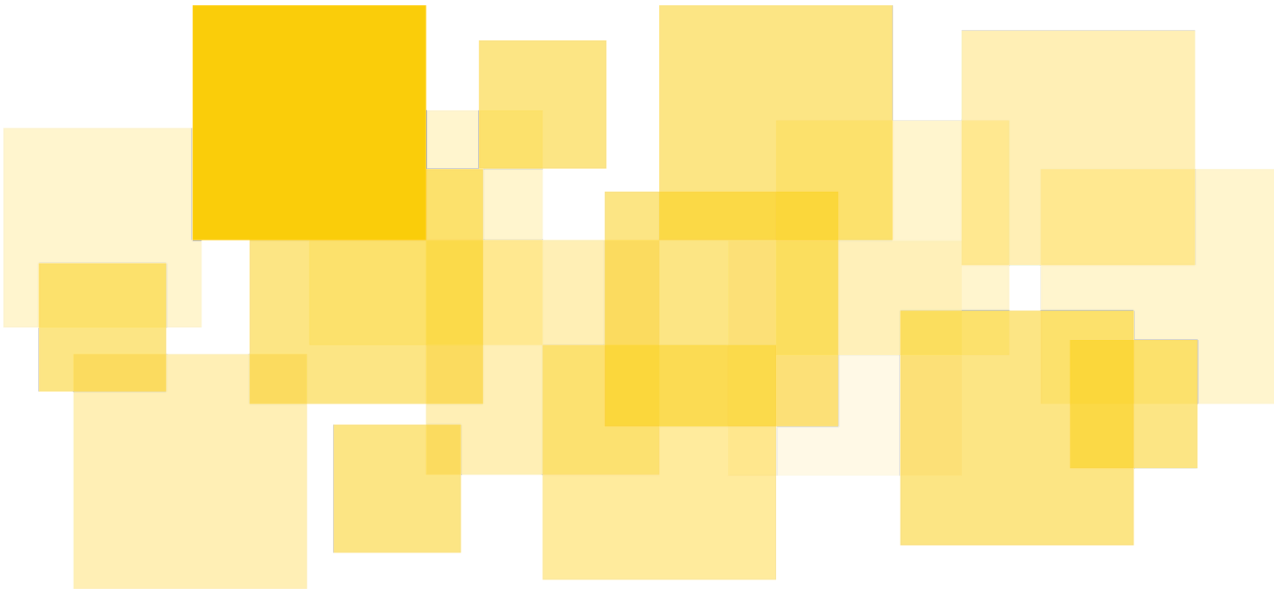
Security Audit Report

Overclock-Validator Mithril Audit Solana

Delivered: April 14, 2026



runtime verification

Table of Contents

- [F-B01] `pkg/shred/shred.go` function `NewShredFromSerialized` panics on unimplemented shred formats instead of gracefully handling them
- [F-B02] `translateInternal` panic on zero-size stack access with invalid address
- [F-B03] `pkg/sealevel/vote_state.go` function `AuthorizedVoters.PurgeAuthorizedVoters` panics when the authorized voters set becomes empty
- [F-B04] Function `calculateDeltaLtHash` missing division-by-zero check
- [F-B05] Function `partitionIdxFromSlotIdx` can overflow
- [F-B06] `SaturatingPow` does not check division by 0
- [F-B07] `pkg/sbpf/opcode_test.go` uses incorrect opcode values for signed division instructions
- [F-B08] `pkg/features/features.go` function `AllEnabled` displays feature gate addresses as raw bytes instead of base58-encoded strings
- [F-B09] `translateInstructionC` and `translateInstructionRust` do not make defensive copy of instruction data
- [F-D01] Missing Divide-by-Zero Check in OpUrem32Reg
- [F-D02] Missing Blockhash Validation in Transaction Sanitization
- [F-D03] Incorrect Exception Handling in VM Interpreter
- [F-D04] Missing Fee Payer Validation Before Transaction Execution
- [F-D05] LoaderV4 `setLoaderV4AcctState` Missing Features Parameter
- [F-D06] Incorrect Section Order Validation in ELF Loader
- [F-D07] Incorrect Executable Flag Checks in BorrowedAccount
- [F-D08] String Parsing in ELF Loader Doesn't Match Agave Behavior
- [F-D09] Incorrect ELF Deploy Checks During Execution
- [F-D10] Missing `verifyCheckSh32` Handler in SBPF Verifier
- [F-D11] Side Effects Occurring After Error in VM Interpreter
- [F-D12] Incorrect Right Shift in Signed High Multiply Instructions
- [F-D13] Missing `ExcUnsupportedInstruction` Error for Invalid Opcodes
- [F-D14] Missing Padding Zeroing in `serializeParametersAligned`
- [F-D15] Inconsistent Error Code for Empty ELF File
- [F-D16] SBPF Version Validation Order in ELF Loader
- [F-D17] Program Header Overlap Check for Empty Program Header Table
- [F-D18] Incorrect Error Code for Program Loading Failures
- [F-D19] Incorrect PC Value on OpExit Stack Pop

- Informative Findings
  - [B01] Unnecessary input validation on flags `endSlot` and `numReplaySlots`
  - [B02] Function parameter `rpcServer` is never used in `replayBlocks` function
  - [B03] Optimization in `VoteStateVersions` struct
  - [B04] Remove code duplication logic in `setVoteAccountState` function
  - [B05] Unauthenticated JSON-RPC Service Bound to All Interfaces
  - [B06] Missing `Destroy` calls of grocksdb leading to memory leak
  - [B07] Unbounded transaction count allows potential memory exhaustion
  - [B08] Inconsistent logging and error handling
  - [B09] Implementation gaps indicated by `TODO: implement` comments
  - [B10] Attribute `ProgramAccounts` of struct `InstructionCtx` is slice, but always contains exactly one entry
  - [B11] `BorrowedAccount` is dropped early in `SyscallInvokeSignedRustImpl`
  - [B12] Snapshot downloader library might skip other snapshot compression-variants provided by an RPC node
  - [B13] RPC address hardcoded to a mainnet address for downloading snapshots
  - [B14] Wrong error message

- [B15] Inconsistent command-line interface argument names
- [B16] Nil-pointer dereference on StatsD client initialization causing panic
- [B17] Panic on RPC failure in `GetLeaderForSlot`
- [B18] Incorrect pointer returned in `BorrowExecutableAccount`
- [B19] ELF loader loads different number of bytes prior to validation compared to Agave
- [B20] Mithril adopts code design decisions from Agave
- [B21] Use-After-Drop in `ExecutionCtx::ExecuteInstruction`
- [B22] Lack of proper pull request and CI workflows

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 may exist. This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of the 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

Solana Foundation engaged Runtime Verification Inc. to perform a security audit of Mithril - a lightweight, Go-based full node implementation for the Solana blockchain, developed by the Overclock team to improve accessibility and decentralization of Solana validator infrastructure. The audit was conducted between September 1st, 2025 and December 5th, 2025. It should be noted that the audit was extended by two weeks, resulting in a revised end date of December 5th, 2025, instead of the original end date of November 21st, 2025. The objective was to assess the implementation's security and correctness, identify exploitable vulnerabilities, and provide recommendations to enhance the client system's reliability.

The audit process involved a design review and codebase review, focusing on manual inspection of critical components in a time-boxed limited scope. Runtime Verification utilized several techniques related to specification generation and invariant analysis to validate the system's behavior under various conditions. In addition, a fuzzing campaign involving both targeted fuzzing and differential fuzzing was performed that identified crashes and checked invariants as well as correctness.

The audit resulted in findings ranging in severity from critical to informative. These findings have been organized into the following sections:

- **Findings of Severity**: considers findings that constitute threats to the validator or cause divergence from the canonical Solana implementation.
- **Fuzzing Findings**: considers findings discovered by the fuzzing campaign, employing both targeted fuzzing and differential fuzzing.
- **Informative Findings**: considers findings that constitute general improvements or potential enhancements to the overall design and implementation of Mithril.

Given the substantial size of the codebase and the fact that we continued to have findings via differential fuzzing late into the audit period, we recommend a follow-up engagement that continues the utilization and extension of the differential fuzzing harness.

Scope

The scope of this audit is limited to the code contained in the public GitHub repository provided by the Overclock team. The Mithril project was identified within this repository as the primary artifact under review for this engagement.

Mithril Repository (public)
https://github.com/Overclock-Validator/mithril

Initial commit: `35e3b74fb29977c6df003e048e443ff1736414ba`

The initial commit was used at the beginning of the audit, though we incorporated updates from the Mithril repository throughout the audit. This was done to both review up-to-date code of the client as well as to incorporate bug fixes into our fuzzing harness.

Mithril is a lightweight, Go-based full node implementation for the Solana blockchain. The audit focused on assessing the correctness, safety, and robustness of Mithril's core execution and state-management logic, as well as its virtual machine and persistence components.

The design review and code analysis targeted the following areas as defined in the engagement scope:

- Execution Pipeline: ReplayBlocks, ProcessBlock, ProcessTransaction
- State Management: AccountsDb, MemAccounts, snapshot ingest
- Virtual Machine & Runtime: SBPF VM, syscall registry, compute metering, native programs, sysvars

Additionally, a concurrent fuzzing campaign was conducted to complement the manual audit, targeting critical components of the execution and runtime pipeline to identify potential safety violations and latent bugs.

The audit is strictly limited to the artifacts contained within the Mithril repository. External Solana dependencies, infrastructure tooling, and deployment configurations are excluded from the scope of this engagement.

Overview

The Solana ecosystem currently uses the Agave client for block generation, proposal, validation, voting, and more. Validator nodes take turns acting as leaders and proposing blocks, while all validators re-execute them and vote on blocks using their available stake. On top of that, Agave has been optimized for throughput with a new block being produced in less than 400 ms on average by scheduled leaders across the validator set. Besides consensus participating nodes, there are also RPC nodes queried by users and dApps. Users and dApps using these RPC nodes rely on the RPC nodes being truthful and reliable. These RPC nodes themselves might depend on the truthfulness of other Solana nodes. As this might not fulfill reliability requirements of end users, there are different approaches to increase the reliability, the most obvious being running a dedicated full node to directly query from. In Solana, this could be achieved by running a full node using the Agave client. There are also other full nodes in development, such as, e.g., Firedancer, which is written in C. Though running a full node that participates in blockchain consensus has higher resource requirements than necessary, as an end user querying an API might only be interested in settled blocks anyway, i.e., blocks that achieved a supermajority of votes weighted by validator stake. Therefore, not participating in the consensus protocol and only executing settled blocks, i.e., not optimistically executing blocks that could possibly fork while also not maintaining consensus protocol state, can yield performance benefits that are critical for users to be able to run their own nodes due to the large resource requirements of full nodes. Mithril is looking to fulfill that role by providing a lighter Solana node implementation that validates all settled blocks while not participating in consensus.

The official canonical implementation of the Solana protocol and semantics is defined to be the Agave implementation that is written in Rust. This means that Mithril, which is written in Go, will have to abide by all protocol implementation details of Agave, including any existing bugs. On top of that, the Solana protocol is changed over time with feature gates that change the behavior and implementation of the protocol at precisely defined points in time. Feature gates that have been activated once and that are unlikely to be disabled again are sometimes removed from the Agave implementation. Mithril optimistically leaves out some feature gate implementations that have since been enabled.

Components

For this purpose, Mithril needs to implement different functionalities, a relevant portion of which will be briefly introduced here.

SBPF interpreter & ELF loader

The Solana virtual machine uses the Solana BPF (SBPF) architecture, which is a custom variation of the Berkeley Packet Filter (BPF) architecture. Mithril implements an interpreter for sBPF programs, and as SBPF programs are contained in Executable and Linkable Format (ELF) files, Mithril also implements an ELF loader. It should be noted that Solana programs can be written in either C or Rust, with most Solana programs being written in Rust.

Block replay

In Solana, a block consists of transactions, a transaction consists of instructions, and an instruction calls a Solana program. A Solana program can be either a native program that is implemented natively in the respective Solana node or an SBPF program invocation. SBPF programs can also recursively invoke programs via Cross Program Invocation (CPI) up to a depth of four. Blocks, on the other hand, are grouped into epochs. While a new block is generated every roughly 400 ms at the current time of writing, an epoch currently lasts roughly 2 days.

Mithril has to replay blocks that have achieved finality in the Solana blockchain. A block achieves finality once it is voted on by a supermajority of stake and subsequent votes on its descendants extend its lockout to the maximum depth (after 32 consecutive votes). Please note that a vote is a transaction and is part of a block.

AccountsDB

Solana stores blockchain state in accounts. Each account has an address that is used to identify an account, with the address being either a public key or a Program Derived Address (PDA). Besides a fixed set of attributes, accounts also have a variable-length data

field, which contains either data or a program. The fixed set of attributes consists of `lamports`, `owner`, `executable`, and `rent_epoch`, with `rent_epoch` being a legacy attribute. `lamports` counts the number of lamports, Solana's currency, a given account holds. `owner` designates the owner account of the respective account, which is always a program. Only the owner of an account can change the data attribute or deduct lamports. Finally, the `executable` flag indicates whether the account stores state or is a program.

Solana nodes must store and update account state. The initial state is usually extracted from a snapshot, while account updates are the result of executing transactions. For efficiency, account state is typically stored both on disk and in RAM. While Agave uses RocksDB for implementing their accounts database, Mithril uses a custom database solution. Mithril's accounts database is located in the Go package `pkg/accountsdb`. It should be noted that Mithril has another package, `pkg/blockstore`, that does make use of RocksDB, though this package is currently not in use.

Native program and sysvar implementations

Solana programs can either be SBPF programs or native programs, as mentioned previously. Therefore, Mithril additionally implements all native programs, such as, e.g., the system program and the vote program. In addition to native programs, Solana programs can query information with sysvars that are system accounts that can be queried by programs either by reading an account or by using a syscall. Finally, syscalls must be implemented in native code. All these implementations must be semantically identical to the implementations in Agave, with respect to active and considered feature gates.

Snapshot downloader

Mithril also uses a snapshot downloader to fetch an initial state. Mithril can only validate transactions for new blocks that it receives. In Solana, snapshots come in pairs, one larger one that is updated less frequently and one smaller one that is updated more frequently. Mithril downloads these snapshots from public RPC nodes and uses these to initialize and populate the AccountsDB.

RPC server

Mithril also provides an RPC server backend for querying information that is located in package `pkg/rpcserver`. It currently supports a few selected JSON-RPC methods, such as, e.g., `getAccountInfo` and `getBlockHeight`.

Methodology

The audit was carried out using a combination of manual code review and dynamic analysis through fuzzing, two complementary approaches that together maximize assurance in both the design soundness and runtime robustness of the system.

Design Review

The audit begins with the design review process, during which the Mithril codebase and its documentation are reviewed to build an initial understanding of the system. In this phase, we decoument all the system components. Further, we document all relevant assumptions and invariants of the system. Using these, we identify potential threat vectors and analyze where the system may be vulnerable. Finally, we may highlight points of improvements in the documentation and design.

Code Review

Manual code review remains the cornerstone of Runtime Verification's audit methodology. While no code review can guarantee the discovery of all vulnerabilities, this process enables systematic reasoning about program correctness, security, and consistency with the intended design.

The audit began with a high-level architectural study to understand Mithril's operational model, focusing on the logic governing node initialization, state replay, and runtime execution. This initial stage established a mental model of how Mithril interacts with Solana's consensus and account-state mechanisms, ensuring that our review proceeded with an accurate understanding of intended behavior.

We then performed a detailed inspection of the implementation, emphasizing correctness of invariants and assumptions rather than surface syntax or stylistic issues. Our reviewers traced data flow and control flow across the main execution paths to ensure that system states evolve consistently and that errors or exceptional conditions are handled deterministically.

Fuzzing and Dynamic Analysis

To complement the manual review, we conducted a structured fuzzing campaign aimed at validating runtime safety and fault tolerance under unexpected input conditions.

Fuzzing is a dynamic testing technique that feeds randomly or systematically generated inputs into a program to trigger rare or edge-case behaviors. In the context of this audit, fuzzing was used to exercise Mithril's data ingestion, parsing, and execution routines under malformed or boundary inputs, observing for crashes, panics, or invariant violations.

This process helps uncover issues that may not be evident through code inspection alone — such as logic errors in error handling, unintended state corruption, or subtle concurrency faults.

Our fuzzing campaign for Mithril encompasses two fuzzing methods:

- **Native Go Fuzzing**
- **Differential Fuzzing**

Our native Go fuzzing harness uses coverage-guided fuzzing using Go's built-in `testing/F` framework to find crashes, panics, and edge-case bugs in individual components. In addition to this, the differential fuzzing harness uses cross-implementation testing using the solfuzz framework to verify behavioral consistency between Mithril and Agave (the reference Rust implementation).

Limitations

The review process focused on correctness and safety of the implementation itself. While fuzzing and code inspection increase assurance, neither technique can guarantee complete absence of defects. The conclusions of this report therefore represent the best achievable confidence level within the defined scope and audit period.

Invariants

Design Review Invariants

Mithril is meant to run continously as a server, fetching new settled blocks and replaying them, thereby validating them. When Mithril starts from empty state, it has to start from a snapshot of the blockchain state since starting from the genesis block is not supported due to Mithril not being backwards-compatible with many blockchain upgrades. Additionally, it would take a very long time for the node to catchup when starting from Genesis block.

Therefore, Mithril's current validation process can be roughly modeled with the following high-level pseudo-code:

```
// Phase 1: setup AccountsDB
if (startFromEmptyState) {
  downloadSnapshotFromRPCNode()
  buildAccountsDBFromSnapshot()
} else {
  loadAccountsDB()
}
startRPCServer()
// Phase 2: replay blocks
forever {
  nextBlock = downloadNextSettledBlock()
  accountsDBChanges, err = replayAndValidateBlock(nextBlock)
  if err {
    // mismatch detected, look for another block
    continue
  }
  applyAccountsDBStateChanges(accountsDBChanges)
}
```

In phase 1, Mithril sets up the accounts database, AccountsDB, and possibly populates it with data taken from a recent snapshot of Solanas current state. For this purpose, Mithril downloads a snapshot from an un untrusted RPC server. Since this snapshot comes from an untrusted input, Mithril must validate the integrity of the content. In the past, this would have happened implicitely by replaying blocks until a full accounts hash calculation would have been incorporated into a blocks bankhash, which happened roughly once per Epoch. However, since this Epoch Accounts Hash has been replaced by a lattice hash based approach, this implicit check is not part of any bankhash anymore. Therefore, Mithril does not implicitely check the snapshots integrity. Instead, Mithril has to explicitely recalculate the lattice hash of the accounts at the beginning, which Mithril does not do. This has been described in more detail in finding [A28] Lattice hash of a snapshot is never validated by re-computing the hash.

In addition, Mithril starts an RPC server backend that can be used to query information about the Solana blockchain. This RPC server should only process information that has been confirmed as valid and final by Mithril. In particular, used snapshots must contain valid data, blocks must replay successfully with matching bankhashes, and provided data should only consider blocks, for which Mithril confirmed finality, i.e., blocks that has been confirmed by a supermajority of stake-weighted votes and at least 31 additional confirmed blocks have been built on top of it, reaching maximum lockout depth.

In phase 2, Mithril replays Solana blocks. For this purpose, Mithril fetches a settled block from an untrusted source such as an external RPC server. Mithril then replays this block, validates this block, and in case of success applies the state changes to AccountsDB.

While typical Solana full nodes replay multiple forks of Solana and only commit blocks to their states permanently once the respective blocks reach finality, Mithril ideally only replays and confirms block that reached finality. Blockchain forks appear, e.g., due to nodes both incorporating a proposed block from a slot and by skipping the same slot. Though, finality will (likely) only ever be reached on one fork due to economic costs for validator nodes of switching forks. Since votes, which are incorporated as transactions in blocks, implicitly vote on past blocks of that fork, Mithril might only be able to confirm a supermajority of votes by processing more than one

block. In addition, finality of a block can only be confirmed by confirming supermajority of votes for 31 additional blocks on top of the respective block.

Therefore, Mithril currently relies on untrusted RPC servers that provide settled blocks to be truthful and correct, which is an assumption that should not be relied upon by Mithril. This is further described in finding [A27] RPC server uses AccountsDB data from blocks for which Mithril could not yet confirm finality.

Finally, in order for Mithril to properly replay and validate blocks, the block replaying that Mithril implements must be semantically identical to the Agave Solana node, which is the canonical implementation of Solana. Here, Agave's implementation is the specification, including all un-intended software bugs in Agave.

Invariants Generated by the RV AI Audit Assistant

Using novel in-house AI tooling at Runtime Verification, we generated pre-conditions, post-conditions, invariants, side-effects, and short summaries for functions and structs in the Mithril Go codebase. Since these are AI generated, they require manual validation before they can be considered valid. On top of that, they are also not exhaustive.

We validated a selected set of these specifications and published them in the git branch `rv-audit-assist` in our GitHub fork of Mithril. Additionally, validating the correctness of these specifications led to a finding, namely [A17] Missing input validation can lead to `endSlot < startSlot`.

Findings

The following are findings that contradict the protocol's desired behavior, violate invariants, impact user experience, or may lead to crashes or deadlocks.

[A01] Path Traversal in Tar Extraction of Append Vector Files

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

Description

In the `readTar` function, the tar header name is used directly without validation:

```
if !isAppendVec(header.Name) {
    continue
}
// ... copy data ...
task := appendVecCopyingTask{TarBuffer: writer, Filename: header.Name}
```

The `header.Name` is passed directly to the worker pool.

In `appendVecCopyingPool` worker, this unsanitized filename is used with `filepath.Join`:

```
filename := task.Filename
outFilename := filepath.Join(accountsDbDir, filename)
err := os.WriteFile(outFilename, appendVecBytes, 0644)
```

Scenarios

A malicious snapshot archive could contain entries like:

- Relative path traversal: `"accounts/../../etc/passwd"`
- Absolute path: `"/etc/passwd"`
- Multiple traversals: `"accounts/../../../../../root/.ssh/authorized_keys"`

When passed to `filepath.Join(accountsDbDir, filename)`, these would resolve to paths outside `accountsDbDir`, allowing arbitrary file writes with the permissions of the Mithril process. This is a ZipSlip (CWE-22) vulnerability that can lead to the arbitrary overwriting of files.

The Same Vulnerability Exists in Multiple Places
This pattern appears in three separate functions:

- `readTar` in `pkg/snapshot/build_db.go`
- `readTarIncr` in `pkg/snapshot/build_db_with_incr.go`
- `UnmarshalManifestFromSnapshot` in `pkg/snapshot/snapshot.go`

All three use `header.Name` without sanitization when constructing output paths.

Why `isAppendVec` doesn't prevent this
The `isAppendVec` check only validates that the path contains `"accounts/"` and `"."`.

```
func isAppendVec(filename string) bool {
    return strings.Contains(filename, "accounts/") && strings.Contains(filename, ".")
}
```

This check is insufficient because:

- `"accounts/../../etc/passwd"` contains both `"accounts/"` and `"."`
- It doesn't validate that the path stays within the intended directory
- It doesn't check for absolute paths

Impact

This vulnerability is critical because:

- Snapshot files are downloaded from external sources - the `BuildAccountsDbWithIncr` function downloads snapshots from `api.mainnet-beta.solana.com`
- Arbitrary file write - An attacker controlling the snapshot source could overwrite any file writable by the Mithril process
- Privilege escalation potential - Could overwrite configuration files, SSH keys, or other sensitive files
- No user interaction required - The vulnerability triggers automatically during snapshot processing

---

Recommendation

The code should validate that the resolved path stays within `accountsDbDir`:

```
// In appendVecCopyingPool worker
filename := task.Filename
outFilename := filepath.Join(accountsDbDir, filename)

// Validate the path doesn't escape accountsDbDir
cleanPath := filepath.Clean(outFilename)
if !strings.HasPrefix(cleanPath, filepath.Clean(accountsDbDir)+string(os.PathSeparator)) {
    mlog.Log.Errorf("invalid path in tar archive: %s", filename)
    appendVecCopyingInProgress.Add(-1)
    return
}

err := os.WriteFile(cleanPath, appendVecBytes, 0644)
```

Status

Addressed in commit ce46067.

[A02] Unbounded memory allocation during deserialization of vectors

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

Description

Multiple deserialization functions in the gossip schema read untrusted length values and immediately allocate slices of that size without any validation.

The Vulnerable Functions

1. `deserialize_vector_CompressedSlots` - this function reads a length from the deserializer and immediately allocates a slice. An attacker could provide `length = 2^64 - 1`, causing Go to attempt allocating an enormous slice, exhausting memory, and crashing the node process.

```
length, err := deserializer.DeserializeLen()
if err != nil {
    return nil, err
}
obj := make([]CompressedSlots, length)
```

2. `deserialize_vector_CrdsValue` - same pattern - reads `length` and allocates without validation.
3. `deserialize_vector_Pubkey` - same vulnerability.
4. `deserialize_vector_SlotHash` - same vulnerability.
5. `deserialize_vector_u64` - same vulnerability.
6. `deserialize_vector_u8` - same vulnerability.

Scenario

These deserialization functions are used to parse gossip messages from untrusted network peers. For example, `EpochSlots` contains a `Slots` field of type `[]CompressedSlots` which gets deserialized using `deserialize_vector_CompressedSlots`.
An attacker could:

1. Craft a malicious gossip message with an enormous length field (e.g., length = 10^9)
2. Send it to the Mithril node
3. The node attempts to allocate gigabytes of memory
4. The process crashes due to OOM (out of memory)

Recommendation

This is a classic unbounded allocation vulnerability (CWE-789) in deserialization code. All six vulnerable functions follow the same pattern and require the same fix: adding reasonable upper bounds for each vector type. The bounds should be chosen based on realistic maximum sizes for each data type in the Solana gossip protocol.

```
const (
    MaxCompressedSlotsLen = 10000
    MaxCrdsValueLen = 1000
    MaxPubkeyLen = 10000
    MaxSlotHashLen = 10000
    MaxU64VecLen = 100000
```

```
    MaxU8VecLen = 10485760  // 10MB
)

func deserialize_vector_CompressedSlots(deserializer serde.Deserializer) ([]CompressedSlots, error) {
    length, err := deserializer.DeserializeLen()
    if err != nil {
        return nil, err
    }
    if length > MaxCompressedSlotsLen {
        return nil, fmt.Errorf("CompressedSlots vector length %d exceeds maximum %d", length, MaxCompressedSlotsLen)
    }
    obj := make([]CompressedSlots, length)
    // ... rest of function
}
```

[A03] Missing signature validation leading to potential malleability

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

Description

The `Secp256k1ProgramExecute` implementation in `pkg/sealevel/secp256k1_program.go` decodes and uses the provided signature directly in `secp256k1.RecoverPubkey` without performing canonical signature validation checks.

```
signature, err := Secp256k1GetDataSlice(txCtx, uint16(secpOffsets.SignatureInstructionIndex), secpOffsets.SignatureOffset,
SignatureSerializedSize+1)
// ... no validation here ...
recoveredPubkeyBytes, err := secp256k1.RecoverPubkey(messageHash, signature)
```

The Validation Function Exists But Isn't Used

The `parseAndValidateSignature` function implements the necessary checks:

- Size validation: Ensures signature is exactly 64 bytes
- Overflow checks: Uses `isSignatureOverflowing` to verify both `r` (bytes 0-32) and `s` (bytes 32-64) are less than the curve order `SECP256K1_N`

Although the helper function `parseAndValidateSignature` enforces these checks, it is never invoked in `Secp256k1ProgramExecute`.

Security Impact

Without these checks, `Secp256k1ProgramExecute` can accept:

- Non-canonical signatures where `r` or `s` >= curve order
- High-S signatures (`s` values in the upper half of the range)

This creates signature malleability - the same message can have multiple valid signatures that recover to the same public key, which can lead to:

- Transaction replay attacks
- Inconsistent signature verification across different implementations
- Potential consensus issues if different validators accept different signature forms

Recommendation

Before recovering the public key, invoke `parseAndValidateSignature` on the 65-byte signature slice to enforce length and overflow checks.

```
signature, err := Secp256k1GetDataSlice(txCtx, uint16(secpOffsets.SignatureInstructionIndex), secpOffsets.SignatureOffset,
SignatureSerializedSize+1)
err = parseAndValidateSignature(signature)
if err != nil {
    return syscallSuccess(3) // Secp256k1RecoverError::InvalidSignature
}
// ...
recoveredPubkeyBytes, err := secp256k1.RecoverPubkey(messageHash, signature)
```

[A04] Static SBPF verifier bypassed

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

Description

The static SBPF bytecode verifier is stubbed out and never actually called. In `pkg/sbpf/program.go`, the `Verify` method always returns `nil` and does not invoke `NewVerifier(p).VerifyProgram()`. The `Verify()` method is called in the program deployment flow, effectively disabling all bytecode validation.

```
func (p *Program) Verify() error {
    return nil
    //return NewVerifier(p).VerifyProgram()
}
```

Security Implications

- **Invalid Opcodes:** Programs with undefined or malformed opcodes can be loaded and will only fail at execution time, potentially causing panics
- **Out-of-Bounds Jumps:** Jump instructions with invalid target addresses can cause the interpreter to access invalid memory or execute arbitrary code
- **Division by Zero:** Programs with division/modulo by zero won't be caught until runtime, causing panics that could DoS the validator
- **Invalid Syscalls:** Programs can reference non-existent syscalls that will only fail during execution
- **Memory Safety:** Without verification, programs can have memory access patterns that violate safety guarantees

Recommendation

Restore the call to the static bytecode verifier in `Program.Verify` by uncommenting `return NewVerifier(p).VerifyProgram()`. Ensure that all SBPF bytecode is validated before execution.

[A05] `MinimumBalance` function might overflow returning an invalid result

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

Description

The `MinimumBalance` function in `pkg/sealevel/sysvar_rent.go` should prevent overflows, but currently it does not.

The function performs this calculation:

```
func (sr *SysvarRent) MinimumBalance(dataLen uint64) uint64 {
    min := float64(((dataLen + rentAccountStorageOverhead) * sr.LamportsPerUint8Year) * uint64(sr.ExemptionThreshold))
    return uint64(min)
}
```

The Overflow Risk

1. Addition overflow: dataLen + rentAccountStorageOverhead (where `rentAccountStorageOverhead = 128`) could overflow if `dataLen` is close to `math.MaxUint64`
2. Multiplication overflow: the result multiplied by `LamportsPerUint8Year` could overflow
3. Second multiplication: Multiplying by `ExemptionThreshold`

This function is called throughout the codebase in critical validation paths:

- Vote Program initialization
- Stake Program initialization
- System Program nonce initialization
- BPF Loader deployment
- Address Lookup Table creation

Impact

An incorrect minimum balance calculation could allow underfunded accounts to be created or cause incorrect rent exemption checks.
The overflow risk varies across different native programs depending on whether users can control the account data length.
The overflow vulnerability is most concerning in:

- Loader V4 - Users directly control the newLen parameter
- BPF Loaders - Users control program sizes when deploying/upgrading
- Rent Collection - Processes arbitrary accounts with potentially large data

The vulnerability is less concerning in:

- Vote/Stake/Nonce Programs - Fixed data sizes prevent exploitation
- Address Lookup Tables - Practical limits on number of addresses

Recommendation

The function should use saturating arithmetic from the `safemath` package:

```
func (sr *SysvarRent) MinimumBalance(dataLen uint64) uint64 {
    dataLenWithOverhead := safemath.SaturatingAddU64(dataLen, rentAccountStorageOverhead)
    lamportsPerYear := safemath.SaturatingMulU64(dataLenWithOverhead, sr.LamportsPerUint8Year)
    min := float64(lamportsPerYear) * sr.ExemptionThreshold
```

```
    // Saturate at MaxUint64 if the result exceeds it
    if min > float64(math.MaxUint64) {
        return math.MaxUint64
    }
    return uint64(min)
}
```

This would prevent silent overflow and ensure the calculation saturates at `math.MaxUint64` rather than wrapping around.

[A06] Denial of Service via Untrusted RPC-Controlled Reward Partitions

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

Description

The function `DeterminePartitionedStakingRewardsInfo` (`pkg/rewards/rewards.go`) and its use in `beginPartitionedEpochRewardsDistribution` (`pkg/replay/rewards.go`) blindly trust values from the `RpcClient` :

- `numRewardPartitions` from `GetNumRewardPartitions`
- `rewardSlots` from `GetStakingRewardSlots`

Scenario

An attacker controlling or spoofing the RPC endpoint could return an excessively large `numRewardPartitions` or a malformed/empty `rewardSlots` array, which could cause:

- Out-of-memory or CPU exhaustion in subsequent calls to `CalculateTotalPointsAndPartitions` , which allocates data structures sized by `numRewardPartitions` .
- Index out-of-bounds panic when accessing `rewardSlots[len(rewardSlots)-1]` if the slice is empty.

Exploitation Path:

1. **Attacker controls RPC endpoint:** The node operator configures `--rpc-endpoint` to point to a malicious server
2. **Malicious responses:** At epoch boundary, the attacker returns:

- `numRewardPartitions = math.MaxUint64` (or any very large number)
- `rewardSlots = []` (empty array)

3. **Node crashes:** Either from the empty slice panic or memory exhaustion

This vulnerability is particularly concerning because:

- The RPC endpoint is user-configurable via command-line flags
- The attack triggers automatically at epoch boundaries (every ~2 days)
- Both panic and memory exhaustion would crash the entire node
- There is no recovery mechanism - the node would crash repeatedly at each epoch boundary

The code assumes the RPC endpoint is trustworthy, which is a dangerous assumption for a verification/auditing tool that may connect to third-party RPC providers.

Recommendation

Validate and bound-check values returned by external RPC calls before using them. For example:

- Enforce a maximum ceiling on `numRewardPartitions` (e.g., based on Solana protocol limits).
- Verify `rewardSlots` is non-empty before indexing. Return a clear error instead of panicking on malformed RPC data.
- Handle RPC errors gracefully to avoid unintended crashes or resource exhaustion.

```
// In DeterminePartitionedStakingRewardsInfo
numRewardPartitions, err := rpcc.GetNumRewardPartitions(firstSlotInEpoch)
```

```
if err != nil {
    panic(err)
}
// ADD: Validate numRewardPartitions is reasonable
if numRewardPartitions > 10000 { // reasonable upper bound
    panic(fmt.Sprintf("numRewardPartitions too large: %d", numRewardPartitions))
}

rewardSlots, err := rpcc.GetStakingRewardSlots(firstSlotInEpoch, numRewardPartitions)
if err != nil {
    panic(err)
}
// ADD: Validate rewardSlots is not empty
if len(rewardSlots) == 0 {
    panic("rewardSlots is empty")
}
```

Status

Addressed in commit 897d05b.

[A07] Unchecked error on return data length mismatch in `SyscallGetReturnDataImpl`

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

Description

In `SyscallGetReturnDataImpl` (`pkg/sealevel/syscalls_call.go`), when the length of the guest-provided buffer (`returnDataResult`) does not match the expected return data length, the code calls `syscallErr(SyscallErrInvalidLength)` but does not return immediately. As a result, the subsequent copy operation will proceed with mismatched buffer lengths, potentially causing out-of-bounds memory writes or unintended data truncation and fails to properly signal an error to the guest program.

```
if len(returnData) != len(returnDataResult) {
    syscallErr(SyscallErrInvalidLength)  // Error signaled but no return!
}

copy(returnDataResult, returnData)  // This still executes
```

When the length mismatch occurs:

- The error is logged but not returned to the caller
- The copy proceeds anyway with mismatched buffer sizes
- Go's `copy()` copies `min(len(dst), len(src))` bytes, so it won't panic, but:
  - If returnDataResult is smaller, data gets silently truncated
  - The guest program receives incomplete data without knowing an error occurred

Recommendation

The code should immediately return after signaling the error:

```
if len(returnData) != len(returnDataResult) {
    return syscallErr(SyscallErrInvalidLength)
}
```

Note:
This appears to be a defensive check that should theoretically never trigger if `vm.Translate()` works correctly. However, if it does trigger (due to a bug in the VM or memory translation), the current code silently proceeds with potentially unsafe memory operations instead of properly failing. The error constant `SyscallErrInvalidLength` is defined in the errors file and should be returned to the guest program to signal the failure.

Status

Addressed in commit e5cb413.

[A08] ELF Loader - Missing Text Section Size Validation

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

Description

The ELF loader in `pkg/sbpf/loader` panics when loading malformed ELF files with `.text` section sizes that are not divisible by 8, instead of gracefully rejecting them with an error.

Location

- **File**: `pkg/sbpf/loader/loader.go`
- **Function**: `parseSlots()` (line 222)
- **Trigger point**: `getProgram()` → `parseSlots(l.text)` (line 254)
- **Missing validation**: `copy.go` functions `getText()` and `checkSectionAddrs()`

Issue

The `parseSlots` function requires that SBPF instruction data be divisible by 8 bytes (each instruction is 8 bytes), but this constraint is not validated during ELF parsing. When a malformed ELF file contains a `.text` section with a size not divisible by 8, the code panics instead of returning a validation error.

Vulnerable Code Path

```
// In copy.go - getText() extracts .text section WITHOUT size validation
func (l *Loader) getText() error {
    if err := l.checkSectionAddrs(l.shText); err != nil {
        return fmt.Errorf("invalid .text: %w", err)
    }
    l.textRange = addrRange{min: l.shText.Off, max: l.shText.Off + l.shText.Size}
    // ❌ Missing: validation that l.shText.Size % 8 == 0
    return nil
}

// In copy.go - copySections() creates text slice
func (l *Loader) copySections() error {
    // ... allocation and copying ...
    l.text = l.getRange(l.textRange)  // Sets l.text to section data
    return nil
}

// In loader.go - getProgram() calls parseSlots without size check
func (l *Loader) getProgram() *sbpf.Program {
    return &sbpf.Program{
        // ...
        Text: parseSlots(l.text),  // ❌ Panics if len(l.text) % 8 != 0
        // ...
    }
}

// In loader.go - parseSlots panics on invalid input
func parseSlots(bs []byte) []sbpf.Slot {
    if len(bs)%8 != 0 {
        panic(fmt.Sprintf("expected len(bs)=%d to be divisible by 8", len(bs)))
        // ❌ Should return error, not panic
    }
```

```
    // ...
}
```

Impact

- **DoS vulnerability**: Malicious ELF files can crash the loader/validator
- **Violation of error handling best practices**: Input validation should return errors, not panic
- **Security issue**: Panics from untrusted input are exploitable for denial-of-service
- **Late detection**: The error is detected in `parseSlots()` after the ELF has been parsed and copied into memory, when it should be caught during initial validation

Fuzzing Discovery

- Fuzz test: `FuzzELFLoadWithValidBase`
- Failing input corpus: `testdata/fuzz/FuzzELFLoadWithValidBase/4305f3a1676e222f`
- Trigger:
  - ELF file with `.text` section size of **121 bytes** (0x79)
  - 121 % 8 = 1 (not divisible by 8)

Error output:

```
panic: expected len(bs)=121 to be divisible by 8

goroutine 27 [running]:
github.com/Overclock-Validator/mithril/pkg/sbpf/loader.parseSlots({0xc0002b2000, 0x79, 0x2000})
    pkg/sbpf/loader/loader.go:224 +0x1df
github.com/Overclock-Validator/mithril/pkg/sbpf/loader.(*Loader).getProgram(0xc0002b4008)
    pkg/sbpf/loader/loader.go:254 +0x51
github.com/Overclock-Validator/mithril/pkg/sbpf/loader.(*Loader).Load(0xc0002b4008)
    pkg/sbpf/loader/loader.go:201 +0x145
```

Root Cause

The `.text` section size validation is missing from the ELF parsing pipeline. A valid SBPF `.text` section must contain an integer number of 8-byte instructions, but this invariant is only checked (via panic) in `parseSlots()` after all parsing and copying is complete.

Why This is a Bug (Not a Test Issue)

1. Invalid ELF structure: A SBPF ELF with `.text` section size not divisible by 8 is malformed and should be rejected during parsing
2. Panic is inappropriate: Using `panic()` for input validation in production code is a security vulnerability
3. Defense in depth violation: Validation should happen at the earliest possible point (during ELF header/section validation)
4. Inconsistent with other validations: The loader already validates many ELF properties and returns errors (not panics) for invalid files

---

Suggested Fixes

Option 1: Validate in `getText()` (Recommended)

Add validation when extracting the `.text` section range:

```go
// In pkg/sbpf/loader/copy.go
func (l *Loader) getText() error {
    if err := l.checkSectionAddrs(l.shText); err != nil {
        return fmt.Errorf("invalid .text: %w", err)
    }
```

```
    // ✅ Validate that .text size is divisible by 8
    if l.shText.Size % 8 != 0 {
        return fmt.Errorf(".text section size must be divisible by 8 (SBPF instruction size), got %d", l.shText.Size)
    }

    l.textRange = addrRange{min: l.shText.Off, max: l.shText.Off + l.shText.Size}
    return nil
}
```

Pros:

- Validates at the earliest appropriate point
- Consistent with other section validation in `checkSectionAddrs()`
- Clear error message for debugging
- No changes needed to `parseSlots()`

Option 2: Validate in `checkSectionAddrs()`

Add the validation to the general section address checking function:

```
// In pkg/sbpf/loader/copy.go
func (l *Loader) checkSectionAddrs(sh *elf.Section64) error {
    // ... existing validation ...

    // ✅ For executable sections, validate instruction alignment
    if sh.Flags & uint64(elf.SHF_EXECINSTR) != 0 {
        if sh.Size % 8 != 0 {
            return fmt.Errorf("executable section size must be divisible by 8 (SBPF instruction size), got %d", sh.Size)
        }
    }

    return nil
}
```

Pros:

- More general - validates all executable sections
- Centralizes size validation logic
- Reusable if other executable sections are added

Option 3: Convert `parseSlots()` to Return Error

Change `parseSlots()` to return an error instead of panicking:

```
// In pkg/sbpf/loader/loader.go
func parseSlots(bs []byte) ([]sbpf.Slot, error) {
    if len(bs)%8 != 0 {
        return nil, fmt.Errorf("expected instruction data length to be divisible by 8, got %d", len(bs))
    }
    out := make([]sbpf.Slot, len(bs)/8)
    for i := 0; i < len(bs); i += 8 {
        out[i/8] = sbpf.Slot(binary.LittleEndian.Uint64(bs[i : i+8]))
    }
    return out, nil
}
```

```go
// Update getProgram() to handle error
func (l *Loader) getProgram() (*sbpf.Program, error) {
    text, err := parseSlots(l.text)
    if err != nil {
        return nil, err
    }
    return &sbpf.Program{
        RO:          l.program,
        Text:        text,
        TextVA:      sbpf.VaddrProgram + l.textRange.min,
        Entrypoint:  l.entrypoint,
        Funcs:       l.funcs,
        SbpfVersion: sbpfver.SbpfVersion{Version: l.eh.Flags},
    }, nil
}

// Update Load() to propagate error
func (l *Loader) Load() (*sbpf.Program, error) {
    if err := l.parse(); err != nil {
        return nil, err
    }
    if err := l.copy(); err != nil {
        return nil, err
    }
    if err := l.relocate(); err != nil {
        return nil, err
    }
    return l.getProgram()  // Now returns (prog, error)
}
```

Pros:

- Eliminates panic completely
- More Go-idiomatic error handling
- Better separation of concerns

Cons:

- Requires changes to multiple function signatures
- More invasive refactoring

Recommendation

Use Option 1 (validate in `getText()`):

- Minimal code changes
- Validates at the right abstraction level
- Clear error message
- Catches the error early in the loading pipeline
- Preserves `parseSlots()` as an internal helper that can safely assume valid input

---

Testing

After implementing the fix, verify:

```
# Test the specific failing corpus entry
go test ./pkg/sbpf/loader -run=FuzzELFLoadWithValidBase/4305f3a1676e222f -v

# Run fresh fuzzing to ensure no panics
go test ./pkg/sbpf/loader -fuzz=FuzzELFLoadWithValidBase -fuzztime=5m

# Verify the error message
# Expected: error containing ".text section size must be divisible by 8"
# Not: panic
```

References

- **Fuzzing corpus**: `pkg/sbpf/loader/testdata/fuzz/FuzzELFLoadWithValidBase/4305f3a1676e222f`
- **Related code**:
  - `pkg/sbpf/loader/loader.go`: `Load()`, `getProgram()`, `parseSlots()`
  - `pkg/sbpf/loader/copy.go`: `copy()`, `getText()`, `checkSectionAddrs()`
  - `pkg/sbpf/loader/loader_fuzz_test.go`: `FuzzELFLoadWithValidBase()`

Severity

High - This is a DoS vulnerability that allows malicious ELF files to crash the validator. Any network peer or malicious program deployer could trigger this panic with a crafted ELF file.

Status

Addressed in commit e7e4b57.

[A09] Off-by-one bounds check in `IsInstructionAccountSigner` and `IsInstructionAccountWritable` leading to potential panic

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

Description

The methods `IsInstructionAccountSigner` and `IsInstructionAccountWritable` incorrectly check if `instrAcctIdx` is greater than `len(instrCtx.InstructionAccounts)` rather than greater than `len(instrCtx.InstructionAccounts) - 1`. When `instrAcctIdx == len(instrCtx.InstructionAccounts)`, the out-of-bounds index is missed, and the code proceeds to access `instrCtx.InstructionAccounts[instrAcctIdx]`, causing a runtime panic.

```
func (instrCtx *InstructionCtx) IsInstructionAccountSigner(instrAcctIdx uint64) (bool, error) {
    if len(instrCtx.InstructionAccounts) == 0 || instrAcctIdx > uint64(len(instrCtx.InstructionAccounts)) {
        return false, InstrErrMissingAccount
    }

    return instrCtx.InstructionAccounts[instrAcctIdx].IsSigner, nil
}

func (instrCtx *InstructionCtx) IsInstructionAccountWritable(instrAcctIdx uint64) (bool, error) {
    if len(instrCtx.InstructionAccounts) == 0 || instrAcctIdx > uint64(len(instrCtx.InstructionAccounts)) {
        return false, InstrErrMissingAccount
    }

    return instrCtx.InstructionAccounts[instrAcctIdx].IsWritable, nil
}
```

Recommendation

Change both methods to use the correct bounds check:

```
func (instrCtx *InstructionCtx) IsInstructionAccountSigner(instrAcctIdx uint64) (bool, error) {
    if len(instrCtx.InstructionAccounts) == 0 || instrAcctIdx >= uint64(len(instrCtx.InstructionAccounts)) {
        return false, InstrErrMissingAccount
    }
    return instrCtx.InstructionAccounts[instrAcctIdx].IsSigner, nil
}

func (instrCtx *InstructionCtx) IsInstructionAccountWritable(instrAcctIdx uint64) (bool, error) {
    if len(instrCtx.InstructionAccounts) == 0 || instrAcctIdx >= uint64(len(instrCtx.InstructionAccounts)) {
        return false, InstrErrMissingAccount
    }
    return instrCtx.InstructionAccounts[instrAcctIdx].IsWritable, nil
}
```

Status

Fixed in commit ebc579b.

[A10] Undercharging compute units for Ristretto scalar multiplication

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

Description

In the `handleRistrettoCurveGroupOps` function, the `CurveOpMul` case (scalar multiplication) incorrectly charges for an addition operation:

```
case CurveOpMul:
    {
        err := execCtx.ComputeMeter.Consume(cu.CUCurve25519RistrettoAddCost)  // WRONG!
        if err != nil {
            return syscallCuErr()
    }
```

This should be `CUCurve25519RistrettoMulCost` instead.

Security Impact: Denial-of-Service Vulnerability

- Scalar multiplication is computationally expensive - it involves multiple point additions and is significantly more costly than a single addition operation
- Undercharging allows resource exhaustion - A malicious SBPF program can perform many scalar multiplications while consuming compute units as if they were simple additions
- CPU exhaustion - The attacker can exhaust the host node's CPU resources by repeatedly calling the Ristretto multiplication syscall, causing the node to become unresponsive

Recommendation

This is a copy-paste error where the multiplication case was likely copied from the addition case, but the compute unit cost wasn't updated.
Use `execCtx.ComputeMeter.Consume(cu.CUCurve25519RistrettoMulCost)` instead of `cu.CUCurve25519RistrettoAddCost` in the `CurveOpMul` case within `handleRistrettoCurveGroupOps` to correctly bill the cost of Ristretto scalar multiplication.

```
err := execCtx.ComputeMeter.Consume(cu.CUCurve25519RistrettoMulCost)
```

Status

Fixed in commit 326b106.

[A11] Unbounded entries in `SysvarRecentBlockhashes` Unmarshal

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

Description

The `UnmarshalWithDecoder` method in `pkg/sealevel/sysvar_recent_blockhashes.go` reads the number of blockhash entries from on-chain data but does not enforce the `recentBlockhashesMaxEntries` limit (150). A malicious or corrupted sysvar account with a forged count could declare a very large number of entries, causing the node to allocate excessive memory or spend excessive CPU time in the loop.

Attack Scenario:

1. A malicious or corrupted sysvar account provides `numBlockhashes = 2^64 - 1` (or any very large value)
2. The loop attempts to iterate billions of times, each iteration:
   - Reading 32 bytes for the `blockhash`
   - Reading 8 bytes for `LamportsPerSignature`
   - Appending a `RecentBlockHashesEntry` struct (40 bytes) to the slice
3. This causes either:
   - Memory exhaustion: Go's slice append will repeatedly reallocate and copy, consuming gigabytes of memory
   - CPU exhaustion: Even if memory doesn't run out immediately, the loop will run for an extremely long time

Recommendation

Add validation before the loop:

```
func (recentBlockhashes *SysvarRecentBlockhashes) UnmarshalWithDecoder(decoder *bin.Decoder) error {
   numBlockhashes, err := decoder.ReadUint64(bin.LE)
   if err != nil {
       return err
   }

   // ADD: Validate against maximum
   if numBlockhashes > recentBlockhashesMaxEntries {
       return fmt.Errorf("recent blockhashes count %d exceeds maximum %d", numBlockhashes, recentBlockhashesMaxEntries)
   }

   for count := uint64(0); count < numBlockhashes; count++ {
       // ... rest of function
   }
}
```

[A12] Incorrect dynamic table entry count calculation

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

Description

In `newDynIter` (`parse.go`), the number of dynamic table entries is calculated as `count: uint32(off / dynLen)` (offset divided by entry size) instead of using the section size (`size / dynLen`). This logic error can cause the loader to iterate well past the actual dynamic section boundaries, leading to out-of-bounds reads and premature EOF errors when parsing the dynamic table. A malformed program could exploit this to prematurely terminate loading (denial of service) or trigger unexpected parsing behavior.

```
iter := &dynTableIter{
        l:        l,
        off:      off,
        count:    uint32(off / dynLen),  // BUG: should be size / dynLen
        elemSize: dynLen,
    }
```

Scenario

The number of entries should be calculated as section size divided by entry size, not section offset divided by entry size. Using the offset means:

- **Incorrect count**: If the dynamic section starts at offset 0x1000 (4096 bytes), the code would calculate count = 4096 / 16 = 256 entries, regardless of the actual section size
- **Out-of-bounds iteration**: The iterator will attempt to read far beyond the actual dynamic section boundaries

Impact

The `dynTableIter.getNext()` method checks `if it.i >= it.count` to determine when to stop iterating. With an incorrect count, it will:

- Continue reading past the end of the dynamic section
- Attempt to read from invalid file offsets
- Eventually hit `EOF` or read garbage data
- Potentially cause premature termination with `io.ErrUnexpectedEOF`

Recommendation

Compute the entry count using the dynamic section size

```
iter := &dynTableIter{
        l:        l,
        off:      off,
        count:    uint32(size / dynLen),
        elemSize: dynLen,
    }
```

Status

Fixed in commit fc85c30.

[A13] Denial of Service via Unbounded Memory Allocation and Panic on Decode Failure in StakeHistory Sysvar

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

Description

The method `UnmarshalWithDecoder` reads an untrusted `entriesLen` value from the sysvar stake history account data and allocates/loops over that many entries without enforcing the `SysvarStakeHistoryMaxEntries` limit. A malicious or corrupted sysvar account can supply a very large length, leading to excessive memory allocation (CWE-789) or long execution times, potentially exhausting resources. Additionally, its wrapper `MustUnmarshalWithDecoder` panics on any decode error, causing the node to crash (CWE-703) when encountering malformed data.

Mitigation

- **Sysvar Account Protection:** The stake history sysvar account has a well-known address (`SysvarStakeHistory1111111111111111111111111`) and is owned by the sysvar program. Normal transactions cannot directly modify sysvar accounts - they can only be updated through specific system operations.
- **Requires Account Corruption:** An attacker would need to either:
  - Exploit a separate vulnerability that allows writing to sysvar accounts
  - Corrupt the AccountsDB storage layer directly (requires file system access)
  - Provide a malicious snapshot with corrupted sysvar data
- **Snapshot Loading Vector:** The most realistic attack vector is through malicious snapshots. When loading a snapshot, the stake history is deserialized from the snapshot manifest. If a node loads a snapshot from an untrusted source, the corrupted stake history could trigger the vulnerability.

Recommendation

Before allocating or iterating, validate that `entriesLen` does not exceed `SysvarStakeHistoryMaxEntries`. For example:

```
if entriesLen > SysvarStakeHistoryMaxEntries {
    return fmt.Errorf("too many stake history entries: %d", entriesLen)
}
```

Remove the panic in `MustUnmarshalWithDecoder` and propagate errors instead, allowing callers to handle decode failures gracefully without crashing the process.

[A14] `updateCallerAccount` zeroes beginning of truncated account data instead of ending after CPI

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

Description

The function `updateCallerAccount` updates accounts after a cross-program invocation (CPI). If the data of an account is smaller after a CPI compared to before, the truncated data is zeroed out by `updateCallerAccount`.

However, the function makes use of the enumerator variable of a for loop, which always begins at index 0. This causes the beginning of account data to be zeroed out instead of the ending, as was most likely intended by the code:

```
if postLen < prevLen {
    if uint64(len(callerAcct.SerializedData)) < postLen {
        return InstrErrAccountDataTooSmall
    }
    for i := range callerAcct.SerializedData[postLen:] {
        callerAcct.SerializedData[i] = 0
    }
}
```

Recommendation

Add an offset to the index of `SerializedData`, such that the ending instead of beginning of account data is zeroed out.

```
for i := range callerAcct.SerializedData[postLen:] {
    callerAcct.SerializedData[postLen+i] = 0  // Add postLen offset
}
```

Or more idiomatically:

```
for i := postLen; i < uint64(len(callerAcct.SerializedData)); i++ {
    callerAcct.SerializedData[i] = 0
}
```

Status

Addressed in commit ca643b9.

[A15] Unbounded memory allocation reading TAR entries causing Denial-of-Service

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

Description

The tar file reading code in functions `pkg/snapshot/build_db.go:readTar` and `pkg/snapshot/build_db_with_incr.go:readTarIncr` preallocate memory based on the size of a TAR file entry, see e.g.:

```
header, err := tarReader.Next()
if err == io.EOF {
    break
} else if err != nil {
    mlog.Log.Errorf("reading next tar: %s\n", err)
    return err
}

if !isAppendVec(header.Name) {
    continue
}

writer := bytes.NewBuffer(make([]byte, 0, header.Size))
```

A maliciously crafted TAR file could contain size fields that exceed available memory on the running machine, causing an out-of-memory crash, thereby causing Denial-of-Service.

Recommendation

Enforce a reasonable upper bound on `header.Size` before allocating memory by, e.g., rejecting entries above a maximum size.

[A16] `isOverlap` function does not cover all cases of overlapping regions

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

Description

The function `pkg/sbpf/loader/parse.go:isOverlap` takes two offsets and two sizes that describe two intervals, respectively. It then checks whether these intervals are overlapping. This is used during ELF parsing to check whether sections and headers are overlapping.

This is the current implementation of `isOverlap`:

```
func isOverlap(startA uint64, sizeA uint64, startB uint64, sizeB uint64) bool {
    if startA > startB {
        startA, sizeA, startB, sizeB = startB, sizeB, startA, sizeA
    }
    endA, endB := startA+sizeA, startB+sizeB
    if endA < startA || endB < startB {
        panic("isOverlap: integer overflow")
    }
    return sizeA != 0 && sizeB != 0 && (startA == startB || endA > endB)
}
```

In case both intervalls have a non-zero size, it checks whether either the starting offsets are identical `startA == startB`, or, whether the end of the first intervall points to after the end of the second interval.

This second condition, `endA > endB`, is wrong, as it will fail to catch the case where the first interval ends in the middle of the second interval.

Recommendation

We recommend to change the condition to `return sizeA != 0 && sizeB != 0 && (startA == startB || endA > startB)`. This way, it is instead checked whether the first interval ends after the beginning of the second interval.

Status

To be discussed with the client

[A17] Missing input validation can lead to `endSlot < startSlot`

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

Description

The function `runVerifier` can receive two different parameters to specify the range of slots to be verified:

```
numReplaySlots    int64
endSlot           int64
```

The values are updated and checked in the `runVerifier` function as following:

runtimeverification/_audits_Overclock-Validator_mithril/cmd/mithril/node/node.go

Line 217 to 231 in 06f7b95

```
217   startSlot := int64(manifest.Bank.Slot + 1)
218   if endSlot != -1 {
219       numReplaySlots = endSlot - startSlot
220   } else if numReplaySlots != 0 {
221       endSlot = startSlot + numReplaySlots
222   }
223
224   // just processing the snapshot - not executing blocks.
225   if numReplaySlots == 0 {
226       return
227   }
228
229   if endSlot != -1 && endSlot < startSlot {
230       klog.Fatalf("end slot cannot be lower than start slot")
231   }
```

The `replayBlocks` function has a pre-condition that `startSlot <= endSlot` to perform block validation from `startSlot` until `endSlot`. However, there is an edge case in which missing the input validation will cause `endSlot < startSlot`.

Scenario

1. The user provides `endSlot == -1` and `numReplaySlots == -(startSlot + 1)`
2. In this case, `endSlot` would be assigned to `startSlot + numReplaySlots`.
3. At this point `endSlot == -1` and `numReplaySlots != 0`
4. The check at line 229 would pass since `endSlot == -1`,
5. The `replayBlocks` pre-condition (`endSlot < startSlot`) does not hold

Recommendation

1. Ensure that `numReplaySlots >= 0`

```
if numReplaySlots < 0 {
    klog.Fatalf("numReplaySlots cannot be negative")
}
```

or
2. Remove the condition `endSlot != -1` from the `if` condition:

```
if endSlot < startSlot {
     klog.Fatalf("end slot cannot be lower than start slot")
}
```

Status

Not addressed

[A18] `runCatchup` might not behave as intended

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

Description

The `runCatchup` function is designed for validators that want to catch up to and stay synchronized with the live Solana mainnet. It downloads the latest Solana snapshot, builds the AccountsDB, and continuously replays new blocks to remain synchronized with the mainnet chain.
Fetching new blocks is done asynchronously by the function `StartAsyncBlockStream`, which calls `fetchAndParseBlock`. This function only handles temporary unavailability with a retry loop if `!blockConsumer.useRpc`.

runtimeverification/_audits_Overclock-Validator_mithril/pkg/blockstream/consumer.go
Line 80 to 102 in 06f7b95

```
80  if blockConsumer.useRpc {
81      blockResult, err = blockConsumer.rpcClient.GetBlockConfirmed(uint64(slot))
82      if err == rpcclient.SlotSkipped {
83          return nil
84      } else if err != nil {
85          panic(fmt.Sprintf("error fetching block: %s\n", err))
86      }
87  } else {
88      blockResult, err = blockConsumer.tryGetBlockResultFromFile(slot)
89      if err != nil {
90          for {
91              blockResult, err = blockConsumer.rpcClient.GetBlockConfirmed(uint64(slot))
92              if err == nil {
93                  break
94              } else if err == rpcclient.SlotSkipped {
95                  return nil
96              } else if strings.Contains(err.Error(), "Block not available for slot") { // we're too early. wait for a bit.
97                  time.Sleep(500 * time.Millisecond)
98              } else {
99                  panic(fmt.Sprintf("error fetching block: %s\n", err))
100             }
101         }
102     }
```

The `blockConsumer.useRpc` flag is initialized according to the `blockDir` argument provided by the user. The `blockDir` parameter is by default set to `""` in `runVerifier` mode and to `"tmp"` in `runCatchup` mode.

runtimeverification/_audits_Overclock-Validator_mithril/pkg/blockstream/consumer.go

Line 38 to 50 in 06f7b95

```
38  func NewBlockConsumer(rpcClient *rpcclient.RpcClient, streamChan chan *b.Block, startSlot, endSlot uint64, numInitialSlots uint64,
blockDir string) *BlockConsumer {
39    return &BlockConsumer{
40        rpcClient:         rpcClient,
41        streamChan:        streamChan,
42        totalSlotsToFetch: endSlot - startSlot,
43        currentSlot:       startSlot,
44        startSlot:         startSlot,
45        endSlot:           endSlot,
46        numInitialSlots:   min(numInitialSlots, endSlot-startSlot),
47        blockDir:          blockDir,
48        useRpc:            blockDir == "",
49    }
50  }
```

Scenario

1. Validator A runs Mithril in verifier mode, providing `blockDir != ""` and an `endSlot` that is not available yet
2. Validator B runs Mithril in the catchup mode but provides `blockdir == ""`
3. Since the `!blockConsumer.useRpc` is the condition for the retry loop when a block is not available yet, Validator A will now be running Mithril in catchup mode (until it reaches the `endSlot`). Validator B will run Mithril until the tip of the chain and exit with panic when it reaches the first unavailable block.

Recommendation

The condition to handle blocks that are not available yet should be based on the `isLive` flag.

[A19] `DownloadSnapshot` does not handle the case where all downloads attempts fail

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

Description

The function `DownloadSnapshot` downloads a complete blockchain snapshot from the best available RPC nodes with a retry mechanism. However, the function doesn't properly handle the case where all downloads fail.

runtimeverification/_audits_Overclock-Validator_mithril/pkg/snapshotdl/snapshotdl.go

Line 31 to 41 in 06f7b95

```
31      var finalPath string
32      for _, rpc := range bestRPCs {
33          finalPath, err = snapshot.DownloadSnapshot(rpc, cfg, "snapshot-", referenceSlot)
34          if err == nil {
35              break
36          } else {
37              fmt.Printf("failed to download snapshot from %s. trying next.\n", rpc)
38          }
39      }
40
41      snapshotSlot := extractFullSnapshotSlot(finalPath)
```

Scenario

1. The download attempts are failing from all the RPC nodes in the list
2. The `finalPath` variable remains empty or gets set to an invalid value
3. The `extractFullSnapshotSlot` function is called with an invalid argument, triggering the panic.

Recommendation

The code should check if `finalPath` is empty before calling `extractFullSnapshotSlot` and throw an error with an informative message:

```
if finalPath == "" || err != nil {
    return "", 0, 0, fmt.Errorf("failed to download snapshot from all available RPC nodes")
}

snapshotSlot := extractFullSnapshotSlot(finalPath)
```

Status

To be discussed with the client

[A20] Unbounded slice allocation when decoding `SysvarSlotHistory`

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

Description

The `UnmarshalWithDecoder` function reads `BlocksLen` from the decoder without any upper bound validation before using it in a loop.

```
    opt, err := decoder.ReadByte()

    if opt != 0 {
        sh.Bits.Bits.BlocksLen, err = decoder.ReadUint64(bin.LE)
        if err != nil {
            return fmt.Errorf("failed to read BlocksLen when decoding SysvarSlotHistory: %w", err)
        }

        for count := 0; count < int(sh.Bits.Bits.BlocksLen); count++ {
            block, err := decoder.ReadUint64(bin.LE)
            if err != nil {
                return fmt.Errorf("failed to read a block when decoding Bitvec in SysvarSlotHistory: %w", err)
            }
            sh.Bits.Bits.Blocks = append(sh.Bits.Bits.Blocks, block)
        }
    }
    // ...
}
```

A malicious or corrupted sysvar account could specify an extremely large `BlocksLen` value (e.g., `math.MaxUint64`), which might cause:

- Memory exhaustion: The loop would attempt to append billions of `uint64` values to the `Blocks` slice
- CPU exhaustion: The loop would run for an extremely long time
- DoS: The validator would become unresponsive

Mitigating factors:

- Sysvar accounts are owned by the system program and should only contain valid data in production
- The data comes from trusted sources (snapshot files or RPC)
- The vulnerability requires corrupted/malicious sysvar data, which shouldn't occur in normal operation

Recommendation

Add validation after reading the length value:

```
sh.Bits.Bits.BlocksLen, err = decoder.ReadUint64(bin.LE)
if err != nil {
    return fmt.Errorf("failed to read BlocksLen when decoding SysvarSlotHistory: %w", err)
}

if sh.Bits.Bits.BlocksLen > slotHistoryMaxEntries {
    return fmt.Errorf("BlocksLen %d exceeds maximum %d", sh.Bits.Bits.BlocksLen, maxBlocks)
}
```

[A21] Ignored error from `crypto/rand.Read` in `initRandom`

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

Description

The `initRandom()` method ignores the error returned by `rand.Read()`, which could lead to weakened randomness if the CSPRNG fails.

```
func (ltHash *LtHash) initRandom() *LtHash {
    randBytes := unsafe.Slice((*uint8)(unsafe.Pointer(&ltHash.value[0])), numElements*2)
    rand.Read(randBytes)
    return ltHash
}
```

The function calls `rand.Read(randBytes)` but doesn't check the returned error. According to Go's `crypto/rand` documentation, `rand.Read` returns `(n int, err error)` where `err` will be non-nil if the system's CSPRNG fails or cannot provide enough entropy.

The Risk

If `rand.Read` fails:

- It may fill fewer than the requested 2048 bytes (since `numElements = 1024` and each element is 2 bytes)
- The unfilled portion of `ltHash.value` would remain at its zero-initialized state
- This weakens the randomness of the hash, potentially making it predictable or causing collisions

Severity Assessment

Low:

- The function is currently only used in tests, not production code
- CSPRNG failures are rare in practice on properly configured systems
- However, if this function were used in production (e.g., for generating random account hashes), the impact would be significant

Recommendation

Check and handle the error returned by `rand.Read`.

```
func (ltHash *LtHash) initRandom() (*LtHash, error) {
    randBytes := unsafe.Slice((*uint8)(unsafe.Pointer(&ltHash.value[0])), numElements*2)
    n, err := rand.Read(randBytes)
    if err != nil {
        return nil, fmt.Errorf("failed to generate random bytes: %w", err)
    }
    if n != numElements*2 {
        return nil, fmt.Errorf("insufficient random bytes: got %d, expected %d", n, numElements*2)
    }
    return ltHash, nil
}
```

[A22] Ignored error during feature account unmarshalling

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

Description

The `UnmarshalFeatureAcct` function silently ignores decoding errors, which could lead to operating on partially initialized or corrupted `FeatureAcct` structs.

```
func UnmarshalFeatureAcct(data []byte) *FeatureAcct {
    decoder := bin.NewBinDecoder(data)
    featureAcct := new(FeatureAcct)

    featureAcct.UnmarshalWithDecoder(decoder)
    return featureAcct
}
```

Impact

This bug is used in the critical `scanAndEnableFeatures` function, which determines which protocol features are enabled.

```
featureAcct := features.UnmarshalFeatureAcct(acct.Data)
// Checks if featureAcct.ActivatedAt != nil to determine if feature is already activated
if featureAcct.ActivatedAt != nil && slot >= *featureAcct.ActivatedAt {
   // ...
}
// Checks if featureAcct.ActivatedAt == nil to determine if feature is pending activation
if featureAcct.ActivatedAt == nil && startOfEpoch {
// ...
}
```

If unmarshaling fails silently, the code continues to check `featureAcct.ActivatedAt`, which could be in an undefined state. With a corrupted or partially initialized struct, these checks could produce incorrect results, leading to:

- Features being incorrectly enabled or disabled
- Potential activation of features at the wrong slots

Recommendation

The function should return an error so callers can handle invalid data appropriately:

```
func UnmarshalFeatureAcct(data []byte) (*FeatureAcct, error) {
    decoder := bin.NewBinDecoder(data)
    featureAcct := new(FeatureAcct)

    err := featureAcct.UnmarshalWithDecoder(decoder)
    if err != nil {
        return nil, err
    }

    return featureAcct, nil
}
```

This would allow `scanAndEnableFeatures` to skip corrupted feature accounts rather than misinterpreting them as unactivated features.

[A23] Incorrect Overflow Detection in CheckedMulU128

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

Description

The `CheckedMulU128` function detects overflow by comparing the result to the first operand (`if result.Cmp(a) == -1`). This incorrectly flags overflow when the second operand is zero and the first operand is non-zero (e.g., a*0 yields 0, which is less than a), even though multiplication by zero cannot overflow. This will cause valid `wide.Uint128` multiplications to return `ErrOverflowMul` unexpectedly, potentially causing stake or token calculations involving zero amounts to error out and disrupt consensus or transaction processing.

```
func CheckedMulU128(a, b wide.Uint128) (wide.Uint128, error) {
    result := a.Mul(b)
    if result.Cmp(a) == -1 {
        return wide.NewUint128(0, 0), ErrOverflowMul
    }
    return result, nil
}
```

Recommendation

To modify the overflow check to handle the zero multiplier case properly.

```
func CheckedMulU128(a, b wide.Uint128) (wide.Uint128, error) {
    // Handle zero cases - no overflow possible
    zero := wide.Uint128FromUint64(0)
    if a.Cmp(zero) == 0 || b.Cmp(zero) == 0 {
        return zero, nil
    }

    result := a.Mul(b)

    // Check for overflow by dividing back
    if result.Div(a).Cmp(b) != 0 {
        return wide.NewUint128(0, 0), ErrOverflowMul
    }
    return result, nil
}
```

Status

Addressed in commit eac2f86.

[A24] Snapshot downloader library incorrectly parses `Content-Disposition` http header field in function `DownloadSnapshot`

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

Description

Mithril makes use of an external Go library `solana-snapshot-finder-go` for downloading Solana blockchain snapshots, which the Overclock team cloned into their own repository, see here.

In this library, the function `DownloadSnapshot` in `pkg/snapshot/download_utils.go` checks whether the given RPC server address provides a snapshot that is newer than a given slot number. For this purpose, this function queries the filename from either the given RPC URL or the optional HTTP header entry `Content-Disposition`.

In case the HTTP header `Content-Disposition` is available, the content is parsed by splitting the text at `'filename='`. This heuristic can fail if the server provides additional parameters in the header field after the `filename` parameter such as, the optional parameter `filename*`. In this case the extracted file name would be wrong and the code for extracting the slot from the filename would fail. As a consequence, this snapshot file would be skipped by the function.

Additionally, the `Content-Disposition` parser does not consider the alternative parameter `filename*` in case `filename` is not available, though this would not result in the snapshot being skipped as the `DownloadSnapshot` function would default to using the filename provided in the URL.

```
// Try to get filename from the Content-Disposition header or final URL
fileName := filepath.Base(resp.Request.URL.String())
if contentDisposition := resp.Header.Get("Content-Disposition"); contentDisposition != "" {
    parts := strings.Split(contentDisposition, "filename=")
    if len(parts) > 1 {
        fileName = strings.Trim(parts[1], `"' `)
        log.Printf("Filename from Content-Disposition: %s", fileName)
    }
}
```

There is also an instance of duplicated code in the function `writeSnapshotToFile` in the same file that implements the same functionality slightly differently:

```
// Extract the slot and hash from the response headers or URL
if contentDisposition := resp.Header.Get("Content-Disposition"); contentDisposition != "" {
    if strings.Contains(contentDisposition, "filename=") {
        fileName = strings.Split(contentDisposition, "filename=")[1]
        fileName = strings.Trim(fileName, `"'`)
        log.Printf("Filename from Content-Disposition: %s", fileName)
    }
}
```

Recommendation

Adjust the parser to correctly parse the `Content-Disposition` HTTP header field in all cases as specified in the HTTP documentation. We also recommend to additionally trying to parse the `filename*` parameter as an alternative to the `filename` parameter.

On top of that, the instance of duplicated code should be replaced by a dedicated function call to guarantee that both instances use the same correct implementation.

[A25] Hardcoded Epoch return in `newWarmupCooldownRateEpoch`

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

Description

The function `newWarmupCooldownRateEpoch` in `pkg/replay/rewards.go` currently returns a hardcoded epoch value (`565`) instead of computing the proper activation epoch based on feature flags and epoch schedule. The dynamic code is commented out. This can lead to incorrect reward calculations, effectively disabling or misconfiguring stake warmup/cooldown logic across all epochs.

```
func newWarmupCooldownRateEpoch(epochSchedule *sealevel.SysvarEpochSchedule, f *features.Features) *uint64 {
    /*slot, existed := f.ActivationSlot(features.ReduceStakeWarmupCooldown)
    if !existed {
        return nil
    }

    epoch := epochSchedule.GetEpoch(slot)*/
    epoch := uint64(565)
    return &epoch
}
```

Recommendation

Re-enable and test the original feature detection logic: use `features.Features.ActivationSlot` and `epochSchedule.GetEpoch(slot)` to compute the epoch for the warmup/cooldown rate dynamically. Remove the hardcoded constant once the logic is validated.

Status

Fixed in commit 3e4f73e.

[A26] Partial reads in `Unmarshal` functions

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

Description

Several `Unmarshal` functions, e.g., `AccountMeta.Unmarshal`, use `io.Read` to read a fixed number of bytes but ignore the count of bytes actually read. `io.Read` is an interface that leaves it up to the implementation how many bytes are actually read:

```
type Reader interface {
    Read(p []byte) (n int, err error)
}
```

The code base makes extensive use of, e.g., the `bytes.Reader` reader that always reads as many bytes as fit into the given byte slice `p` or as many bytes are available, whatever is lower, see implementation:

```
// for `bytes.Reader`
func (r *Reader) Read(b []byte) (n int, err error) {
    if r.i >= int64(len(r.s)) {
        return 0, io.EOF
    }
    r.prevRune = -1
    n = copy(b, r.s[r.i:])
    r.i += int64(n)
    return
}
```

This can cause `Read` to read less bytes than expected, leaving the rest of the bytes un-initialized, or even potentially undefined as `Read` functions may use all available space of a given byte slice as a scratchpad.

While `bytes.Reader` that is extensively used in the code base reads as many bytes as possible, some other implementations might not and require multiple runs of `Read` to read as many bytes as desired. Since the `Unmarshal` functions accept a `io.Reader` argument instead of `bytes.Reader`, it should not be relied upon that `Read` reads as many bytes as possible.

Recommendation

We recommend to replace all of these occurrences of `io.Read`, where a fixed size read is expected, with a call to `io.ReadFull` that always reads exactly as many bytes as expected.

Status

To be discussed with the client

[A27] RPC server uses AccountsDB data from blocks for which Mithril could not yet confirm finality

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

Description

When Mithril replays blocks, it propagates changes immediately to the same AccountsDB that is used by the RPC server. Therefore, the RPC server will provide data from blocks that have been successfully replayed by Mithril. Though as explained in Invariants, Mithril should only use data from blocks for which it could confirm finality.

This is impossible for Mithril to confirm by replaying blocks individually and propagating changes to AccountsDB immediately. Therefore, Mithril currently relies on potentially untrusted RPC servers to provide solely blocks that have reached finality. In practice, these RPC nodes could instead provide blocks from forks that do not reach finality, yet execute successfully on Mithril. This would eventually also cause a bankhash mismatch since blocks would eventually stop being proposed for such a fork that won't reach consensus.

Here, a block is considered to have reached finality when it has at least 31 descendant blocks that have reached a supermajority of votes, i.e., two-thirds of votes weighted by stake. In turn, a block is considered being voted on when a descendant block contains a vote for this block. This implies that Mithril might need to replay multiple descendant blocks before confirming that a given block has achieved a supermajority of votes weighted by stake.

Recommendation

We recommend to not rely on the assumption that untrusted RPC servers only provide blocks that have achieved finality. Instead, we recommend to optimistically execute blocks for which finality is assumed and to only commit changes to the AccountsDB that is used by Mithril's RPC server backend once finality of the respective block has been confirmed. Likewise, Mithril must be able to roll back replay of blocks if it turns out that the blocks that were optimistically executed cause a bankhash mismatch or never reach finality.

Status

The client has acknowledged the issue but has not yet implemented a fix as of this writing.

[A28] Lattice hash of a snapshot is never validated by re-computing the hash

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

Description

When Mithril is started from empty state, instead of replaying blocks from genesis up to the most recent blocks, Mithril instead downloads a recent snapshot of Solana that contains all Solana accounts at a given slot. Mithril then initializes a new AccountsDB and populates it with data from that snapshot.

As these snapshot are from potentially untrusted RPC servers, Mithril must ensure validity of the given snapshot, i.e., that all accounts have not been altered and that metadata is correct, such as, e.g., the parent block bankhash.

In the past, this check was performed implicitly by Mithril when replaying blocks eventually, since epochs, i.e., blocks, used to contain Epoch Account Hashes (EAH) that hashed all accounts roughly once per epoch. This mechanism was implemented in Mithril, but has since been replaced in Solana and Mithril by the lattice hash mechanism. The lattice hash is a hash that can be computed from all accounts and is used in all bankhash calculations since replacing EAH. At the same time, nodes do not have to hash all accounts for every bankhash, since the lattice hash of a previous slot can be transmuted with the account deltas between both slots. This significantly reduces compute requirements for Solana.

However, since Mithril continuously uses the lattice hash of the previous slot, no implicit check of the snapshot is ever performed for all accounts. As Mithril parses and uses the lattice hash provided in the snapshot, Mithril will instead run into a bankhash mismatch when processing a transaction that uses account data altered by an altered snapshot. This can happen at any given point in the future and could be exploited to cause a Denial-of-Service by introducing a transaction on the Solana blockchain that would trigger the bankhash mismatch by using the altered account data.

Recommendation

We recommend to validate the lattice hash provided in a snapshot by re-computing the lattice hash from all accounts in the snapshot. In case of a mismatch, the respective snapshot should be discarded.

Status

The client has acknowledged the issue but has not yet implemented a fix as of this writing.

Fuzzing Discussion and Findings

This section communicates specific results and findings from the fuzzing efforts performed over the duration of the Mithril audit. The fuzzing was performed using two complementary approaches: native Go fuzzing for isolated component testing, and differential fuzzing against Agave (solfuzz-agave) for consensus-critical behavioral verification. The end of this section contains references to all identified issues.

Approach Overview

Our fuzzing strategy targeted Mithril, a Go-based Solana validator implementation, with two distinct but complementary methodologies:

1. **Native Go Fuzzing**: Coverage-guided fuzzing using Go's built-in `testing/F` framework to find crashes, panics, and edge-case bugs in individual components.
2. **Differential Fuzzing**: Cross-implementation testing using the solfuzz framework to verify behavioral consistency between Mithril and Agave (the reference Rust implementation).

The differential fuzzing approach is particularly critical for blockchain validators, as even minor behavioral divergences can lead to consensus failures, chain splits, or economic exploits.

Time and Compute Resources

Due to time constraints during the audit period, our fuzzing efforts were primarily exploratory rather than representing a comprehensive long-running campaign. The fuzzing was performed on a local development machine, focusing on identifying low-hanging fruit and validating the correctness of key execution paths.

Native Go Fuzzing: Approximately 1-2 days of cumulative fuzzing time distributed across 41 fuzz targets covering various subsystems of Mithril.

Differential Fuzzing: Exploratory runs focusing on crash file analysis and targeted testing of specific components. A dedicated long-running fuzzing campaign was not conducted due to time limitations.

Target	Coverage	Approximate Time	Issues Found
Native Go Fuzzing (41 targets)	~70% of targeted functions	1-2 days	20+ issues
Differential: Transaction Execution	`sol_compat_txn_execute_v1`	Exploratory	10+ issues
Differential: ELF Loader	`sol_compat_elf_loader_v2`	Exploratory	1 issue
Differential: VM Interpreter	`sol_compat_vm_interp_v1`	Exploratory	5+ issues
Differential: Instruction Execution	`sol_compat_instr_execute_v1`	Exploratory	0 issues *

* No issues found via `sol_compat_instr_execute_v1`; the harness implementation may require further refinement for effective testing.

Native Go Fuzzing

Overview

We developed 41 Go fuzz test files targeting core subsystems of Mithril. These tests leverage Go's native fuzzing infrastructure to generate random inputs and detect crashes, panics, and assertion failures.

The tests can be found in Runtime Verifications GitHub fork of Mithril, specifically in the branch `go-fuzzing-tests`.

Targets Covered

The following packages and components were fuzzed:

Package	Fuzz Test File	Components Tested
pkg/accountsdb	accountsdb_fuzz_test.go	Account storage and retrieval
pkg/accountsdb	appendvec_fuzz_test.go	Append vector parsing and serialization
pkg/accountsdb	index_fuzz_test.go	Account index operations
pkg/bankhash	bankhash_fuzz_test.go	Bank hash computation
pkg/base58	base58_fuzz_test.go	Base58 encoding/decoding
pkg/block	block_fuzz_test.go	Block parsing and validation
pkg/blockstore	bincode_fuzz_test.go	Bincode serialization
pkg/blockstore	meta_fuzz_test.go	Block metadata handling
pkg/compactindex	compactindex_fuzz_test.go	Compact index operations
pkg/cu	cu_fuzz_test.go	Compute unit metering
pkg/features	features_fuzz_test.go	Feature gate handling
pkg/fees	fees_fuzz_test.go	Fee calculation
pkg/genesis	genesis_fuzz_test.go	Genesis parsing
pkg/gossip	gossip_fuzz_test.go	Gossip message handling
pkg/poh	poh_fuzz_test.go	Proof of History
pkg/rent	rent_fuzz_test.go	Rent calculations
pkg/replay	transaction_fuzz_test.go	Transaction replay
pkg/rewards	rewards_fuzz_test.go	Reward calculations
pkg/safemath	safemath_fuzz_test.go	Safe arithmetic operations
pkg/sbpf/loader	loader_fuzz_test.go	SBPF ELF loader
pkg/sbpf	sbpf_fuzz_test.go	SBPF VM operations
pkg/sealevel	borrowed_account_fuzz_test.go	Account borrowing
pkg/sealevel	bpf_loader_fuzz_test.go	BPF loader program
pkg/sealevel	compute_budget_program_fuzz_test.go	Compute budget processing
pkg/sealevel	ed25519_program_fuzz_test.go	Ed25519 precompile
pkg/sealevel	execution_ctx_fuzz_test.go	Execution context
pkg/sealevel	instruction_fuzz_test.go	Instruction parsing
pkg/sealevel	loader_v4_fuzz_test.go	Loader v4 program
pkg/sealevel	secp256k1_program_fuzz_test.go	Secp256k1 precompile
pkg/sealevel	secp256r1_program_fuzz_test.go	Secp256r1 precompile
pkg/sealevel	syscalls_hash_fuzz_test.go	Hash syscalls
pkg/sealevel	syscalls_log_fuzz_test.go	Log syscalls
pkg/sealevel	syscalls_mem_fuzz_test.go	Memory syscalls
pkg/sealevel	syscalls_pda_fuzz_test.go	PDA syscalls
pkg/sealevel	system_program_fuzz_test.go	System program
pkg/sealevel	vote_program_fuzz_test.go	Vote program
pkg/sealevel	vote_state_fuzz_test.go	Vote state handling
pkg/shred	shred_fuzz_test.go	Shred parsing
pkg/snapshot	manifest_fuzz_test.go	Snapshot manifest
pkg/solana	types_fuzz_test.go	Solana type parsing
pkg/tpu	tpu_fuzz_test.go	TPU transaction parsing

Issue Categories

The native Go fuzzing uncovered several categories of bugs:

Category	Count	Examples
Panic on malformed input	8+	Index out of bounds, nil pointer dereference
DoS via unbounded allocation	6+	Missing bounds checks on array lengths
Arithmetic issues	4+	Integer overflow, division by zero
Memory safety	3+	Write permission bypass, memory overlap
Logic errors	4+	Incorrect validation, wrong error handling

Differential Fuzzing

Overview

We implemented a solfuzz-compatible harness for Mithril (`libsolfuzz_mithril.so`) that enables differential fuzzing against Agave's solfuzz harness (`libsolfuzz_agave.so`). This approach tests behavioral consistency between the two implementations by:

1. Feeding the same input to both implementations
2. Comparing their outputs for equivalence
3. Flagging any divergences as potential bugs

Harness Implementation

The `pkg/solfuzz` directory contains our implementation of the solfuzz harness for Mithril. The following C-compatible APIs were implemented:

API Function	File	Status	Description
`sol_compat_init`	`main.go`	✅ Implemented	Initialize the solfuzz target
`sol_compat_fini`	`main.go`	✅ Implemented	Cleanup and finalize
`sol_compat_get_features_v1`	`main.go`	✅ Implemented	Return supported feature set
`sol_compat_get_metadata_v1`	`main.go`	✅ Implemented	Return target metadata
`sol_compat_txn_execute_v1`	`txn_execute.go`	✅ Implemented & Tested	Transaction execution
`sol_compat_elf_loader_v2`	`elf_loader_v2.go`	✅ Implemented & Tested	ELF loader (flatbuffer)
`sol_compat_vm_interp_v1`	`vm_interp.go`	✅ Implemented & Tested	VM interpreter
`sol_compat_instr_execute_v1`	`instr_execute.go`	✅ Implemented & Tested	Instruction execution
`sol_compat_pack_compute_budget_v1`	`pack_compute_budget.go`	🟡 Partial (Not Tested)	Compute budget packing
`sol_compat_shred_parse_v1`	`shred_parse.go`	🟡 Partial (Not Tested)	Shred parsing
`sol_compat_block_execute_v1`	`block_execute.go`	🚧 Stub (Not Tested)	Block execution
`sol_compat_vm_syscall_execute_v1`	`vm_syscall_execute.go`	🚧 Stub (Not Tested)	VM syscall execution

Legend:

- ✅ **Implemented & Tested**: Working implementation used for differential fuzzing. Note: These implementations are functional but still evolving; ongoing refinements are needed to achieve full behavioral parity with Agave.
- 🟡 **Partial (Not Tested)**: Basic implementation exists but not validated through differential fuzzing
- 🚧 **Stub (Not Tested)**: Placeholder implementation with minimal or no real logic

Implementation Details

Our harness implementation:

1. **Reuses Mithril's existing infrastructure**: Rather than re-implementing validation logic, we leveraged Mithril's existing packages (`pkg/replay` , `pkg/sealevel` , `pkg/sbpf` , etc.) to ensure the fuzzing accurately reflects production behavior.
2. **Protobuf/Flatbuffer compatibility**: Input and output data is marshaled using the same protobuf and flatbuffer schemas as solfuzz-agave, enabling direct comparison of results.
3. **Feature set alignment**: The harness exposes Mithril's feature gates in a format compatible with solfuzz's feature negotiation protocol.
4. **Error code mapping**: We implemented error code translation to match Agave's error taxonomy for accurate divergence detection.

Components Tested

Component	Description	Testing Coverage
Transaction Execution	End-to-end transaction processing including sanitization, fee calculation, instruction execution, and state updates	✅ Actively tested via differential fuzzing
ELF Loader	SBPF ELF binary parsing and validation	✅ Actively tested via differential fuzzing
VM Interpreter	SBPF bytecode execution	✅ Actively tested via differential fuzzing
Instruction Execution	Individual instruction processing	✅ Actively tested via differential fuzzing
Compute Budget	Compute unit limit and price processing	🟡 Partial implementation, not tested
Shred Parsing	Shred data parsing	🟡 Partial implementation, not tested
Block Execution	Full block processing	🚧 Stub only (placeholder values, no execution)
VM Syscalls	Individual syscall execution	🚧 Stub only (recognizes but doesn't execute)

Components Not Tested

Due to time constraints, the following solfuzz APIs were not fully implemented or tested:

- `sol_compat_block_execute_v1` : Stub implementation with placeholder values. Full implementation would require Bank, AccountsDB, entry processing pipeline, leader schedule computation, and bank hash calculation.
- `sol_compat_vm_syscall_execute_v1` : Stub implementation that recognizes syscalls but doesn't execute them. Full implementation requires extensive InvokeContext, TransactionContext, SysvarCache, and per-syscall implementations.
- `sol_compat_pack_compute_budget_v1` : Partial implementation that parses compute budget instructions but has not been validated against Agave.
- `sol_compat_shred_parse_v1` : Partial implementation with basic shred header validation but has not been validated against Agave.
- `sol_compat_type_execute_v1` : Not implemented (type-level fuzzing).

Identified Issues

Native Go Fuzzing Findings

The following issues were identified through native Go fuzzing.

High Severity (Findings)

ID	Finding	Location
F-A01	`Write64` uses wrong permission flag	`pkg/accountsdb/appendvec.go`
F-A02	`isOverlap` panics on integer overflow	`pkg/sbpf/loader/parse.go`
F-A03	Missing bounds check on lockout count causing DoS via unbounded memory allocation	`pkg/sealevel/vote_program.go`
F-A04	Missing bounds check on lockout count in vote state deserialization	`pkg/sealevel/vote_state.go`
F-A05	Missing bounds check on epoch credits count causing DoS	`pkg/sealevel/vote_state.go`
F-A06	`ParseTx` does not validate transaction structure consistency	`pkg/tpu/tpu.go`

Medium Severity (Findings)

ID	Finding	Location
F-A07	`Decode32` panics with index out of range	`pkg/base58/base58.go`
F-A08	`SaturatingMulU8/U16` can miss some overflows	`pkg/safemath/saturating.go`
F-A09	`AppendVecAccount.Unmarshal` is missing `DataLen` validation	`pkg/accountsdb/appendvec.go`
F-A10	`SlotAcctVecs.UnmarshalWithDecoder` is missing `numAcctVecs` validation	`pkg/snapshot/manifest_decoder.go`
F-A11	`Account database missing buffer length validation in append vector parser	`pkg/accountsdb/appendvec.go`

Low Severity (Findings)

ID	Finding	Location
F-A12	`CheckedMulU8/U16` can return false positive	`pkg/safemath/checked.go`
F-A13	`isNonOverlapping` incorrectly handles zero-length regions	`pkg/sealevel/syscalls_common.go`

Informative Findings

ID	Finding	Location
F-B01	Panic on unimplemented shred formats	`pkg/shred/shred.go`
F-B02	`translateInternal` panic on zero-size stack access	`pkg/sbpf/memory.go`
F-B03	`PurgeAuthorizedVoters` panics on empty set	`pkg/sealevel/vote_state.go`
F-B04	Missing division-by-zero check in `calculateDeltaLtHash`	`pkg/bankhash/`
F-B05	`partitionIdxFromSlotIdx` can overflow	`pkg/bankhash/`
F-B06	`SaturatingPow` does not check division by zero	`pkg/safemath/saturating.go`
F-B07	Incorrect opcode values for signed division instructions	`pkg/sbpf/` (test file)
F-B08	Feature gate addresses displayed as raw bytes	`pkg/features/features.go`
F-B09	`translateInstruction*` functions do not make defensive copy of instruction data	`pkg/sealevel/syscalls_cpi.go`

Differential Fuzzing Findings

The following issues were identified through differential fuzzing against Agave.

Confirmed Issues

ID	Finding	Component	Status
F-D01	Missing Divide-by-Zero Check in `OpUrem32Reg`	VM Interpreter	Needs Review
F-D02	Missing blockhash validation in transaction sanitization	Transaction	Fixed
F-D03	Incorrect exception handling in VM interpreter	VM Interpreter	Needs Review
F-D04	Missing fee payer validation before transaction execution	Transaction	Needs Review
F-D05	LoaderV4 setLoaderV4AcctState missing features parameter	LoaderV4	Needs Review
F-D06	Incorrect section order validation in ELF loader	ELF Loader	Needs Review
F-D07	Incorrect executable flag checks in BorrowedAccount	Sealevel	Needs Review
F-D08	String parsing in ELF loader doesn't match Agave behavior	ELF Loader	Needs Review
F-D09	Incorrect ELF deploy checks during execution	ELF Loader	Needs Review
F-D10	Missing verifyCheckSh32 handler in SBPF verifier	SBPF Verifier	Needs Review
F-D11	Side Effects Occurring After Error in VM Interpreter	VM Interpreter	Needs Review
F-D12	Incorrect Right Shift in Signed High Multiply Instructions	VM Interpreter	Needs Review
F-D13	Missing ExcUnsupportedInstruction Error for Invalid Opcodes	VM Interpreter	Needs Review
F-D14	Missing Padding Zeroing in serializeParametersAligned	VM Interpreter	Needs Review
F-D15	Inconsistent Error Code for Empty ELF File	ELF Loader	Needs Review
F-D16	SBPF version validation order in ELF loader	ELF Loader	Needs Review
F-D17	Program header overlap check for empty headers	ELF Loader	Needs Review
F-D18	Incorrect error code for program loading failures	ELF Loader	Needs Review
F-D19	Incorrect PC Value on OpExit Stack Pop	VM Interpreter	Needs Review

Recommendations for Future Fuzzing

Based on our experience during this audit, we recommend the following for continued fuzzing efforts:

1. **Long-running differential fuzzing campaign**: The exploratory differential fuzzing identified multiple divergences. A sustained fuzzing campaign with dedicated compute resources would likely uncover additional issues.
2. **Corpus development**: Build a comprehensive corpus of valid transactions, ELF binaries, and other inputs to seed the fuzzer with realistic test cases.
3. **Complete syscall implementation**: The `sol_compat_block_execute_v1` and `sol_compat_vm_syscall_execute_v1` stub should be fully implemented to enable differential fuzzing of individual syscalls.
4. **Continuous integration**: Integrate fuzzing into CI/CD pipelines to catch regressions early.
5. **Address typed error architecture**: Mithril's use of string-based errors vs. Agave's typed error enums creates fragile error code mapping. A typed error system would improve maintainability and reduce false divergences.

Conclusion

Our fuzzing efforts, while limited in duration, successfully identified numerous issues across Mithril's codebase. The native Go fuzzing uncovered 25 bugs ranging from high-severity DoS vulnerabilities to low-severity edge cases. The differential fuzzing against Agave revealed 12 consensus-critical behavioral divergences that could lead to chain splits in a production environment.

The solfuzz harness implementation for Mithril provides a foundation for continued differential fuzzing. We recommend the Mithril team dedicate resources to a comprehensive, long-running fuzzing campaign to further validate the implementation's correctness and Agave compatibility.

[F-A01] `Write64` uses wrong permission flag in translateInternal call

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

Description

- `pkg/sbpf/interpreter.go` function `Write64` uses the wrong permission flag in translateInternal call

```
// BUGGY CODE:
func (ip *Interpreter) Write64(addr uint64, x uint64) error {
    ptr, err := ip.translateInternal(addr, 8, false)  // ❌ Should be true!
    if err != nil {
        return err
    }
    *(*uint64)(ptr) = x
    return nil
}
```

Issue: The `Write64` function passes `false` (indicating a read operation) as the `write` parameter to `translateInternal`, when it should pass `true` since it's performing a write operation. This causes the function to bypass write permission checks.

Impact:

- **Critical security bug**: Allows writing to read-only memory regions (program memory, input memory)
- Write permission validation is completely bypassed for 64-bit writes
- Inconsistent with `Write8`, `Write16`, `Write32` which correctly pass `true`
- Could corrupt read-only data or instruction memory
- Allows modification of input data that should be immutable

Root Cause: Copy-paste error or typo - the `write` parameter was set to `false` instead of `true`.

Recommendation

Suggested Fix:

```
func (ip *Interpreter) Write64(addr uint64, x uint64) error {
    ptr, err := ip.translateInternal(addr, 8, true)  // ✅ Correct: true for write operations
    if err != nil {
        return err
    }
    *(*uint64)(ptr) = x
    return nil
}
```

Verification: Compare with other Write functions:

- `Write8(addr, x)` → `translateInternal(addr, 1, true)` ✅
- `Write16(addr, x)` → `translateInternal(addr, 2, true)` ✅
- `Write32(addr, x)` → `translateInternal(addr, 4, true)` ✅
- `Write64(addr, x)` → `translateInternal(addr, 8, false)` ❌ **BUG!**

Testing: This bug could be detected with a simple test:

```
// Try to write to program memory (read-only)
err := interpreter.Write64(0x100000000, 0x1234567890ABCDEF)
// Should fail with "write to program" error
// But currently succeeds due to false flag
```

Status

Fixed in commit acf1ad6.

[F-A02] `pkg/sbpf/loader/parse.go` function `isOverlap` panics on integer overflow instead of returning error

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

Description

- `pkg/sbpf/loader/parse.go` function `isOverlap` panics on integer overflow instead of returning error

```
// BUGGY CODE:
func isOverlap(startA uint64, sizeA uint64, startB uint64, sizeB uint64) bool {
    if startA > startB {
        startA, sizeA, startB, sizeB = startB, sizeB, startA, sizeA
    }
    endA, endB := startA+sizeA, startB+sizeB
    if endA < startA || endB < startB {
        panic("isOverlap: integer overflow")  // ❌ PANIC instead of error
    }
    return sizeA != 0 && sizeB != 0 && (startA == startB || endA > endB)
}
```

Issue: The function panics when detecting integer overflow instead of returning an error. This occurs when malformed ELF files contain extremely large offset/size values that cause `start + size` to overflow `uint64` .

Impact:

- **DoS vulnerability**: Malicious ELF files can crash the loader/validator
- **Violation of error handling best practices**: Validation functions should never panic
- **Triggered during ELF header validation** at line 279 when checking if program headers and section headers overlap
- Caught by fuzzing with malformed ELF header values

Trigger Example: A malformed ELF with:

- `Phoff = 0xFFFFFFFFFFFF0000` (program header offset)
- `Phnum = 0x1000` (number of program headers)
- Multiplication: `0x1000 * phEntLen` gives large size
- Addition: `Phoff + size` overflows → panic

Root Cause: The function explicitly panics on overflow detection instead of propagating an error to the caller. While the overflow detection logic is correct (`endA < startA` catches wraparound), the response mechanism (panic vs error) is wrong.

Recommendation

Suggested Fix:

```
func isOverlap(startA uint64, sizeA uint64, startB uint64, sizeB uint64) (bool, error) {
    if startA > startB {
        startA, sizeA, startB, sizeB = startB, sizeB, startA, sizeA
    }
    endA, endB := startA+sizeA, startB+sizeB
    if endA < startA || endB < startB {
        return false, fmt.Errorf("isOverlap: integer overflow")  // ✅ Return error
    }
```

```
    return sizeA != 0 && sizeB != 0 && (startA == startB || endA > endB), nil
}
```

Call Site Updates: Update all callers to handle the error:

```
// In validateElfHeader (line 279):
overlap, err := isOverlap(eh.Phoff, uint64(eh.Phnum)*phEntLen, eh.Shoff, uint64(eh.Shnum)*shEntLen)
if err != nil {
    return err
}
if overlap {
    return fmt.Errorf("program and section header overlap")
}
```

Alternative Fix: Check for overflow BEFORE the addition to avoid any wraparound:

```
func isOverlap(startA uint64, sizeA uint64, startB uint64, sizeB uint64) (bool, error) {
    if startA > startB {
        startA, sizeA, startB, sizeB = startB, sizeB, startA, sizeA
    }

    // Check for overflow before addition
    if sizeA > 0 && startA > math.MaxUint64 - sizeA {
        return false, fmt.Errorf("isOverlap: integer overflow in range A")
    }
    if sizeB > 0 && startB > math.MaxUint64 - sizeB {
        return false, fmt.Errorf("isOverlap: integer overflow in range B")
    }

    endA, endB := startA+sizeA, startB+sizeB
    return sizeA != 0 && sizeB != 0 && (startA == startB || endA > endB), nil
}
```

Status

Addressed in commit fae3b04

[F-A03] `pkg/sealevel/vote_program.go` functions `VoteInstrUpdateVoteState.UnmarshalWithDecoder` and `VoteInstrTowerSync.UnmarshalWithDecoder` missing bounds check on lockout count causing DoS via unbounded memory allocation

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

Description

- `pkg/sealevel/vote_program.go` functions `VoteInstrUpdateVoteState.UnmarshalWithDecoder` and `VoteInstrTowerSync.UnmarshalWithDecoder` missing bounds check on lockout count causing DoS via unbounded memory allocation

```
// BUGGY CODE in VoteInstrUpdateVoteState.UnmarshalWithDecoder:
func (updateVoteState *VoteInstrUpdateVoteState) UnmarshalWithDecoder(decoder *bin.Decoder) error {
    numLockouts, err := decoder.ReadUint64(bin.LE)
    if err != nil {
        return err
    }

    _, err = safemath.CheckedMulU64(numLockouts, 12)
    if err != nil {
        return err
    }

    updateVoteState.Lockouts.Clear()
    updateVoteState.Lockouts.SetBaseCap(int(numLockouts))  // ❌ No bounds check!
    for count := uint64(0); count < numLockouts; count++ {
        var lockout VoteLockout
        err = lockout.UnmarshalWithDecoder(decoder)
        if err != nil {
            return err
        }
        updateVoteState.Lockouts.PushBack(lockout)  // ❌ Triggers panic in deque.growIfFull
    }
    // ... rest of function
}

// BUGGY CODE in VoteInstrTowerSync.UnmarshalWithDecoder:
func (towerSync *VoteInstrTowerSync) UnmarshalWithDecoder(decoder *bin.Decoder) error {
    // ... earlier code ...

    var lockoutOffsetsLen int
    lockoutOffsetsLen, err = decoder.ReadCompactU16()
    if err != nil {
        return err
    }

    // ...

    towerSync.Lockouts.Clear()
```

```
    towerSync.Lockouts.SetBaseCap(int(lockoutOffsetsLen))  // ❌ No bounds check!
    for i := uint64(0); i < uint64(lockoutOffsetsLen); i++ {
        // ... similar pattern
    }
}
```

Issue: The functions read untrusted `numLockouts` or `lockoutOffsetsLen` values from input data and use them directly to allocate memory without validating against the protocol's `MaxLockoutHistory` constant (31). This allows attackers to trigger memory allocation panics or Out-of-Memory crashes.

Impact:

- **Critical DoS vulnerability**: Malicious vote instructions can crash validators
- **Resource exhaustion**: Even moderately large values (e.g., millions) can cause OOM before the panic
- **Multiple attack vectors**: Both UpdateVoteState and TowerSync instructions are vulnerable
- **Bypass of overflow check**: The `CheckedMulU64(numLockouts, 12)` check only prevents arithmetic overflow, not unreasonable allocation sizes
- Runtime panic: `makeslice: len out of range` when trying to allocate huge slices
- Caught by fuzzing with input containing large lockout counts

Trigger Example:

- Craft vote instruction with `numLockouts = 0x7FFFFFFFFFFFFFFF` (max uint64)
- Overflow check passes: `0x7FFFFFFFFFFFFFFF * 12` might not overflow depending on value
- `SetBaseCap(int(0x7FFFFFFFFFFFFFFF))` attempts allocation
- Deque library calls `make([]VoteLockout, cap)`
- **Panic**: Go runtime refuses allocation

Root Cause: Missing validation against protocol constant `MaxLockoutHistory = 31` before memory allocation. The code validates arithmetic overflow but not semantic correctness.

Suggested Fix:

```
// For VoteInstrUpdateVoteState.UnmarshalWithDecoder:
func (updateVoteState *VoteInstrUpdateVoteState) UnmarshalWithDecoder(decoder *bin.Decoder) error {
    numLockouts, err := decoder.ReadUint64(bin.LE)
    if err != nil {
        return err
    }

    // ✅ Validate against protocol maximum
    if numLockouts > MaxLockoutHistory {
        return fmt.Errorf("too many lockouts: %d (max %d)", numLockouts, MaxLockoutHistory)
    }

    _, err = safemath.CheckedMulU64(numLockouts, 12)
    if err != nil {
        return err
    }

    updateVoteState.Lockouts.Clear()
    updateVoteState.Lockouts.SetBaseCap(int(numLockouts))
    for count := uint64(0); count < numLockouts; count++ {
        var lockout VoteLockout
        err = lockout.UnmarshalWithDecoder(decoder)
```

```
        if err != nil {
            return err
        }
        updateVoteState.Lockouts.PushBack(lockout)
    }
    // ... rest of function
}

// For VoteInstrTowerSync.UnmarshalWithDecoder:
func (towerSync *VoteInstrTowerSync) UnmarshalWithDecoder(decoder *bin.Decoder) error {
    // ... earlier code ...

    var lockoutOffsetsLen int
    lockoutOffsetsLen, err = decoder.ReadCompactU16()
    if err != nil {
        return err
    }

    // ✅ Validate against protocol maximum
    if lockoutOffsetsLen > MaxLockoutHistory {
        return fmt.Errorf("too many lockout offsets: %d (max %d)", lockoutOffsetsLen, MaxLockoutHistory)
    }

    towerSync.Lockouts.Clear()
    towerSync.Lockouts.SetBaseCap(int(lockoutOffsetsLen))
    for i := uint64(0); i < uint64(lockoutOffsetsLen); i++ {
        // ... rest of loop
    }
}
```

Alternative Fix: Add a general safety check for unreasonably large allocations:

```
// Conservative limit: even if protocol allows more, prevent DoS
const MaxReasonableAllocation = 10000

if numLockouts > MaxReasonableAllocation {
    return fmt.Errorf("allocation too large: %d (max %d)", numLockouts, MaxReasonableAllocation)
}
```

[F-A04] `pkg/sealevel/vote_state.go` functions `VoteState0_23_5.UnmarshalWithDecoder`, `VoteState1_14_11.UnmarshalWithDecoder`, and `VoteState.UnmarshalWithDecoder` missing bounds check on lockout count causing DoS via unbounded memory allocation

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

Description

- `pkg/sealevel/vote_state.go` functions `VoteState0_23_5.UnmarshalWithDecoder`, `VoteState1_14_11.UnmarshalWithDecoder`, and `VoteState.UnmarshalWithDecoder` missing bounds check on lockout count causing DoS via unbounded memory allocation

```
// BUGGY CODE in VoteState0_23_5.UnmarshalWithDecoder:
numLockouts, err := decoder.ReadUint64(bin.LE)
if err != nil {
    return err
}

voteState.Votes.Clear()
voteState.Votes.SetBaseCap(int(numLockouts))  // ❌ No bounds check!
for count := uint64(0); count < numLockouts; count++ {
    var lockout VoteLockout
    err = lockout.UnmarshalWithDecoder(decoder)
    if err != nil {
        return err
    }
    voteState.Votes.PushBack(lockout)  // ❌ Triggers panic in deque.growIfFull
}

// BUGGY CODE in VoteState1_14_11.UnmarshalWithDecoder:
// Same pattern - reads numLockouts, allocates without validation

// BUGGY CODE in VoteState.UnmarshalWithDecoder:
// Same pattern - reads numLockouts, allocates LandedVote without validation
```

Issue: All three VoteState deserialization functions read untrusted `numLockouts` values from input data and use them directly to allocate memory without validating against the protocol's `MaxLockoutHistory` constant (31). This allows attackers to trigger memory allocation panics or Out-of-Memory crashes when deserializing vote account data.

Impact:

- **Critical DoS vulnerability**: Malicious vote account data can crash validators during account deserialization
- **Resource exhaustion**: Even moderately large values (e.g., millions) can cause OOM before the panic
- **Multiple attack vectors**: All three vote state version formats are vulnerable
- **Affects account storage**: Unlike the instruction-based bug (VoteInstrUpdateVoteState/TowerSync), this affects stored vote account data
- Runtime panic: `panic: runtime error: makeslice: len out of range` when trying to allocate huge slices
- Triggered by fuzzing: `FuzzVoteState0_23_5` with seed corpus entry `017f4c3db25de093`

Trigger Example:

- Craft malicious vote account data with `numLockouts = 0x7FFFFFFFFFFFFFFF` (max uint64)
- Store in account or provide via RPC/snapshot
- Validator attempts to deserialize account state
- `SetBaseCap(int(0x7FFFFFFFFFFFFFFF))` attempts allocation
- Deque library calls `make([]VoteLockout, cap)`
- **Panic**: Go runtime refuses allocation → validator crash

Root Cause: Missing validation against protocol constant `MaxLockoutHistory = 31` before memory allocation. While the instruction-handling code (VoteInstrUpdateVoteState, VoteInstrTowerSync) correctly validates bounds, the vote state storage format deserialization does not.

Comparison to Instruction Bug: This bug is similar to but distinct from the VoteInstrUpdateVoteState/TowerSync bug documented earlier:

- **Instruction bug**: Affects processing of incoming vote instructions
- **Storage bug (this)**: Affects deserialization of stored vote account data
- **Both**: Allow unbounded allocation from untrusted input

Suggested Fix:

```
// For VoteState0_23_5.UnmarshalWithDecoder:
func (voteState *VoteState0_23_5) UnmarshalWithDecoder(decoder *bin.Decoder) error {
    // ... earlier code ...

    numLockouts, err := decoder.ReadUint64(bin.LE)
    if err != nil {
        return err
    }

    // ✅ Validate against protocol maximum
    if numLockouts > MaxLockoutHistory {
        return fmt.Errorf("too many lockouts: %d (max %d)", numLockouts, MaxLockoutHistory)
    }

    voteState.Votes.Clear()
    voteState.Votes.SetBaseCap(int(numLockouts))
    for count := uint64(0); count < numLockouts; count++ {
        var lockout VoteLockout
        err = lockout.UnmarshalWithDecoder(decoder)
        if err != nil {
            return err
        }
        voteState.Votes.PushBack(lockout)
    }
    // ... rest of function
}

// Apply same fix to VoteState1_14_11.UnmarshalWithDecoder
// Apply same fix to VoteState.UnmarshalWithDecoder
```

Alternative Fix: Add a general safety check for unreasonably large allocations:

```
// Conservative limit: even if protocol allows more, prevent DoS
const MaxReasonableAllocation = 10000
```

```
if numLockouts > MaxReasonableAllocation {
    return fmt.Errorf("allocation too large: %d (max %d)", numLockouts, MaxReasonableAllocation)
}
```

Testing / Mitigation:

- Add regression test that attempts to deserialize vote state with large lockout counts and verifies graceful error handling
- Fuzz test already catching this: `FuzzVoteState0_23_5`, `FuzzVoteState1_14_11`, `FuzzVoteState`
- Consider adding corpus entries with edge-case lockout counts (0, 31, 32, MaxUint64)

Recommendation: All deserialization paths for protocol-constrained data structures must validate input against protocol constants before allocation. This is especially critical for vote accounts which are frequently accessed during block processing.

[F-A05] `pkg/sealevel/vote_state.go` functions `VoteState0_23_5.UnmarshalWithDecoder`, `VoteState1_14_11.UnmarshalWithDecoder`, and `VoteState.UnmarshalWithDecoder` missing bounds check on epoch credits count causing DoS via unbounded memory allocation

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

Description

- `pkg/sealevel/vote_state.go` functions `VoteState0_23_5.UnmarshalWithDecoder`, `VoteState1_14_11.UnmarshalWithDecoder`, and `VoteState.UnmarshalWithDecoder` missing bounds check on epoch credits count causing DoS via unbounded memory allocation

```
// BUGGY CODE in VoteState.UnmarshalWithDecoder:
numEpochCredits, err := decoder.ReadUint64(bin.LE)
if err != nil {
    return err
}

voteState.EpochCredits = slices.Grow(voteState.EpochCredits, int(numEpochCredits))  // ❌ No bounds check!
for count := uint64(0); count < numEpochCredits; count++ {
    var epochCredits EpochCredits
    err = epochCredits.UnmarshalWithDecoder(decoder)
    if err != nil {
        return err
    }
    voteState.EpochCredits = append(voteState.EpochCredits, epochCredits)
}

// BUGGY CODE in VoteState0_23_5.UnmarshalWithDecoder:
// Same pattern - reads numEpochCredits, allocates without validation using append

// BUGGY CODE in VoteState1_14_11.UnmarshalWithDecoder:
// Same pattern - reads numEpochCredits, allocates without validation using append
```

Issue: All three VoteState deserialization functions read untrusted `numEpochCredits` values from input data and use them directly to allocate memory (via `slices.Grow` or `append`) without validating against the protocol's `MaxEpochCreditsHistory` constant (64). This allows attackers to trigger Out-of-Memory crashes when deserializing vote account data.

Impact:

- **Critical DoS vulnerability**: Malicious vote account data can crash validators during account deserialization
- **Resource exhaustion**: Large values cause OOM - fuzzer triggered with ~7.9 billion epoch credits → 96 GB allocation attempt
- **Multiple attack vectors**: All three vote state version formats are vulnerable
- **Affects account storage**: Stored vote account data with excessive epoch credits will crash on load
- Runtime crash: `fatal error: runtime: out of memory` when `slices.Grow` attempts huge allocation
- Triggered by fuzzing: `FuzzVoteState`

- **Note**: This bug was discovered AFTER the lockouts bug was fixed, showing the importance of comprehensive fuzzing

Trigger Example:

- Craft malicious vote account data with `numEpochCredits = 0x1d89d89d8` (~7.9 billion)
- Store in account or provide via RPC/snapshot
- Validator attempts to deserialize account state
- `slices.Grow(voteState.EpochCredits, int(0x1d89d89d8))` attempts to allocate 96 GB
- **Fatal error**: `runtime: out of memory` → immediate process termination

Root Cause: Missing validation against protocol constant `MaxEpochCreditsHistory = 64` before memory allocation. While the lockouts validation was added, the epoch credits validation was overlooked. Both `numLockouts` and `numEpochCredits` come from untrusted input and must be validated.

Comparison to Lockouts Bug: This bug is similar to the VoteState lockouts bug that was already fixed:

- **Lockouts bug (fixed)**: Missing validation of `numLockouts` against `MaxLockoutHistory` (31)
- **Epoch credits bug (this)**: Missing validation of `numEpochCredits` against `MaxEpochCreditsHistory` (64)
- **Both**: Read from untrusted input, cause unbounded allocation, crash validator

Suggested Fix:

```
// For VoteState.UnmarshalWithDecoder:
func (voteState *VoteState) UnmarshalWithDecoder(decoder *bin.Decoder) error {
    // ... earlier code ...

    numEpochCredits, err := decoder.ReadUint64(bin.LE)
    if err != nil {
        return err
    }

    // ✅ Validate against protocol maximum
    if numEpochCredits > MaxEpochCreditsHistory {
        return fmt.Errorf("too many epoch credits: %d (max %d)", numEpochCredits, MaxEpochCreditsHistory)
    }

    voteState.EpochCredits = slices.Grow(voteState.EpochCredits, int(numEpochCredits))
    for count := uint64(0); count < numEpochCredits; count++ {
        var epochCredits EpochCredits
        err = epochCredits.UnmarshalWithDecoder(decoder)
        if err != nil {
            return err
        }
        voteState.EpochCredits = append(voteState.EpochCredits, epochCredits)
    }
    // ... rest of function
}

// Apply same fix to VoteState0_23_5.UnmarshalWithDecoder (line ~1018)
// Apply same fix to VoteState1_14_11.UnmarshalWithDecoder (line ~1672)
```

Alternative Fix: Add a general safety check for unreasonably large allocations:

```
// Conservative limit: even if protocol allows more, prevent DoS
const MaxReasonableEpochCredits = 1000
```

```
if numEpochCredits > MaxReasonableEpochCredits {
    return fmt.Errorf("allocation too large: %d epoch credits (max %d)", numEpochCredits, MaxReasonableEpochCredits)
}
```

[F-A06] `pkg/tpu/tpu.go` function `ParseTx` does not validate transaction structure consistency (signature count, instruction indices)

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

Description

- `pkg/tpu/tpu.go` function `ParseTx` does not validate transaction structure consistency (signature count, instruction indices)

```
// VULNERABLE CODE (lines 29-46):
func ParseTx(p []byte) (tx *solana.Transaction, err error) {
    defer func() {
        if r := recover(); r != nil {
            err = errors.New("ParseTx panic")
        }
    }()

    tx, err = solana.TransactionFromDecoder(bin.NewBinDecoder(p))
    if err != nil {
        return nil, err
    }
    return  // ❌ No validation of signature count or instruction indices
}
```

Issue: The `ParseTx` function successfully parses transactions with multiple structural inconsistencies without performing validation:

1. **Signature count mismatch**: `len(tx.Signatures)` doesn't match `tx.Message.Header.NumRequiredSignatures`
2. **Header constraint violation**: `tx.Message.Header.NumRequiredSignatures` exceeds the number of account keys `len(tx.Message.AccountKeys)`
3. **Out-of-bounds instruction indices**: `tx.Message.Instructions[i].ProgramIDIndex` can reference account indices that exceed `len(tx.Message.AccountKeys)`
4. **Out-of-bounds account references**: Instruction account indices can exceed the account keys array

Impact:

- **DoS vulnerability**: Malformed transactions can crash the validator when instruction execution attempts to access out-of-bounds indices
- **Late failure**: Errors are only detected when processing the transaction (e.g., `tx.ResolveProgramIDIndex()` in `pkg/replay/transaction.go:306`), not during parsing
- **Attack surface**: Network peers or RPC clients can send malformed transactions that pass parsing but crash during execution
- **Missing defense-in-depth**: Validation should happen at parse time to reject malformed data early
- **Signature verification bypass potential**: If signature count is inconsistent, verification logic may fail in unexpected ways
- **ExtractSigners logic errors**: If `NumRequiredSignatures` exceeds account count, signer extraction fails with unexpected results
- Caught by fuzzing:
  - `FuzzTransactionInstructions` with input `{numInstrs=1, programIdIdx=255, numAcctIndices=0, instrData=[]}`
  - `FuzzTransactionDeserialization` with malformed transaction having 0 signatures but header claiming 48
  - `FuzzExtractSigners` with transaction having `NumRequiredSignatures=3` but only 2 account keys

Trigger Example 1 - Signature Count Mismatch:

Fuzzer input (from corpus `acdfca9b945299d0`: `[]byte("\x00000\x0000000000000000000000000000000000\x00")`):

- Transaction binary with `NumRequiredSignatures = 48` in header
- But only 0 actual signature entries in transaction

Test output:

```
--- FAIL: FuzzTransactionDeserialization (0.00s)
    tpu_fuzz_test.go:83: Signature count mismatch: got 0 signatures but header says 48
```

Trigger Example 2 - Out-of-Bounds Program Index:

Fuzzer input (seed #3):

- `numInstrs = 1` (1 instruction)
- `programIdIdx = 255` (references non-existent account)
- `numAcctIndices = 0` (no account indices)
- `instrData = []` (no instruction data)

Transaction structure:

```
Message.AccountKeys = [account0, account1]  // Only 2 accounts
Message.Instructions[0].ProgramIDIndex = 255  // ❌ Out of bounds (>= 2)
```

Parsing succeeds, but later when processing:

```
// In pkg/replay/transaction.go:306
programId, err := tx.ResolveProgramIDIndex(compiledInstr.ProgramIDIndex)
// This will fail or panic trying to access AccountKeys[255]
```

Test output:

```
--- FAIL: FuzzTransactionInstructions (0.00s)
    --- FAIL: FuzzTransactionInstructions/seed#2 (0.00s)
        tpu_fuzz_test.go:362: Instruction 0 has invalid program_id_index 255 (only 2 accounts)
```

Trigger Example 3 - Header Constraint Violation:

Fuzzer input (from corpus `e7fb9b44e17eb333`:
`[]byte("\x03000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000`
):

- Transaction binary with `NumRequiredSignatures = 3` in header
- But only 2 account keys present in `tx.Message.AccountKeys`
- Both account keys identical: `4F7BsTMVPKFshM1MwLf6y23cid6fL3xMpazVoF9krzUw`

Transaction structure:

```
Message.Header.NumRequiredSignatures = 3
Message.Header.NumReadonlySignedAccounts = 48
Message.Header.NumReadonlyUnsignedAccounts = 48
Message.AccountKeys = [account0, account1]  // Only 2 accounts
```

Parsing succeeds, but later when extracting signers:

```
// In pkg/tpu/tpu.go:142 (ExtractSigners)
for i := 0; i < int(tx.Message.Header.NumRequiredSignatures); i++ {
```

```
    // Tries to access AccountKeys[2] which doesn't exist
}
```

Test output:

```
--- FAIL: FuzzExtractSigners (0.00s)
    tpu_fuzz_test.go:465: Transaction header: NumRequiredSignatures=3, NumReadonlySignedAccounts=48,
NumReadonlyUnsignedAccounts=48
    tpu_fuzz_test.go:467: Number of account keys: 2
    tpu_fuzz_test.go:471: ExtractSigners returned 2 signers, expected 3
```

Root Cause: The underlying `solana.TransactionFromDecoder` function (from the external solana-go library) does not validate that instruction account indices are within bounds. The client code should add this validation layer.

Security Implications:

1. **Panic risk**: Attempting to access `tx.Message.AccountKeys[255]` when only 2 accounts exist will cause array index out of bounds
2. **Header constraint violation**: `NumRequiredSignatures` exceeding account count leads to logic errors in `ExtractSigners` and signature verification
3. **Signature bypass**: Inconsistent signature counts may lead to verification logic failures
4. **Processing waste**: Invalid transactions consume CPU/memory resources before being rejected
5. **Error handling complexity**: Errors detected late in the pipeline are harder to attribute and debug
6. **Byzantine resistance**: A validator must reject malformed transactions to maintain consensus

Suggested Fix:

Add comprehensive validation to `ParseTx`:

```
func ParseTx(p []byte) (tx *solana.Transaction, err error) {
    defer func() {
        if r := recover(); r != nil {
            err = errors.New("ParseTx panic")
        }
    }()

    tx, err = solana.TransactionFromDecoder(bin.NewBinDecoder(p))
    if err != nil {
        return nil, err
    }

    // ✅ Validate signature count matches header
    if len(tx.Signatures) != int(tx.Message.Header.NumRequiredSignatures) {
        return nil, fmt.Errorf("signature count mismatch: got %d signatures but header requires %d",
            len(tx.Signatures), tx.Message.Header.NumRequiredSignatures)
    }

    // ✅ Validate header constraints
    numAccounts := len(tx.Message.AccountKeys)
    if int(tx.Message.Header.NumRequiredSignatures) > numAccounts {
        return nil, fmt.Errorf("invalid header: NumRequiredSignatures (%d) exceeds account count (%d)",
            tx.Message.Header.NumRequiredSignatures, numAccounts)
    }

    // ✅ Validate instruction account indices
    for i, instr := range tx.Message.Instructions {
        // Validate program_id_index
```

```
        if int(instr.ProgramIDIndex) >= numAccounts {
            return nil, fmt.Errorf("instruction %d: program_id_index %d out of bounds (only %d accounts)",
                i, instr.ProgramIDIndex, numAccounts)
        }

        // Validate all account indices referenced by the instruction
        for j, acctIdx := range instr.Accounts {
            if int(acctIdx) >= numAccounts {
                return nil, fmt.Errorf("instruction %d account %d: index %d out of bounds (only %d accounts)",
                    i, j, acctIdx, numAccounts)
            }
        }
    }

    return tx, nil
}
```

Alternative Fix (if modifying ParseTx is not desired):

Create a separate validation function and call it immediately after parsing:

```
func ValidateTransaction(tx *solana.Transaction) error {
    // Validate signature count
    if len(tx.Signatures) != int(tx.Message.Header.NumRequiredSignatures) {
        return fmt.Errorf("signature count mismatch: got %d signatures but header requires %d",
            len(tx.Signatures), tx.Message.Header.NumRequiredSignatures)
    }

    // Validate header constraints
    numAccounts := len(tx.Message.AccountKeys)
    if int(tx.Message.Header.NumRequiredSignatures) > numAccounts {
        return fmt.Errorf("invalid header: NumRequiredSignatures (%d) exceeds account count (%d)",
            tx.Message.Header.NumRequiredSignatures, numAccounts)
    }

    // Validate instruction indices
    for i, instr := range tx.Message.Instructions {
        if int(instr.ProgramIDIndex) >= numAccounts {
            return fmt.Errorf("instruction %d: program_id_index %d out of bounds (only %d accounts)",
                i, instr.ProgramIDIndex, numAccounts)
        }

        for j, acctIdx := range instr.Accounts {
            if int(acctIdx) >= numAccounts {
                return fmt.Errorf("instruction %d account %d: index %d out of bounds (only %d accounts)",
                    i, j, acctIdx, numAccounts)
            }
        }
    }

    return nil
}

// Use it in all parsing paths:
tx, err := ParseTx(data)
if err != nil {
    return err
```

```
}
if err := ValidateTransaction(tx); err != nil {
    return err
}
```

Verification: After fixing, the tests should pass:

```
# Re-run the failing signature count case
go test -run=FuzzTransactionDeserialization/acdfca9b945299d0 ./pkg/tpu -v

# Re-run the failing instruction index case
go test -run=FuzzTransactionInstructions/seed#2 ./pkg/tpu -v

# Re-run the failing header constraint case
go test -run=FuzzExtractSigners/e7fb9b44e17eb333 ./pkg/tpu -v

# Run fresh fuzzing on all related tests
go test -fuzz=FuzzTransactionDeserialization -fuzztime=1m ./pkg/tpu
go test -fuzz=FuzzTransactionInstructions -fuzztime=1m ./pkg/tpu
go test -fuzz=FuzzExtractSigners -fuzztime=1m ./pkg/tpu
```

Additional Validation that could be added:

- Header field constraints (e.g., `NumRequiredSignatures <= len(AccountKeys)`)
- Recent blockhash is non-zero (if required by protocol)
- Instruction data lengths don't exceed reasonable limits
- Total account keys doesn't exceed maximum allowed

Recommendation: Apply the comprehensive validation fix immediately to prevent DoS attacks from malformed transactions. This is a critical security issue that allows network peers to crash validators with crafted payloads.

[F-A07] Function `Decode32` can panic with index out of range

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

Description

The function `pkg/base58/base58.go/Decode32` should return `false` if any character in the input is not a valid `base58` character.

runtimeverification/_audits_Overclock-Validator_mithril/pkg/base58/base58.go

Line 192 to 200 in 06f7b95

```
192   for _, c := range encoded {
193       idx := int(c) - int(inverseLUTOffset)
194       if idx > int(inverseLUTSentinel) {
195           idx = int(inverseLUTSentinel)
196       }
197       if inverseLUT[idx] == invalidChar {
198           return false
199       }
200   }
```

The `inverseLUTOffset` is '1' (ASCII 49), and `inverseLUTSentinel` is `1 + 'z' - '1' = 75`. The `inverseLUT` array is defined with exactly 75 elements:

runtimeverification/_audits_Overclock-Validator_mithril/pkg/base58/base58.go

Line 23 to 38 in 06f7b95

```
23  var inverseLUT = [75]byte{
24    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
25    0x08, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
26    0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F, 0x10,
27    0xFF, 0x11, 0x12, 0x13, 0x14, 0x15, 0xFF, 0x16,
28    0x17, 0x18, 0x19, 0x1A, 0x1B, 0x1C, 0x1D, 0x1E,
29    0x1F, 0x20, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
30    0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27, 0x28,
31    0x29, 0x2A, 0x2B, 0xFF, 0x2C, 0x2D, 0x2E, 0x2F,
32    0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37,
33    0x38, 0x39, 0xFF,
34  }
35
36  const invalidChar uint8 = 0xFF
37  const inverseLUTOffset uint8 = '1'
38  const inverseLUTSentinel uint8 = 1 + 'z' - inverseLUTOffset
```

Scenario

Any string containing characters with ASCII values less than '1' (0x31) will cause an index out of range panic instead of returning `false`. This includes: ' ' (space), '!', "", '#', ... '0', and all control characters.

Recommendation

The code should check if `idx` is negative before accessing the array:

```
  for _, c := range encoded {
    idx := int(c) - int(inverseLUTOffset)
    if idx < 0 || idx > int(inverseLUTSentinel) {
        idx = int(inverseLUTSentinel)
    }
    if inverseLUT[idx] == invalidChar {
        return false
    }
}
```

Status

Fixed in commit 7894ffe.

[F-A08] `SaturatingMulU8` and `SaturatingMulU16` functions can miss some overflows

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

Description

The functions `SaturatingMulU8` and `SaturatingMulU16` in `pkg/safemath/saturating.go` should detect all overflow cases and saturate at the maximum value. However, in the current implementation, they can miss overflows in certain cases:

```
// BUGGY CODE:
func SaturatingMulU8(a, b uint8) uint8 {
    if a == 0 || b == 0 {
        return 0
    }
    result := a * b
    if result < a {  // ❌ Only catches SOME overflows
        return math.MaxUint8
    }
    return result
}

// BUGGY CODE:
func SaturatingMulU16(a, b uint16) uint16 {
    if a == 0 || b == 0 {
        return 0
    }
    result := a * b
    if result < a {  // ❌ Only catches SOME overflows
        return math.MaxUint16
    }
    return result
}
```

Scenario:

- `a = 190` and `b = 163`
- True result: `190 * 163 = 30970`
- Wrapped: `30970 % 256 = 250`
- Check: `250 < 190` -> False -> Returns 250 ❌ (should be 255)

Recommendation

```
func SaturatingMulU8(a, b uint8) uint8 {
    result := uint16(a) * uint16(b)
    if result > math.MaxUint8 {
        return math.MaxUint8
    }
    return uint8(result)
}

func SaturatingMulU16(a, b uint16) uint16 {
```

```
    result := uint32(a) * uint32(b)
    if result > math.MaxUint16 {
        return math.MaxUint16
    }
    return uint16(result)
}
```

Status

Fixed in commit 7ea833d.

[F-A09] `AppendVecAccount.Unmarshal` - Missing `DataLen` Validation

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

Summary

The `AppendVecAccount.Unmarshal()` function in `pkg/accountsdb` allocates memory based on an untrusted `DataLen` field read from append vector files without validation, causing out-of-memory crashes when processing malicious or corrupted snapshot data.

Location

- **File**: `pkg/accountsdb/appendvec.go`
- **Function**: `AppendVecAccount.Unmarshal()`
- **Trigger point**: Any snapshot ingestion or account database loading operation
- **Missing validation**: No bounds check on `DataLen` before `make([]byte, acct.DataLen)`

Issue

The `Unmarshal` function reads a `DataLen` field (uint64) from untrusted binary input and immediately uses it to allocate a byte slice without any validation. Since `DataLen` can be up to 2^64 bytes (18 exabytes), a malicious or corrupted append vector file can cause the node to attempt allocating hundreds of terabytes of memory, resulting in an immediate out-of-memory crash.

Issue

The `Unmarshal` function reads a `DataLen` field (uint64) from untrusted binary input and immediately uses it to allocate a byte slice without any validation. Since `DataLen` can be up to 2^64 bytes (18 exabytes), a malicious or corrupted append vector file can cause the node to attempt allocating hundreds of terabytes of memory, resulting in an immediate out-of-memory crash.

Vulnerable Code Path

```
// In appendvec.go - Unmarshal() reads DataLen from untrusted input
func (acct *AppendVecAccount) Unmarshal(buf io.Reader) error {
    var err error
    var hdrBytes [hdrLen]byte
    _, err = buf.Read(hdrBytes[:])
    if err != nil {
        return err
    }

    acct.WriteVersion = binary.LittleEndian.Uint64(hdrBytes[:8])
    acct.DataLen = binary.LittleEndian.Uint64(hdrBytes[8:16])  // Line 151 - Read untrusted DataLen
    copy(acct.Pubkey[:], hdrBytes[16:48])
    acct.Lamports = binary.LittleEndian.Uint64(hdrBytes[48:56])
    acct.RentEpoch = binary.LittleEndian.Uint64(hdrBytes[56:64])
    copy(acct.Owner[:], hdrBytes[64:96])
    acct.Executable = hdrBytes[96] != 0
    copy(acct.Padding[:], hdrBytes[97:104])
    copy(acct.Hash[:], hdrBytes[104:136])

    acct.Data = make([]byte, acct.DataLen)  // Line 160 - ❌ No validation!
```

```
    _, err = buf.Read(acct.Data)

    return err
}
```

Impact

- **DoS vulnerability**: Malicious snapshots crash validators immediately upon loading
- **Resource exhaustion**: System becomes unresponsive before crash as kernel attempts memory allocation
- **Network partition**: Multiple validators crash simultaneously if malicious snapshot is distributed
- **Bootstrap attack**: New validators syncing from corrupted snapshot cannot start
- **Cascade failure**: Single malicious snapshot can take down multiple nodes in the network

---

Fuzzing Discovery

Fuzz test: `FuzzAppendVecAccountUnmarshal`

Failing input corpus: `testdata/fuzz/FuzzAppendVecAccountUnmarshal/24d4a37af740d8fe`

Trigger:

- Append vector file with `DataLen` = `0xcf0e33bfab26` (227,659,904,712,704 bytes = **227 TB**)
- Fuzzer discovered this within 1 second (~36,000 iterations)

Error output:

```
runtime: out of memory: cannot allocate 227659904712704-byte block (3833856 in use)
fatal error: out of memory

goroutine 21 [running]:
runtime.throw({0x7c7997?, 0x4?})
runtime.(*mcache).allocLarge(0x0?, 0xcf0e33bfab26, 0x1)
runtime.mallocgc(0xcf0e33bfab26, 0x73c180, 0x1)
runtime.makeslice(0xc00020b6e0?, 0xc0001e82d0?, 0x28?)
github.com/Overclock-Validator/mithril/pkg/accountsdb.(*AppendVecAccount).Unmarshal(0xc000201728, {0x8436e0, 0xc00020b6e0})
    /home/yiyiwang/Workspace/_audits_Overclock-Validator_mithril/pkg/accountsdb/appendvec.go:160 +0x125
github.com/Overclock-Validator/mithril/pkg/accountsdb.FuzzAppendVecAccountUnmarshal.func1(0xc0001a1180, {0xc0001810f0, 0x0?, 0x10})
    /home/yiyiwang/Workspace/_audits_Overclock-Validator_mithril/pkg/accountsdb/appendvec_fuzz_test.go:28 +0xd3
```

Root Cause

The function trusts the `DataLen` field from untrusted binary data without validation. The `DataLen` field is a `uint64`, allowing values up to 18 exabytes (2^64 bytes). There are no checks to ensure:

1. `DataLen` is within reasonable bounds for an account (Solana max is 10 MB)
2. `DataLen` matches available buffer size
3. `DataLen` doesn't exceed system memory limits

Suggested Fixes

Option 1: Add Maximum Size Validation (Recommended)

Add validation when reading `DataLen` from header:

```
// In pkg/accountsdb/appendvec.go
func (acct *AppendVecAccount) Unmarshal(buf io.Reader) error {
```

```
    const MaxAccountDataLen = 10 * 1024 * 1024 // 10 MB max (Solana limit)

    var err error
    var hdrBytes [hdrLen]byte
    _, err = buf.Read(hdrBytes[:])
    if err != nil {
        return err
    }

    acct.WriteVersion = binary.LittleEndian.Uint64(hdrBytes[:8])
    acct.DataLen = binary.LittleEndian.Uint64(hdrBytes[8:16])

    // ✅ Validate DataLen is reasonable
    if acct.DataLen > MaxAccountDataLen {
        return fmt.Errorf("account data length %d exceeds maximum %d", acct.DataLen, MaxAccountDataLen)
    }

    copy(acct.Pubkey[:], hdrBytes[16:48])
    acct.Lamports = binary.LittleEndian.Uint64(hdrBytes[48:56])
    acct.RentEpoch = binary.LittleEndian.Uint64(hdrBytes[56:64])
    copy(acct.Owner[:], hdrBytes[64:96])
    acct.Executable = hdrBytes[96] != 0
    copy(acct.Padding[:], hdrBytes[97:104])
    copy(acct.Hash[:], hdrBytes[104:136])

    acct.Data = make([]byte, acct.DataLen)
    _, err = buf.Read(acct.Data)

    return err
}
```

Pros:

- Simple one-line check
- Matches Solana's account size limits
- Clear error message for debugging
- Prevents allocation before it happens
- Minimal performance impact

Option 2: Check Against Available Buffer Size

Validate `DataLen` against remaining buffer size:

```
// In pkg/accountsdb/appendvec.go
func (acct *AppendVecAccount) Unmarshal(buf io.Reader) error {
    var err error
    var hdrBytes [hdrLen]byte
    _, err = buf.Read(hdrBytes[:])
    if err != nil {
        return err
    }

    acct.WriteVersion = binary.LittleEndian.Uint64(hdrBytes[:8])
    acct.DataLen = binary.LittleEndian.Uint64(hdrBytes[8:16])

    // ✅ Check if DataLen fits in available buffer
    // For appendVecParser, check against remaining file size
```

```
    if sizer, ok := buf.(interface{ Size() int64 }); ok {
        if int64(acct.DataLen) > sizer.Size() - hdrLen {
            return fmt.Errorf("data length %d exceeds available buffer size", acct.DataLen)
        }
    }

    copy(acct.Pubkey[:], hdrBytes[16:48])
    acct.Lamports = binary.LittleEndian.Uint64(hdrBytes[48:56])
    acct.RentEpoch = binary.LittleEndian.Uint64(hdrBytes[56:64])
    copy(acct.Owner[:], hdrBytes[64:96])
    acct.Executable = hdrBytes[96] != 0
    copy(acct.Padding[:], hdrBytes[97:104])
    copy(acct.Hash[:], hdrBytes[104:136])

    acct.Data = make([]byte, acct.DataLen)
    _, err = buf.Read(acct.Data)

    return err
}
```

Pros:

- Catches truncated files
- More dynamic validation
- Works with different buffer sizes

Cons:

- Requires interface type assertion
- May not work with all io.Reader implementations

Cons:

- Requires interface type assertion
- May not work with all io.Reader implementations

Option 3: Combined Approach (Defense in Depth)

Use both absolute limit and buffer size checks:

```
func (acct *AppendVecAccount) Unmarshal(buf io.Reader) error {
    const MaxAccountDataLen = 10 * 1024 * 1024 // 10 MB

    var err error
    var hdrBytes [hdrLen]byte
    _, err = buf.Read(hdrBytes[:])
    if err != nil {
        return err
    }

    acct.WriteVersion = binary.LittleEndian.Uint64(hdrBytes[:8])
    acct.DataLen = binary.LittleEndian.Uint64(hdrBytes[8:16])

    // ✅ Validate against absolute maximum
    if acct.DataLen > MaxAccountDataLen {
        return fmt.Errorf("account data length %d exceeds maximum %d", acct.DataLen, MaxAccountDataLen)
    }
```

```
    // ✅ Validate against available buffer (if possible)
    if sizer, ok := buf.(interface{ Len() int }); ok {
        if int64(acct.DataLen) > int64(sizer.Len()) {
            return fmt.Errorf("data length %d exceeds remaining buffer size %d", acct.DataLen, sizer.Len())
        }
    }

    copy(acct.Pubkey[:], hdrBytes[16:48])
    acct.Lamports = binary.LittleEndian.Uint64(hdrBytes[48:56])
    acct.RentEpoch = binary.LittleEndian.Uint64(hdrBytes[56:64])
    copy(acct.Owner[:], hdrBytes[64:96])
    acct.Executable = hdrBytes[96] != 0
    copy(acct.Padding[:], hdrBytes[97:104])
    copy(acct.Hash[:], hdrBytes[104:136])

    acct.Data = make([]byte, acct.DataLen)
    _, err = buf.Read(acct.Data)

    return err
}
```

Pros:

- Most secure approach
- Catches multiple error conditions
- Provides clear error messages

Cons:

- Slightly more complex
- Two validation checks

Recommendation

Use Option 1 (maximum size validation):

- Simplest and most effective
- Aligns with Solana protocol limits
- Easy to maintain
- Sufficient for production use

If paranoid about defense-in-depth, use **Option 3** (combined approach).

[F-A10] `SlotAcctVecs.UnmarshalWithDecoder` - Missing `numAcctVecs` Validation

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

Summary

The `SlotAcctVecs.UnmarshalWithDecoder()` function in `pkg/snapshot` allocates memory based on an untrusted `numAcctVecs` field read from snapshot manifest files without validation, causing out-of-memory crashes when processing malicious or corrupted snapshot data.

Location

- **File**: `pkg/snapshot/manifest_decoder.go`
- **Function**: `SlotAcctVecs.UnmarshalWithDecoder()`
- **Trigger point**: Any snapshot manifest deserialization operation
- **Missing validation**: No bounds check on `numAcctVecs` before `make([]AcctVec, 0, numAcctVecs)`

Issue

The `UnmarshalWithDecoder` function reads a `numAcctVecs` field (uint64) from untrusted binary input and immediately uses it to allocate a slice without any validation. Since `numAcctVecs` can be up to 2^64 elements, a malicious or corrupted snapshot manifest can cause the node to attempt allocating hundreds of exabytes of memory, resulting in an immediate out-of-memory crash.

Vulnerable Code Path

```
// In manifest_decoder.go - UnmarshalWithDecoder() reads numAcctVecs from untrusted input
func (slotAcctVecs *SlotAcctVecs) UnmarshalWithDecoder(decoder *bin.Decoder) error {
    var err error

    slotAcctVecs.Slot, err = decoder.ReadUint64(bin.LE)
    if err != nil {
        return err
    }

    var numAcctVecs uint64
    numAcctVecs, err = decoder.ReadUint64(bin.LE)  // Line 1833 - Read untrusted numAcctVecs
    if err != nil {
        return err
    }

    slotAcctVecs.AcctVecs = make([]AcctVec, 0, numAcctVecs)  // Line 1838 - ❌ No validation!
    for count := uint64(0); count < numAcctVecs; count++ {
        var acctVec AcctVec
        err = acctVec.UnmarshalWithDecoder(decoder)
        if err != nil {
            return err
        }
        slotAcctVecs.AcctVecs = append(slotAcctVecs.AcctVecs, acctVec)
    }
```

```
    return nil
}
```

Impact

- **DoS vulnerability**: Malicious snapshot manifests crash validators immediately upon loading
- **Resource exhaustion**: System becomes unresponsive before crash as kernel attempts memory allocation
- **Network partition**: Multiple validators crash simultaneously if malicious snapshot is distributed
- **Bootstrap attack**: New validators syncing from corrupted snapshot cannot start
- **Cascade failure**: Single malicious snapshot can take down multiple nodes in the network

---

Fuzzing Discovery

Fuzz test: `FuzzSnapshotManifestDeserialization`

Failing input corpus: `testdata/fuzz/FuzzSnapshotManifestDeserialization/343dd7a6b769cfbc`

Trigger:

- Snapshot manifest with `numAcctVecs` field set to extremely large value
- Fuzzer discovered this within 2 seconds

Error output:

```
panic: runtime error: makeslice: cap out of range

goroutine 487 [running]:
github.com/Overclock-Validator/mithril/pkg/snapshot.(*SlotAcctVecs).UnmarshalWithDecoder(0xc0008536a8, 0xc006ede840)
    /home/yiyiwang/Workspace/_audits_Overclock-Validator_mithril/pkg/snapshot/manifest_decoder.go:1838 +0x126
github.com/Overclock-Validator/mithril/pkg/snapshot.(*AccountsDbFields).UnmarshalWithDecoder(0xc000853768, 0xc006ede840)
    /home/yiyiwang/Workspace/_audits_Overclock-Validator_mithril/pkg/snapshot/manifest_decoder.go:2057 +0x177
github.com/Overclock-Validator/mithril/pkg/snapshot.FuzzSnapshotManifestDeserialization.func1(0x0?, {0xc000055c00, 0x18, 0x400})
    /home/yiyiwang/Workspace/_audits_Overclock-Validator_mithril/pkg/snapshot/manifest_fuzz_test.go:37 +0x165
```

Root Cause

The function trusts the `numAcctVecs` field from untrusted binary data without validation. The `numAcctVecs` field is a `uint64`, allowing values up to 18 quintillion (2^64). There are no checks to ensure:

1. `numAcctVecs` is within reasonable bounds for a slot
2. `numAcctVecs` matches available buffer size
3. `numAcctVecs` doesn't exceed system memory limits

Additional Vulnerable Instances

The same pattern exists in multiple other deserialization functions in `manifest_decoder.go`:

Function	Field
`Stakes0_23_5.UnmarshalWithDecoder`	`VoteAccounts` (via `numVoteAccts`)
`Stakes0_23_5.UnmarshalWithDecoder`	`Delegations` (via `numStakeDelegations`)
`Stakes.UnmarshalWithDecoder`	`VoteAccounts` (via `numVoteAccts`)
`Stakes.UnmarshalWithDecoder`	`StakeDelegations` (via `numStakeDelegations`)
`VersionedEpochStakes.UnmarshalWithDecoder`	`NodeIdToVoteAccounts` (via `numAccts`)
`VersionedEpochStakes.UnmarshalWithDecoder`	`EpochAuthorizedVoters` (via `numEpochAuthVoters`)
`SlotAcctVecs.UnmarshalWithDecoder`	**`AcctVecs` (via `numAcctVecs`)**
`BankFields.UnmarshalWithDecoder`	`VersionedEpochStakes` (via `numVersionedEpochStakes`)

All require similar fixes.

Suggested Fixes

Option 1: Add Maximum Size Validation (Recommended)

Add validation when reading `numAcctVecs` from decoder:

```go
// In pkg/snapshot/manifest_decoder.go
func (slotAcctVecs *SlotAcctVecs) UnmarshalWithDecoder(decoder *bin.Decoder) error {
    const MaxAcctVecs = 1_000_000 // Adjust based on Solana limits

    var err error

    slotAcctVecs.Slot, err = decoder.ReadUint64(bin.LE)
    if err != nil {
        return err
    }

    var numAcctVecs uint64
    numAcctVecs, err = decoder.ReadUint64(bin.LE)
    if err != nil {
        return err
    }

    // ✅ Validate numAcctVecs is reasonable
    if numAcctVecs > MaxAcctVecs {
        return fmt.Errorf("account vector count %d exceeds maximum %d", numAcctVecs, MaxAcctVecs)
    }

    slotAcctVecs.AcctVecs = make([]AcctVec, 0, numAcctVecs)
    for count := uint64(0); count < numAcctVecs; count++ {
        var acctVec AcctVec
        err = acctVec.UnmarshalWithDecoder(decoder)
        if err != nil {
            return err
        }
        slotAcctVecs.AcctVecs = append(slotAcctVecs.AcctVecs, acctVec)
    }

    return nil
}
```

Pros:

- Simple one-line check

- Prevents allocation before it happens
- Clear error message for debugging
- Minimal performance impact

Option 2: Incremental Allocation

Allocate without pre-sizing and add bounds check in loop:

```
// Start with zero capacity, grow as needed
slotAcctVecs.AcctVecs = make([]AcctVec, 0)
for count := uint64(0); count < numAcctVecs; count++ {
    // Add safety check to prevent excessive allocations
    if count > 1_000_000 {
        return fmt.Errorf("too many account vectors")
    }

    var acctVec AcctVec
    err = acctVec.UnmarshalWithDecoder(decoder)
    if err != nil {
        return err
    }
    slotAcctVecs.AcctVecs = append(slotAcctVecs.AcctVecs, acctVec)
}
```

Pros:

- Avoids upfront allocation
- Only allocates what's actually read

Cons:

- Slightly less efficient (repeated reallocation)
- Still needs bounds check to prevent infinite loop

Option 3: Combined Approach (Defense in Depth)

Use both absolute limit and incremental growth:

```
func (slotAcctVecs *SlotAcctVecs) UnmarshalWithDecoder(decoder *bin.Decoder) error {
    const MaxAcctVecs = 1_000_000

    var err error

    slotAcctVecs.Slot, err = decoder.ReadUint64(bin.LE)
    if err != nil {
        return err
    }

    var numAcctVecs uint64
    numAcctVecs, err = decoder.ReadUint64(bin.LE)
    if err != nil {
        return err
    }

    // ✅ Validate against absolute maximum
    if numAcctVecs > MaxAcctVecs {
        return fmt.Errorf("account vector count %d exceeds maximum %d", numAcctVecs, MaxAcctVecs)
```

```
    }

    slotAcctVecs.AcctVecs = make([]AcctVec, 0, numAcctVecs)
    for count := uint64(0); count < numAcctVecs; count++ {
        var acctVec AcctVec
        err = acctVec.UnmarshalWithDecoder(decoder)
        if err != nil {
            return err
        }
        slotAcctVecs.AcctVecs = append(slotAcctVecs.AcctVecs, acctVec)
    }

    return nil
}
```

Pros:

- Most secure approach
- Provides clear error messages
- Efficient once validated

Cons:

- Slightly more code

Recommendation

Use Option 1 (maximum size validation):

- Simplest and most effective
- Sufficient for production use
- Easy to maintain

If paranoid about defense-in-depth, use **Option 3** (combined approach).

[F-A11] Account Database - Missing Buffer Length Validation in Append Vector Parser

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

Summary

The `ParseNextAcct` method of `appendVecParser` validates offsets against the `FileSize` parameter but accesses data from the `Buf` slice without validating `len(Buf)`. This causes a runtime panic when `FileSize > len(Buf)`, which can occur with corrupted or truncated append vector files, instead of gracefully rejecting them with an error.

Location

- **File**: `pkg/accountsdb/appendvec.go`
- **Function**: `ParseNextAcct()` (line 104)
- **Trigger point**: `BuildIndexEntriesFromAppendVecs()` → `parser.ParseNextAcct()` (line 142)
- **Missing validation**: No check that `len(parser.Buf) >= parser.FileSize`

Issue

The `ParseNextAcct` method assumes that the `Buf` slice contains at least `FileSize` bytes, but this constraint is not validated. When a corrupted or truncated file results in `len(Buf) < FileSize`, the code panics with "slice bounds out of range" instead of returning a validation error.

Vulnerable Code Path

```
// In appendvec.go - ParseNextAcct() accesses Buf WITHOUT length validation
func (parser *appendVecParser) ParseNextAcct(pk *solana.PublicKey, a *AccountIndexEntry) error {
    if parser.Offset+hdrLen > parser.FileSize {  // ✅ Validates against FileSize
        return fmt.Errorf("overflow")
    }

    dataLen := binary.LittleEndian.Uint64(parser.Buf[parser.Offset+dataLenOffset : parser.Offset+dataLenOffset+8])

    // ❌ PANIC HERE: No validation that len(Buf) >= parser.Offset+pubkeyOffset+32
    *pk = solana.PublicKeyFromBytes(parser.Buf[parser.Offset+pubkeyOffset : parser.Offset+pubkeyOffset+32])

    a.Slot = parser.Slot
    a.FileId = parser.FileId
    a.Offset = parser.Offset

    parser.Offset += hdrLen

    if parser.Offset+dataLen > parser.FileSize {
        return fmt.Errorf("overflow")
    }

    parser.Offset += util.AlignUp(dataLen, 8)

    return nil
}
```

```
// In index.go - BuildIndexEntriesFromAppendVecs creates parser without validation
func BuildIndexEntriesFromAppendVecs(data []byte, fileSize uint64, slot uint64, fileId uint64) ([]solana.PublicKey,
[]AccountIndexEntry, error) {
    // ❌ Missing: validation that len(data) >= fileSize
    parser := &appendVecParser{Buf: data, FileSize: fileSize, FileId: fileId, Slot: slot}

    for {
        pubkeys = append(pubkeys, solana.PublicKey{})
        acctIdxEntries = append(acctIdxEntries, AccountIndexEntry{})
        err = parser.ParseNextAcct(&pubkeys[len(pubkeys)-1], &acctIdxEntries[len(acctIdxEntries)-1])
        if err != nil {
            break
        }
    }

    return pubkeys, acctIdxEntries, nil
}
```

Panic Trigger:

```
panic: runtime error: slice bounds out of range [:X] with capacity Y
```

Where `X > Y` because `len(Buf) < FileSize`.

Impact

- **DoS vulnerability**: Corrupted append vector files crash the node process
- **Violation of error handling best practices**: Input validation should return errors, not panic
- **Poor error handling**: No graceful degradation for corrupted data
- **Security issue**: Panics from corrupted disk data are exploitable for denial-of-service
- **Late detection**: The error is detected during parsing after FileSize has been accepted, when it should be validated upfront

---

Fuzzing Discovery

Fuzz test: `FuzzBuildIndexEntriesFromAppendVecs`

Trigger: Fuzzer generates inputs where `len(data) < fileSize`

Reproducing:

```
go test ./pkg/accountsdb -run=^$ -fuzz=FuzzBuildIndexEntriesFromAppendVecs$ -fuzztime=5s
```

Example manual trigger:

```
parser := &appendVecParser{
    Buf:      make([]byte, 10),  // Only 10 bytes
    FileSize: 1000,              // Claims 1000 bytes
    FileId:   1,
    Slot:     100,
}

var pk solana.PublicKey
var entry AccountIndexEntry
```

```
// Panics: slice bounds out of range
parser.ParseNextAcct(&pk, &entry)
```

Root Cause

The parser assumes `Buf` contains at least `FileSize` bytes, but this constraint is not enforced:

1. **Mismatch between FileSize and actual buffer length**: `FileSize` parameter represents the *expected* file size, but `Buf` may be smaller if:
   - File is truncated during reading
   - File was corrupted on disk
   - Fuzzing provides mismatched parameters
   - I/O error occurred during file read
2. **Inconsistent validation**: The code validates `parser.Offset+hdrLen > parser.FileSize` but should validate `parser.Offset+hdrLen > len(parser.Buf)`
3. **Assumes well-formed input**: The parser doesn't defend against corrupted or malicious input data

---

Production Usage Analysis

The caller in `BuildIndexEntriesFromAppendVecs` typically reads a file and passes `len(data)` as `fileSize`, so in normal operation this works. However:

1. If file is truncated during read, `len(data) < actual file size`
2. If file metadata is corrupted, fileSize metadata may be wrong
3. The function doesn't enforce `len(data) >= fileSize` precondition

Current code works in production because files are read completely, but it's **fragile** and vulnerable to corruption.

Suggested Fixes

Option 1: Validate in `ParseNextAcct()` (Recommended)

Add defensive check at the start of `ParseNextAcct`:

```
func (parser *appendVecParser) ParseNextAcct(pk *solana.PublicKey, a *AccountIndexEntry) error {
    // ✅ Validate buffer is large enough for claimed FileSize
    if uint64(len(parser.Buf)) < parser.FileSize {
        return fmt.Errorf("buffer underflow: buffer has %d bytes but FileSize claims %d",
            len(parser.Buf), parser.FileSize)
    }

    if parser.Offset+hdrLen > parser.FileSize {
        return fmt.Errorf("overflow")
    }

    // Now safe to access parser.Buf up to FileSize
    dataLen := binary.LittleEndian.Uint64(parser.Buf[parser.Offset+dataLenOffset : parser.Offset+dataLenOffset+8])
    // ... rest unchanged
}
```

Pros:

- Validates at the earliest appropriate point

- Clear error message for debugging
- Minimal code changes
- Preserves existing API

Option 2: Use `len(Buf)` Instead of `FileSize`

Change all validations to use actual buffer length:

```
func (parser *appendVecParser) ParseNextAcct(pk *solana.PublicKey, a *AccountIndexEntry) error {
    bufLen := uint64(len(parser.Buf))

    // ✅ Validate against actual buffer length
    if parser.Offset+hdrLen > bufLen {
        return fmt.Errorf("overflow: offset %d + header %d exceeds buffer size %d",
            parser.Offset, hdrLen, bufLen)
    }

    dataLen := binary.LittleEndian.Uint64(parser.Buf[parser.Offset+dataLenOffset : parser.Offset+dataLenOffset+8])

    *pk = solana.PublicKeyFromBytes(parser.Buf[parser.Offset+pubkeyOffset : parser.Offset+pubkeyOffset+32])
    a.Slot = parser.Slot
    a.FileId = parser.FileId
    a.Offset = parser.Offset

    parser.Offset += hdrLen

    // ✅ Validate data length against buffer
    if parser.Offset+dataLen > bufLen {
        return fmt.Errorf("overflow: data length %d exceeds remaining buffer", dataLen)
    }

    parser.Offset += util.AlignUp(dataLen, 8)
    return nil
}
```

Pros:

- Eliminates the mismatch between FileSize and Buf length
- More robust - validates against actual data
- Makes the code self-documenting

Cons:

- Changes validation semantics
- May affect other code that relies on FileSize checks

Option 3: Enforce Precondition in `BuildIndexEntriesFromAppendVecs()`

Add validation in the caller:

```
func BuildIndexEntriesFromAppendVecs(data []byte, fileSize uint64, slot uint64, fileId uint64) ([]solana.PublicKey,
[]AccountIndexEntry, error) {
    // ✅ Enforce precondition
    if uint64(len(data)) < fileSize {
        return nil, nil, fmt.Errorf("invalid parameters: data length %d < fileSize %d", len(data), fileSize)
    }
```

```
    parser := &appendVecParser{Buf: data, FileSize: fileSize, FileId: fileId, Slot: slot}
    // ... rest unchanged
}
```

Pros:

- Validates at API boundary
- Fails fast before any parsing

Cons:

- Doesn't protect against other callers of `ParseNextAcct`
- Less defensive than Option 1

Recommendation

Use Option 1 (validate in `ParseNextAcct`):

- Validates at the right abstraction level
- Provides defensive programming within the parser
- Clear error messages
- Minimal code changes
- Protects against all callers, not just `BuildIndexEntriesFromAppendVecs`

Testing

After implementing the fix, verify:

```
# Fuzz test should complete without panics
go test ./pkg/accountsdb -fuzz=FuzzBuildIndexEntriesFromAppendVecs -fuzztime=1m

# Unit test for explicit edge case
func TestParseNextAcctBufferUnderflow(t *testing.T) {
    parser := &appendVecParser{
        Buf:      make([]byte, 50),  // Small buffer
        FileSize: 1000,              // Large FileSize
        FileId:   1,
        Slot:     100,
    }

    var pk solana.PublicKey
    var entry AccountIndexEntry

    err := parser.ParseNextAcct(&pk, &entry)
    if err == nil {
        t.Error("Expected error for buffer underflow")
    }
    if !strings.Contains(err.Error(), "overflow") && !strings.Contains(err.Error(), "underflow") {
        t.Errorf("Expected descriptive error, got: %v", err)
    }
}
```

References

- **Related code**:

- `pkg/accountsdb/appendvec.go`：`ParseNextAcct()`
- `pkg/accountsdb/index.go`：`BuildIndexEntriesFromAppendVecs()`
- `pkg/accountsdb/index_fuzz_test.go`：`FuzzBuildIndexEntriesFromAppendVecs()`

Severity

High - This is a DoS vulnerability that allows corrupted append vector files to crash the node. While less likely than malicious network input, disk corruption or filesystem issues could trigger this panic during normal operation.

[F-A12] `CheckedMulU8` and `CheckedMulU16` can return False Positive

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

Description

The `CheckedMulU8` and `CheckedMulU16` functions in `pkg/safemath/checked.go` return a false positive (incorrectly report overflow) when `b == 0`.

runtimeverification/_audits_Overclock-Validator_mithril/pkg/safemath/checked.go

Line 34 to 40 in 06f7b95

```
34 func CheckedMulU8(a, b uint8) (uint8, error) {
35     result := a * b
36     if result < a {
37         return 0, ErrOverflowMul
38     }
39     return result, nil
40 }
```

runtimeverification/_audits_Overclock-Validator_mithril/pkg/safemath/checked.go

Line 72 to 78 in 06f7b95

```
72 func CheckedMulU16(a, b uint16) (uint16, error) {
73   result := a * b
74   if result < a {
75       return 0, ErrOverflowMul
76   }
77   return result, nil
78 }
```

Scenario

- If `a > 0` and `b == 0`:
  - `result = a * 0 = 0`
  - The check if `result < a` evaluates to `true`
  - The function incorrectly returns `ErrOverflowMul` even though `a * 0 = 0` is mathematically correct and doesn't overflow

Impact

Looking at the usage in the codebase:

- `CheckedMulU8` is used in `pkg/sealevel/syscalls_log.go` for calculating memory size.
  - If `len == 0`, this would incorrectly return an error instead of allowing the syscall to proceed with zero-length data
- `CheckedMulU16` doesn't appear to be used in the retrieved code snippets, but the bug exists nonetheless.

Recommendation

- **Option 1** - Use division to verify (safe for all sizes):

```go
func CheckedMulU8(a, b uint8) (uint8, error) {
    if a == 0 || b == 0 {
        return 0, nil
    }
    result := a * b
    if result / a != b {
        return 0, ErrOverflowMul
    }
    return result, nil
}

func CheckedMulU16(a, b uint16) (uint16, error) {
    if a == 0 || b == 0 {
        return 0, nil
    }
    result := a * b
    if result / a != b {
        return 0, ErrOverflowMul
    }
    return result, nil
}
```

- **Option 2** - Use Wider Type (simple and clear):

```go
func CheckedMulU8(a, b uint8) (uint8, error) {
    result := uint16(a) * uint16(b)
    if result > math.MaxUint8 {
        return 0, ErrOverflowMul
    }
    return uint8(result), nil
}

func CheckedMulU16(a, b uint16) (uint16, error) {
    result := uint32(a) * uint32(b)
    if result > math.MaxUint16 {
        return 0, ErrOverflowMul
    }
    return uint16(result), nil
}
```

Status

Fixed in commit 45ecdb1.

[F-A13] `pkg/sealevel/syscalls_common.go` function `isNonOverlapping` incorrectly handles zero-length memory regions

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

Description

- `pkg/sealevel/syscalls_common.go` function `isNonOverlapping` incorrectly handles zero-length memory regions

```
// BUGGY CODE:
func isNonOverlapping(src, srcLen, dst, dstLen uint64) bool {
    if src > dst {
        return src-dst >= dstLen
    } else {
        return dst-src >= srcLen
    }
}
```

Issue: The function incorrectly reports that zero-length regions overlap with other regions at the same address. A zero-length region (empty region) should **never** be considered as overlapping with any other region because it occupies no memory space.

Impact:

- **Semantic violation**: Zero-length memory operations are incorrectly rejected as overlapping
- **Breaks valid use cases**: `memcpy(addr, addr, 0)` should be valid but is rejected when `src == dst`
- **Inconsistent with official Solana**: Solana's implementation correctly treats zero-length regions as non-overlapping
- Caught by fuzzing with input: `src = 70, srcLen = 111, dst = 70, dstLen = 0`

Failing Test Case:

- Input: `src = 70, srcLen = 111, dst = 70, dstLen = 0`
- Current result: `false` (incorrectly reports overlap)
- Expected result: `true` (should be non-overlapping)
- Explanation:
  - `src` range is `[70, 181)` (111 bytes)
  - `dst` range is `[70, 70)` (empty, 0 bytes)
  - These don't overlap because `dst` is empty
- Current implementation goes to `else` branch:
  - Returns `dst-src >= srcLen` → `70-70 >= 111` → `0 >= 111` → **false** ❌

Root Cause: The function doesn't handle zero-length regions as a special case. Empty regions should always be considered non-overlapping because they don't occupy any address space.

Comparison with Solana Implementation:

In Solana's `rbpf` crate (`src/elf_parser/mod.rs`), the overlap check is:

```
fn check_that_there_is_no_overlap(
    range_a: &Range<usize>,
    range_b: &Range<usize>,
) -> Result<(), ElfParserError> {
```

```
    if range_a.end <= range_b.start || range_b.end <= range_a.start {
        Ok(())
    } else {
        Err(EbpfError::Overlap)
    }
}
```

This correctly handles zero-length ranges because:

- When `range_b` has zero length: `range_b.start == range_b.end`
- The condition `range_a.end <= range_b.start` is satisfied (non-overlapping)
- Empty ranges are **never** reported as overlapping

Why Zero-Length Regions Should Be Non-Overlapping:

1. **Mathematical definition**: Two memory regions `[a, b)` and `[c, d)` overlap if and only if there exists an address `x` such that `a ≤ x < b` AND `c ≤ x < d`. For a zero-length region `[n, n)`, no such `x` exists.
2. **Practical use cases**:
   - `memcpy(addr, addr, 0)` is a no-op and should be valid
   - Zero-length reads/writes are common in syscall interfaces
   - After the overlap check in `syscalls_mem.go`, there's already `if n == 0 { return syscallSuccess(0) }`, showing that zero-length copies are expected to succeed
3. **Consistency**: The fuzz test oracle correctly expects zero-length regions to be non-overlapping:

   ```
   // Regions don't overlap if one ends before the other starts
   expectedNonOverlap := (srcEnd <= dst) || (dstEnd <= src)
   ```

   When `dstLen = 0`: `dstEnd = dst`, so `dstEnd <= src` becomes `dst <= src` which is true when `dst = src`.

Suggested Fix:

```
func isNonOverlapping(src, srcLen, dst, dstLen uint64) bool {
    // Zero-length regions never overlap
    if srcLen == 0 || dstLen == 0 {
        return true
    }

    if src > dst {
        return src-dst >= dstLen
    } else {
        return dst-src >= srcLen
    }
}
```

Alternative Fix (matching Solana's range-based approach):

```
func isNonOverlapping(src, srcLen, dst, dstLen uint64) bool {
    srcEnd := src + srcLen
    dstEnd := dst + dstLen

    // Check for overflow
    if srcEnd < src || dstEnd < dst {
        // Overflow occurred, cannot determine overlap safely
        // Conservative: assume overlapping
```

```
        return false
    }

    // Regions don't overlap if one ends before or at the point where the other starts
    // This correctly handles zero-length regions
    return srcEnd <= dst || dstEnd <= src
}
```

Testing: The bug was caught by the fuzz test `FuzzIsNonOverlapping`:

```
$ go test -run='^$'  -fuzz=FuzzIsNonOverlapping$ -fuzztime=1m ./pkg/sealevel
--- FAIL: FuzzIsNonOverlapping (1.24s)
    --- FAIL: FuzzIsNonOverlapping (0.00s)
        syscalls_mem_fuzz_test.go:68: isNonOverlapping(src=70, srcLen=111, dst=70, dstLen=0) = false, want true
```

Recommendation: Use the first fix (explicit zero-length check) as it's clearer and more efficient. The alternative fix is also correct but performs unnecessary arithmetic for the common case.

Status

Fixed in commit e1cba84.

[F-B01] `pkg/shred/shred.go` function `NewShredFromSerialized` panics on unimplemented shred formats instead of gracefully handling them

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

Description

- `pkg/shred/shred.go` function `NewShredFromSerialized` panics on unimplemented shred formats instead of gracefully handling them

```
// BUGGY CODE (line 96):
case variant == LegacyCodeID:
    //s.loadCode()
    panic("todo legacy code shred")  // ❌ PANIC on unimplemented format

// BUGGY CODE (line 125):
case variant&MerkleTypeMask == MerkleCodeID:
    panic("todo merkle code shred")  // ❌ PANIC on unimplemented format
    //return MerkleCodeFromPayload(shred)
```

Issue: The function panics when encountering code shred variants (both legacy and merkle formats) that are not yet implemented, instead of gracefully returning an empty/invalid shred. This occurs when deserializing network data or fuzzed inputs with variant bytes matching:

- `LegacyCodeID` (0x5A) - Legacy erasure coding shreds
- `MerkleCodeID` (0x40 mask) - Merkle tree-based erasure coding shreds

Impact:

- **Critical DoS vulnerability**: Malicious actors can crash validators by sending shreds with code variant bytes
- **Fuzzing failures**: All fuzz tests expect no panics but fail when code shreds are generated
- **Production instability**: Network data with unimplemented formats will crash the process
- **Incomplete implementation**: The `"todo"` comments indicate these are placeholder panics for unfinished features
- Caught by fuzzing: `FuzzShredDeserializationV2`, `FuzzShredVariantParsing`

Trigger Example:

- Craft or receive a shred with `variant = 0x5A` (LegacyCodeID) or `variant = 0x4X` (MerkleCodeID)
- Call `NewShredFromSerialized(maliciousShred, RevisionV1)`
- **Panic**: Process crashes with "todo legacy/merkle code shred"
- Validator goes offline, network participation disrupted

Root Cause: The deserialization code treats unimplemented formats as programming errors (panic) instead of malformed input (graceful failure). This violates the principle that **untrusted input must never cause panics**.

Fuzz Test Evidence:

```
$ go test -fuzz=FuzzShredDeserializationV1$ -fuzztime=1m ./pkg/shred
--- FAIL: FuzzShredDeserializationV2 (0.00s)
    --- FAIL: FuzzShredDeserializationV2/seed#3 (0.00s)
        shred_fuzz_test.go:68: NewShredFromSerialized panicked: todo merkle code shred
--- FAIL: FuzzShredVariantParsing (0.00s)
```

```
    --- FAIL: FuzzShredVariantParsing/seed#64-79 (0.00s)
        shred_fuzz_test.go:102: Unexpected panic on variant parsing: todo merkle code shred
```

Suggested Fix:

```
func NewShredFromSerialized(shred []byte, revision int) (s Shred) {
    if len(shred) < 88 {
        return
    }
    variant := shred[64]
    switch {
    case variant == LegacyCodeID:
        // ✅ Return empty shred instead of panic
        // TODO: Implement legacy code shred deserialization
        return
    case variant == LegacyDataID:
        // ... existing implementation ...
    case variant&MerkleTypeMask == MerkleCodeID:
        // ✅ Return empty shred instead of panic
        // TODO: Implement merkle code shred deserialization
        return
    case variant&MerkleTypeMask == MerkleDataID:
        // ... existing implementation ...
    default:
        // Unknown variant - return empty shred
        return
    }
    // ... rest of function ...
}
```

Alternative Fix with Logging:

```
case variant == LegacyCodeID:
    // Log the unimplemented format for debugging
    mlog.Log.Debugf("Legacy code shred not yet implemented, skipping (variant=0x%02x)", variant)
    return

case variant&MerkleTypeMask == MerkleCodeID:
    // Log the unimplemented format for debugging
    mlog.Log.Debugf("Merkle code shred not yet implemented, skipping (variant=0x%02x)", variant)
    return
```

Why This Fix Is Correct:

1. **Consistent with test expectations**: Fuzz tests explicitly check that no panics occur:

   ```
   defer func() {
       if r := recover(); r != nil {
           t.Errorf("NewShredFromSerialized panicked: %v", r)
       }
   }()
   ```

2. **Safe default behavior**: Returning an empty `Shred` (with zero-valued fields) signals parsing failure without crashing. Callers already check if a shred is valid by examining its fields.

3. **Matches existing patterns**: The function already returns empty shreds for other error conditions:

- `if len(shred) < 88 { return }`
- `if payloadSize < 0 { return }`
- `if len(shred) < int(s.DataHeader.Size) { return }`

4. **Production ready**: Once code shred implementation is added, these `return` statements can be replaced with actual deserialization logic without changing the function signature.

Testing: After fix, verify with:

```
# All fuzz tests should pass
go test -fuzz=FuzzShredDeserializationV1 -fuzztime=1m ./pkg/shred
go test -fuzz=FuzzShredDeserializationV2 -fuzztime=1m ./pkg/shred
go test -fuzz=FuzzShredVariantParsing -fuzztime=1m ./pkg/shred
```

Note on Implementation Status: Code shreds are used for erasure coding (forward error correction) in Solana. While data shreds are sufficient for basic block processing, a production validator will eventually need code shred support for handling network packet loss and participating in repair protocols.

Recommendation: Apply the suggested fix immediately to prevent DoS vulnerabilities. Then prioritize implementing code shred deserialization if full validator functionality is required.

[F-B02] `translateInternal` panic on zero-size stack access with invalid address

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

Description

- `pkg/sbpf/interpreter.go` function `translateInternal` panic on zero-size stack access with an invalid address

```
// BUGGY CODE:
case VaddrStack >> 32:
    mem := ip.stack.GetFrame(uint32(addr))
    if size > uint64(len(mem)) {
        return nil, NewExcBadAccess(addr, size, write, "out-of-bounds stack access")
    }
    return unsafe.Pointer(&mem[0]), nil  // ❌ PANIC when mem is nil or empty
```

Issue: When `GetFrame` returns `nil` (for invalid stack addresses like gaps or out-of-bounds) and `size == 0`, the bounds check passes (`0 > 0` is false), then attempting `&mem[0]` on a nil slice causes a panic: `runtime error: index out of range [0] with length 0`.

Impact:

- Runtime panic on zero-size operations to invalid stack addresses
- VM crash instead of proper error handling
- Caught by fuzzing with input: `addr = 0x2fffffff3` (invalid stack address), `size = 0`, `write = true`

Root Cause: The code attempts to dereference `mem[0]` without checking if the slice is empty. `GetFrame` returns `nil` for invalid addresses (gaps, out-of-bounds), but the bounds check `size > len(mem)` doesn't catch the case where both are 0.

Recommendation

Suggested Fix:

```
case VaddrStack >> 32:
    mem := ip.stack.GetFrame(uint32(addr))
    if size > uint64(len(mem)) {
        return nil, NewExcBadAccess(addr, size, write, "out-of-bounds stack access")
    }
    if size == 0 {
        return nil, nil  // Valid zero-size access, no pointer needed
    }
    return unsafe.Pointer(&mem[0]), nil
```

Alternative: Add explicit nil check if you want to reject invalid addresses even for zero-size operations:

```
case VaddrStack >> 32:
    mem := ip.stack.GetFrame(uint32(addr))
    if mem == nil {
        return nil, NewExcBadAccess(addr, size, write, "invalid stack address")
    }
    if size > uint64(len(mem)) {
        return nil, NewExcBadAccess(addr, size, write, "out-of-bounds stack access")
    }
```

```
    if size == 0 {
        return nil, nil
    }
    return unsafe.Pointer(&mem[0]), nil
```

Note: The suggested fix avoids the panic while still allowing zero-size operations to "succeed" (return `nil, nil`) even for invalid addresses. This is acceptable since zero-size operations don't actually access memory.

Status

Addressed in commit 28b4714. The function returns an empty slice when size is zero.

[F-B03] `pkg/sealevel/vote_state.go` function `AuthorizedVoters.PurgeAuthorizedVoters` panics when the authorized voters set becomes empty

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

Description

- `pkg/sealevel/vote_state.go` function `AuthorizedVoters.PurgeAuthorizedVoters` panics when the authorized voters set becomes empty

```
// BUGGY CODE:
for _, key := range expiredKeys {
    _, success := authVoters.AuthorizedVoters.Delete(key)
    if !success {
        panic("there was no key to remove - programming error")
    }
}

if authVoters.AuthorizedVoters.Len() == 0 {
    panic("invariant - AuthorizedVoters should not be empty")
}
return true
```

Issue: The method panics when the internal B-tree becomes empty or when a delete operation unexpectedly fails. Fuzzing produced inputs that legitimately lead to an empty `AuthorizedVoters` set, causing a panic and test/VM crash.

Impact:

- **DoS / Crash**: Malformed or adversarial vote-state inputs can crash the process
- **Incorrect invariant handling**: The code treats malformed/edge-case input as a programming error instead of returning a recoverable error

Root Cause: The code enforces a hard invariant (non-empty set) by panicking instead of returning an error/boolean to the caller. Fuzzing found valid-but-edge-case inputs that violate this invariant.

Suggested Fix:

- Prefer returning an error or a boolean result instead of panicking. Minimal, safe change:

```
for _, key := range expiredKeys {
    _, success := authVoters.AuthorizedVoters.Delete(key)
    if !success {
        // deletion failed - treat as non-fatal and continue
        return false
    }
}

if authVoters.AuthorizedVoters.Len() == 0 {
    // No voters left after purge; return false so caller can handle it
    return false
}
return true
```

- Alternatively, change the method signature to return `(bool, error)` and return a descriptive error when the set becomes empty.

Testing / Mitigation:

- Add a fuzz regression test that feeds the previously failing seed and asserts the function returns `false` rather than panicking:

```
// Reproducer: run Unmarshal path that leads to empty AuthorizedVoters and ensure PurgeAuthorizedVoters returns false
```

Recommendation: Treat input-derived state as potentially invalid and avoid panics in library code; convert hard panics to error returns so callers can decide how to proceed.

[F-B04] Function `calculateDeltaLtHash` missing division-by-zero check

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

Description

The function `pkg/bankhash/lthash.go/calculateDeltaLtHash` does not prevent a panic due to a division by zero if `numberWorkers == 0`:

runtimeverification/_audits_Overclock-Validator_mithril/pkg/bankhash/lthash.go

Line 18 to 22 in 06f7b95

```
18 func calculateDeltaLtHash(slotCtx *sealevel.SlotCtx, modifiedAccts []*accounts.Account) *lthash.LtHash {
19     numWorkers := min(32, len(modifiedAccts))
20
21     hashes := make([]*lthash.LtHash, len(modifiedAccts))
22     chunkSize := (len(modifiedAccts) + numWorkers - 1) / numWorkers
```

Note

The bug doesn't manifest in practice because `calculateDeltaLtHash` is never called with an empty `modifiedAccts` slice in the current codebase.
The function has only one caller: `updateAcctsLtHash`, which is called from `CalculateBankHash`.
The reason the division by zero doesn't occur is that `writableAccts` is passed to `CalculateBankHash`, and `writableAccts` is constructed from multiple sources in `compileWritableAndModifiedAccts`. The function always adds at least the `sysvar` accounts at the end.

runtimeverification/_audits_Overclock-Validator_mithril/pkg/replay/block.go

Line 713 to 715 in 06f7b95

```
713     sysvarAccts := collectAndUpdateSysvarAcctsForAdh(slotCtx)
714     writableAccts = append(writableAccts, sysvarAccts...)
715     modifiedAccts = append(modifiedAccts, sysvarAccts...)
```

So in practice, `writableAccts` is never empty because system variable accounts are always included, which means `calculateDeltaLtHash` is never called with an empty slice.

Recommendation

While the bug doesn't manifest in practice due to the `sysvar` accounts always being present, it's still a latent issue. If the code were refactored to skip `sysvar` updates in some cases, or if `calculateDeltaLtHash` were called from a different context, the division by zero would occur.
Adding a defensive early return check would make the code more robust:

```
func calculateDeltaLtHash(slotCtx *sealevel.SlotCtx, modifiedAccts []*accounts.Account) *lthash.LtHash {
    if len(modifiedAccts) == 0 {
        return &lthash.LtHash{}
    }

    numWorkers := min(32, len(modifiedAccts))
```

```
    // ... rest of function
}
```

Status

Client acknowledged the issue but accepts the risk on the assumption that `len(modifiedAccts)` will always be non-zero due to certain sysvars (such as clock) being updated every slot.

[F-B05] Function `partitionIdxFromSlotIdx` can overflow

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

Description

The function `pkg/rent/partition.go/partitionIdxFromSlotIdx` might cause partition index overflow in rent collection if `EpochCountPerCycle` is greater than `1`.

runtimeverification/_audits_Overclock-Validator_mithril/pkg/rent/partition.go

Line 53 to 57 in 06f7b95

```
53  func partitionIdxFromSlotIdx(slotIdxInEpoch uint64, cycleParams RentCollectionCycleParams) uint64 {
54      epochOffset := cycleParams.Epoch - cycleParams.BaseEpoch
55      epochIdxInCycle := epochOffset % cycleParams.EpochCountPerCycle
56      return slotIdxInEpoch + (epochIdxInCycle * cycleParams.SlotCountPerEpoch)
57  }
```

The overflow can occur in the return statement when computing `slotIdxInEpoch + (epochIdxInCycle * cycleParams.SlotCountPerEpoch)`. There are two potential overflow points:

- Multiplication overflow: `epochIdxInCycle * cycleParams.SlotCountPerEpoch` could overflow if both values are large enough
- Addition overflow: even if the multiplication doesn't overflow, adding `slotIdxInEpoch` to the result could cause the final sum to exceed `math.MaxUint64`

`EpochCountPerCycle` is currently hard-coded to `1`, so the bug is dormant for now.

runtimeverification/_audits_Overclock-Validator_mithril/pkg/rent/partition.go

Line 70 to 74 in 06f7b95

```
70  func partitionFromNormalSlotIndices(startSlotIdx uint64, endSlotIdx uint64, epoch uint64, epochSchedule
*sealevel.SysvarEpochSchedule) Partition {
71      slotsPerEpoch := epochSchedule.SlotsInEpoch(epoch)
72      cycleParams := RentCollectionCycleParams{Epoch: epoch, SlotCountPerEpoch: slotsPerEpoch, MultiEpochCycle: false, BaseEpoch: 0,
EpochCountPerCycle: 1, PartitionCount: slotsPerEpoch}
73      return partitionFromSlotIndices(cycleParams, startSlotIdx, endSlotIdx)
74  }
```

Recommendation

Add overflow protection.

Status

The client acknowledged the issue but accepts the risk on the assumption that the values involved will never reach a point where an overflow could occur.

[F-B06] `SaturatingPow` does not check division by 0

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

Description

The function `SaturatingPow` in `pkg/safemath/saturating.go` might panic if `n==0`

```
func SaturatingPow(n uint64, m uint32) uint64 {
    if m == 0 {
        return 1
    }

    if m == 1 {
        return n
    }

    result := n
    for i := uint32(2); i <= m; i++ {
        x := result * n
        if (x / n) != result {  // ← Line 357: DIVISION BY ZERO!
            return math.MaxUint64
        } else {
            result = x
        }
    }
    return result
}
```

Scenario

- Example input: `n = 0` and `m = 17`

Impact - This function is being used in epoch schedule calculations. In `SlotsInEpoch`, it computes `2^epoch`, so `n` is always `2`, never `0`. Similarly, in `FirstSlotInEpoch`, it also uses base `2`.

So while the bug exists, it's not currently triggered in practice because the function is only called with `n = 2` in the codebase.

Recommendation

Although this is a dormant bug, it could cause panics if `SaturatingPow` were ever called with `n == 0` and `m > 1`. The "saturating" behavior is supposed to prevent panics, but the overflow detection logic itself can panic on division by zero. A proper fix would add a check like if `n == 0`.

```
func SaturatingPow(n uint64, m uint32) uint64 {
    // Handle base cases
    if n == 0 {
        if m == 0 {
            return 1  // 0^0 = 1 by convention
        }
        return 0  // 0^m = 0 for m > 0
    }

    if m == 0 {
```

```
        return 1  // n^0 = 1 for any n != 0
    }

    if m == 1 {
        return n
    }

    if n == 1 {
        return 1  // 1^m = 1 for any m
    }

    result := n
    for i := uint32(2); i <= m; i++ {
        x := result * n
        if (x / n) != result {
            return math.MaxUint64
        }
        result = x
    }
    return result
}
```

Status

Fixed in commit 92516a9.

[F-B07] `pkg/sbpf/opcode_test.go` uses incorrect opcode values for signed division instructions

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

Description

- `pkg/sbpf/opcode_test.go` Incorrect opcode values for signed division instructions

```
// BUGGY CODE:
{0xe4, OpSdiv32Imm},
{0xe7, OpSdiv64Imm},
{0xec, OpSdiv32Reg},
{0xef, OpSdiv64Reg},
```

Issue: The test cases for signed division opcodes used incorrect hexadecimal values that don't match the actual opcode constant definitions. These values don't correspond to the correct bit patterns for the SBPF instruction encoding.

Impact:

- Test validation failure: Tests would fail when validating opcode definitions
- Incorrect instruction encoding expectations
- Could lead to wrong bytecode generation or instruction decoding
- Breaks consensus with actual SBPF specification

Root Cause: The opcode values were calculated incorrectly. The correct values must follow the SBPF encoding formula:

- `OpSdiv32Imm = ClassPqr | SrcK | PqrSdiv = 0x06 | 0x00 | 0xc0 = 0xc6`
- `OpSdiv32Reg = ClassPqr | SrcX | PqrSdiv = 0x06 | 0x08 | 0xc0 = 0xce`
- `OpSdiv64Imm = ClassPqr | 0x10 | SrcK | PqrSdiv = 0x06 | 0x10 | 0x00 | 0xc0 = 0xd6`
- `OpSdiv64Reg = ClassPqr | 0x10 | SrcX | PqrSdiv = 0x06 | 0x10 | 0x08 | 0xc0 = 0xde`

Suggested Fix:

```
// RV Fix:
{0xc6, OpSdiv32Imm},
{0xd6, OpSdiv64Imm},
{0xce, OpSdiv32Reg},
{0xde, OpSdiv64Reg},
```

Verification: The corrected values match the opcode constant definitions in `pkg/sbpf/opcode.go`:

- OpSdiv32Imm: 0xe4 ❌ → 0xc6 ✅
- OpSdiv64Imm: 0xe7 ❌ → 0xd6 ✅
- OpSdiv32Reg: 0xec ❌ → 0xce ✅
- OpSdiv64Reg: 0xef ❌ → 0xde ✅

[F-B08] `pkg/features/features.go` function `AllEnabled` displays feature gate addresses as raw bytes instead of base58-encoded strings

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

Description

- `pkg/features/features.go` function `AllEnabled` displays feature gate addresses as raw bytes instead of base58-encoded strings

```
// BUGGY CODE:
func (f *Features) AllEnabled() []string {
    enabledFeatureStrs := make([]string, 0)
    for feat, enabled := range *f {
        if enabled.Enabled {
            enabledFeatureStrs = append(enabledFeatureStrs, fmt.Sprintf("feature %s (%s) enabled", feat.Name, feat.Address))
            // ❌ %s formats [32]byte as raw bytes like "\x00\x05\xee\xe8..."
        }
    }
    return enabledFeatureStrs
}
```

Issue: The function uses `%s` format specifier with the `Address` field (type `[32]byte`), which causes Go to format it as raw binary bytes instead of the human-readable base58 string representation. This produces unreadable output like `(\x00\x05\xee\xe8\x9f\xd8Q/Q\x9c\xc1v'\xe0\xf6:9a\t␣M\x9a\xb7\x98\xa3\x01\rZ\xb8\x0f)` instead of the expected base58 string like `(16FMCmgLzCNNz6eTwGanbyN2ZxvTBSLuQ6DZhgeMshg)`.

Impact:

- **Cosmetic/usability bug**: Feature gate addresses are unreadable in logs and debugging output
- **Makes verification difficult**: Developers/operators cannot easily verify which features are enabled by comparing addresses
- **Inconsistent with input format**: Feature gates are created using base58 strings (e.g., `base58.MustDecodeFromString("16FMCmgLzCNNz6eTwGanbyN2ZxvTBSLuQ6DZhgeMshg")`), so output should match
- **Test failure**: `TestFflags_ListEnabled` expects base58 format, revealing the bug

Root Cause: The `Address` field stores the decoded binary bytes (`[32]byte`), but the formatting code doesn't encode it back to base58 for display.

Suggested Fix:

```
func (f *Features) AllEnabled() []string {
    enabledFeatureStrs := make([]string, 0)
    for feat, enabled := range *f {
        if enabled.Enabled {
            // ✅ Encode address to base58 string for display
            addressStr := base58.Encode(feat.Address[:])
            enabledFeatureStrs = append(enabledFeatureStrs,
                fmt.Sprintf("feature %s (%s) enabled", feat.Name, addressStr))
        }
    }
    return enabledFeatureStrs
}
```

Alternative Fix: Add a `String()` method to `FeatureGate` type:

```
func (fg FeatureGate) String() string {
    return fmt.Sprintf("%s (%s)", fg.Name, base58.Encode(fg.Address[:]))
}

// Then in AllEnabled():
func (f *Features) AllEnabled() []string {
    enabledFeatureStrs := make([]string, 0)
    for feat, enabled := range *f {
        if enabled.Enabled {
            enabledFeatureStrs = append(enabledFeatureStrs,
                fmt.Sprintf("feature %s enabled", feat.String()))
        }
    }
    return enabledFeatureStrs
}
```

Testing: Caught by existing unit test `TestFflags_ListEnabled`:

```
$ go test ./pkg/features
--- FAIL: TestFflags_ListEnabled (0.00s)
    features_test.go:26:
        Expected: "feature StopTruncatingStringsInSyscalls (16FMCmgLzCNNz6eTwGanbyN2ZxvTBSLuQ6DZhgeMshg) enabled"
        Actual:   "feature StopTruncatingStringsInSyscalls (\x00\x05\xee\xe8...) enabled"
```

Recommendation: Use the first fix (direct encoding in `AllEnabled()`) as it's simpler and only affects the display function. The `String()` method approach would be beneficial if other parts of the codebase also need to display `FeatureGate` values in a human-readable format.

[F-B09] `translateInstructionC` and `translateInstructionRust` do not make defensive copy of instruction data

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

Description

- `pkg/sealevel/syscalls_cpi.go` functions `translateInstructionC` and `translateInstructionRust` do not make defensive copy of instruction data

```
// BUGGY CODE in translateInstructionC:
data, err := vm.Translate(ix.DataAddr, ix.DataLen, false)
if err != nil {
    return Instruction{}, err
}
// ...
return Instruction{Accounts: accounts, Data: data, ProgramId: programId}, nil
// ❌ data slice shares backing array with VM memory

// BUGGY CODE in translateInstructionRust:
data, err := vm.Translate(ix.DataAddr, ix.DataLen, false)
if err != nil {
    return Instruction{}, err
}
// ...
return Instruction{Accounts: accountMetas, Data: data, ProgramId: ix.Pubkey}, nil
// ❌ data slice shares backing array with VM memory
```

Issue: Both functions create `Instruction` structs with a `Data` field that directly references the slice returned from `vm.Translate()`. This means the instruction data **shares the same backing array** as the VM memory region, creating an alias that can lead to unintended side effects.

Impact:

- **Data corruption risk**: If instruction data is modified after creation, it would corrupt the original VM memory
- **Unexpected aliasing**: Multiple instructions pointing to overlapping VM memory would have shared data
- **Race condition potential**: Concurrent access to instruction data and VM memory could cause races
- **Semantic violation**: Instruction data should be independent of its source after parsing
- **Caught by fuzzing**: `FuzzInstructionDataParsing` detected shared backing array

Trigger Example:

```
// Create instruction from VM memory
instr, _ := translateInstructionC(vm, ...)

// Modify instruction data
instr.Data[0] = 0xFF

// VM memory is also modified! (unexpected side effect)
// If another instruction references same memory region, it sees the change
```

Root Cause: Direct slice assignment in Go shares the backing array. The code doesn't make a defensive copy of the data slice before assigning it to the `Instruction` struct.

Suggested Fix:

```
// For translateInstructionC (around line 166):
func translateInstructionC(vm sbpf.VM, addr uint64) (Instruction, error) {
    // ... existing code to translate and read instruction ...

    data, err := vm.Translate(ix.DataAddr, ix.DataLen, false)
    if err != nil {
        return Instruction{}, err
    }

    // ✅ Make defensive copy of data
    dataCopy := make([]byte, len(data))
    copy(dataCopy, data)

    // ... process accounts ...

    return Instruction{Accounts: accounts, Data: dataCopy, ProgramId: programId}, nil
}

// For translateInstructionRust (around line 240):
func translateInstructionRust(vm sbpf.VM, addr uint64) (Instruction, error) {
    // ... existing code to translate and read instruction ...

    data, err := vm.Translate(ix.DataAddr, ix.DataLen, false)
    if err != nil {
        return Instruction{}, err
    }

    // ✅ Make defensive copy of data
    dataCopy := make([]byte, len(data))
    copy(dataCopy, data)

    // ... process accounts ...

    return Instruction{Accounts: accountMetas, Data: dataCopy, ProgramId: ix.Pubkey}, nil
}
```

Alternative Optimization: If performance is critical and you can guarantee the data won't be modified, document this invariant clearly:

```
// IMPORTANT: The returned Instruction.Data slice shares its backing array with VM memory.
// DO NOT MODIFY the Data field, as it will corrupt VM memory.
// This is an optimization to avoid copying large instruction payloads.
func translateInstructionC(vm sbpf.VM, addr uint64) (Instruction, error) {
    // ... existing code ...
    return Instruction{Accounts: accounts, Data: data, ProgramId: programId}, nil
}
```

Trade-offs:

- **Defensive copy (recommended)**:
  - Pros: Safe, prevents unexpected aliasing, follows Go best practices

- Cons: Small performance cost (one extra allocation + copy per instruction)

- **No copy (current)**:
  - Pros: Slightly faster (avoids allocation)
  - Cons: Dangerous aliasing, violates principle of least surprise, easy to introduce bugs

Comparison with Solana: In Rust's Solana implementation, instruction data is owned by the instruction struct (no shared references by default), so this aliasing issue doesn't occur. The defensive copy brings the Go implementation closer to Rust's safety guarantees.

Testing: Caught by `FuzzInstructionDataParsing`:

```
$ go test -run='^$' -fuzz=FuzzInstructionDataParsing$ -fuzztime=1m ./pkg/sealevel
--- FAIL: FuzzInstructionDataParsing (0.23s)
    instruction_fuzz_test.go:451: Instruction data shares backing array with input (no defensive copy)
```

Recommendation: Implement the defensive copy. The performance cost is negligible compared to the safety benefit, and it prevents subtle bugs. If profiling shows this is a bottleneck, it can be optimized later with clear documentation of the aliasing behavior.

[F-D01] Missing Divide-by-Zero Check in OpUrem32Reg

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

Summary

The SBPF interpreter's `OpUrem32Reg` instruction handler is missing a divide-by-zero check, causing a Go runtime panic instead of returning the proper `ExcDivideByZero` error when the source register contains zero.

Root Cause

In `pkg/sbpf/interpreter.go`, the `OpUrem32Reg` case directly performs the modulo operation without first checking if the divisor (source register value) is zero:

```
case OpUrem32Reg:
    if !ip.sbpfVersion.EnablePqr() {
        err = ExcInvalidInstr
        break
    }
    r[ins.Dst()] = uint64(uint32(r[ins.Dst()]) % uint32(r[ins.Src()]))  // No zero check!
    pc++
```

This causes a Go runtime panic (`integer divide by zero`) when `r[ins.Src()]` is zero.

Other similar register-based division/modulo operations correctly check for zero before performing the operation. For example, `OpUrem64Reg`:

```
case OpUrem64Reg:
    if !ip.sbpfVersion.EnablePqr() {
        err = ExcInvalidInstr
        break
    }
    if src := r[ins.Src()]; src != 0 {
        r[ins.Dst()] %= r[ins.Src()]
    } else {
        err = ExcDivideByZero
    }
    pc++
```

Impact

- **Runtime Panic**: The missing check causes a Go runtime panic that crashes the entire process
- **Differential Fuzzing Failure**: Agave correctly returns `DivideByZero` error (error code 5), while Mithril panics
- **Denial of Service**: A malicious program could intentionally trigger this panic to crash the validator

Fix

Modified `pkg/sbpf/interpreter.go` to add the divide-by-zero check:

```
case OpUrem32Reg:
    if !ip.sbpfVersion.EnablePqr() {
        err = ExcInvalidInstr
        break
```

```
    }
    // === START RV FIX ===
    if src := uint32(r[ins.Src()]); src != 0 {
        r[ins.Dst()] = uint64(uint32(r[ins.Dst()]) % src)
    } else {
        err = ExcDivideByZero
    }
    // === END RV FIX ===
    pc++
```

Files Modified

- `pkg/sbpf/interpreter.go` (modified)

Status

The client addressed this issue in pull request https://github.com/Overclock-Validator/mithril/pull/167. However, we reviewed the fix and find it to be incomplete.

Specifically, the clients fix is to change the affected code to:

```
if src := r[ins.Src()]; src != 0 {
    r[ins.Dst()] = uint64(r[ins.Dst()] % src)
} else {
    err = ExcDivideByZero
}
```

This fix is lacking the necessary uint32 cast of the uint64 register `r[ins.Src()]` in both the comparsion and modulo operation.

[F-D02] Missing Blockhash Validation in Transaction Sanitization

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

Summary

Mithril does not validate that a transaction's recent blockhash exists in the blockhash queue during transaction sanitization. This allows transactions with invalid or expired blockhashes to proceed to execution, diverging from Agave's behavior which rejects such transactions during the `verify_transaction` phase with a sanitization error.

Root Cause

In Agave, the `Bank::verify_transaction` method performs comprehensive transaction sanitization before execution, including validating that the transaction's `recent_blockhash` exists in the blockhash queue. Agave's validation flow:

```
// Agave's txn_fuzzer.rs (lines 469-492)
let sanitized_transaction = match bank.verify_transaction(
    versioned_transaction,
    TransactionVerificationMode::HashAndVerifyPrecompiles,
) {
    Ok(v) => v,
    Err(e) => {
        // Returns sanitization_error: true, status: 8
        return Some(TxnResult {
            sanitization_error: true,
            status: 8,  // SanitizeFailure
            ...
        });
    }
};
```

Mithril's transaction processing flow lacks this blockhash validation step:

1. `ProcessTransaction` in `pkg/replay/transaction.go` calls `instrsAndAcctMetasFromTx`
2. `instrsAndAcctMetasFromTx` performs some validation (instruction duplicates, program ID checks) but **not blockhash validation**
3. No blockhash queue is maintained or checked during transaction processing

Differential Behavior

Test Case: Transaction with recent_blockhash not in blockhash queue

- Crash file: `crash-457dc623c16fc0912ac81ba3062eab8c729d3a33`
- Transaction blockhash: `30c7d6b2d53ffb089e1c4dab8d4ce8d5856f50bcec d369146c4a8fc5050453)`
- Blockhash queue contains: 4 different hashes (none matching transaction blockhash)

Agave Behavior:

```
OUTPUT 0 ======================
sanitization_error: true
status: 8
```

Mithril Behavior (before fix):

```
OUTPUT 1 ======================
executed: true
executed_units: 1050
is_ok: true
```

Mithril proceeds to execute the transaction and successfully completes it, while Agave correctly rejects it during sanitization.

Impact

Severity: Medium

This validation gap has several implications:

1. **Consensus Divergence**: Mithril may accept and execute transactions that Agave would reject, potentially leading to state divergence in a production validator environment.
2. **Replay Protection**: Blockhash validation is part of Solana's replay protection mechanism. Without it, transactions with expired or invalid blockhashes could be processed.
3. **Resource Waste**: Invalid transactions should be rejected early during sanitization to avoid wasting computational resources on execution.
4. **Differential Fuzzing Impact**: This causes false positives in differential fuzzing where Mithril's successful execution diverges from Agave's sanitization rejection.

Recommended Fix

The fix leverages Mithril's existing blockhash validation infrastructure.

Implementation in `pkg/replay/transaction.go`

1. Add Error Constant

```go
const (
    TxErrInvalidBlockhash = "TxErrInvalidBlockhash"
)
```

2. Creat Blockhash Validation Function

```go
// === START RV FIX [F-D04] ===
// ValidateTransactionBlockhash checks if the transaction's recent blockhash
// exists in the recent blockhashes sysvar using Mithril's existing validation.
// Returns an error if the blockhash is not found (sanitization failure).
func ValidateTransactionBlockhash(tx *solana.Transaction, execCtx *sealevel.ExecutionCtx) error {
    // Use Mithril's existing function to read recent blockhashes sysvar
    // This reads from either SysvarCache or the accounts database
    recentBlockhashes, err := sealevel.ReadRecentBlockHashesSysvar(execCtx)
    if err != nil {
        // If sysvar is not available, skip validation for backward compatibility
        // In production, this sysvar should always be available
        return nil
    }

    // Use Mithril's existing validation function
    if !recentBlockhashes.IsBlockhashAgeValid(tx.Message.RecentBlockhash) {
        return fmt.Errorf(TxErrInvalidBlockhash)
    }
```

```
    return nil
}
// === END RV FIX [F-D04] ===
```

3. Integrate into ProcessTransaction

Modify `ProcessTransaction` to validate blockhash early in the transaction processing flow:

```
func ProcessTransaction(slotCtx *sealevel.SlotCtx, ...) (*fees.TxFeeInfo, error) {
    // ... existing code to create ExecutionCtx ...

    // === START RV FIX [F-D04] ===
    // Validate blockhash before processing transaction
    if err := ValidateTransactionBlockhash(tx, execCtx); err != nil {
        return nil, err
    }
    // === END RV FIX [F-D04] ===

    instrs, acctMetasPerInstr, err := instrsAndAcctMetasFromTx(tx, slotCtx.Features)
    // ... rest of function ...
}
```

This implementation:

- ✅ Reuses Mithril's existing `ReadRecentBlockHashesSysvar` function
- ✅ Leverages Mithril's existing `IsBlockhashAgeValid` validation logic
- ✅ Works with Mithril's SysvarCache for performance
- ✅ Maintains consistency with how other sysvars are handled
- ✅ Validates blockhash early in transaction processing, before instruction parsing
- ✅ Returns error that can be propagated up the call stack
- ✅ Backward compatible (skips validation if sysvar not available)

Fuzzing Harness Implementation

For the solfuzz integration, the validation is also implemented in `pkg/solfuzz/txn_execute.go` . Since the fuzzing harness uses a separate execution path that doesn't call `ProcessTransaction` , it implements blockhash validation directly:

```
// In executeTransaction function
// === START RV FIX [F-D04] ===
if len(txnCtx.BlockhashQueue) > 0 {
    // Build RecentBlockhashes sysvar from blockhash queue
    var recentBlockhashes sealevel.SysvarRecentBlockhashes
    for _, queueHash := range txnCtx.BlockhashQueue {
        if len(queueHash) == 32 {
            var hashArray [32]byte
            copy(hashArray[:], queueHash)
            recentBlockhashes = append(recentBlockhashes, sealevel.RecentBlockHashesEntry{
                Blockhash:     hashArray,
                FeeCalculator: sealevel.FeeCalculator{LamportsPerSignature: 5000},
            })
        }
    }

    // Populate SysvarCache with blockhash queue for later use by instructions
```

```
    sealevel.SysvarCache.RecentBlockHashes.Sysvar = &recentBlockhashes

    // Validate transaction blockhash using Mithril's existing validation function
    if !recentBlockhashes.IsBlockhashAgeValid(solanaTx.Message.RecentBlockhash) {
        // Blockhash not found in queue - sanitization failure
        return &txn.TxnResult{
            Executed:          false,
            SanitizationError: true,
            Status:            8, // SanitizeFailure
            // ... rest of error result
        }
    }
}
// === END RV FIX [F-D04] ===
```

Why the fuzzing harness needs separate implementation:

The fuzzing harness doesn't provide the RecentBlockhashes sysvar account in the accounts database - it only provides the raw blockhash queue as `[][]byte`. Therefore, it needs to:

1. Build the `SysvarRecentBlockhashes` structure from the fuzzer's input
2. Populate `SysvarCache` with it
3. Use `IsBlockhashAgeValid()` for validation

This maintains the same validation logic as the client code while adapting to the fuzzing environment.

References

- **Agave Implementation**: `solfuzz-agave/src/txn_fuzzer.rs` lines 469-492 (`bank.verify_transaction`)
- **Solana Blockhash Queue**: Maintains last ~150 blockhashes for transaction validation
- **Error Code**: Status 8 = `SanitizeFailure` (generic transaction sanitization error)

Status
Fixed in commit 3d1b77b.

[F-D03] Incorrect Exception Handling in VM Interpreter

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

Summary

Mithril's sBPF VM interpreter has two critical bugs in its exception handling path that cause divergence from Agave's behavior:

1. **Incorrect CU consumption reporting**: Returns `0` for compute units consumed when a program fails, instead of the actual CU consumed before the exception
2. **Incorrect exception PC (program counter)**: Reports the wrong instruction address when an exception occurs

Both bugs are in the `Run()` method of `pkg/sbpf/interpreter.go`.

Root Cause

The client's `Run()` function has incomplete exception handling logic:

```
if err != nil {
    exc := &Exception{
        PC:     pc,  // BUG 1: Wrong PC - doesn't account for pc++ already executed
        Detail: fmt.Errorf("tx: %s, programId: %s - %s:", ip.txSignature, ip.programId, err),
    }
    if IsLongIns(ins.Op()) {
        exc.PC-- // Only corrects for long instruction's extra slot
    }

    return 0, 0, exc  // BUG 2: Returns 0 for cuConsumed instead of actual consumption
}
```

Bug 1: Wrong Program Counter in Exceptions

The interpreter advances `pc` INSIDE each instruction handler (e.g., `pc++` at the end of the case). When an error occurs, `pc` has often already been modified, but the fix must distinguish between cases where PC was modified vs not modified.

Cases where PC is NOT modified (should NOT subtract 1):

- `ExcInvalidInstr` with `break` (version check failures use `break` which skips `pc++`)
- `OpCall` with `ExcCallDepth` (internal call, stack.Push failed)
- `OpCall` with `ExcCallDest` (unknown function destination)
- `OpCallx` with `ExcCallDepth` (stack.Push failed)

Cases where PC IS modified (should subtract 1):

- Most instructions with errors (they do `pc++` before error is detected)
- `OpCall` syscall with error (syscall path does `pc++`)
- `OpCallx` with `ExcBadAccess` (sets `pc = target` before bounds check)

Bug 2: Missing CU Consumption Tracking

When an exception occurs, the function returns `0` for `cuConsumed`. Programs should be charged for CU even when they fail.

Differential Behavior

Test Case: `crash-9dc629324ba48fd8c6981b998a238ef64dacce2e`

Field	Agave	Mithril (before fix)
error	13	13
cu_avail	14964	15000
pc	35	36

Impact

Severity: Medium-High

1. **Resource Accounting**: Programs not charged for CU when they fail - potential DoS vector
2. **Consensus Risk**: Different CU consumption could cause blockchain state divergence
3. **Debugging**: Stack traces point to wrong instruction

Recommended Fix

Apply the following changes to `pkg/sbpf/interpreter.go` in the `Run()` method:

```
if err != nil {
    // === START RV FIX ===
    // Calculate CU consumed before returning exception
    cuConsumed = ip.initialInstrMeter - ip.computeMeter.Remaining()

    // Determine the correct exception PC.
    // For most instructions, pc is advanced (pc++) inside the handler,
    // so we need to subtract 1 to get the original instruction PC.
    //
    // However, for certain errors, pc is NOT modified:
    // - ExcInvalidInstr with break (version check failures, etc.)
    // - OpCall with ExcCallDepth (internal call, stack.Push failed)
    // - OpCall with ExcCallDest (unknown function destination)
    // - OpCallx with ExcCallDepth (stack.Push failed)
    exceptPC := pc
    _, isCallDestError := err.(ExcCallDest)
    isCallDepthError := err == ExcCallDepth
    isInvalidInstrError := err == ExcInvalidInstr

    // PC was NOT modified in these specific cases
    pcNotModified := false
    if isInvalidInstrError {
        // ExcInvalidInstr errors use 'break' which skips pc++
        pcNotModified = true
    } else if ins.Op() == OpCall && (isCallDepthError || isCallDestError) {
        pcNotModified = true
    } else if ins.Op() == OpCallx && isCallDepthError {
        pcNotModified = true
    }

    if !pcNotModified {
        // PC was modified, so revert the pc increment
        exceptPC = pc - 1
        if IsLongIns(ins.Op()) {
            exceptPC-- // Long instructions take 2 slots
        }
    }
```

```
    exc := &Exception{
        PC:     exceptPC,
        Detail: fmt.Errorf("tx: %s, programId: %s - %s", ip.txSignature, ip.programId, err),
    }

    return 0, cuConsumed, exc
    // === END RV FIX ===
}
```

Testing

```
cd ~/Workspace/_audits_Overclock-Validator_mithril
make -f Makefile.solfuzz

cd ~/Workspace/solfuzz
./build/fuzz_sol_vm_interp ./crash-9dc629324ba48fd8c6981b998a238ef64dacce2e
```

Both implementations should report matching error code, CU consumed, and PC.

Files Affected

- `pkg/sbpf/interpreter.go` - `Run()` method

[F-D04] Missing Fee Payer Validation Before Transaction Execution

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

Summary

Mithril is missing fee payer validation that Agave performs before transaction execution. This causes multiple divergences:

1. **Missing fee payer validation**: The client does not validate fee payer funds and rent state transitions during account loading. Agave validates this in `validate_fee_payer` BEFORE execution.
2. **Incorrect fee calculation**: Fee payer validation must use the **full transaction fee** (base + priority fees), not just the base fee.
3. **Rent state validation timing**: The client validates rent state transitions via `VerifyRentStateChanges()` AFTER instruction execution. Agave validates fee payer rent state BEFORE execution.

These issues cause Mithril to either execute transactions that Agave rejects, or return incorrect error codes.

Details

Root Cause

The original client code in `loadAndValidateTxAccts` did **not** perform any fee payer validation:

```
// Original client code - NO fee payer validation
func loadAndValidateTxAccts(slotCtx *sealevel.SlotCtx, acctMetasPerInstr [][]sealevel.AccountMeta, tx *solana.Transaction, instrs
[]sealevel.Instruction, instrsAcct *accounts.Account, loadedAcctBytesLimit uint32) (*sealevel.TransactionAccounts, error) {
    txAcctMetas, err := tx.AccountMetaList()
    if err != nil {
        return nil, err
    }
    // ... account loading logic ...
    // No fee payer validation here - validation only happens post-execution
}
```

The divergence is caused by:

1. **No fee payer validation** during account loading phase
2. **Function signature limitation** - only `loadedAcctBytesLimit` was passed, not the full `computeBudgetLimits` needed for priority fee calculation
3. **Rent sysvar not available** during account loading (loaded later in `ProcessTransaction`)

Expected Behavior (Agave)

Agave's transaction validation flow:

1. Load fee payer account
2. Load rent sysvar
3. Calculate **full transaction fee** (base + priority)
4. **Validate fee payer** - check funds sufficiency AND rent state transition - BEFORE execution
5. Execute instructions
6. Validate rent state changes for all accounts - AFTER execution

From `svm/src/account_loader.rs` :

```
/// Check whether the payer_account is capable of paying the fee. The
/// side effect is to subtract the fee amount from the payer_account
/// balance of lamports. If the payer_acount is not able to pay the
/// fee, the error_metrics is incremented, and a specific error is
/// returned.
pub fn validate_fee_payer(
    payer_address: &Pubkey,
    payer_account: &mut AccountSharedData,
    payer_index: IndexOfAccount,
    error_metrics: &mut TransactionErrorMetrics,
    rent: &Rent,
    fee: u64,
) -> Result<()> {
    if payer_account.lamports() == 0 {
        error_metrics.account_not_found += 1;
        return Err(TransactionError::AccountNotFound);
    }
    let system_account_kind = get_system_account_kind(payer_account).ok_or_else(|| {
        error_metrics.invalid_account_for_fee += 1;
        TransactionError::InvalidAccountForFee
    })?;
    let min_balance = match system_account_kind {
        SystemAccountKind::System => 0,
        SystemAccountKind::Nonce => {
            // Should we ever allow a fees charge to zero a nonce account's
            // balance. The state MUST be set to uninitialized in that case
            rent.minimum_balance(NonceState::size())
        }
    };

    payer_account
        .lamports()
        .checked_sub(min_balance)
        .and_then(|v| v.checked_sub(fee))
        .ok_or_else(|| {
            error_metrics.insufficient_funds += 1;
            TransactionError::InsufficientFundsForFee
        })?;

    let payer_pre_rent_state = get_account_rent_state(rent, payer_account);
    payer_account
        .checked_sub_lamports(fee)
        .map_err(|_| TransactionError::InsufficientFundsForFee)?;

    let payer_post_rent_state = get_account_rent_state(rent, payer_account);
    check_rent_state_with_account(
        &payer_pre_rent_state,
        &payer_post_rent_state,
        payer_address,
        payer_index,
    )
}
```

Key points from Agave's implementation:

1. **Zero lamports check** → `TransactionError::AccountNotFound`
2. **System account kind check** → Must be a system or nonce account, otherwise `TransactionError::InvalidAccountForFee`
3. **Nonce account min_balance** → Nonce accounts (80 bytes) must maintain rent-exemption after fee deduction
4. **Insufficient funds check** → `lamports - min_balance - fee` must not underflow, otherwise `TransactionError::InsufficientFundsForFee`
5. **Rent state transition check** → After fee deduction, validate rent state doesn't go from exempt to paying

The fee passed to `validate_fee_payer` includes:

1. Base fee: `numSignatures * lamports_per_signature` (typically 5000 per signature)
2. Priority fee: `(compute_unit_price * compute_unit_limit + 999999) / 1000000`

Actual Behavior (Mithril - Before Fix)

Mithril's flow:

1. `loadAndValidateTxAccts()` - loads accounts, NO fee payer validation
2. `ProcessTransaction()` - creates execution context
3. `ReadRentSysvar(execCtx)` - loads rent sysvar (too late for fee payer validation)
4. Execute instructions
5. `VerifyRentStateChanges()` - validates rent transitions AFTER execution

The client HAS the correct rent validation logic in `checkRentStateTransition()`, but it's only called post-execution, not during fee payer validation.

Crash File Evidence

Crash 1: Rent State Validation Timing

Testing with `crash-ad8559d4c9e4a370802b352183f33aa0206b9a00`:

Before Fix:

```
Agave: InsufficientFundsForRent { account_index: 0 }, status=32, not executed
Mithril: Transaction executed (rent validation happened too late)
Assertion failed: transaction execution boolean differs
```

After Fix:

```
Agave: InsufficientFundsForRent { account_index: 0 }, status=32, not executed
Mithril: ValidateFeePayer failed: TxErrInsufficientFundsForRent, status=32, not executed
✓ Both reject during validation phase before execution
```

Crash 2: Priority Fee Calculation

Testing with `crash-d20d4ea80620470dc7b60768c9e02f1d948bd516`:

Transaction Details:

- 4 instructions including ComputeBudget `SetComputeUnitPrice` (10,000,000 micro-lamports) and `SetComputeUnitLimit` (400,000 CUs)
- Fee payer lamports: 890,880
- 1 signature

Fee Calculation:

- Base fee: 1 × 5,000 = 5,000 lamports

- Priority fee: (10,000,000 × 400,000 + 999,999) / 1,000,000 = 4,000,000 lamports
- **Total fee: 4,005,000 lamports**

Before Fix (with only base fee):

```
DEBUG [Mithril]: ValidateFeePayer - fee payer lamports: 890880, fee: 5000
DEBUG [Mithril]: ValidateFeePayer - rent transition check failed
Assertion failed: status Expected: 5 Actual: 32
```

After Fix (with full fee):

```
DEBUG [Mithril]: ValidateFeePayer - fee payer lamports: 890880, fee: 4005000
DEBUG [Mithril]: ValidateFeePayer - insufficient lamports: 890880 < 4005000
✓ Both return status 5 (InsufficientFundsForFee)
```

Impact

Severity: High
Likelihood: High

This is a consensus divergence with multiple manifestations:

1. **Incorrect Transaction Execution**: Mithril executes transactions that Agave rejects during pre-execution validation
2. **Incorrect Error Codes**: Wrong error type returned (rent vs fee) when priority fees are involved
3. **Consensus Failure**: Validators running Mithril would accept blocks that Agave validators reject
4. **Economic Impact**: Fee payer accounts could lose rent exemption unintentionally

---

Reproduction

Test 1 (rent validation timing):

1. Create a transaction where fee payer has lamports exactly at rent exemption threshold
2. Set transaction fee such that fee payer would fall below threshold after payment
3. Agave rejects during `validate_fee_payer` with `InsufficientFundsForRent`
4. Mithril (before fix) executes the transaction

Test file: `crash-ad8559d4c9e4a370802b352183f33aa0206b9a00`

Test 2 (priority fee calculation):

1. Create a transaction with high priority fee via ComputeBudget instructions
2. Fee payer has enough for base fee but not full fee
3. Agave rejects with `InsufficientFundsForFee`
4. Mithril (before fix) returns `InsufficientFundsForRent`

Test file: `crash-d20d4ea80620470dc7b60768c9e02f1d948bd516`

Recommended Fix

Fix 1: Update Function Signature for Full Fee Calculation

File: `pkg/replay/accounts.go`

Issue: The original function signature only accepted `loadedAcctBytesLimit`, not the full `computeBudgetLimits` needed for priority fee calculation.

Original client code:

```
func loadAndValidateTxAccts(slotCtx *sealevel.SlotCtx, acctMetasPerInstr [][]sealevel.AccountMeta, tx *solana.Transaction, instrs
[]sealevel.Instruction, instrsAcct *accounts.Account, loadedAcctBytesLimit uint32) (*sealevel.TransactionAccounts, error) {
```

Fixed code:

```
// === START RV FIX ===
func loadAndValidateTxAccts(slotCtx *sealevel.SlotCtx, acctMetasPerInstr [][]sealevel.AccountMeta, tx *solana.Transaction, instrs
[]sealevel.Instruction, instrsAcct *accounts.Account, computeBudgetLimits *sealevel.ComputeBudgetLimits)
(*sealevel.TransactionAccounts, error) {
// === END RV FIX ===
```

Fix 2: Load Rent Sysvar During Account Loading

File: `pkg/replay/accounts.go` (inside `loadAndValidateTxAccts`)

Issue: The client loads rent sysvar in `ProcessTransaction()` via `ReadRentSysvar(execCtx)`, but this happens AFTER account loading completes.

Fix: Load rent sysvar directly from `slotCtx.Accounts` during `loadAndValidateTxAccts()`, using the same pattern as `ReadRentSysvar()` (panic if not found):

```
// === START RV FIX ===
// Get rent sysvar for rent state validation
// This matches ReadRentSysvar behavior - panic if rent sysvar is not found
// In a properly initialized runtime, rent sysvar should always exist
rentAddr := sealevel.SysvarRentAddr
rentAcct, err := slotCtx.Accounts.GetAccount(&rentAddr)
if err != nil {
    panic("failed to read rent sysvar account")
}

decoder := bin.NewBinDecoder(rentAcct.Data)
var rent sealevel.SysvarRent
err = rent.UnmarshalWithDecoder(decoder)
if err != nil {
    panic(fmt.Sprintf("failed to deserialize rent sysvar: %v", err))
}
// === END RV FIX ===
```

Fix 3: Calculate Full Fee and Validate Fee Payer

File: `pkg/replay/accounts.go` (inside `loadAndValidateTxAccts`, after loading fee payer)

Fix: Calculate full transaction fee (including priority) and validate fee payer:

```
// === START RV FIX ===
// Calculate full transaction fee including priority fee
// This matches Agave's fee calculation in validate_fee_payer
txFeeInfo := fees.CalculateTxFees(tx, nil, instrs, computeBudgetLimits, slotCtx.Features)
totalTxFee := txFeeInfo.TotalFee

// Validate fee payer
err = ValidateFeePayer(feePayerAcct, &rent, totalTxFee)
if err != nil {
    return nil, err
}
// === END RV FIX ===
```

Fix 4: Add ValidateFeePayer Function

File: `pkg/replay/accounts.go` (new function)

Fix: Add `ValidateFeePayer()` that matches Agave's `validate_fee_payer` logic:

```
// === START RV FIX ===
// NonceStateSize is the size of a nonce account's state (80 bytes)
// This matches Agave's NonceState::size()
// solana-sdk/nonce/src/state.rs
const NonceStateSize = 80

// ValidateFeePayer validates that the fee payer can afford the transaction fee
// while maintaining valid rent state. This matches Agave's validate_fee_payer().
//
// NOTE: Unlike Agave's validate_fee_payer which deducts fees during validation,
// Mithril separates validation from fee deduction. The actual fee deduction happens
// later in pkg/fees/fees.go:CalculateAndDeductTxFees. This separation is cleaner
// as it keeps validation pure (no side effects).
func ValidateFeePayer(
    feePayerAcct *accounts.Account,
    rentSysvar *sealevel.SysvarRent,
    fee uint64,
) error {
    // Check zero lamports - matches Agave's first check
    if feePayerAcct.Lamports == 0 {
        return TxErrAccountNotFound
    }

    // Determine system account kind - must be system or nonce account
    // Matches Agave's get_system_account_kind() check
    accountKind, err := getSystemAccountKind(feePayerAcct)
    if err != nil {
        return TxErrInvalidAccountForFee
    }

    // Determine minimum balance based on account type
    // System accounts can go to zero, but nonce accounts must maintain rent-exemption
    // Matches Agave's min_balance calculation
    var minBalance uint64
    switch accountKind {
    case SystemAccountKindSystem:
        minBalance = 0
    case SystemAccountKindNonce:
        // Nonce accounts must maintain rent-exemption for nonce state (80 bytes)
        minBalance = rentSysvar.MinimumBalance(NonceStateSize)
    }

    // Check if account has sufficient funds for fee + minimum balance
    // This uses checked arithmetic to prevent underflow, matching Agave's:
    //   payer_account.lamports().checked_sub(min_balance).and_then(|v| v.checked_sub(fee))
    available := feePayerAcct.Lamports
    if available < minBalance {
        return TxErrInsufficientFundsForFee
    }
    available -= minBalance
    if available < fee {
        return TxErrInsufficientFundsForFee
```

```
    }

    // Use client's existing rent state logic for rent state transition check
    preRentState := rent.RentStateFromAcct(feePayerAcct, rentSysvar)

    originalLamports := feePayerAcct.Lamports
    feePayerAcct.Lamports -= fee
    postRentState := rent.RentStateFromAcct(feePayerAcct, rentSysvar)
    feePayerAcct.Lamports = originalLamports  // Restore (validation only)

    // Use client's existing transition validation logic
    if err := rent.CheckRentStateTransition(preRentState, postRentState, feePayerAcct.Key); err != nil {
        return TxErrInsufficientFundsForRent
    }

    return nil
}

// SystemAccountKind represents the type of system account
type SystemAccountKind int

const (
    SystemAccountKindSystem SystemAccountKind = iota
    SystemAccountKindNonce
)

// getSystemAccountKind determines if an account is a system or nonce account
// This matches Agave's get_system_account_kind() in solana-nonce-account crate
func getSystemAccountKind(acct *accounts.Account) (SystemAccountKind, error) {
    // Check if owned by system program
    if acct.Owner != solana.SystemProgramID {
        return 0, TxErrInvalidAccountForFee
    }

    // Empty data means it's a regular system account
    if len(acct.Data) == 0 {
        return SystemAccountKindSystem, nil
    }

    // Check if it's a nonce account by examining the data
    // Nonce accounts have NonceStateSize (80) bytes of data
    if len(acct.Data) == NonceStateSize {
        // TODO: Could also verify the nonce state is initialized
        // by checking the first 4 bytes for the version number
        return SystemAccountKindNonce, nil
    }

    return 0, TxErrInvalidAccountForFee
}
// === END RV FIX ===
```

Key points:

1. **Zero lamports check** → Returns `TxErrAccountNotFound` (matches Agave)
2. **System account kind check** → Uses `getSystemAccountKind()` to verify fee payer is system or nonce account
3. **Nonce account min_balance** → Nonce accounts (80 bytes) must maintain rent-exemption after fee deduction

4. **Checked arithmetic** → Uses checked subtraction to prevent underflow, matching Agave's behavior
5. **Rent state transition** → Uses client's existing `rent.RentStateFromAcct()` and `rent.CheckRentStateTransition()`

Fix 5: Update LoadedAccountBytes References

File: `pkg/replay/accounts.go`

Replace `loadedAcctBytesLimit` with `computeBudgetLimits.LoadedAccountBytes`:

Original client code:

```
if loadedBytesAccumulator > loadedAcctBytesLimit {
    return nil, TxErrMaxLoadedAccountsDataSizeExceeded
}
```

Fixed code:

```
// === START RV FIX ===
if loadedBytesAccumulator > computeBudgetLimits.LoadedAccountBytes {
    return nil, TxErrMaxLoadedAccountsDataSizeExceeded
}
// === END RV FIX ===
```

Fix 6: Refactor and Export Client's Rent State Functions

File: `pkg/rent/rent.go` (refactored and added exported wrappers)

Issue: The client's rent state transition logic was embedded in `checkRentStateTransitionAllowed()`, which requires a `TransactionCtx` to look up the account's pubkey. For fee payer validation, we already have the pubkey directly and don't have a `TransactionCtx` yet.

Original client code structure:

```
// Original: All logic was in checkRentStateTransitionAllowed
func checkRentStateTransitionAllowed(preRentState *RentStateInfo, postRentState *RentStateInfo, txCtx *sealevel.TransactionCtx, idx
uint64) error {
    // ... nil checks ...

    acct, err := txCtx.AccountAtIndex(idx)  // Requires TransactionCtx
    if err != nil {
        panic("programming error - acct didn't exist in TransactionAccounts")
    }

    // Special case: incinerator address
    if acct.Key == a.IncineratorAddr {
        return nil
    }

    // Rent state transition logic was inline here...
    if postRentState.RentState == RentStateUninitialized {
        return nil
    } else if postRentState.RentState == RentStateRentExempt {
        return nil
    } else if postRentState.RentState == RentStateRentPaying {
        // ... validation logic ...
    }
    return nil
}
```

Refactored structure:

1. **Extract core logic** into `checkRentStateTransition()` that takes pubkey directly:

```
// Core logic extracted - takes pubkey directly, no TransactionCtx needed
func checkRentStateTransition(preRentState *RentStateInfo, postRentState *RentStateInfo, pubkey solana.PublicKey) error {
    // Special case: incinerator address can accept any transition
    if pubkey == a.IncineratorAddr {
        return nil
    }

    if postRentState.RentState == RentStateUninitialized {
        return nil
    } else if postRentState.RentState == RentStateRentExempt {
        return nil
    } else if postRentState.RentState == RentStateRentPaying {
        if preRentState.RentState == RentStateUninitialized {
            return fmt.Errorf("[1] rent state transition not allowed...")
        } else if preRentState.RentState == RentStateRentExempt {
            return fmt.Errorf("[2] rent state transition not allowed...")
        } else if preRentState.RentState == RentStateRentPaying {
            if postRentState.RentPayingInfo.DataSize == preRentState.RentPayingInfo.DataSize &&
               postRentState.RentPayingInfo.Lamports <= preRentState.RentPayingInfo.Lamports {
                return nil
            } else {
                return fmt.Errorf("[3] rent state transition not allowed...")
            }
        }
    }
    return nil
}
```

2. **Simplify** `checkRentStateTransitionAllowed()` to call the extracted function:

```
// Now just extracts pubkey from TransactionCtx and delegates to checkRentStateTransition
func checkRentStateTransitionAllowed(preRentState *RentStateInfo, postRentState *RentStateInfo, txCtx *sealevel.TransactionCtx, idx
uint64) error {
    // ... nil checks ...

    acct, err := txCtx.AccountAtIndex(idx)
    if err != nil {
        panic("programming error - acct didn't exist in TransactionAccounts")
    }

    return checkRentStateTransition(preRentState, postRentState, acct.Key)
}
```

3. **Rename functions to be exported** (instead of adding wrapper functions):

Rather than adding wrapper functions like `RentStateFromAcct()` → `rentStateFromAcct()`, the cleaner approach is to directly rename the existing functions to be exported (capitalize them) and update all call sites:

```
// === START RV FIX ===
// Rename rentStateFromAcct -> RentStateFromAcct (export directly)
func RentStateFromAcct(acct *accounts.Account, rent *sealevel.SysvarRent) *RentStateInfo {
    // ... existing implementation ...
```

```
}

// Rename checkRentStateTransition -> CheckRentStateTransition (export directly)
func CheckRentStateTransition(preRentState *RentStateInfo, postRentState *RentStateInfo, pubkey solana.PublicKey) error {
    // ... existing implementation ...
}
// === END RV FIX ===
```

Note: This requires updating all existing call sites in the codebase to use the new capitalized names.

Actual Implementation

Summary of Changes

Modified files:

1. `pkg/replay/accounts.go` :
   - Changed `loadAndValidateTxAccts` signature to accept `*sealevel.ComputeBudgetLimits`
   - Added rent sysvar loading (using same pattern as `ReadRentSysvar()`)
   - Added fee calculation using `fees.CalculateTxFees`
   - Added `ValidateFeePayer()` function with nonce account handling
   - Updated `loadedAcctBytesLimit` references to `computeBudgetLimits.LoadedAccountBytes`
   - Added `fees` and `rent` package imports
2. `pkg/rent/rent.go` :
   - Extracted core transition logic into `CheckRentStateTransition()` (exported)
   - Renamed `rentStateFromAcct()` to `RentStateFromAcct()` (exported)
   - Updated all internal call sites to use new names

Key Design Decisions

1. **Rent Loading Pattern**: The rent sysvar loading code follows the same pattern as `ReadRentSysvar()` in `pkg/sealevel/sysvar_rent.go` - it panics if the rent sysvar account is not found. This is appropriate because in a properly initialized runtime, the rent sysvar should always exist.
2. **Full Fee Calculation**: Use `fees.CalculateTxFees()` to calculate the complete fee including priority fees, matching Agave's behavior.
3. **Nonce Account Handling**: Nonce accounts must maintain rent-exemption for nonce state (80 bytes), so `min_balance` is non-zero for nonce fee payers.
4. **Fee Deduction Separation**: Unlike Agave which deducts fees in `validate_fee_payer` , Mithril separates validation from fee deduction. Fees are deducted later in `pkg/fees/fees.go:CalculateAndDeductTxFees` . This is a cleaner separation of concerns.
5. **Reuse Existing Logic**: Instead of duplicating rent validation logic, we **reuse the client's existing functions**:
   - `rent.RentStateFromAcct()` - same logic as used in `NewRentStateInfo()`
   - `rent.CheckRentStateTransition()` - same logic as used in `VerifyRentStateChanges()`

This ensures fee payer validation uses the **exact same rent state transition rules** as post-execution validation, maintaining consistency with the client's existing behavior.

Test Results

Crash 1 (`crash-ad8559d4c9e4a370802b352183f33aa0206b9a00`):

```
Before: Agave returns status=32, Mithril executes transaction
After:  ✓ Both return status 32 (InsufficientFundsForRent), both reject before execution
```

Crash 2 (`crash-d20d4ea80620470dc7b60768c9e02f1d948bd516`):

```
Before: Agave returns status=5, Mithril returns status=32
After:  ✓ Both return status 5 (InsufficientFundsForFee)
```

References

- Agave fee payer validation: `svm/src/account_loader.rs` - `validate_fee_payer()` function
- Agave rent state checking: `svm/src/rent_calculator.rs` - `check_rent_state_with_account()` function
- Mithril transaction loading: `pkg/replay/accounts.go` - `loadAndValidateTxAccts()` function
- Mithril rent state validation: `pkg/rent/rent.go` - `checkRentStateTransition()` function
- Mithril rent sysvar reading: `pkg/sealevel/sysvar_rent.go` - `ReadRentSysvar()` function
- Mithril fee calculation: `pkg/fees/fees.go` - `CalculateTxFees()` function

[F-D05] LoaderV4 `setLoaderV4AcctState` Missing Features Parameter

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

Summary

The `setLoaderV4AcctState` function in `pkg/sealevel/loader_v4.go` was passing `nil` to `DataMutable()` instead of the actual features set, causing executable LoaderV4 program accounts to fail modification with `InstrErrExecutableDataModified` even when the `RemoveAccountsExecutableFlagChecks` feature was enabled.

Root Cause

In `setLoaderV4AcctState`, the function called `acct.DataMutable(nil)` instead of passing the actual features:

```
func setLoaderV4AcctState(acct *BorrowedAccount, state *LoaderV4State) error {
    stateSlice, err := acct.DataMutable(nil)  // BUG: nil features
    // ...
}
```

The `DataMutable` function calls `DataCanBeChanged` which checks:

```
func (acct *BorrowedAccount) DataCanBeChanged(f features.Features) error {
    if !f.IsActive(features.RemoveAccountsExecutableFlagChecks) && acct.IsExecutable() {
        return InstrErrExecutableDataModified
    }
    // ...
}
```

When `nil` features are passed, `IsActive()` returns `false`, so the check incorrectly rejects modifications to executable accounts even when the feature is enabled.

Impact

LoaderV4 program state updates (deploy, retract, finalize, set program length initialization) would fail with `InstrErrExecutableDataModified` (error code 27) when:

1. The `RemoveAccountsExecutableFlagChecks` feature is enabled
2. The program account is executable (which LoaderV4 programs are)

This prevents normal LoaderV4 program lifecycle operations from succeeding.

Crash File

`crash-53504f288575dd30824705773dfbe0d590efbc69`

Divergence

- **Agave**: Transaction executes successfully (OK)
- **Mithril**: Transaction fails with `InstructionError` (status 9, instruction_error 28 = ExecutableDataModified)

Fix

Updated `setLoaderV4AcctState` to accept a `features.Features` parameter and pass it to `DataMutable`:

File: `pkg/sealevel/loader_v4.go`

Before:

```
func setLoaderV4AcctState(acct *BorrowedAccount, state *LoaderV4State) error {
    stateSlice, err := acct.DataMutable(nil)
    if err != nil {
        return err
    }
    // ...
}
```

After:

```
func setLoaderV4AcctState(acct *BorrowedAccount, state *LoaderV4State, f features.Features) error {
    stateSlice, err := acct.DataMutable(f)
    if err != nil {
        return err
    }
    // ...
}
```

Updated all callers to pass `execCtx.Features`:

- `LoaderV4ProcessSetProgramLength` (initialization path)
- `LoaderV4ProcessDeploy`
- `LoaderV4ProcessRetract`
- `LoaderV4ProcessFinalize`

[F-D06] Incorrect Section Order Validation in ELF Loader

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

Summary

Mithril's ELF loader incorrectly validates section order by comparing section data offsets against the section header table offset (`eh.Shoff`) instead of tracking the previous section's end offset. This causes valid ELF files to be rejected and differs from Agave's behavior.

Root Cause

Agave's ELF parser validates that sections are non-overlapping and in ascending order by their data offsets. This is important because sections should not overlap each other's data regions.

Mithril's original code compared each section's data offset against the wrong value:

```
// pkg/sbpf/loader/parse.go (before fix)
func (l *Loader) readSectionHeaderTable() error {
    eh := &l.eh
    // ...
    sectionDataOff := uint64(0)

    for iter.Next() && iter.Err() == nil {
        i, sh := iter.Index(), iter.Item()
        // ... overlap checks ...

        // BUG: This compared sh.Off against eh.Shoff (section header table offset)
        // instead of sectionDataOff (previous section's end offset)
        if eh.Shoff < sectionDataOff {  // WRONG: should be sh.Off < sectionDataOff
            return fmt.Errorf("sections not in order")
        }

        sectionDataOff = shend
    }
}
```

The bug was comparing `eh.Shoff` (the section header table's offset) against `sectionDataOff` , instead of comparing `sh.Off` (the current section's data offset) against `sectionDataOff` .

Differential Behavior

Test Case: Valid ELF file with sections in ascending order

Agave Behavior:

- Validates sections are in ascending order by data offset
- Each section's `sh_offset` must be >= previous section's end
- Accepts valid ELF files

Mithril Behavior (before fix):

- Compared wrong values (`eh.Shoff` vs `sectionDataOff`)
- The check was logically incorrect

- Could reject valid files or accept invalid files

Fix

The fix corrects the comparison to check section data offsets in order:

File: `pkg/sbpf/loader/parse.go`

```
func (l *Loader) readSectionHeaderTable() error {
    eh := &l.eh
    // ...
    sectionDataOff := uint64(0)

    for iter.Next() && iter.Err() == nil {
        i, sh := iter.Index(), iter.Item()
        // ... overlap checks ...

        // === START RV FIX ===
        // Fixed: Check that section data offsets are in ascending order,
        // not that section header table offset > section data end.
        // This matches Agave's sbpf behavior: sections must be non-overlapping
        // and in ascending order by their data offsets.
        if sh.Off < sectionDataOff {
            return fmt.Errorf("sections not in order")
        }
        // === END RV FIX ===

        if shend > l.fileSize {
            return fmt.Errorf("section %d out of bounds", i)
        }

        sectionDataOff = shend
    }
}
```

Impact

Severity: Medium

This bug causes incorrect validation of section ordering:

- Valid ELF files might be incorrectly rejected
- The check was comparing unrelated values, making it meaningless
- Could affect programs with specific section layouts

Files Affected

- `pkg/sbpf/loader/parse.go` - `readSectionHeaderTable()` function

Agave Reference

Agave's sbpf validates section order in `sbpf/src/elf_parser/mod.rs`:

```
// Error definition (lines 42-44)
/// Sections are not sorted in ascending order
#[error("sections not in ascending order")]
```

```
SectionNotInOrder,

// Validation code (lines 199-215)
let mut offset = 0usize;
for section_header in section_header_table.iter() {
    if section_header.sh_type == SHT_NOBITS {
        continue;
    }
    let section_range = section_header.sh_offset as usize
        ..(section_header.sh_offset as usize)
            .err_checked_add(section_header.sh_size as usize)?;
    check_that_there_is_no_overlap(&section_range, &file_header_range)?;
    check_that_there_is_no_overlap(&section_range, &program_header_table_range)?;
    check_that_there_is_no_overlap(&section_range, &section_header_table_range)?;
    if section_range.start < offset {
        return Err(ElfParserError::SectionNotInOrder);
    }
    offset = section_range.end;
    if offset > elf_bytes.len() {
        return Err(ElfParserError::OutOfBounds);
    }
}
```

The key validation is:

```
if section_range.start < offset {
    return Err(ElfParserError::SectionNotInOrder);
}
offset = section_range.end;
```

This ensures that each section's data offset (`section_range.start` , which is `sh_offset`) must be >= the previous section's end offset. Mithril's bug was comparing the wrong value (`eh.Shoff`) instead of the current section's offset (`sh.Off`).

[F-D07] Incorrect Executable Flag Checks in BorrowedAccount

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

Summary

Mithril's `BorrowedAccount` implementation incorrectly checks the executable flag in several methods (`DataCanBeChanged` , `SetOwner` , `SetLamports` , `SetExecutable`), causing transaction execution divergence from Agave. Agave's equivalent functions do not perform executable flag checks in these contexts.

Root Cause

Mithril added executable flag checks that don't exist in Agave's implementation:

1. **`DataCanBeChanged`** : Mithril checked `!RemoveAccountsExecutableFlagChecks && IsExecutable()` and returned `InstrErrExecutableDataModified` . Agave's `can_data_be_changed()` only checks writability and ownership.
2. **`SetOwner`** : Mithril checked the executable flag and returned `InstrErrModifiedProgramId` . Agave's `set_owner()` only checks ownership, writability, and zeroed data.
3. **`SetLamports`** : Mithril checked the executable flag and returned `InstrErrExecutableLamportChange` . Agave's `set_lamports()` only checks ownership (for decreases) and writability.
4. **`SetExecutable`** : Mithril prevented removing the executable flag with the feature check. Agave's `set_executable()` allows changing executable from true to false.

Affected Code

File: `pkg/sealevel/borrowed_account.go`

DataCanBeChanged (Before Fix)

```
func (acct *BorrowedAccount) DataCanBeChanged(f features.Features) error {
    if !f.IsActive(features.RemoveAccountsExecutableFlagChecks) && acct.IsExecutable() {
        return InstrErrExecutableDataModified
    }
    if !acct.IsWritable() {
        return InstrErrReadonlyDataModified
    }
    if !acct.IsOwnedByCurrentProgram() {
        return InstrErrExternalAccountDataModified
    }
    return nil
}
```

DataCanBeChanged (After Fix)

```
func (acct *BorrowedAccount) DataCanBeChanged(f features.Features) error {
    // === START RV FIX ===
    // Removed the executable flag check that was incorrectly added.
    // Agave's can_data_be_changed() only checks:
    // 1. If the account is writable
    // 2. If the account is owned by the current program
```

```
    // It does NOT check the executable flag.
    // === END RV FIX ===
    if !acct.IsWritable() {
        return InstrErrReadonlyDataModified
    }
    if !acct.IsOwnedByCurrentProgram() {
        return InstrErrExternalAccountDataModified
    }
    return nil
}
```

SetOwner (Before Fix)

```
func (acct *BorrowedAccount) SetOwner(f features.Features, owner solana.PublicKey) error {
    if !acct.IsOwnedByCurrentProgram() {
        return InstrErrModifiedProgramId
    }
    if !acct.IsWritable() {
        return InstrErrModifiedProgramId
    }
    if !f.IsActive(features.RemoveAccountsExecutableFlagChecks) && acct.IsExecutable() {
        return InstrErrModifiedProgramId
    }
    if !acct.IsZeroed() {
        return InstrErrModifiedProgramId
    }
    // ...
}
```

SetOwner (After Fix)

```
func (acct *BorrowedAccount) SetOwner(f features.Features, owner solana.PublicKey) error {
    // === START RV FIX ===
    // Removed the executable flag check that was incorrectly added.
    // Agave's set_owner() only checks:
    // 1. If the account is owned by the current program
    // 2. If the account is writable
    // 3. If the data is zeroed
    // It does NOT check the executable flag.
    // === END RV FIX ===
    if !acct.IsOwnedByCurrentProgram() {
        return InstrErrModifiedProgramId
    }
    if !acct.IsWritable() {
        return InstrErrModifiedProgramId
    }
    if !acct.IsZeroed() {
        return InstrErrModifiedProgramId
    }
    // ...
}
```

SetLamports (Before Fix)

```
func (acct *BorrowedAccount) SetLamports(lamports uint64, f features.Features) error {
    if !acct.IsOwnedByCurrentProgram() && lamports < acct.Lamports() {
```

```
        return InstrErrExternalAccountLamportSpend
    }
    if !acct.IsWritable() {
        return InstrErrReadonlyLamportChange
    }
    if !f.IsActive(features.RemoveAccountsExecutableFlagChecks) && acct.IsExecutable() {
        return InstrErrExecutableLamportChange
    }
    // ...
}
```

SetLamports (After Fix)

```
func (acct *BorrowedAccount) SetLamports(lamports uint64, f features.Features) error {
    // === START RV FIX ===
    // Removed the executable flag check that was incorrectly added.
    // Agave's set_lamports() only checks:
    // 1. If the account is owned by the current program (for lamport decrease)
    // 2. If the account is writable
    // It does NOT check the executable flag.
    // === END RV FIX ===
    if !acct.IsOwnedByCurrentProgram() && lamports < acct.Lamports() {
        return InstrErrExternalAccountLamportSpend
    }
    if !acct.IsWritable() {
        return InstrErrReadonlyLamportChange
    }
    // ...
}
```

SetExecutable (Before Fix)

```
func (acct *BorrowedAccount) SetExecutable(f features.Features, isExecutable bool) error {
    // ...rent exempt check, ownership check, writable check...

    // can't remove executable flag
    if !f.IsActive(features.RemoveAccountsExecutableFlagChecks) && acct.IsExecutable() && !isExecutable {
        return InstrErrExecutableModified
    }
    // ...
}
```

SetExecutable (After Fix)

```
func (acct *BorrowedAccount) SetExecutable(f features.Features, isExecutable bool) error {
    // === START RV FIX ===
    // Removed the "can't remove executable flag" check that was incorrectly added.
    // Agave's set_executable() only checks:
    // 1. If the account is rent exempt
    // 2. If the account is owned by the current program
    // 3. If the account is writable
    // It does NOT prevent removing the executable flag.
    // === END RV FIX ===
    if !acct.TxCtx.Rent.IsExempt(acct.Lamports(), uint64(len(acct.Data()))) {
        return InstrErrExecutableAccountNotRentExempt
    }
```

```
    if !acct.IsOwnedByCurrentProgram() {
        return InstrErrExecutableModified
    }
    if !acct.IsWritable() {
        return InstrErrExecutableModified
    }
    // ...
}
```

Agave Reference

From `agave/transaction-context/src/lib.rs` :

```
pub fn can_data_be_changed(&self) -> Result<(), InstructionError> {
    // and only if the account is writable
    if !self.is_writable() {
        return Err(InstructionError::ReadonlyDataModified);
    }
    // and only if we are the owner
    if !self.is_owned_by_current_program() {
        return Err(InstructionError::ExternalAccountDataModified);
    }
    Ok(())
}

pub fn set_owner(&mut self, pubkey: &[u8]) -> Result<(), InstructionError> {
    if !self.is_owned_by_current_program() {
        return Err(InstructionError::ModifiedProgramId);
    }
    if !self.is_writable() {
        return Err(InstructionError::ModifiedProgramId);
    }
    if !is_zeroed(self.get_data()) {
        return Err(InstructionError::ModifiedProgramId);
    }
    // ... (no executable check)
}

pub fn set_lamports(&mut self, lamports: u64) -> Result<(), InstructionError> {
    if !self.is_owned_by_current_program() && lamports < self.get_lamports() {
        return Err(InstructionError::ExternalAccountLamportSpend);
    }
    if !self.is_writable() {
        return Err(InstructionError::ReadonlyLamportChange);
    }
    // ... (no executable check)
}

pub fn set_executable(&mut self, is_executable: bool) -> Result<(), InstructionError> {
    if !self.transaction_context.rent.is_exempt(self.get_lamports(), self.get_data().len()) {
        return Err(InstructionError::ExecutableAccountNotRentExempt);
    }
    if !self.is_owned_by_current_program() {
        return Err(InstructionError::ExecutableModified);
    }
    if !self.is_writable() {
```

```
        return Err(InstructionError::ExecutableModified);
    }
    // ... (no check preventing removal of executable flag)
}
```

Impact

Transactions that modify data, lamports, owner, or executable flag of accounts marked as executable would fail in Mithril but succeed in Agave, causing consensus divergence between validators.

Crash File

`crash-1ac273802f08e17eac4e6e369bf418392a7170f5`

[F-D08] String Parsing in ELF Loader Doesn't Match Agave Behavior

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

Summary

Mithril's ELF loader `getString()` function doesn't properly handle string parsing from ELF string tables when the requested maximum length would exceed section bounds. Agave clamps the read length to the section boundary and returns `StringTooLong` if no null terminator is found within that range. Additionally, Agave uses checked arithmetic to detect integer overflow and returns `OutOfBounds` when overflow occurs. Mithril was returning generic errors instead of properly matching this behavior.

Root Cause

Agave's `get_string_in_section()` function in `sbpf/src/elf_parser/mod.rs` has specific behavior:

```
// sbpf/src/elf_parser/mod.rs:468-495
pub fn get_string_in_section(
    elf_bytes: &'a [u8],
    section_header: &Elf64Shdr,
    offset_in_section: Elf64Word,
    maximum_length: usize,
) -> Result<&'a [u8], ElfParserError> {
    if section_header.sh_type != SHT_STRTAB {
        return Err(ElfParserError::InvalidSectionHeader);
    }
    // Uses checked arithmetic - returns ArithmeticOverflow on overflow
    let offset_in_file =
        (section_header.sh_offset as usize).err_checked_add(offset_in_section as usize)?;

    // KEY: Range is clamped to minimum of section end and offset+maximum_length
    // All additions use err_checked_add for overflow detection
    let string_range = offset_in_file
        ..(section_header.sh_offset as usize)
            .err_checked_add(section_header.sh_size as usize)?
            .min(offset_in_file.err_checked_add(maximum_length)?);

    let unterminated_string_bytes = elf_bytes
        .get(string_range)
        .ok_or(ElfParserError::OutOfBounds)?;

    // Look for null terminator within range
    unterminated_string_bytes
        .iter()
        .position(|byte| *byte == 0x00)
        .and_then(|string_length| unterminated_string_bytes.get(0..string_length))
        .ok_or_else(|| {
            // Return StringTooLong if no null terminator found
            ElfParserError::StringTooLong(
                String::from_utf8_lossy(unterminated_string_bytes).to_string(),
                maximum_length,
            )
        })
}
```

The `err_checked_add` function returns `ArithmeticOverflow` on integer overflow, which is converted to `ElfParserError::OutOfBounds`:

```
// sbpf/src/elf_parser/mod.rs:671-675
impl From<ArithmeticOverflow> for ElfParserError {
    fn from(_: ArithmeticOverflow) -> ElfParserError {
        ElfParserError::OutOfBounds
    }
}
```

Key behaviors:

1. **Checked arithmetic**: All additions use `err_checked_add()` which returns `OutOfBounds` on overflow
2. **Range clamping**: Read length is `min(section_end, offset + max_len)` - not an error if max_len exceeds section
3. **OutOfBounds**: Returned if the starting offset is beyond section/file bounds OR if arithmetic overflow occurs
4. **StringTooLong**: Returned if no null terminator is found within the clamped range

Mithril's original implementation had different behavior:

- Did not use checked arithmetic for additions, risking silent integer overflow
- Could return wrong error types in overflow scenarios

Fix

The fix implements Agave's exact behavior for string parsing with checked arithmetic:

File: `pkg/sbpf/loader/parse.go`

```
func (l *Loader) getString(strtab *elf.Section64, stroff uint32, maxLen uint16) (string, error) {
    if elf.SectionType(strtab.Type) != elf.SHT_STRTAB {
        return "", fmt.Errorf("invalid strtab")
    }
    // === START RV FIX ===
    // Match Agave's behavior with checked arithmetic:
    // Agave uses err_checked_add() which returns ArithmeticOverflow on overflow,
    // which is converted to ElfParserError::OutOfBounds.
    // See sbpf/src/elf_parser/mod.rs:671-675

    // offset_in_file = section_header.sh_offset + offset_in_section
    offset, carry := bits.Add64(strtab.Off, uint64(stroff), 0)
    if carry != 0 {
        return "", ErrOutOfBounds
    }

    // section_end = section_header.sh_offset + section_header.sh_size
    sectionEnd, carry := bits.Add64(strtab.Off, strtab.Size, 0)
    if carry != 0 {
        return "", ErrOutOfBounds
    }

    // maxEnd = offset + maxLen
    maxEnd, carry := bits.Add64(offset, uint64(maxLen), 0)
    if carry != 0 {
        return "", ErrOutOfBounds
    }
```

```
    // string_range = offset_in_file..min(section_end, maxEnd)
    // Clamp read length to minimum of section end and offset + maxLen
    if sectionEnd < maxEnd {
        maxEnd = sectionEnd
    }

    // Check if offset is within file bounds
    if offset > l.fileSize {
        return "", ErrOutOfBounds
    }

    // Calculate read length
    readLen := maxEnd - offset
    if maxEnd > l.fileSize {
        // Clamp to file size if needed
        readLen = l.fileSize - offset
    }
    if readLen == 0 {
        return "", ErrOutOfBounds
    }

    rd := bufio.NewReader(io.NewSectionReader(l.rd, int64(offset), int64(readLen)))
    var builder strings.Builder
    for {
        b, err := rd.ReadByte()
        if err != nil {
            // Agave returns StringTooLong if no null terminator found within range
            return "", ErrStringTooLong
        }
        if b == 0 {
            break
        }
        builder.WriteByte(b)
    }
    // === END RV FIX ===
    return builder.String(), nil
}
```

File: `pkg/sbpf/loader/loader.go` (error type definitions)

```
// Error types matching Agave's ELF parser errors
var ErrOutOfBounds = errors.New("out of bounds")
var ErrStringTooLong = errors.New("string too long")
```

Impact

Severity: Medium

This affects how the ELF loader handles:

- Section names that might be near section boundaries
- Symbol names in malformed ELF files
- Any string table access where max length exceeds available data
- Maliciously crafted ELF files with large offset values designed to trigger integer overflow

Incorrect error types cause differential fuzzing failures and could affect error handling in callers. The lack of overflow checking could potentially lead to unexpected behavior when processing malformed ELF files.

Files Affected

- `pkg/sbpf/loader/parse.go` - `getString()` function (modified)
- `pkg/sbpf/loader/loader.go` - Added `ErrOutOfBounds` and `ErrStringTooLong` error types

Agave Reference

- `sbpf/src/elf_parser/mod.rs:468-495` - `get_string_in_section()` implementation
- `sbpf/src/elf_parser/mod.rs:32` - `ElfParserError::StringTooLong` definition
- `sbpf/src/elf_parser/mod.rs:26` - `ElfParserError::OutOfBounds` definition
- `sbpf/src/elf_parser/mod.rs:671-675` - `ArithmeticOverflow` to `OutOfBounds` conversion
- `sbpf/src/lib.rs:51-65` - `err_checked_add()` implementation

[F-D09] Incorrect ELF Deploy Checks During Execution

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

Summary

Mithril was incorrectly applying ELF deployment validation checks during program execution, causing potential execution failures for programs that would execute successfully in Agave.

Affected Code

File: `pkg/sealevel/bpf_loader.go`
Function: `executeProgramFromBytes`

Root Cause

The `executeProgramFromBytes` function was calling `NewLoaderWithSyscalls` with `elfDeployChecks=true`, which triggers strict validation of syscall references in the ELF. This validation is only appropriate during program deployment, not during execution.

The `elfDeployChecks` parameter (equivalent to Agave's `reject_broken_elfs`) controls whether the loader should reject ELF files that reference syscalls not present in the syscall registry. During deployment, this check ensures that programs don't reference invalid syscalls. During execution, this check is unnecessary because:

1. The program has already passed deployment validation
2. Some syscalls may be conditionally available based on features
3. Agave defaults to `reject_broken_elfs=false` and only sets it to `true` during deployment

Agave Implementation

In Agave's sbpf (`sbpf/src/vm.rs`), the default configuration is:

```
impl Default for Config {
    fn default() -> Self {
        Self {
            // ...
            reject_broken_elfs: false,  // Default is false
            // ...
        }
    }
}
```

During deployment, Agave explicitly enables strict checking in `morph_into_deployment_environment_v1` (`agave/programs/bpf_loader/src/lib.rs`):

```
fn morph_into_deployment_environment_v1(
    from: Arc<BuiltinProgram<InvokeContext>>,
) -> Result<BuiltinProgram<InvokeContext>, ElfError> {
    let mut config = from.get_config().clone();
    config.reject_broken_elfs = true;  // Only true during deployment
    // ...
}
```

The check itself is in `sbpf/src/elf.rs`:

```
if config.reject_broken_elfs
    && loader.get_function_registry().lookup_by_key(hash).is_none()
{
    return Err(ElfError::UnresolvedSymbol(
        String::from_utf8_lossy(name).to_string(),
        r_offset.checked_div(ebpf::INSN_SIZE).unwrap_or(0),
        r_offset,
    ));
}
```

The Bug

Before (incorrect):

```
func executeProgramFromBytes(execCtx *ExecutionCtx, programAddr solana.PublicKey, programData []byte, syscallRegistry
sbpf.SyscallRegistry) error {
    start := time.Now()
    loader, err := loader.NewLoaderWithSyscalls(programData, syscallRegistry, true, &execCtx.Features)
    // ...
}
```

After (correct):

```
func executeProgramFromBytes(execCtx *ExecutionCtx, programAddr solana.PublicKey, programData []byte, syscallRegistry
sbpf.SyscallRegistry) error {
    start := time.Now()
    // === START RV FIX ===
    // Pass elfDeployChecks=false during execution (not deployment).
    // Deployment checks validate that all syscalls in the ELF exist in the registry.
    // During execution, the program has already been deployed and validated,
    // so we skip these checks. This matches Agave's behavior where
    // reject_broken_elfs is false during normal program execution.
    loader, err := loader.NewLoaderWithSyscalls(programData, syscallRegistry, false, &execCtx.Features)
    // === END RV FIX ===
    // ...
}
```

Correct Usage Pattern

Function	Context	`elfDeployChecks`
`deployProgram`	Program deployment	`true`
`executeProgramFromBytes`	Program execution	`false`

Impact

Programs that reference syscalls which:

- Were valid at deployment time but removed later
- Are conditionally available based on feature flags
- Exist in some configurations but not others

...could fail to execute in Mithril while executing successfully in Agave.

Status

Addressed in commit a5e9a13.

[F-D10] Missing `verifyCheckSh32` Handler in SBPF Verifier

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

Summary

The SBPF bytecode verifier in Mithril is missing a case handler for `verifyCheckSh32`, causing valid 32-bit shift instructions to be incorrectly rejected as "invalid opcode". This prevents legitimate sBPF programs containing left shift 32-bit instructions (LSH32_IMM, RSH32_IMM, ARSH32_IMM) from executing.

Agave Reference

In Agave's sbpf verifier (`sbpf/src/verifier.rs`), 32-bit shift instructions are validated using `check_imm_shift` with a 32-bit limit:

```
// sbpf/src/verifier.rs

/// Check that the imm is a valid shift operand
fn check_imm_shift(insn: &ebpf::Insn, insn_ptr: usize, imm_bits: u64) -> Result<(), VerifierError> {
    let shift_by = insn.imm as u64;
    if insn.imm < 0 || shift_by >= imm_bits {
        return Err(VerifierError::ShiftWithOverflow(
            shift_by, imm_bits, insn_ptr,
        ));
    }
    Ok(())
}

// In the main verification loop:
ebpf::LSH32_IMM  => { check_imm_shift(&insn, insn_ptr, 32)?; },
ebpf::RSH32_IMM  => { check_imm_shift(&insn, insn_ptr, 32)?; },
ebpf::ARSH32_IMM => { check_imm_shift(&insn, insn_ptr, 32)?; },
```

Root Cause

In `pkg/sbpf/verifier.go` , the validation map correctly maps opcodes 0x64, 0x74, and 0xc4 to `verifyCheckSh32` :

```
/* 0x64 */ verifyCheckSh32 /* 0x65 */, checkJmp ...
/* 0x74 */ verifyCheckSh32 /* 0x75 */, checkJmp ...
/* 0xc4 */ verifyCheckSh32 /* 0xc5 */, checkJmp ...
```

However, the switch statement in `VerifyProgram()` that handles each validation code is missing the `case verifyCheckSh32:` handler. This causes the verifier to fall through to the `default` case, which returns "invalid opcode" error.

The `verifyCheckSh32` constant is defined but never handled:

```
const (
    verifyValid = iota
    verifyCheckSt
    // ...
    verifyCheckDiv
    verifyCheckSh32  // Defined here
    verifyCheckSh64
    verifyInvalid
```

```
    // ...
)
```

Impact

- Any sBPF program containing 32-bit shift-by-immediate instructions fails verification
- Programs that would execute correctly on Agave are rejected by Mithril
- This affects opcodes:
  - `0x64` - LSH32_IMM (left shift 32-bit by immediate)
  - `0x74` - RSH32_IMM (right shift 32-bit by immediate)
  - `0xc4` - ARSH32_IMM (arithmetic right shift 32-bit by immediate)

Fix

Add the missing case handler for `verifyCheckSh32` in `pkg/sbpf/verifier.go`:

File: `pkg/sbpf/verifier.go`

```
// === START RV FIX ===
case verifyCheckSh32:
    {
        if ins.Imm() >= 32 {
            return fmt.Errorf("sh overflow")
        }
    }
// === END RV FIX ===
```

The handler validates that the shift amount (immediate value) is less than 32 for 32-bit operations, matching the behavior of `verifyCheckSh64` which checks for values >= 64.

Status

Addressed in commit fa91b0a.

[F-D11] Side Effects Occurring After Error in VM Interpreter

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

Summary

Mithril's sBPF VM interpreter contains a systematic pattern where side effects (register modifications, PC updates) occur even after an error is set. In Agave, the `throw_error!` macro immediately returns without executing any further code. In Mithril, the code continues executing and modifies state (registers, PC) after setting the error.

This bug affects multiple instruction handlers in `pkg/sbpf/interpreter.go` and causes divergent behavior between Mithril and Agave when errors occur.

Root Cause

In Agave's interpreter, the `throw_error!` macro immediately returns from the function:

```
macro_rules! throw_error {
    ($self:expr, $err:expr) => {{
        $self.vm.registers[11] = $self.reg[11];  // Copy current PC to VM registers
        $self.vm.program_result = ProgramResult::Err($err);
        return false;  // Return immediately - no side effects after error
    }};
    (DivideByZero; $self:expr, $src:expr, $ty:ty) => {
        if $src as $ty == 0 {
            throw_error!($self, EbpfError::DivideByZero);
        }
    };
    (DivideOverflow; $self:expr, $src:expr, $dst:expr, $ty:ty) => {
        if $dst as $ty == <$ty>::MIN && $src as $ty == -1 {
            throw_error!($self, EbpfError::DivideOverflow);
        }
    };
}
```

In Mithril, after setting `err`, the code continues to execute and may modify registers and PC before the error is checked at the end of the instruction loop.

Affected Instructions

1. Call Instructions (OpCall, OpCallx)

Issue: PC is modified even when call depth is exceeded or target is invalid.

OpCallx (before fix):

```
case OpCallx:
    var target uint64
    // ... target calculation ...

    ok = ip.stack.Push(r[:], pc+1)
    if !ok {
        err = ExcCallDepth
    }
    // BUG: These statements execute even when err is set!
```

```
    if target < ip.textVA || target >= VaddrStack || target >= ip.textVA+uint64(len(ip.text)*8) {
        err = NewExcBadAccess(target, 8, false, "jump out-of-bounds")
    }
    pc = int64((target - ip.textVA) / 8)  // PC modified even on error!
```

Agave:

```
ebpf::CALL_REG   => {
    let target_pc = ...;
    if !self.push_frame(config) {
        return false;  // Returns immediately via throw_error!, no PC modification
    }
    check_pc!(self, next_pc, target_pc...);  // Also uses throw_error! on failure
},
```

2. Signed Division/Remainder Instructions

Issue: Register is modified even when divide overflow is detected.

OpSdiv64Reg (buggy):

```
case OpSdiv64Reg:
    if !ip.sbpfVersion.EnablePqr() {
        err = ExcInvalidInstr
        break
    }
    if src := int64(r[ins.Src()]); src != 0 {
        if int64(r[ins.Dst()]) == math.MinInt64 && src == -1 {
            err = ExcDivideOverflow  // Error set here...
        }
        r[ins.Dst()] = uint64(int64(r[ins.Dst()]) / src)  // ...but division still happens!
    } else {
        err = ExcDivideByZero
    }
    pc++
```

Agave:

```
ebpf::SDIV64_REG if self.executable.get_sbpf_version().enable_pqr() => {
    throw_error!(DivideByZero; self, self.reg[src], i64);     // Returns if src == 0
    throw_error!(DivideOverflow; self, self.reg[src], self.reg[dst], i64);  // Returns if overflow
    self.reg[dst] = (self.reg[dst] as i64 / self.reg[src] as i64) as u64;  // Only reaches here if no error
},
```

The following signed division/remainder instructions have this bug:

- `OpSdiv32Imm` - DivideOverflow check followed by register modification
- `OpSdiv32Reg` - DivideOverflow check followed by register modification
- `OpSdiv64Imm` - DivideOverflow check followed by register modification
- `OpSdiv64Reg` - DivideOverflow check followed by register modification
- `OpSrem32Imm` - DivideOverflow check followed by register modification
- `OpSrem32Reg` - DivideOverflow check followed by register modification
- `OpSrem64Imm` - DivideOverflow check followed by register modification
- `OpSrem64Reg` - DivideOverflow check followed by register modification

---

Differential Behavior

Test Case 1: VM execution crash file `crash-fe2fc8a78769d2a577ea3fe099e9650f3ae29322`

- Program triggers `CallDepthExceeded` at instruction PC=296

Agave Behavior (Rust):

```
error: 3 (CallDepthExceeded)
pc: 296
```

Mithril Behavior (Go, before fix):

```
error: 3 (CallDepthExceeded)
pc: 295  (or some other incorrect value based on target)
```

Test Case 2: VM execution crash file `crash-c59334862f68c1b131824dbe70e8f55cca82b206`

- Program triggers `CallOutsideTextSegment` at PC=96

Agave Behavior (Rust):

```
error: 8 (CallOutsideTextSegment)
pc: 96
```

Mithril Behavior (Go, before fix):

```
error: 13 (AccessViolation)  // Wrong error type!
pc: 2305843008676823039      // Corrupted PC value!
```

Impact

Severity: Medium

1. **Register Corruption**: When divide overflow occurs, the destination register is incorrectly modified with an undefined/overflow value
2. **Incorrect Error Reporting**: Exception PC values are wrong, making debugging difficult
3. **Wrong Error Types**: Using `ExcBadAccess` instead of `ExcCallOutsideTextSegment` for call target bounds check
4. **Agave Incompatibility**: Mithril produces different VM state than the reference implementation on errors
5. **Security Risk**: Register values after errors may be observable and differ from expected behavior

---

Recommended Fix

Fix 1: Call Instructions (OpCall, OpCallx)

Wrap PC modification in the success branch and use correct error type for bounds check:

OpCallx fix:

```go
case OpCallx:
    var target uint64
    if ip.sbpfVersion.CallXUsesSrcReg() {
        target = r[ins.Src()]
    } else {
        target = r[ins.Uimm()]
    }
    target &= ^(uint64(0x7))

    var ok bool
    ok = ip.stack.Push(r[:], pc+1)
```

```
    if !ok {
        err = ExcCallDepth
        // === START RV FIX ===
        // Don't modify PC when call depth exceeded
        // Agave's throw_error! returns immediately without jumping
        // === END RV FIX ===
    } else {
        // === START RV FIX ===
        // Check if target is within text segment bounds before modifying PC.
        // In Agave's check_pc! macro, if the target is out of bounds,
        // throw_error! is called which returns immediately without modifying next_pc.
        // Use ExcCallOutsideTextSegment (error 8) instead of ExcBadAccess (error 13)
        // to match Agave's error mapping.
        if target < ip.textVA || target >= VaddrStack || target >= ip.textVA+uint64(len(ip.text)*8) {
            err = ExcCallOutsideTextSegment
            // Don't modify PC - Agave's throw_error! returns immediately
        } else {
            pc = int64((target - ip.textVA) / 8)
        }
        // === END RV FIX ===
    }
```

OpCall fix:

```
case OpCall:
    if sc, ok := ip.syscalls(ins.Uimm()); ok {
        r[0], err = sc.Invoke(ip, r[1], r[2], r[3], r[4], r[5])
        pc++
    } else if target, ok := ip.funcs[ins.Uimm()]; ok {
        ok = ip.stack.Push(r[:], pc+1)
        if !ok {
            err = ExcCallDepth
            // === START RV FIX ===
            // Don't modify PC when call depth exceeded
            // Agave's throw_error! returns immediately without jumping
            // === END RV FIX ===
        } else {
            pc = target
        }
    } else {
        err = ExcCallDest{ins.Uimm()}
    }
```

Fix 2: Signed Division/Remainder Instructions

Add early return after setting DivideOverflow error. For each affected instruction, wrap the operation in an `else` block:

OpSdiv64Reg fix:

```
case OpSdiv64Reg:
    if !ip.sbpfVersion.EnablePqr() {
        err = ExcInvalidInstr
        break
    }
    if src := int64(r[ins.Src()]); src != 0 {
        if int64(r[ins.Dst()]) == math.MinInt64 && src == -1 {
            err = ExcDivideOverflow
```

```
            // === START RV FIX ===
            // Don't execute division - Agave's throw_error! returns immediately
            // === END RV FIX ===
        } else {
            r[ins.Dst()] = uint64(int64(r[ins.Dst()]) / src)
        }
    } else {
        err = ExcDivideByZero
    }
    pc++
```

OpSdiv32Imm fix:

```
case OpSdiv32Imm:
    if !ip.sbpfVersion.EnablePqr() {
        err = ExcInvalidInstr
        break
    }
    if int32(r[ins.Dst()]) == math.MinInt32 && ins.Imm() == -1 {
        err = ExcDivideOverflow
        // === START RV FIX ===
        // Don't execute division - Agave's throw_error! returns immediately
        // === END RV FIX ===
    } else {
        r[ins.Dst()] = uint64(uint32(int32(r[ins.Dst()]) / ins.Imm()))
    }
    pc++
```

OpSdiv32Reg fix:

```
case OpSdiv32Reg:
    if !ip.sbpfVersion.EnablePqr() {
        err = ExcInvalidInstr
        break
    }
    if src := int32(r[ins.Src()]); src != 0 {
        if int32(r[ins.Dst()]) == math.MinInt32 && src == -1 {
            err = ExcDivideOverflow
            // === START RV FIX ===
            // Don't execute division - Agave's throw_error! returns immediately
            // === END RV FIX ===
        } else {
            r[ins.Dst()] = uint64(uint32(int32(r[ins.Dst()]) / src))
        }
    } else {
        err = ExcDivideByZero
    }
    pc++
```

OpSdiv64Imm fix:

```
case OpSdiv64Imm:
    if !ip.sbpfVersion.EnablePqr() {
        err = ExcInvalidInstr
        break
    }
```

```
    if int64(r[ins.Dst()]) == math.MinInt64 && ins.Imm() == -1 {
        err = ExcDivideOverflow
        // === START RV FIX ===
        // Don't execute division - Agave's throw_error! returns immediately
        // === END RV FIX ===
    } else {
        r[ins.Dst()] = uint64(int64(r[ins.Dst()]) / int64(ins.Imm()))
    }
    pc++
```

OpSrem32Imm fix:

```
case OpSrem32Imm:
    if !ip.sbpfVersion.EnablePqr() {
        err = ExcInvalidInstr
        break
    }
    if int32(r[ins.Dst()]) == math.MinInt32 && ins.Imm() == -1 {
        err = ExcDivideOverflow
        // === START RV FIX ===
        // Don't execute remainder - Agave's throw_error! returns immediately
        // === END RV FIX ===
    } else {
        r[ins.Dst()] = uint64(uint32(int32(r[ins.Dst()]) % ins.Imm()))
    }
    pc++
```

OpSrem32Reg fix:

```
case OpSrem32Reg:
    if !ip.sbpfVersion.EnablePqr() {
        err = ExcInvalidInstr
        break
    }
    if src := int32(r[ins.Src()]); src != 0 {
        if int32(r[ins.Dst()]) == math.MinInt32 && src == -1 {
            err = ExcDivideOverflow
            // === START RV FIX ===
            // Don't execute remainder - Agave's throw_error! returns immediately
            // === END RV FIX ===
        } else {
            r[ins.Dst()] = uint64(uint32(int32(r[ins.Dst()]) % int32(r[ins.Src()])))
        }
    } else {
        err = ExcDivideByZero
    }
    pc++
```

OpSrem64Imm fix:

```
case OpSrem64Imm:
    if !ip.sbpfVersion.EnablePqr() {
        err = ExcInvalidInstr
        break
    }
    if int64(r[ins.Dst()]) == math.MinInt64 && ins.Imm() == -1 {
```

```
        err = ExcDivideOverflow
        // === START RV FIX ===
        // Don't execute remainder - Agave's throw_error! returns immediately
        // === END RV FIX ===
    } else {
        r[ins.Dst()] = uint64(int64(r[ins.Dst()]) % int64(ins.Imm()))
    }
    pc++
```

OpSrem64Reg fix:

```
case OpSrem64Reg:
    if !ip.sbpfVersion.EnablePqr() {
        err = ExcInvalidInstr
        break
    }
    if src := int64(r[ins.Src()]); src != 0 {
        if int64(r[ins.Dst()]) == math.MinInt64 && src == -1 {
            err = ExcDivideOverflow
            // === START RV FIX ===
            // Don't execute remainder - Agave's throw_error! returns immediately
            // === END RV FIX ===
        } else {
            r[ins.Dst()] = uint64(int64(r[ins.Dst()]) % int64(r[ins.Src()]))
        }
    } else {
        err = ExcDivideByZero
    }
    pc++
```

Fix 3: Add ExcCallOutsideTextSegment Error Type

Add new error type in `pkg/sbpf/vm.go`:

```
var (
    ExcDivideByZero            = errors.New("division by zero")
    ExcDivideOverflow          = errors.New("divide overflow")
    ExcOutOfCU                 = errors.New("compute unit overrun")
    ExcCallDepth               = errors.New("call depth exceeded")
    ExcInvalidInstr            = errors.New("invalid instruction - feature not enabled")
    ExcCallOutsideTextSegment  = errors.New("invalid call destination")  // NEW
)
```

Fix 4: Update Exception PC Handling (Related to [F-D03])

The exception PC calculation logic in the error handling path must be updated to account for the new `ExcCallOutsideTextSegment` error. This is related to finding **[F-D03] Incorrect Exception Handling in VM Interpreter** which established the pattern for determining when PC was modified vs not modified.

When `OpCallx` encounters an `ExcCallOutsideTextSegment` error, the PC is NOT modified (similar to `ExcCallDepth`), so we should NOT subtract 1 from the exception PC.

Add to the exception handling logic in `pkg/sbpf/interpreter.go`:

```
// Determine the correct exception PC based on instruction type and error.
// ...
// However, for certain errors, pc is NOT modified:
```

```
// - ExcInvalidInstr with break (version check failures, etc.)
// - OpCall with ExcCallDepth (internal call, stack.Push failed)
// - OpCall with ExcCallDest (unknown function destination)
// - OpCallx with ExcCallDepth (stack.Push failed)
// - OpCallx with ExcCallOutsideTextSegment (target out of bounds)  // NEW
// In these cases, we should NOT subtract 1.
//
// Note: OpCall syscall errors DO have pc++ executed, so we subtract 1.
exceptPC := pc
_, isCallDestError := err.(ExcCallDest)
isCallDepthError := err == ExcCallDepth
isInvalidInstrError := err == ExcInvalidInstr
isCallOutsideTextSegmentError := err == ExcCallOutsideTextSegment  // NEW

// PC was NOT modified in these specific cases
pcNotModified := false
if isInvalidInstrError {
    // ExcInvalidInstr errors use 'break' which skips pc++
    pcNotModified = true
} else if ins.Op() == OpCall && (isCallDepthError || isCallDestError) {
    pcNotModified = true
} else if ins.Op() == OpCallx && (isCallDepthError || isCallOutsideTextSegmentError) {  // UPDATED
    pcNotModified = true
}
```

Testing

The fix can be verified using the differential fuzzing harness:

```
cd ~/Workspace/_audits_Overclock-Validator_mithril
make -f Makefile.solfuzz  # Rebuild with fix

cd ~/Workspace/solfuzz
./build/fuzz_sol_vm_interp ./crash-fe2fc8a78769d2a577ea3fe099e9650f3ae29322
./build/fuzz_sol_vm_interp ./crash-c59334862f68c1b131824dbe70e8f55cca82b206
```

Expected output (after fix):

```
Executed crash-* in N ms
***
*** NOTE: fuzzing was not performed, you have only
***       executed the target code on a fixed set of inputs.
***
```

Files Affected

Client Code (Modified):

1. **`pkg/sbpf/vm.go`** - Add `ExcCallOutsideTextSegment` error type
2. **`pkg/sbpf/interpreter.go`** - Multiple handlers:
   - `OpCall` handler - wrap PC modification in success branch
   - `OpCallx` handler - wrap PC modification in success branch, use correct error type
   - `OpSdiv32Imm`, `OpSdiv32Reg` - wrap operation in else block after overflow check
   - `OpSdiv64Imm`, `OpSdiv64Reg` - wrap operation in else block after overflow check
   - `OpSrem32Imm`, `OpSrem32Reg` - wrap operation in else block after overflow check

- `OpSrem64Imm`, `OpSrem64Reg` - wrap operation in else block after overflow check

---

References

- **Agave VM Interpreter**: `sbpf/src/interpreter.rs`
  - `throw_error!` macro definition showing immediate return behavior (lines 48-64)
  - `check_pc!` macro for call bounds checking (lines 67-80)
  - `CALL_REG` and signed division handlers
- **Related Finding**: `[F-D03] Incorrect Exception Handling in VM Interpreter.md`
  - Establishes the exception PC calculation pattern
  - This finding extends F-D03's exception handling to cover additional error cases
- **Differential Fuzzing Tests**:
  - `crash-fe2fc8a78769d2a577ea3fe099e9650f3ae29322` (CallDepthExceeded)
  - `crash-c59334862f68c1b131824dbe70e8f55cca82b206` (CallOutsideTextSegment)

[F-D12] Incorrect Right Shift in Signed High Multiply Instructions

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

Summary

The SBPF interpreter's signed high multiply instructions (`OpShmul64Imm` and `OpShmul64Reg`) use an incorrect right shift operation that produces wrong results for negative products. The `Int128.RShiftN` function uses sign-magnitude semantics instead of treating the value as two's complement, causing the high 64 bits of signed multiplications to be computed incorrectly.

Root Cause

In `pkg/sbpf/interpreter.go` , the signed high multiply instructions use `Int128.RShiftN(64)` to extract the high 64 bits of a 128-bit signed multiply result:

```
case OpShmul64Imm:
    if !ip.sbpfVersion.EnablePqr() {
        err = ExcInvalidInstr
        break
    }
    dst128 := wide.Int128FromInt64(int64(r[ins.Dst()]))
    imm128 := wide.Int128FromInt64(int64(ins.Imm()))
    r[ins.Dst()] = dst128.Mul(imm128).RShiftN(64).Uint64()  // Bug: RShiftN uses sign-magnitude!
    pc++
```

The `Int128.RShiftN` function in the `wide` library implements signed right shift using sign-magnitude semantics:

```
func (x Int128) RShiftN(n uint) (z Int128) {
    neg := false
    if x.Hi < 0 {
        x = x.Neg()   // Convert negative to positive (sign-magnitude)
        neg = true
    }
    // ... shift the magnitude ...
    if neg {
        return z.Neg()  // Negate result back
    }
    return z
}
```

This is incorrect for extracting the high bits of a two's complement multiplication. When the product is negative (e.g., `-683639278233902435`), the `RShiftN` function:

1. Negates to get the positive magnitude
2. Shifts the magnitude right by 64 (resulting in 0 for small magnitudes)
3. Negates back (0 negated is still 0)

This produces `0` instead of the correct value of `-1` (0xFFFFFFFFFFFFFFFF).

Example

For inputs:

- dst = 321305695 (0x1326BC5F)
- imm = -2127691133 (0x8132BE83)

The 128-bit product is `-683639278233902435`, represented in two's complement as:

- Hi = -1 (0xFFFFFFFFFFFFFFFF)
- Lo = 0xF68339B2D25D229D

The correct high 64 bits should be `-1` (0xFFFFFFFFFFFFFFFF), but the buggy `RShiftN` returns `0`.

Agave Reference Implementation

Agave (Rust) correctly uses wrapping operations that preserve two's complement semantics:

```
// From solana-labs/rbpf/src/interpreter.rs
self.reg[dst] = (self.reg[dst] as i64 as i128)
    .wrapping_mul(insn.imm as i128)
    .wrapping_shr(64) as u64;
```

The key difference is that Rust's `wrapping_shr` performs a **logical** (unsigned) right shift on the 128-bit result, preserving all bits correctly.

Impact

- **Incorrect Computation**: Signed high multiply instructions produce wrong results for negative products
- **Register Corruption**: Registers that should contain `-1` (0xFFFFFFFFFFFFFFFF) instead contain `0`
- **Differential Fuzzing Failure**: Mithril and Agave produce different register values after executing the same program
- **Program Behavior Divergence**: Programs using SHMUL instructions may behave differently on Mithril vs Agave

---

Files Modified

- `pkg/sbpf/interpreter.go` (modified)

Fix

Convert the `Int128` product to `Uint128` before shifting to perform a logical (unsigned) right shift that preserves the bit pattern:

```
case OpShmul64Imm:
    if !ip.sbpfVersion.EnablePqr() {
        err = ExcInvalidInstr
        break
    }
    // === START RV FIX ===
    // Fix: Int128.RShiftN uses sign-magnitude semantics which is incorrect.
    // We need to do a logical (unsigned) right shift on the 128-bit product.
    // Convert to Uint128 first to preserve the bit pattern, then shift.
    dst128 := wide.Int128FromInt64(int64(r[ins.Dst()]))
    imm128 := wide.Int128FromInt64(int64(ins.Imm()))
    r[ins.Dst()] = dst128.Mul(imm128).Uint128().RShiftN(64).Uint64()
    // === END RV FIX ===
    pc++
case OpShmul64Reg:
```

```
    if !ip.sbpfVersion.EnablePqr() {
        err = ExcInvalidInstr
        break
    }
    // === START RV FIX ===
    // Fix: Int128.RShiftN uses sign-magnitude semantics which is incorrect.
    // We need to do a logical (unsigned) right shift on the 128-bit product.
    // Convert to Uint128 first to preserve the bit pattern, then shift.
    dst128 := wide.Int128FromInt64(int64(r[ins.Dst()]))
    src128 := wide.Int128FromInt64(int64(r[ins.Src()]))
    r[ins.Dst()] = dst128.Mul(src128).Uint128().RShiftN(64).Uint64()
    // === END RV FIX ===
    pc++
```

The fix:

1. Computes the signed 128-bit product using `Int128.Mul` (correct)
2. Converts to `Uint128` using `.Uint128()` which preserves the bit pattern
3. Uses `Uint128.RShiftN(64)` for a logical (unsigned) right shift
4. Extracts the low 64 bits with `.Uint64()`

[F-D13] Missing `ExcUnsupportedInstruction` Error for Invalid Opcodes

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

Summary

Mithril's sBPF VM interpreter panics when encountering an unimplemented/invalid opcode instead of returning the `UnsupportedInstruction` error that Agave returns. This causes a crash in the fuzzer and represents divergent behavior from Agave.

Root Cause

In the interpreter's main switch statement, the `default` case uses `panic()` instead of returning a proper error:

Before Fix (`pkg/sbpf/interpreter.go`):

```
default:
    panic(fmt.Sprintf("unimplemented opcode %#02x", ins.Op()))
```

In Agave's interpreter (`sbpf/src/interpreter.rs`), the default case properly returns an error:

```
_ => throw_error!(self, EbpfError::UnsupportedInstruction),
```

Differential Behavior

Test Case: `crash-efec51314c1131e825fee0d48a35cb3858a50987`

Behavior	Agave	Mithril (before fix)
Result	error: 16 (UnsupportedInstruction)	panic: "unimplemented opcode 0x00"

Impact

Severity: Medium

1. **Crashes**: Any program with an invalid opcode causes a panic instead of graceful error handling
2. **Fuzzing**: Prevents differential fuzzing from completing - crashes appear as Mithril bugs rather than expected divergences
3. **Consensus Risk**: Different error handling could cause blockchain state divergence

Fix

1. Add `ExcUnsupportedInstruction` Error Type

File: `pkg/sbpf/vm.go`

```
// Exception codes.
var (
    ExcDivideByZero           = errors.New("division by zero")
    ExcDivideOverflow         = errors.New("divide overflow")
    ExcOutOfCU                = errors.New("compute unit overrun")
    ExcCallDepth              = errors.New("call depth exceeded")
    ExcInvalidInstr           = errors.New("invalid instruction - feature not enabled")
    ExcCallOutsideTextSegment = errors.New("invalid call destination")
    // === START RV FIX ===
    // Add ExcUnsupportedInstruction error for unimplemented/invalid opcodes
```

```
    // This matches Agave's EbpfError::UnsupportedInstruction
    ExcUnsupportedInstruction = errors.New("unsupported instruction")
    // === END RV FIX ===
)
```

2. Return Error Instead of Panicking

File: `pkg/sbpf/interpreter.go`

```
default:
    // === START RV FIX ===
    // Return ExcUnsupportedInstruction error instead of panicking
    // This matches Agave's behavior for unimplemented opcodes
    // See sbpf/src/interpreter.rs: _ => throw_error!(self, EbpfError::UnsupportedInstruction)
    err = ExcUnsupportedInstruction
    // === END RV FIX ===
```

3. Related Fix: PC Tracking for UnsupportedInstruction

This fix also requires updating the exception PC handling in `[F-D03] Incorrect Exception Handling in VM Interpreter`.

Since `ExcUnsupportedInstruction` is set in the `default` case where `pc++` is never executed, it must be added to the list of errors where `pcNotModified = true`:

```
isUnsupportedInstrError := err == ExcUnsupportedInstruction

// PC was NOT modified in these specific cases
pcNotModified := false
if isInvalidInstrError || isUnsupportedInstrError {
    // ExcInvalidInstr and ExcUnsupportedInstruction errors use 'break' which skips pc++
    pcNotModified = true
} else if ins.Op() == OpCall && (isCallDepthError || isCallDestError) {
    pcNotModified = true
} else if ins.Op() == OpCallx && (isCallDepthError || isCallOutsideTextSegmentError) {
    pcNotModified = true
}
```

Testing

```
cd ~/Workspace/_audits_Overclock-Validator_mithril
make -f Makefile.solfuzz

cd ~/Workspace/solfuzz
./build/fuzz_sol_vm_interp ./crash-efec51314c1131e825fee0d48a35cb3858a50987
```

After the fix, both implementations should return:

- error: 16 (UnsupportedInstruction)
- Matching PC values
- Matching CU consumption

Files Affected

- `pkg/sbpf/vm.go` - Added `ExcUnsupportedInstruction` error constant

- `pkg/sbpf/interpreter.go` - Changed `default` case from panic to error return; updated PC tracking (related to NEEDS_REVIEW-03)

Status

Addressed in pull request https://github.com/Overclock-Validator/mithril/pull/166.

[F-D14] Missing Padding Zeroing in `serializeParametersAligned`

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

Summary

Mithril's `serializeParametersAligned` function does not zero the padding bytes between account data and rent epoch, leaving uninitialized memory in the serialized output. This causes divergence with Agave's behavior, which explicitly zeros this padding area using `fill_write(MAX_PERMITTED_DATA_INCREASE + align_offset, 0)`.

Root Cause

In Agave's `program-runtime/src/serialization.rs`, when serializing account data, the padding area (used for potential realloc space and alignment) is explicitly zeroed:

```
// Agave serialization.rs line 137
self.fill_write(MAX_PERMITTED_DATA_INCREASE + align_offset, 0)
    .map_err(|_| InstructionError::InvalidArgument)?;
```

The `fill_write` function fills the specified number of bytes with the given value (0 in this case), ensuring the padding area is zeroed.

In Mithril's original `serializeParametersAligned` implementation, after copying account data, the code immediately writes the rent epoch without zeroing the intermediate padding bytes:

```
// Original Mithril code (before fix)
// data in account
copy(serializedData[l+88:l+88+len(borrowedAcct.Data())], borrowedAcct.Data())

// rent epoch (missing zeroing of padding between data and rent epoch)
var rentEpoch uint64
// ...
binary.LittleEndian.PutUint64(serializedData[len(serializedData)-8:], rentEpoch)
```

The serializedData slice is pre-allocated with the correct size (including padding), but the padding bytes are not explicitly zeroed. While Go normally zero-initializes slices created with `make()`, the issue arises due to the use of a memory arena that reuses previous allocations:

```
var serializedData []byte
if execCtx.SlotCtx.SerializedParameterArena != nil {
    arenaData, _ := execCtx.SlotCtx.SerializedParameterArena.AllocN(size)
    serializedData = arenaData[:0]
} else {
    serializedData = make([]byte, 0, size) // No arena configured - this is zero-initialized
}
```

When the arena is used (production code path), `AllocN()` returns a sub-slice of a pre-allocated pool. Although the arena's `Reset()` method clears the entire pool, it is only called at block boundaries (in `ProcessBlock()`), not between individual transactions. This means within a block, multiple transactions can write to and reuse arena memory, leaving residual data from previous serializations in the padding regions.

Impact

- **Non-deterministic Behavior in Production**: When using the memory arena (production code path), the uninitialized padding could contain different values across transactions within the same block, as the arena memory is reused without clearing between transactions
- **Potential Information Leak**: Residual data from previous memory allocations could potentially be exposed to programs that read beyond their data boundaries
- **Divergence from Agave**: Agave explicitly zeros this padding, ensuring deterministic behavior regardless of memory reuse patterns

---

Fix

Zero the padding bytes between account data and rent epoch:

File: `pkg/sealevel/bpf_loader.go`

```
// === START RV FIX ===
// Zero the padding bytes between account data and rent epoch.
// Agave explicitly uses fill_write(MAX_PERMITTED_DATA_INCREASE + align_offset, 0) to zero this area.
// Without this, the padding contains uninitialized memory, causing differential fuzzing mismatches.
paddingStart := l + 88 + len(borrowedAcct.Data())
paddingEnd := l + 88 + len(borrowedAcct.Data()) + int(numPaddingBytes)
for j := paddingStart; j < paddingEnd; j++ {
    serializedData[j] = 0
}
// === END RV FIX ===
```

Where `numPaddingBytes` is calculated as:

```
numPaddingBytes := ReallocSpace + util.AlignUp(dataLen, 8) - dataLen
```

This matches Agave's calculation of `MAX_PERMITTED_DATA_INCREASE + align_offset`.

[F-D15] Inconsistent Error Code for Empty ELF File

Severity: Informative | Recommended Action: Fix Code | Partially addressed by client

Summary

Mithril's ELF loader returns a different error code than Agave when encountering an empty ELF file. Mithril originally returned error code 1 (FailedToParse/EOF) while Agave returns error code 21 (ValueOutOfBounds). This discrepancy suggests different validation strategies between the implementations.

Root Cause

When given empty ELF data (0 bytes):

- **Agave's SBPF library (Rust)**: Performs bounds checking and returns `ValueOutOfBounds` error (code 21)
- **Mithril's loader (Go, before fix)**: Attempts to read the ELF header, gets EOF, and returns `FailedToParse` error (code 1)

The issue is in `pkg/sbpf/loader/parse.go` in the `readHeader()` function, which attempts to read the header without first validating that the file has sufficient data:

```
func (l *Loader) readHeader() error {
    var hdrBuf [ehLen]byte
    if _, err := io.ReadFull(io.NewSectionReader(l.rd, 0, ehLen), hdrBuf[:]); err != nil {
        return err  // Returns EOF for empty file
    }
    // ...
}
```

When `io.ReadFull` encounters an empty file, it returns `io.EOF`, which the fuzzing wrapper maps to error code 1 (FailedToParse) instead of code 21 (ValueOutOfBounds).

Underlying Architecture Issue: Lack of Typed Errors

This specific discrepancy highlights a more fundamental problem: **Mithril's SBPF loader doesn't use typed errors with error codes**.

As explained in [NEEDS_REVIEW-01], the client code uses `fmt.Errorf()` to return generic string errors, forcing the fuzzing harness to parse error messages with brittle string matching. This is why `io.EOF` gets mapped to the wrong error code - there's no type information to guide the correct mapping.

Agave (Rust) has explicit error types:

```
pub enum ElfError {
    FailedToParse(_),      // Code 1 - parsing failures
    ValueOutOfBounds,      // Code 21 - bounds/size issues
    // ... 21 more variants with explicit codes
}
```

Mithril (Go) returns untyped errors:

```
return err  // Just a generic error interface, no code information
```

The fuzzing harness tries to infer error codes from strings, but `io.EOF` doesn't contain keywords like "value out of bounds", so it defaults to code 1.

Proper solution: Implement typed errors in client code so the loader can return `ElfErrValueOutOfBounds` explicitly, eliminating the need for string parsing (see "Recommended Fix" section).

Differential Behavior

Test Case: Empty ELF data

- Crash file: `crash-9dc629324ba48fd8c6981b998a238ef64dacce2e`
- ELF field: Present but empty (length 0)
- Expected behavior: Both implementations should return the same error code

Agave Behavior (Rust):

```
error: 21  // ValueOutOfBounds
```

Mithril Behavior (Go, before fix):

```
error: 1   // FailedToParse (mapped from EOF)
```

This causes differential fuzzing to abort:

```
ELFLoaderEffects differ
Deploy checks: 0
==151740== ERROR: libFuzzer: deadly signal
```

Impact

Severity: Low

This is likely an **intentional design difference** rather than a critical bug:

1. **Semantic Equivalence**: Both error codes indicate the ELF file is invalid and cannot be loaded. The specific error code difference doesn't affect execution semantics.
2. **Error Reporting**: Different error codes provide slightly different diagnostic information, but both correctly reject the invalid input.
3. **Early vs Late Validation**: Agave performs bounds checking before attempting I/O, while Mithril attempts I/O and catches the error. Both are valid strategies.
4. **Fuzzing Impact**: This difference causes fuzzing to abort when comparing outputs, but doesn't indicate a consensus-critical bug.

Recommended Fix

Quick Fix (What We Implemented)

For immediate differential fuzzing compatibility, add early validation in `pkg/sbpf/loader/parse.go` :

File: `pkg/sbpf/loader/parse.go` (modified)

```go
func (l *Loader) readHeader() error {
    // === START RV FIX [NEEDS_REVIEW-02] ===
    // Check for empty file before attempting to read header
    // This matches Agave's behavior of returning ValueOutOfBounds for empty ELF
    if l.fileSize == 0 {
        return fmt.Errorf("value out of bounds: empty ELF file")
    }
    if l.fileSize < ehLen {
        return fmt.Errorf("value out of bounds: file too small for ELF header")
    }
```

```
    // === END RV FIX [NEEDS_REVIEW-02] ===

    var hdrBuf [ehLen]byte
    if _, err := io.ReadFull(io.NewSectionReader(l.rd, 0, ehLen), hdrBuf[:]); err != nil {
        return err
    }
    // ... rest of function
}
```

This ensures:

- **Early validation**: Checks file size before attempting I/O
- **Consistent error**: Returns "value out of bounds" which maps to error code 21
- **Matches Agave**: Provides the same error code for the same input

Error Mapping

The fuzzing wrapper already has the mapping for "value out of bounds" → error code 21:

```
// elfErrToNum in pkg/solfuzz/elf_loader.go
case strings.Contains(errStr, "value out of bounds") ||
    strings.Contains(errStr, "out of bounds") ||
    strings.Contains(errStr, "overlaps"):
    return 21
```

Alternative Approaches

1. Keep Current Behavior (No Fix)

- **Pros**: Simpler code, less validation overhead
- **Cons**: Differential fuzzing will report this as a difference
- **Recommendation**: Not suitable if exact Agave compatibility is required

2. Add Workaround in Fuzzing Harness Only

- **Pros**: No changes to client code
- **Cons**: Doesn't fix the underlying discrepancy
- **Code**: Special case in `executeElfLoader`:

```
if len(ctx.Elf.Data) == 0 {
    result.Error = 21  // ValueOutOfBounds
    return result
}
```

3. Implement Client Code Fix (Current Approach)

- **Pros**: Matches Agave exactly, cleaner error semantics
- **Cons**: Adds a small validation overhead
- **Impact**: Minimal performance impact, better compatibility

4. Proper Architectural Fix (Recommended Long-Term)

Implement typed errors with error codes in the client's SBPF loader. This would eliminate the need for fragile string matching in the fuzzing harness.

Step 1: Define typed error enum in client code:

```go
// pkg/sbpf/loader/errors.go (new file)
package loader

type ElfErrorCode int32

const (
    ElfErrFailedToParse              ElfErrorCode = 1
    ElfErrEntrypointOutOfBounds      ElfErrorCode = 2
    ElfErrInvalidEntrypoint          ElfErrorCode = 3
    ElfErrFailedToGetSection         ElfErrorCode = 4
    ElfErrUnresolvedSymbol           ElfErrorCode = 5
    ElfErrSectionNotFound            ElfErrorCode = 6
    ElfErrRelativeJumpOutOfBounds    ElfErrorCode = 7
    ElfErrSymbolHashCollision        ElfErrorCode = 8
    ElfErrWrongEndianess             ElfErrorCode = 9
    ElfErrWrongAbi                   ElfErrorCode = 10
    ElfErrWrongMachine               ElfErrorCode = 11
    ElfErrWrongClass                 ElfErrorCode = 12
    ElfErrNotOneTextSection          ElfErrorCode = 13
    ElfErrWritableSectionNotSupported ElfErrorCode = 14
    ElfErrAddressOutsideLoadableSection ElfErrorCode = 15
    ElfErrInvalidVirtualAddress      ElfErrorCode = 16
    ElfErrUnknownRelocation          ElfErrorCode = 17
    ElfErrFailedToReadRelocationInfo ElfErrorCode = 18
    ElfErrWrongType                  ElfErrorCode = 19
    ElfErrUnknownSymbol              ElfErrorCode = 20
    ElfErrValueOutOfBounds           ElfErrorCode = 21
    ElfErrUnsupportedSBPFVersion     ElfErrorCode = 22
    ElfErrInvalidProgramHeader       ElfErrorCode = 23
)

type ElfError struct {
    Code ElfErrorCode
    Msg  string
}

func (e *ElfError) Error() string {
    return e.Msg
}

func (e *ElfError) ErrorCode() int32 {
    return int32(e.Code)
}
```

Step 2: Update client code to return typed errors:

```go
// pkg/sbpf/loader/parse.go
func (l *Loader) readHeader() error {
    // Early validation with typed error
    if l.fileSize == 0 {
        return &ElfError{
            Code: ElfErrValueOutOfBounds,
            Msg:  "value out of bounds: empty ELF file",
        }
    }
    if l.fileSize < ehLen {
```

```
        return &ElfError{
            Code: ElfErrValueOutOfBounds,
            Msg:  fmt.Sprintf("value out of bounds: file too small for ELF header (need %d bytes, got %d)", ehLen, l.fileSize),
        }
    }

    var hdrBuf [ehLen]byte
    if _, err := io.ReadFull(io.NewSectionReader(l.rd, 0, ehLen), hdrBuf[:]); err != nil {
        // Wrap I/O errors as FailedToParse
        return &ElfError{
            Code: ElfErrFailedToParse,
            Msg:  fmt.Sprintf("failed to parse: %v", err),
        }
    }
    // ...
}
```

Step 3: Simplify fuzzing harness to use error codes directly:

```
// pkg/solfuzz/elf_loader.go
func elfErrToNum(err error) int32 {
    // Type-safe error code extraction
    if elfErr, ok := err.(*loader.ElfError); ok {
        return elfErr.ErrorCode()
    }
    // Fallback for non-ELF errors
    return -1
}
```

Benefits of typed errors:

- **Type safety**: Compile-time guarantees about error handling
- **No string matching**: Error codes are explicit, not inferred
- **Clear ownership**: Client code owns error semantics, not fuzzing harness
- **Maintainability**: Adding new errors forces updates throughout the codebase
- **Consensus-aware**: Client code knows which error codes matter for consensus
- **Agave compatibility**: Direct mapping to Agave's error enum

Why this matters for consensus:
The SBPF loader's error codes are part of the consensus protocol. Different error codes for the same input could lead to consensus divergence if validators disagree on transaction outcomes. Typed errors make this relationship explicit in the client code.

Files Affected

Client Code (Modified):

1. `pkg/sbpf/loader/parse.go` - Added early file size validation in `readHeader()`

Fuzzing Harness (No Changes):

- `pkg/solfuzz/elf_loader.go` - Error mapping already handles "value out of bounds" → code 21

Testing

The fix can be verified using the crash file:

```
cd ~/Workspace/solfuzz

# Test empty ELF data - both should return error code 21
./build/fuzz_sol_elf_loader crash-9dc629324ba48fd8c6981b998a238ef64dacce2e
```

Expected output:

```
Executed crash-9dc629324ba48fd8c6981b998a238ef64dacce2e in N ms
***
*** NOTE: fuzzing was not performed, you have only
***       executed the target code on a fixed set of inputs.
***
```

Verify error codes match by checking the dump:

```
cat /home/yiyiwang/Workspace/solfuzz_dump/msg_expected.txt  # Should show: error: 21
cat /home/yiyiwang/Workspace/solfuzz_dump/msg_actual.txt   # Should show: error: 21
```

Review Questions

1. **Is this error code difference intentional?** Should Mithril match Agave's error codes exactly, or are semantic differences acceptable?
2. **Performance trade-off**: Is the early validation worth the (minimal) overhead for every ELF load?
3. **Error semantics**: Is "ValueOutOfBounds" (21) more semantically correct than "FailedToParse" (1) for empty files?
4. **Compatibility requirements**: How strict is the requirement for exact Agave compatibility in differential fuzzing?

References

- **Agave Implementation**: Uses Rust SBPF library with bounds checking before I/O operations
- **Error Code 21**: Corresponds to `ElfError::ValueOutOfBounds` in Solana SBPF
- **Error Code 1**: Corresponds to `ElfError::FailedToParse` in Solana SBPF
- **Go io.ReadFull**: Returns `io.EOF` when reading from empty source

---

Status

Partially addressed in pull request https://github.com/Overclock-Validator/mithril/pull/165.

The client adapted the error message that is returned. However, the error code that is returned by Mithril was not changed and not adapted to be the same as Agave in case of an error.

[F-D16] SBPF Version Validation Order in ELF Loader

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

Summary

Mithril's ELF loader validates the SBPF version (from `e_flags`) **after** validating the ELF magic number, while Agave validates the SBPF version **before** any other ELF validation. This causes differential behavior where Mithril returns "not an ELF file" for garbage data, while Agave returns "UnsupportedSBPFVersion" (error code 22).

Root Cause

In Agave's ELF loader (`sbpf/src/elf.rs`), the SBPF version is checked immediately after reading the `e_flags` field at offset 48, **before** parsing any other ELF structures:

```
// sbpf/src/elf.rs:376-393
pub fn load(bytes: &[u8], loader: Arc<BuiltinProgram<C>>) -> Result<Self, ElfError> {
    const E_FLAGS_OFFSET: usize = 48;
    let e_flags = LittleEndian::read_u32(
        bytes
            .get(E_FLAGS_OFFSET..E_FLAGS_OFFSET.saturating_add(std::mem::size_of::<u32>()))
            .ok_or(ElfParserError::OutOfBounds)?,
    );
    let config = loader.get_config();
    let sbpf_version = match e_flags {
        0 => SBPFVersion::V0,
        1 => SBPFVersion::V1,
        2 => SBPFVersion::V2,
        3 => SBPFVersion::V3,
        4 => SBPFVersion::V4,
        _ => SBPFVersion::Reserved,
    };
    if !config.enabled_sbpf_versions.contains(&sbpf_version) {
        return Err(ElfError::UnsupportedSBPFVersion);  // Error code 22
    }
    // ... then continues with ELF parsing ...
}
```

Mithril's original code validated ELF magic first, then SBPF version:

```
// pkg/sbpf/loader/parse.go (before fix)
func (l *Loader) validateElfHeader() error {
    eh := &l.eh
    ident := &eh.Ident

    // ELF magic checked FIRST
    if string(ident[:elf.EI_CLASS]) != elf.ELFMAG {
        return fmt.Errorf("not an ELF file")
    }

    // SBPF version checked LATER
    if eh.Flags < l.minSbpfVersion || eh.Flags > l.maxSbpfVersion {
        return fmt.Errorf("invalid sbpf version")
    }
```

```
    // ...
}
```

Differential Behavior

Test Case: Garbage data with `e_flags` value outside valid SBPF versions

Agave Behavior:

- Reads `e_flags` at offset 48
- Finds invalid version (e.g., value > 4 and not 32)
- Returns `ElfError::UnsupportedSBPFVersion` (error code 22)

Mithril Behavior (before fix):

- Checks ELF magic first
- Returns "not an ELF file" error
- Never reaches SBPF version check

This causes differential fuzzing failures because the error codes don't match.

Fix

The fix reorders validation to check SBPF version before ELF magic, matching Agave's behavior:

File: `pkg/sbpf/loader/parse.go`

```
func (l *Loader) validateElfHeader() error {
    eh := &l.eh
    ident := &eh.Ident

    // === START RV FIX ===
    // Check SBPF version BEFORE checking ELF magic to match Agave's validation order
    // Agave returns UnsupportedSBPFVersion (22) before checking magic for invalid headers

    // First, reject any unreasonably large version value (garbage data)
    // Valid SBPF versions are 0, 1, 2, 3 (and EF_SBPF_V2 = 0x20 = 32)
    // Anything > 32 is definitely invalid
    if eh.Flags > 32 {
        return fmt.Errorf("invalid sbpf version")
    }

    if l.maxSbpfVersion != sbpfver.SbpfVersionV0 {
        if eh.Flags > l.maxSbpfVersion {
            return fmt.Errorf("invalid sbpf version")
        }
    } else {
        if eh.Flags == EF_SBPF_V2 {
            return fmt.Errorf("invalid sbpf version")
        }
    }

    if eh.Flags < l.minSbpfVersion {
        return fmt.Errorf("invalid sbpf version")
    }
    // === END RV FIX ===
```

```
    // Now check ELF magic
    if string(ident[:elf.EI_CLASS]) != elf.ELFMAG {
        return fmt.Errorf("not an ELF file")
    }
    // ...
}
```

Impact

Severity: Low

This is an error reporting order issue that only affects the specific error code returned for malformed inputs. It does not affect:

- Valid ELF file loading
- Program execution correctness
- Security of the loader

However, it causes differential fuzzing failures that can mask real bugs.

Files Affected

- `pkg/sbpf/loader/parse.go` - `validateElfHeader()` function

Agave Reference

- `sbpf/src/elf.rs:376-393` - `Executable::load()` checks `e_flags` before ELF parsing

[F-D17] Program Header Overlap Check for Empty Program Header Table

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

Summary

Mithril's ELF loader incorrectly rejects valid ELF files that have zero program headers (`e_phnum = 0`) or zero section headers (`e_shnum = 0`) because it checks for header table overlap with the file header even when the respective table is empty. Agave's range-based overlap checks naturally handle these cases correctly.

Note: While this fix ensures semantic equivalence with Agave's validation logic, a valid SBPF ELF **requires at least one program header** (the readable-executable segment) to construct the RAM image. ELF files with `e_phnum = 0` would fail at a later validation stage when checking for required program segments.

Root Cause

When `e_phnum = 0` or `e_shnum = 0` , the respective header table is empty. In this case:

- `e_phoff` / `e_shoff` can be 0 or any value since they reference an empty table
- There's no actual header data to overlap with anything

Agave's `sbpf/src/elf_parser/mod.rs` uses range-based overlap checking that naturally handles empty ranges:

```
// sbpf/src/elf_parser/mod.rs:120-128
fn check_that_there_is_no_overlap(
    range_a: &Range<usize>,
    range_b: &Range<usize>,
) -> Result<(), ElfParserError> {
    if range_a.end <= range_b.start || range_b.end <= range_a.start {
        Ok(())
    } else {
        Err(ElfParserError::Overlap)
    }
}
```

When `e_phnum = 0` , the program header table range is `e_phoff..(e_phoff + 0)` , which is an empty range and never overlaps with anything. The same applies to section headers when `e_shnum = 0` .

Agave uses this function for both program and section header overlap validation:

```
// sbpf/src/elf_parser/mod.rs:297-304 (program headers)
let program_header_table_range = file_header.e_phoff as usize
    ..mem::size_of::<Elf64Phdr>()
        .err_checked_mul(file_header.e_phnum as usize)?
        .err_checked_add(file_header.e_phoff as usize)?;
check_that_there_is_no_overlap(&file_header_range, &program_header_table_range)?;

// sbpf/src/elf_parser/mod.rs:314-319 (section headers)
let section_header_table_range = file_header.e_shoff as usize
    ..mem::size_of::<Elf64Shdr>()
        .err_checked_mul(file_header.e_shnum as usize)?
        .err_checked_add(file_header.e_shoff as usize)?;
check_that_there_is_no_overlap(&file_header_range, &section_header_table_range)?;
```

Mithril's original code unconditionally checked if offsets were within the file header region:

```
// pkg/sbpf/loader/parse.go (before fix)
func (l *Loader) validateElfHeader() error {
    // ...
    if eh.Phoff < ehLen {
        return fmt.Errorf("program header overlaps with file header")
    }
    if eh.Shoff < ehLen {
        return fmt.Errorf("section header overlaps with file header")
    }
    // ...
}
```

This incorrectly rejects files where:

- `e_phnum = 0` and `e_phoff = 0` (a valid combination for ELF, though not for SBPF)
- `e_shnum = 0` and `e_shoff = 0` (a valid combination)

Differential Behavior

Test Case: Valid ELF file with `e_phnum = 0` and `e_phoff = 0`

Agave Behavior:

- Calculates program header range: `0..0` (empty)
- Empty range never overlaps → passes validation
- Continues to parse ELF (would fail later for SBPF validation)

Mithril Behavior (before fix):

- Checks `if e_phoff (0) < ehLen (64)`
- Returns "program header overlaps with file header"
- Rejects file with wrong error

Fix

The fix adds checks for `Phnum > 0` and `Shnum > 0` before checking their respective offsets:

File: `pkg/sbpf/loader/parse.go`

```
func (l *Loader) validateElfHeader() error {
    // ...

    // === START RV FIX ===
    // Only check program header offset when there are program headers (Phnum > 0).
    // When Phnum = 0, there are no program headers, so Phoff = 0 is valid.
    // This matches Agave's sbpf behavior which uses range-based overlap checks
    // that naturally handle the empty program header case.
    // Note: A valid SBPF ELF requires at least one program header for the
    // readable-executable segment; e_phnum = 0 would fail later validation.
    if eh.Phnum > 0 && eh.Phoff < ehLen {
        return fmt.Errorf("program header overlaps with file header")
    }
    // Only check section header offset when there are section headers (Shnum > 0).
    // When Shnum = 0, there are no section headers, so Shoff = 0 is valid.
    if eh.Shnum > 0 && eh.Shoff < ehLen {
```

```
        return fmt.Errorf("section header overlaps with file header")
    }
    // === END RV FIX ===

    // ...
}
```

Impact

Severity: Low

This bug causes Mithril to reject ELF files with empty header tables using incorrect error messages. While rare in practice and not affecting valid SBPF programs (which require program headers), this is a correctness issue that ensures:

- Semantic equivalence with Agave's validation logic
- Correct error messages for malformed files
- Consistent behavior in differential fuzzing

Files Affected

- `pkg/sbpf/loader/parse.go` - `validateElfHeader()` function

Agave Reference

- `sbpf/src/elf_parser/mod.rs:120-128` - `check_that_there_is_no_overlap()` handles empty ranges
- `sbpf/src/elf_parser/mod.rs:297-304` - Program header table range calculation
- `sbpf/src/elf_parser/mod.rs:314-319` - Section header table range calculation

[F-D18] Incorrect Error Code for Program Loading Failures

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

Summary

Mithril was returning `InstrErrInvalidAccountData` (error code 3) when program loading failed during BPF program invocation, while Agave returns `InstrErrUnsupportedProgramId` (error code 30). This caused a divergence in error codes between the two implementations for the same transaction.

Affected Code

File: `pkg/sealevel/bpf_loader.go`
Function: `BpfLoaderProgramExecute`

Root Cause

In Agave, program loading is handled through a **program cache**. When a BPF program is invoked:

1. Agave looks up the program in `program_cache_for_tx_batch`
2. If the program is not found or has an invalid state (FailedVerification, Closed, DelayVisibility), Agave returns `InstructionError::UnsupportedProgramId`

In Mithril, program loading happens on-the-fly during execution:

1. When parsing the upgradeable loader state fails, Mithril was returning `InstrErrInvalidAccountData`
2. When programdata is uninitialized, Mithril was returning `InstrErrInvalidAccountData`
3. When the program's deployment slot >= current slot, Mithril was returning `InstrErrInvalidAccountData`

This caused a divergence because Agave's program cache abstraction maps all these failures to a single error code (`UnsupportedProgramId`), while Mithril's on-the-fly loading was returning different error codes based on the specific failure reason.

Agave Implementation

In Agave's `programs/bpf_loader/src/lib.rs` , the program cache lookup and invocation is handled as:

```
// Program Invocation
let executor = invoke_context
    .program_cache_for_tx_batch
    .find(program_id)
    .ok_or_else(|| {
        ic_logger_msg!(log_collector, "Program is not cached");
        InstructionError::UnsupportedProgramId  // ← All cache misses return UnsupportedProgramId
    })?;

match &executor.program {
    ProgramCacheEntryType::FailedVerification(_)
    | ProgramCacheEntryType::Closed
    | ProgramCacheEntryType::DelayVisibility => {
        ic_logger_msg!(log_collector, "Program is not deployed");
        Err(Box::new(InstructionError::UnsupportedProgramId) as Box<dyn std::error::Error>)
    }
    ProgramCacheEntryType::Loaded(executable) => execute(executable, invoke_context),
```

```
    _ => Err(Box::new(InstructionError::UnsupportedProgramId) as Box<dyn std::error::Error>),
}
```

Key observations:

1. If the program is not in the cache at all → `UnsupportedProgramId`
2. If the program has `FailedVerification` (bad data/verification failed) → `UnsupportedProgramId`
3. If the program is `Closed` (closed by owner) → `UnsupportedProgramId`
4. If the program has `DelayVisibility` (deployed in current slot) → `UnsupportedProgramId`
5. Any other non-Loaded state → `UnsupportedProgramId`

The Bug

Mithril was returning different error codes for various program loading failures:

Before (incorrect):

```
// When UnmarshalUpgradeableLoaderState fails (from accountsdb path):
programAcctState, err = UnmarshalUpgradeableLoaderState(paTmp.Data)
if err != nil {
    return err  // Returns InstrErrInvalidAccountData
}

// When UnmarshalUpgradeableLoaderState fails (from direct account data):
programAcctState, err = UnmarshalUpgradeableLoaderState(programAcct.Data())
if err != nil {
    return err  // Returns InstrErrInvalidAccountData
}

// When deployment slot check fails:
if programCacheEntry.DeploymentSlot >= execCtx.SlotCtx.Slot {
    return InstrErrInvalidAccountData  // Incorrect error code
}

// When programdata unmarshal fails:
programDataAcctState, err := UnmarshalUpgradeableLoaderState(programDataAcct.Data)
if err != nil {
    return err  // Returns InstrErrInvalidAccountData
}

// When programdata is uninitialized:
if programDataAcctState.Type == UpgradeableLoaderStateTypeUninitialized {
    return InstrErrInvalidAccountData  // Incorrect error code
}

// When programdata slot check fails:
if programDataSlot >= execCtx.SlotCtx.Slot {
    return InstrErrInvalidAccountData  // Incorrect error code
}
```

After (correct):

```
// When UnmarshalUpgradeableLoaderState fails (from accountsdb path):
programAcctState, err = UnmarshalUpgradeableLoaderState(paTmp.Data)
if err != nil {
    // Return UnsupportedProgramId when program account state is invalid,
```

```
    // matching Agave's behavior where program cache lookup would fail.
    return InstrErrUnsupportedProgramId
}

// When UnmarshalUpgradeableLoaderState fails (from direct account data):
programAcctState, err = UnmarshalUpgradeableLoaderState(programAcct.Data())
if err != nil {
    // Return UnsupportedProgramId when program account state is invalid,
    // matching Agave's behavior where program cache lookup would fail.
    return InstrErrUnsupportedProgramId
}

// When deployment slot check fails:
if programCacheEntry.DeploymentSlot >= execCtx.SlotCtx.Slot {
    // Return UnsupportedProgramId when program was deployed after current slot,
    // matching Agave's behavior where program cache would return DelayVisibility.
    return InstrErrUnsupportedProgramId
}

// When programdata unmarshal fails:
programDataAcctState, err := UnmarshalUpgradeableLoaderState(programDataAcct.Data)
if err != nil {
    // Return UnsupportedProgramId when programdata account state is invalid,
    // matching Agave's behavior where program cache lookup would fail.
    return InstrErrUnsupportedProgramId
}

// When programdata is uninitialized:
if programDataAcctState.Type == UpgradeableLoaderStateTypeUninitialized {
    // Return UnsupportedProgramId when programdata is uninitialized,
    // matching Agave's behavior where program cache lookup would fail.
    return InstrErrUnsupportedProgramId
}

// When programdata slot check fails:
if programDataSlot >= execCtx.SlotCtx.Slot {
    // Return UnsupportedProgramId when programdata slot >= current slot,
    // matching Agave's behavior where program cache would return DelayVisibility.
    return InstrErrUnsupportedProgramId
}
```

Differential Fuzzing Discovery

Crash file: `crash-5a18830244a3cb6b9e5df1754b01931646053a4d`

Divergence observed:

- **Agave**: `InstructionError(3, UnsupportedProgramId)` → instruction error 31 (30 + 1)
- **Mithril**: `InstrErrInvalidAccountData` → instruction error 4 (3 + 1)

Both implementations failed on the same instruction (index 3), but returned different error codes.

Impact

1. **Consensus Divergence**: Different error codes mean validators would report different transaction results
2. **Client Compatibility**: Applications relying on specific error codes for error handling would behave differently

3. **Block Production**: During block production, different error codes could affect transaction inclusion decisions

[F-D19] Incorrect PC Value on OpExit Stack Pop

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

Summary

Mithril's sBPF VM interpreter incorrectly sets the program counter (PC) to 0 when a program exits from the top-level function, instead of preserving the PC pointing to the `exit` instruction. This causes divergence from Agave's behavior in the final register state.

Root Cause

The bug is in the `OpExit` handler in `pkg/sbpf/interpreter.go`. When the program exits from the top-level function (stack is empty), the `Pop()` method returns `0, false`, and this `0` value overwrites the `pc` variable:

Original client code:

```
case OpExit:
    var ok bool
    pc, ok = ip.stack.Pop(r[:])  // BUG: pc is set to 0 when stack is empty
    if !ok {
        ret = r[0]
        break mainLoop  // pc is now 0, not the exit instruction's address!
    }
```

The `stack.Pop()` function returns `0, false` when the stack is empty:

```
func (s *Stack) Pop(r []uint64) (pc int64, ok bool) {
    if len(s.shadow) <= 1 {
        return  // Returns 0, false (zero values)
    }
    // ...
}
```

This means:

1. Program executes normally and reaches the `exit` instruction at some PC (e.g., PC=4)
2. `OpExit` calls `stack.Pop()` which returns `0, false` for the top-level exit
3. `pc` is assigned `0` from the return value before checking `!ok`
4. The `break mainLoop` exits with `pc=0` instead of the correct `pc=4`
5. Final register state has wrong PC value

Differential Behavior

Test Case: VM execution crash file `crash-61541668b6dcd10a023d179cbda2a63eeaff6401`

- Program executes 5 instructions (PC 0 through 4)
- Instruction at PC=4 is `exit` (opcode `0x95`)

Agave Behavior (Rust):

```
pc: 4
```

Mithril Behavior (Go, before fix):

```
(pc not output, meaning pc=0)
```

After applying our fix:

```
pc: 4  (matches Agave)
```

Protobuf comparison shows the divergence:

```
Expected (Agave): pc: 4
Actual (Mithril): (no pc field - defaults to 0)
```

Impact

Severity: Low-Medium

1. Register State Mismatch

- The final PC value in the register trace is incorrect
- This affects any code that relies on the final register state after VM execution

2. Differential Fuzzing Failures

- Causes assertion failures in differential fuzzing tests
- The protobuf output differs between implementations

3. Debugging and Tracing

- When tracing program execution, the final PC doesn't point to the exit instruction
- Makes it harder to understand program execution flow

Proof of Concept

The bug was discovered through differential fuzzing against Agave:

```
cd ~/Workspace/solfuzz
./build/fuzz_sol_vm_interp crash-61541668b6dcd10a023d179cbda2a63eeaff6401
```

Before fix:

```
Assertion failed:
  Expected: pc: 4
  Actual:   (missing pc field)
```

Instructions in the program:

```
[MITHRIL VM]   Instruction 0: opcode=0x14
[MITHRIL VM]   Instruction 1: opcode=0x46
[MITHRIL VM]   Instruction 2: opcode=0x4c
[MITHRIL VM]   Instruction 3: opcode=0x64
[MITHRIL VM]   Instruction 4: opcode=0x95  <-- exit instruction
```

Recommended Fix

Preserve the current PC value before calling `stack.Pop()`, and restore it when the stack is empty (top-level exit):

```
case OpExit:
    // === START RV FIX ===
    // When exiting from top-level, preserve the current PC (pointing to the exit instruction)
    // so that the final register state matches Agave's behavior.
    // Pop modifies pc, so we need to save it first.
    exitPC := pc
```

```
    var ok bool
    pc, ok = ip.stack.Pop(r[:])
    if !ok {
        ret = r[0]
        pc = exitPC // Restore PC to point to the exit instruction
        break mainLoop
    }
    // === END RV FIX ===
```

Testing

The fix can be verified using the differential fuzzing harness:

```
cd ~/Workspace/_audits_Overclock-Validator_mithril
make -f Makefile.solfuzz  # Rebuild with fix

cd ~/Workspace/solfuzz
./build/fuzz_sol_vm_interp ./crash-61541668b6dcd10a023d179cbda2a63eeaff6401
```

Expected output (after fix):

```
Executed crash-61541668b6dcd10a023d179cbda2a63eeaff6401 in N ms
***
*** NOTE: fuzzing was not performed, you have only
***       executed the target code on a fixed set of inputs.
***
```

Both implementations should now report the same PC value.

Files Affected

Client Code (Fixed):

1. **`pkg/sbpf/interpreter.go`** - `OpExit` handler in `RunWithRegisters()` method
   - Save PC before calling `stack.Pop()`
   - Restore PC when exiting from top-level function

Comparison with Agave

In Agave's SBPF VM interpreter:

- Register state is captured to `register_trace` BEFORE each instruction executes
- `register_trace.last()` returns the register state from the last executed instruction
- This naturally includes the correct PC value (pointing to the exit instruction)

Mithril's approach of modifying `pc` via the return value of `Pop()` causes the PC to be lost for top-level exits.

References

- **Agave VM Interpreter**: `solfuzz-agave/src/vm_interp.rs`
  - Lines 378-381: Using `register_trace.last()` for final register state
- **Mithril Stack Implementation**: `pkg/sbpf/stack.go`
  - `Pop()` returns zero values when stack is empty

- **Differential Fuzzing Test**: `crash-61541668b6dcd10a023d179cbda2a63eeaff6401`

Informative Findings

The following findings do not impact the protocol's current behavior but either relate to documentation or could lead to potential problems in future protocol expansions.

[B01] Unnecessary input validation on flags `endSlot` and `numReplaySlots`

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

Description

The function `runVerifier` can receive two parameters to specify the range of slots to verify:

```
numReplaySlots    int64
endSlot           int64
```

The function enforces that at most one flag is provided:

runtimeverification/_audits_Overclock-Validator_mithril/cmd/mithril/node/node.go

Line 129 to 131 in 06f7b95

```
129    if endSlot != -1 && numReplaySlots != 0 {
130        klog.Fatalf("specify at most one of --endslot and --num-replay-slots")
131    }
```

However, later these values are updated as following:

runtimeverification/_audits_Overclock-Validator_mithril/cmd/mithril/node/node.go

Line 218 to 222 in 06f7b95

```
218    if endSlot != -1 {
219        numReplaySlots = endSlot - startSlot
220    } else if numReplaySlots != 0 {
221        endSlot = startSlot + numReplaySlots
222    }
```

If both flags are provided, then the `endSlot` flag would take priority and the `numReplaySlots` would be replaced with `endSlot - startSlot`, therefore, the enforcing is not necessary.

[B02] Function parameter `rpcServer` is never used in `replayBlocks` function

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

Description

The `replayBlocks` function receives as argument the `rpcServer *rpcserver.RpcServer`, however this argument is never used in the function.

runtimeverification/_audits_Overclock-Validator_mithril/pkg/replay/block.go

Line 491 to 505 in 06f7b95

```
491 func ReplayBlocks(
492   ctx context.Context,
493   acctsDb *accountsdb.AccountsDb,
494   acctsDbPath string,
495   snapshotManifest *snapshot.SnapshotManifest,
496   startSlot, endSlot uint64,
497   rpcEndpoint string,
498   blockDir string,
499   txParallelism int,
500   blockChan chan *b.Block,
501   isLive bool,
502   dbgOpts *DebugOptions,
503   metricsWriter io.Writer,
504   rpcServer *rpcserver.RpcServer,
505 ) error {
```

Recommendation

If the argument is being passed for compatibility with other implementations or for future feature implementation document accordingly.

[B03] Optimization in `VoteStateVersions` struct

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

Description

The `VoteStateVersions` struct currently embeds all three vote state versions directly:

```
type VoteStateVersions struct {
    Type     uint32
    V0_23_5  VoteState0_23_5
    V1_14_11 VoteState1_14_11
    Current  VoteState
}
```

Each `VoteStateVersions` instance allocates space for all three structs simultaneously, even though only one is actually used (determined by the `Type` field). The total memory footprint is the sum of all three structs, which is substantial given the dynamic data structures (deques, slices, btrees) they contain.

Recommendation

Using pointers would significantly reduce memory usage. Instead of allocating space for all three, it would allocate only 32 bytes (4 bytes for Type + 3 × 8-byte pointers on 64-bit systems) plus the size of whichever single struct is actually used.

```
type VoteStateVersions struct {
    Type     uint32
    V0_23_5  *VoteState0_23_5
    V1_14_11 *VoteState1_14_11
    Current  *VoteState
}
```

Status

Some trade-offs likely explain the current implementation:

1. Unmarshaling Simplicity
2. No Nil Checks Required

The current design prioritizes simplicity and safety over memory optimization, since it avoids potential nil pointer panics in critical code. Vote accounts are relatively few compared to other account types (there are only thousands of validators, not millions), so the memory overhead is acceptable.

This issue should be addressed if memory optimization becomes a priority.

[B04] Remove code duplication logic in `setVoteAccountState` function

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

Description

The `setVoteAccountState` function in `pkg/sealevel/vote_state.go` has significant code duplication that could be refactored. The same logic for converting to V1_14_11 format and marshaling appears in two places:

- When a resize is needed but fails or isn't rent-exempt
- When the `VoteStateAddVoteLatency` feature is not active

Recommendation

The function could be refactored to extract this common logic:

```
func setVoteAccountState(acct *BorrowedAccount, voteState *VoteState, f features.Features) error {
    var err error

    // Helper function for V1_14_11 serialization
    setV1_14_11State := func() error {
        newVoteState := newVoteState1_14_11FromCurrent(voteState)
        newVoteStateVersioned := new(VoteStateVersions)
        newVoteStateVersioned.Type = VoteStateVersionV1_14_11
        newVoteStateVersioned.V1_14_11 = *newVoteState
        voteStateBytes, err := marshalVersionedVoteState(newVoteStateVersioned)
        defer voteStateBufPool.Put(voteStateBytes)
        if err != nil {
            return err
        }
        return acct.SetState(f, voteStateBytes)
    }

    if f.IsActive(features.VoteStateAddVoteLatency) {
        vsz := VoteStateV3Size
        resizeNeeded := len(acct.Data()) < vsz
        resizeRentExempt := acct.IsRentExemptAtDataLength(uint64(vsz))
        resizeFailed := false

        if resizeNeeded && resizeRentExempt {
            err = acct.SetDataLength(VoteStateV3Size, f)
            if err != nil {
                resizeFailed = true
            }
        }

        // Use V1_14_11 format if resize needed but failed or not rent-exempt
        if resizeNeeded && (!resizeRentExempt || resizeFailed) {
            return setV1_14_11State()
        }

        // Use current format
        newCurrent := new(VoteStateVersions)
```

```
        newCurrent.Type = VoteStateVersionCurrent
        newCurrent.Current = *voteState
        voteStateBytes, err := marshalVersionedVoteState(newCurrent)
        defer voteStateBufPool.Put(voteStateBytes)
        if err != nil {
            return err
        }
        return acct.SetState(f, voteStateBytes)
    } else {
        // Feature not active - use V1_14_11 format
        return setV1_14_11State()
    }
}
```

This refactoring eliminates the duplication while maintaining the same logic and behavior.
The current implementation works correctly, but violates the DRY (Don't Repeat Yourself) principle. The refactored version would be more maintainable - any future changes to the V1_14_11 serialization logic would only need to be made in one place.

[B05] Unauthenticated JSON-RPC Service Bound to All Interfaces

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

Description

The RPC server is initialized to listen on all network interfaces (0.0.0.0) and registers all exported methods on the RpcServer struct as JSON-RPC endpoints without any authentication or authorization. This allows any remote client to invoke RPC methods, potentially leading to unauthorized data access or operations.

Exposed RPC Methods

The following methods are exposed without any access control:

1. `getBankHash`
2. `getEpochInfo`
3. `getAccountInfo`

Security Implications

The lack of authentication means:

- Any remote client can query account data, bank hashes, and epoch information
- There's no rate limiting or access control

Mitigation Context

However, it's important to note:

- The RPC server is disabled by default - it only starts when `--rpc-server-port` is explicitly set to a non-zero value
- The exposed methods are read-only - they only query state from `AccountsDb` and don't perform any write operations or state modifications
- This is a verifier node, not a validator - Mithril is designed for state verification and auditing, not consensus participation, so the security model may intentionally prioritize data accessibility over access control

Recommendation

When deploying Mithril in a production environment, it is ideal to restrict the RPC listener to trusted interfaces or configure a firewall to ensure security. Consider adding middleware for request validation and enabling TLS to encrypt communications.

[B06] Missing `Destroy` calls of grocksdb leading to memory leak

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

Description

The Mithril code base, particularly code in `pkg/blockstore`, makes use of the Go library `grocksdb`. This library uses FFI calls into a C library and therefore requires the Go user to Destroy allocated objects with the respective `Destroy` methods.

The Mithril code base does not consistently call `Destroy` on `grocksdb` objects such as e.g. in the function `GetBincode` at `pkg/blockstore/bincode_rocks.go` for the `grocksdb` object `opts := grocksdb.NewDefaultReadOptions()`:

```
func GetBincode[T any](db *grocksdb.DB, cf *grocksdb.ColumnFamilyHandle, key []byte) (*T, error) {
    opts := grocksdb.NewDefaultReadOptions()
    res, err := db.GetCF(opts, cf, key)
    if err != nil {
        return nil, err
    }
    if !res.Exists() {
        return nil, ErrNotFound
    }
    defer res.Free()
    return ParseBincode[T](res.Data())
}
```

This can lead to memory leaks that can cause Mithril to shut-down due to memory exhaustion and impacts the availability of Mithril. Though it currently seems that the code in the `pkg/blockstore` package is only used by the tool `radiance` that is also built by the Mithril project, which is why this finding was classified as an informational finding.

Recommendation

We recommend to refactor the code that uses `grocksdb` to properly clean-up memory by calling `Destroy` on all objects. The use of custom static code analysis rules may help to identify these missing `.Destroy()` invocations in the code base. An alternative could be to separate code unused by Mithrils core tooling into their own code base and repository.

Status

To be discussed with the client

[B07] Unbounded transaction count allows potential memory exhaustion

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

Description

In the function `UnmarshalWithDecoder` in `pkg/shred/entry.go`, the number of transactions is read from untrusted input and used directly to allocate the `Txns` slice without enforcing a proper upper bound. The code only verifies that `numTxns` does not exceed the number of remaining bytes, which is a unit mismatch; as a result, an attacker could potentially specify a larger `numTxns` that still satisfies this check, causing excessive memory allocation and a potential denial-of-service due to memory exhaustion.

```
// read the number of transactions:
numTxns, err := decoder.ReadUint64(bin.LE)
if err != nil {
    return fmt.Errorf("failed to read number of transactions: %w", err)
}
if numTxns > uint64(decoder.Remaining()) {
    return fmt.Errorf("not enough bytes to read %d transactions", numTxns)
}
// read the transactions:
en.Txns = make([]solana.Transaction, numTxns)
for i := uint64(0); i < numTxns; i++ {
    if err = en.Txns[i].UnmarshalWithDecoder(decoder); err != nil {
        return fmt.Errorf("failed to read transaction %d: %w", i, err)
    }
}
return
```

Recommendation

We recommend to enforce a strict upper bound on `numTxns` based on protocol limits. Additionally, validate that `numTxns * estimatedTransactionSize` does not exceed `decoder.Remaining()`, and reject values outside expected ranges.

Status

To be discussed with the client

[B08] Inconsistent logging and error handling

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

Description

Mithril makes use of different kinds of loggers.

For logging:

- `klog.Warning`
- `klog.Info`
- `mlog.Log.Infof`
- `mlog.Log.Errorf`
- `fmt.Println`
- `fmt.Printf`

For immediate program exits, the following functions are utilized:

- `panic`
- `klog.Fatalf`
- `klog.Exit`
- some logging command combined with `os.Exit(1)`

This broad set of functions and libraries makes the use of logging very inconsistent, especially in regards to the user experience, where the user sets a logging level as part of the CLI configuration.

Additionally, the use of generic `fmt.Errof()` strings for errors forces the fuzzing harness to parse error messages with fragile string matching. This architectural decision creates maintenance burden, reduces type safety, and makes consensus compatibility harder to verify.

Current State: String-Based Errors

Agave (Rust) - Strongly Typed

Agave uses a `ElfError` enum with 23 variants, each mapping to a specific error code:

```
// solana-sbpf/src/elf.rs (conceptually)
pub enum ElfError {
    FailedToParse(String),           // Code 1
    EntrypointOutOfBounds,           // Code 2
    InvalidEntrypoint,               // Code 3
    FailedToGetSection(String),      // Code 4
    UnresolvedSymbol(String, u64, String), // Code 5
    // ... 18 more variants ...
    ValueOutOfBounds,                // Code 21
    UnsupportedSBPFVersion,          // Code 22
    InvalidProgramHeader,            // Code 23
}

// Explicit error code mapping in solfuzz-agave
pub fn elf_err_to_num(error: &ElfError) -> i32 {
    match error {
        ElfError::FailedToParse(_) => 1,
        ElfError::EntrypointOutOfBounds => 2,
```

```
        // ... explicit mapping for all 23 variants
        ElfError::ValueOutOfBounds => 21,
        ElfError::UnsupportedSBPFVersion => 22,
        ElfError::InvalidProgramHeader => 23,
    }
}
```

Mithril (Go) - Generic Strings

Mithril uses `fmt.Errorf()` throughout, with no error code information:

```
// pkg/sbpf/loader/parse.go (examples)
func (l *Loader) readHeader() error {
    if _, err := io.ReadFull(...); err != nil {
        return err  // Generic error interface
    }
    // ...
}

func (l *Loader) validateElfHeader() error {
    if string(ident[:elf.EI_CLASS]) != elf.ELFMAG {
        return fmt.Errorf("not an ELF file")  // String only
    }
    if eh.Type != elf.ET_DYN {
        return fmt.Errorf("wrong type")  // String only
    }
    // ... more fmt.Errorf calls
}
```

Fuzzing Harness - Brittle String Matching

The fuzzing harness must parse error strings to infer codes:

```
// pkg/solfuzz/elf_loader.go
func elfErrToNum(err error) int32 {
    if err == nil {
        return 0
    }
    errStr := err.Error()

    // 23 fragile string matches
    if strings.Contains(errStr, "entrypoint out of bounds") {
        return 2
    }
    if strings.Contains(errStr, "invalid entrypoint") {
        return 3
    }
    // Problem: Multiple errors can match the same substring!
    if strings.Contains(errStr, "value out of bounds") ||
       strings.Contains(errStr, "out of bounds") ||
       strings.Contains(errStr, "overlaps") {
        return 21  // Could match error code 2, 7, or 21!
    }
    // ... 20 more string matches ...

    // Unknown error - breaks differential fuzzing
```

```
    return -1
}
```

Problems with Current Approach

1. Brittleness

Error messages can change without breaking compilation, causing fuzzing to fail:

```
// Before
return fmt.Errorf("value out of bounds")  // Maps to code 21

// After refactoring (accidentally)
return fmt.Errorf("bounds exceeded")  // Now maps to -1 (unknown)
```

2. Ambiguity

Multiple errors contain similar substrings:

- "entrypoint out of bounds" (code 2)
- "value out of bounds" (code 21)
- "relative jump out of bounds" (code 7)

Current string matching uses `strings.Contains()`, which can match incorrectly.

3. No Compile-Time Safety

Adding a new error to the loader doesn't force updates to the fuzzing harness:

```
// New error added to loader
return fmt.Errorf("section alignment invalid")

// Fuzzing harness unchanged - maps to -1 (unknown error)
// No compiler error, silent breakage
```

4. Client Code Unaware of Consensus Impact

The loader has no knowledge that error codes are part of consensus protocol:

```
// Developer thinks this is just for debugging
return fmt.Errorf("not an ELF file")

// Actually affects consensus: error code must match Agave exactly
```

5. Maintenance Burden

Fuzzing harness must be manually kept in sync with loader error messages across 23 error types.

6. Testing Difficulty

Can't easily test that specific errors return specific codes without running full fuzzing infrastructure.

Proposed Solution: Typed Errors

Architecture

```
┌──────────────────────────────────┐
│  Client Code (Mithril SBPF Loader) │
│                                  │
│  Returns: *loader.ElfError       │
│  - Code: ElfErrorCode (1-23)     │
```

```
|  - Msg:  string                   |
└──────────────┬────────────────────┘
               | Typed error with explicit code
               |
               ▼

┌──────────────────────────────────┐
|  Fuzzing Harness                 |
|                                  |
|  elfErr.ErrorCode() → int32      |
|  (No string parsing needed)      |
└──────────────────────────────────┘
```

Implementation Plan

Step 1: Define Error Types (New File)

Create `pkg/sbpf/loader/errors.go`:

```go
package loader

type ElfErrorCode int32

// Error codes matching Agave's ElfError enum
// See: solfuzz-agave/src/utils/err_map.rs
const (
    ElfErrFailedToParse              ElfErrorCode = 1
    ElfErrEntrypointOutOfBounds      ElfErrorCode = 2
    ElfErrInvalidEntrypoint          ElfErrorCode = 3
    ElfErrFailedToGetSection         ElfErrorCode = 4
    ElfErrUnresolvedSymbol           ElfErrorCode = 5
    ElfErrSectionNotFound            ElfErrorCode = 6
    ElfErrRelativeJumpOutOfBounds    ElfErrorCode = 7
    ElfErrSymbolHashCollision        ElfErrorCode = 8
    ElfErrWrongEndianess             ElfErrorCode = 9
    ElfErrWrongAbi                   ElfErrorCode = 10
    ElfErrWrongMachine               ElfErrorCode = 11
    ElfErrWrongClass                 ElfErrorCode = 12
    ElfErrNotOneTextSection          ElfErrorCode = 13
    ElfErrWritableSectionNotSupported ElfErrorCode = 14
    ElfErrAddressOutsideLoadableSection ElfErrorCode = 15
    ElfErrInvalidVirtualAddress      ElfErrorCode = 16
    ElfErrUnknownRelocation          ElfErrorCode = 17
    ElfErrFailedToReadRelocationInfo ElfErrorCode = 18
    ElfErrWrongType                  ElfErrorCode = 19
    ElfErrUnknownSymbol              ElfErrorCode = 20
    ElfErrValueOutOfBounds           ElfErrorCode = 21
    ElfErrUnsupportedSBPFVersion     ElfErrorCode = 22
    ElfErrInvalidProgramHeader       ElfErrorCode = 23
)

// ElfError represents an ELF loading error with an explicit error code
// for consensus compatibility with Agave.
type ElfError struct {
    Code ElfErrorCode
    Msg  string
}
```

```go
func (e *ElfError) Error() string {
    return e.Msg
}

func (e *ElfError) ErrorCode() int32 {
    return int32(e.Code)
}

// Helper constructors for common error patterns
func NewElfError(code ElfErrorCode, msg string) *ElfError {
    return &ElfError{Code: code, Msg: msg}
}

func NewElfErrorf(code ElfErrorCode, format string, args ...interface{}) *ElfError {
    return &ElfError{
        Code: code,
        Msg:  fmt.Sprintf(format, args...),
    }
}
```

Step 2: Update Loader Code

Modify `pkg/sbpf/loader/parse.go` to return typed errors:

```go
func (l *Loader) readHeader() error {
    // Early validation with typed errors
    if l.fileSize == 0 {
        return NewElfError(ElfErrValueOutOfBounds, "value out of bounds: empty ELF file")
    }
    if l.fileSize < ehLen {
        return NewElfErrorf(ElfErrValueOutOfBounds,
            "value out of bounds: file too small for ELF header (need %d bytes, got %d)",
            ehLen, l.fileSize)
    }

    var hdrBuf [ehLen]byte
    if _, err := io.ReadFull(io.NewSectionReader(l.rd, 0, ehLen), hdrBuf[:]); err != nil {
        return NewElfErrorf(ElfErrFailedToParse, "failed to parse ELF header: %v", err)
    }
    // ... rest of function
}

func (l *Loader) validateElfHeader() error {
    eh := &l.eh
    ident := &eh.Ident

    if string(ident[:elf.EI_CLASS]) != elf.ELFMAG {
        return NewElfError(ElfErrFailedToParse, "not an ELF file: invalid magic number")
    }

    if eh.Type != elf.ET_DYN {
        return NewElfErrorf(ElfErrWrongType, "wrong type: expected ET_DYN, got %d", eh.Type)
    }

    if eh.Machine != elf.EM_BPF {
        return NewElfErrorf(ElfErrWrongMachine, "wrong machine: expected EM_BPF, got %d", eh.Machine)
    }
```

```
    // ... convert all other fmt.Errorf calls to typed errors
}
```

Step 3: Simplify Fuzzing Harness

Update `pkg/solfuzz/elf_loader.go`:

```
func elfErrToNum(err error) int32 {
    if err == nil {
        return 0
    }

    // Type-safe error code extraction
    if elfErr, ok := err.(*loader.ElfError); ok {
        return elfErr.ErrorCode()
    }

    // Fallback for non-ELF errors (shouldn't happen in normal operation)
    // This is a clear signal that an error isn't properly typed
    return -1
}
```

That's it! No more string matching, no more fragility.

Step 4: Add Unit Tests

Create `pkg/sbpf/loader/errors_test.go`:

```
func TestElfErrorCodes(t *testing.T) {
    tests := []struct {
        name string
        err  *ElfError
        want int32
    }{
        {"FailedToParse", NewElfError(ElfErrFailedToParse, "test"), 1},
        {"ValueOutOfBounds", NewElfError(ElfErrValueOutOfBounds, "test"), 21},
        // ... test all 23 error codes
    }

    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            if got := tt.err.ErrorCode(); got != tt.want {
                t.Errorf("ErrorCode() = %v, want %v", got, tt.want)
            }
        })
    }
}
```

Benefits

1. Type Safety

Compile-time guarantees about error handling:

```
var err error = loader.Load(...)
if elfErr, ok := err.(*loader.ElfError); ok {
    switch elfErr.Code {
```

```
    case loader.ElfErrFailedToParse:
        // Handle parsing errors
    case loader.ElfErrValueOutOfBounds:
        // Handle bounds errors
    // Compiler warns if cases are missing
    }
}
```

2. No String Matching
Error codes are explicit, not inferred from message content.

3. Clear Ownership
Client code owns error semantics, not fuzzing harness. The loader is responsible for consensus compatibility.

4. Maintainability
Adding new errors forces updates throughout codebase:

- Adding `ElfErrNewError` requires updating const declaration
- Compiler catches missing error handling
- Tests verify error codes

5. Consensus-Aware
Error codes are first-class citizens in the client code, making consensus implications explicit.

6. Better Testing
Can unit test that specific conditions return specific error codes without full fuzzing infrastructure.

7. Agave Compatibility
Direct 1:1 mapping to Agave's error enum makes differential fuzzing trivial.

Migration Strategy

Phase 1: Infrastructure (Low Risk)

1. Add `pkg/sbpf/loader/errors.go` with type definitions
2. Add tests for error code mapping
3. No changes to existing code yet

Phase 2: Incremental Adoption (Medium Risk)

1. Update one function at a time (e.g., `readHeader()`)
2. Test each function's error codes with unit tests
3. Update fuzzing harness as functions are converted
4. Old fmt.Errorf still works (returns -1 in fuzzing)

Phase 3: Complete Conversion (Medium Risk)

1. Convert all loader functions to use typed errors
2. Remove string matching from fuzzing harness
3. Add integration tests

Phase 4: Validation (Low Risk)

1. Run differential fuzzing with corpus
2. Verify all error codes match Agave
3. Update documentation

Recommendation

We recommend to refactor all logging commands to utilize the same set of logging and program exit functions. On top of that, implement typed errors with explicit error codes, similar to Agave's approach.

[B09] Implementation gaps indicated by `TODO: implement` comments

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

Description

There are multiple left-over `// TODO: implement` comments in the code base that indicate unimplemented, but important functionality, sometimes immediately causing a panic.

The following functions contain such as left-over `TODO` notice:

- `CalculateAndDeductTxFees` in `pkg/fees/fees.go`
- `RentCollectionPartitions` in `pkg/rent/partition.go`, which also panics when the unimplemented code path is run
- `partitionFromSlotIndices` in `pkg/rent/partition.go`
- `pubkeyRangeFromPartition` in `pkg/rent/partition.go`
- `calculateRentResult` in `pkg/rent/rent.go`
- `Update` in `pkg/sealevel/params.go` - implement the authorization checks

Recommendation

We recommend to fully implement missing functionality to establish feature and correctness parity with the reference implementation Agave.

Status

To be discussed with the client

[B10] Attribute `ProgramAccounts` of struct `InstructionCtx` is slice, but always contains exactly one entry

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

Description

The `InstructionCtx` struct contains the attribute `ProgramAccounts`, which is of type `[]uint64`. This attribute contains the indices to the program accounts in the respective transaction. Though this struct always contains only one entry.

Recommendation

Refactor the type of this attribute to be a single unsigned integer or pointer to an account. This will simplify the handling of the respective program account elsewhere in the code as well.

Status

To be discussed with the client

[B11] `BorrowedAccount` is dropped early in `SyscallInvokeSignedRustImpl`

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

Description

The function `SyscallInvokeSignedRustImpl` drops a `BorrowedAccount` early and noticeably twice, without the use of `defer` in contrast to the similar function `SyscallInvokeSignedCImpl`.

In particular, the following loop borrows an account, then immediately drops it:

```
for _, acct := range accounts {
    if acct.CallerAccount != nil {
        var calleeAcct *BorrowedAccount
        calleeAcct, err = instructionCtx.BorrowInstructionAccount(txCtx, acct.IndexOfAccount)
        if err != nil {
            //mlog.Log.Debugf("calling BorrowInstructionAccount for updateCallerAccount failed! %s", err)
            return syscallErr(err)
        }
        calleeAcct.Drop()
        err = updateCallerAccount(vm, acct.CallerAccount, calleeAcct)
        if err != nil {
            return syscallErr(err)
        }
        calleeAcct.Drop()
    }
}
```

Recommendation

Remove the early `calleeAcct.Drop()`.

Status

Fixed in commit 72592eb.

[B12] Snapshot downloader library might skip other snapshot compression-variants provided by an RPC node

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

Description

Mithril makes use of an external Go library `solana-snapshot-finder-go` for downloading Solana blockchain snapshots, which the Overclock team cloned into their own repository, see here.

In this library, the function `DownloadSnapshot` in `pkg/snapshot/download_utils.go` checks whether the given RPC server address provides a snapshot that is newer than a given slot number. The RPC server can be queried for a snapshot with `bz2` compression or `zst` compression. `DownloadSnapshot` will try both `bz2` and `zst`, in this order respectively. However, if the function successfully extracts the slot from a provided snapshot that does not fulfill the slot constraints, it will cancel immediately all tries for this RPC server. This means that when the `bz2` compressed snapshot does not fulfill the slot constraint, the `zst` compressed snapshot will be skipped entirely, even though the provided slot from the snapshot with `zst` compression might differ from the `bz2` compressed snapshot.

```
// Compare with existing slot and skip download if not newer
if existingSlot > 0 && remoteSlot <= existingSlot {
    /*log.Printf("Remote snapshot (slot %d) is not newer than existing snapshot (slot %d). Skipping download.",
    remoteSlot, existingSlot)*/
    return "", nil
}
```

Recommendation

We recommend to not `return` from the function in case the slot constraints are not fulfilled and instead `continue` the loop to check all other compressions.

Status

To be discussed with the client

[B13] RPC address hardcoded to a mainnet address for downloading snapshots

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

Description

The CLI that is specified in `cmd/mithril/node/node.go` allows for specifying the RPC server to be used for querying data from the blockchain such as the initial state by downloading snapshots, querying for other RPC servers, and for getting new shreds of blocks.

However mithril currently hardcodes the RPC server address for fetching snapshots to the address `https://api.mainnet-beta.solana.com`. This address corresponds to the Solana mainnet, while users of the tool might want to run the tool on other Solana chains, such as devnet or testnet, which are used for testing purposes. Also, the user might want to initially query a different RPC server such as their own.

Recommendation

Change the implementation to use the RPC address specified by the CLI arguments also for downloading the snapshot. In case no RPC address was specified by the user, a default value could be used instead.

Status

To be discussed with the client

[B14] Wrong error message

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

Description

The codebase contains wrong error messages, namely:

- `cmd/mithril/node/node.go:249`
  - `klog.Fatalf("must specify a path to download a snapshot to")`
  - This error message should instead inform the user that the CLI argument `--out <path>` is missing for specifying the output path of the built accountsdb.

Recommendation

We recommend to replace the wrong error messages with proper ones.

Status

To be discussed with the client

[B15] Inconsistent command-line interface argument names

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

Description

The command-line interface for `mithril verifier` accepts two parameters `--path <path>` and `--out <path>` that both take a path as an argument. The paths might point to a snapshot or an accountsdb. The exact semantics depends on additional supplied CLI flags such as `--snapshot`, `--accountsdb`, and `--download-snapshot <path>`.

The names `--path` and `--out` do not properly reflect the path arguments they take, especially since the passed paths change in semantics depending on other flags.

Scenario

When `--snapshot` is supplied to `mithril verifier`, `--path <path>` will point to a local copy of a snapshot and `--out <path>` will be the path where the accountsdb will be built.

When `--accountsdb` is supplied, `--path <path>` will point to a local copy of an accountsdb.

When `--download-snapshot <path>` is supplied, `--out <path>` will be a path where the accountsdb will be built.

Recommendation

Change the CLI argument names to names that better reflect their content and keep the semantics of the arguments consistent, no matter which other flags are passed. E.g., `--path <path>` and `--out <path>` could be renamed to `--snapshot-path <path>` and `--accountsdb-path <path>`.

Status

To be discussed with the client

[B16] Nil-pointer dereference on StatsD client initialization causing panic

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

Description

In `init()` (`pkg/statsd/statsd.go`), if `statsd.New` fails (e.g., StatsD server unavailable), `statsdClient` remains `nil` . Immediately after logging the error, the code unconditionally dereferences `statsdClient` to set Namespace, causing a nil-pointer dereference and application panic. This results in startup crash and Denial of Service if the StatsD server is unreachable.

Recommendation

The code should handle the failure case gracefully by returning early or checking for nil before dereferencing, allowing the application to continue running without StatsD metrics rather than crashing entirely.

```
func init() {
    var err error
    statsdClient, err = statsd.New("127.0.0.1:8125")
    if err != nil {
        mlog.Log.Errorf("couldn't start statsdClient: %v", err)
        return  // or just skip the next lines
    }
    statsdClient.Namespace = "mithril."
    periodicallySendRuntimeMetrics()
}
```

[B17] Panic on RPC failure in `GetLeaderForSlot`

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

Description

In the fallback path for missing block reward information, the code panics if `rpcc.GetLeaderForSlot` returns an error. Network issues, RPC provider failures, or malicious responses could trigger this panic, crashing the node.

Recommendation

Implement retry logic and `panic` only if **ALL** retries fail. This approach balances robustness against transient failures while maintaining the critical requirement that block reward information must be present for validation.

```
if rpcc != nil {
    var leaderForSlot solana.PublicKey
    var err error

    maxRetries := 10
    for attempt := 0; attempt < maxRetries; attempt++ {
        leaderForSlot, err = rpcc.GetLeaderForSlot(slot)
        if err == nil {
            break
        }
        if attempt < maxRetries-1 {
            time.Sleep(time.Duration(attempt+1) * 100 * time.Millisecond)
        }
    }

    if err != nil {
        panic(fmt.Sprintf("unable to get blockreward for slot %d after %d attempts: %v", slot, maxRetries, err))
    }

    block.BlockReward = &BlockRewardsInfo{Leader: leaderForSlot}
}
```

This approach:

- Tolerates transient failures (network hiccups, temporary RPC unavailability)
- Maintains validation integrity (panics if the information truly cannot be obtained)
- Provides clear error messages (includes attempt count and error details)
- Uses exponential backoff (increasing delays between retries)

[B18] Incorrect pointer returned in `BorrowExecutableAccount`

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

Description

The `BorrowExecutableAccount` method iterates over `txCtx.ExecutableAccounts` by value, and returns the address of the loop variable (`execAcct`) instead of the slice element. In Go, the loop iteration variable is reused for each iteration, so the returned pointer refers to a heap-escaped copy, not the actual element in the slice. This means modifications to the returned `BorrowedAccount` may not reflect in the transaction context, and could lead to inconsistent state or memory safety issues.

```
func (instrCtx *InstructionCtx) BorrowExecutableAccount(txCtx *TransactionCtx, pubkey solana.PublicKey) (*BorrowedAccount, error) {
    for _, execAcct := range txCtx.ExecutableAccounts {
        if execAcct.Key() == pubkey && execAcct.AccountExists() {
            return &execAcct, nil
        }
    }
    return nil, fmt.Errorf("unknown account")
}
```

Recommendation

Since the function isn't currently used, the loop variable bug doesn't pose an immediate risk. However, if someone were to start using this function in the future without noticing the bug, it could lead to subtle issues where modifications through the returned `BorrowedAccount` pointer don't affect the actual executable account in the transaction context.

```
func (instrCtx *InstructionCtx) BorrowExecutableAccount(txCtx *TransactionCtx, pubkey solana.PublicKey) (*BorrowedAccount, error) {
    for i := range txCtx.ExecutableAccounts {
        if txCtx.ExecutableAccounts[i].Key() == pubkey && txCtx.ExecutableAccounts[i].AccountExists() {
            return &txCtx.ExecutableAccounts[i], nil
        }
    }
    return nil, fmt.Errorf("unknown account")
}
```

[B19] ELF loader loads different number of bytes prior to validation compared to Agave

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

Description

Mithril's ELF parser performs the following steps in function `pkg/sbpf/loader/parse.go:parse`:

- `readHeader`
- `validateElfHeader`
- `loadProgramHeaderTable`
- `readSectionHeaderTable`
- `parseSections`
- `parseDynamic`
- `validate`

Here, the function `readHeader` reads the first 64 bytes of the header with `io.ReadFull` returning an error if it cannot read all 64 bytes, see:

```
if _, err := io.ReadFull(io.NewSectionReader(l.rd, 0, ehLen), hdrBuf[:]); err != nil {
    return err
}
```

Afterwards, the header is validated in function `validateElfHeader`.

This order of steps is different to Agave, with Agave first parsing up to the 52nd byte in `src/elf.rs:load` in crate `solana-sbpf`:

```
const E_FLAGS_OFFSET: usize = 48;
let e_flags = LittleEndian::read_u32(
    bytes
        .get(E_FLAGS_OFFSET..E_FLAGS_OFFSET.saturating_add(std::mem::size_of::<u32>()))
        .ok_or(ElfParserError::OutOfBounds)?,
);
let config = loader.get_config();
let sbpf_version = match e_flags {
    0 => SBPFVersion::V0,
    1 => SBPFVersion::V1,
    2 => SBPFVersion::V2,
    3 => SBPFVersion::V3,
    4 => SBPFVersion::V4,
    _ => SBPFVersion::Reserved,
};
if !config.enabled_sbpf_versions.contains(&sbpf_version) {
    return Err(ElfError::UnsupportedSBPFVersion);
}
```

This means that when passed an ELF file with at least 52 bytes and less than 64 bytes that at the same time has an invalid SBPF version field, Mithril would report an Error due to running out of memory while Agave would report an Error due to the SBPF version field being invalid.

Severity: Low

This is an error reporting order issue that only affects the specific error code returned for malformed inputs. It does not affect:

- Valid ELF file loading
- Program execution correctness
- Security of the loader

However, it causes differential fuzzing failures that can mask real bugs.

Recommendation
We recommend adjusting the parser to strictly follow Agave's order of parsing and validation to eliminate this difference, even when it only affects error reporting in logs.

Status
To be discussed with the client

[B20] Mithril adopts code design decisions from Agave

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

Description

Agave makes use of index variables of type unsigned integer that refer to a specific element in a slice. This pattern is used throughout the code base for referring to accounts.

Mithril adopts the same pattern, utilizing `uint64` typed variables to refer to elements in slices. This results in structs with `uint64` variables that only implicitly refer to the types they point to:

```
type InstructionAccount struct {
    IndexInTransaction uint64
    IndexInCaller      uint64
    IndexInCallee      uint64
    IsSigner           bool
    IsWritable         bool
}
```

```
type BorrowedAccount struct {
    TxCtx              *TransactionCtx
    InstrCtx           *InstructionCtx
    IndexInTransaction uint64
    IndexInInstruction uint64
    Account            *accounts.Account
}
```

The actual type of the element these indices point to can be inferred by helper functions that utilize these indices:

```
func (instrCtx *InstructionCtx) IndexOfInstructionAccountInTransaction(instrAcctIdx uint64) (uint64, error) {
    if len(instrCtx.InstructionAccounts) == 0 || instrAcctIdx > uint64(len(instrCtx.InstructionAccounts)-1) {
        return 0, InstrErrNotEnoughAccountKeys
    }
    return instrCtx.InstructionAccounts[instrAcctIdx].IndexInTransaction, nil
}
```

This pattern reduces readability of Mithril's code.

However, since Go has native garbage collection support, Mithril could instead employ a different pattern with pointer references to accounts instead of the utilization of index variables and helper functions. This would improve readability.

Recommendation

We recommend to refactor the codebase and replace index variables with Go pointers to the underlying datatype. E.g., the struct `BorrowedAccount` implementation could be changed to:

```
type BorrowedAccount struct {
    TxCtx            *TransactionCtx
    InstrCtx         *InstructionCtx
    Key                *solana.PublicKey
    InstAccount *InstructionAccount
    Account         *accounts.Account
```

```
        // [...] all other references that were inferred using the indices
}
```

Status

To be discussed with the client

[B21] Use-After-Drop in `ExecutionCtx::ExecuteInstruction`

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

Summary

`ExecutionCtx::ExecuteInstruction` in `pkg/sealevel/execution_ctx.go` accesses a borrowed account's key after calling `Drop()` on the account reference, leading to use-after-free behavior. This violates the borrow-checker semantics that Mithril's account borrowing system attempts to emulate.

Discovery Method

Discovered during code review while investigating the native program override bug (F-D05). The issue was identified in the program-resolution logic of `ExecuteInstruction`.

Root Cause Analysis

Borrow Checker Semantics

Mithril implements a Rust-style borrow checker pattern for account access:

- `BorrowProgramAccount` / `BorrowInstructionAccount` acquire a reference to an account
- `Drop()` releases the reference and invalidates it
- Accessing the account after `Drop()` violates borrow semantics and can lead to undefined behavior

The Bug

In `pkg/sealevel/execution_ctx.go`, the `ExecuteInstruction` function contains this code:

```
func (execCtx *ExecutionCtx) ExecuteInstruction() error {
    start := time.Now()
    txCtx := execCtx.TransactionContext
    instrCtx, err := txCtx.CurrentInstructionCtx()
    if err != nil {
        return err
    }

    borrowedRootAccount, err := instrCtx.BorrowProgramAccount(txCtx, 0)
    if err != nil {
        return InstrErrUnsupportedProgramId
    }

    ownerId := borrowedRootAccount.Owner()
    borrowedRootAccount.Drop()  // Account reference is dropped here

    var builtinId solana.PublicKey
    if ownerId == a.NativeLoaderAddr {
        builtinId = borrowedRootAccount.Key()  //  BUG: Accessing dropped account
    } else {
        builtinId = ownerId
    }

    // ... rest of function
}
```

Sequence:

1. Borrow the program account (`borrowedRootAccount`)
2. Read the owner (`ownerId := borrowedRootAccount.Owner()`)
3. Drop the account reference (`borrowedRootAccount.Drop()`)
4. Attempt to access the dropped reference (`borrowedRootAccount.Key()`) ← **BUG**

Impact

- **Severity**: Medium
- **Memory Safety**: Violates borrow-checker semantics, potentially leading to use-after-free
- **Undefined Behavior**: Accessing dropped references has undefined behavior depending on Go's memory management
- **Maintainability**: Makes code harder to reason about and maintain
- **Crashes**: Could lead to panics or memory corruption in certain scenarios
- **Design Integrity**: Undermines the purpose of the borrow checker pattern

---

Technical Details

Why This Is Dangerous

The `Drop()` method is designed to release the account lock and mark the reference as invalid:

```
func (ba *BorrowedAccount) Drop() {
    ba.accounts.Unlock(ba.index)
    // After this point, the BorrowedAccount should not be used
}
```

After `Drop()` :

- The account may be modified by other code
- The underlying memory may be reallocated
- The reference is semantically invalid

Accessing `borrowedRootAccount.Key()` after `Drop()` reads from a reference that should no longer be valid.

Current Behavior

In practice, this bug may not cause immediate crashes because:

- Go's garbage collector may not have reclaimed the memory yet
- The account key is a value type that was already copied
- No other code modifies the account between `Drop()` and the access

However, relying on this is incorrect and fragile.

Recommendation

Save the account key before dropping the account reference. This follows proper borrow-checker semantics where all data needed from a borrowed reference is extracted before dropping it.

Fix Implementation

Modified `pkg/sealevel/execution_ctx.go` to save the key before dropping:

```go
func (execCtx *ExecutionCtx) ExecuteInstruction() error {
    start := time.Now()
    txCtx := execCtx.TransactionContext
    instrCtx, err := txCtx.CurrentInstructionCtx()
    if err != nil {
        return err
    }

    borrowedRootAccount, err := instrCtx.BorrowProgramAccount(txCtx, 0)
    if err != nil {
        return InstrErrUnsupportedProgramId
    }

    // === START RV FIX [F-D06] ===
    // Save account key BEFORE dropping the reference
    // Accessing the account after Drop() violates borrow-checker semantics
    rootAccountKey := borrowedRootAccount.Key()
    // === END RV FIX [F-D06] ===

    ownerId := borrowedRootAccount.Owner()
    borrowedRootAccount.Drop()

    var builtinId solana.PublicKey
    if ownerId == a.NativeLoaderAddr {
        builtinId = rootAccountKey  //  Use saved key
    } else {
        builtinId = ownerId
    }

    // ... rest of function
}
```

Key changes:

- Extract `rootAccountKey` before calling `Drop()`
- Use the saved key instead of accessing the dropped reference
- Maintains correct borrow-checker semantics

Files Modified

- `pkg/sealevel/execution_ctx.go` (modified) – Fixed use-after-drop in `ExecuteInstruction`

References

- Rust's borrow-checker documentation
- Mithril's account borrowing implementation in `pkg/sealevel/transaction_context.go`
- Related borrow/drop patterns throughout the codebase

Status

Fixed in commit f10883c.

[B22] Lack of proper pull request and CI workflows

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

Description

Development on Mithril happens for the most part directly on the Mithril default branch, namely the `dev` branch. In addition, Mithril's CI consists solely of building the Mithril Go project. Considering the complexity of the Mithril project and that Mithril is developed by a team in an open-source repository, this increases the risk of introducing bugs in the project. In addition, the lack of pull requests that group related commits together as part of a collective pull request makes tracing changes and their reasoning harder. Finally, code review of commits is not possible if the commits are pushed directly to the default branch instead of being introduced as a pull request.

Recommendation

To increase code quality, reduce the risk of introducing bugs, and introduce more documentation about codebase changes, we recommend to disable the ability to push changes directly to the default branch. Instead, changes should only be possible by opening a pull request containing the requested changes and merging that pull request into the default branch. In addition, we advise to use the code review functionality of pull requests as well as making the code review process mandatory by requiring at least one approval from another maintainer on a pull request before merging. Further, pull requests should contain details about the changes and their motivation, possibly referencing other materials or issues.

We also recommend to introduce more CI workflows that run all available unit and integration tests of the code base. A pull request should only be allowed to be merged once all workflows successfully pass. In the ideal case, bugs and new features introduce respective regression tests to make sure that intended functionality or fixes do not regress due to other later codebase changes.

Status

To be discussed with the client