verify()` must pass cryptographic verification. Besides, The state root of the witness must match the original state's root.

Thus we design the harness for `Witness` module to verify the correness and identical state root. The witness fuzzing harness first establishes baseline state by applying randomized key operations(e.g., key insertion, or key deletion). After that, it constructs a witness by invoking the function `Witness::create()`. Then it performs two checks where `Witness.verify()` must pass the cryptographic validation and the variable `Witness.state_root()` must match the result of the function `StateRoot::rebuild()`.

3.3.1.2 Oracle

A fuzzing oracle is a mechanism or check that determines whether a given input triggers a bug (such as a crash, assertion failure, memory error, or unintended behavior) during fuzz testing. In this section, we describe the general oracles that are applied to every fuzzing harness, in addition to the per-module-specific oracles presented in the previous section for each module.

Benchmark	Total Time	Average Time per KV
Salt trie update 1k KVs	20 ms	0.02 ms
Salt trie update 10k KVs	80 ms	0.008 ms
Salt trie update 100k KVs	600 ms	0.006 ms
Salt trie update 1m KVs	5 s	0.005 ms
salt trie incremental update 10 * 100 KVs	40 ms	0.04 ms
Salt trie update 100k expansion KVs	600 ms	0.006 ms
Rebuild 1m KVs	3.5 s	0.0035 ms

Table 3.2: Stateless oracle for `Stateless Validator`.

- **Cpu Execution Time per KV Limitation**. We evaluate the benchmark on our fuzz machine

with the feature test-bucket-resize (i.e., the variable `NUM_DATA_BUCKETS` is set to 2 and the variable `BUCKET_RESIZE_LOAD_FACTOR_PCT` is set to 1) to trigger more operations on buckets rehash and resize. Table 3.2 shows the result.

We first choose the maximum average time multiplied by 20 times (i.e., 0.8 ms) as the limitation for the CPU execution time per KV. It turns out that it results in many FP crashes in our experiment. After investigation, we identified two main causes for the excessive false positives. First, instrumentation (e.g., for coverage collection) added to the fuzzing binary significantly slows down the execution. Second, the number of generated key-value pairs impacts the average CPU time per KV. Under identical conditions, processing 10 KVs costs about 0.1 ms per KV on average, whereas processing 1,000 KVs reduces the average to approximately 0.02 ms per KV.

To mitigate false positives arising from these effects, we set the CPU execution-time per KV limitation to 2 ms and require each fuzzing iteration to generate at least 1,000 KVs. If the average execution time exceeds the limitation, it indicates potentially large loops or any other unexpected behaviors.

- **Memory Consumption Limitation**. After discussion with developers from MegaETH, we set the memory consumption limitation to 2 GB for each fuzz iteration. If the resident memory usage exceeds the limitation, it indicates unreasonable memory usage.
- **Address Sanitizer**. We enabled AddressSanitizer to detect memory safety violations, including out-of-bounds access, use-after-free, and memory leaks. If any memory safety violations are detected, it indicates that memory corruption may happen due to potential security bugs.
- **Panic/Assertion Failures**. The panic or assertion failures in a Rust program indicate potential safety violations.

3.3.1.3 Mutation

In addition to conventional coverage-guided mutation, we modify libFuzzer's default mutator to incorporate resource-aware feedback, specifically taking into account CPU execution time per key–value (KV) pair and memory consumption. Below, we describe the CPU-time–aware mechanism; the memory-aware counterpart follows an analogous design.

LibFuzzer traditionally relies on a coverage bitmap to estimate the "interestingness" of an input: the more unique bits an input sets, the deeper or rarer the exercised code paths are, and the fuzzer increases that seed's energy (mutation priority). We extend this by associating resource consumption with coverage. After each execution, we collect per-KV CPU execution time and discretize it into five bands based on percentile ranges within the observed distribution: 0–20%, 20–40%, 40–60%, 60–80%, and 80–100%. Inputs whose execution time falls into higher bands cause additional bits to be set in the coverage bitmap. As a result, libFuzzer's existing energy scheduling naturally assigns higher mutation priority to seeds that reach computationally expensive regions, steering the fuzzer toward inputs that trigger longer CPU execution times.

3.3.2 Evaluation and Findings

Setup We run the fuzzing on the machine with an Intel(R) Xeon(R) Silver 4214R CPU and 64 GB RAM running Ubuntu 22.04 LTS. We run each harness with 12 CPU cores for 36 hours. To trigger more operations on buckets rehash and resize, we enable the test-bucket-resize feature and set the variable `NUM_DATA_BUCKETS` to 2 and `BUCKET_RESIZE_LOAD_FACTOR_PCT` to 1 during our fuzzing process.

Module	File	Regions	Miss	Cover	Lines	Miss	Cover
`state`	hasher.rs	48	10	79.17%	25	4	84.00%
	state.rs	747	125	83.27%	482	83	82.78%
	hasher.rs	41	14	65.85%	33	8	75.76%
	Total	836	149	82.18%	540	95	82.41%
`trie`	node_utils.rs	76	0	100.00%	48	0	100.00%
	trie.rs	1518	434	71.41%	914	222	75.71%
	TOTAL	1594	434	72.77%	962	222	76.92%
`proof`	prover.rs	320	33	89.69%	215	27	87.44%
	salt_witness.rs	135	73	45.93%	97	60	38.14%
	shape.rs	119	0	100.00%	88	0	100.00%
	subtrie.rs	295	40	86.44%	213	35	83.57%
	witness.rs	233	117	49.79%	142	79	44.37%
	Total	1102	263	76.13%	755	201	73.38%
`ipa-multipoint`	crs.rs	134	83	38.06%	74	44	40.54%
	ipa.rs	470	99	78.94%	224	43	80.80%
	lagrange_basis.rs	248	56	77.42%	165	37	77.58%
	math_utils.rs	64	0	100.00%	30	0	100.00%
	multiproof.rs	396	44	88.89%	227	26	88.55%
	transcript.rs	79	14	82.28%	38	7	81.58%
	Total	1391	296	78.72%	758	157	79.29%
`banderwagon`	element.rs	249	19	92.37%	154	19	87.66%
	lib.rs	21	21	0.00%	10	10	0.00%
	msm.rs	70	70	0.00%	43	43	0.00%
	salt_committer.rs	198	8	95.96%	140	7	95.00%
	scalar_multi_asm.rs	10	0	100.00%	418	0	100.00%
	Total	548	118	78.47%	765	79	89.67%

Table 3.3: Code coverage of the stateless fuzzing.

3.3.2.1 Overall Results

We conducted coverage statistics for the modules in `Stateless Validator`. To obtain accurate and meaningful coverage metrics, we applied filtering rules to exclude functions that do not represent actual business logic or are unreachable at runtime. Specifically:

1. **Generic placeholder functions**. Rust's monomorphization process creates generic function signatures with placeholder types (e.g., <_>, <_, _>) that serve as templates for type specialization. These placeholders are never instantiated into actual executable code—they exist only as intermediate representations during compilation.
2. **Debug and display trait implementations**. Functions implementing `core::fmt::Debug` and `core::fmt::Display` traits are used exclusively for debugging output and error mes-

sage formatting. These functions are not part of the core business logic. We exclude these implementations to focus coverage metrics on the functional correctness of the system.

3. **Embedded external dependencies**. For the `state` module, we exclude the `ahash` from coverage statistics. This directory contains vendored code from the ahash crate that has been embedded into the project for deterministic hashing. Since this code originates from an external dependency and includes utility functions (such as type conversion helpers) that are not directly related to `Stateless Validator`'s core functionality, including it would introduce noise into the coverage metrics.

As shown in Table 3.3, nearly all modules achieve both *Region* coverage and `Line` coverage close to or exceeding 80%. We consider this level of code coverage sufficient and adequate for effective fuzz testing.

In total, we have **one** recommendation, as shown in Table 3.4.

- Recommendation: 1

ID	Severity	Description	Category	Status
3.3-1	-	Avoid Panics Caused by Fixed-Length `SaltValue` struct	Recommendation	Confirmed

Table 3.4: Summary of stateless fuzzing findings for `SALT`.

The details are presented in the sections below.

3.3.2.2 Avoid Panics Caused by Fixed-Length `SaltValue` struct

Status Confirmed

Description The function `SaltValue::new()` accepts arbitrary-length byte slices for parameters key and value but performs no boundary check before copying data into a fixed-size (i.e., 94 bytes) buffer. This may lead to potential panic when executing the function `copy_form_slice()`. This same problem also exists in the functions `key()` and `value()`.

```
267  pub const MAX_SALT_VALUE_BYTES: usize = 94;
```

Listing 3.1: salt/src/types.rs

```
274 pub struct SaltValue {
275     /// Fixed-size array accommodating the largest possible encoded data (94 bytes).
276     #[deref]
277     #[deref_mut]
278     #[serde(with = "serde_arrays")]
279     pub data: [u8; MAX_SALT_VALUE_BYTES],
280 }
281
282 impl SaltValue {
283     /// Create a new encoded value from separate key and value byte slices.
284     ///
285     /// Encodes the data in the format: `key_len`(1) | `value_len`(1) | `key` | `value`
286     pub fn new(key: &[u8], value: &[u8]) -> Self {
287         let key_len = key.len();
288         let value_len = value.len();
289
```

```
290         let mut data = [0u8; MAX_SALT_VALUE_BYTES];
291         data[0] = key_len as u8;
292         data[1] = value_len as u8;
293         data[2..2 + key_len].copy_from_slice(key);
294         data[2 + key_len..2 + key_len + value_len].copy_from_slice(value);
295
296         Self { data }
297     }
298
299     /// Extract the key portion from the encoded data.
300     pub fn key(&self) -> &[u8] {
301         let key_len = self.data[0] as usize;
302         &self.data[2..2 + key_len]
303     }
304
305     /// Extract the value portion from the encoded data.
306     pub fn value(&self) -> &[u8] {
307         let key_len = self.data[0] as usize;
308         let value_len = self.data[1] as usize;
309         &self.data[2 + key_len..2 + key_len + value_len]
310     }
```

Listing 3.2: salt/src/types.rs

Impact Potential panic due to the unchecked variables `key_len` and `value_len`.

Suggestion Add proper bound checks.

3.4 Stateful Fuzzing

Building upon the state snapshots captured during the replay testing phase, *stateful fuzzing* operates from realistic initial states derived from actual blockchain history. The fuzzer randomly selects a continuous range of blocks from the replayed data and loads the corresponding state as the starting point for fuzzing. From this realistic baseline, it generates mutated inputs that are informed by the existing state.

This approach is specifically designed to target bucket expansion and contraction scenarios. As buckets grow beyond capacity thresholds (256, 512 slots, etc.) and shrink, the bucket subtree structure dynamically changes, creating complex interactions in the authentication layer. Stateful fuzzing systematically explores these transitions to test the stability and correctness of commitment computations across structural changes, and verifying that each module maintains its invariants during rehashing operations.

3.4.1 System Design

Similar to the stateless fuzzing setup, we use libFuzzer as the fuzzing engine and design targeted harnesses, oracles, and mutation strategies specifically for the stateful fuzzing approach.

3.4.1.1 Harness

In this section, we present our harness design for the SALT stateful fuzzing campaign.

State Loading Mechanism The stateful fuzzing approach leverages historical blockchain state captured during the replaying phase. The state loader reads JSON-formatted block data files from the data directory. Each file contains:

- Pre-block state: key-value pairs needed for current block execution
- KV updates: state modifications from transaction execution
- Accessed keys: keys read during block processing

The loader processes a configurable number of consecutive blocks, applying their state updates sequentially to construct a realistic initial state. This approach ensures that the fuzzer operates on state configurations that reflect actual blockchain workloads.

Fuzzing Scenarios The fuzzing harness uses a two-scenario architecture, where each fuzzing iteration randomly selects one testing strategy. Both scenarios are designed to exercise critical code paths related to bucket management and state root computation.

- **Delete-restore Cycle**. This scenario targets bucket contraction logic and subtree updates by executing a complete delete–restore cycle. In the first phase, the harness loads historical state from replayed block data and computes the initial state root. It then randomly selects existing key–value pairs for deletion and applies these deletions at the state layer. For buckets that have expanded beyond the minimum size, the harness explicitly triggers rehashing to exercise the contraction path. After computing the incremental state root, a witness is generated and verified for the deletion operations.
  In the second phase, all previously deleted key–value pairs are restored with their original values. The harness computes the new state root and verifies that it matches the initial state root, then generates and verifies a witness for the restoration operations.
  This scenario validates three key properties: incremental–rebuild consistency (the incremental state root must match a full rebuild after each phase), witness correctness (generated witnesses must pass cryptographic verification), and reversibility (deleting and then restoring the same keys must reproduce the original state root, confirming that bucket contraction and expansion are correctly inverse operations).
- **Batch Updates Invariant**. This scenario verifies the equivalence of SALT's two API patterns for applying state updates. The first execution path uses atomic updates, where each batch applies updates to both the state layer and trie layer and then computes the state root. The second execution path uses incremental updates, where each batch accumulates updates in memory without immediate finalization, followed by processing all pending updates together before finalizing and computing the state root. Both execution paths must produce identical final state roots, ensuring that the two-phase update API and the atomic API are semantically equivalent.

3.4.1.2 Oracle

Our harness employs multiple layers of oracle checks to detect correctness violations.

- **State Root Consistency**. After any state modification, recomputing the state root from scratch (rebuild) must match the incrementally computed root. This check catches any

divergence between the optimized incremental update path and the canonical rebuild computation, which would indicate bugs in delta propagation, commitment caching, or bucket subtree management.

- **Witness Verification**. After each phase of operations, the harness generates a cryptographic witness and verifies it. This oracle validates the entire proof generation and verification pipeline, catching bugs in IPA proof construction, authentication path computation, or commitment serialization.
- **Reversibility**. Delete-then-restore operations must return to the original state. This check is particularly sensitive to bugs in bucket contraction/expansion logic, as any asymmetry in these operations would cause the restored state to differ from the original.
- **Update Mode Equivalence**. Different update strategies (atomic vs. incremental) must produce identical results. This ensures that implementation details of the incremental vs. atomic APIs do not affect the final cryptographic state.
- **Memory Consumption Limitation**. After discussion with developers from MegaETH, we set the memory consumption limitation to 2 GB for each fuzz iteration to detect excessive memory allocation that could indicate algorithmic complexity issues or memory leaks.
- **Address Sanitizer**. We enabled AddressSanitizer to detect memory safety violations, including out-of-bounds access, use-after-free, and memory leaks.
- **Panic/Assertion Failures**. The panic or assertion failures in a Rust program indicate potential safety violations.

3.4.1.3 Mutation

In addition to libFuzzer's conventional coverage-guided mutation strategy, our harness incorporates domain-specific mutation through structured fuzzing capabilities.

- **Structured Input Generation**. Rather than mutating raw bytes directly, the fuzzer generates structured inputs through automatic derivation. The input structure contains three components: a block offset for selecting historical state, a random seed for deterministic reproduction, and a list of key-value entries representing Ethereum account and storage data. This structured approach ensures that all generated inputs conform to `SALT`'s expected data formats while maintaining sufficient randomness to explore edge cases.
- **State-aware Mutation**. The harness implements state-aware mutation by using the loaded historical state as context for generating new operations:
  - Deletion targets. Keys to delete are randomly selected from actually existing entries in the current state, ensuring that deletion operations always target valid keys.
  - Insertion templates. New keys are derived from fuzzer-provided templates combined with existing state patterns, creating inputs that are structurally valid but explore new key space.
  - Value mutation. Values are derived from existing state entries, ensuring they conform to expected sizes while introducing variation.
- **Bucket-focused Mutation**. To maximize coverage of bucket management logic, we configure the fuzzer to concentrate keys into a minimal number of buckets with a lowered expansion threshold. Additionally, we prioritize seeds based on the frequency of bucket re-

size and rehash operations they trigger. Seeds that cause more bucket structural changes receive higher mutation priority, directing the fuzzer to preferentially explore inputs that exercise the bucket expansion, contraction, and rehashing code paths. This feedback-driven approach ensures that the fuzzing campaign intensively tests the dynamic bucket management logic that is critical to SALT's correctness.

3.4.2 Evaluation and Findings

Setup Our fuzzing was executed on a machine equipped with a 12th Gen Intel(R) Core(TM) i7-12700 processor and 32 GB of RAM. The fuzzer was configured to use 8 CPU cores and ran continuously for 24 hours.

To aggressively stress the bucket management logic, particularly rehashing and resizing, we enabled SALT's `test-bucket-resize` feature with the following configuration:

- `NUM_DATA_BUCKETS = 2`: Concentrates all keys into just 2 buckets (versus the default 16.7M buckets), dramatically increasing collision rates and forcing rapid bucket growth.
- `BUCKET_RESIZE_LOAD_FACTOR_PCT = 60`: Lowers the expansion threshold to 60% load factor, triggering bucket resizing much more frequently than in production.

This configuration ensures that bucket expansion and contraction operations occur with high frequency during fuzzing, providing comprehensive coverage of the dynamic bucket management code paths that are critical to SALT's correctness.

3.4.2.1 Overall Results

In total, we found **one** security issue, as shown in Table 3.5.

- Medium Risk: 1

ID	Severity	Description	Category	Status
3.4-1	Medium	Incorrect Parent Node Index in Bucket Subtree Updates	Security Issue	Fixed

Table 3.5: Summary of statelful fuzzing findings for SALT.

The details of this issue are presented in the sections below.

3.4.2.2 Incorrect Parent Node Index in Bucket Subtree Updates

Severity Medium

Status Fixed in Commit 7120f43

Description When calling `shi_rehash()` with a `new_capacity` that does not satisfy the constraint of $256 * 2^n$, there is a probability that witness verification will fail after witness creation. Through debugging, we identified the root cause in the `update_bucket_subtrees()` function. In Step 3 of this function, the calculation of `capacity_start_index` and `capacity_end_index` is incorrect. Specifically, for a node with index 257, its parent node index should be 1 rather than 0, but the current bit-shift implementation produces 0.

```
435  let mut capacity_start_index =
436    subtree_change.new_capacity >> MIN_BUCKET_SIZE_BITS as NodeId;
```

```
437  let mut capacity_end_index =
438    subtree_change.old_capacity >> MIN_BUCKET_SIZE_BITS as NodeId;
439  let bucket_id = (*bucket_id as NodeId) << BUCKET_SLOT_BITS as NodeId;
440
441  for level in (subtree_change.old_top_level + 1..MAX_SUBTREE_LEVELS).rev() {
442    let extrat_end = ((capacity_start_index + MIN_BUCKET_SIZE as NodeId)
443      & (NodeId::MAX - (MIN_BUCKET_SIZE as NodeId - 1)))
444      .min(capacity_end_index)
445      + bucket_id
446      + STARTING_NODE_ID[level] as NodeId;
447
448    let updates = (capacity_start_index..capacity_end_index)
449      .into_par_iter()
450      .filter_map(|i| {
451        let node_id = bucket_id + i + STARTING_NODE_ID[level] as NodeId;
452        let old_c = self.commitment(node_id).expect("node should exist in trie");
453        let new_c = default_commitment(node_id);
454        (new_c != old_c).then_some((node_id, (old_c, new_c)))
455      })
456      .collect::<Vec<_>>();
457
458    let split_point = updates.partition_point(|node| node.0 < extrat_end);
459    uncomputed_updates.extend(updates[split_point..].iter());
460    extra_updates[level].extend(updates[0..split_point].iter());
461
462    capacity_start_index >>= MIN_BUCKET_SIZE_BITS as NodeId;
463    capacity_end_index >>= MIN_BUCKET_SIZE_BITS as NodeId;
464  }
```

Listing 3.3: salt/src/trie/trie.rs

Impact When bucket capacity is set to a non-power-of-two multiple of 256 (e.g., 257 x 256), the parent node index calculation in bucket subtree updates produces incorrect results, leading to witness verification failures.

Suggestion Fix the parent node index calculation logic in `update_bucket_subtrees()` to correctly handle arbitrary capacity values.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS