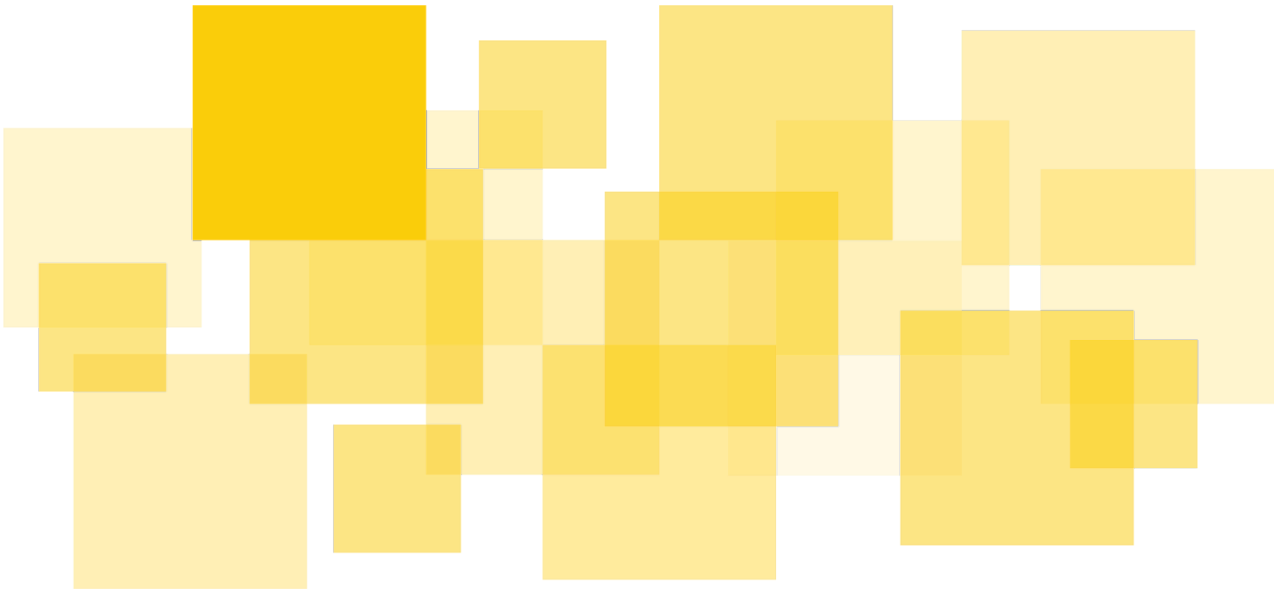
Security Audit Report

Monad Execution Other

Delivered: January 29, 2026



runtime verification

Table of Contents

Disclaimer

This report does not constitute legal or investment advice. You understand and agree that this report relates to new and emerging technologies and that there are significant risks inherent in using such technologies that cannot be completely protected against. While this report has been prepared based on data and information that has been provided by you or is otherwise publicly available, there are likely additional unknown risks that otherwise exist. This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system. This report is for informational purposes only and is provided on an "as-is" basis, and you acknowledge and agree that you are making use of this report and the information contained herein at your own risk. The preparers of this report make no representations or warranties of any kind, either express or implied, regarding the information in or the use of this report and shall not be liable to you or any third parties for any acts or omissions undertaken by you or any third parties based on the information contained herein.

Smart contracts are still a nascent software arena, and their deployment and public offering carry substantial risk.

Finally, the possibility of human error in the manual review process is very real, and we recommend seeking multiple independent opinions on any claims that impact a large quantity of funds.

Executive Summary

Monad Foundation engaged Runtime Verification Inc. to conduct a 5-week expedited review of the Monad protocol and code in preparation for the Monad Mainnet launch. The audit began on November 10, 2025, with the objective of evaluating the correctness and security of the protocol, reviewing existing quality assurance measures, identifying potential vulnerabilities, and providing recommendations for improvements in terms of code, testing, and documentation.

The audit's primary focus was on Monad's Execution component, with a secondary focus on the Consensus component primarily in regard to how the two components interact. Core mechanisms that have been introduced more recently, such as the Staking Contract and the Reserve Balance mechanism, were given priority.

Runtime Verification's audit process started with a detailed design review, where we familiarized ourselves with the protocol by reviewing documentation, formal specifications, and existing test harnesses. We then proceeded to compile an overview of existing quality assurance measures for the different components with the objective of identifying gaps and providing recommendations on how to address them. Using the understanding developed during the design review phase, we conducted detailed analyses of different parts of the system, including how the implementation, specification, formal models, and tests relate and compare to each other.

Scope

This engagement focused on reviewing the Monad execution and consensus codebases with an emphasis on security- and safety-critical interactions between execution and consensus. Given the size and complexity of the system, the scope was intentionally risk-driven rather than exhaustive. The review concentrated on components and code paths most likely to impact correctness, liveness, and economic integrity under adversarial conditions, rather than attempting full line-by-line coverage.

Specifically, the review covered the Monad execution repository (category-labs/monad) and the consensus repository (category-labs/monad-bft), with particular attention to execution–consensus coordination, staking-related logic that spans both layers, and the reserve balance mechanism used to support delayed and pipelined execution. In addition to a targeted manual review, we developed and exercised fuzzing harnesses for selected execution and consensus-facing components to explore edge cases, failure modes, and invariant violations, especially around stake handling and in-flight transaction semantics. Areas not directly related to these interactions, or not reachable within the engagement timeline, were considered out of scope.

In addition to the direct review of the in-scope repositories, we examined prior security work performed by external vendors and internal teams, including fuzzing harnesses developed by Asymmetric Research and Category Labs. This prior work was reviewed to understand existing coverage and assumptions, identify gaps in the current security posture, and inform our own testing strategy. Where appropriate, we extended and refined existing fuzzing harnesses to better exercise execution–consensus interactions, reserve balance handling, and staking-related logic, with the goal of improving coverage of adversarial and edge-case behaviors rather than duplicating prior analysis.

Methodology

Although manual code review cannot guarantee discovery of all possible invariant violations, we followed a systematic approach to verify the formal properties defined in the staking contract specification.

The code review process began with a thorough examination of the formal specification to understand each invariant's requirements and the intended system behavior. This specification review provided the foundation for assessing whether the implementation correctly maintains each property throughout its lifecycle.

Following the specification review, we conducted systematic static analysis of the staking contract implementation (`staking_contract.cpp` and related files). For each invariant, we identified the relevant code paths that maintain the property, traced state modifications through execution paths, and compared the implementation logic against the specification requirements. This process involved function-level analysis of precompile implementations, data flow analysis to trace state changes, control flow analysis to examine conditional branches, and cross-reference checking between related functions.

The static nature of our review means we examined code without execution, assessing logic correctness, specification compliance, error handling, and state transition preservation. We verified that the implementation's logic appears sound and matches the formal specification's requirements. In critical areas such as solvency, we traced balance changes through operations to verify that equations hold.

Certain properties came with inherent limitations to our review methodology. Properties involving consensus-execution coordination require integration review with the consensus layer, which falls outside the scope of a standalone execution layer code review. While we examined the execution layer's side of these operations, verifying cross-layer coordination would require integrated testing with the actual consensus implementation.

Monad Components and Quality Assurance Status

The following is an overview of the main components that comprise the Monad infrastructure, along with links to relevant documentation, coverage by past audits, and a breakdown of what kinds of quality assurance have been performed to validate each component (fuzzing harnesses, formal proofs, etc.).

Note: The past audit reports by Spearbit and Zellic linked in this document refer to audits conducted approximately during the months of July and August, 2025. As such, they don't cover more recent changes to the protocol from v0.11 onwards (see Releases in the Monad documentation). Notable changes since then include:

- Opcode repricing
- Changes to the gas model
- Introduction of Staking
- Introduction of Reserve Balance
- Support for various EIPs: EIP-2935, EIP-7951, EIP-2537, and EIP-7702.

Consensus

MonadBFT

Byzantine Fault Tolerant (BFT) consensus protocol used by Monad, specifically designed to be resistant to tail-forking.

Documentation

- Monad Docs: MonadBFT
- MonadBFT Research Paper
- Category Labs Blog Post: MonadBFT
- Category Labs Blog Post: MonadBFT Update

Past Audits

- Zellic Audit Report - Consensus Layer. The review covered consensus protocol implementation, including vote validation, block sync mechanisms, round voting bounds checking, stake amplification vulnerabilities, RLP decoding, timeout certificate processing, and signature verification. Notable gaps remain in integration testing between consensus and execution layers.

Networking

Monad implements a number of custom protocols for communication between nodes:

- RaptorCast is used to send block proposals from leaders to other validators during MonadBFT.
- Statesync is used by nodes to synchronize their view of the chain state as close as possible to the current tip.
- Blocksync is used by nodes to acquire missing blocks that have been accepted by the network but that they have not personally received.
- Peer discovery allows a new node to connect to the existing validator network.
- Pending transactions are also transmitted from RPC nodes to upcoming leaders so they can be added to the leaders' local mempools.

This section of the Monad infrastructure has been de-prioritized in the current audit in the interest of focusing on business logic. However, bugs were identified in RaptorCast during concurrent bug bounty efforts.

Documentation

- Monad Docs: RaptorCast
- Monad Docs: Local Mempool
- Monad Docs: Statesync
- Monad Docs: Blocksync
- Monad Docs: Peer Discovery
- Monad Docs: Transport Protocol Usage

Past Audits

- Zellic Audit Report - Networking Layer. The review covered assessment of Monad's networking stack, examining RaptorCast (both primary and secondary implementations), peer discovery mechanisms, dataplane components, router filtering logic, and router scheduling algorithms.
- Spearbit Audit Report

Execution

Staking Contract

Precompile used to register validators, delegate stake to them, and re-stake or withdraw validator rewards.

Documentation

- Monad Docs: Staking Precompile

Quality Assurance

- Category Labs fuzzing
  - ✅ Introduced Oct 9 2025. Up to date.

Past Audits

- Zellic Audit Report - Execution Layer. This review treated staking as one component among many in the execution layer, resulting in surface-level coverage rather than comprehensive security analysis.
- Spearbit Audit Report. This review put a particular focus on access control mechanisms, cryptographic operations, and economic security properties. The assessment also evaluated input validation and edge cases in delegation operations. However, the review did not cover all precompile functions, nor did it verify formal specification compliance. The integration between staking contract state changes and consensus layer validator set updates requires further review.

EVM Interpreter and JIT Compiler

Frequently-used contracts are compiled to x86 in order to increase performance. Execution of interpreted and compiled code should be equivalent.

Documentation

- Monad Docs: JIT Compilation

Quality Assurance

- Category Labs fuzzing
  - Introduced Feb 19 2025. Last updated Oct 1.
  - ⚠️ The compiler has an LLVM backend which is not covered by the fuzzer.
  - ⚠️ Support for the CLZ opcode was implemented and the instruction data for the fuzzing generator has not been updated.

- Asymmetric Research fuzzing
  - ✅ Introduced Nov 6 2025. Up to date.
  - Very basic, sends raw bytes as a contract to the compiler.

Past Audits

Detailed analysis document providing comprehensive review of EVM interpreter and JIT compiler scenarios not covered by past audits.

- Zellic Audit Report - Execution Layer . The review covered assessment of the execution engine, examining core infrastructure components, state synchronization mechanisms, and memory safety properties. However, it didn't extend to the following cases:
  - Multi-block attack scenarios.
  - Reserve balance mechanism (introduced after audit).
  - Systematic verification of parallel execution determinism.
  - Chain reorganization handling.
  - Compiled vs. interpreted equivalence under all edge cases.
  - Comprehensive review of interpreter-level gas calculation edge cases (integer underflow/overflow, CALL gas forwarding boundaries, EIP-2200 SSTORE boundaries).
  - Stack/memory/storage operation edge cases (boundary conditions, overflow/underflow scenarios, EIP-2929 access list tracking).
  - General CALL operation edge cases (value transfer scenarios, depth limit enforcement, EIP-7702 delegation detection).
- Zellic Audit Report - Compiler Layer. A focused security review was conducted by examining the JIT compilation pipeline, including IR generation, basic block construction, x86 code generation, and type system correctness. However, the following gaps are identified:
  - LLVM backend not covered by existing fuzzers.
  - Stress testing with complex contracts combining many opcodes.
  - Verification of compiler optimizations preserving exact EVM semantics.
- Spearbit Audit Report. The audit focused on staking contract security, mempool denial-of-service vectors, protocol correctness (EIP compliance), and RPC security. While specific opcode issues were covered (CREATE/CREATE2 max initcode, DELEGATECALL delegation flag), comprehensive EVM interpreter edge case testing was not performed. The following gaps are identified:
  - Comprehensive EVM interpreter opcode execution edge cases.
  - Gas calculation scenarios and boundary conditions (integer overflow/underflow, CALL gas forwarding boundaries, EIP-2200 SSTORE boundaries).
  - Stack, memory, and storage operation edge cases, including boundary conditions.
  - General call operation scenarios (gas forwarding, depth limits, value transfers) - only delegation-specific scenarios were covered.
  - Parallel execution determinism with the interpreter.
  - Interpreter-compiler equivalence verification.

Parallel Execution

Block execution in Monad happens in parallel, with each transaction being optimistically executed by a different process under the assumption that it is independent of the other transactions. Afterwards, the inputs read and outputs written by each transaction are compared. If one of the inputs to a transaction is an output of a previous transaction, the subsequent transaction is executed again with the corrected data. This re-execution still benefits from cached values from the original execution. The final result of executing the block should be the same as if it were executed sequentially.

Documentation

- Monad Docs: Parallel Execution

Past Audits

- Zellic Audit Report - Execution Layer. Parallel execution was examined as part of execution layer review, but it did not receive dedicated systematic security analysis. Gaps remain in systematic determinism testing to verify that identical blocks produce identical state roots, regardless of thread scheduling, execution order, or system load conditions.

Consensus-Execution Interaction

Gas Model

Due to asynchronous execution, where blocks are built before being executed, there is no way to know exactly how much gas a transaction will need before it is added to a block. To discourage transactions with a high gas limit parameter (which would take up a lot of the block gas limit) but that consume much less gas in practice, Monad always charges the sender based on the gas limit rather than the gas actually consumed (in other words, it does not refund excess gas).

Monad also uses a custom gas controller to determine the base price per unit of gas. This controller is designed to increase more slowly and decrease more quickly than the gas controller used in Ethereum.

Documentation

- Monad Docs: Gas Pricing
- Category Labs Blog Post: Redesigning a Base Fee for Monad

Past Audits

- Zellic Audit Report - Execution Layer. The gas model received review as part of execution layer assessment, with particular attention to Monad's unique gas pricing mechanisms necessitated by asynchronous execution. While the core gas model logic was reviewed, the interaction between gas pricing, reserve balance enforcement, and Consensus-Execution synchronization may benefit from additional focused analysis.

Reserve Balance

The Reserve Balance mechanism is used to guarantee that if a transaction is included in a block, by the time this block is executed the transaction sender will have enough balance to pay for its gas. This is necessary under asynchronous execution, where the Consensus component does not have an up-to-date view of the chain state, including account balances. In the most basic form of this mechanism, the Execution component guarantees that it will revert a transaction if it would transfer enough funds to bring the sender's balance below a certain reserve amount (currently set to 10 MON), and the Consensus component guarantees that it will not include transactions by the same sender with total gas consumption above this reserve balance (or the sender's balance in the latest available state, whichever is lower). In a specialized safety case, the mechanism can let one transaction (called an "emptying transaction") through even if it would cause the balance to drop below the reserve.

Documentation

- Monad Docs: Reserve Balance
- Monad Initial Specification Proposal

Quality Assurance

- Coq model and proof of safety
  - ✅ Last updated Nov 26 2025. Up-to-date as of Monad v0.12.2.

Past Audits

- Reserve balance mechanism received zero coverage in past audits.

MonadDB

Database used to store the blockchain state on disk after executing a block, implemented directly as a Patricia Trie. Consensus reads from this database to query the delayed state of the chain.

Documentation

- Monad Docs: MonadDB

Quality Assurance

Merkle Patricia Trie

- Category Labs fuzzing
  - ✅ Introduced Jul 16 2025. Up to date.

Past Audits

- Zellic Audit Report - Database Layer. The review covered assessment of MonadDB, examining the database operation safety, asynchronous I/O handling, core I/O utilities, TrieDB components, and WAL mechanisms.

Validator Set

The active validator set is determined by the state of the Staking Contract (specifically, it is composed of the top N validators by stake). However, the Consensus component's view of this state is updated only at the start of each epoch, based on a snapshot taken at the boundary block of the previous epoch. There is a delay between the boundary block and the start of the next epoch to allow time for this snapshot information to be propagated throughout the network. However, given that consensus reads from this database to query delayed chain state, correctness and consistency properties remain necessary for protocol security.

Documentation

- Monad Docs: Staking Behavior

Past Audits

- Zellic Audit Report - Consensus Layer. Validator set synchronization was examined as part of consensus layer audit, with focus on epoch transition mechanisms, snapshot handling at epoch boundaries, and validator set update logic.
- Spearbit Audit Report

RPC Layer

Past Audits

- Zellic Audit Report - RPC Layer. The review covered assessment of the JSON-RPC interface, examining implementation security, access control mechanisms, and resource protection measures.
- Spearbit Audit Report

Compiler Layer

Past Audits

- Zellic Audit Report - Compiler Layer. The review focused on assessment of the JIT compiler implementation, examining IR generation, basic block construction, x86 code generation, and type system correctness. The compiler's security properties are primarily validated through the execution layer's testing infrastructure.

Fuzzing

Methodology

We began the engagement by familiarizing ourselves with the Monad ecosystem at both the design and source-code levels, and by identifying focus areas for our fuzzing efforts. To support this process, we relied on the following sources:

- Monad design documentation and blog posts.
- Documentation from previous audits.
- Code walkthrough calls with the client, during which they answered questions and demonstrated relevant portions of the codebase, particularly the interfaces between the execution and consensus layers.
- Source code from the Monad repositories.

We also requested and gained access to existing fuzzing harnesses implemented by a third party (https://github.com/category-labs/monad-fuzzing-external).

We then conducted a systematic review of the fuzzing harnesses present in the Monad ecosystem (see the next section). Since many system components were already covered by at least one fuzz test, in order to minimize duplication of effort we decided to allocate additional time to reviewing existing fuzzers for high-risk components:

- The Staking precompile: Staking Precompile Fuzzer Review.
- The Monad VM.

Overview of Fuzzing Harnesses

Target	Component	Engine	Scope
`monad/category/staking`	`category/execution/monad/staking`	custom	Staking precompile execution.
`monad/category/mpt/generated_kv`	`category/mpt`	`libFuzzer`	Inserting a key-value pair into a trie.
`monad/category/mpt/one_hundred_updates`	`category/mpt`	`libFuzzer`	One hundred updates to a trie.
`monad/category/statesync/fuzz_statesync`	`category/statesync`	`libFuzzer`	Statesync client-server interaction.
`monad/test/vm/fuzzer`	`category/vm`	custom	Contract deployment and execution.
`monad/test/vm/typechecker_fuzzer`	`category/vm`	custom	Contract type checking.
`monad-fuzzing-external/bft/consensus-types/fuzz_block`	`monad-consensus`	`libFuzzer` via `cargo-fuzz`	Block creation, RLP serialization-deserialization.
`monad-fuzzing-external/bft/consensus-types/fuzz_checkpoint`	`monad-consensus`	`libFuzzer` via `cargo-fuzz`	Checkpoint creation, JSON serialization-deserialization.
`monad-fuzzing-external/bft/consensus-types/fuzz_signature_collection`	`monad-consensus`	`libFuzzer` via `cargo-fuzz`	BLS signature collections.
`monad-fuzzing-external/bft/consensus-types/fuzz_timeout`	`monad-consensus`	`libFuzzer` via `cargo-fuzz`	Timeout creation, RLP serialization-deserialization.
`monad-fuzzing-external/bft/merkle_tree/fuzz_merkle`	`monad-merkle`	`libFuzzer` via `cargo-fuzz`	Merkle tree manipulation and computations.
`monad-fuzzing-external/bft/txpool/eth_txpool`	`monad-eth-txpool`	`libFuzzer` via `cargo-fuzz`	Exercising an ETH transaction pool with transactions through an IPC socket.
`monad-fuzzing-external/execution/int`	`category/vm`	`libFuzzer`	Integer arithmetic.
`monad-fuzzing-external/execution/rlp_datatype`	`category/execution/monad/core/rlp`	`libFuzzer`	RLP protocol buffer.
`monad-fuzzing-external/execution/vm/compile_only`	`category/vm`	`libFuzzer`	Contract compilation.
`monad-fuzzing-external/execution/vm/execution_holistic`	`category/vm`	`libFuzzer`	Comparing interpreted and compiled execution results.
`monad-fuzzing-external/execution/vm/execution_valid_opcodes`	`category/vm`	`libFuzzer`	Comparing interpreted and compiled execution results.
`monad-fuzzing-external/holistic/rlp_block_differential`	`category/execution/ethereum`	`libFuzzer` via `cargo-fuzz`	Comparing block validation results between `monad-bft` and `monad`.
`monad-fuzzing-external/holistic/rlp_transaction_differential`	`category/execution/ethereum`, `monad-eth-block-policy`	`libFuzzer` via `cargo-fuzz`	Comparing transaction validation results between `monad-bft` and `monad`.

Staking Precompile Fuzzer Review

We performed a code review of the Staking precompile fuzzer (commit 57cea41), guided by the `2025-09-04 Staking contract properties` specification. In particular, we focused on the following aspects:

- How are inputs generated?
- What preconditions, as written in the specification, are ensured during input generation?
- What postconditions, as written in the specification, are checked after execution?

For a complete description of these aspects, see Appendix: Staking Precompile Fuzzer.

The specification is detailed and well-written, and overall the code aligns closely with the concepts it defines. In cases where discrepancies arose, this review treated the implementation as the canonical reference. Throughout the review, we assumed the implemented test model to be correct.

The fuzzer itself implements a custom input generation procedure that has been manually tuned for coverage (e.g. it favors boundary values in certain cases). In most cases, the input generator attempts to produce a valid setup and valid inputs (i.e. those that satisfy the preconditions outlined in the specification), so the primary focus of the fuzzer is verifying functional correctness of correct inputs. With the exception of `getDelegator`, view methods are not included in fuzzing.

Suggestions

Overall, the fuzzer does a good job of verifying functional correctness for valid inputs. To complement it and improve code coverage, consider adding an alternative harness which:

- Uses a coverage-guided fuzzing engine (e.g. `libFuzzer`) instead of a custom, manually optimized generator focused primarily on valid inputs.
- Directly generates the environment (validators, delegators, balances, etc.) instead of relying on a sequence of transitions for test setup. To reduce the computational overhead, expensive operations, particularly cryptographic ones such as key pair generation, can be replaced with test doubles.
- Tests all methods, including view methods.
- Tests both positive and negative cases.

Additionally, we prepared a list of discrepancies between the specification and implementation (mostly minor or informational issues, such as typos and overlooked postconditions) and shared it with the Category Labs team.

Adapting Category Labs' Fuzzing Harness to Greybox Fuzzing

The fuzzing harness developed by Category Labs for the Monad compiler sources its randomness with a uint64_t seed from which all of its behavior is derived for an iteration. We wanted to try modifying this to use a random byte array that is consumed as a source of entropy as it is needed throughout the iteration. This allows for a greybox fuzzing harness like libFuzzer (used here) to perform coverage guided fuzzing, modifying specific parts of the execution through specific sections of the byte array.

To achieve this, we changed the testing harness from using a `std::mt19937_64` - which is initialized with an integer seed - to a custom `UnstructuredRandomBitGenerator` class which is initialized with a byte array and consumes bytes from the array to provide an integer. The testing code was then worked into the entry point that libFuzzer expects and then custom compiled into a binary that's instrumented for greybox fuzzing.

One change that was made is known as 'configuration fuzzing', where in this case the probabilities listed in the `GeneratorFocus` for Monad's random EVM instruction generation were randomized. This technique - where the random generation is parameterized and those parameters themselves are randomized - has been known to increase coverage.

This resulted in the target very quickly hitting a failure case that the single-seeded fuzzer failed to discover after hours of runtime. It wasn't clear if this failure was interesting, it was possibly just an invalid bytecode that caused the compiler to fail. Despite this, it demonstrates that configuration fuzzing can help guide the random generation to explore radically different paths and thus hit new coverage faster.

I major benefit provided by libFuzzer is it offers a lot of utility for corpus management (a saved set of inputs that cover all of the source that the target reached), saving and debugging crashing inputs, and replaying the corpus for generating coverage reports. It is our recommendation to consider using this library when creating new fuzzing targets.

Fuzz Runs and Statistics

The following is a table listing each fuzzing target that was run, for how long, and the number of iterations completed within that timeframe.

The `eth_txpool` and `fuzz_statesync` targets are not available because there were issues running them. `eth_txpool` had a tendency to hang, while `fuzz_statesync` required io_uring privileges which were not available in the containerized environment these targets ran in.

A summary of the coverage that all of these targets made over the source can be found in Appendix: Fuzzing Coverage Summary Table

Fuzzer	Iterations	Time
execution-compile-only	281889569	14h
execution-holistic	10192109	14h
execution-valid-opcodes	21799827	14h
monad-compiler (compiler)	2880000	14h
monad-compiler (interpreter)	5760000	14h
monad-staking-contract	5760000	14h
monad-typechecker	9000000	14h
rlp_block_differential	27143415	3h
rlp_transaction_differential	2471185	3h
int	213376861	3h
rlp_datatype	40758412	3h
fuzz_merkle	43628281	3h
fuzz_timeout	1049942	3h
fuzz_signature_collection	1249715	3h
fuzz_checkpoint	1345009	3h
fuzz_block	272026	3h
generated_kv	359476	60s
one_hundred_updates	98738	60s
eth_txpool	N/A	N/A
fuzz_statesync	N/A	N/A

Coq Reserve Balance Proof Overview

We have performed a review of the Coq proof of the Reserve Balance mechanism. A detailed description of the main theorem being proved and the assumptions that the proof depends on can be found in Appendix: Coq Reserve Balance Proof Analysis.

According to Category Labs, the Coq model should be considered the most up-to-date specification of the Reserve Balance mechanism (even taking priority over the Initial Specification Proposal). Since the proof is based on this model, it proves the mechanism correct only under the assumption that the source code implementation is equivalent to this model. With that in mind, we have also conducted a review of the implementation to try to identify discrepancies with the model that could invalidate the proof.

Although under our manual analysis the implementation logic appears to largely adhere to the specification, it is worthwhile to focus additional quality assurance efforts on ensuring that the two are and remain consistent (especially on the side of the Consensus component, which received less focus during this engagement compared with the Execution component). This can include, for example, fuzzing harnesses focused on validating the behavior of the Reserve Balance, ones which currently do not exist. It's also important to ensure that transaction execution remains consistent with the explicit assumptions that the proof depends on (which are also listed in the Appendix).

Observations

- As of Monad v1.12.2, the code has been updated to match the model by not applying reserve balance checks to accounts that had smart contracts deployed to them in the same transaction. This makes the Reserve Balance mechanism more permissive, avoiding a special case that could revert transactions unnecessarily, and is safe under the assumption that a smart contract address cannot also correspond to an EOA.
- The proof accounts for the possibility of reserve-balance-reconfig transactions, which are special transactions that allow altering the reserve balance for a specific account, implemented via a precompile. However, this functionality has not yet been implemented in the latest Monad version, which instead uses a fixed 10 MON reserve balance across all accounts.
- Assumption `execTxCannotDebitNonDelegatedNonContractAccountsCore` depends on the assumption that it's not possible for the balance of an account to overflow. This should be safe considering that the total supply of native tokens is supposed to remain below 2^{256} (and if it doesn't we likely have bigger problems).
- The proof uses unbounded integers when performing the reserve balance calculations. However, it later defines an alternative version of the calculations using the fixed-width `U256` type, and proves its equivalence to the unbounded version. Specifically, it proves that if all inputs are $< 2^{256}$, then either both versions will return the same non-negative result $< 2^{256}$, or the unbounded version will return a negative result and the fixed-width version will return `None` (both of which trigger the transaction to be excluded).
- The implementation of the checks in the Execution component has a hard-coded assumption that `k = 3` (they only check the parent and grandparent blocks for previous transactions or changes in the delegation status of the sender). If the value of `k` ever changes, it's important to ensure that this code is modified appropriately. Ideally, it should be generalized to work for any `k`.
- In the implementation of the mechanism in the Execution component, a sender is allowed to dip into the reserve when the transaction is considered an emptying transaction even if the gas limit of the transaction would exceed the balance of the sender. This is in principle safe because this case should never be allowed by Consensus, but it would be recommended to add an assert to this code path to ensure that it will never happen.
- Compared to the Execution component, the implementation of the Reserve Balance mechanism in the Consensus component is harder to check for equivalence with the Coq model. This is in part due to a slightly higher complexity (due to having to predict future balances under asynchronous execution), the divergence between the more imperative style of the implementation and the more functional style of the model, and other differences in how the mechanism is implemented (for example, the model extends the state with additional information such as the last block when each sender has submitted a transaction, while the implementation calculates this information dynamically). With that in mind, additional measures to ensure the Consensus implementation is aligned with the model deserve attention.

Token Flow Summary

We have performed an analysis of the different ways that the operation of the Monad protocol can modify account balances via minting, transferring, or burning native tokens (MON). By understanding how and when changes in account balances take place, we can develop confidence that balance adjustments are calculated correctly and can't be exploited by attackers, and that the token supply remains consistent.

A full breakdown can be found in Appendix: Token Flow Analysis, but the following is a summary of the ways account balances can change:

- Tokens are minted to the Staking Contract by the `syscallReward` system call (which is included on every block) and stored within the contract as rewards to the block author and its delegators.
- The validator and its delegators can claim these rewards, getting them transferred from the Staking Contract to EOAs. They can also undelegate previously-delegated stake and similarly withdraw them from the Staking Contract.
- Transaction senders are charged the gas limit for their transactions. Part of this cost is transferred to the block builder as priority fees, while the rest is burned.
- Tokens can be transferred between accounts during the execution of calls or contract deployments.
- The balance of a smart contract can also be transferred or burned if the contract self-destructs.

To validate that balance adjustments always happen as intended and token supply doesn't diverge, it is worthwhile to consider automated approaches, such as differential fuzzing or real-time monitoring. The former can be done as an extension of fuzzing harnesses over transaction execution, by including checks that ensure that only expected balance changes occur. The latter can be done by implementing a node that tracks all events that produce balance changes, and keeps track of the total token supply in order to ensure that it remains consistent with the events observed.

Findings

This section contains all issues identified during the audit that could lead to unintended behavior, security vulnerabilities, or failure to enforce the protocol's intended logic. Each issue is documented with a description, potential impact, and recommended remediation steps.

A01: Unauthenticated RPC Endpoints Expose Privileged Operations

Severity: Medium | Difficulty: High | Recommended Action: Fix Design | Addressed by client

The Monad RPC server binds to all network interfaces (`0.0.0.0`) on port 8080 by default and exposes debug, admin, and transaction submission methods without any authentication mechanism. The `rpc_select` function processes all incoming requests without verifying caller identity or authorization level, allowing any network-accessible client to invoke privileged methods.

Anyone with network access can call debug methods (`debug_getRawBlock` , `debug_traceTransaction`), admin methods (`admin_ethCallStatistics`), and transaction submission methods (`eth_sendRawTransaction`). This allows attackers to dump internal state, trace transactions, and spam the transaction pool. While global rate limiting exists for `eth_sendRawTransaction` , there is no per-client limiting, so a single IP can still flood the endpoint.

Notes

This finding was identified with the assistance of the Almanax tool.

Recommendation

1. Change default binding: Update `monad-bft/monad-rpc/src/cli.rs:32` to bind to `127.0.0.1` instead of `0.0.0.0` .
2. Implement method-level authorization in the `rpc_select` function.
3. Implement per-IP and per-credential rate limiting for `eth_sendRawTransaction` .
4. Add a configuration option to disable debug methods in production builds.
5. Document security model and deployment guidelines for different node types.

Auditor Notes

No auth middleware anywhere. Default binding is `0.0.0.0` , which exposes everything. This is exploitable if the RPC port is reachable from the network. While operational mitigations (such as nginx proxies) exist in practice, these aren't code-level protections.

Recommendations 2 and 3 (method-level authorization and per-IP rate limiting) are mitigated by operational practices, as RPC operators deploy nginx proxies that provide method filtering and rate limiting at the infrastructure layer rather than implementing these controls within the RPC server itself.

Client response

The team acknowledges both configuration issues in the RPC server. Zellic previously reported this as a conformance issue with Geth. In practice, RPC operators deploy nginx proxies in front of their nodes to distribute requests and limit exposed RPC methods. The team agrees that changing the default binding from `0.0.0.0` to localhost is appropriate and will match Geth's standard of disabling debug endpoints by default in production.

Status

Acknowledged by Category Labs, but ultimately the use of proxies has been considered sufficient mitigation.

A02: Panic-Based Error Handling Enables Denial of Service

Severity: Low | Difficulty: Low | Recommended Action: Fix Code | Addressed by client

The consensus layer block policy implementation contains an assertion in `check_coherency` that could theoretically panic. The assertion `assert_eq!(first_block.get_seq_num(), self.last_commit + SeqNum(1))` checks that the first block extends `last_commit + 1`, but it executes before proper error handling can return `Err(BlockPolicyError::BlockNotCoherent)`.

There are three panic locations identified:

1. `assert_eq!` (in `check_coherency`) checking that first block extends `last_commit + 1`.
2. `unwrap_or_else` on sequence numbers in `CommittedBlkBuffer`.
3. `assert_eq!` with panic when retrieving committed blocks from cache.

The panics at 2 and 3 are protected by invariant maintenance. The assertion at 1 is not exploitable in practice because `extending_blocks` is always non-empty when the network is live. After startup, a QC-of-QC commits a block, which becomes the root of the block tree, preventing attackers from creating valid TCs that extend the root block.

Recommendation

- Remove the assertion in `check_coherency` or convert it to proper error handling:

```
if first_block.get_seq_num() != self.last_commit + SeqNum(1) {
    warn!("Block seq_num mismatch: expected {}, got {}",
          self.last_commit + SeqNum(1), first_block.get_seq_num());
    return Err(BlockPolicyError::BlockNotCoherent);
}
```

Or remove the assertion entirely since this line already validates sequence numbers.

Client response

The team confirmed that the assertion at 1 can only be tripped by an adversary at startup. After startup, `extending_blocks` is always non-empty because a QC-of-QC commits a block, which becomes the root of the block tree. Once that happens, an attacker can't create a valid TC that extends the root block. Additionally, `extending_blocks` have a QC over them, so their `seq_nums` have already been validated. Therefore, the assertion cannot be triggered during normal operation.

Status

The assertion has been removed in PR #2704.

A03: Empty Validator Set Causes Division by Zero

Severity: Low | Difficulty: Low | Recommended Action: Fix Code | Addressed by client

The `get_leader` function performs `round.0 as usize % validators.len()` without checking if the validator set is empty. If `validators.len()` is zero, this panics. An attacker who can trigger validator set bugs (through delegation/slashing issues) could cause this.

Notes

This finding was identified with the assistance of the Almanax tool.

Auditor Notes

While initialization prevents empty sets through an assertion at config load time:

```
assert!(!validators_config.validator_sets.is_empty());
```

However, there's no runtime protection against potential bugs in epoch transition logic or validator set management that could empty the set at runtime.

For best practice, we suggest adding runtime checks to account for potential future bugs in epoch transitions or validator set management.

Recommendation

- Add adefensive check for empty validator sets and change the return type to propagate errors.

```
fn get_leader(
    &self,
    round: Round,
    _epoch: Epoch,
    validators: &BTreeMap<NodeId<Self::NodeIdPubKey>, Stake>,
) -> Result<NodeId<PT>, LeaderElectionError> {
    if validators.is_empty() {
        error!("INVARIANT VIOLATION: Empty validator set in leader election");
        return Err(LeaderElectionError::EmptyValidatorSet);
    }

    let validators: Vec<_> = validators.iter().map(|(node_id, _)| node_id).collect();
    Ok(*validators[round.0 as usize % validators.len()])
}
```

Alternatively, add a check in `get_leader()` to return an error if the validator set is empty.

Status

The Category Labs team acknowledges that empty validator sets are invalid and should trigger an abort. To prevent this case, PR #2682 modifies the `ValidatorSetFactory::create` method to return an error if trying to create an empty validator set.

A04: Unbounded Block Sync Request Sizes

Severity: Medium | Difficulty: Low | Recommended Action: Fix Code | Partially addressed by client

The block sync request handler accepts requests without validating the range size. While TCP-level rate limiting exists, there was no per-request SIZE validation. An attacker can request millions of blocks in a single request, consuming significant memory and bandwidth. The TCP rate limiting controls request rate (requests per second), not request size (blocks per request). There's also no limit on concurrent requests, so `headers_requests` and `payload_requests` maps can grow unbounded.

Recommendation

- Add limits on `headers_requests` and `payload_requests` maps to prevent unbounded growth from multiple simultaneous requests:

```
const MAX_CONCURRENT_SYNC_REQUESTS: usize = 10;

fn handle_peer_request(&mut self, request: BlockSyncRequestMessage) -> Result<(), SyncError> {
    if self.headers_requests.len() + self.payload_requests.len() >= MAX_CONCURRENT_SYNC_REQUESTS {
        return Err(SyncError::TooManyRequests);
    }
}
```

- Add per-peer rate limiting to prevent individual peers from overwhelming the sync subsystem, even with the 800 block limit in place.

Client response

The client confirmed that `BLOCKSYNC_MAX_NUM_HEADERS` limits the request size to 800 blocks. Additionally, a commit drops any blocksync requests that can hit the disk after the block cache is hydrated with 1000 blocks, making handling blocksync requests cheap.

Auditor Notes

The client has addressed the primary concern: `BLOCKSYNC_MAX_NUM_HEADERS` limit (800 blocks). However, recommended enhancements remain: concurrent request limits and per-peer rate limiting for additional production hardening.

Status

Primary Concern Addressed. The Category Labs team is aware of blocksync and statesync DoS issues and is planning systematic mitigations. Currently, the client relies on operators banning attacker IPs.

A05: Unbounded Map Growth in State Sync

Severity: Medium | Difficulty: Low | Recommended Action: Fix Code | Not addressed by client

The state sync implementation maintains two unbounded HashMaps (`prefix_peers` and `seen_nonces`) without size limits or eviction policies. An attacker could send requests with many distinct prefix values or nonces, causing unbounded memory growth. Even without malicious intent, normal operation over prolonged periods could cause gradual memory growth.

Recommendation

- Implement LRU eviction policies for both maps:

```
use lru::LruCache;

const MAX_PREFIX_PEERS: usize = 1000;
const MAX_SEEN_NONCES: usize = 10000;

struct OutboundRequests {
    prefix_peers: LruCache<u64, NodeId<PT>>,
    seen_nonces: LruCache<u64, Instant>,
}
```

- Implement time-based expiration for `seen_nonces` .

Status

The Category Labs team confirmed awareness of DoS attacks on statesync and is working on mitigations. The hashmap growth is minor because it occurs on the client side, which is not live in the network yet. The client will consider hashmap growth limits as part of systematic mitigations.

A06: Database Credentials Logged in Plain Text

Severity: Medium | Difficulty: Low | Recommended Action: Fix Code | Addressed by client

The RPC server logs the connection URL at info level during initialization without sanitizing embedded credentials.

```
info!(url, db_name, "Initializing MongoDB archive reader");
```

If the URL contains embedded credentials (e.g., `mongodb://username:password@host/database`), these credentials are written to log files in plain text, which may be collected by centralized logging systems or included in debug bundles.

Recommendation

Sanitize URLs before logging by redacting the authentication portion:

```
fn sanitize_url(url: &str) -> String {
    if let Some(at_pos) = url.find('@') {
        if let Some(scheme_end) = url.find("://") {
            return format!("{}://***@{}", &url[..scheme_end], &url[at_pos + 1..]);
        }
    }
    url.to_string()
}

info!(url = %sanitize_url(&url), "Initializing MongoDB connection");
```

Alternatively, use structured logging with automatic redaction or consider avoiding logging of connection strings entirely.

Status

PR #2681 adds a `redact_mongo_url` function, using it to sanitize MongoDB connection strings when these are logged.

Informative Findings

This section includes observations that are not directly exploitable, but highlight areas for improvement in code clarity, maintainability, or best practices. While not critical, addressing these can strengthen the system's overall robustness.

B01: Missing Block Gas Limit Validation

Severity: Informative | Recommended Action: Fix Code | Not addressed by client

The execution layer's block validation function

runtimeverification/_audits_category-labs_monad/category/execution/ethereum/validate_block.cpp

Line 239 to 247 in a5880d8

```
239 template <Traits traits>
240 Result<void> static_validate_block(Block const &block)
241 {
242     BOOST_OUTCOME_TRY(static_validate_header<traits>(block.header));
243
244     BOOST_OUTCOME_TRY(static_validate_body<traits>(block));
245
246     return success();
247 }
```

does not verify that the cumulative sum of transaction gas limits within a block remains within the protocol-specified block gas limit.. However, the consensus layer performs this validation before blocks reach execution.

Notes

This finding was identified with the assistance of the Almanax tool.

Auditor's Notes

The `static_validate_block` function is called during block processing. This function validates the header and body but does not check that the sum of transaction gas limits (`Σ tx.gas_limit`) is within the block gas limit. However, since consensus already enforces this constraint and execution cannot receive unvalidated blocks; therefore, this is not exploitable.

Recommendation

For best practice, add validation in the execution layer, but this is not required since this constraint is already enforced by consensus, and execution cannot receive unvalidated blocks.

B02: Per-Transaction Gas Limit Not Validated in Execution Layer

Severity: Informative | Recommended Action: Fix Code | Not addressed by client

The execution layer does not validate that the individual transaction gas limits remain within the 30-million-gas limit. However, the consensus layer performs this validation where `TfmValidator` checks `if tx.gas_limit() > TFM_MAX_GAS_LIMIT` and rejects transactions exceeding this limit.

```
    fn validate(
        tx: &TxEnvelope,
        chain_params: &ChainParams,
        execution_chain_params: &ExecutionChainParams,
    ) -> Result<(), StaticValidationError> {
        if execution_chain_params.tfm_enabled {
            // tfm-specific gating
            if tx.gas_limit() > TFM_MAX_GAS_LIMIT {
                return Err(StaticValidationError::GasLimitOverTFMGasLimit {
                    tx_gas_limit: tx.gas_limit(),
                    tfm_max_gas_limit: TFM_MAX_GAS_LIMIT,
                });
            }
        }
```

Notes

This finding was identified with the assistance of the Almanax tool.

Auditor Notes

This is not exploitable because consensus validates the per-transaction gas limit before blocks reach execution, and execution has no network access. Also, consensus rejects transactions exceeding 30M gas limit at validation.

Recommendation

Add validation in execution layer for best practice, but not required.

B03: Upfront Payment Uses `max_fee` Instead of `gas_bid`

Severity: Informative | Recommended Action: Fix Code | Not addressed by client

The execution layer's balance validation uses `max_fee * gas_limit` instead of the specification-required `gas_bid * gas_limit` formula, where `gas_bid = min(priority_fee + base_fee, max_fee)`. This causes transactions to reserve a higher balance amount than necessary upfront.

runtimeverification/_audits_category-labs_monad/category/execution/ethereum/validate_transaction.cpp

Line 211 to 219 in a5880d8

```
211 Result<void> validate_transaction(
212     Transaction const &tx, std::optional<Account> const &sender_account,
213     std::span<uint8_t const> code)
214 {
215     // YP (70)
216     uint512_t v0 = tx.value + max_gas_cost(tx.gas_limit, tx.max_fee_per_gas);
217     if (tx.type == TransactionType::eip4844) {
218         v0 += tx.max_fee_per_blob_gas * get_total_blob_gas(tx);
219     }
```

Notes

This finding was identified with the assistance of the Almanax tool.

Auditor Notes

If `priority_fee + base_fee < max_fee`, the specification requires only `(priority_fee + base_fee) * gas_limit` to be reserved, but execution layer reserves `max_fee * gas_limit`. However, the consensus layer correctly uses the `gas_bid` calculation. Since consensus validates transactions before execution receives blocks, this is not exploitable but represents a specification compliance issue.

Recommendation

Update the execution layer's balance validation logic to use `gas_bid` instead of `max_fee` as specified in the protocol, matching the consensus layer's calculation.

```
// Calculate gas_bid = min(priority_fee + base_fee, max_fee)
uint256_t priority_fee = tx.max_priority_fee_per_gas().unwrap_or(0);
uint256_t base_fee = get_base_fee_per_gas(block_header);
uint256_t gas_bid = std::min(max_fee, base_fee + priority_fee);
uint512_t v0 = tx.value + gas_bid * tx.gas_limit;
```

B04: Validator Reward Calculated on `gas_used` Instead of `gas_limit`

Severity: Informative | Addressed by client

The validator reward calculation uses `gas_used * priority_fee`

runtimeverification/_audits_category-labs_monad/category/execution/ethereum/transaction_gas.cpp

Line 209 to 218 in a5880d8

```
209 template <Traits traits>
210 uint256_t calculate_txn_award(
211     Transaction const &tx, uint256_t const &base_fee_per_gas,
212     uint64_t const gas_used) noexcept
213 {
214     if constexpr (traits::evm_rev() < EVMC_LONDON) {
215         return gas_used * gas_price<traits>(tx, base_fee_per_gas);
216     }
217     return gas_used * priority_fee_per_gas(tx, base_fee_per_gas);
218 }
```

instead of the specification-required `gas_limit * priority_fee` as in Section 2.5:

```
Reward_k = Σ_t ∈ block (gas_bid_t - base_fee_k) · gas_limit_t
```

This simplifies to `priority_fee_t * gas_limit_t` (since `gas_bid - base_fee = priority_fee`). This results in validators receiving less reward than specified, potentially affecting economic incentives.

runtimeverification/_audits_category-labs_monad/category/execution/ethereum/execute_transaction.cpp

Line 366 to 368 in a5880d8

```
366     auto const reward = calculate_txn_award<traits>(
367         tx_, header_.base_fee_per_gas.value_or(0), gas_used);
368     state.add_to_balance(header_.beneficiary, reward);
```

Recommendation

Update the reward calculation to use `gas_limit` as specified in the protocol. If paying on `gas_used` is intentional, it may be for better incentive alignment (rewarding actual computation versus reserved capacity), but this should be documented in the specification.

Status

Confirmed as a non-issue. Refunds are zeroed out when calculating validator rewards, so `gas_used = gas_limit` at that point, matching the specification requirement.

B05: Reserve Balance Documentation Suggestions

Severity: Informative | Recommended Action: Document Prominently | Not addressed by client

- The definition of "emptying transaction" is ambiguous in the documentation, compared to the Monad Initial Spec. The definition in the documentation seems to suggest that it's any transaction that, if executed, might reduce the balance below the reserve balance. However, the spec is clear that it's meant to refer to transactions that *are allowed* by the execution component to lower the balance below the reserve balance. In other words, an emptying transaction is one that satisfies the two conditions that allow it to bypass reserve balance checks in execution: the sender is not a delegated account and this is the first transaction by that sender in the past `k` blocks. The documentation should be updated to clarify this.
- When describing the simple version of the Reserve Balance mechanism, the documentation says that "execution reverts any transaction that causes an account's balance to dip below `user_reserve_balance`, except due to transaction fees". It's unclear how to interpret "except due to transaction fees", since both gas spend (transaction fees) and execution spend contribute to reducing the balance. In practice, it means that a transaction should revert if its execution spend would independently be sufficient to reduce the balance below the reserve balance, prior to accounting for any gas spend. This would be good to clarify. It's also worth noting that this includes the case when the balance is already below the reserve balance and execution spend is greater than 0.

B06: Predictable Peer Selection Due to Hardcoded RNG Seed

Severity: Informative | Recommended Action: Fix Code | Not addressed by client

The block sync module uses a hardcoded RNG seed, which initializes ChaCha8Rng with a hardcoded seed value of 123456 (`seed_from_u64(123456)`).

Recommendation

This isn't exploitable in practice: each node's RNG state diverges quickly since it mutates with each `pick_peer()` call, and predicting peer selection would require tracking the exact RNG state (knowing all previous calls and timing). However, we recommend using entropy from the operating system: `rng: ChaCha8Rng::from_entropy()` to ensure nodes have independent peer selection patterns and prevent any theoretical attack scenarios.

Status

The client has acknowledged the observation and committed to addressing it.

B07: Task Panics Propagate to Parent

Severity: Informative | Recommended Action: Fix Code | Addressed by client

The transaction pool future implementation uses `JoinHandle::unwrap()` to retrieve results from background tasks. If a background task panics, the unwrap propagates the panic to the RPC server, potentially causing it to crash. This violates fault isolation, where background task failures should be logged and handled without affecting the main service.

```
impl Future for EthTxPoolBridgeHandle {
    type Output = ();

    fn poll(mut self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output> {
        let Poll::Ready(result) = self.handle.poll_unpin(cx) else {
            return Poll::Pending;
        };

        Poll::Ready(result.unwrap())  // Line 43: Panic propagates if task panicked
    }
}
```

The background task only forwards transactions via IPC and handles errors, so a panic would require an internal bug, not external input. This is more of a code quality issue.

Recommendation

- Handle background task failures gracefully.
- Implement supervision with restart limits to prevent restart loops while allowing recovery from failures.

Status

PR #2717 modifies `EthTxPoolBridgeHandle.poll(...)` to propagate the error. Category Labs has also noted that the `EthTxPoolBridgeHandle` was never actually polled, so background task panics would be silently ignored by the RPC.

B08: Unbounded Debug Trace Operations Enable Resource Exhaustion

Severity: Informative | Recommended Action: Fix Code | Addressed by client

The RPC server exposes debug trace methods (`debug_traceTransaction` , `debug_traceBlockByHash` , `debug_traceBlockByNumber`) that perform computationally expensive operations without per-IP rate limiting. These methods re-execute transactions with full EVM tracing, recording every opcode execution, state access, and gas consumption.

While configuration options exist to limit concurrent trace operations globally (`--eth_trace_block_max_concurrent_requests`), there is no per-IP rate limiting. An attacker can send multiple trace requests concurrently from the same source, potentially consuming all available execution threads and causing a denial of service.

Recommendation

Implement per-IP rate limiting for trace operations.

Client response

The client confirmed this issue was reported and tracked in Debug namespace methods enabled by default without authentication, and a patch is available in PR #2362.

Auditor Notes

With PR #2362, debug methods are disabled by default, significantly reducing the attack surface. However, when debug methods are enabled (for trusted operators), the vulnerability remains: the global rate limit (default: 20 concurrent requests) applies across all IPs, not per IP. A single attacker can exhaust all available execution threads by sending multiple concurrent requests from one IP address.

The concern about per-IP rate limiting remains when debug methods are enabled (for trusted operators). However, in production all requests are expected to be routed through proxies, which are expected to handle IP rate limiting.

Appendix: Staking Precompile Fuzzer

Fuzzing process

Execute transitions (`StakingContractMachine::Transition`) on a machine (`StakingContractMachine`).

Machine initialization:

- 0.1% Insert 199 (`ACTIVE_VALSET_SIZE - 1`) active validators (close to maximum active)
- 0.1% Insert 260 (`max_initial_active_vals`) active validators (to fill up a bitset bucket)

Transition generation:

- 20% `syscallReward(address)`
- 10% `externalReward(uint64)`
- 70% the following with uniform distribution
  - Syscalls:
    - `syscallOnEpochChange(uint64)`
    - `syscallSnapshot()`
    - `syscallReward(address)`
  - External state-modifying methods:
    - `addValidator(bytes,bytes,bytes)`
    - `delegate(uint64)`
    - `undelegate(uint64,uint256,uint8)`
    - `compound(uint64)`
    - `withdraw(uint64,uint8)`
    - `claimRewards(uint64)`
    - `changeCommission(uint64,uint256)`
    - `externalReward(uint64)`
  - External view methods:
    - `getDelegator(uint64,address)`

The following external view methods are not exercised:

- `getValidator(uint64)`
- `getWithdrawalRequest(uint64,address,uint8)`
- `getConsensusValidatorSet(uint32)`
- `getSnapshotValidatorSet(uint32)`
- `getExecutionValidatorSet(uint32)`
- `getDelegations(address,uint64)`
- `getDelegators(uint64,address)`
- `getEpoch()`

After each transition, there's a 20% chance that `WITHDRAWAL_DELAY` number of epochs are skipped (so that withdrawal requests are ready).

`syscallOnEpochChange` transition

Setup

If `epoch = 0`, then let `next_epoch` in `1..1,000,000` (with uniform distribution). Call `syscall_on_epoch_change(next_epoch)`.

Otherwise, if `in_epoch_delay_period`, call `syscall_snapshot()`. Let `next_epoch = epoch + 1`. Call `syscall_on_epoch_change(next_epoch)`.

Preconditions

- `next_epoch > epoch`
- Moreover, if `epoch /= 0`, then
  - `in_epoch_delay_period`
  - `next_epoch = epoch + 1`

Postconditions

- Not `in_epoch_delay_period`
- `epoch = next_epoch`
- `active_consensus_stake(v) = consensus_view(v).stake`

The following postconditions are not explicitly checked:

- `active_consensus_commission(v) = consensus_view(v).commission`

`syscallSnapshot` transition

Setup

If `epoch = 0`, do nothing.

Otherwise, if the system is `in_epoch delay period`, call `syscall_on_epoch_change(epoch + 1)`. Call `syscall_snapshot()`.

Preconditions

- Not `in_epoch_delay_period`
- `epoch /= 0`

Postconditions

- `in_epoch_delay_period`
- For all `v` in `valset_execution`, `val_execution(v).stake >= ACTIVE_VALIDATOR_STAKE`.
- Let `u not in valset_consensus` and `v in valset_consensus`, `val_u = val_execution(u)`, `val_v = val_execution(v)`, and `del_u = delegator(u, val_u.auth_address)`.
  Then if `val_u.stake > val_v.stake`, then
  `del_u.stake + del_u.delta_stake + del_u.next_delta_stake < MIN_VALIDATE_STAKE`.

`syscallReward` transition

Setup

Pick a validator `v` from `active_consensus_view` (i.e. if `in_epoch_delay_period` then from `valset_snapshot`, else from `valset_consensus`). If such a `v` does not exist, return. Pick a random reward value `reward` between 0 and 1,000,000 MON (favoring boundary values). Call `syscall_reward(v, reward)`.

Preconditions

- `active_consensus_stake(v) > 0`

Postconditions

- `error_bound = old(error_bound) + 1`
- Let `commission = reward * active_consensus_commission(v) / MON`, `d = delegator_stake(v, a, epoch) + withdrawal_stake(v, a, epoch)`, `rpt = (reward - commission) * UNIT_BIAS / active_consensus_stake(v)`, `r = d * rpt`, then
  - if `a = val_execution(v).auth_address`, then `unit_bias_rewards(v, a) = old(unit_bias_rewards(v, a)) + r + commission * UNIT_BIAS`,
  - otherwise, `unit_bias_rewards(v, a) = old(unit_bias_rewards(v, a)) + r`.
  - Moreover, `val_execution(v).unclaimed_rewards = old(val_execution(v).unclaimed_rewards) + reward - commission`.

`addValidator` transition

Setup

Generate input data `(msg, secp_sig, bls_sig, signer, sender, value)` where

- `msg` is `(secp_pubkey, bls_pubkey, auth_address, stake, commission)`, encoded (see fuzzer implementation for the details), where
  - `secp_pubkey` is the SECP public key belonging to a secret key `secp_seckey`,
  - `bls_pubkey` is the BLS public key belonging to a secret key `bls_seckey`,
  - `auth_address` is either a new or an old address,
  - `MIN_VALIDATE_STAKE <= stake <= MAX_STAKE`,
  - `0 <= commission <= MAX_COMMISSION`,
- `secp_sig` is `msg` signed with `secp_seckey`,
- `bls_pubkey` is `msg` signed with `bls_seckey`,
- `signer = address_from_secpkey(secp_pubkey)`,
- `sender` is either a new or an old address,
- `value = stake`.

Call `v = precompile_add_validator(signer, msg, secp_sig, bls_sig, sender, value)`.

Preconditions

- `stake = amount`
- `amount >= MIN_VALIDATE_STAKE >= DUST_THRESHOLD`
- `secp_pubkey` and `secp_seckey` is a SECP keypair, `secp_sig` is `msg` signed with `secp_seckey`
- `bls_pubkey` and `bls_seckey` is a BLS keypair, `bls_sig` is `msg` signed with `bls_seckey`
- `commission <= MAX_COMMISSION`

The following preconditions are not explicitly ensured or checked:

- `balance_of(STAKING_CA) + amount` does not overflow
- `last_val_id + 1` does not overflow
- `amount <= UNIT_BIAS`
- `val_id(address_from_secpkey(secp_pubkey)) = 0`
- `val_id_bls(address_from_bls_key(bls_pubkey)) = 0`

Postconditions

- `balance_of(STAKING_CA) = old(balance_of(STAKING_CA)) + value`
- `last_val_id = old(last_val_id) + 1`
- Let `val = val_execution(v)` .
  - `val.keys.secp_pubkey = secp_pubkey`
  - `val.keys.bls_pubkey = bls_pubkey`
  - `val.commission = commission`
  - `val.stake = stake`
- If `in_epoch_delay_period` then `delegator_stake(v, auth_address, e) = stake` for `epoch + 2 <= e < epoch + 5`
- If not `in_epoch_delay_period` then `delegator_stake(v, auth_address, e) = stake` for `epoch + 1 <= e < epoch + 5`
- If `v in valset_execution` then `stake >= ACTIVE_VALIDATOR_STAKE`
- `v in get_validators_for_delegator(auth_address)`

The following postconditions are not explicitly checked:

- `v = last_val_id`

`delegate` transition

Setup

Generate input data `(v, sender, amount)` where either

- `sender` is either a new or an old address,
- `v` is either an existing validator ID such that `val_execution(v).stake <= MAX_DELEGABLE_STAKE` , or a new validator created with a process similar to the `addValidator` transition,
- `MIN_DELEGATE_STAKE <= amount <= MAX_STAKE - val_execution(v).stake` ,

or

- `sender` is either a new or an old address,
- `v` is an arbitrary, existing or unknown validator ID,
- `amount = 0` .

Call `r = precompile_delegate(v, sender, amount)` .

Preconditions

- If `amount /= 0` then `val_execution(v) /= 0`
- If `amount /= 0` then `amount >= MIN_DELEGATE_STAKE >= DUST_THRESHOLD`
- `val_execution(v).stake + amount <= MAX_STAKE <= UNIT_BIAS`

The following preconditions are not explicitly ensured or checked:

- `balance_of(STAKING_CA) + amount` does not overflow

Postconditions

- `r = 1`

Moreover, if `amount /= 0` :

- `balance_of(STAKING_CA) = old(balance_of(STAKING_CA)) + amount`
- If `in_epoch_delay_period` , then `delegator_stake(v, sender, e) = old(delegator_stake(v, sender, e)) + amount` for `epoch + 2 < e < epoch + 6`

- If not `in_epoch_delay_period`, then `delegator_stake(v, sender, e) = old(delegator_stake(v, sender, e)) + amount` for `epoch + 1 < e < epoch + 6`
- `val_execution(v).stake = old(val_execution(v).stake) + amount`
- Let `val = val_execution(v)` and `del = delegator(v, val.auth_address)`. If `val.stake >= ACTIVE_VALIDATOR_STAKE` and `del.stake + del.delta_stake + del.next_delta_stake >= MIN_VALIDATE_STAKE`, then `v in valset_execution`.
- `sender in get_delegators_for_validator(v)`
- `error_bound = old(error_bound) + 3`

`undelegate` transition

Setup

Generate input data `(v, stake, wid, sender, value)`:

- Pick a pair `(v, sender)` such that address `sender` delegates to validator `v`. If there are not validators, add one (`precompile_add_validator`).
- Pick an available withdrawal ID `wid` for `(v, sender)`. If there's none, add a new `v` and pick the corresponding `sender`.
- Generate `stake` between `0` and `delegator(v, sender).stake`.
- Let `value = 0`.

Call `r = precompile_undelegate(v, stake, wid, sender, value)`.

Preconditions

- `value = 0`
- `stake <= delegator(v, sender).stake`
- `withdrawal_request(v, sender, wid).amount = 0` (i.e. `wid` is available).

The following preconditions are not explicitly ensured or checked:

- `epoch + WITHDRAWAL_DELAY + 2` does not overflow

Postconditions

- `r = 1`

Let `del = delegator(v, sender)`. Let `wreq = withdrawal_request(v, sender, wid)`. Assume `stake > 0`.

- `wreq.epoch = epoch + 2` if `old(in_epoch_delay_period)`
- `wreq.epoch = epoch + 1` if not `old(in_epoch_delay_period)`
- Let `sigma = del.stake + sigma1 + sigma2` where
  - `sigma1 = del.delta_stake` if `del.delta_epoch <= epoch` else `sigma1 = 0`
  - `sigma2 = del.next_delta_stake` if `del.next_delta_epoch <= epoch` else `sigma2 = 0`

  Let `effective_stake = sigma` if `sigma - stake < DUST_THRESHOLD`, else `effective_stake = stake`.
  Then `wreq.amount = old(effective_stake)`.
- For `epoch <= e < wreq.epoch`, `withdrawal_stake(v, sender, e) = old(withdrawal_stake(v, sender, e)) + effective_stake`.
- For `wreq.epoch <= e < old(epoch) + 6`, `delegator_stake(v, sender, e) = old(delegator_stake(v, sender, e)) - effective_stake`.
- If `sigma = 0`, then `v not in get_validators_for_delegator(sender)`.
- `error_bound = old(error_bound) + 3`.

The following postconditions are not explicitly checked:

- Let `val = val_execution(v)`. If `val.stake >= ACTIVE_VALIDATOR_STAKE` and `old(del.stake + del.delta_stake + del.next_delta_stake) >= MIN_VALIDATE_STAKE`, then `v in valset_execution`.

`compound` transition

Setup

Generate input data `(v, sender, value)`:

- With a small probability, pick an arbitrary pair `(v, sender)`. Let `val = val_execution(v)` and `del = delegator(v, sender)`.
  Let `rewards = del.reward + unaccumulated_rewards(v, sender)`. If `val.stake + rewards <= MAX_STAKE`, and either `rewards = 0` or `rewards >= MIN_DELEGATE_STAKE`,
  pick `(v, sender, 0)` as input data.
- Otherwise, try ten times to find a suitable delegator.
  Pick a pair `(v, sender)` such that address `sender` delegates to validator `v`. If there are not validators, add one (`precompile_add_validator`).
  Let `rewards = del.reward + unaccumulated_rewards(v, sender)`.
  If `rewards = 0`, pick `(v, sender, 0)` as input.
  If `rewards >= MIN_DELEAGET_STAKE` and `val.stake + rewards <= MAX_STAKE`, pick `(v, sender, 0)` as input.

Call `r = precompile_compound(v, sender, value)`.

Preconditions

- `amount = 0`
- Let `rewards = delegator(v, sender).reward + unaccumulated_rewards(v, sender)`.
  - `rewards = 0` or `rewards >= MIN_DELEAGATE_STAKE >= DUST_THRESHOLD`.
  - `val_execution(v).stake + rewards <= MAX_STAKE <= UNIT_BIAS`.

Postconditions

- `r = 1`

Let `val = val_execution(v)`. Let `del = delegator(v, sender)`. Let `rewards = del.reward + unaccumulated_rewards(v, sender)`. Assume `rewards > 0`.

- If `old(in_epoch_delay_period)`, for `old(epoch) + 2 <= e < old(epoch) + 6`, `delegator_stake(v, sender, e) = delegator_stake(v, sender, e) + rewards`.
- If `not old(in_epoch_delay_period)`, for `old(epoch) + 1 <= e < old(epoch) + 6`, `delegator_stake(v, sender, e) = delegator_stake(v, sender, e) + rewards`.
- `val.stake = old(val.stake) + rewards`.
- Let `auth_del = delegator(v, val.auth_address)`. If `val.stake >= ACTIVE_VALIDATOR_STAKE` and `auth_del.stake + auth_del.delta_stake + auth_del.next_delta_stake >= MIN_VALIDATE_STAKE`, then `v in valset_execution`.
- `error_bound = old(error_bound + 3)`
- `unit_bias_rewards(v, sender) = old(unit_bias_rewards(v, sender)) - rewards * UNIT_BIAS`

`withdraw` transition

Setup

Generate input `(v, wid, sender, amount)`:

- If there are no withdrawal requests, perform an `undelegate` transition.
- Pick a withdrawal `wit = withdrawal_reqeust(v, sender, wid)`.
- If `epoch < wit.epoch + WITHDRAWAL_DELAY`, skip `wit.epoch + WITHDRAWAL_DELAY - epoch` epochs.
- Take `(v, wid, sender, 0)` as input.

Call `precompile_withdraw(v, wid, sender, amount)`.

Preconditions

- `amount = 0`
- `withdrawal_request(v, sender, wid).amount > 0`
- `withdrawal_request(v, sender, wid).epoch + WITHDRAWAL_DELAY <= epoch`

The following preconditions are not explicitly checked or ensured:

- `epoch + WITHDRAWAL_DELAY` does not overflow
- `balance_of(sender) + withdrawal_request(v, sender, wid).amount + withdrawal_reward(v, sender, wid)` does not overflow

Postconditions

Let `w_reward = old(withdrawal_reward(v, sender, wid))`. Let `reward = old(withdrawal_request(v, sender, wid).amount) + w_reward`.

- `balance_of(STAKING_CA) = old(balance_of(STAKING_CA)) - reward`.
- `balance_of(sender) = old(balance_of(sender)) + reward`.
- Let `val = val_execution(v)`. `val.unclaimed_rewards = old(val.unclaimed_rewards) - w_reward`
- `error_bound = old(error_bound) + 1`
- `unit_bias_rewards(v, sender) = old(unit_bias_rewards(v, sender)) - w_reward * UNIT_BIAS`

`changeCommission` transition

Setup

Pick a validator `v` with `auth_address` `sender` (if none exists, add a validator). Pick `commission` between `0` and `MAX_COMMISSION`. Let `amount = 0`.

Call `r = change_commission(v, sender, commission, amount)`.

Preconditions

- `amount = 0`.
- `val_execution(v).auth_address = sender`
- `commission <= MAX_COMMISSION`

Postconditions

- `r = 1`
- `val_execution(v).commission = commission`

`claimRewards` transition

Setup

Generate input `(v, sender, amount)`:

- With a small probability, let `v` and `sender` arbitrary. Let `amount = 0`.
- Pick a pair `(v, sender)` such that address `sender` delegates to validator `v`. If there are not validators, add one (`precompile_add_validator`). Let `amount = 0`.

Call `r = precompile_claim_rewards(v, sender, amount)`.

Preconditions

- `amount = 0`

Postconditions

- `r = 1`

Let `reward = old(delegator(v, sender).rewards + unaccumulated_rewards(v, sender))`

- `balance_of(sender) = old(balance_of(sender)) + reward`
- `balance_of(STAKING_CA) = old(balance_of(STAKING_CA)) - reward`
- `error_bound = old(error_bound) + 3`
- `unit_bias_rewards(v, sender) = old(unit_bias_rewards(v, sender)) - reward * UNIT_BIAS`

`externalReward` transition

Setup

Pick a validator `v` from `active_consensus_view` (if one does not exist, return). Pick an arbitrarry `sender` address. Generate `value` between `MIN_EXTERNAL_REWARD` and `MAX_EXTERNAL_REWARD`.

Call `r = precompile_external_reward(v, sender, value)`.

Preconditions

- `active_consensus_stake /= 0`
- `MIN_EXTERNAL_REWARD <= value <= MAX_EXTERNAL_REWARD`

Postconditions

Let `rpt = value * UNIT_BIAS / active_consensus_stake(v)`. Let `d(a) = delegator_stake(v, a, epoch) + withdrawal_stake(v, a, epoch)`.

- `r = 1`
- `error_bound = old(error_bound) + 1`
- `unit_bias_rewards(v, a) = old(unit_bias_rewards(v, a)) + rpt * d(a)` for all `a`.
- Let `val = val_execution(v)`. Then `val.unclaimed_rewards = old(val.unclaimed_rewards) + value`.

`getDelegator` transition

Setup

Generate input (v, a, sender, value):

Either:

- Pick a pair `(v, a)` such that address `a` delegates to validator `v`. If there are not validators, add one (`precompile_add_validator`).

- Pick an arbitrary `sender`.
- Let `value = 0`

Or:

- Let `v`, `a`, `sender` arbitrary, `value = 0`

Call `r = precompile_get_delegator(v, a, sender, value)`.

Postconditions

- `r = 1`
- `error_bound = old(error_bound) + 3`
- `unaccumulated_rewards(v, a) = 0`
- Let `del = delegator(v, a)`. Then `del.rewards = old(del.rewards + unaccumulated_rewards(v, a))`.

Invariants

After initialization and each transition, invariants are asserted.

Valset invariants

- Each element in `valset_execution` is unique.
- If `v` is an element of `valset_execution` then `val_execution(v).auth_address /= 0`.
- Each element in `valset_consensus` is unique.
- `valset_consensus` has at most `ACTIVE_VALSET_SIZE` elements.
- Each element in `valset_snapshot` is unique.
- `valset_snapshot` has at most `ACTIVE_VALSET_SIZE` elements.
- `valset_consensus` is a subset of `valset_execution`.
- `valset_consensus` contains exactly those elements `v` of `consensus_view` where `consensus_view(v).stake >= ACTIVE_VALIDATOR_STAKE`.
- `valset_snapshot` contains exactly those elements `v` of `snapshot_view` where `snapshot_view(v).stake >= ACTIVE_VALIDATOR_STAKE`.
- `consensus_view(v).commission <= MAX_COMISSION`.
- `snapshot_view(v).commission <= MAX_COMISSION`.
- `val_bitset_bucket` faithfully encodes `valset_execution`.
- Let `val = val_execution(v)`. If `val.stake >= ACTIVE_VALIDATOR_STAKE` and `delegator(v, val.auth_address) >= MIN_VALIDATE_STAKE`, then `v` is an element of `val_execution`.
- If not `in_epoch_delay_period`, then `consensus_view(v).stake = active_consensus_stake(v)`.
- If not `in_epoch_delay_period`, then `consensus_view(v).commission = active_consensus_commission(v)`.
- If `in_epoch_delay_period`, then `snapshot_view(v).stake = active_consensus_stake(v)`.
- If `in_epoch_delay_period`, then `snapshot_view(v).commission = active_consensus_commission(v)`.

`val_execution` invariants

- For a valid `v` (`0 < v <= last_val_id`), `val_execution(v).auth_address /= 0`.
- For an invalid `v` (`v = 0` or `v > last_val_id`), `val_execution(v).auth_address = 0`.
- For `v /= 0`, if `val_execution(v + 1).auth_address /= 0`, then `val_execution(v) /= 0`.
- Let `v` valid, `val = val_execution(v)` and `del = delegator(v, val.auth_address)`. Then `del.stake + del.delta_stake + del.next_delta_stake >= MIN_VALIDATE_STAKE` iff the `ValidatorFlagsWithdrawn` flag is unset in `val.flags`.

- `val_execution.commission(v) <= MAX_COMMISSION`.
- `ValidatorFlagsDoubleSign` is not set in `val_execution(v).flags`.
- Let `v` valid and `val = val_execution(v)`, Then `val.stake >= ACTIVE_VALIDATOR_STAKE` iff the `ValidatorFlagsStakeTooLow` flag is set in `val.flags`.
- `v` is valid iff `val_id_secp(address_from_secpkey(val_execution(v).keys.secp_pubkey)) =/ 0`.
- `v` is valid iff `val_id_bls(address_from_bls_key(val_execution(v).keys.bls_pubkey)) =/ 0`.

Delegator invariants

- Let `del = delegator(v, a)`.
  - `del.next_delta_stake = 0` or `del.next_delta_stake >= DUST_THRESHOLD`.
  - `del.delta_stake = 0` or `del.delta_stake >= DUST_THRESHOLD`.
  - `del.stake = 0` or `del.stake >= DUST_THRESHOLD`.
  - `del.next_delta_stake = 0` iff `del.nextDeltaEpoch = 0`.
  - `del.delta_stake = 0` iff `del.deltaEpoch = 0`.
  - If `del.nextDeltaEpoch /= 0` and `del.deltaEpoch /= 0`, then `del.nextDeltaEpoch = del.deltaEpoch + 1`
  - `del.deltaEpoch <= epoch + 1`.
  - `del.nextDeltaEpoch <= epoch + 2`.
- For `i` active (i.e. an element of `active_withdrawal_ids`), `withdrawal_request(v, a, i).amount /= 0` and `withdrawal_request(v, a, i).epoch /= 0` and. Related comment from the code:

```
// The invariant
// withdrawal_request.amount = 0 iff withdrawal_request.epoch = 0
// is slowing, slowing down the invariant checking too much.
// Instead just verify that the known withdrawal requests have
// non-zero epoch and value.
```

- Let `del = delegator(v, a)`.
  - `unit_bias_rewards(v, a) <= (del.rewards + pending_rewards(v, a) + error_bound + 3 + 256) * UNIT_BIAS`.
  - `(del.rewards + pending_rewards(v, a) + error_bound + 3 + 256) * UNIT_BIAS <= unit_bias_rewards(v, a) + (erro` .
- Let `del = delegator(v, a)`, then `delegator_stake(v, a, epoch) = del.stake + d1 + d2`, where
  - `d1 = del.delta_stake` if `del.delta_epoch <= epoch` and `d1 = 0` otherwise,
  - `d2 = del.next_delta_stake` if `del.next_delta_epoch <= epoch` and `d2 = 0` otherwise.
- `withdrawal_stake(v, a, epoch)` is the sum of `withdrawal_request(v, a, i).amount` over all `i` with `withdrawal_request(v, a, i).epoch > epoch`.
- `active_consensus_stake(v) = 0` or `active_consensus_stake` is the sum over all `a` of `delegator_stake(v, a) + withdrawal_stake(v, a)`.

Accumulated rewards invariants

- For a given epoch `t` and validator `v`, `accumulated_reward_per_token(t, v).refcount` is
  - the number of delegators `del = delegator(v, a)` for some `a` where `del.delta_epoch = t`, plus
  - the number of delegators `del = delegator(v, a)` for some `a` where `del.next_delta_epoch = t`, plus
  - the number of withdrawal requests `wreq = withdrawal_request(v, a, i)` for some `a` and active `i` where `wreq.epoch = t`.

- For an epoch `t` such that `t = 0` or `epoch + 2 < t`, `accumulated_reward_per_token(t, v).refcount = 0`.
- If `accumulated_reward_per_token(t, v).refcount = 0` then `accumulated_reward_per_token(t, v).value = 0`.
- For `t1 <= t2`, if `accumulated_reward_per_token(t1, v).refcount /= 0`, then `accumulated_reward_per_token(t1, v).refcount <= accumulated_reward_per_token(t2, v).refcount`. (In code, checked for `t2 = t1 + 1`, for all `t1`.)
- `accumulated_reward_per_token(t, v) <= val_execution(v).accumulated_reward_per_token`.
- For `wreq = withdrawal_request(v, a, i)` with `i` active, `wreq.acc <= accumulated_reward_per_token(wreq.epoch, v)`.
- Let `del = delegator(v, a)` and `val = val_execution(v)`.
  - `del.accumulated_reward_per_token <= val.accumulated_reward_per_token`.
  - If `0 < del.delta_epoch <= epoch`, then `del.accumulated_reward_per_token <= accumulated_reward_per_token(del.delta_epoch, v)`.
  - If `0 < del.next_delta_epoch <= epoch`, then `del.accumulated_reward_per_token <= accumulated_reward_per_token(del.next_delta_epoch, v)`.
  - If `0 < del.next_delta_epoch <= epoch`, then `accumulated_reward_per_token(del.delta_epoch, v) <= accumulated_reward_per_token(del.next_delta_epoch, v)`.

Linked list invariants

- Each element of `get_delegators_for_validator` is unique.
- Each element of `get_validators_for_delegator` is unique.
- `v` is in `get_validators_for_delegator(a)` iff `a` is in `get_delegators_for_validator(v)`.
- Let `del = delegator(v, a)`, then `a` is in `get_delegators_for_validator(v)` iff `del.stake + del.delta_stake + del.next_delta_stake > 0`.

Solvency invariants

- `balance_of(STAKING_CA)` is
  - the sum of `val_execution(v).stake` over `v`, plus
  - the sum of `val_execution(v).unclaimed_rewards` over `v`, plus
  - the sum of `delegator(v, a).rewards` over `v` and `a`, plus
  - the sum of `withdrawal_request(v, a, i).amount` over `v`, `a` and active `i`.
- Let `pending_rewards_sum` the sum of `pending_rewards(v, a)` over `a`. Then `pending_rewards_sum <= val_execution(v).unclaimed_rewards <= pending_rewards_sum + e`, where the error term `e` is
  - `error_bound`, plus
  - the number of withdrawals `wreq = withdrawal_request(v, a, i)` for some `a` and active `i`.
  - three times the number of delegators `del = delegator(v, a)` for some `a` where `del.stake + del.delta_stake + del.next_delta_stake > 0`.
- `val_execution(v).stake` is the sum of `delegator(v, a).stake + delegator(v, a).delta_stake + delegator(v, a).next_delta_stake` over `a`.

Appendix: Coq Reserve Balance Proof Analysis

Main Theorem Statement

The main theorem of the Reserve Balance mechanism is given by `fullBlockStep` in the Coq proof, and can be expressed as follows.

Theorem:

Let s be a chain state and $t_1, \ldots, t_m$ be a list of transactions such that

- For every pair of transactions t_j and $t_{j'}$, $1 \leq j < j' \leq m$, the block number of t_j is in the range $[n-k+1, n]$, where n is the block number of $t_{j'}$ (in other words, block numbers for the transactions monotonically increase and are all within a k-block window).
- $t_1, \ldots, t_m$ are consensus-acceptable (see below) from state s (according to the Coq model of the Consensus component).
- All senders of transactions $t_1, \ldots, t_m$ are EOAs (not smart contracts) in state s (in other words, there is no code deployed at the sender addresses).
- No transaction among $t_1, \ldots, t_m$, when executed, can turn any of the sender accounts from an EOA to a smart contract (in other words, they can't deploy code at any of the sender addresses).

Then, if we execute from state s (according to the Coq model of the Execution component) the transactions $t_1, \ldots, t_i$ corresponding to the first block, we have that (a) the sender for each transaction $t_1, \ldots, t_i$ has enough balance to pay the gas fees when the transaction is executed, and (b) the resulting state s' is such that all of the above conditions for $t_1, \ldots, t_m$ and s remain true for $t_{i+1}, \ldots, t_m$ and s':

- For every pair of transactions t_j and $t_{j'}$, $i+1 \leq j < j' \leq m$, the block number of t_j is in the range $[n-k+1, n]$, where n is the block number of $t_{j'}$.
- $t_{i+1}, \ldots, t_m$ are consensus-acceptable from state s' (according to the Coq model of the Consensus component).
- All senders of transactions $t_{i+1}, \ldots, t_m$ are EOAs (not smart contracts) in state s'.
- No transaction among $t_{i+1}, \ldots, t_m$, when executed, can turn any of the sender accounts from an EOA to a smart contract.

Consensus-Acceptability

A list of transactions $t_1, \ldots, t_m$ is consensus-acceptable from state s, according to the Coq model of the Consensus component, if the effective reserve balance of every sender remains non-negative after all transactions are executed over state s (note that the effective reserve balance includes adjustments due to emptying transactions).

Intuitively, this can be understood as modelling the behavior of the Consensus component as follows:

- s is the state immediately after the execution of block $n-k$ (which Consensus has access to when building block n).
- $t_1, \ldots, t_m$ are the transactions in blocks $n-k+1, \ldots, n$. These can be split into the in-flight transactions $t_1, \ldots, t_j$ included in blocks $n-k+1, \ldots, n-1$ and the transactions $t_{j+1}, \ldots, t_m$ being included in the block n that Consensus is currently building.
- If $t_1, \ldots, t_m$ are consensus-acceptable, this means that Consensus has determined that $t_{j+1}, \ldots, t_m$ are safe to include in block n after $t_1, \ldots, t_j$ are executed from state s (that is, they won't consume gas beyond the effective reserve balance).

Therefore, consensus-acceptability corresponds to the safety checks performed by Consensus when building block n, based on the state after block $n-k$ and the in-flight transactions in blocks $n-k+1, \ldots, n-1$. Obviously, the correctness of the proof depends on this check correctly reflecting the implementation of the Consensus component in the source code.

Note: If the in-flight transactions $t_1, \ldots, t_j$ were to already exceed the effective reserve balance from state s, then $t_1, \ldots, t_m$ would not be consensus-acceptable regardless of what $t_{j+1}, \ldots, t_m$ are. In other words, it would be impossible for Consensus to build a

valid block. However, since the main theorem above guarantees that the mechanism preserves consensus-acceptability after executing a block, we can be assured when building block n that the in-flight transactions $t_1, \ldots, t_j$ will be by themselves consensus-acceptable, and therefore will not drain the effective reserve balance in this way.

Assumptions

The main theorem depends on the following assumptions, which form an abstract model of how the virtual machine updates the state when executing transactions.

Note: Originally the assumptions below have an additional condition that the transaction t is not a "reserve-balance-reconfig" transaction, a special type of transaction used solely to update the reserve balance amount for a user, implemented via a custom precompile. Since currently the implementation uses a constant reserve balance of 10 MON for all users, this does not apply to the current version of the Reserve Balance mechanism.

`balanceOfRevertSender` and `balanceOfRevertOther`

If a transaction t with sender a and gas limit g reverts on state s due to going below the reserve balance, and $g \leq b$, where b is a's balance on state s, then

- The balance of a in the resulting state s' is $b - g$.
- The balance of every other account $a' \neq a$ in s' is the same as its balance in state s.

In other words, if a transaction reverts because it would dip below the reserve balance, it deducts the gas limit from the sender's balance and doesn't change the balance of any other accounts.

`revertTxDelegationUpdCore` and `execTxDelegationUpdCore`

If a transaction t is executed over state s, then regardless of whether t reverts or not, we have that for every account a, a is delegated in the resulting state s' iff either

- a was delegated on state s and a is not in the list of accounts undelegated by t, or
- a is in the list of accounts delegated by t.

In other words, the transaction's delegating/undelegating behavior is consistent with its list of delegated/undelegated accounts.

Note: The delegation state of the accounts is updated even if the transaction reverts. This is consistent with the EIP-7702 behavior as defined in the proposal.

`execTxSenderBalCore`

If a transaction t with sender a, gas limit g and value v is executed on state s, and

- $g \leq b$, where b is a's balance on state s.
- a is not delegated in the resulting state s'.

then either $b - (g + v) \leq b'$ or $b - g = b'$, where b' is the balance of a on state s'.

In other words, when a transaction is executed it either succeeds and deducts the appropriate fees and transfer value from the sender's balance (but it can also increase the balance by other means, thus $\leq$), or it reverts and only deducts the gas cost (in which case no other change to the balance happens).

Note: The expression $b - (g + v) \leq b'$ actually appears in the Coq proof as $b - (g + v + f) \leq b'$, where f is the max storage fee. However, in the current implementation of the system, $f = 0$.

`execTxCannotDebitNonDelegatedNonContractAccountsCore`

If a transaction t with sender a is executed on state s, resulting in state s', then for every account $a' \neq a$ such that

- a' is an EOA (not a smart contract).
- a' is not a delegated account in state s'.

we have that the balance of a' in state s' cannot be lower than the balance of a in state s.

In other words, a transaction cannot lower the balance of any undelegated EOA besides its sender.

`changedAccountSetSound`

If a transaction t is executed on state s, resulting in state s' and producing a set of changed accounts C, then for every account $a \notin C$, the state of a in s' is the same as the state of a in s (including balance, delegation status, etc.).

In other words, all accounts that are actually modified by the transaction are reported in the set of changed accounts.

Appendix: Token Flow Analysis

The following is an overview of all points in the code of the Monad Execution component where account balances increase or decrease, representing minting, transfers, or burning of Monad native tokens (MON). By analyzing how each of these operations affects the token supply, we can build confidence that they cannot be manipulated by an attacker in order to extract undue value.

Minting Rewards

MON is minted by the `syscallReward(address blockAuthor)` syscall invoked on the Staking Contract. This syscall is a special transaction included at the start of each block by the block builder, and is configured as follows:

- The `blockAuthor` input parameter is the account address of the block builder.
- The `value` component of the transaction is the configured block reward.

This syscall changes the chain state in the following ways:

- The balance of the Staking Contract increases by `value`.
- The Staking Contract's internal accounting assigns `value` as rewards to the validator identified by `blockAuthor`. Of those, a percentage corresponding to the commission configured by the validator is immediately added to the rewards of the validator's auth address. The remaining amount is added to the validator as unclaimed rewards, and the validator's accumulated rewards per token increase accordingly. Later, the delegators will be able to claim or compound these rewards proportionally to their stake.

Since this operation mints `value` MON to the Staking Contract without changing any other account's balance, the total supply of MON increases by `value` as well.

Transferring from the Staking Contract to Delegators

There are two ways for delegators to retrieve funds from the Staking Contract: claiming rewards that were distributed to them and completing withdrawal requests for stake they have undelegated from a validator. These are the only two ways to transfer funds out of the Staking Contract.

Claiming Rewards

When an account calls the `claimRewards(uint64 validatorId)` precompile function in the Staking Contract, the function first updates the caller's rewards as a delegator for the validator given by `validatorId`. For each epoch that has passed since the delegator was last updated, the additional accumulated rewards per token accrued in that epoch are multiplied by the delegator's stake in that epoch, and this amount is removed from the validator's unclaimed rewards and added to the delegator's rewards. If `rewards` is the resulting total of the caller's rewards after this update, the function then changes the state as follows:

- The balance of the Staking Contract decreases by `rewards`.
- The balance of the caller increases by `rewards`.
- The `rewards` for the delegator is set to 0.

Since the balance of the Staking Contract and the caller respectively increase and decrease by the same amount, the total supply of MON remains the same.

Withdrawing

When an address calls the `withdraw(uint64 validatorId, uint8 withdrawId)` precompile function in the Staking Contract, the function retrieves the withdrawal request identified by the triple `(validatorId, withdrawId, msg.sender)` (which must have been created by a previous call to the `undelegate` precompile). If the withdrawal request exists and the withdraw delay has passed since the request was created, the function completes the withdrawal, transferring the amount from the Staking Contract to the caller. The amount transferred includes both:

- `withdrawal_amount`, the original amount included in the withdrawal request, and
- `rewards`, the amount of rewards accrued by the funds in the withdrawal request between when the request was submitted and when the withdrawal came into effect (either one or two epochs later, depending on whether it was submitted before or after the boundary block).

Specifically, the chain state is updated as follows:

- The withdrawal request is deleted.
- The validator's unclaimed rewards decrease by `rewards`.
- The balance of the Staking Contract decreases by `withdrawal_amount + rewards`.
- The balance of `msg.sender` increases by `withdrawal_amount + rewards`.

Again, since the balance of the Staking Contract and the caller respectively increase and decrease by the same amount, the total supply of MON remains the same.

Charging Gas

`ExecuteTransaction::operator()` (here) handles the gas cost of transactions. This is done in two parts:

- The `irrevocable_change` function (called by `ExecuteTransaction::operator()` via `execute_impl2` and then `ExecuteTransactionNoValidation::operator()`) deducts the gas cost from the transaction sender.
- The `execute_final` function pays the transaction rewards to the block's beneficiary (the block author).

In the `irrevocable_change` function, the cost paid by the sender is `gas_limit * (priority_fee_per_gas + base_fee_per_gas)`, where

- `gas_limit` is the gas limit specified in the transaction.
- `base_fee_per_gas` is the base fee for the block, calculated according to the gas controller.
- `priority_fee_per_gas` is the priority fee specified by the transaction sender, capped so it doesn't exceed the maximum fee per gas when added to the base fee.

(Note that Monad doesn't support EIP-4844 transactions, so blob gas is not added to the gas cost.)

In the `execute_final` function, since the gas refund for Monad is 0, the value paid to the beneficiary is `gas_limit * priority_fee_per_gas`. Additionally, there are no other balance adjustments, since there is no gas refund to the sender and the gas limit is guaranteed by the Consensus component to not be lower than the floor gas.

If the transaction fails for any reason, `irrevocable_change` is executed but `executed_final` is not, in which case only the transaction sender is charged and no value is paid to the beneficiary. Therefore, `gas_limit * (priority_fee_per_gas + base_fee_per_gas)` is burned.

On the other hand, if the transaction succeeds, then both `irrevocable_change` and `executed_final` are executed. In that case, `gas_limit * priority_fee_per_gas` is transferred from the transaction sender to the beneficiary, while the remaining `gas_limit * (priority_fee_per_gas + base_fee_per_gas) - gas_limit * priority_fee_per_gas = gas_limit * base_fee_per_` is burned.

Therefore, the gas accounting for a transaction reduces the MON supply by either `gas_limit * base_fee_per_gas` or `gas_limit * (priority_fee_per_gas + base_fee_per_gas)`, depending on whether the transaction succeeds or not.

Transferring Between Accounts

The function `transfer_balances` in `evm.cpp` (here) is used to transfer `msg.value` when an account calls another account or deploys a smart contract:

- The balance of `msg.sender` decreases by `msg.value`.

- The balance of the receiver increases by `msg.value`.

Therefore, the total supply of MON remains the same.

Self-destructing

The `selfdestruct` function in `state.cpp` (here) moves the balance of the contract to the provided beneficiary:

- The balance of the beneficiary increases by the balance of the contract.
- The balance of the contract is set to 0.

Since Monad follows `EVMC_CANCUN`, the contract may or may not be destroyed depending on whether `SELFDESTRUCT` is called in the same transaction that the contract was created or not. If the contract is destroyed and the beneficiary was set to the address of the contract itself, the funds will be lost and the total MON supply will decrease by that amount.

Other

The following are the remaining instances of calls to `add_to_balance` and `subtract_from_balance` used in the code of the Monad Execution component (other than in code used only for testing):

- The function `apply_block_reward` in `block_reward.cpp` calls `add_to_balance` twice. However, since Monad implements proof-of-stake rather than proof-of-work, the amount added is set to 0 in both cases.
- The function `process_withdrawal` in `execute_block.cpp` calls `add_to_balance`, but the call will never be made within Monad because the Consensus component ensures that the list of withdrawals for a block will always be empty (as stake withdrawals are instead implemented by the Staking Contract precompile).
- The function `transfer_balance_dao` in `execute_block.cpp` calls both `add_to_balance` and `subtract_from_balance`, however it does not apply to Monad as it implements the 2016 DAO fork of Ethereum mainnet.
- The function `execute_staking_syscall` in `execute_system_transaction.cpp` and the function `read_valset` in `read_valset.cpp` both call `add_to_balance` on the Staking Contract with a value of 0. This is only meant to create an account for the Staking Contract in the state if it doesn't already exist, and doesn't change any balance or the MON supply.
- The function `eth_call_impl` in `monad_executor.cpp` uses `add_to_balance` and `subtract_from_balance`, however it is the implementation of the `eth_call` RPC method, which simulates a call to a contract without creating a transaction. Therefore, any state changes are not persisted on-chain.

In all of the cases above, no change is made to the MON supply or balance of any accounts.

Appendix: Fuzzing Coverage Summary Table

Filename	Lines	Missed-Lines	Cover
monad-bft/monad-bls/src/aggregation_tree.rs	231	28	87.88%
monad-bft/monad-bls/src/bls.rs	354	199	43.79%
monad-bft/monad-bls/src/lib.rs	50	41	18.00%
monad-bft/monad-chain-config/src/execution_revision.rs	7	3	57.14%
monad-bft/monad-chain-config/src/lib.rs	8	0	100.00%
monad-bft/monad-chain-config/src/revision.rs	3	0	100.00%
monad-bft/monad-consensus-types/src/block.rs	284	139	51.06%
monad-bft/monad-consensus-types/src/block_validator.rs	25	25	0.00%
monad-bft/monad-consensus-types/src/checkpoint.rs	18	18	0.00%
monad-bft/monad-consensus-types/src/lib.rs	37	24	35.14%
monad-bft/monad-consensus-types/src/metrics.rs	7	7	0.00%
monad-bft/monad-consensus-types/src/no_endorsement.rs	24	24	0.00%
monad-bft/monad-consensus-types/src/payload.rs	39	24	38.46%
monad-bft/monad-consensus-types/src/quorum_certificate.rs	57	32	43.86%
monad-bft/monad-consensus-types/src/timeout.rs	214	120	43.93%
monad-bft/monad-consensus-types/src/tip.rs	18	18	0.00%
monad-bft/monad-consensus-types/src/validator_data.rs	145	145	0.00%
monad-bft/monad-consensus-types/src/voting.rs	15	15	0.00%
monad-bft/monad-crypto/src/certificate_signature.rs	67	67	0.00%
monad-bft/monad-crypto/src/hasher.rs	53	30	43.40%
monad-bft/monad-crypto/src/lib.rs	10	10	0.00%
monad-bft/monad-crypto/src/signing_domain.rs	8	7	0.00%
monad-bft/monad-eth-block-policy/src/lib.rs	944	944	0.00%
monad-bft/monad-eth-block-policy/src/nonce_usage.rs	70	70	0.00%
monad-bft/monad-eth-block-policy/src/validation.rs	178	17	90.45%
monad-bft/monad-merkle/src/lib.rs	91	0	100.00%
monad-bft/monad-secp/src/lib.rs	79	62	21.52%
monad-bft/monad-secp/src/recoverable_address.rs	15	15	0.00%
monad-bft/monad-secp/src/secp.rs	134	102	23.88%
monad-bft/monad-testutil/src/signing.rs	34	0	100.00%
monad-bft/monad-testutil/src/validators.rs	50	0	100.00%
monad-bft/monad-types/src/lib.rs	271	260	4.06%
monad-bft/monad-validator/src/epoch_manager.rs	37	37	0.00%
monad-bft/monad-validator/src/signature_collection.rs	14	14	0.00%
monad-bft/monad-validator/src/simple_round_robin.rs	11	11	0.00%
monad-bft/monad-validator/src/validator_mapping.rs	8	3	62.50%
monad-bft/monad-validator/src/validator_set.rs	94	74	21.28%
monad-bft/monad-validator/src/validators_epoch_mapping.rs	51	51	0.00%
monad-bft/monad-validator/src/weighted_round_robin.rs	47	47	0.00%
monad/category/async/config.hpp	77	77	0.00%
monad/category/async/connected_operation.hpp	59	59	0.00%
monad/category/async/detail/connected_operation_storage.hpp	161	161	0.00%
monad/category/async/detail/scope_polyfill.hpp	41	41	0.00%
monad/category/async/detail/start_lifetime_as_polyfill.hpp	9	9	0.00%
monad/category/async/erased_connected_operation.hpp	200	200	0.00%
monad/category/async/io.cpp	623	618	0.80%
monad/category/async/io.hpp	255	255	0.00%
monad/category/async/io_senders.hpp	203	203	0.00%
monad/category/async/sender_errc.hpp	147	147	0.00%
monad/category/async/storage_pool.cpp	877	877	0.00%
monad/category/async/storage_pool.hpp	104	104	0.00%
monad/category/async/util.cpp	185	185	0.00%
monad/category/async/util.hpp	13	9	30.77%
monad/category/core/assert.c	44	44	0.00%
monad/category/core/backtrace.cpp	106	106	0.00%
monad/category/core/backtrace.hpp	5	5	0.00%
monad/category/core/basic_formatter.hpp	3	3	0.00%
monad/category/core/blake3.hpp	9	0	100.00%

Filename	Lines	Missed-Lines	Cover
monad/category/core/byte_string.hpp	12	3	75.00%
monad/category/core/bytes.hpp	21	3	85.71%
monad/category/core/bytes_hash_compare.hpp	6	0	100.00%
monad/category/core/hash.hpp	17	17	0.00%
monad/category/core/io/buffer_pool.cpp	15	15	0.00%
monad/category/core/io/buffer_pool.hpp	11	11	0.00%
monad/category/core/io/buffers.cpp	92	92	0.00%
monad/category/core/io/buffers.hpp	51	51	0.00%
monad/category/core/io/ring.cpp	33	33	0.00%
monad/category/core/io/ring.hpp	24	24	0.00%
monad/category/core/keccak.c	14	0	100.00%
monad/category/core/keccak.hpp	8	3	62.50%
monad/category/core/likely.h	1	1	0.00%
monad/category/core/lru/static_lru_cache.hpp	50	50	0.00%
monad/category/core/math.hpp	7	1	85.71%
monad/category/core/mem/allocators.hpp	106	56	47.17%
monad/category/core/mem/huge_mem.cpp	47	47	0.00%
monad/category/core/mem/huge_mem.hpp	17	17	0.00%
monad/category/core/monad_exception.cpp	38	38	0.00%
monad/category/core/nibble.h	22	0	100.00%
monad/category/core/rlp/encode.hpp	90	15	83.33%
monad/category/core/runtime/uint256.cpp	62	6	90.32%
monad/category/core/runtime/uint256.hpp	808	174	78.47%
monad/category/core/small_prng.hpp	26	26	0.00%
monad/category/core/tl_tid.c	3	0	100.00%
monad/category/core/tl_tid.h	6	0	100.00%
monad/category/core/unaligned.hpp	9	0	100.00%
monad/category/execution/ethereum/block_hash_buffer.cpp	136	129	5.15%
monad/category/execution/ethereum/block_hash_buffer.hpp	1	0	100.00%
monad/category/execution/ethereum/block_hash_history.cpp	46	46	0.00%
monad/category/execution/ethereum/core/account.hpp	8	1	87.50%
monad/category/execution/ethereum/core/address.hpp	10	4	60.00%
monad/category/execution/ethereum/core/block.hpp	1	1	0.00%
monad/category/execution/ethereum/core/contract/abi_decode.hpp	26	6	76.92%
monad/category/execution/ethereum/core/contract/abi_decode_error.cpp	9	9	0.00%
monad/category/execution/ethereum/core/contract/abi_encode.hpp	79	27	65.82%
monad/category/execution/ethereum/core/contract/big_endian.hpp	15	0	100.00%
monad/category/execution/ethereum/core/contract/checked_math.cpp	37	18	51.35%
monad/category/execution/ethereum/core/contract/events.hpp	15	0	100.00%
monad/category/execution/ethereum/core/contract/storage_array.hpp	35	0	100.00%
monad/category/execution/ethereum/core/contract/storage_variable.hpp	49	0	100.00%
monad/category/execution/ethereum/core/fmt/account_fmt.hpp	15	15	0.00%
monad/category/execution/ethereum/core/fmt/address_fmt.hpp	7	7	0.00%
monad/category/execution/ethereum/core/fmt/bytes_fmt.hpp	7	7	0.00%
monad/category/execution/ethereum/core/fmt/int_fmt.hpp	4	4	0.00%
monad/category/execution/ethereum/core/rlp/account_rlp.cpp	19	12	36.84%
monad/category/execution/ethereum/core/rlp/address_rlp.hpp	27	6	77.78%
monad/category/execution/ethereum/core/rlp/block_rlp.cpp	158	57	63.92%
monad/category/execution/ethereum/core/rlp/bytes_rlp.hpp	14	0	100.00%
monad/category/execution/ethereum/core/rlp/int_rlp.hpp	16	4	75.00%
monad/category/execution/ethereum/core/rlp/receipt_rlp.cpp	122	116	4.92%
monad/category/execution/ethereum/core/rlp/signature_rlp.cpp	6	0	100.00%
monad/category/execution/ethereum/core/rlp/transaction_rlp.cpp	277	137	50.54%
monad/category/execution/ethereum/core/rlp/withdrawal_rlp.cpp	36	28	22.22%
monad/category/execution/ethereum/core/signature.cpp	23	6	73.91%
monad/category/execution/ethereum/create_contract_address.cpp	18	0	100.00%
monad/category/execution/ethereum/db/block_db.cpp	59	59	0.00%
monad/category/execution/ethereum/db/db.hpp	16	16	0.00%

Filename	Lines	Missed-Lines	Cover
monad/category/execution/ethereum/db/db_snapshot.cpp	349	349	0.00%
monad/category/execution/ethereum/db/db_snapshot_filesystem.cpp	165	165	0.00%
monad/category/execution/ethereum/db/file_db.cpp	29	29	0.00%
monad/category/execution/ethereum/db/trie_db.cpp	660	407	38.33%
monad/category/execution/ethereum/db/trie_db.hpp	12	3	75.00%
monad/category/execution/ethereum/db/util.cpp	855	677	20.82%
monad/category/execution/ethereum/db/util.hpp	3	0	100.00%
monad/category/execution/ethereum/evm.cpp	204	39	80.88%
monad/category/execution/ethereum/evm.hpp	1	1	0.00%
monad/category/execution/ethereum/evmc_host.cpp	127	58	54.33%
monad/category/execution/ethereum/evmc_host.hpp	60	19	68.33%
monad/category/execution/ethereum/precompiles.cpp	80	5	93.75%
monad/category/execution/ethereum/precompiles.hpp	7	7	0.00%
monad/category/execution/ethereum/precompiles_bls12.cpp	307	307	0.00%
monad/category/execution/ethereum/precompiles_bls12.hpp	3	3	0.00%
monad/category/execution/ethereum/precompiles_impl.cpp	272	249	8.46%
monad/category/execution/ethereum/rlp/decode.hpp	101	22	78.22%
monad/category/execution/ethereum/rlp/decode_error.cpp	14	14	0.00%
monad/category/execution/ethereum/rlp/encode2.hpp	54	0	100.00%
monad/category/execution/ethereum/state2/block_state.cpp	180	104	42.22%
monad/category/execution/ethereum/state2/block_state.hpp	3	0	100.00%
monad/category/execution/ethereum/state2/fmt/state_deltas_fmt.hpp	39	39	0.00%
monad/category/execution/ethereum/state3/account_state.cpp	68	11	83.82%
monad/category/execution/ethereum/state3/account_state.hpp	82	48	41.46%
monad/category/execution/ethereum/state3/account_substate.hpp	36	6	83.33%
monad/category/execution/ethereum/state3/state.cpp	479	173	63.88%
monad/category/execution/ethereum/state3/version_stack.hpp	51	12	76.47%
monad/category/execution/ethereum/trace/call_frame.cpp	63	63	0.00%
monad/category/execution/ethereum/trace/call_tracer.cpp	176	172	2.27%
monad/category/execution/ethereum/trace/call_tracer.hpp	1	0	100.00%
monad/category/execution/ethereum/trace/rlp/call_frame_rlp.cpp	92	92	0.00%
monad/category/execution/ethereum/transaction_gas.cpp	114	35	69.30%
monad/category/execution/ethereum/transaction_gas.hpp	3	0	100.00%
monad/category/execution/ethereum/tx_context.cpp	23	0	100.00%
monad/category/execution/ethereum/types/fmt/incarnation_fmt.hpp	11	11	0.00%
monad/category/execution/ethereum/types/incarnation.hpp	19	6	68.42%
monad/category/execution/ethereum/validate_block.cpp	170	102	40.00%
monad/category/execution/ethereum/validate_transaction.cpp	140	105	25.00%
monad/category/execution/monad/monad_precompiles.cpp	45	45	0.00%
monad/category/execution/monad/monad_precompiles_impl.cpp	40	40	0.00%
monad/category/execution/monad/staking/fuzzer/staking_contract_fuzzer.cpp	85	4	95.29%
monad/category/execution/monad/staking/fuzzer/staking_contract_machine.cpp	2188	62	97.17%
monad/category/execution/monad/staking/fuzzer/staking_contract_machine.hpp	6	0	100.00%
monad/category/execution/monad/staking/fuzzer/staking_contract_model.cpp	461	11	97.61%
monad/category/execution/monad/staking/staking_contract.cpp	1206	285	76.37%
monad/category/execution/monad/staking/staking_contract.hpp	120	5	95.83%
monad/category/execution/monad/staking/test/input_generation.cpp	132	29	78.03%
monad/category/execution/monad/staking/util/bls.cpp	6	0	100.00%
monad/category/execution/monad/staking/util/bls.hpp	37	0	100.00%
monad/category/execution/monad/staking/util/consensus_view.cpp	5	0	100.00%
monad/category/execution/monad/staking/util/consensus_view.hpp	6	0	100.00%
monad/category/execution/monad/staking/util/constants.hpp	36	10	72.22%
monad/category/execution/monad/staking/util/delegator.cpp	5	0	100.00%
monad/category/execution/monad/staking/util/delegator.hpp	60	0	100.00%
monad/category/execution/monad/staking/util/secp256k1.cpp	17	0	100.00%
monad/category/execution/monad/staking/util/secp256k1.hpp	38	3	92.11%
monad/category/execution/monad/staking/util/staking_error.cpp	39	39	0.00%
monad/category/execution/monad/staking/util/val_execution.cpp	5	0	100.00%

Filename	Lines	Missed-Lines	Cover
monad/category/execution/monad/staking/util/val_execution.hpp	43	0	100.00%
monad/category/mpt/compute.cpp	93	29	68.82%
monad/category/mpt/compute.hpp	225	155	31.11%
monad/category/mpt/copy_trie.cpp	264	264	0.00%
monad/category/mpt/db.cpp	1113	1038	6.74%
monad/category/mpt/db.hpp	52	52	0.00%
monad/category/mpt/db_error.hpp	10	0	100.00%
monad/category/mpt/deserialize_node_from_receiver_result.hpp	52	52	0.00%
monad/category/mpt/detail/boost_fiber_workarounds.hpp	47	47	0.00%
monad/category/mpt/detail/collected_stats.hpp	4	0	100.00%
monad/category/mpt/detail/db_metadata.hpp	276	276	0.00%
monad/category/mpt/detail/unsigned_20.hpp	21	21	0.00%
monad/category/mpt/find.cpp	51	16	68.63%
monad/category/mpt/find_notify_fiber.cpp	325	325	0.00%
monad/category/mpt/find_request_sender.hpp	161	161	0.00%
monad/category/mpt/merkle/compact_encode.hpp	25	10	60.00%
monad/category/mpt/merkle/node_reference.hpp	10	0	100.00%
monad/category/mpt/nibbles_view.hpp	218	87	60.09%
monad/category/mpt/nibbles_view_fmt.hpp	7	7	0.00%
monad/category/mpt/node.cpp	419	78	81.38%
monad/category/mpt/node.hpp	145	100	31.03%
monad/category/mpt/node_cache.hpp	30	30	0.00%
monad/category/mpt/node_cursor.hpp	11	0	100.00%
monad/category/mpt/read_node_blocking.cpp	38	38	0.00%
monad/category/mpt/request.hpp	46	7	84.78%
monad/category/mpt/state_machine.hpp	4	3	25.00%
monad/category/mpt/test/fuzz/generated_kv.cpp	16	0	100.00%
monad/category/mpt/test/fuzz/one_hundred_updates.cpp	11	0	100.00%
monad/category/mpt/test/fuzz/test_fixtures_fuzz.hpp	171	7	95.91%
monad/category/mpt/test/test_fixtures_base.hpp	146	98	32.88%
monad/category/mpt/traverse.hpp	265	265	0.00%
monad/category/mpt/traverse_util.hpp	77	77	0.00%
monad/category/mpt/trie.cpp	1516	1062	29.95%
monad/category/mpt/trie.hpp	670	532	20.60%
monad/category/mpt/update.hpp	28	9	67.86%
monad/category/mpt/update_aux.cpp	1556	1533	1.48%
monad/category/mpt/upward_tnode.hpp	98	61	37.76%
monad/category/mpt/util.hpp	124	115	7.26%
monad/category/statesync/statesync_client.cpp	140	140	0.00%
monad/category/statesync/statesync_client_context.cpp	105	105	0.00%
monad/category/statesync/statesync_protocol.cpp	181	181	0.00%
monad/category/statesync/statesync_protocol.hpp	1	1	0.00%
monad/category/statesync/statesync_server.cpp	541	541	0.00%
monad/category/statesync/statesync_server_context.cpp	209	209	0.00%
monad/category/statesync/statesync_version.cpp	6	6	0.00%
monad/category/statesync/test/fuzz_statesync.cpp	145	145	0.00%
monad/category/vm/code.hpp	28	7	75.00%
monad/category/vm/compiler.cpp	93	14	84.95%
monad/category/vm/compiler.hpp	48	37	22.92%
monad/category/vm/compiler/ir/basic_blocks.cpp	54	27	50.00%
monad/category/vm/compiler/ir/basic_blocks.hpp	331	95	71.30%
monad/category/vm/compiler/ir/instruction.hpp	238	199	16.39%
monad/category/vm/compiler/ir/local_stacks.cpp	300	0	100.00%
monad/category/vm/compiler/ir/local_stacks.hpp	52	52	0.00%
monad/category/vm/compiler/ir/poly_typed.cpp	510	172	66.27%
monad/category/vm/compiler/ir/poly_typed.hpp	3	3	0.00%
monad/category/vm/compiler/ir/poly_typed/exceptions.hpp	12	2	83.33%
monad/category/vm/compiler/ir/poly_typed/infer.cpp	594	4	99.33%

Filename	Lines	Missed-Lines	Cover
monad/category/vm/compiler/ir/poly_typed/infer_state.cpp	164	0	100.00%
monad/category/vm/compiler/ir/poly_typed/infer_state.hpp	9	0	100.00%
monad/category/vm/compiler/ir/poly_typed/kind.cpp	532	150	71.80%
monad/category/vm/compiler/ir/poly_typed/kind.hpp	20	14	30.00%
monad/category/vm/compiler/ir/poly_typed/strongly_connected_components.cpp	71	0	100.00%
monad/category/vm/compiler/ir/poly_typed/subst_map.cpp	246	45	81.71%
monad/category/vm/compiler/ir/poly_typed/subst_map.hpp	6	0	100.00%
monad/category/vm/compiler/ir/poly_typed/unify.cpp	776	92	88.14%
monad/category/vm/compiler/ir/x86.cpp	364	6	98.35%
monad/category/vm/compiler/ir/x86.hpp	3	0	100.00%
monad/category/vm/compiler/ir/x86/emitter.cpp	6554	585	91.07%
monad/category/vm/compiler/ir/x86/emitter.hpp	134	3	97.76%
monad/category/vm/compiler/ir/x86/types.hpp	50	17	66.00%
monad/category/vm/compiler/ir/x86/virtual_stack.cpp	649	41	93.68%
monad/category/vm/compiler/ir/x86/virtual_stack.hpp	144	10	93.06%
monad/category/vm/compiler/transactional_unordered_map.hpp	63	0	100.00%
monad/category/vm/core/assert.cpp	11	11	0.00%
monad/category/vm/evm/delegation.cpp	31	0	100.00%
monad/category/vm/evm/monad/revision.h	25	1	96.00%
monad/category/vm/evm/opcodes.hpp	66	33	50.00%
monad/category/vm/fuzzing/generator/choice.hpp	85	2	97.65%
monad/category/vm/fuzzing/generator/generator.hpp	902	69	92.35%
monad/category/vm/fuzzing/generator/instruction_data.cpp	35	3	91.43%
monad/category/vm/fuzzing/generator/instruction_data.hpp	5	5	0.00%
monad/category/vm/host.hpp	20	12	40.00%
monad/category/vm/interpreter/call_runtime.hpp	75	0	100.00%
monad/category/vm/interpreter/debug.hpp	7	7	0.00%
monad/category/vm/interpreter/execute.cpp	26	0	100.00%
monad/category/vm/interpreter/instruction_table.hpp	791	96	87.86%
monad/category/vm/interpreter/intercode.cpp	31	0	100.00%
monad/category/vm/interpreter/intercode.hpp	18	0	100.00%
monad/category/vm/interpreter/push.hpp	47	6	87.23%
monad/category/vm/interpreter/stack.hpp	53	18	66.04%
monad/category/vm/llvm/dependency_blocks.hpp	410	410	0.00%
monad/category/vm/llvm/emitter.hpp	991	991	0.00%
monad/category/vm/llvm/execute.cpp	149	149	0.00%
monad/category/vm/llvm/llvm.cpp	54	28	48.15%
monad/category/vm/llvm/llvm_state.hpp	357	357	0.00%
monad/category/vm/runtime/bin.hpp	37	0	100.00%
monad/category/vm/runtime/cached_allocator.hpp	57	2	96.49%
monad/category/vm/runtime/call.hpp	191	2	98.95%
monad/category/vm/runtime/context.cpp	175	26	85.14%
monad/category/vm/runtime/create.hpp	78	10	87.18%
monad/category/vm/runtime/data.hpp	136	14	89.71%
monad/category/vm/runtime/environment.hpp	31	5	83.87%
monad/category/vm/runtime/exit.cpp	13	5	61.54%
monad/category/vm/runtime/keccak.hpp	12	0	100.00%
monad/category/vm/runtime/log.hpp	64	0	100.00%
monad/category/vm/runtime/math.hpp	56	3	94.64%
monad/category/vm/runtime/memory.hpp	27	0	100.00%
monad/category/vm/runtime/selfdestruct.hpp	46	8	82.61%
monad/category/vm/runtime/storage.cpp	21	0	100.00%
monad/category/vm/runtime/storage.hpp	58	0	100.00%
monad/category/vm/runtime/storage_costs.hpp	6	3	50.00%
monad/category/vm/runtime/transmute.hpp	60	0	100.00%
monad/category/vm/runtime/types.hpp	94	22	76.60%
monad/category/vm/utils/evmc_utils.cpp	6	6	0.00%
monad/category/vm/utils/evmc_utils.hpp	25	0	100.00%

Filename	Lines	Missed-Lines	Cover
monad/category/vm/utils/lru_weight_cache.hpp	150	32	78.67%
monad/category/vm/utils/rc_ptr.hpp	82	0	100.00%
monad/category/vm/varcode_cache.cpp	29	0	100.00%
monad/category/vm/varcode_cache.hpp	12	9	25.00%
monad/category/vm/vm.cpp	99	29	70.71%
monad/category/vm/vm.hpp	68	53	22.06%
monad/test/vm/fuzzer/assertions.cpp	70	0	100.00%
monad/test/vm/fuzzer/compiler_hook.hpp	636	0	100.00%
monad/test/vm/fuzzer/fuzzer.cpp	411	61	85.16%
monad/test/vm/fuzzer/typechecker_fuzzer.cpp	113	9	92.04%
monad/test/vm/vm/test_vm.cpp	173	76	56.07%
monad/test/vm/vm/test_vm.hpp	19	19	0.00%