# Soroban-poseidon

*Stellar*

HALBORN

# Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ABOUT THIS REPORT

ASSESSMENT TYPE
Assurance Assessment

DATE OF ENGAGEMENT
Feb 9, 2026 - Mar 7, 2026

SOURCE CODE
rs-soroban-poseidon

ASSESSED COMMIT ID
2959369

SEVERITY BREAKDOWN

| 7 | 6 | 1 | 0 |
|---|---|---|---|
| All Findings | Solved | Risk Accepted | Acknowledged |


## INTRODUCTION

This security assessment was commissioned by **Stellar**, a blockchain network designed to facilitate fast, low-cost cross-border payments and asset transfers.

The review was conducted by Halborn's experienced security team, focusing on the rs-soroban-poseidon repository's Poseidon and Poseidon2 cryptographic hash implementations —specifically the sponge constructions, parameter tables, and public hashing interfaces used in Soroban smart contracts—corresponding to the v25.0.0 release (commit 2959369), from February 9th, 2026, to March 06th, 2026.

## ASSESSMENT SUMMARY

The team at Halborn assigned a full-time security engineer to verify the security of the cryptographic hashing library used in Soroban smart contracts. The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this assessment is to:

- Assess collision resistance, domain separation, and input-encoding safety of the provided sponge-based hash APIs in a smart-contract setting.
- Validate compatibility claims against external reference implementations and test vectors (circom for Poseidon, noir for Poseidon2), including parameter provenance.
- Evaluate runtime safety and operational risks in Soroban (panic/abort behavior, input validation, and compute-budget/performance characteristics).

In summary, Halborn identified some improvements to reduce the likelihood and impact of risks, which were addressed by the **Stellar team**. The main recommendations were the following:

- `Prevent non-injective hashing under a single configuration by enforcing fixed-arity inputs or adopting an injective length/padding scheme for variable-length inputs, and document the supported hashing domains clearly.`
- `Implement multi-block absorption for Poseidon2 so inputs longer than a single rate can be processed safely, restoring expected ecosystem compatibility and eliminating avoidable abort paths.`
- `Provide a fallible, contract-friendly error surface for invalid inputs (e.g., checked APIs and/or shared validation helpers) to avoid unrecoverable panics and improve integration safety.`
- `Reduce per-invocation compute cost for repeated hashing by enabling parameter reuse/caching and by clearly steering integrators away from single-use convenience functions in loops.`
- `Tighten documentation around compatibility boundaries and edge-case semantics (parameter sources per field, empty-input behavior, and state/index conventions) to reduce integration mismatches and future maintenance risk.`

## TEST APPROACH AND METHODOLOGY

A layered review approach was followed using the artifacts supplied in the repository and accompanying documentation. The sequence of work was as follows:

- Review the public API surface and documentation for input preconditions, panic behavior, and compatibility guarantees across supported configurations and fields.
- Perform manual code review of sponge initialization/absorb/squeeze logic and parameter handling, focusing on injective encoding, domain separation, and state transitions.
- Cross-check parameter provenance against referenced generation scripts and external implementations, and validate behavior using included cross-ecosystem test vectors.

- Develop and review proof-of-concept cases and tests for identified issues, and evaluated practical impact under Soroban execution and budgeting constraints.

## Risk Methodology

Halborn assesses the severity of findings using either the Common Vulnerability Scoring System (CVSS) framework or the Impact/Likelihood Risk scale, depending on the engagement. CVSS is an industry standard framework for communicating characteristics and severity of vulnerabilities in software. Details can be found in the CVSS Specification Document published by F.I.R.S.T.

Vulnerabilities or issues observed by Halborn scored on the Impact/Likelihood Risk scale are measured by the LIKELIHOOD of a security incident and the IMPACT should an incident occur. This framework works for communicating the characteristics and impacts of technology vulnerabilities. The quantitative model ensures repeatable and accurate measurement while enabling users to see the underlying vulnerability characteristics that were used to generate the Risk scores. For every vulnerability, a risk level will be calculated on a scale of 5 to 1 with 5 being the highest likelihood or impact.

### RISK SCALE - LIKELIHOOD

5 - Almost certain an incident will occur.
4 - High probability of an incident occurring.
3 - Potential of a security incident in the long term.
2 - Low probability of an incident occurring.
1 - Very unlikely issue will cause an incident.

### RISK SCALE - IMPACT

5 - May cause devastating and unrecoverable impact or loss.
4 - May cause a significant level of impact or loss.
3 - May cause a partial impact or loss to many.
2 - May cause temporary impact or loss.
1 - May cause minimal or un-noticeable impact.

The risk level is then calculated using a sum of these two values, creating a value of 10 to 1 with 10 being the highest level of security risk.

| CRITICAL | HIGH | MEDIUM | LOW | INFORMATIONAL |
|---|---|---|---|---|

**10** - CRITICAL
**9 - 8** - HIGH

**7 - 6** - MEDIUM

**5 - 4** - LOW

**3 - 1** - VERY LOW AND INFORMATIONAL

## Scope

REPOSITORY

(a) Repository: rs-soroban-poseidon

(b) Assessed Commit ID: 295936916b7379c9e3f5727be4dcf055a0c7481c

(c) Items in scope:

- src 7 files
  - poseidon 3 files
    - mod.rs Rust
    - params.rs Rust
    - sponge.rs Rust
  - poseidon2 3 files
    - mod.rs Rust
    - params.rs Rust
    - sponge.rs Rust
  - lib.rs Rust

REMEDIATION COMMIT ID:

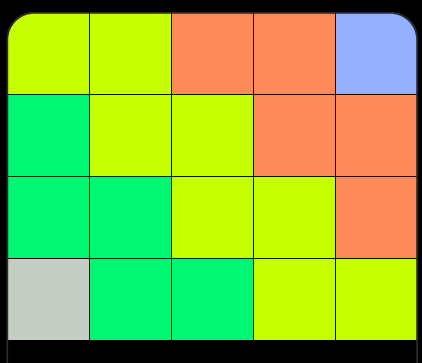
ceb20d3593fc4a8a951a7e99d8fa2344f8250a8c

08fa50d9fcbf8e150ffd11f51b445151867c79fa

Out-of-Scope: New features/implementations after the remediation commit IDs.

## Assessment Summary & Findings Overview

IMPACT X LIKELIHOOD



| # | TITLE | SEVERITY | SCORE | STATUS |
|---|---|---|---|---|
| HAL-001 | Poseidon V1 Accepts Variable-Length Inputs Without Injective Padding (Suffix-Zero Collisions) | High | 7.5 | SOLVED 05/04/2026 |
| HAL-002 | Poseidon2 Sponge Lacks Multi-Round Absorption, Causing Runtime Panics And Breaking Noir/Aztec Compatibility | Low | 3.4 | SOLVED 05/08/2026 |
| HAL-003 | Poseidon Convenience Hash APIs Rebuild Parameter Tables Per Call, Risking Budget Exhaustion In Multi-Hash Invocations | Low | 3.4 | RISK ACCEPTED 05/08/2026 |
| HAL-004 | Missing Diagnostic Message in Rate-Bound Assert | Informational | — | SOLVED 05/11/2026 |
| HAL-005 | Poseidon2 Sponge Comments Use Ambiguous Inclusive Ranges For State Indices | Informational | — | SOLVED 05/08/2026 |
| HAL-006 | Poseidon Documentation Does Not Distinguish BN254 and BLS12-381 Parameter Provenance | Informational | — | SOLVED 05/08/2026 |
| HAL-007 | Poseidon2 Hash Accepts Empty Inputs Without Documented Semantics | Informational | — | SOLVED 05/08/2026 |

## Findings & Tech Details

HAL-001 SOLVED

# Poseidon V1 Accepts Variable-Length Inputs Without Injective Padding (Suffix-Zero Collisions)

High · 7.5 AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:H/D:N/Y:N

DESCRIPTION

`PoseidonSponge::compute_hash()` for `T ≥ 3` (RATE ≥ 2) lacks injective encoding for variable-length inputs because `absorb()` overwrites only the provided rate positions on a zero-initialized state, making writing field element `0` to a slot indistinguishable from leaving it untouched. The `T=2` (RATE=1) configuration is not affected: only one input is accepted at that arity and the empty-input edge case is already blocked.

Any caller who can influence the inputs to a protocol component using `compute_hash()` for variable-length vectors can exploit this as follows:

1. Choose any message `m = [x₁, …, xₖ]` with `k < RATE`, leaving at least one unfilled rate slot.
2. Construct `m' = [x₁, …, xₖ, 0, …, 0]` by appending up to `RATE − k` trailing field zeros.
3. Submit `m'`; both messages produce the identical pre-permutation state `[0, x₁, …, xₖ, 0, …, 0]` and therefore the same digest.

The collision is deterministic and requires no computation beyond appending zeros. Notably, the existing `inputs.is_empty()` panic guard demonstrates developer awareness of this exact class of issue — the inline comment correctly identifies that `hash([]) == hash([0])` for the same root cause (zero-initialized rate, unwritten slots remain zero) — but the fix is scoped only to the empty-input edge case, leaving the general variable-length path unprotected.

## Code Location

`reset_state()` in `src/poseidon/sponge.rs` initializes all rate elements to zero; any slot not written by `absorb()` retains this value:

RUST 179–187

```rust
fn reset_state(&mut self) {
    // initialize the state with CAPACITY elements (CAPACITY = 1 in our sponge) at the 0-th element
    // The initial value is 0 for standard Poseidon
    let iv = U256::from_u32(&self.env, 0);
    self.state = vec![&self.env, iv];
    for _ in 0..Self::RATE {
        self.state.push_back(U256::from_u32(&self.env, 0));
    }
}
```

`absorb()` in `src/poseidon/sponge.rs` overwrites only the positions covered by the input slice, leaving remaining rate slots at zero — writing `0` explicitly is identical to not writing at all:

RUST 219–227

```rust
pub(crate) fn absorb(&mut self, inputs: &Vec<U256>) {
    assert!(inputs.len() <= Self::RATE);
    let modulus = F::modulus(&self.env);
    for i in 0..inputs.len() {
        let v = inputs.get_unchecked(i);
        assert!(v < modulus, "input exceeds field modulus");
        self.state.set(i + CAPACITY, v);
    }
}

```

`src/lib.rs` documents `T ≥ inputs.len() + 1` rather than requiring equality, and the `compute_hash()` docstring claims circom compatibility while not enforcing fixed arity — both actively encourage variable-length usage under a single `T`:

RUST 64

```rust
/// - `T`: State size. Must be ≥ `inputs.len() + 1` (rate = T-1, capacity = 1).
```

## Impact

Any higher-level construction relying on `compute_hash()` to uniquely bind an input sequence is broken: two distinct committed values open to the same commitment (binding failure), two distinct Merkle leaf preimages map to the same leaf hash (inclusion proof soundness failure), and two distinct key-derivation input tuples yield the same derived key (impersonation or key confusion).

PROOF OF CONCEPT

## Code

The PoC code has been added to the existing test suite located in the `src/tests/poseidon.rs` file:

RUST

```rust
// With T=3 (RATE=2), variable-length inputs are effectively right-padded with zeros,
// so [x] and [x, 0] collide.
#[test]
fn test_poseidon_bn254_suffix_zero_collision_poc_t3() {
    let env = Env::default();

    let x = U256::from_u32(&env, 42);

    let inputs_short = vec![&env, x.clone()];
    let inputs_with_trailing_zero = vec![&env, x, U256::from_u32(&env, 0)];

    // Sanity: distinct messages (different lengths)
```

```
    assert_ne!(inputs_short.len(), inputs_with_trailing_zero.len());

    let h1 = poseidon_hash::<3, BnScalar>(&env, &inputs_short);
    let h2 = poseidon_hash::<3, BnScalar>(&env, &inputs_with_trailing_zero);

    // Collision by construction (suffix-zero collapse)
    assert_eq!(h1, h2);
}
```

### Execution

Run the following command:

```
SH

cargo test --package soroban-poseidon --lib --
tests::poseidon::test_poseidon_bn254_suffix_zero_collision_poc_t3 --exact
```

### Result

```
SH

running 1 test
test tests::poseidon::test_poseidon_bn254_suffix_zero_collision_poc_t3 ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 53 filtered out; finished in
0.01s
```

### Analysis

The proof-of-concept demonstrates a critical collision vulnerability in the Poseidon hash function with `T=3`, where variable-length inputs padded with zeros produce identical hashes despite different input lengths, compromising the hash's collision resistance property.

#### RECOMMENDATION

To mitigate this issue, we recommend the following improvements:

- Enforce fixed arity at the API level by requiring `inputs.len() == RATE` in `compute_hash()` or `absorb()`, ensuring callers select the `T` that exactly matches their input count, consistent with circom's fixed-arity circuit design where `Poseidon(n)` always takes exactly `n` inputs under `T = n + 1`.
- Correct the `lib.rs` documentation from `T: State size. Must be ≥ inputs.len() + 1` to `T: State size. Must be == inputs.len() + 1`, and explicitly state that `compute_hash()` is defined only for exactly `RATE` inputs per `T`.
- If variable-length hashing under a single `T` is a required feature, adopt the length-encoded capacity IV already used by `Poseidon2Sponge::compute_hash()` in this codebase `inputs.len() << 64`), which provides length-domain separation — noting that this produces outputs incompatible with circom and must be documented as a distinct hash domain.

REMEDIATION COMMENT

Addressed in: ceb20d3593fc4a8a951a7e99d8fa2344f8250a8c

**SOLVED:** The **Stellar team** solved this finding in the specified commit by changing the `absorb()` assertion from `inputs.len() <= Self::RATE` to `inputs.len() == Self::RATE`, enforcing fixed arity and eliminating suffix-zero collisions across all affected code locations.

HAL-002 SOLVED

# Poseidon2 Sponge Lacks Multi-Round Absorption, Causing Runtime Panics And Breaking Noir/Aztec Compatibility

Low · 3.4 AO:A/AC:L/AX:M/R:N/S:U/C:N/A:M/I:N/D:N/Y:N

DESCRIPTION

`Poseidon2Sponge::compute_hash()` in `src/poseidon2/sponge.rs` hard-panics when `inputs.len() > T − 1` because multi-round (multi-block) sponge absorption is not implemented.

Any caller who provides more inputs than `RATE` will trigger this panic. If a Soroban contract built on this library forwards a user-controlled input vector to `poseidon2_hash` without a prior length check, any user of that contract can abort their own invocation by exceeding the rate limit. The panic is documented API behavior, the risk is that downstream contracts may not guard against it:

1. The caller invokes `poseidon2_hash::<4, BnScalar>(&env, &inputs)` with `inputs.len() == 4`.
2. `compute_hash()` calls `absorb(&inputs)` once with the full vector.
3. `absorb()` asserts `inputs.len() <= RATE` (here `RATE = 3`) and panics.

This library does not provide a supported in-crate workaround for hashing 4 inputs under `T=4`: Poseidon2 is only implemented for `T ∈ {2, 3, 4}`, and changing `T` would not reproduce noir/Aztec `T=4` multi-block reference behavior.

## Code Location

`src/tests/poseidon2.rs` includes (but ignores) the Aztec/barretenberg 4-input `T=4` reference case pending multi-round absorption:

RUST 19-35

```rust
// This test matches barretenberg test case for hashing 4 inputs:
// TODO: Re-enable once multi-round absorption is implemented
#[test]
#[ignore]
fn test_poseidon2_hash() {
    // ...
    let inputs = vec![
        &env,
        input_value.clone(),
        input_value.clone(),
        input_value.clone(),
        input_value,
    ]; // len = 4, RATE = 3
    let mut sponge = Poseidon2Sponge::<4, BnScalar>::new(&env);
    let result = sponge.compute_hash(&inputs); // panics before producing any output
    assert_eq!(result, expected);
}

```

`src/poseidon2/sponge.rs` panics if the input vector exceeds the rate:

RUST 176–185

```rust
pub(crate) fn absorb(&mut self, inputs: &Vec<U256>) {
    // Absorb into rate portion of state (positions 0..RATE)
    assert!(inputs.len() <= Self::RATE);
    let modulus = F::modulus(&self.env);
    for i in 0..inputs.len() {
        let v = inputs.get_unchecked(i);
        assert!(v < modulus, "input exceeds field modulus");
        self.state.set(i, v);
    }
}
```

`compute_hash()` performs only a single absorption (no chunking / interleaved permutations):

RUST 210–215

```rust
pub fn compute_hash(&mut self, inputs: &Vec<U256>) -> U256 {
    let iv = U256::from_u128(&self.env, (inputs.len() as u128) << 64);
    self.reset_state(iv);
    self.absorb(inputs);
    self.squeeze()
}
```

`src/lib.rs` documents the public wrapper's panic condition as "rate exceeded":

RUST 145–146

```rust
/// - if `inputs.len() > T - 1` (rate exceeded)
/// - if any input value >= the field modulus (inputs must be valid field elements)
```

## Impact

- Soroban contracts that call `poseidon2_hash` / `compute_hash` without first validating input length will abort on any call where `inputs.len() > T − 1`. The primary impact is the Noir/Aztec incompatibility: multi-block input sizes cannot be hashed regardless of how the library is called.
- Contracts that need to verify noir/Aztec `T=4` Poseidon2 hashes for vectors longer than the rate (e.g., the 4-input barretenberg reference case) cannot do so with this library as-is.

RECOMMENDATION

To mitigate this issue, we recommend the following improvements:

- Implement multi-round absorption in `compute_hash()` by processing inputs in `RATE`-sized chunks and invoking the permutation between chunks, while keeping the capacity IV initialized as `inputs.len() << 64`.
- For absorption after the first block, combine inputs into the existing rate state via field addition (rather than overwriting state positions) to match standard sponge absorption semantics.
- Re-enable `test_poseidon2_hash` (Aztec/barretenberg vector) once implemented, and update `poseidon2_hash` docs to explicitly call out the current single-block limitation.

REMEDIATION COMMENT

Addressed in: 08fa50d9fcbf8e150ffd11f51b445151867c79fa

**SOLVED:** The **Stellar team** solved this finding in the specified commit by following the mentioned recommendation.

---

HAL-003 RISK ACCEPTED

# Poseidon Convenience Hash APIs Rebuild Parameter Tables Per Call, Risking Budget Exhaustion In Multi-Hash Invocations

● Low · 3.4 AO:A/AC:L/AX:M/R:N/S:U/C:N/A:M/I:N/D:N/Y:N

DESCRIPTION

`poseidon_hash()` and `poseidon2_hash()` in `src/lib.rs` instantiate a brand-new sponge on every call, and the sponge constructors unconditionally rebuild the full parameter tables (MDS/diagonal matrix and round constants) from hard-coded byte literals with no caching or reuse.

Any transaction submitter who invokes a contract function that performs repeated hashing via these convenience APIs can trigger the high-cost path:

1. The integrating contract loops and calls `poseidon_hash::<T, F>(env, &inputs)` (or `poseidon2_hash`) once per item (e.g., per leaf/commitment).
2. Each iteration constructs a fresh sponge, and `Sponge::new()` rebuilds the parameter vectors via `get_*` functions (materializing many `U256` and `Vec` host objects).
3. Repeated parameter rebuilding is metered; with enough iterations and/or large configurations, the invocation traps due to budget exhaustion.

## Code Location

`src/lib.rs::poseidon_hash` and `src/lib.rs::poseidon2_hash` create a fresh sponge per call:

RUST 122–128

```rust
pub fn poseidon_hash<const T: u32, F: Field>(env: &Env, inputs: &Vec<U256>) -> U256
where
    PoseidonSponge<T, F>: PoseidonConfig<T, F>,
{
    let mut sponge = PoseidonSponge::<T, F>::new(env);
    sponge.compute_hash(inputs)
}
```

RUST 192–198

```rust
pub fn poseidon2_hash<const T: u32, F: Field>(env: &Env, inputs: &Vec<U256>) -> U256
where
    Poseidon2Sponge<T, F>: Poseidon2Config<T, F>,
{
    let mut sponge = Poseidon2Sponge::<T, F>::new(env);
    sponge.compute_hash(inputs)
}
```

`src/poseidon/sponge.rs::PoseidonSponge::new` rebuilds MDS and round constants unconditionally:

RUST 189–204

```rust
pub fn new(env: &Env) -> Self {
    let params = PoseidonParams {
        rounds_f: <Self as PoseidonConfig<T, F>>::ROUNDS_F,
        rounds_p: <Self as PoseidonConfig<T, F>>::ROUNDS_P,
        mds: <Self as PoseidonConfig<T, F>>::get_mds(env),
        rc: <Self as PoseidonConfig<T, F>>::get_rc(env),
    };
    let mut inner = Self {
        env: env.clone(),
        state: vec![env],
        params,
        _phantom: core::marker::PhantomData,
    };
    inner.reset_state();
    inner
}
```

`src/poseidon/params.rs::get_rc_bn254_t_6` is representative: it constructs nested vectors by converting many hard-coded `bytesn!` literals into `U256` values on every invocation:

RUST 1639–1651

```rust
pub(crate) fn get_rc_bn254_t_6(e: &Env) -> Vec<Vec<U256>> {
    vec![e,
        vec![e,
            U256::from_be_bytes(e, &bytesn!(e, 0x1448614598e00f98e7ae7dea45fbd83bd968653ef8390cde2e86b706ad40c651).into()),
            U256::from_be_bytes(e, &bytesn!(e, 0x0ab7b291388e5c9e43c0dc1f591fb83ecdb65022e1b70af43b8a7b40c1dff7c3).into()),
            U256::from_be_bytes(e, &bytesn!(e, 0x2b7cbb217896f52c9a8c088e654af21e84cde754a3cef5b15c4d5466612d6adf).into()),
            U256::from_be_bytes(e, &bytesn!(e, 0x2bc6b0ddbe1d701b6570428bdc1ca1bf0da59ff3bbbb95fc2bc71c0c6e67a65c).into()),
            U256::from_be_bytes(e, &bytesn!(e, 0x123a55a31980384f3d20b2cecbc44ed60c38c11f7d20e9271efab9a905eefd3c).into()),
            U256::from_be_bytes(e, &bytesn!(e, 0x037501cc8c9dc819309a769f4df098e588b01858bc8eb7e279e2883be9fb8c53).into())
        ],
        // ... many more rows
    ]
}
```

`src/poseidon/sponge.rs` acknowledges the lack of a storage-backed caching layer:

RUST 26–28

```rust
// Internal struct storing the Poseidon constants, in the future we can make
// this a #[contracttype], which can be stored as contract data (to reduce the
// actual contract size)
```

## Impact

- Any Soroban contract invocation that performs multiple hashes via `poseidon_hash` / `poseidon2_hash` may exceed the fixed budget and abort mid-execution.
- The transaction submitter pays for the failed invocation (fees are determined from simulation), and higher-level workflows that rely on multi-hash loops can become unreliable unless they reuse a sponge instance.

RECOMMENDATION

To mitigate this issue, we recommend the following improvements:

- Add prominent warnings in `poseidon_hash` and `poseidon2_hash` docstrings (and crate-level docs) stating that these functions are intended for single-use hashing and are not suitable for multi-hash loops; include a short example that reuses `PoseidonSponge` / `Poseidon2Sponge` across multiple `compute_hash()` calls.
- Provide a cached-constructor option (e.g., `PoseidonSponge::new_cached` / `Poseidon2Sponge::new_cached`) that stores and loads parameter tables from instance storage (leveraging `#[contracttype]` as already noted), so repeated calls can avoid rebuilding constants from literals.
- Add a budget-sensitive regression test (separate from correctness tests) that exercises a small multi-hash loop using the convenience APIs and asserts a documented budget bound or a deliberate fast-fail, preventing accidental cost regressions from reaching production.

REMEDIATION COMMENT

Addressed in: 08fa50d9fcbf8e150ffd11f51b445151867c79fa

**RISK ACCEPTED:** The **Stellar team** made a business decision to accept the residual risk of this finding as it stems from a known design limitation. To mitigate potential misuse, they added explicit **WARNING** performance notes to the `poseidon_hash` and `poseidon2_hash` docstrings, directing integrators to construct the sponge once outside multi-hash loops, while maintaining the underlying per-call parameter rebuild behavior and declining to implement a cached constructor.

---

HAL-004 SOLVED

# Missing Diagnostic Message in Rate-Bound Assert

● Informational · AO:A/AC:L/AX:L/R:F/S:U/C:N/A:N/I:N/D:N/Y:N

DESCRIPTION

The `assert!(inputs.len() <= Self::RATE);` statements in `absorb()` in both **src/poseidon/sponge.rs** and **src/poseidon2/sponge.rs** carries no panic message. The adjacent assertion sites in both files include descriptive messages, making the rate-bound assert the only message-less check in the validation path.

## Code Location

Poseidon v1 (`src/poseidon/sponge.rs`) has three `assert!()` sites; note the rate-bound assert carries no error message, unlike the other two:

RUST

```rust
// compute_hash() - Line 254
assert!(!inputs.is_empty(), "Poseidon: inputs cannot be empty");

// absorb() - Lines 220,224
assert!(inputs.len() <= Self::RATE);           // no message
assert!(v < modulus, "input exceeds field modulus");
```

Poseidon2 (`src/poseidon2/sponge.rs`) has two `assert!()` sites — the empty-input guard is absent:

RUST

```rust
// absorb() - Lines 178,182
assert!(inputs.len() <= Self::RATE);           // no message
assert!(v < modulus, "input exceeds field modulus");
```

## Impact

Developer experience only. When the rate-bound assertion fires during testing or simulation, the panic output provides no context on which precondition was violated or what the applicable limit is, making the failure harder to diagnose than the adjacent assertion sites in the same code path.

RECOMMENDATION

To mitigate this issue, we recommend the following improvements:

- Add an explicit panic message to `assert!(inputs.len() <= Self::RATE)` in both `poseidon/sponge.rs` and `poseidon2/sponge.rs` to match the other assertion sites and improve developer diagnostics during testing and simulation.

- Retain the existing panic-based API as-is: it is correctly documented under `# Panics` and is the appropriate interface for callers that treat invalid inputs as programmer error.

REMEDIATION COMMENT

Addressed in: 08fa50d9fcbf8e150ffd11f51b445151867c79fa

**SOLVED:** The **Stellar team** solved this finding in commits ceb20d3 and 08fa50d by adding a diagnostic message to the Poseidon rate-bound assert and removing the Poseidon2 rate-bound assert entirely in favor of multi-round absorption.

HAL-005 SOLVED

# Poseidon2 Sponge Comments Use Ambiguous Inclusive Ranges For State Indices

● Informational · AO:A/AC:L/AX:L/R:F/S:U/C:N/A:N/I:N/D:N/Y:N

DESCRIPTION

`Poseidon2Sponge` uses the notation "positions 0..RATE" in inline comments to describe the rate portion of the state, but the valid indices are `0` through `RATE-1` (exclusive upper bound), with the capacity element at index `T-1`. Readers may misinterpret "0..RATE" as an inclusive range and form an incorrect mental model of the state layout.

## Code Location

`reset_state()` in `src/poseidon2/sponge.rs` comments "0..RATE" while the loop pushes exactly `RATE` elements at indices `0..RATE-1`, and the capacity IV lands at index `T-1`:

RUST 134-143

```rust
fn reset_state(&mut self, iv: U256) {
    // State layout: [rate elements...][capacity element]
    // Rate elements are at positions 0..RATE, capacity (IV) is at position T-1 (last)
    self.state = vec![&self.env];
    for _ in 0..Self::RATE {
        self.state.push_back(U256::from_u32(&self.env, 0));
    }
    // IV goes at the last position (capacity element)
    self.state.push_back(iv);
}

```

`absorb()` in `src/poseidon2/sponge.rs` repeats the same ambiguous "0..RATE" caption while `set(i, v)` indexes `0` through `inputs.len()-1`:

RUST 176-185

```rust
pub(crate) fn absorb(&mut self, inputs: &Vec<U256>) {
    // Absorb into rate portion of state (positions 0..RATE)
    assert!(inputs.len() <= Self::RATE);
    let modulus = F::modulus(&self.env);
    for i in 0..inputs.len() {
        let v = inputs.get_unchecked(i);
        assert!(v < modulus, "input exceeds field modulus");
        self.state.set(i, v);
    }
}

```

## Impact

If a future refactor treats the rate range as inclusive, an off-by-one error at the rate/capacity boundary could overwrite the capacity element or alter absorption positions, producing incorrect

hash outputs without an obvious signal beyond mismatched test vectors/compatibility failures.

RECOMMENDATION

To mitigate this issue, we recommend the following improvements:

- Replace "positions 0..RATE" with "indices 0 to RATE-1 (exclusive)" in both `reset_state()` and `absorb()` comments, and
- Replace "position T-1 (last)" with "index T-1 (capacity, exclusive of rate)" to eliminate any inclusive-range ambiguity.

REMEDIATION COMMENT

Addressed in: 08fa50d9fcbf8e150ffd11f51b445151867c79fa

**SOLVED:** The **Stellar team** solved this finding in the specified commit by replacing all ambiguous `0..RATE` comment notation with Rust inclusive-range syntax `0..=RATE-1` in both `reset_state()` and `absorb()`, and by explicitly documenting that `state[T-1]` (the capacity cell) is not touched during absorption.

---

HAL-006 SOLVED

# Poseidon Documentation Does Not Distinguish BN254 and BLS12-381 Parameter Provenance

● Informational · AO:A/AC:L/AX:L/R:F/S:U/C:N/A:N/I:N/D:N/Y:N

DESCRIPTION

The crate's public Poseidon v1 API and documentation state that the hash "matches circom's Poseidon implementation" without qualifying that this claim applies only to BN254. Circomlib does not provide BLS12-381 Poseidon parameters; the BLS12-381 parameters are self-generated and intentionally match a separate external reference (`poseidon-bls12381-circom`), not circomlib. As a result, the BLS12-381 `ROUNDS_P` values (56 for `T={2,3,4}`) are not divisible by T and differ from BN254's circom-rounded values — by design, but without disclosure.

Consequently, systems may be produced in which off-chain tooling and circuits and on-chain hashing disagree, resulting in proof verification failures, mismatched commitments, or rejected transactions—without any compile-time or runtime indication beyond "hash outputs don't match".

## Code Location

`src/lib.rs` documents `poseidon_hash` as matching circom while advertising BLS12-381 as a supported field:

RUST 60-71

```rust
/// Computes a Poseidon hash matching circom's
/// [implementation](https://github.com/iden3/circomlib/blob/master/circuits/poseidon.circom).
///
/// # Type Parameters
///
/// - `T`: State size. Must be ≥ `inputs.len() + 1` (rate = T-1, capacity = 1).
/// - `F`: Field type. Use [`BnScalar`] for BN254 or [`BlsScalar`] for BLS12-381.
///
/// # Supported Configurations
///
/// - BN254: `T` ∈ {2, 3, 4, 5, 6} (i.e., 1–5 inputs)
/// - BLS12-381: `T` ∈ {2, 3, 4, 5, 6} (i.e., 1–5 inputs)

```

`src/poseidon/params.rs` states BN254 parameters were generated to “stick to” circom’s `rounds_p` rounding convention:

RUST 4-10

```rust
// Poseidon preset parameters for bn254
//
// These parameters are generated with the reference sage script (...)
// ...
// caveat: if you run the script as is, it calculates the round numbers internally
(via function `calc_final_numbers_fixed`) and uses it,
// which is different from the official values (...). Circom rounds up the `rounds_p`
to the nearest integer that divides by t
```

```
9  // (it's likely an early convention) so we gonna stick to it. Therefore, the parameter
   sets are generated from running the script with hardcoded rounds constants (e.g.
   rounds_p = 57 for t = 3).
10
```

`src/poseidon/params.rs` separately states the BLS12-381 parameters are self-generated and explicitly “do not round up rounds_p”, and references an external BLS12-381 circom implementation for consistency:

RUST 2188–2193

```
2188 // Poseidon preset parameters for bls12-382
2189 // ...
2190 // Since the official circomlib does not support bls12-382 parameters. We
     // generated them ourselves. We do not round up rounds_p -- they are calculated
2191 // internally (which is 56 for t={2,3,4}). The parameter choices also match
     // https://github.com/jmagan/poseidon-bls12381-circom/tree/main
2192
```

The resulting BLS12-381 `ROUNDS_P` values differ from BN254’s circom-rounded values for the same `T`:

RUST

```rust
// BN254 implementations
impl PoseidonConfig<3, BnScalar> for PoseidonSponge<3, BnScalar> {
    const ROUNDS_F: u32 = 8;
    const ROUNDS_P: u32 = 57;
    // ...
}
impl PoseidonConfig<5, BnScalar> for PoseidonSponge<5, BnScalar> {
    const ROUNDS_F: u32 = 8;
    const ROUNDS_P: u32 = 60;
    // ...
}
impl PoseidonConfig<6, BnScalar> for PoseidonSponge<6, BnScalar> {
    const ROUNDS_F: u32 = 8;
    const ROUNDS_P: u32 = 60;
    // ...
}

// BLS12-381 implementations
impl PoseidonConfig<3, BlsScalar> for PoseidonSponge<3, BlsScalar> {
    const ROUNDS_F: u32 = 8;
    const ROUNDS_P: u32 = 56;
    // ...
}
impl PoseidonConfig<5, BlsScalar> for PoseidonSponge<5, BlsScalar> {
    const ROUNDS_F: u32 = 8;
    const ROUNDS_P: u32 = 56;
    // ...
}
impl PoseidonConfig<6, BlsScalar> for PoseidonSponge<6, BlsScalar> {
    const ROUNDS_F: u32 = 8;
    const ROUNDS_P: u32 = 57;
    // ...
}
```

## Impact

This does not indicate a reduced security level by itself, but it creates a high risk of **integration mismatches**:

- A contract and a ZK circuit/tooling may compute different Poseidon hashes for the “same” inputs if they assume circomlib-style parameters for BLS12-381.
- Developers may incorrectly assume BLS12-381 and BN254 share the same parameter generation convention, since the documentation presents a single "matches circom" claim for both fields without distinguishing their separate reference implementations.
- Cross-field comparisons (BN254 vs BLS12-381) may be attempted under the mistaken belief that the implementation is “the same function in a different field”, yielding inconsistent commitments or failing proofs.

RECOMMENDATION

To mitigate this issue, we recommend the following improvements:

- Clarify the public documentation in `src/lib.rs` and `src/poseidon/sponge.rs` to explicitly distinguish parameter sources and compatibility boundaries:
  - BN254 Poseidon v1: circomlib-compatible parameters (circom-rounded `rounds_p`).
  - BLS12-381 Poseidon v1: self-generated parameters (no circom rounding) intended to match `poseidon-bls12381-circom`, not circomlib.
- Add a short “Compatibility” note (or a dedicated section) stating that **BLS12-381 Poseidon is not interchangeable with circomlib Poseidon**, and that integrators must use the exact matching parameter set in any external circuit/tooling.
- Fix the “bls12-382” typos in `src/poseidon/params.rs` to “bls12-381” to reduce confusion when auditors and integrators trace parameter provenance.

REMEDIATION COMMENT

Addressed in: 08fa50d9fcbf8e150ffd11f51b445151867c79fa

**SOLVED:** The **Stellar team** solved this finding in the specified commit by following the mentioned recommendation.

---

HAL-007 SOLVED

# Poseidon2 Hash Accepts Empty Inputs Without Documented Semantics

● Informational · Impact: 0 / Likelihood: 0

DESCRIPTION

`Poseidon2Sponge::compute_hash()` (and the public `poseidon2_hash` wrapper) accepts `inputs.is_empty()` without documenting the resulting semantics, while `PoseidonSponge::compute_hash()` explicitly panics on empty input.

Any caller (i.e., any contract or integrator using this crate) can pass an empty vector to `poseidon2_hash`:

- Provide `inputs = []` (e.g., due to missing validation or a mistaken assumption that the API will reject empties).
- `compute_hash()` derives `iv = (inputs.len() << 64) = 0`, resets the state with a zero IV, absorbs nothing, then squeezes.
- The function returns the Poseidon2 permutation of the all-zero state (a deterministic constant for the chosen `(T, F)` configuration).

This behavior does not introduce a collision within Poseidon2 (e.g., `poseidon2_hash([]) != poseidon2_hash([0])` due to the length-derived IV), but it is a silent behavioral divergence from `poseidon_hash` unless documented.

## Code Location

`PoseidonSponge::compute_hash()` explicitly rejects empty inputs:

RUST 234–239

```rust
234 pub fn compute_hash(&mut self, inputs: &Vec<U256>) -> U256 {
        assert!(!inputs.is_empty(), "Poseidon: inputs cannot be empty");
235     self.reset_state();
        self.absorb(inputs);
236     self.squeeze()
237 }
238
```

`Poseidon2Sponge::compute_hash()` has no corresponding empty-input guard and derives `iv` from `inputs.len()`:

RUST 193–199

```rust
193 pub fn compute_hash(&mut self, inputs: &Vec<U256>) -> U256 {
        // The initial value for the capacity element: input.len() * 2^64 for Poseidon2
194     let iv = U256::from_u128(&self.env, (inputs.len() as u128) << 64);
        self.reset_state(iv);
195
```

```
        self.absorb(inputs);
        self.squeeze()
    }
```

The public `poseidon2_hash` documentation does not define behavior for `inputs.is_empty()`:

RUST 145–148

```
/// # Panics
///
/// - if `inputs.len() > T - 1` (rate exceeded)
/// - if any input value ≥ the field modulus (inputs must be valid field elements)
```

## Impact

- Contracts or protocol code that (implicitly) relies on “hash panics on empty input” as an input-presence guard may accept and process empty vectors instead of aborting.
- Call sites that do not anticipate a deterministic fixed value for empty input may perform unexpected equality/commitment checks against that constant.
- No cryptographic collision is introduced by this behavior: Poseidon2’s length-derived IV ensures `poseidon2_hash([]) != poseidon2_hash([0])`.

RECOMMENDATION

To mitigate this issue, we recommend the following improvements:

- Explicitly document the empty-input semantics for Poseidon2 in both `Poseidon2Sponge::compute_hash()` and `poseidon2_hash` (e.g., state that empty inputs are permitted and return the permutation of the all-zero state with `iv = 0`).
- Provide a non-empty enforcing convenience API (or document a recommended caller-side guard) for consumers that require “must provide at least one input” invariants, to avoid accidental reliance on silent empty-input hashing.

REMEDIATION COMMENT

Addressed in: 08fa50d9fcbf8e150ffd11f51b445151867c79fa

**SOLVED:** The **Stellar team** solved this finding in the specified commit by adding explicit `# Empty Inputs` documentation sections to both `poseidon2_hash` in **src/lib.rs** and `Poseidon2Sponge::compute_hash` in **src/poseidon2/sponge.rs**, clearly specifying the permitted empty-input behavior and its domain separation guarantee, and contrasting it with V1 `poseidon_hash` which rejects empty input.

# Disclaimer

*Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.*