Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Fluent STF

Veridise Inc.

July 3, 2025

► Prepared For:

Fluent
https://www.fluent.xyz/

► Prepared By:

Petr Susil
Kostas Ferles

► Contact Us:

contact@veridise.com

► Version History:

July 02, 2025 V2
May 28, 2025 V1

Contents

1 Executive Summary

From May 12, 2025 to May 25, 2025, Fluent engaged Veridise to conduct a security assessment of their Fluent STF. The security assessment covered the fluent-stf-sp1 repository that implements a state transition function for Ethereum-compatible blockchains, enabling block execution and verification within a zero-knowledge virtual machine (zkVMs) environment. Veridise conducted the assessment over 4 person-weeks, with 2 security analysts reviewing the project over 2 weeks on commit `0d04ea95`. The review strategy involved a tool-assisted analysis of the program source code performed by Veridise security analysts as well as thorough code review.

Project Summary. The Fluent STF project implements a state transition function (STF) for Ethereum-compatible blockchains, designed for integration with zero-knowledge virtual machines (zkVMs). It facilitates the execution and verification of Ethereum blocks within a zkVM environment, enabling the generation of zero-knowledge proofs for block execution.

The STF processes Ethereum blocks by fetching necessary data from a JSON-RPC node, executing the block's transactions, and verifying the resulting state transitions. It supports multiple EVM-compatible chains, including Ethereum Mainnet and OP Stack Mainnet. Its design aligns with Ethereum's execution specifications, ensuring that block headers and bodies are validated according to consensus rules. It leverages existing libraries, such as REVM, for EVM execution and validation tasks.

Code Assessment. The Fluent STF developers provided the source code of the Fluent STF contracts for the code review. The source code is based on `https://github.com/succinctlabs/rsp`, developed by succinct labs, with modifications by the Fluent STF developers. It contains some documentation in the form of READMEs and documentation comments on functions. The Veridise security analysts had access to some limited documentation by succinct labs about the core ZK application. Also, the Veridise team met with the Fluent STF developers to ask questions related to their additions to the code base. These additions were mainly undocumented, however, they were relatively small additions.

The source code contained a test suite, which the Veridise security analysts noted covered most critical aspects of the code base. It also contained integration tests that tested the composition of all components of the transition function.

Summary of Issues Detected. The security assessment uncovered 7 issues, 4 of which are assessed to be of high or critical severity by the Veridise analysts. Specifically, auditors discovered that default placeholder event topics in `BRIDGE_INFO` prevent accurate tracking of bridge events, which renders protocol-level message processing inoperable (V-FLUENT-VUL-001); the `ClientExecutor::execute` function fails to validate block headers before execution, violating consensus rules and risking acceptance of invalid blocks (V-FLUENT-VUL-002); an incorrect offset is used to extract the `messageHash` from the `SentMessage` event log, breaking log decoding logic and downstream processing (V-FLUENT-VUL-003); and several mandatory block

validations required by the Ethereum execution specification are missing from `ClientExecutor::execute` (V-FLUENT-VUL-004). The Veridise analysts also identified 3 informational findings.

The Fluent STF developers acknowledged one informational issue and implemented fixes for all other identified issues. These fixes have been reviewed and verified by Veridise analysts.

Recommendations. After conducting the assessment of the protocol, the security analysts had a few suggestions to improve the Fluent STF.

Leverage Type System to Enforce Invariants: The codebase would benefit from introducing stronger type-level abstractions to enforce key invariants at compile time. For example, separating validated and non-validated data structures using distinct wrapper types can prevent misuse and reduce reliance on runtime assertions (e.g., as described in issue V-FLUENT-VUL-007).

Mark Guest-Accessible Interfaces Explicitly: The current directory structure and documentation do not clearly delineate which functions and modules are safe for invocation from a guest program. To mitigate the risk of future security-critical bugs, it is recommended to improve documentation and the directory structure of the project to explicitly label guest-facing APIs. This may involve re-organizing certain modules or conventions that clearly signal safe vs. unsafe interfaces.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Fluent STF	`0d04ea95`	Rust	Ethereum, Optimism

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
May 12–May 25, 2025	Manual & Tools	2	4 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	1	1	1
High-Severity Issues	3	3	3
Medium-Severity Issues	0	0	0
Low-Severity Issues	0	0	0
Warning-Severity Issues	0	0	0
Informational-Severity Issues	3	3	2
TOTAL	7	7	6

Table 2.4: Category Breakdown.

Name	Number
Logic Error	2
Data Validation	2
Maintainability	2
Proof Optimization	1

3 Security Assessment Goals and Scope

3.1 Security Assessment Goals

The engagement was scoped to provide a security assessment of Fluent STF's zkVM application codes. During the assessment, the security analysts aimed to answer questions such as:

- Does the implementation of the state transition function (`ClientExecutor`) conform to the EVM execution specification?
- Does the guest program of the zkVM application validate all of its inputs?
- Is the Merkle Patricia Trie (MPT) data structure implemented correctly?
- Are the zkVM and on-chain smart contracts in sync with one another?

3.2 Security Assessment Methodology & Scope

Security Assessment Methodology. To address the questions above, human experts thoroughly reviewed the relevant code and documentation.

Scope. The scope of this security assessment is limited to the `crates` folder, excluding the `crates/storage` and `crates/executor/host` sub-folders, of the source code provided by the Fluent STF developers, which contains the smart contract implementation of the Fluent STF.

Methodology. Veridise security analysts inspected the provided tests and read the Fluent STF documentation. They also regularly consulted with Fluent STF developers to ask questions about the code.

3.3 Classification of Vulnerabilities

When Veridise security analysts discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise.

The severity of a vulnerability is evaluated according to the Table 3.1.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

The likelihood of a vulnerability is evaluated according to the Table 3.2.

The impact of a vulnerability is evaluated according to the Table 3.3:

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Vulnerability Report

This section presents the vulnerabilities found during the security assessment. For each issue found, the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.) is specified. Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-FLUENT-VUL-001	Default Event Topics in BRIDGE_INFO . . .	Critical	Fixed
V-FLUENT-VUL-002	Missing Validation of Block Header in . . .	High	Fixed
V-FLUENT-VUL-003	Incorrect Log Topic Offset for . . .	High	Fixed
V-FLUENT-VUL-004	Missing Block Validations in . . .	High	Fixed
V-FLUENT-VUL-005	Unnecessary Bloom Filter Recalculation in . . .	Info	Fixed
V-FLUENT-VUL-006	Redundant Caching Logic	Info	Fixed
V-FLUENT-VUL-007	Prefix Byte in Decoded MPT Node Paths . . .	Info	Acknowledged

4.1 Detailed Description of Issues

4.1.1 V-FLUENT-VUL-001: Default Event Topics in BRIDGE_INFO Prevent Accurate Log Tracking

Severity	Critical	**Commit**	0d04ea9
Type	Logic Error	**Status**	Fixed
File(s)	`crates/executor/client/src/executor.rs`		
Location(s)	static BRIDGE_INFO		
Confirmed Fix At	c417312		

The global `BRIDGE_INFO` in `executor.rs` sets default values for the `withdrawal_topic` and `deposit_topic` fields as zero hashes:

```
1 static BRIDGE_INFO: BridgeInfo = BridgeInfo {
2     bridge_address: address!("0x00961Ef480Eb55e80D19ad83579A64c007002123"),
3     withdrawal_topic: b256!("0
      x0000000000000000000000000000000000000000000000000000000000000000"),
4     deposit_topic: b256!("0
      x0000000000000000000000000000000000000000000000000000000000000000"),
5 };
```

Snippet 4.1: Snippet from `executor.rs`

These placeholders do not match the actual event topics emitted by the bridge contract. More critically, the application's on-chain contracts track *two distinct topics, namely `ReceivedMessage` and `RollbackMessage`,* for withdrawal events, indicating that a single `withdrawal_topic` field is insufficient for accurate log parsing.

Impact The `ClientExectutor` will not be able to properly track withdrawals and deposits which will render the entire protocol unusable since the on-chain will not be able to properly process blocks.

Recommendation Replace the hardcoded zeroed hashes with the actual `keccak256` hashes of the relevant event signatures. For withdrawals, extend the implementation to support tracking multiple topics, either by changing the type of `withdrawal_topic` to a list or introducing an auxiliary structure to represent multiple monitored events. Ensure that event parsing logic reflects this multiplicity and aligns with the the on-chain logic of the protocol.

Developer Response The developers have fixed this issue in commit c417312.

4.1.2 V-FLUENT-VUL-002: Missing Validation of Block Header in ClientExecutor::execute

Severity	High	**Commit**	0d04ea9
Type	Data Validation	**Status**	Fixed
File(s)	`crates/executor/client/src/executor.rs`		
Location(s)	function ClientExecutor::execute		
Confirmed Fix At	f82ce57		

In the `ClientExecutor::execute` function, the block header from `input.current_block` is used to initialize the block execution strategy and perform sender recovery without validating the block header. Specifically, the code does not invoke the `reth_ethereum_consensus::EthBeaconConsensus::validate_header` function to ensure the header adheres to Ethereum consensus rules (see snippet below).

```
let block = profile!("recover senders", {
    F::Primitives::from_input_block(input.current_block.clone())
        .try_into_recovered()
        .map_err(|_| ClientError::SignatureRecoveryFailed)
})?;
```

Snippet 4.2: Snippet from `ClientExecutor::execute()`

This violates the Ethereum execution specification, which requires clients to validate the block header during a state transition (see here).

Impact Without validating the block header, the executor may process blocks with invalid or adversarial headers. Such behavior breaks compliance with Ethereum's consensus protocol and could destabilize applications or nodes relying on this executor.

Recommendation Call functions `EthBeaconConsensus::validate_header`, `EthBeaconConsensus::validate_header_with_total_difficulty`, and `EthBeaconConsensus::validate_header_against_parent` on `input.current_block.header` before it is used in any further processing. This ensures alignment with the execution specification and protects against processing invalid headers.

Developer Response The developers have fixed this issue in commit f82ce57.

4.1.3 V-FLUENT-VUL-003: Incorrect Log Topic Offset for messageHash in SentMessage Event

Severity	High	**Commit**	0d04ea9
Type	Logic Error	**Status**	Fixed
File(s)	`crates/executor/client/src/events_hash.rs`		
Location(s)	const SEND_EVENT_MESSAGE_HASH_OFFSET		
Confirmed Fix At	586c136		

The developers intend to track the `messageHash` field in the `SentMessage` Solidity event:

```
event SentMessage(
    address indexed sender,
    address indexed to,
    uint256 value,
    uint256 chainId,
    uint256 blockNumber,
    uint256 nonce,
    bytes32 messageHash,
    bytes data
);
```

Snippet 4.3: Snippet from the smart contracts

This event includes two indexed parameters (`sender` and `to`), making the remaining parameters - including `messageHash` - part of the unindexed data section of the log (`data` field in the receipt). However, the code currently defines the offset as:

```
const SEND_EVENT_MESSAGE_HASH_OFFSET: usize = 64;
```

Snippet 4.4: Snippet from `events_hash.rs`

This value erroneously assumes that `messageHash` starts at byte offset 64 in the unindexed data payload.

Impact Incorrectly calculating the offset causes the client to extract the wrong data from logs, leading to failures in message verification, proof generation, or downstream event processing that depends on the `messageHash`. This may cause message mismatches, bridging failures, or incorrect state updates.

Recommendation Recalculate the correct offset of `messageHash` within the event's unindexed data section. Each unindexed field occupies 32 bytes. Given that `messageHash` follows four 256-bit values (`value`, `chainId`, `blockNumber`, `nonce`), its correct offset is:
`4 fields \times 32 bytes = 128 bytes.`

Update the constant to:

```
const SEND_EVENT_MESSAGE_HASH_OFFSET: usize = 128;
```

Developer Response The developers have fixed this issue in commit 586c136.

4.1.4 V-FLUENT-VUL-004: Missing Block Validations in ClientExecutor::execute

Severity	High	**Commit**	0d04ea9
Type	Data Validation	**Status**	Fixed
File(s)	`crates/executor/client/src/executor.rs`		
Location(s)	function execute		
Confirmed Fix At	51a3ddf		

The `ClientExecutor::execute` function does not currently perform several validations mandated by the Ethereum execution specification. Specifically, it omits the following checks:

1. `block.ommers` must be empty; otherwise, the block is invalid:

```
if block.ommers != ():
    raise InvalidBlock
```

2. The `transactions_root` derived from the block body must match the value in the block header:

```
if transactions_root != block.header.transactions_root:
    raise InvalidBloc
```

3. The `withdrawals_root` must match the one in the header:

```
if withdrawals_root != block.header.withdrawals_root:
    raise InvalidBlock
```

4. The `blob_gas_used` reported by the EVM must equal the value in the block header:

```
if block_output.blob_gas_used != block.header.blob_gas_used:
    raise InvalidBlock
```

Checks 1-3 are already encapsulated by the REVM function `validate_body_against_header`.

However, `ClientExecutor::execute` does not currently invoke this function or replicate its logic, nor does it independently enforce the `blob_gas_used` check.

Impact:

Without these checks, the executor may accept blocks that:

- Include deprecated fields (e.g., ommers) not allowed in recent forks,
- Mismatch declared and actual root hashes (transactions, withdrawals),
- Misreport blob gas usage, potentially undermining data availability assumptions.

These discrepancies violate the consensus specification and may result in accepting invalid blocks, risking consensus divergence, state corruption, or denial-of-service vectors.

Recommendation:

Integrate these validations into the `ClientExecutor::execute` function:

- Call `validate_body_against_header` to enforce checks 1–3.
- Add a separate check comparing `block_output.blob_gas_used` with `block.header.blob_gas_used`.

Developer Response The developers have fixed this issue in commit 51a3ddf.

4.1.5 V-FLUENT-VUL-005: Unnecessary Bloom Filter Recalculation in ClientExecutor::execute

Severity	Info	**Commit**	0d04ea9
Type	Proof Optimization	**Status**	Fixed
File(s)	`crates/executor/client/src/executor.rs`		
Location(s)	function ClientExecutor::execute		
Confirmed Fix At	4ea59ba		

The function `ClientExecutor::execute` redundantly recomputes the bloom filter from the emitted logs during execution. However, bloom filter validation is already handled in `validate_block_post_execution` (see fluent/crates/ethereum/consensus/src/validation.rs) where the bloom value from input is verified against logs.

```
**pub fn execute(
        &self,
        mut input: ClientExecutorInput<F::Primitives>,
    ) -> Result<(Header, BridgeHashes), ClientError> {
            ...**
        // Accumulate the logs bloom.
        let mut logs_bloom = Bloom::default();
        profile!("accrue logs bloom", {
            executor_output.receipts.iter().for_each(|r| {
                logs_bloom.accrue_bloom(&r.bloom());
            })
        });
                ...
                let header = Header {
                        ...,
            logs_bloom,
                        ...,
                };
                ...
}
```

Snippet 4.5: Snippet from `execute()`

Impact This redundancy leads to:

- **Unnecessary computation**, potentially degrading performance.
- **Code duplication**, which increases maintenance burden.

Recommendation Remove the redundant bloom filter computation from `ClientExecutor::execute`. Rely on the centralized and dedicated validation in `validate_block_post_execution` to ensure correctness. This reduces overhead and maintains a clear separation of concerns.

Developer Response The developers have fixed this issue in commit **4ea59ba**.

4.1.6 V-FLUENT-VUL-006: Redundant Caching Logic

Severity	Info	**Commit**	0d04ea9
Type	Maintainability	**Status**	Fixed
File(s)	`crates/mpt/src/mpt.rs`		
Location(s)	hash, reference_encode, reference_length		
Confirmed Fix At	39b95a8		

The methods `hash`, `reference_encode`, and `reference_length` each directly invoke the expression:

```
self.cached_reference
    .borrow_mut()
    .get_or_insert_with(|| self.calc_reference())
    .clone()
```

This logic duplicates functionality already provided by the `reference()` method, which serves to lazily compute and cache the reference value. By bypassing the `reference()` method and repeating this logic inline, these functions risk diverging behavior if the caching mechanism is later modified or extended.

```
#[inline]
pub fn reference(&self) -> MptNodeReference {
   self.cached_reference.borrow_mut().get_or_insert_with(|| self.calc_reference()).
    clone()
}
```

Impact This duplication introduces **redundancy and maintainability risks**:

- **Inconsistent behavior** may arise if future updates change how `reference()` computes or caches data.
- **Code duplication** increases the chance of bugs and hampers readability.
- **Harder to refactor**, since caching logic is not centralized.

Recommendation Refactor `hash`, `reference_encode`, and `reference_length` to call the `reference()` method instead of duplicating its logic. This ensures consistent behavior, simplifies maintenance, and localizes caching logic to a single method.

Developer Response The developers have fixed this issue in commit 39b95a8.

4.1.7 V-FLUENT-VUL-007: Prefix Byte in Decoded MPT Node Paths May Lead to Ambiguity

Severity	Info	**Commit**	0d04ea9
Type	Maintainability	**Status**	Acknowledged
File(s)	`crates/mpt/src/mpt.rs`		
Location(s)	Functions encode and decode		
Confirmed Fix At	N/A		

The procedure for decoding MPT paths distinguishes between leaf and extension nodes using a prefix byte, added via the `to_encoded_path` function:

```
pub fn to_encoded_path(mut nibs: &[u8], is_leaf: bool) -> Vec<u8> {
    let mut prefix = (is_leaf as u8) * 0x20;
    if nibs.len() % 2 != 0 {
        prefix += 0x10 + nibs[0];
        nibs = &nibs[1..];
    }
    iter::once(prefix).chain(nibs.chunks_exact(2).map(|byte| (byte[0] << 4) + byte
    [1])).collect()
}
```

Snippet 4.6: Snippet from `to_encoded_path`

During decoding, however, the prefix byte is retained in the path used to construct the MPT node:

```
let path: Vec<u8> = rlp.val_at(0)?;
let prefix = path[0];
if (prefix & (2 << 4)) == 0 {
    let node: MptNode = Decodable::decode(&rlp.at(1)?)?;
    Ok(MptNodeData::Extension(path, Box::new(node)).into())
} else {
    Ok(MptNodeData::Leaf(path, rlp.val_at(1)?).into())
}
```

This design requires all logic that constructs new MPT nodes to always use `to_encoded_path`, or risk creating nodes with incorrect or ambiguous encodings.

Impact Retaining the prefix in the internal representation:

- Introduces fragility, as developers must remember to pre-process every node path using `to_encoded_path`.
- Increases risk of malformed MPT nodes if this step is missed or applied inconsistently.

Recommendation Normalize MPT node representations by stripping the prefix byte during decoding. Store paths internally in a clean, non-prefixed format. Apply `to_encoded_path` only during RLP encoding when serializing nodes for storage or transmission. This simplifies logic, avoids duplication of metadata in runtime structures, and ensures consistent and unambiguous node construction.

Developer Response The developers have acknowledged the issue.

Glossary

zero-knowledge circuit A cryptographic construct that allows a prover to demonstrate to a verifier that a certain statement is true, without revealing any specific information about the statement itself. See `https://en.wikipedia.org/wiki/Zero-knowledge_proof` for more. 16

zkVM A general-purpose zero-knowledge circuit that implements proving the execution of a virtual machine. This enables general purpose programs to prove their execution to outside observers, without the manual constraint writing usually associated with zero-knowledge circuit development . 1