Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Succinct

SP1 Call Contract

Veridise Inc.

June 30, 2025

► **Prepared For:**

Succinct
https://www.succinct.xyz/

► **Prepared By:**

Jon Stephens
Petr Susil

► **Contact Us:**

contact@veridise.com

► **Version History:**

June 30, 2025 Initial Draft

Contents

1 Executive Summary

From June 23, 2025 to June 26, 2025, Succinct engaged Veridise to conduct a security assessment of their SP1 Call Contract. The security assessment covered the SP1 Call Contract library which allows developers to create proofs about block information and calls performed off-chain over on-chain state with the SP1 zkVM. Additionally, the security assessment covered a solidity library to validate the public information made available in a proof and an example zkVM application that queries Uniswap. Veridise conducted the assessment over 8 person-days, with 2 security analysts reviewing the project over 4 days on commit `6ccad76`. 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. SP1 Contract-Call enables the generation of zero-knowledge proofs (ZKPs) for off-chain contract execution. It allows complex, computationally heavy contract calls to be run off-chain, producing a cryptographic proof that verifies the correct result. This proof can then be verified on-chain at minimal cost, reducing gas usage while preserving trust in the outcome and supporting more efficient decentralized applications.

The system consists of an off-chain client executor that gathers blockchain state at a specific block and re-executes the target contract logic inside a zero-knowledge virtual machine, and generates a proof. Utilities are also provided to query the execution block header, such as querying the logs of a given block. On-chain, a Solidity contract library (such as ContractCall.sol and example implementations like UniswapCall.sol) verifies the proof's public outputs, allowing applications to incorporate the results securely without repeating expensive computations.

Code Assessment. The SP1 Call Contract developers provided the source code of the library for the code review. The source code appears to be mostly original code written by the SP1 Call Contract developers. It contains some documentation in the form of READMEs and documentation comments on functions and important structs. To facilitate the Veridise security analysts' understanding of the code, the developers also provided the developer documentation site for the project as well as documentation for the RSP library, which contains the bulk of the contract execution logic. The source code contained a set of example applications to test the library and demonstrate the functionality. These examples exercised most of the main features provided by the library but a few were incomplete.

Summary of Issues Detected. The security assessment uncovered 14 issues, 2 of which are assessed to be of high severity by the Veridise analysts. Specifically, auditors discovered that users can incorrectly prove the absence of a log, which could lead to false claims about contract event emissions and compromise downstream processing (V-SP1CC-VUL-001). Additionally, critical anchor consistency checks are only performed during contract calls, leaving proof generation susceptible to inconsistencies that could undermine trust in the verified result (V-SP1CC-VUL-002).

The assessment additionally identified 3 medium-severity issues as well as 4 low-severity issues and 5 warnings. Auditors found that public values may omit execution environment metadata, potentially limiting the ability of verifiers to fully assess the context of a proof (V-SP1CC-VUL-003). The length of a branch is not enforced, which could allow malformed proofs that bypass certain structural checks (V-SP1CC-VUL-004). Additionally, an unused 'state_requests' field can be manipulated without affecting proof validity, introducing unnecessary surface area for misuse (V-SP1CC-VUL-005).

The SP1 Call Contract developers have provided fixes for all of the identified issues.

Recommendations. After conducting the assessment of the protocol, the security analysts had a few suggestions to improve the SP1 Call Contract.

Documentation. The security analysts recommend extending the project's documentation with a security section that explains the security implications of the validation performed in the guest and the smart contract code. This is to ensure that developers are aware of the validation required to securely interact with the library's components in case they decide to add any custom logic. Additionally, we recommend including a best practices page to ensure developers know at a minimum to:

1. Always construct the client executor rather than using the guest input as the client executor performs vital validation.
2. On-chain they must always validate the authenticity of the block (unless in some rare cases they want to allow proofs over artificial states to be posted).
3. It should be encouraged to check the chain configuration hash in addition to the block to ensure users configured the environment correctly.

Examples. The library already contains several examples that demonstrate how to use the library in practice. The Veridise security analysts noticed that one of these examples was incomplete (`example-deploy`) and one skipped important validation (`events`). We recommend ensuring that all provided examples are complete and checked to ensure that the library is used safely as external users may use these examples as they develop similar applications. Additionally, we recommend that the developers ensure that these examples cover major use-cases of the library (the only example to demonstrate deploying a contract is incomplete).

Disclaimer. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
SP1 Call Contract	`6ccad76`	Rust, Solidity	SP1, Ethereum

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
June 23–June 26, 2025	Manual & Tools	2	8 person-days

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	0	0	0
High-Severity Issues	2	2	2
Medium-Severity Issues	3	3	3
Low-Severity Issues	4	4	4
Warning-Severity Issues	5	5	5
Informational-Severity Issues	0	0	0
TOTAL	14	14	14

Table 2.4: Category Breakdown.

Name	Number
Data Validation	7
Usability Issue	4
Logic Error	2
Dead Code	1

3 Security Assessment Goals and Scope

3.1 Security Assessment Goals

The engagement was scoped to provide a security assessment of SP1 Call Contract's smart contracts. During the assessment, the security analysts aimed to answer questions such as:

- Does the guest validate the accuracy of the information provided by the host?
- Does the commitment contain all the information necessary to validate the EVM environment in the guest?
- Can a malicious user prove inaccurate information assuming the zkVM application is honest and implemented correctly?
- Does the smart contract library provide all necessary utilities to validate a commitment?
- Can a malicious user bypass the smart contract library's validation assuming that the smart contract is honest and implemented correctly?
- Does the uniswap example correctly demonstrate how to build a zkVM application with the SP1 Call Contract library?
- Are anchor values and block metadata properly enforced to prevent state inconsistencies?
- Is the interface between the guest and the smart contract verifier free from ambiguity or missing data that could be exploited?
- Are there clear guidelines or safeguards in place to prevent common integration mistakes when using the SP1 Call Contract library?

3.2 Security Assessment Methodology & Scope

Security Assessment Methodology. To address the questions above, the audit involved a thorough manual review of the protocol by human experts.

Scope. The scope of this security assessment is limited to the `crates/client-executor/src/`, `contracts/src/` and `examples/uniswap/` directories of the source code provided by the SP1 Call Contract developers. These directories contain the SP1 library implementation, smart contract validation library and uniswap example respectively. From these directories, the following files were in-scope:

- `crates/client-executor/src/anchor.rs`
- `crates/client-executor/src/errors.rs`
- `crates/client-executor/src/io.rs`
- `crates/client-executor/src/lib.rs`
- `contracts/src/ContractCall.sol`
- `examples/uniswap/client/src/main.rs`
- `examples/uniswap/contracts/src/UniswapCall.sol`

Methodology. Veridise security analysts reviewed the reports of previous audits for SP1 Call Contract, inspected the provided tests, and read the SP1 Call Contract documentation. They then began a manual review of 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.

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

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

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 Trust Model

4.1 Operational Assumptions.

In addition to assuming that any out-of-scope components behave correctly, Veridise analysts assumed the following properties held when modeling security for SP1 Call Contract:

- That users of the SP1 Call Contract zkVM library invoke the library utilities correctly. More specifically, it was assumed that users would commit any produced public information, invoke the necessary functions to validate the guest input data.
- That users of the SP1 Call Contract smart contract library properly use the provided library utilities. More specifically, it was assumed that users would invoke the SP1 verifier to verify the proof and would validate the accuracy of public information using either the provided library functionality or would perform custom validation if necessary.
- Users have taken any necessary precautions to protect private inputs provided to the zkVM application.

5 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 5.1 summarizes the issues discovered:

Table 5.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-SP1CC-VUL-001	Users can Incorrectly Prove Log Absence	High	Fixed
V-SP1CC-VUL-002	Critical Anchor Consistency Checks Only . . .	High	Fixed
V-SP1CC-VUL-003	Length of branch not enforced	Medium	Fixed
V-SP1CC-VUL-004	Unused ‘state_requests‘ Field can be . . .	Medium	Fixed
V-SP1CC-VUL-005	Public Values May Omit Execution . . .	Medium	Fixed
V-SP1CC-VUL-006	Insufficient Block Number Validation in . . .	Low	Fixed
V-SP1CC-VUL-007	The execute Function May Encourage . . .	Low	Fixed
V-SP1CC-VUL-008	Loss of Variant Type When Converting . . .	Low	Fixed
V-SP1CC-VUL-009	Missing Validation of genesisHash in . . .	Low	Fixed
V-SP1CC-VUL-010	Misleading Error Message on Transaction . . .	Warning	Fixed
V-SP1CC-VUL-011	Unused BeaconBlockField Enum	Warning	Fixed
V-SP1CC-VUL-012	get_beacon_root_from_state Deviates . . .	Warning	Fixed
V-SP1CC-VUL-013	Inconsistent AnchorType in Guest and . . .	Warning	Fixed
V-SP1CC-VUL-014	No Genesis Hash Validation Limits . . .	Warning	Fixed

5.1 Detailed Description of Issues

5.1.1 V-SP1CC-VUL-001: Users can Incorrectly Prove Log Absence

Severity	High	**Commit**	6ccad76
Type	Logic Error	**Status**	Fixed
File(s)	`./crates/client-executor/src/lib.rs`		
Location(s)	./crates/client-executor/src/lib.rs:249-255		
Confirmed Fix At	0984bb9		

Description The `ClientExecutor` initializes its `logs` field by iterating over the transaction receipts provided in the input `EvmSketchInput`. However, the logic assumes that if receipts are not provided (i.e., the `receipts` field is `None`), then the block contained no logs. Consequently, the logs vector is set to an empty list regardless of whether receipts were omitted or the block truly contained no logs:

```
1
2  impl<'a, P: Primitives> ClientExecutor<'a, P> {
3      fn new(state_sketch: &'a EvmSketchInput) -> Result<Self, ClientError> {
4          ...
5
6          let logs = state_sketch
7              .receipts
8              .as_ref()
9              .unwrap_or(&vec![])
10             .iter()
11             .flat_map(|r| r.logs().to_vec())
12              .collect();
13
14         ...
15     }
16 }
```

This behavior conflates two distinct cases:

1. A block genuinely contains no logs (e.g., no transactions or transactions did not emit events).
2. Receipts were excluded from the input, and logs could not be determined.

This distinction becomes critical in contexts where logs are queried in zkVM applications. Developers that make use of this functionality will likely invoke the `get_logs` function to filter the internal logs vector. This function determines if an emitted log matches the input filter and so it could return an empty list of matching logs if no such logs were found *or* if the user provided no receipts in the input. This opens up a potential vector for proof forgery: a malicious user could omit receipts from the input and generate a proof that falsely demonstrates the absence of events.

Impact Consumers of the proof may misinterpret the absence of logs as an indication that no events were emitted, which may not be true. This could lead to incorrect or misleading proofs

to be emitted. For instance, a proof consumer may incorrectly conclude that a certain event did not occur, enabling malicious actors to craft proofs that omit receipts to suppress relevant log data.

Recommendation Disambiguate between "no logs present" and "logs unavailable" by making the `logs` field an Option. This would allow `get_logs` to throw an error if logs are queried but were not provided by the user.

Developer response The suggested fix was implemented by the developers.

5.1.2 V-SP1CC-VUL-002: Critical Anchor Consistency Checks Only Performed During Call Execution

Severity	High	**Commit**	6ccad76
Type	Data Validation	**Status**	Fixed
File(s)	./crates/client-executor/src/lib.rs		
Location(s)	./crates/client-executor/src/lib.rs:269-287		
Confirmed Fix At	804a19a		

Description The `ClientExecutor` is responsible for performing blockchain state validation and call execution. When a contract call is executed via the `execute` method, the associated `Anchor` is resolved using the `resolve()` function as shown below.

```
impl<'a, P: Primitives> ClientExecutor<'a, P> {
    pub fn execute(&self, call: ContractInput) -> eyre::Result<ContractPublicValues>
    {
        let cache_db = CacheDB::new(&self.witness_db);
        let tx_output =
            P::transact(&call, cache_db, self.anchor.header(), U256::ZERO, self.
    chain_spec.clone())
                .unwrap();
        let tx_output_bytes = tx_output.result.output().ok_or_eyre("Error decoding
    result")?;
        let resolved = self.anchor.resolve();

        let public_values = ContractPublicValues::new(
            call,
            tx_output_bytes.clone(),
            resolved.id,
            resolved.hash,
            self.anchor.ty(),
            self.genesis_hash,
        );

        Ok(public_values)
    }
}
```

This function performs critical consistency checks, such as ensuring the execution block matches the referenced beacon block (e.g., in the case of EIP-4788 or consensus anchors). However, this logic is only invoked when a contract call is executed. Applications that use the library solely for log retrieval or extracting block metadata do not trigger `resolve()` and therefore skip anchor consistency checks entirely.

As an example, the `get_logs()` method does not resolve the anchor, nor does the creation of the `ClientExecutor` itself enforce consistency between execution and beacon blocks. This allows proofs to be generated over inconsistent or manipulated anchor data if no contract call is made-particularly problematic in applications that only query event logs or metadata without executing a transaction. Notably, this occurs in the `events` example provided in the call contracts library.

Impact Applications using this library without executing a contract call may unknowingly operate over an unresolved and potentially inconsistent anchor. In such cases:

1. **No public values are committed to the journal**, meaning the authenticity of the block would not be enforced.
2. **No consistency checks are applied**, allowing mismatches between the execution block and beacon block (e.g., forged alignment between the state and a fake beacon root).

This enables an attacker to prove arbitrary information about a block by constructing inconsistent anchor inputs-e.g., using the correct block state but a mismatched beacon block root-without being detected.

The issue is particularly concerning for light clients or zero-knowledge applications that rely on event querying or read-only metadata, such as those inspired by the `events` example in the repository.

Recommendation Ensure that anchor resolution and consistency validation are enforced regardless of whether a contract call is executed.

Developer response The developers moved the anchor resolution into the creation of the client executor. The construction of this struct will therefore ensure that all state is consistent regardless of whether a call is invoked. Succinct should ensure their documentation states that the construction of the ClientExecutor is required (or strongly recommended) and that information should not be manually retrieved from the input.

5.1.3 V-SP1CC-VUL-003: Length of branch not enforced

Severity	Medium	**Commit**	6ccad76
Type	Data Validation	**Status**	Fixed
File(s)	`crates/client-executor/src/anchor.rs`		
Location(s)	crates/client-executor/src/anchor.rs:372-390		
Confirmed Fix At	10fcc99		

Description The `rebuild_merkle_root` function reconstructs a Merkle root by iteratively hashing a provided `leaf` with elements of a `branch` slice, guided by the specified `generalized_index`:

```
pub fn rebuild_merkle_root(leaf: B256, generalized_index: usize, branch: &[B256]) ->
    B256 {
    let mut current_hash = leaf;
    let depth = generalized_index.ilog2();
    let mut index = generalized_index - (1 << depth);
    let mut hasher = Sha256::new();

    for sibling in branch {
        ...
    }

    current_hash
}
```

The expected length of `branch` is derived from the Merkle depth, computed as `depth = generalized_index.ilog2()`. However, the function does not validate that the provided `branch` has the correct number of elements. Specifically:

- If `branch.len() < depth`, the function will stop early and return an invalid root.
- If `branch.len() > depth`, excess elements are silently ignored.
- If `branch.is_empty()`, the function simply returns the `leaf` unchanged, regardless of `generalized_index`.

These behaviors rely entirely on the caller to provide a well-formed Merkle proof and do not enforce structural correctness internally.

Impact An incorrect `branch` length can increase the likelihood of a pre-image attack, but can also result in the following issues:

- **API Misuse:** Callers may accidentally pass malformed proofs that yield incorrect roots without immediate failure.
- **Logic Bugs:** An unexpectedly short or long branch may signal an upstream bug that is silently ignored.
- **Weak Verification Contracts:** If this function is reused in lower-trust contexts (e.g., generic proof consumers), it may incorrectly accept invalid proofs.

In particular, returning the leaf when `branch` is empty can result in verification logic treating an unverified leaf as a valid Merkle root, which is a subtle but dangerous pitfall.

Recommendation Enforce `assert_eq!(branch.len(), depth )`.

Developer response The developers implemented the recommended fix.

5.1.4 V-SP1CC-VUL-004: Unused ‘state_requests‘ Field can be Manipulated

Severity	Medium	**Commit**	6ccad76
Type	Data Validation	**Status**	Fixed
File(s)	`./crates/client-executor/src/io.rs`		
Location(s)	./crates/client-executor/src/io.rs:55-56		
Confirmed Fix At	2c3d21d		

Description The `EvmSketchInput` struct contains a `state_requests` field, which is documented as representing the storage slots requested for a particular address:

```
pub struct EvmSketchInput {
    ...

    /// Requests to account state and storage slots.
    pub state_requests: HashMap<Address, Vec<U256>>,

    ...
}
```

However, this field does not appear to be referenced or used within the client or guest libraries responsible for validating input and the field does not appear in the public output. It appears that this field is used by the host to help determine the state required by the guest, but the guest itself does not consume it in any way.

Issue Because `state_requests` is unused and unvalidated, a malicious host can arbitrarily manipulate its contents without detection. This presents an opportunity for confusion or abuse, especially if users of the contracts call library assume the field was committed to or checked during execution. As an example, a user could decide to use this field to quickly determine whether a value in a contract was accessed.

Recommendation Remove or validate the `state_requests` field.

Developer response The developers implemented the recommended fix. Note that the RSP library has a similar issue that we would also recommend fixing as it does not appear that `state_requests` is used by the library either.

5.1.5 V-SP1CC-VUL-005: Public Values May Omit Execution Environment Metadata

Severity	Medium	**Commit**	6ccad76
Type	Data Validation	**Status**	Fixed
File(s)	`./crates/client-executor/src/lib.rs`		
Location(s)	./crates/client-executor/src/lib.rs:146-155		
Confirmed Fix At	0774a16		

The `ContractPublicValues` struct shown below includes information about the query and execution context, including fields such as `anchorHash`, `anchorType`, `callerAddress`, and `genesisHash`.

```
struct ContractPublicValues {
    uint256 id;
    bytes32 anchorHash;
    AnchorType anchorType;
    bytes32 genesisHash;
    address callerAddress;
    address contractAddress;
    bytes contractCalldata;
    bytes contractOutput;
}
```

The `genesisHash` field appears to represent a hash derived from the chain's configuration, however in the RSP library it is computed to be a hash of the chain-id for well-known chains or a hash of the custom chain configuration otherwise. In the case where the configuration corresponds to a well-known chain, however, one cannot distinguish between different versions of the same chain.

Impact The ambiguity between versions of the same chain can make it difficult to determine when a zkVM becomes outdated over time as the network is forked. Without an explicit commitment to the network fork configuration in the public values, such discrepancies may go undetected by verifiers. This can undermine trust in the correctness of proofs and open the door to subtle proof forgery.

Recommendation Consider modifying `genesisHash` or incorporating new fields to indicate either the entire network configuration state or the selected execution fork.

Developer response The developers modified the `genesisHash` to instead be a `chain_config_hash` that corresponds to the hash of the chain ID together with a fork identifier. This should allow consumers of a zkVM application that uses the library to very explicitly determine whether a stale fork is used in the future.

5.1.6 V-SP1CC-VUL-006: Insufficient Block Number Validation in Block Anchor Verification

Severity	Low	Commit	6ccad76
Type	Data Validation	Status	Fixed
File(s)	`contracts/src/ContractCall.sol`		
Location(s)	contracts/src/ContractCall.sol:50-58		
Confirmed Fix At	9e2bcb1		

Description The function `verifyBlockAnchor` is designed to verify that a provided `blockNumber` and corresponding `blockHash` form a valid anchor point within the EVM's retrievable block history:

```
1
2     function verifyBlockAnchor(uint256 blockNumber, bytes32 blockHash) internal view
      {
3         if (block.number - blockNumber > 256) {
4             revert ExpiredAnchor();
5         }
6
7         if (blockHash != blockhash(blockNumber)) {
8             revert AnchorMismatch();
9         }
10    }
```

The intention is to ensure:

- `blockNumber` is within the last 256 blocks (or fewer).
- `blockHash` matches the canonical block hash for that `blockNumber`.

However, this validation is incomplete:

- The check `block.number - blockNumber > 256` passes when `blockNumber == block.number`.
- But per the EVM specification, `blockhash(block.number)` always returns `0x0`. See EVM Codes - BLOCKHASH (0x40).

This means `verifyBlockAnchor(block.number, 0x0)` **passes the expiration check**, but then either:

- Incorrectly accepts a `blockHash` of `0x0` (if the `blockhash` equality check is missing or flawed).
- Or creates confusion if `blockhash(block.number)` is compared to `0x0` as expected but without clear error semantics.

Impact **Bypassing Freshness Checks**

- The current logic allows `blockNumber == block.number`, which is invalid for anchor purposes since `blockhash(block.number)` is always `0x0`.

Proof Verification Weakness

- An attacker could supply `(block.number, 0x0)` and pass the expiration check.
- If the `blockhash` equality check is skipped or wrongly implemented, invalid proofs could be treated as valid.

Recommendation Explicitly forbid `blockNumber >= block.number`

```
1 if (blockNumber >= block.number || block.number - blockNumber > 256)
```

Developer response The developers implemented the recommended fix.

5.1.7 V-SP1CC-VUL-007: The execute Function May Encourage Unsafe Usage Patterns

Severity	Low	**Commit**	6ccad76
Type	Usability Issue	**Status**	Fixed
File(s)	`./crates/client-executor/src/lib.rs`		
Location(s)	./crates/client-executor/src/lib.rs:269-287		
Confirmed Fix At	d163238, 6c99b61		

Description The `execute` method is used to simulate a smart contract call and produce a `ContractPublicValues` object that summarizes the outcome. The method returns a `Result<ContractPublicValues>`, which places the burden on the caller to:

1. Check for and correctly handle any execution errors (e.g., transaction reversion), and
2. Ensure the resulting public values are committed to the journal or otherwise made public for inclusion in the ZK proof.

Failure to perform either of these steps would result in incomplete or misleading proofs.

Impact Users of this API may accidentally ignore failed executions or omit important public data from the ZK proof. Because no enforcement is provided by the API itself, mistakes may be subtle and undetectable at proof verification. This could result in unverifiable or misleading proofs, particularly if the call reverted.

Recommendation Consider renaming `execute` to `execute_unsafe` then create a safe version of the API that invokes `execute_unsafe` but will panic on an error and possibly commit the public values.

Developer response The developers implemented the recommended fix by adding the `execute_and_commit` function.

5.1.8 V-SP1CC-VUL-008: Loss of Variant Type When Converting BeaconAnchorId to U256

Severity	Low	**Commit**	6ccad76
Type	Logic Error	**Status**	Fixed
File(s)	`./crates/client-executor/src/anchor.rs`		
Location(s)	./crates/client-executor/src/anchor.rs:245-252		
Confirmed Fix At	9155689		

Description The `BeaconAnchorId` enum is used to represent either a `Timestamp(u64)` or `Slot(u64)` variant:

```
pub enum BeaconAnchorId {
    Timestamp(u64),
    Slot(u64),
}
```

However, when converting this enum to a `U256`, the variant information is lost:

```
impl From<&BeaconAnchorId> for U256 {
    fn from(value: &BeaconAnchorId) -> Self {
        match value {
            BeaconAnchorId::Timestamp(t) => U256::from(*t),
            BeaconAnchorId::Slot(s) => U256::from(*s),
        }
    }
}
```

Because both variants are converted identically to a `U256` numeric value, there is no way for downstream verifiers to distinguish whether the anchor identifier refers to a timestamp or a slot. This creates ambiguity at verification time, particularly since the verifier must guess which variant was used. This is exacerbated by the fact that the on-chain verifier currently appears to accept only timestamps, making any slot-based input potentially unverifiable. Note that it is possible that the `AnchorType` is intended to indicate the anchor ID type, but the relationship is not enforced in the guest.

Impact Loss of variant type may result in incorrect verification or proof rejection due to ambiguity. For instance, if a proof is generated with a `Slot` identifier but the on-chain verifier expects a `Timestamp`, the proof may fail unexpectedly or be misinterpreted.

Recommendation Consider either enforcing the relationship between the `AnchorType` and the `BeaconAnchorId` type or adding a flag to indicate the type of ID.

Developer response The developers implemented the recommended fix by ensuring the expected `BeaconAnchorId` is used based on the `AnchorType`.

5.1.9 V-SP1CC-VUL-009: Missing Validation of genesisHash in Uniswap Proof Verification

Severity	Low	**Commit**	6ccad76
Type	Data Validation	**Status**	Fixed
File(s)	`./examples/uniswap/contracts/src/UniswapCall.sol`		
Location(s)	./examples/uniswap/contracts/src/UniswapCall.sol:27-38		
Confirmed Fix At	7253bba		

Description The `UniswapCall` contract verifies ZK proofs of off-chain contract calls using the SP1 framework. The `verifyUniswapCallProof` function decodes the `ContractPublicValues` and calls the `verify()` method to check the anchor, but does not validate the `genesisHash` field. The `genesisHash` is included in `ContractPublicValues` to allow downstream verifiers to confirm that the execution environment was initialized with the expected chain configuration (e.g., fork settings, chain ID). However, because `genesisHash` is not checked in `UniswapCall`, it is possible for a malicious user to produce a valid proof using the correct blockchain state but an incorrect or manipulated EVM configuration.

Impact Without `genesisHash` validation, proofs may be accepted even if they were generated with an altered REVM configuration. This can be particularly problematic in cases where differences in fork rules or chain parameters affect transaction semantics, leading to mismatches between expected and actual behavior.

For example, a proof could simulate a state transition using fork rules that exclude a recently introduced opcode or incorrectly apply gas costs, while still appearing valid based on state and anchor checks alone.

Recommendation Explicitly validate the `genesisHash` field in the `verifyUniswapCallProof` function by checking it against a known or whitelisted value.

Developer response The developer implemented the suggested recommendation.

5.1.10 V-SP1CC-VUL-010: Misleading Error Message on Transaction Reversion

Severity	Warning	**Commit**	6ccad76
Type	Usability Issue	**Status**	Fixed
File(s)	`./crates/client-executor/src/lib.rs`		
Location(s)	./crates/client-executor/src/lib.rs:274-275		
Confirmed Fix At	4046754		

Description The `execute` method of the `ClientExecutor` is responsible for simulating a contract call and returning the corresponding public values. When the transaction result is retrieved, the code uses the following logic to extract the output:

```
impl<'a, P: Primitives> ClientExecutor<'a, P> {
    pub fn execute(&self, call: ContractInput) -> eyre::Result<ContractPublicValues>
    {
        ...

        let tx_output_bytes = tx_output.result.output().ok_or_eyre("Error decoding
    result")?;

        ...
    }
}
```

If the transaction reverts or halts during execution, `output()` will return `None`, causing this line to return an error with the message `"Error decoding result"`. However, in this context, the absence of output is not caused by a decoding failure but rather by the EVM transaction reverting. As a result, the error message may mislead developers or users into thinking there was an issue with output formatting or serialization, rather than the actual failure reason.

Issue Users may misinterpret the source of transaction failure due to the misleading error message. More importantly, the lack of exposure of the revert reason obscures important information that could help diagnose failures. In debugging or proof-generation scenarios, this could lead to confusion or improper handling of legitimate transaction failures.

Recommendation Update the error handling to more accurately reflect the cause of the failure. For example, change the error message to indicate that the transaction reverted or halted. If possible, expose the actual revert reason provided by the EVM execution engine to improve developer visibility and debugging capabilities.

Developer response The developers updated the error result to indicate the type and reason of the failure.

5.1.11 V-SP1CC-VUL-011: Unused BeaconBlockField Enum

Severity	Warning	**Commit**	6ccad76
Type	Dead Code	**Status**	Fixed
File(s)	`./crates/client-executor/src/anchor.rs`		
Location(s)	./crates/client-executor/src/anchor.rs:320-323		
Confirmed Fix At	f9ad661		

Description The `BeaconBlockField` enum appears to be designed as an abstraction for referring to fields in a beacon block that may be accessed via Merkle proofs:

```
pub enum BeaconBlockField {
    BlockHash,
    StateRoot,
}
```

The enum maps each variant to a corresponding generalized Merkle tree leaf index (e.g., `BLOCK_HASH_LEAF_INDEX` or `STATE_ROOT_LEAF_INDEX`). Despite this, the enum is never used in the codebase-callers reference the constants (e.g., `BLOCK_HASH_LEAF_INDEX`) directly instead.

Recommendation Consider removing the enum or replace direct constant accesses with enum references.

Developer response The `BeaconBlockField` is used in the host so the developers moved the struct from the guest crate to the host crate.

5.1.12 V-SP1CC-VUL-012: get_beacon_root_from_state Deviates from EIP-4788 Specification

Severity	Warning	**Commit**	6ccad76
Type	Data Validation	**Status**	Fixed
File(s)	`./crates/client-executor/src/anchor.rs`		
Location(s)	./crates/client-executor/src/anchor.rs:409-417		
Confirmed Fix At	8e21789		

Description The `get_beacon_root_from_state` function is intended to retrieve a historical beacon root from Ethereum state based on the logic defined in EIP-4788. The function currently computes the index as follows:

```
pub fn get_beacon_root_from_state(state: &EthereumState, timestamp: U256) -> B256 {
    let db = TrieDB::new(state, HashMap::default(), HashMap::default());
    let timestamp_idx = timestamp % HISTORY_BUFFER_LENGTH;
    let root_idx = timestamp_idx + HISTORY_BUFFER_LENGTH;

    let root = db.storage_ref(BEACON_ROOTS_ADDRESS, root_idx).unwrap();

    root.into()
}
```

However, this implementation omits two important checks defined in the EIP-4788 specification and reflected in on-chain behavior:

1. **Zero Timestamp Check:** A timestamp value of 0 is invalid and should be rejected. The current implementation does not enforce this and proceeds with the lookup, which may return incorrect data.
2. **Timestamp Consistency Check:** EIP-4788 requires that the value at `timestamp_idx` in the contract's storage corresponds to the actual timestamp of the returned beacon root. This ensures that the root retrieved from `root_idx` corresponds to the correct timestamp. This consistency check is missing from the implementation.

Although downstream validation of the retrieved root (e.g., via Merkle proof or commitment) may mitigate some risks, omitting these checks deviates from spec and increases the likelihood of subtle inconsistencies, especially in edge cases or malformed inputs.

Impact Failure to follow EIP-4788's canonical retrieval and validation logic may result in incorrect or unverifiable beacon root values being used in proofs or commitments. In particular, allowing a timestamp of 0 may lead to unintended lookups, while not validating the stored timestamp may allow mismatched or stale roots to be accepted silently.

Recommendation Update the function to adhere fully to the EIP-4788 spec.

Developer response The developers implemented the recommended fix.

5.1.13 V-SP1CC-VUL-013: Inconsistent AnchorType in Guest and Verifier Contract

Severity	Warning	**Commit**	6ccad76
Type	Usability Issue	**Status**	Fixed
File(s)	`crates/client-executor/src/lib.rs`, `contracts/src/ContractCall.sol`		
Location(s)	crates/client-executor/src/lib.rs:140,contracts/src/ContractCall.sol:18-20		
Confirmed Fix At	a78ba0a		

Description The `AnchorType` enum is used in both Solidity and Rust code to denote the type of anchor used for a contract call proof. However, the variants defined in each environment are inconsistent:

Rust

```
enum AnchorType { BlockHash, Eip4788, Consensus }
```

Solidity

```
enum AnchorType { BlockHash, BeaconRoot }
```

This inconsistency may lead to misinterpretation of the `anchorType` field within the `ContractPublicValues` structure, which is expected to be encoded identically between the two environments.

Impact Discrepancies between the guest and verifier smart contract may cause difficulties for users that want to implement the missing behaviors in the verifier.

Recommendation Ensure that the `AnchorType` enum is defined identically in both guest and verifier smart contract.

Developer response The developers implemented the recommended fix.

5.1.14 V-SP1CC-VUL-014: No Genesis Hash Validation Limits Assurance of Chain Configuration

Severity	Warning	**Commit**	6ccad76
Type	Usability Issue	**Status**	Fixed
File(s)	`./contracts/src/ContractCall.sol`		
Location(s)	./contracts/src/ContractCall.sol:34-75		
Confirmed Fix At	a336a64		

Description The `ContractCall` library is responsible for verifying `ContractPublicValues` that result from off-chain execution, including the `anchorHash`, `anchorType`, and various contextual fields such as `genesisHash`. While the `verify` function performs validation of anchor information through `verifyBlockAnchor` and `verifyBeaconAnchor`, there is no logic to validate that the provided `genesisHash` corresponds to a known or expected chain configuration.

This omission means that callers must rely on external validation or assumptions to determine whether the `genesisHash` corresponds to the intended network or EVM configuration used during off-chain execution. Without this, it is possible to construct otherwise valid-looking proofs against arbitrary forks or misconfigured chains, with no on-chain enforcement of the expected genesis configuration.

Impact If a user or verifier fails to check the `genesisHash` manually, they may inadvertently accept proofs generated with an incorrect or malicious EVM environment. This could lead to misinterpretation of contract behavior, particularly around forks, differing gas costs, or opcode semantics. Since the genesis configuration underpins the EVM environment (e.g., fork rules, chain ID), failing to verify it weakens the trust guarantees of the proof system.

Recommendation Provide a utility function or mapping to allow consumers to validate the `genesisHash` against an expected or whitelisted set of known configurations.

Developer response The developers added a utility to validate the chain configuration.

Glossary

AMM Automated Market Maker. 26

Uniswap One of the most famous deployed AMMs. See `https://uniswap.org` to learn more. 1

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. 26

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