Security Audit Report

Stellar Core & Soroban Environment Audit Stellar

Delivered: Not published

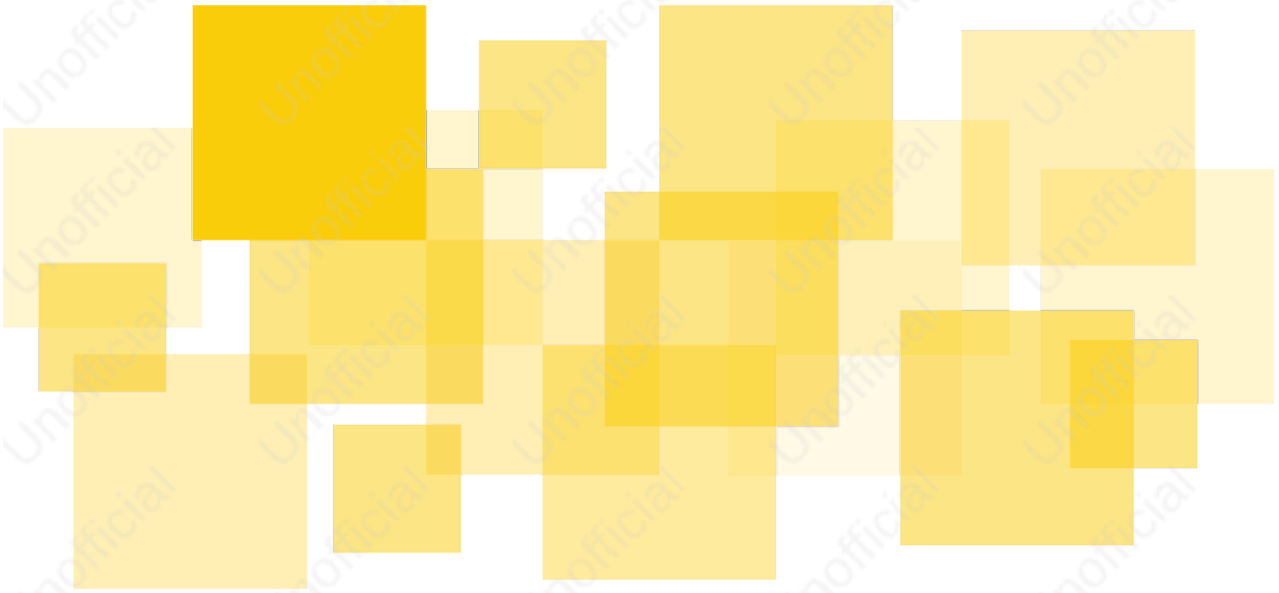


Table of Contents

Disclaimer

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

Blockchain technology is still a nascent software arena, and any related implementation and public offering carries substantial risk.

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

Executive Summary

The Stellar Development Foundation (SDF) engaged Runtime Verification Inc. to conduct a security audit of the Soroban smart contract platform in preparation for the Protocol 25 network upgrade. The audit began on January 5, 2026, with the objective of reviewing the logic and implementation of critical components and identifying any issues that could cause erroneous or undefined behavior, potentially leading to exploitation or malicious interaction with the Stellar network.

The audit was conducted over approximately 5 weeks, concluding on February 6, 2026. It focused on analyzing the changes introduced since Protocol 24, including the implementation of CAP-0074 (BN254 Host Functions) and CAP-0075 (Poseidon and Poseidon2 Primitives), as well as stability improvements across both stellar-core and rs-soroban-env.

The target code resides in two repositories:

- https://github.com/stellar/rs-soroban-env (commit d2ff024b72f7f3f75737402ac74ca5d0093a4690)
- https://github.com/stellar/stellar-core (commit: e9748b05a70d613437a52c8388dc0d8e68149394)

Prior audits covered stellar-core up to v24.0.0 and rs-soroban-env up to v23.0.2. This audit addressed all changes since those baselines.

Overall, the codebase is well structured and adheres to established engineering practices. The stellar-core repository consistently uses RAII patterns, separates test and production code, and applies defensive error handling with proper use of `releaseAssertOrThrow` for internal invariants and `catch_unwind()` at Rust FFI boundaries. The fuzzing results provide additional assurance that the CAP implementations behave correctly across a wide range of inputs, while the code review provides confidence that Protocol 25 wiring, invariants, and failure paths match the intended design.

Overview

Protocol 25 introduces new Soroban capabilities along with stability and performance improvements. The upgrade touches both stellar-core, which received changes to how Soroban transactions are validated and applied and how Soroban-related state is managed, and rs-soroban-env, which received new host functionality and associated metering work.

The protocol changes are documented in two Core Advancement Proposals: CAP-0074 introduces BN254 host functions for zero-knowledge proof support, and CAP-0075 adds Poseidon and Poseidon2 hash functions.

Stellar-Core Upgrade

We reviewed ten Soroban-related pull requests in stellar-core:

- #4968: State Archival Consistency Invariant
- #4985: Startup Check Optimization
- #5016: Fee Bump Soroban Memo
- #4942: Soroban Apply Optimization
- #4951: Parallel Apply Optimization
- #5008: Shutdown Sequence Refactoring
- #5022: Code Hardening
- #5015: In-Memory Loading
- #4897: Mempool Optimizations
- #4952: Disk Write Removal

An assessment of all stellar-core PRs in the v24.0.0 -> v25.0.0 delta was performed to ensure comprehensive coverage of Soroban-related changes. PRs affecting general ledger infrastructure (checkpoint handling, bucket management, general state eviction) were determined to be out of scope as they are not Soroban-specific. All Soroban-related changes between Protocol 24 and 25 are covered by the 10 PRs listed above.

State Archival Consistenc

Protocol 25 introduces state archival for Soroban ledger entries, managing contract data lifecycle through eviction to and restoration from the hot archive. Entries with expired TTL are evicted from live state and can be restored when needed. The `ArchivedStateConsistency` invariant ensures that entries do not exist in both live and archived states simultaneously. The invariant runs during ledger close on nodes configured to run invariants. On watcher nodes, it is configured as strict, triggering node termination on violations to serve as an alert mechanism. Background eviction scanning operates deterministically. The eviction iterator is stored in the ledger state, so all nodes scan identical entries in the same order. PR #4985 adds an iterator-based startup scan that improves performance for large state sizes by replacing the batch-based approach, processing entries incrementally rather than loading entire batches into memory.

Protocol-Gated Validation

PR #5016 fixes a bug in fee bump Soroban transactions with memos by moving memo validation from the flooding path to protocol validation for Protocol 25+, preserving Protocol 24 behavior. Protocol gating uses the ledger header version correctly, so all honest nodes apply validation at the same pipeline point for a given ledger. The validation remains a read-only, O(1) check on memo format and account types. Both normal and fee-bump transactions follow identical validation rules, with no protocol version window where validation could be bypassed. The protocol version check ensures all nodes apply the same validation rules at consensus time.

Performance Optimizations

Parallel Apply Optimization (PR #4951) reduces redundant key-collection work when merging read/write sets from worker threads during the commit phase. The optimization in `GlobalParallelApplyLedgerState::commitChangesFromThreads()` eliminates duplicate passes over the same data. The commit path waits for all workers for a given stage to complete before aggregating keys via `getReadWriteKeysForStage()`, preserving stage boundaries and transaction ordering. The optimization maintains the same semantics as the pre-optimization version while reducing computational overhead.

Soroban Apply Caching (PR #4942) caches frequently accessed objects (module cache, hot archive snapshot, configuration) across the apply phase, reducing repeated lookups and lock acquisitions. The `shared_ptr` usage for hot archive snapshots ensures proper lifetime management across threads. The module cache remains the single source of truth, with internal synchronization, and cached references remain valid for the duration of the apply phase.

Mempool Surge Pricing Skip (PR #4897) allows block construction to skip surge pricing calculations when all transactions fit within resource limits. The skip logic in `SurgePricingUtils` checks per-lane and generic limits using `countTxsResources()` before bypassing the full algorithm; when limits are exceeded, it falls back to full surge pricing. The skip path produces equivalent results to the full path when all transactions fit, using identical resource accounting functions.

Shutdown and Code Hardening

Shutdown Sequence Refactoring (PR #5008) centralizes cleanup logic in `idempotentShutdown()` and ensures proper thread coordination. `gracefulStop()` calls `idempotentShutdown()`, then `joinAllThreads()` ensures all worker threads complete before final cleanup. Shutdown always runs on the main thread after workers are joined (enforced via `releaseAssert(threadIsMain())`), preventing cleanup steps from being dropped or reordered. The idempotent design makes repeated shutdown calls safe, and try-catch blocks prevent exceptions from escaping destructors.

Code Hardening (PR #5022) adds defensive checks for parallel execution features, improving robustness during edge cases and partial failures. The changes add additional validation and error handling for parallel apply operations.

Bucket Storage and I/O Optimizations

In-Memory Bucket Creation (PR #4952) introduces `freshInMemoryOnly()`, which creates level-1 buckets without disk I/O. The call site at `BucketListBase.cpp` creates the temporary bucket and immediately merges it into level 0; the merge result is what gets adopted via `setNextInMemory()`, so the intermediate bucket never needed disk persistence. Only adopted buckets participate in state hashing and recovery. The optimization includes a fallback path for startup scenarios where the bucket is not in memory.

In-Memory Streams (PR #5015) replaces disk-backed streams with in-memory streams while preserving ordering and content guarantees. Fallback handling for startup scenarios with incomplete in-memory state ensures compatibility with existing recovery mechanisms. The changes maintain all ordering and content guarantees while eliminating disk I/O for temporary data structures.

Configuration and Database Updates

Protocol 25 updates Soroban operations' resource cost parameters based on performance measurements. These are numerical changes to cost tables rather than logic modifications, updated through the standard configuration upgrade mechanism. The database schema migration performs cleanup operations, removing outdated data structures no longer referenced by Protocol 25 code. The migration executes within a database transaction for atomicity.

Dalek Ed25519 Integration

Protocol 25 integrates Dalek Ed25519 signature verification through FFI to Rust, providing an alternative to libsodium. The new host function (`verify_ed25519_signature_dalek`) is callable from Soroban contracts. The implementation uses `verify_strict` semantics to match libsodium's behavior in rejecting small-order and mixed-order points. The FFI boundary includes proper buffer validation, error handling, and safe use of unsafe blocks. Invalid public keys and verification failures return `false` rather than panicking, ensuring the function never crashes even with malicious input.

CAP: 0074 - Host Functions for BN254

The Protocol 24 to Protocol 25 (P25) upgrade includes the Core Advancement Proposal (CAP) 0074, which introduces three new Soroban host functions for the BN254 elliptic curve: `bn254_g1_add`, `bn254_g1_mul`, and `bn254_multi_pairing_check`. These allow for addition of two G1 points, multiplication of a G1 point with a scalar, and evaluating the pairing function for a list of input point pairs, respectively.

This section provides a high-level overview of the new host functions. For the complete technical specification, refer to CAP 0074.

The BN254 curve is a Weierstrass elliptic curve that supports evaluation of the pairing function. This curve is expected to be used for privacy focused applications on the blockchain. The addition of BN254 in Stellar is motivated by Ethereum also supporting this curve in particular. Because of this, CAP-0074 specifies that the encoding of host function input and output values must use the same encoding as specified by Ethereum, see EIP-196 and EIP-197.

Host Functions

`bn254_g1_add`

```
fn bn254_g1_add(p0: BytesObject, p1: BytesObject) -> Result<BytesObject, HostError>
```

The addition host function takes two serialized G_1 points `p0` and `p1` and returns a serialized G_1 point. The input points must be on-curve, except for infinity, which is encoded as $(0, 0)$. It adds both points and returns the resulting point.

`bn254_g1_mul`

```
fn bn254_g1_mul(p0: BytesObject, scalar: U256Val) -> Result<BytesObject, HostError>
```

The scalar multiplication host function takes a serialized G_1 point `p0` and a scalar `u256` value `scalar` and returns a serialized G_1 point. The input point must be on-curve, except for infinity. It multiplies the point `p0` with the scalar value `scalar` and returns the resulting point.

`bn254_multi_pairing_check(vp1: VecObject, vp2: VecObject) -> Result<Bool, Hos`

```
fn bn254_multi_pairing_check(vp1: VecObject, vp2: VecObject) -> Result<Bool, HostError>
```

The multi-pairing check host function takes two vectors of serialized points and returns a binary validity check. The inputs are passed as individual vectors: `vp1` containing points in G_1, and `vp2` containing points in G_2, which is a subgroup on a twisted version of the curve. All points must be on-curve, except for infinity. Additionally, the points in `vp2` must belong to the G_2 subgroup. Each pair of points is used to perform a bilinear pairing operation. Afterwards the product of all these pairings is calculated and compared with 1 in $F_{p^{12}}$. The host function returns `true` if both are equal.

CAP: 0075 - Cryptographic Primitives for Poseidon/Poseidon2 Hash Functions

The Protocol 24 to Protocol 25 (P25) upgrade includes the Core Advancement Proposal (CAP) 0075, which introduces two new Soroban host functions implementing the Poseidon and Poseidon2 cryptographic permutation functions: `poseidon_permutation` and `poseidon2_permutation`. These host functions enable efficient, deterministic permutation of vectors of field elements inside the Soroban VM and are intended to support zero-knowledge–friendly hashing and proof systems.

This section provides a high-level overview of the new host functions. For the complete technical specification, refer to CAP-0075.

Poseidon and Poseidon2 are cryptographic permutations designed for use in zk-SNARK-friendly environments. Unlike traditional hash functions, they are optimized for arithmetic circuits and operate natively over prime fields. Poseidon2 is a newer variant that improves performance and security margins by modifying the linear layer and round structure while retaining the same general permutation framework.

Both permutations operate over a state vector of size `t`, apply alternating full and partial rounds, and consist of three core components: an S-box exponentiation layer parameterized by `d`, a linear mixing layer, and the addition of round constants. CAP-0075 exposes these permutations in a parameterized form, allowing smart contracts to supply custom parameters and constants while relying on the host for efficient execution.

Host Functions

`poseidon_permutation`

```
poseidon_permutation(
    input: VecObject,
    field: Symbol,
    t: U32Val,
    d: U32Val,
    rounds_f: U32Val,
    rounds_p: U32Val,
    mds: VecObject,
    round_constants: VecObject,
) -> Result<VecObject, HostError>
```

The Poseidon permutation host function takes an input vector of field elements and applies the Poseidon permutation, returning the resulting state vector.

- `input` is a vector of field elements representing the initial state. Its length must equal `t`.
- `field` specifies the prime field over which the permutation is defined. Can be `BN254` or `BLS12_381`
- `t` is the width of the permutation state.
- `d` is the S-box exponent applied during the nonlinear layer.
- `rounds_f` and `rounds_p` specify the number of full and partial rounds, respectively.
- `mds` is the MDS matrix used for the linear mixing layer.
- `round_constants` is a `(rounds_f+rounds_p) x t` matrix of constants added during each round.

The host function performs input validation, applies the Poseidon round function as specified by the supplied parameters, and returns the permuted state. All arithmetic is performed within the specified field. Errors are returned if the input sizes or parameters are inconsistent.

`poseidon2_permutation`

```
poseidon2_permutation(
    input: VecObject,
    field: Symbol,
    t: U32Val,
    d: U32Val,
    rounds_f: U32Val,
    rounds_p: U32Val,
    mat_internal_diag_m_1: VecObject,
    round_constants: VecObject,
) -> Result<VecObject, HostError>
```

The Poseidon2 permutation host function applies the Poseidon2 permutation to an input vector of field elements and returns the resulting state vector.

The parameters are similar to those of `poseidon_permutation`, with the following differences:

- Instead of a full MDS matrix, Poseidon2 uses a modified linear layer defined by `mat_internal_diag_m_1`, which represents the diagonal component of the internal matrix `(M − I)`.
- The linear layer construction is optimized to reduce arithmetic cost while preserving diffusion properties.

As with the Poseidon permutation, the input vector must have length `t`, all parameters must be consistent, and all operations are carried out in the specified prime field. The host function returns an error if validation fails or if the permutation cannot be evaluated safely.

Together, these host functions allow Soroban smart contracts to efficiently execute Poseidon-based cryptographic permutations, enabling applications such as zero-knowledge proofs, Merkle constructions, and privacy-preserving protocols to be implemented directly on Stellar with strong performance and correctness guarantees.

Methodology

This audit employed both code review and dynamic analysis through fuzzing and empirical analysis. The Protocol 25 engagement covered multiple components of the Soroban platform, combining a structured security review of the Soroban-related stellar-core changes with targeted fuzzing and code review of the rs-soroban-env cryptographic components.

The work was split into two main tracks. The stellar-core review focused on the C++ integration layer, error propagation across the Rust FFI boundary, transaction processing correctness under failure, concurrency and thread safety, resource bounds, and assert safety. The rs-soroban-env review and fuzzing focused on the CAP implementations (CAP-0074 BN254, CAP-0075 Poseidon/Poseidon2), metering, parameter validation, and robustness under adversarial inputs.

Detailed methodology for each pull request and analysis approach will be described separately in this chapter, and code review is further discussed in Code Review Discussion.

Stellar-Core Analysis

For stellar-core, the review focused on how changes from Protocol 24 to 25 affect security-relevant properties, such as crash behavior, FFI safety, concurrency, and resource bounds. The review examined all Soroban-related changes between v24.1.0 and v25.0.0, with particular attention to components that participate directly in transaction execution, invariant checking, and shutdown.

The review verified a consistent set of security properties across all changes:

- Error propagation and crash resistance
- FFI boundary safety
- Thread safety and concurrency
- Resource bounds
- Assert and invariant safety

Each of the ten in-scope pull requests was reviewed in detail, examining changed code in context, tracing error propagation paths, verifying concurrency and synchronization patterns, and checking resource-related logic for consistency with protocol limits. The review also considered interactions between PRs that modify shared components (module cache, state archival, shutdown sequencing, transaction processing) to ensure compatibility and avoid conflicting assumptions. Resource metering timing was verified through code flow analysis from transaction validation through execution to result handling.

Concurrency analysis examined parallel execution code for thread safety, synchronization patterns, and proper isolation between worker threads. Resource consumption patterns were analyzed to verify operations remain bounded and properly metered. Assertions and invariants were reviewed to confirm they check internal state rather than user-controlled inputs.

Precondition and postcondition analysis were applied to critical functions to verify function contracts and error handling guarantees. Control flow and data flow analysis tracked user-controlled input through validation and processing, verifying bounds checking and resource limit enforcement. For PRs that modified shared components (module cache, state archival, shutdown sequencing, transaction processing), interactions were analyzed to ensure compatibility.

CAP 0074 and CAP 0075 Review

CAP 0074 and CAP 0075, which were introduced in Overview, introduce new Soroban host functions. The review of these focuses on showing correct utilization of libraries, correctness, absence of crashes, absence of missing metering, and absence of metering anomalies. For this purpose, the following methods were utilized: code review, fuzzing for crashes, differential fuzzing, and empirical metering measuring.

The BN254 and Poseidon host functions were manually code reviewed regarding a variety of properties that are documented thoroughly in Soroban Host Functions.

As host functions must never crash, fuzzing was utilized to find crashes in either BN254 or Poseidon host functions. More details about the fuzzing harness and the fuzzing targets are documented in Fuzzing.

The Poseidon and Poseidon2 implementations in Soroban are re-implementations of another Rust crate. Therefore, differential fuzzing was used for Poseidon to find unintentional differences in both implementations. This differential fuzzing campaign is described in more detail in Fuzzing Targets for Poseidon Host Functions. The BN254 host functions use the `arkworks` library, which is assumed to be a correct and robust implementation due to wide-spread use in the industry.

Finally, empirical metering measuring was utilized to find CPU instruction metering anomalies. Here, a metering anomaly refers to a situation where a host function charges significantly less for compute resources than actually consumed, which poses a Denial-of-Service risk. This method is described in more detail in Empirical Metering Measurements.

Code Review Discussion

This section outlines the results of our manual code review, where our team conducted a thorough analysis of the target codebase. The focus was on identifying potential vulnerabilities, logical errors, and areas of improvement, while ensuring that the implementation adheres to best practices and the intended design. Findings are categorized by severity and include recommendations for remediation.

Stellar-Core Protocol 25 Review

The review examined all ten in-scope PRs, focusing on how Protocol 24 to 25 changes affect security-relevant properties: error propagation and crash resistance, FFI boundary safety, thread safety and concurrency, resource bounds, and assertion safety.

State Archival Consistency and Monitoring

PRs #4968 and #4985 introduce the `ArchivedStateConsistency` invariant that validates consistency between live and archived Soroban state, along with an optimized iterator-based startup scan. The invariant enforces a partition property: entries cannot exist in both live and archived state simultaneously.

The iterator-based optimization preserves logical equivalence with the batch-based approach while improving performance for large state sizes. The invariant runs under snapshot isolation, so the check always sees a self-consistent ledger view. Exception paths were traced and confirmed to trigger controlled shutdown rather than undefined behavior.

These PRs add consistency checks and an optimized startup scan. The strict-invariant crash on watcher nodes is intentional: the crash serves as the alert mechanism. Validators do not run this invariant, so production consensus is unaffected.

Protocol-Gated Validation Change

PR #5016 moves Soroban memo validation for fee-bump transactions from the flooding path to protocol validation for Protocol 25+, while preserving Protocol 24 behavior. We verified that memo validation applies to exactly the same transactions before and after the protocol transition, confirmed that protocol gating uses the ledger header version correctly, checked fee-bump delegation logic, and traced exception paths to ensure invalid memos fail transactions cleanly without crash paths. We also verified that no protocol version window exists where validation could be bypassed.

Soroban Apply, Caching, and the FFI Boundary

PR #4942 optimizes Soroban apply by caching frequently accessed objects (module cache, hot archive snapshot, configuration) across the apply phase, reducing repeated lookups and lock acquisitions. Data access patterns were compared and confirmed to be logically equivalent to the original path—cached references replace repeated lookups and live for the apply phase duration.

The FFI boundary remains intact. The call site invokes `rust_bridge::invoke_host_function()`, which crosses into Rust where `panic::catch_unwind()` contains any panics, converting them to `CoreHostError::General` errors rather than node crashes. The panic handler extracts panic messages via `downcast_ref` and wraps them in error types that C++ can handle gracefully. The `shared_ptr` usage for hot archive snapshots ensures correct lifetime management across threads. The module cache remains the single source of truth with internal synchronization. Error propagation was traced from `invoke_host_function()` through `InvokeHostFunctionOpFrame::invokeHostFunction()` to `doApply()`, confirming host errors correctly fail transactions via `mOpFrame.innerResult(mRes).code(INVOKE_HOST_FUNCTION_TRAPPED)`.

Parallel Apply and Concurrency Safety

PR #4951 optimizes the parallel apply commit path by reducing redundant key-collection work when merging read/write sets from worker threads. The core parallel execution flow proceeds through `applySorobanStageClustersInParallel()`, which dispatches work to thread pools via `applyThread()`, then `commitChangesFromThreads()` aggregates results after all workers complete. The optimization eliminates duplicate passes over read/write sets by computing `getReadWriteKeysForStage()` once per stage rather than per thread.

The key collection logic before and after the optimization was compared and verified to preserve stage boundaries and transaction ordering. The commit path waits for all worker threads in a given stage to complete before aggregating keys, preventing interleaved commits across stages. Edge cases, including empty key sets, duplicate keys, and high-contention batches, were analyzed; all were handled identically to the pre-optimization version. Failures remain isolated to the affected transactions, with rollbacks handled through

`LedgerTxn`. The optimization removes redundant passes over the same data without changing which keys are considered or how conflicts are resolved.

Mempool and Block Production Optimizations

PR #4897 optimizes the common case where the mempool is not resource-constrained by allowing block construction to skip surge pricing when all transactions fit under the configured limits. The skip logic in `SurgePricingUtils.cpp` checks per-lane and generic limits using `countTxsResources()` before deciding whether to bypass the full surge-pricing algorithm. The skip path and full path were compared and verified to be equivalent when all transactions fit, confirmed that the same resource accounting functions and limit checks are used in both paths.

Queue initialization and synchronization safety were verified in `TxQueueLimiter.cpp`, where `addTransaction()` and `removeTransaction()` at lines 68-86 maintain both eviction and flood queues atomically. Edge cases, including empty queues, resource usage exactly at limits, and rapid mempool churn, were analyzed. The only difference between paths is ordering within the transaction set, and final ordering is determined later by transaction-set construction and consensus rather than by this selection step. The code falls back to the full surge-pricing algorithm whenever any limit fails.

Shutdown Sequencing and Defensive Hardening

These PRs #5008 and #5022 improve shutdown robustness by centralizing cleanup logic in `idempotentShutdown()` and hardening error handling at lifecycle edges, ensuring safe resource cleanup regardless of shutdown trigger. We verified the shutdown sequence architecture by tracing execution paths from `gracefulStop()` through `joinAllThreads()` to `idempotentShutdown()`, confirmed that shutdown always runs on the main thread after all worker threads are joined, checked that cleanup steps are not dropped or reordered in ways that leave partially destroyed subsystems, and analyzed edge cases including repeated shutdown calls, shutdown during exception unwinding, and partial initialization failures.

The shutdown sequence provides thread safety through guarantees about execution context rather than explicit synchronization primitives. The `gracefulStop()` function enforces main thread execution via `releaseAssert(threadIsMain())`, and `joinAllThreads()` completes before `idempotentShutdown()` executes, meaning all worker threads have stopped before shutdown cleanup begins. The idempotent design ensures repeated calls are safe, and the try-catch around shutdown prevents exceptions from escaping destructors.

Bucket Storage and I/O Optimizations

These PRs #5015 and #4952 reduce unnecessary disk work. PR #4952 introduces `freshInMemoryOnly()`, which creates level-1 buckets without disk I/O. The call site at `BucketListBase.cpp` creates the temporary bucket and immediately merges it into level 0; the merge result is what gets adopted, so the intermediate bucket never needed disk persistence. PR #5015 replaces disk-backed streams with in-memory streams while preserving ordering and content guarantees.

Startup scenarios with incomplete in-memory state (fallback at `BucketListBase.cpp`), merge failures, and hot archive interactions were analyzed. Recovery and hashing logic are unchanged: only adopted buckets participate in state hashing and recovery, bucket ordering is preserved, and hot-archive persistence uses existing disk-based paths.

Node Availability and Crash Analysis

Node termination can occur through a fail-fast design. Crashes affecting production validators have different implications than those limited to monitoring.

Exception Handling and Error Propagation

Transaction execution provides comprehensive exception handling through try-catch blocks in `TransactionFrame::applyOperations()`. The exception handler catches all `std::exception` types, logs diagnostic information via `CLOG_ERROR`, marks transactions as `txINTERNAL_ERROR`, and rolls back state changes through the LedgerTxn mechanism. The handler only crashes nodes if `HALT_ON_INTERNAL_TRANSACTION_ERROR` is enabled (default false), allowing the node to continue processing subsequent transactions. User-submitted transactions cannot crash nodes through malformed data or unexpected edge cases—all exceptions are handled and converted to transaction failures.

Soroban host function invocation includes distinct handling for internal errors. When `invoke_host_function()` returns with `is_internal_error` set, stellar-core throws `std::runtime_error`. Internal errors indicate system-level problems—host implementation bugs, resource exhaustion, corrupted internal state—distinct from contract execution failures. The exception is caught by `TransactionFrame::applyOperations()`, which marks the transaction as failed and continues processing. The `HALT_ON_INTERNAL_TRANSACTION_ERROR` configuration flag (default false) allows operators to crash nodes on internal errors when immediate investigation is preferred, but this is an opt-in behavior for debugging rather than the default production behavior.

Exception paths were traced from `xdr::xdr_from_opaque()` to the transaction application and confirmed that all XDR decoding for Soroban paths is wrapped in the central try-catch block. XDR decoding failures result in transaction failure with state rollback, not node crashes.

Shutdown Sequence

The shutdown sequence architecture provides thread safety through guarantees about execution context rather than explicit synchronization primitives. The `idempotentShutdown()` function lacks mutex protection. The destructor can only be called on the main thread because all threads are joined before shutdown.

The implementation enforces this through multiple mechanisms. The `gracefulStop()` function enforces main thread execution via `releaseAssert(threadIsMain())`. The `joinAllThreads()` call completes before `idempotentShutdown()` executes, meaning all worker threads have stopped before shutdown cleanup begins. The destructor runs only after `gracefulStop()` completes, executing in a single-threaded context where concurrent calls are impossible by design. Exception handling in the destructor can catch any cleanup failures.

Cross-Cutting Concerns

This section covers security properties that span multiple PRs and components.

Resource Metering and Bounds

Resource metering occurs before corresponding operations execute, ensuring no resource consumption without proper charging. All metering follows the correct pattern: disk read operations meter before reading, storage write operations meter before committing, and CPU-intensive operations charge before computation.

The `checkSorobanResources()` function validates resource declarations before transaction execution begins, rejecting transactions with insufficient declared resources. Resource limits are enforced at multiple levels: network configuration parameters define maximum consumption per transaction, transaction declarations must fall within these limits, host function execution metering ensures actual usage doesn't exceed declared amounts, and refundable resources are tracked separately with correct refund calculations.

FFI Boundary Safety

The FFI boundary between C++ and Rust is protected at multiple levels. The primary protection is `panic::catch_unwind()` at the Rust entry point, which wraps the host function invocation in `panic::AssertUnwindSafe` and converts any Rust panics into `CoreHostError::General` errors that C++ can handle gracefully. The panic handler extracts panic messages via `downcast_ref::<String>()` or `downcast_ref::<&'static str>()` and wraps them in error types.

The Dalek Ed25519 signature verification integration demonstrates a complementary pattern: the function is written to never panic in the first place. Invalid public keys return `false` via a `match` statement that handles all error cases, and signature verification failures return `false` via `.is_ok()` checks. The `verify_strict` method matches libsodium's behavior in rejecting small-order and mixed-order points, ensuring cryptographic correctness. Buffer validation ensures all inputs are within bounds before processing.

State Management and Consistency

The storage lifecycle system manages Soroban contract data entries, handling eviction and restoration. TTL helper functions in `ParallelApplyUtils` call `.value()` on `std::optional<LedgerEntry>` parameters without local `has_value()` checks. We examined all call sites and confirmed that every caller validates options before passing them. `ExtendFootprintTTLOpFrame` and `RestoreFootprintOpFrame` both check `if (ltxe)` before calling `ttl(ltxe)`. The design relies on call-site validation rather than defensive checks within the helper functions, which is safe given the controlled call sites but represents a maintenance risk if new call sites are added without proper validation.

Thread Safety

The `idempotentShutdown()` function initially appeared to lack mutex protection. Analysis confirmed that shutdown always runs on the main thread after all worker threads have been joined (enforced via `releaseAssert(threadIsMain())`), so the absence of an explicit mutex is consistent with the single-threaded shutdown model. Thread safety comes from execution context guarantees rather than explicit synchronization: gracefulStop() `enforces main thread execution and` joinAllThreads()` ensures all workers complete before shutdown proceeds.

Soroban Host Functions

We manually reviewed the code of the newly introduced host functions. The host functions are written in Rust, which prevents memory corruption bugs from occurring when used `unsafe` Rust code is correct.

This review assumes the `arkworks` library that is used by the BN254 host functions is correct and robust. Due to the library's size and complexity, a full line-by-line audit of its internal arithmetic was outside the scope of this engagement. However, our review strictly examined the integration layer, verifying that the host correctly translates guest data into arkworks types and that the host's assumptions about the library's behavior, e.g., error handling and API contracts, are valid.

We considered the following criteria during the code review.

Input Validation

The Soroban guest can provide inputs when calling Soroban host functions. These inputs must be of the `Val` type, which is implemented using a 64-bit unsigned integer, which contains both a tag and a body. The tag is used to determine the actual underlying subtype, while the body will contain either the data or a handle to an immutable object stored on the host. When a guest invokes a host function, the host automatically converts the provided `Val` parameters to the ones expected by the host function and validates that the passed `Val` parameters both match the type and fulfill validation criteria defined for the respective type.

In addition, the host function must perform additional validation to ensure that the provided data fulfills the pre-conditions for further processing, if any exist.

Absence of Crashes

The host functions must never crash given any possible input data, since a crash would cause a Denial-of-Service on Stellar nodes that execute the smart contract invocation that is part of a transaction.

Absence of Non-Recoverable Errors Returned to the Soroban Host

Within the `rs-soroban-env` implementation, the `is_recoverable` function in `soroban-env-host/src/host/error.rs:149` defines internal host errors as non-recoverable, noting that they 'should only appear if there is some bug in the host implementation or setup'. Among other conditions, an error is classified as non-recoverable, if it is an `InternalError` but not of the `Contract` type, see `soroban-env-host/src/host/error.rs:152` . Consequently, we focus on verifying that the codebase never returns an `InternalError` that is not of the `Contract` type, as the occurrence would indicate an implementation bug.

Functional Correctness

The soroban host functions should operate functionally correct given any valid input data. Invalid data should trigger the return of the appropriate error.

Correct Metering of Compute Resources

Soroban host functions charge for use of compute resources. Both CPU usage and memory usage are thoroughly tracked, though memory is tracked for the purpose of enforcing an upper bound for memory usage. It's important to charge for compute resources prior to usage, since the transaction may not provide enough gas to cover costs, which should result in the return of a non-recoverable error.

Absence of Memory Leaks

Rust does not guarantee the absence of memory leaks, depending on which components of the standard library are used. Since a memory leak could be used to cause a Denial-of-Service, the absence of memory leaks is critical.

For BN254: Adherence to Ethereums EIP-196 and EIP-197 Encoding

The BN254 host functions specification declares matching encoding with the respectice pre-compiles that Ethereum provides. Ethereums BN254 pre-compiles are specified in EIP-196 and EIP-197. As Ethereums encoding of input and output values does not match the encoding used internally in the arkworks library that is used by Soroban to implement the BN254 host functions, special care must be taken.

Findings

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

[A01] Missing Budget Charge for `MemCpy`

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

Sorobans execution environment defines several contract cost types. One of these is the cost type `MemCpy` that is used when 'copying a chuck of bytes into a pre-allocated host memory', see stellar-xdr.

This is also actively used in the code base such as in function `bn254_field_element_deserialize` :

```
// function bn254_field_element_deserialize
// [...]
self.charge_budget(ContractCostType::MemCpy, Some(EXPECTED_SIZE as u64))?;
let mut buf = [0u8; EXPECTED_SIZE];
buf.copy_from_slice(input);
// [...]
```

This contract cost type is not used in the functions `bn254_g1_affine_deserialize` and `bn254_g2_affine_deserialize` that make use of `copy_from_slice` . Both functions copy 32 bytes or 64 bytes two times, respectively. `bn254_g1_affine_deserialize` is called twice in the host function `bn254_g1_add` and is called once in the host function `bn254_g1_mul` . In host function `bn254_multi_pairing_check` , both `bn254_g1_affine_deserialize` and `bn254_g2_affine_deserialize` are called for each passed G1 or G2 point, respectively.

Recommendation

Since this deviates from the standard practices of cost metering in Soroban, we recommend to add the respective `MemCpy` budget charges in both `bn254_g1_affine_deserialize` and `bn254_g2_affine_deserialize` .

Since the affected host functions nonetheless charge costs, and in the case of `bn254_multi_pairing_check` , charge costs depending on the number of points passed for compute intensive code, this is not considered a critical DoS risk.

Status

The client has acknowledged and fixed this issue for a future network upgrade subsequent to Protocol 25 in pull request https://github.com/stellar/rs-soroban-env/pull/1648. Due to the necessity of a protocol change and required validator consensus, the client determined that the current severity level does not warrant an immediate, out-of-cycle upgrade. Remediation will instead be included in the standard release schedule.

[F01] Overflowing Arithmetic on Poseidon Parameters Instantiation

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

The host functions `poseidon_permutation` and `poseidon2_permutation` take `U32` parameters for `rounds_f` and `rounds_p` for the full and partial round counts, respectively. These are summed together to get the total rounds count when validating and instantiating the internal data structures for performing the permutations.

These two parameters come from the guest and can be arbitrarily large integers. Large enough to overflow when they get added together.

A snippet of one of the involved functions:

```
impl<S: MeteredScalar> PoseidonParams<S> {
    pub fn new(
...
        rounds_f: u32,
        rounds_p: u32,
...
    ) -> Result<Self, HostError> {
...
        let rounds = rounds_f + rounds_p;
...
}
```

Recommendation

It's almost certain that an overflow during this operation would be unintended by a trustworthy user. Given this, we recommend catching any overflows here and returning an error to reduce the attack surface as much as possible, using `checked_add` or any other mechanism to do so.

Status

The client acknowledged this finding and states that it should not be a major issue. The host is compiled without overflow checks so a panic would not occur, and the validation logic after the addition should catch it very quickly. Nevertheless, they state that they plan to address it.

[M01] Host Function `bn254_g1_mul` Overcharges CPU Instructions when Scalar Input is Zero or Has Few 1-bits

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

The new Soroban host function `bn254_g1_mul` performs several operations that all charge for compute resources individually, see:

```
// `soroban-env-host/src/host.rs:bn254_g1_mul`
let p0 = self.bn254_g1_affine_deserialize(p0)?;
let scalar = self.bn254_fr_from_u256val(scalar)?;
let res = self.bn254_g1_mul_internal(p0, scalar)?;
self.bn254_g1_projective_serialize_uncompressed(res)
```

Among these, the function `bn254_g1_mul_internal` performs and charges for the multiplication of a G_1 point and a scalar. Specifically, it charges a flat amount for the multiplication, see:

```
// `soroban-env-host/src/crypto/bn254.rs:bn254_g1_mul_internal`
self.charge_budget(ContractCostType::Bn254G1Mul, None)?;
Ok(p0.mul(scalar))
```

Per convention, described in CAP 0046-10, the metering costs should cover the worst case scenario in regard to executed instruction count, which correlates with execution time.

However, the best case scenario is considerably quicker than the worst case, leading to significant overcharging whenever the number of 1-bits in the scalar is small and especially when the number is zero. In Measurements for BN254 Host Functions, we observed `metered_instructions / execution_time` ratios that are > 61 times larger compared to our worst observed cases for other considered host functions, specifically Poseidon2. When measuring executed instruction count instead of execution time by replaying the same outlier inputs, we can observe `metered_instruction / cpu_instruction` ratios of up to 127, while they should actually be rather close to 1.

We expect these edge cases, specifically scalar values of 0 or 1, to possibly appear in practice. An LLM gave the following opinion regarding potential uses case where these scalar values might be used in junction with the host function:

- Multi-Scalar Multiplication (MSM): ZKP verifiers, like those for Groth16 or Plonk, often need to compute a linear combination of points, which might include 0 (to specify an input as inactive) or 1 scalars (common in polynomial commitments).

Though we did not investigate further whether and to which extend these examples hold. Nonetheless, we recommend to assume that these edge-case scalar values might be used in practice and should therefore not overcharge users for compute resources.

Recommendation

We recommend to adjust the metering to consider the number of 1-bits and possibly the position of the first 1-bit when computing the amount of compute resources charged for the `bn254_g1_mul_internal` function.

Status

The SDF team acknowledged this finding and considers it an informational finding, as the primary goal of the budget system is DoS prevention, with this finding not presenting one. Nevertheless, the SDF team is exploring future enhancements to the metering system.

Informative Findings

The Informative Findings section highlights observations and recommendations that, while not classified as vulnerabilities, are worth addressing to improve code quality, maintainability, or performance. These findings may include suggestions for adhering to best practices, improvements to documentation, or enhancements to the overall design.

[B01] Unnecessary Empty Flag Enforcement in BN254 Deserialization

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

Sorobans BN254 host functions use the same encoding as the BN254 precompiles in Ethereum, see CAP-0075. This encoding, among other things, does not include flag bits that arkworks uses. Specifically, arkworks uses two flag bits to encode whether the y component of a point is negative or not, and whether the point is infinity. In Soroban, i.e., in Ethereum, infinity is encoded instead as $(0, 0)$.

In addition, Soroban makes use of arkworks serialization and deserialization functionality. This means that Soroban must ensure that the arkworks library does not parse any flag bits it might expect to enforce semantic equivalence with the precompiles in Ethereum.

Due to this, the functions `bn254_g1_affine_deserialize` and `bn254_g2_affine_deserialize` enforce empty bits in the serialized elliptic curve point by calling the function `bn254_validate_flags_and_check_infinity`, which among other things, checks that both flag bits are zero.

Since the endianness that arkworks uses differs from Soroban/Ethereum as well, Soroban deserializes the components of a point individually in order to change the endianness of a component by reversing the list of bytes.

Therefore, Soroban makes use of the arkworks deserialization code for individual components, which actually does not check flag bits since flag bits are only encoded in points, which consist of two components, see `ark-ff-0.4.2/src/fields/models/fp/mod.rs:628`:

```
impl<P: FpConfig<N>, const N: usize> CanonicalDeserialize for Fp<P, N> {
    fn deserialize_with_mode<R: ark_std::io::Read>(
        reader: R,
        _compress: Compress,
        _validate: Validate,
    ) -> Result<Self, SerializationError> {
        Self::deserialize_with_flags::<R, EmptyFlags>(reader).map(|(r, _)| r)
    }
}
```

and `ark-serialize-0.4.2/src/flags.rs:43`:

```
#[derive(Default, Clone, Copy, PartialEq, Eq)]
pub struct EmptyFlags;

impl Flags for EmptyFlags {
    const BIT_SIZE: usize = 0;

    #[inline]
    fn u8_bitmask(&self) -> u8 {
        0
    }

    #[inline]
    fn from_u8(_: u8) -> Option<Self> {
        Some(EmptyFlags)
    }
}
```

Recommendation

While this does not cause critical issues or semantic differences with Ethereums precompiles, it is however an unnecessary check. Especially, the code wrongly assumes that flags are parsed during deserialization of individual components, while flags are only parsed while deserializing points.

The serialization code in Soroban however correctly acknowledges this behaviour and does not check or change any flag bits, see function `bn254_g1_affine_serialize_uncompressed` :

```
pub(crate) fn bn254_g1_affine_serialize_uncompressed(
    &self,
    g1: &G1Affine,
) -> Result<BytesObject, HostError> {
    // we do not need to special handle flags on the serialization path,
    // since we are serializing each element directly which does not carry
    // flags, i.e. `Fp<P, N>::serialize_with_mode` just uses `EmptyFlags`.
```

We recommend to adjust the code to not check the flags when deserializing individual components using the arkworks library and to adjust the comments to properly reflect the behaviour of the arkworks library.

Status

The client declines to address this informational finding, arguing that the code ensures specification compliance through explicit flag bit validation. They emphasize that this validation operates independently of the underlying coordinate serialization and parsing logic in the arkworks library.

[B02] Results of cargo-audit for Audited Repositories

Severity: Informative | Not addressed by client

The Rustsec tool `cargo-audit` was run on the two repositories to audit dependency vulnerabilities, and produced the following results:

- `rs-soroban-env` :
  - One security vulnerability RUSTSEC-2025-0055 (CVE-2025-58160, CVSS v4.0: 2.3) related to ANSI escape sequence log poisoning in crate `tracing-subscriber` version 0.3.18, a transitive dependency through `tracy-client` -> `loom` -> `tracing-subscriber` and also through `tracking-allocator` -> `tracing-subscriber` . The vulnerability affects logging infrastructure when user input containing ANSI escape sequences is logged, potentially corrupting terminal output and hiding malicious activity in logs.
  - One warning for yanked crate `rgb` version 0.8.37 used via `textplots` -> `soroban-env-host` -> `soroban-simulation` .
- `stellar-core` : the same vulnerability in crate `tracing-subscriber` version 0.3.17.

Recommendation

- Upgrade `tracing-subscriber` to version >=0.3.20 by adding it as a direct dependency.
- The `textplots` warning relates to `soroban-simulation` , which is out of scope for this audit; if `textplots` is used in production builds, consider moving it to dev-dependencies.

Status

Acknowledged by the client.

[B03] Memory Exhaustion Concerns in State Consistency Check

Severity: Informative | Not addressed by client

The `seenKeys` set in `ArchivedStateConsistency::checkSnapshot()` tracks all ledger keys during validation to detect duplicates. We estimated that for networks with large state sizes (30+ million entries), this set could consume approximately 1.9 GB of memory during the check.

After discussion with the Stellar team, we confirmed that memory usage is bounded by the largest single entry type count rather than total state, since the set is cleared after processing each entry type. This is an operational consideration rather than a security risk, as the memory usage is bounded and temporary.

Recommendation

Consider adding memory usage monitoring or progress logging for nodes with very large states to improve operational visibility.

Status

Documented. This is an operational issue, not a security vulnerability.

Fuzzing

This section outlines our fuzzing strategy, which focused on implementing targets to uncover correctness issues and crashes in the new Soroban host functions.

Fuzzing Harness for Soroban Host Functions

As part of our audit, we utilized fuzzing with the goal of finding issues in the new host functions introduced by CAP: 0074 - Host Functions for BN254 and CAP: 0075 - Cryptographic Primitives for Poseidon/Poseidon2 Hash Functions, i.e., running newly introduced host function code with random values provided by the fuzzer to find crashes and invariant violations. The host functions are written in Rust and are located in the GitHub repository rs-soroban-env. CAP 0074 and CAP 0075 introduce new host functions for the BN254 elliptic curve and poseidon/poseidon2, respectively.

For the fuzzing harnesses we started a new repository with the intention of it providing everything necessary to run them. It contains the Rust crates with our fuzzing harnesses and utility libraries as well as a setup script `scripts/setup.sh` that clones `rs-soroban-env` and the reference poseidon implementation that our fuzzing harness depends on. The setup script additionally patches the visibility of selected internal functions of the `rs-soroban-env` crate `soroban-env-host` to `public` for use inside our fuzzing harness. An accompanying runner script `scripts/run.sh` serves as a simple interface for running the fuzz targets and handling common flags for them. Both a Dockerfile with utility scripts for running inside a container as well as a nix develop shell are available for reproducibility and ease-of-use of our fuzzing harness. The Dockerfile can be built with the script `scripts/docker/build.sh` and the nix develop shell can be entered by running `nix develop`, which will provide a reproducible rust distribution, the honggfuzz utilities, and it's dependencies.

We used the fuzzer Honggfuzz with `honggfuzz-rs` that enables the fuzzing of Rust code using Honggfuzz. Specifically, we performed fuzz-to-crash testing for both the BN254 and poseidon implementations and differential testing for poseidon against the reference implementation. Additionally, we added invariants for the results in our fuzz targets, such as, e.g., asserting that the result of host function calls is never non-recoverable (without considering exceeding the budget). We used the Soroban host environment as the primary entry point to test from and additionally for the BN254 implementation, we also tested against internal functions of `rs-soroban-env` used by the host functions as well as tested against the `ark-bn254` crate that is used in the implementation of the BN254 host functions.

When interfacing with host functions, soroban host values must be passed, which are either values that can be packed into 64 bits, including their tag bits, or are values that contain a handle to host objects that are stored on the host. Therefore, prior to invoking a host function under test, host values and host objects must be set. In general, `rs-soroban-env` allows for easy creation of a host that is capable of invoking multiple host functions and setting up host objects. Specifically, the function `soroban_env_host::Host::test_host()` is provided that both creates a host and sets up mandatory ledger configuration parameters.

With our fuzzing harness setup, our strategy was to start our fuzzing targets right after finalization of each in our distributed compute platform K-as-a-service (KaaS). We created a new fuzzing harness execution platform within KaaS to start, run, and collect fuzz runs using `honggfuzz-rs`. This allowed for easy dispatch of fuzz target runs. Specifically, it collects the details of multiple fuzz target runs that utilize one continued fuzzing input corpus per fuzz target. A dashboard shows the collective runs for each fuzz target. The fuzzing statistics can found in the Fuzz Run Statistics.

Fuzzing Targets for BN254 Host Functions

The BN254 host functions for addition, scalar multiplication, and multi-pairing checks require as inputs points in BN254 G_1, BN254 G_2, and scalar values in a scalar field $\mathbb{F}_r$.

The scalar field $\mathbb{F}_r$ values can be trivially generated from random data provided by the fuzzer.

The G_1 group consists of points that are pairs of elements (x, y) from the base field $\mathbb{F}_p$ that are on the BN254 curve in addition to the point at infinity. While these pairs can be trivially generated from random data, it is almost guaranteed that random values will result in pairs that are not on the elliptic curve. However, random x components can be used to solve for a valid y component such that the resulting (x, y) pair is a valid point on the elliptic curve, though not every possible x component is part of a valid point, i.e., there will be no valid y counterpart. The test in `rv-soroban-lib/tests/g1_point_probability.rs` empirically measures this probability to be roughly 50.05% using 100000 iterations. Furthermore, it is guaranteed that every such point on the elliptic curve is also part of the relevant subgroup G_1. Another way to generate valid G_1 points is to multiply the G_1 generator point with a random scalar value. Both methods are employed in the fuzz targets for BN254.

The G_2 group is a subgroup of points that are pairs (x, y) of elements from the quadratic extension field $\mathbb{F}_{p^2}$ that are located on a twisted variant of the BN254 curve, together with the point at infinity. Each $\mathbb{F}_{p^2}$ value can be considered a pair of two $\mathbb{F}_p$ values, with $\mathbb{F}_{p^2}$ values typically being analogous to complex numbers. These pairs can also be trivially generated from random data, though it is again almost guaranteed that random values will result in pairs that are not on the elliptic curve. While pairs that are on the curve can be generated by choosing a random x component and solving for y, if it exists, these pairs are almost guaranteed to not be part of the G_2 subgroup, and therefore are not valid input values for the host functions. The test in `rv-soroban-lib/tests/g2_point_probability.rs` empirically measures the probability that random x components have a respective y component such that their point is on the curve and whether that point is part of the G_2 subgroup. The probability that a random x component has a respective y component such that the point is on the curve is, according to the measurements using 100000 iterations, roughly 49.97% and the probability that such a point is part of the G_2 subgroup is measured to be roughly 0%, i.e., none of the points were part of the G_2 subgroup. Therefore the only employed method in the fuzz targets to generate valid G_2 points from random data was by multiplying the G_2 generator point with a random scalar derived from the random data provided by the fuzzer.

Fuzzing Targets

The following fuzzing targets fuzz the three BN254 host functions, `bn254_g1_add`, `bn254_g1_mul`, and `bn254_multi_pairing_check`, as well as their internally used functions of interest.

`bn254_g1_add_valid_size`

- **Strategy:** Fuzzes `bn254_g1_add` using arbitrary random data of valid byte length. This hits off-curve points and both component $\geq p$ and $< p$ cases.
- **Assertion:** No Internal Error (recoverable errors allowed).

`bn254_g1_add_valid_points_generator`

- **Strategy:** Fuzzes `bn254_g1_add` using valid points created via a generator point. There are no other curve points besides the points in the subgroup G_1, therefore there's no need to distinguish.
- **Assertion:** `Ok`.

`bn254_g1_add_valid_points_equation_solving`

- **Strategy:** Fuzzes `bn254_g1_add` using points generated by selecting a random x and solving for y.
- **Assertion:** `Ok`.

`bn254_g1_affine_deserialize_invalid_size`

- **Strategy:** Fuzzes `bn254_g1_affine_deserialize` with `BytesObject` inputs of invalid size.
- **Assertion:** `Err` (Result must be an error).

`bn254_g1_affine_deserialize_invalid_data`

- **Strategy:** Fuzzes `bn254_g1_affine_deserialize` with random data of valid size. This targets both $\geq p$ and $< p$ cases.
- **Assertion:** No Internal Error.

`bn254_g1_projective_serialize_uncompressed_random`

- **Strategy:** Fuzzes `bn254_g1_projective_serialize_uncompressed` using random projective points. No curve checks are performed or assumed required, i.e., x, y, and z are entirely random.
- **Assertion:** `Ok`.

`bn254_g1_mul_valid_size`

- **Strategy:** Fuzzes `bn254_g1_mul` with random data of valid size and a random `u256` scalar.
- **Assertion:** No Internal Error.

`bn254_g1_mul_valid_points_generator`

- **Strategy:** Fuzzes `bn254_g1_mul` using a valid point created via a generator point and a random scalar.
- **Assertion:** `Ok`.

`bn254_g1_mul_valid_points_equation_solving`

- **Strategy:** Fuzzes `bn254_g1_mul` using points generated by selecting a random x and solving for y.
- **Assertion:** `Ok`.

`bn254_fr_from_u256val_random_u256small`

- **Strategy:** Fuzzes `bn254_fr_from_u256val` using random `u256` small values (56-bit body).
- **Assertion:** No Internal Error.

`bn254_fr_from_u256val_random_u256bytes`

- **Strategy:** Fuzzes `bn254_fr_from_u256val` using random `u256` object values (host storage backed). The `u256` object values cover the entire $0..2^{256}$ range.
- **Assertion:** No Internal Error.

`bn254_multi_pairing_check_valid_size`

- **Strategy:** Fuzzes `bn254_multi_pairing_check` with random data of valid byte length, ensuring the two input vectors are of equal length.
- **Assertion:** No Internal Error.

`bn254_multi_pairing_check_valid_points_generator`

- **Strategy:** Fuzzes `bn254_multi_pairing_check` using valid G_1 and G_2 points derived from their respective generators. This ensures the pairing logic is exercised with valid curve points in the correct subgroups.
- **Assertion:** No Internal Error.

`bn254_multi_pairing_check_invalid_vector_length`

- **Strategy:** Fuzzes `bn254_multi_pairing_check` with vectors of unequal lengths or where both vectors have length 0.
- **Assertion:** `Err` (Result must be an error).

`bn254_g2_affine_deserialize_invalid_size`

- **Strategy:** Fuzzes `bn254_g2_affine_deserialize` with `BytesObject` inputs of invalid size.
- **Assertion:** `Err` (Result must be an error).

`bn254_g2_affine_deserialize_invalid_data_off_curve`

- **Strategy:** Fuzzes `bn254_g2_affine_deserialize` with random data of valid size. These random points are statistically almost guaranteed to be off-curve.
- **Assertion:** No Internal Error.

`bn254_g2_affine_deserialize_invalid_data_outside_subgroup`

- **Strategy:** Fuzzes `bn254_g2_affine_deserialize` using generated random points that lie on the G_2 curve but are not in the required prime-order subgroup.
- **Assertion:** `Err` (Result must be an error).

`bn254_crate_multi_miller_loop_valid_points_generator`

- **Strategy:** Directly fuzzes the function `multi_miller_loop` from the rust crate `ark_bn254` using valid G_1 and G_2 points generated via generators.
- **Assertion:** No crash (implicit).

`bn254_crate_final_exponentiation_random`

- **Strategy:** Fuzzes `final_exponentiation` with random `MillerLoopOutput` (Fp12) values.
- **Assertion:** No crash (implicit).

`bn254_check_pairing_output_random`

- **Strategy:** Fuzzes `bn254_check_pairing_output` with random `PairingOutput` (Fp12) values.
- **Assertion:** No crash (implicit).

Fuzzing Targets for Poseidon Host Functions

This work focuses on fuzzing the Poseidon and Poseidon2 permutation host functions, with the goal of validating both semantic correctness and robustness at the host boundary. While Poseidon2 is an evolution of Poseidon rather than a completely new design, it introduces meaningful differences in round structure and linear-layer optimizations. These differences make it important to test both implementations independently, even though they share a similar external interface.

At a high level, both permutations operate over a state vector of width `t`, with all arithmetic defined over a prime field. The permutation proceeds through a fixed number of rounds, each consuming round constants, applying S-boxes, and performing a linear mixing step via an MDS matrix. Several invariants must hold for the parameters to be well-formed:

- the input state has exactly `t` elements,
- the MDS matrix is `t × t`,
- the round-constant array has `rounds × t` elements.

Poseidon2 preserves this overall structure but changes the *shape* of the computation. Compared to Poseidon, it modifies the distribution of full and partial rounds and introduces optimized linear layers intended to reduce constraint costs in proof systems. These changes affect how round constants are indexed and how intermediate state elements are mixed. As a result, off-by-one errors or mismatches between `t`, round counts, and matrix dimensions may surface differently in Poseidon2 than in the original Poseidon, even when the high-level API appears similar.

Another important consideration is the host interface itself. Although Poseidon operates on prime-field elements, the host functions accept unsigned 256-bit integers from the client. These values are later interpreted as field elements internally. For fuzzing, we intentionally generate full-range 256-bit integers rather than pre-reduced field elements, expanding coverage to include reduction paths, modulus-boundary cases, and values that would never appear in a strictly field-typed implementation, while still reflecting realistic host-facing inputs.

To cover both correctness and safety, we implemented two complementary fuzz targets. The first is a differential fuzz test that compares the host's Poseidon and Poseidon2 outputs against a reference implementation from HorizenLabs, using hardcoded parameter sets for the BN256 and BLS12 instantiations; any divergence is treated as a correctness bug. The second is a crash-focused fuzz target that fully randomizes the permutation parameters and inputs, intentionally violating invariants to ensure that malformed or adversarial client data cannot trigger panics or out-of-bounds behavior in the host.

Fuzzing Targets

`poseidon_differential`

- **Strategy:** Fuzzes over the two poseidon permutation host functions, `poseidon_permutation` and `poseidon2_permutation` with randomly generated inputs for both the `BN254` and `BLS12_381` symbols and compares the outputs against the reference poseidon implementation by HorizenLabs.
- **Assertion:** The permutations from the host functions are equivalent to the permutations from the reference implementation.

`poseidon`

- **Strategy:** Fuzzes over the poseidon permutation host functions `poseidon_permutation` and `poseidon2_permutation`, over both `BN254` and `BLS12_381` symbols with randomly generated inputs and parameters with the aim of finding any panics or unrecoverable errors.
- **Assertion:** No internal error and no panics.

Empirical Metering Measurements

This section outlines our strategy for empirically analyzing CPU instruction metering in the new Soroban host functions and discusses our measurement data.

Empirical Metering Harness for Soroban Host Functions

The new host functions introduced by CAP: 0074 - Host Functions for BN254 and CAP: 0075 - Cryptographic Primitives for Poseidon/Poseidon2 Hash Functions must also deterministically charge for used compute resources. For this purpose, the Soroban execution environment deterministically estimates the compute resources used by a host function and the Wasm virtual machine, as defined in CAP 0046-10. Among these, Soroban host functions meter both CPU instructions and memory usage. The computation of total fees of a transaction takes into account CPU instructions. Memory usage, on the other hand, is metered for utilization by the network limit.

Soroban host function implementations, as well as their internally used functions, charge prior to CPU computation and memory usage, see e.g. `bn254_g1_mul_internal` in `rs-soroban-env` :

```
// soroban-env-host/src/crypto/bn254.rs
pub(crate) fn bn254_g1_mul_internal(
    &self,
    p0: G1Affine,
    scalar: Fr,
) -> Result<G1Projective, HostError> {
    self.charge_budget(ContractCostType::Bn254G1Mul, None)?;
    Ok(p0.mul(scalar))
}
```

Contract cost types can charge either using a constant term of the form a or a linear term of the form $a + b \cdot x$, where a and b are cost type parameters and x being a supplied variable that has to be passed whenever a given contract cost type specifies a linear term. By differentiating between several different contract cost types, different operations performed by a host function can charge different values. This allows for calibration of the costs of host functions with up to linear approximations. CAP 0074 explains that CPU instructions metering is calibrated by approximately measuring the number of physical CPU instructions executed.

Host functions must not undercharge given any input to the host function. Since the approximation uses only up to linear terms and measures the number of CPU instructions executed instead of actual execution time, the approximated metering will typically deviate from actual compute resources used. Therefore, it must be ensured that the approximation always overestimates the cost instead of underestimating, since underestimation could potentially be abused to cause a Denial-of-Service on the blockchain by having nodes process transactions with lots of compute resources below market price, depending on the scale of underestimation.

To independently confirm that the host functions metering do not severely underestimate CPU instruction metering, we built a tool that utilizes the same random data generation used in the fuzzing harnesses to measure both execution time and metered instructions for random inputs. We use both measurements to calculate the ratio of metered instructions and execution time to try to find metering underestimation outliers. By measuring execution time instead of the number of instructions executed by the actual CPU, our data more closely resembles what would be observed on actual blockchain nodes that run the host functions, not considering CPU model, architecture, and other micro-architectural differences. To avoid host object `Vec` appends to cause reallocation and copying of host objects, we reserve enough space ahead of time prior to measurements in the host object `Vec` , as this otherwise can cause outlier, or rather, wrong measurements.

Since measuring execution is non-deterministic and highly variable in typical environments, the mean of multiple measurements should be evaluated instead of a single measurement. The number of necessary measurements for a low relative error can be determined by computing the relative error of the mean value of the measurements and stopping the measurements for a given input once the relative error is less than a previously specified threshold. However, since adjacent measurements are assumed to be correlated, the relative error cannot be directly computed from all measurements since that would lead to the underestimation of the relative error. Therefore, we make use of batches of measurements and assume that the mean values of adjacent batches of measurements are uncorrelated. This allows us to correctly estimate the relative error, with the disadvantage of requiring more measurements. As the number of required measurements varies and depends on the relative error, different number of batches are

required. Therefore we implemented an algorithm that measures at least b batches and then combines adjacent batches once the number of batches reaches $2 \cdot b$, which halves the number of batches to b. The algorithm ensures that each batch always contains the mean of the same number of measurements. Since we calculate the `metered_instructions / execution_time` ratio using the measurement variable `execution_time`, the accuracy of the Linux monotonic clock used for timing measurements may impact the error of the `metered_instructions / execution_time` ratio. We henceforth assume this propagated error to be sufficiently small as we start our measurements with larger initial batch sizes, resulting in longer running measurements. In the future, the benchmark tool could be adapted to estimate and consider this propagated error as well. The algorithm, methodology, and mathematical considerations of the batch algorithm are further described in Simulation Modeling and Arena, chapter 5.4 [1].

Due to potential metering costs charged within loops and conditionals as well due to linear terms in costs, the metered CPU instructions might vary depending on input data. To isolate specific input patterns that may trigger undercharging, we analyze the `metered_instructions / execution_time` ratio across various input dimensions. By plotting these measurements against properties identified in the input data, such as, e.g., vector lengths, we can visually detect outliers where the execution time is anomalously high relative to the metered cost. For unbounded input data, like, e.g., host vectors, the safety margin should increase with input size. Consequently, the smallest valid inputs should yield the lowest `metered_instructions / execution_time` ratios. This ensures that calibrating for the smallest input data covers the worst-case scenario for the protocol. For data where we cannot identify suited properties in the input data, such as relatively simple host functions, we record the largest and smallest `metered_instructions / execution_time` ratios and evaluate whether they count as anomalies that identify potential undercharging. Ideally, for host functions where we don't identify suited properties in the input data that we can use for plotting, the variance of the measured `metered_instructions / execution_time` ratio should be low relative to other host functions.

The SDF team also contributed code to allow for measurements using Linux and MacOS instruction counters that heuristically measure the number of instructions executed by the physical CPU. Though these measurements are non-deterministic as well, making them suitable for the above algorithm as well. The measurement method can be selected in the tool. Since execution time is heavily dependent on micro-architectural optimizations, it is expected that the number of executed instructions and the execution time are correlated, but not identical. Further, the execution time will vary depending on various environmental factors and specific CPU model used.

As execution time measurements are non-deterministic and sensitive to environmental noise, we performed the benchmarks in an environment that is optimized for stability. Minimizing variance improves accuracy and significantly reduces the total testing time, as the algorithm will require fewer measurements to drive the relative error below the specified threshold. Specifically, we reduced environmental noise by ensuring the following:

- CPU core frequencies were pinned to a constant frequency to prevent thermal throttling.
- Dynamic frequency scaling features, such as,e.g., Intel Turbo Boost, were disabled to maintain fixed clock speeds.
- Hyper-threading was disabled to eliminate noise caused by shared execution units in the CPU micro-architecture.
- Interrupts were moved from the benchmark core to other cores, except for unmovable interrupts.
- Background services causing high load such as backup routines, were disabled.
- A Linux TTY was used without a graphical desktop environment to minimize background compute load.
- The benchmark core was isolated by using the kernel parameter `isolcpus=`. The benchmark process was then scheduled on that specific core by using the `taskset` utility.

This was implemented by using a dedicated NixOS configuration and specialization, see NixOS wiki. This allows for a reproducible measurement environment that can be conveniently selected from the bootloader alongside our already existing NixOS Linux installation. The metering benchmark statistics can found in the Empirical Metering Measurements Statistics.

[1] Rossetti, M.D. (2021). Simulation Modeling and Arena, 3rd and Open Text Edition. Retrieved from https://rossetti.github.io/RossettiArenaBook/ licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

Measurements for BN254 Host Functions

We ran the benchmark tool, introduced in Empirical Metering Harness for Soroban Host Functions, on the new BN254 host functions `bn254_g1_add`, `bn254_g1_mul`, and `bn254_multi_pairing_check`. We used the same code written for the BN254 fuzzing harness, introduced in Fuzzing Targets for BN254 Host Functions, to generate random valid inputs.

`bn254_g1_add`

The host function `bn254_g1_add` adds two G_1 points. Since the inputs are bounded, it is expected that the measured `metered_instructions / execution_time` ratios shouldn't vary too much for different valid inputs. Therefore, to evaluate the measured results, the measured mean is compared with the largest and smallest measured values to validate that there were no outliers in the measured samples.

Specifically, our measurements consist of 9161 samples that were executed with random valid input. On the benchmark hardware, a mean of $1.66e9$ instructions per second were measured. The sample with the largest computed `metered_instructions / execution_time` ratio has a mean of $1.78e9$ instructions per second. This value is relatively close to $1.66e9$ instructions per second and therefore not considered an outlier. The sample with the smallest computed `metered_instructions / execution_time` ratio has a mean of $1.56e9$ instructions per second. This value is also relatively close to $1.66e9$ instructions per second and therefore not considered an outlier.

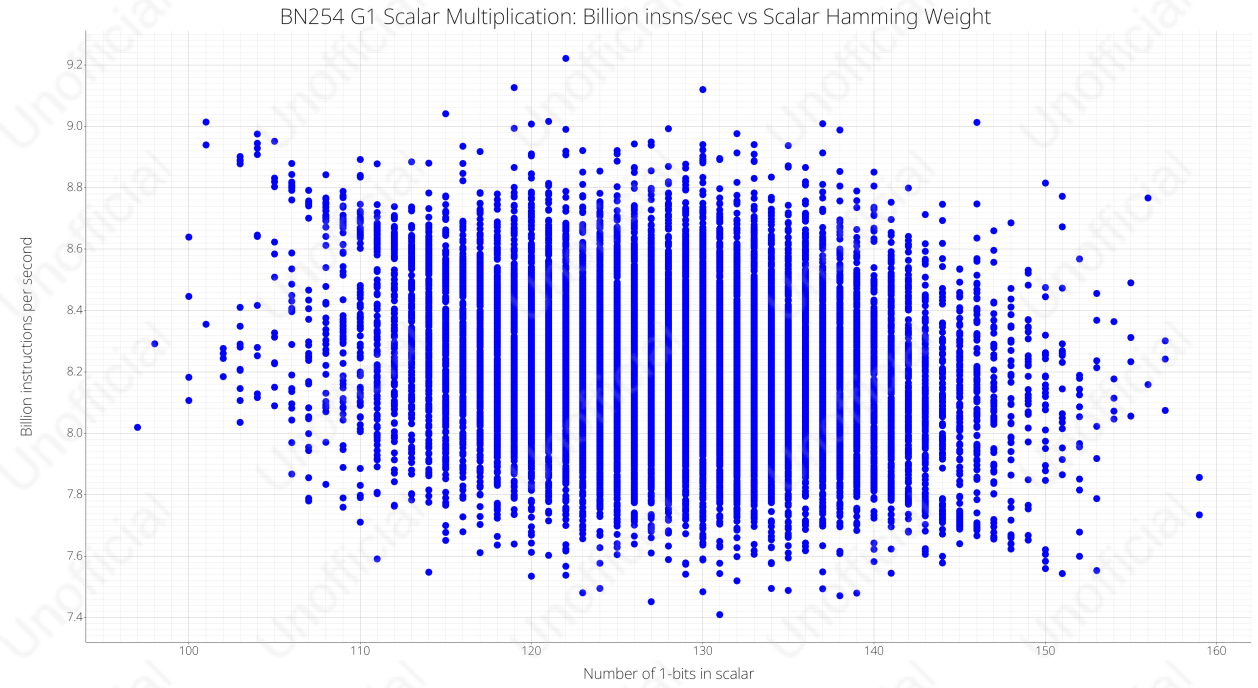
`bn254_g1_mul`

The host function `bn254_g1_mul` multiplies a G_1 point with a scalar.

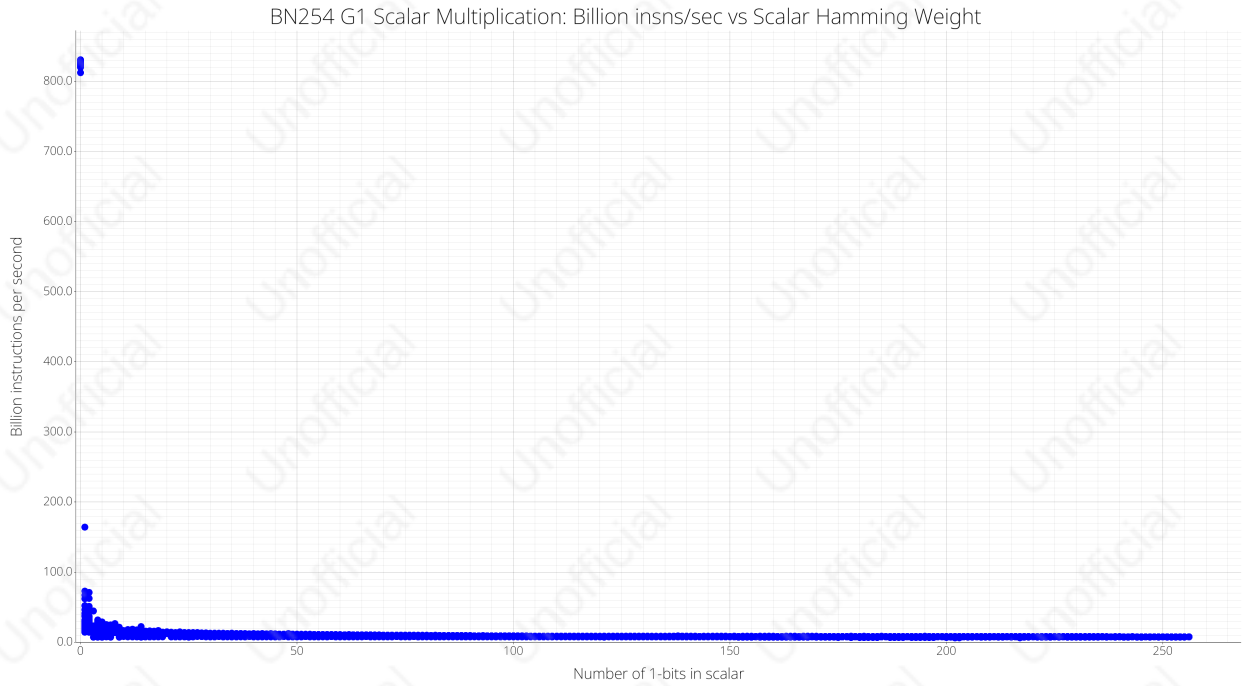
Since the multiplication is implemented with a double-and-add strategy that considers each bit of the scalar, the scalar can impact the execution time of the algorithm, see the implementation in crate `ark-ec-0.4.2`:

```
// `src/models/short_weierstrass/mod.rs`
fn mul_affine(base: &Affine<Self>, scalar: &[u64]) -> Projective<Self> {
    let mut res = Projective::<Self>::zero();
    for b in ark_ff::BitIteratorBE::without_leading_zeros(scalar) {
        res.double_in_place();
        if b {
            res += base
        }
    }
    res
}
```

Therefore, both leading zeros in the bits and the number of 1 bits can impact execution time. We expect that the measured `metered_instructions / execution_time` ratios correlate with the number of 1 bits in the scalar. As the `u256` scalar is bounded, we expect the average `metered_instructions / execution_time` ratio to stay roughly the same given any scalar.

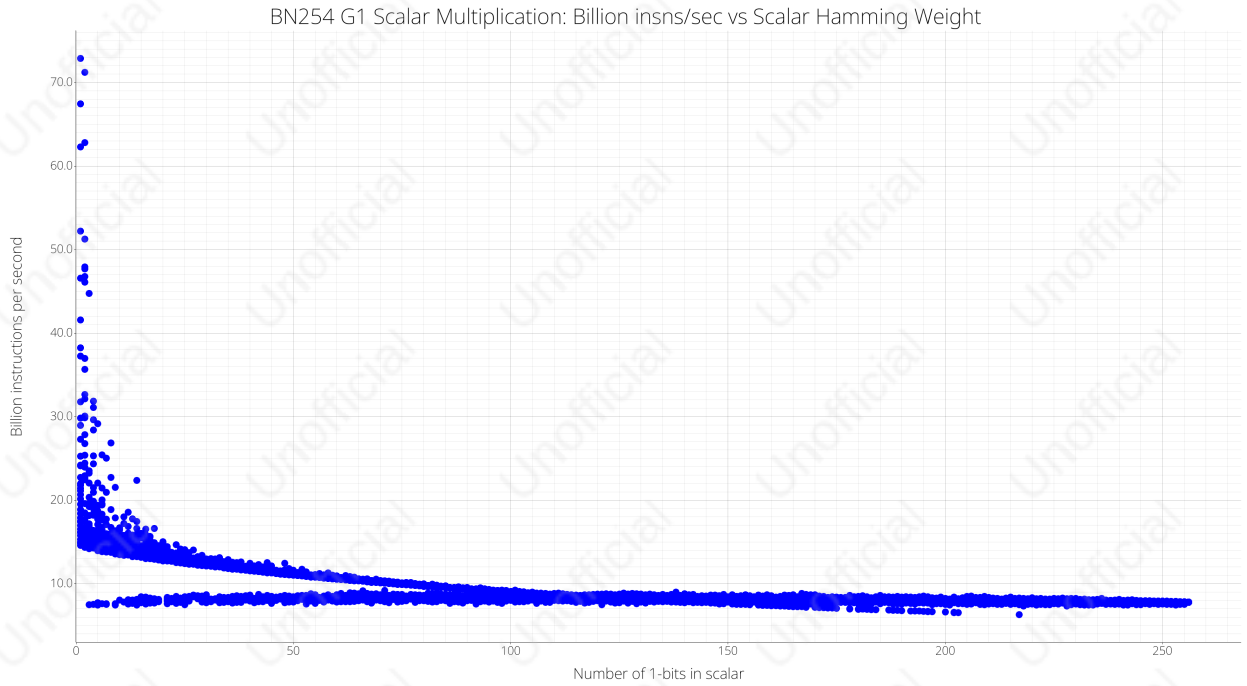


Our measurements consist of 20541 samples that were executed with random valid input. We can observe that the observed samples are over-proportionally distributed between 100 and 160 1-bits. This reflects the distribution of 1-bits in a random scalar. Since this distribution is unsuited to evaluate the average `metered_instructions / execution_time` ratio for all possible inputs, we created a variant of the `bn254_g1_add` metering benchmark that generates random scalars whose number of 1-bits is evenly distributed. We will evaluate the chart generated by that benchmarks data instead.



With even distribution of the number of 1-bits in the scalar, we can observe that samples with a low number of 1-bits in the scalar tend to be large outliers, having large `metered_instructions / execution_time` ratios. Specifically, we can observe samples that have > 61 times larger `metered_instructions / execution_time` ratios than observed in the worst cases of other here considered host functions, specifically Poseidon2. When measuring actual physical CPU instruction count using the same input values of the outliers, we can observe `metered_instruction / cpu_instruction` ratios of up to 127, while they should actually be rather close to 1. We consider these to be out-of-norm outliers with values that can in practice happen, as outlined in [M01] Host Function `bn254_g1_mul` Overcharges CPU Instructions when Scalar Input is Zero or Has Few 1-bits.

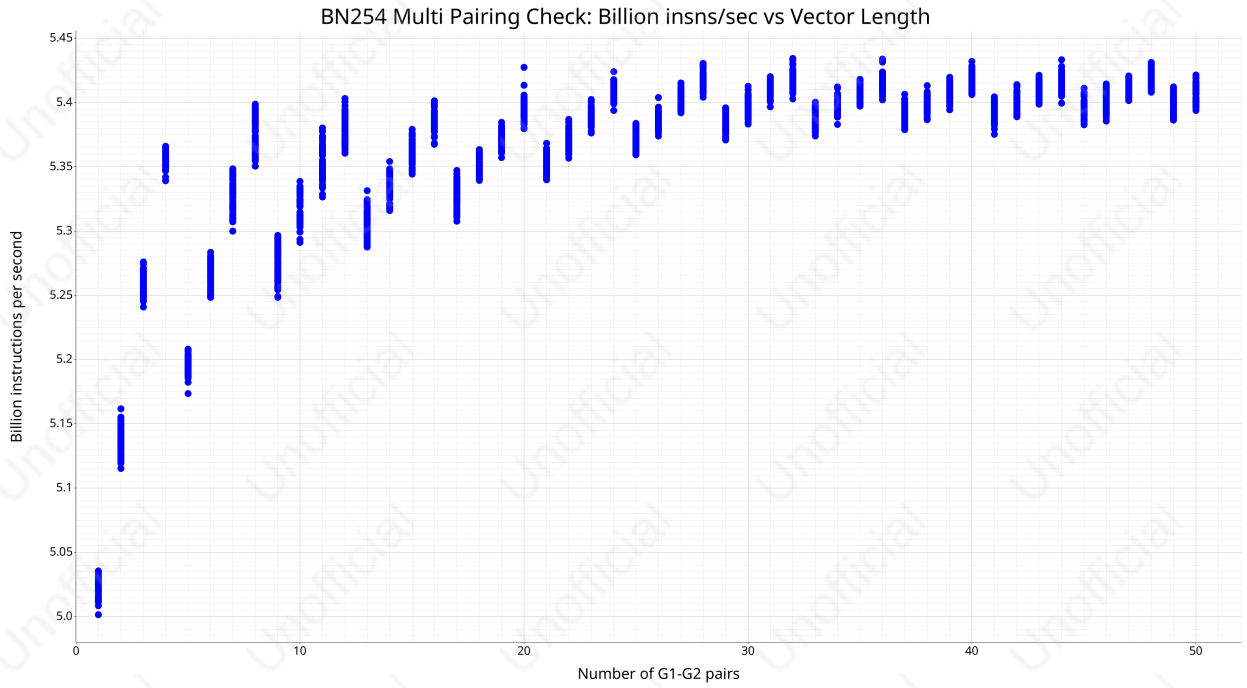
As the chart is hard to further evaluate with the outliers, we created a chart without any samples above $100e9$ `metered_instructions / execution_time`:



This chart allows us to consider the lower end of the distribution of samples. We can observe that the lowest measured ratios are distributed up to roughly $6e9$, which is a large enough ratio, compared to other host functions, that is not considered to contain outliers.

`bn254_multi_pairing_check`

The host function `bn254_multi_pairing_check` takes two vectors of serialized points and confirms whether these points pass a validity check. The host function takes scalable input arguments, namely two host vectors of theroretically arbitrary length that in practice are only constrained by gas limits for large vectors. Therefore, it is expected that the measured `metered_instructions / execution_time` ratios correlate with the length of the input vectors. Since the function expects both input vectors to be of same length, only input vectors of identical length are considered. Furthermore, we expect the average `metered_instructions / execution_time` ratio to increase the larger the input vector lengths. To evaluate the measurements, we plot the measured `metered_instructions / execution_time` ratios on a graph combined with the respective sample's input vector length on the x-axis.



Our measurements consist of 3411 samples with the input vector length constrained to lengths between 1 and 50, since larger vector lengths incur higher execution times. The graph confirms that the measured `metered_instructions / execution_time` ratios increase for larger input vector lengths. Additionally, the measured ratios are distributed between $5.0e9$ and $5.45e9$, which is a narrow range that is not considered to contain outliers.

In addition, the measurements show a pattern, where the measured `metered_instructions / execution_time` ratios are decreased whenever the number of pairs increases from a multiple of four ($4 \cdot n$) to a multiple of four plus one ($4 \cdot n + 1$). We hypothesize that this happens due to the arkworks library using a chunk size of 4 in their algorithm that is used by the host function:

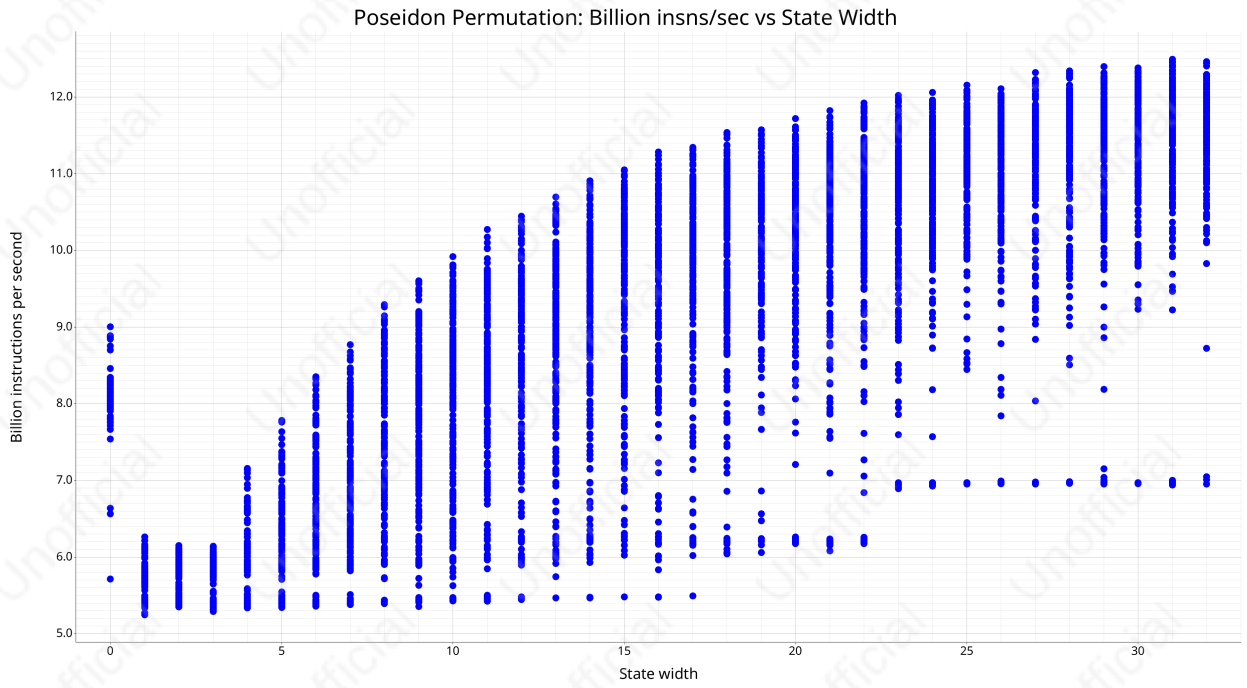
```
// function multi_miller_loop in `soroban-env-host/src/crypto/bn254.rs`
let mut f = cfg_chunks_mut!(pairs, 4)
    .map(|pairs| {
        // [...]
    })
    .product::<<Bn<Self> as Pairing>::TargetField>();
```

Measurements for Poseidon Host Functions

We ran the benchmark tool, introduced in Empirical Metering Harness for Soroban Host Functions, on the Poseidon host functions `poseidon_permutation` and `poseidon2_permutation`. We used the same code written for the Poseidon fuzzing harness, introduced in Fuzzing Targets for Poseidon Host Functions, to generate random valid inputs and parameters.

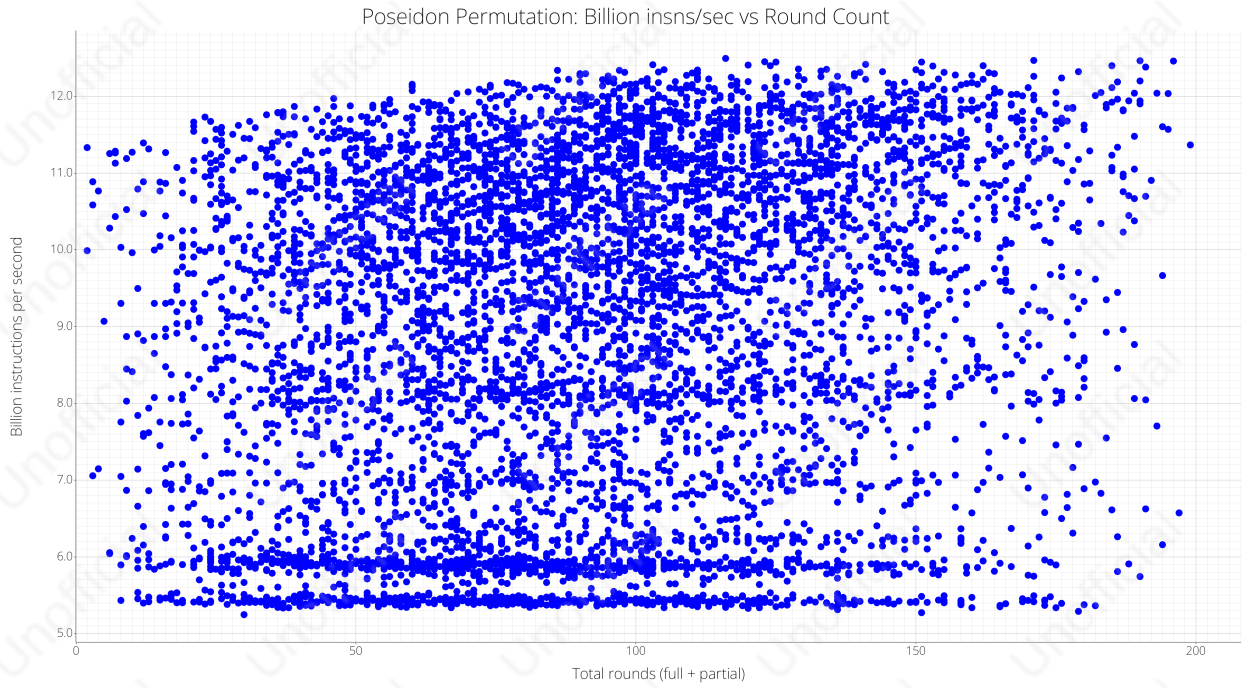
`poseidon_permutation`

The host function `poseidon_permutation` performs the Poseidon hash permutation on a vector of field elements. The function takes scalable input arguments, specifically a state vector of width t and round numbers `rounds_f` and `rounds_p`. Therefore, it is expected that the measured `metered_instructions / execution_time` ratios correlate with the state width and the sum of both round numbers. To evaluate the measurements, we plot the measured `metered_instructions / execution_time` ratios on a graph combined with the respective sample's state width and round number on the x-axis, respectively.

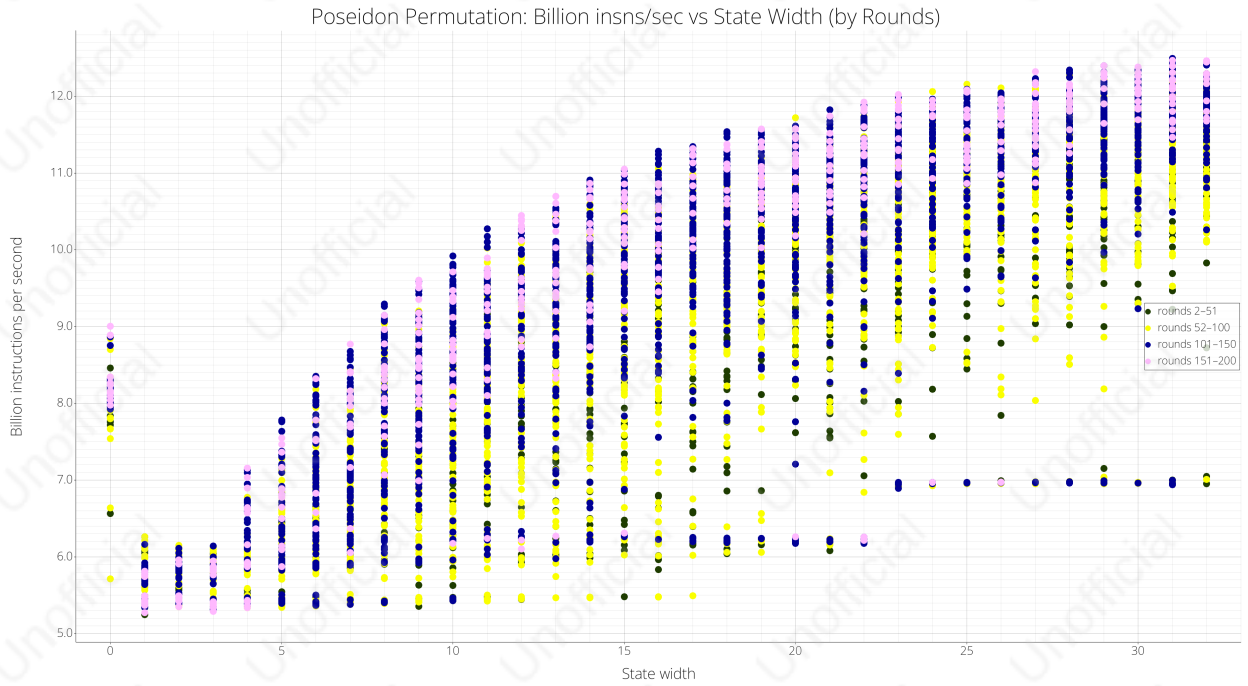


Our measurements consist of 6652 samples with the state width t randomly generated between 0 and 32. The graph confirms that the measured `metered_instructions / execution_time` ratios increase for larger state widths. Additionally, the measured ratios are distributed between $5.2e9$ and $12.6e9$, which is a range that is not considered to contain outliers.

We can also observe that the measured `metered_instructions / execution_time` ratios are differently distributed when the state width t equals 0, though all these observed points are considerably lower than the overall observed maximum `metered_instructions / execution_time` ratio, and therefore not considered a critical outlier.



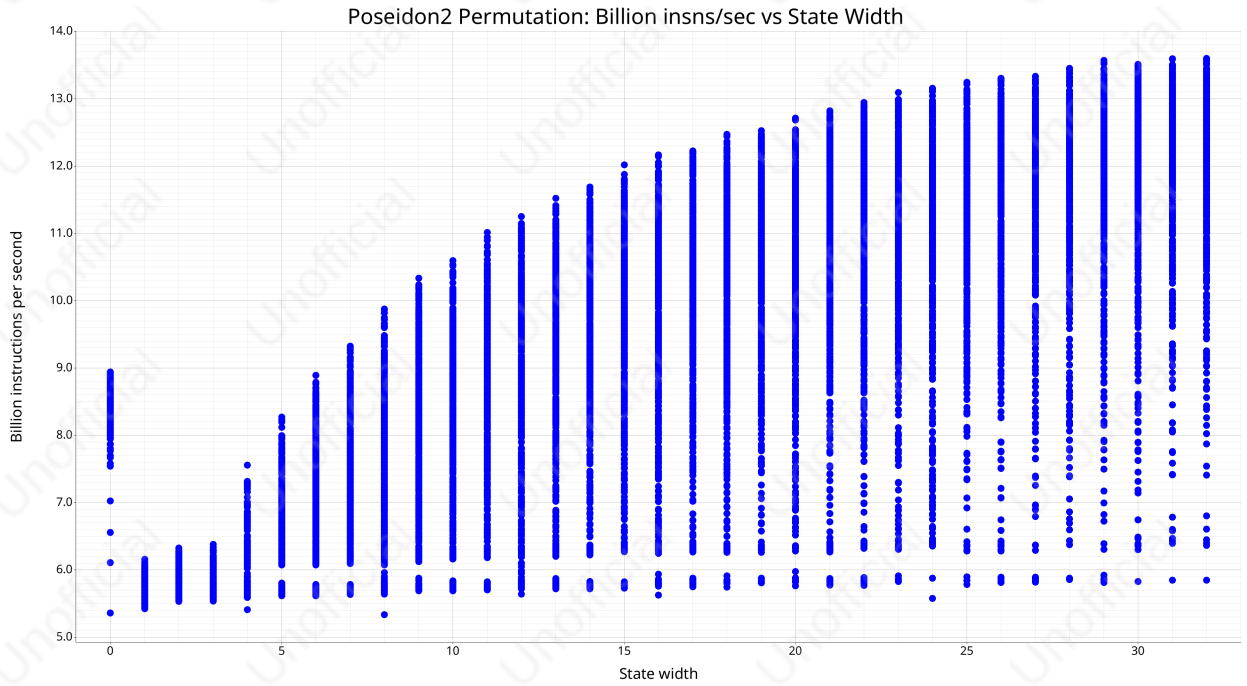
The second graph shows that the round number also correlates with the `metered_instructions / execution_time` ratios, but far less compared to state width. This graph also confirms that the measured `metered_instructions / execution_time` ratios tend to increase for larger round numbers.



As we observed a large distribution of `metered_instructions / execution_time` ratios in both previous charts, we combined both metrics by clustering different ranges of rounds using different colors and having state width be the x-axis. In this chart, we can observe that higher round numbers do tend to result in higher `metered_instructions / execution_time` ratios. Though in addition we can see that the ratios are still distributed across the entire range of observed values.

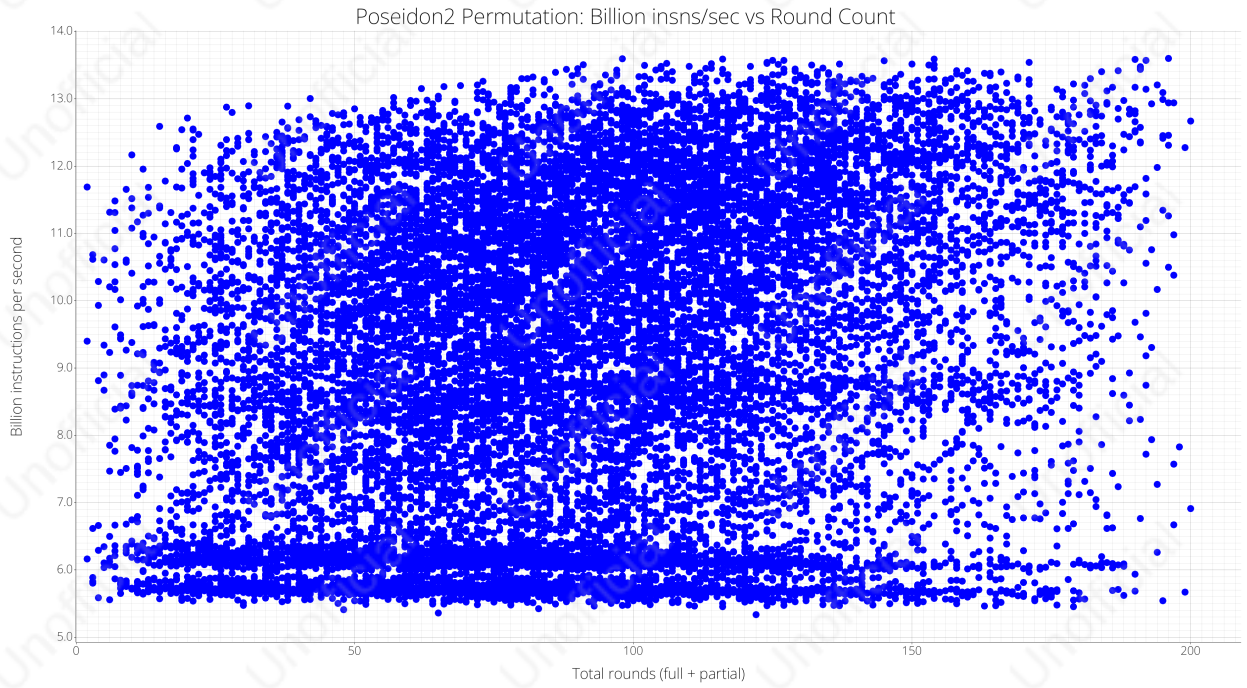
`poseidon2_permutation`

The host function `poseidon2_permutation` performs the optimized Poseidon2 permutation. The function takes scalable input arguments, specifically a state vector of width t and round numbers `rounds_f` and `rounds_p`. Therefore, it is expected that the measured `metered_instructions / execution_time` ratios correlate with the state width and the sum of both round numbers, similarly to `poseidon_permutation`. To evaluate the measurements, we plot the measured `metered_instructions / execution_time` ratios on a graph combined with the respective sample's state width and round number on the x-axis, respectively.

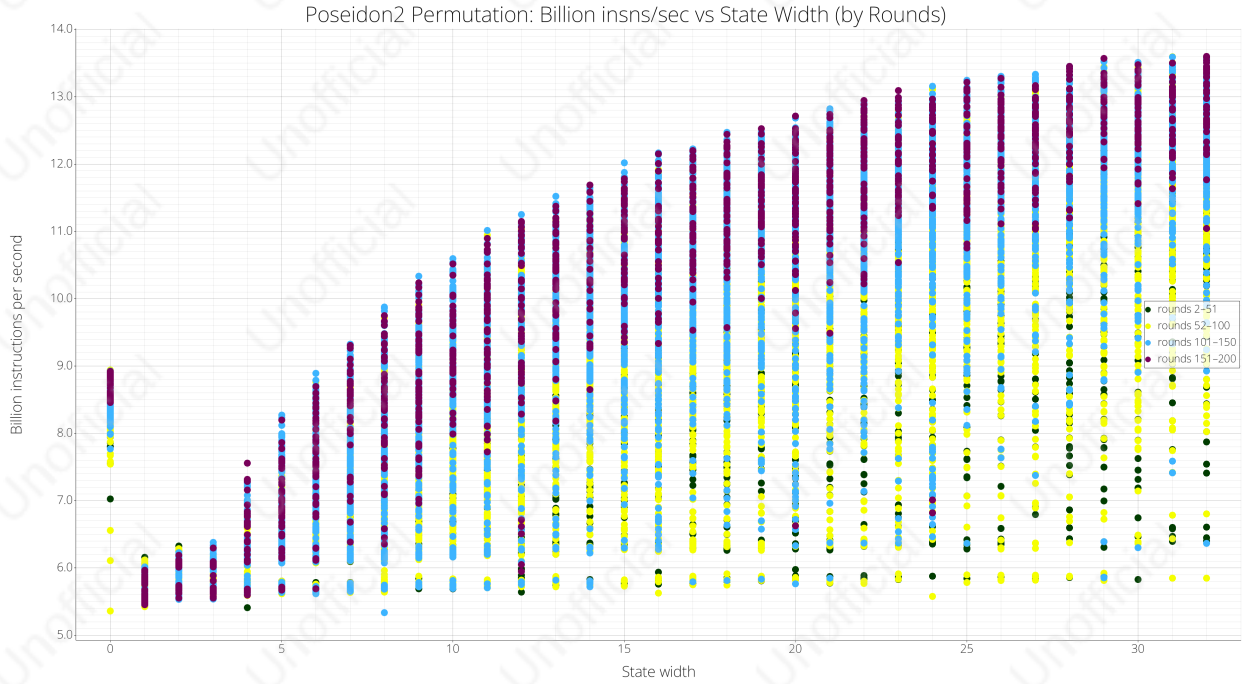


Our measurements consist of 17724 samples with the state width t randomly generated between 0 and 32. The graph confirms that the measured `metered_instructions / execution_time` ratios increase for larger state widths. Additionally, the measured ratios are distributed between $5.3e9$ and $13.7e9$, which is a range that is not considered to contain outliers.

Similar to `poseidon_permutation`, we can observe that the measured `metered_instructions / execution_time` ratios are differently distributed when the state width t equals 0, though all these observed points are considerably lower than the overall observed maximum `metered_instructions / execution_time` ratio, and therefore not considered a critical outlier.



The second graph shows that the round number also correlates with the `metered_instructions / execution_time` ratios, but far less compared to state width. This graph also confirms that the measured `metered_instructions / execution_time` ratios tend to increase for larger round numbers.



As we observed a large distribution of `metered_instructions / execution_time` ratios in both previous charts, we combined both metrics by clustering different ranges of rounds using different colors and having state width be the x-axis. In this chart, we can observe that higher round numbers do tend to result in higher `metered_instructions / execution_time` ratios. Though in addition we can see that the ratios are still distributed across the entire range of observed values.

Appendix

Nothing to preview

Fuzz Run Statistics

Target	Run time	Iterations	Crashes	Report link
poseidon_differential	24h	64,509,485	0	report
poseidon	24h	29,681,876	0	report
bn254_check_pairing_output_random	8h	1,526,818,520	0	report
bn254_crate_final_exponentiation_random	8h	47,663,781	0	report
bn254_crate_multi_miller_loop_valid_points_generator	8h	10,594,624	0	report
bn254_fr_from_u256val_random_u256bytes	72h	21,339,885,052	0	report
bn254_fr_from_u256val_random_u256small	72h	28,574,403,420	0	report
bn254_g1_add_valid_points_equation_solving	8h	582,778,692	0	report
bn254_g1_add_valid_points_generator	8h	257,745,620	0	report
bn254_g1_add_valid_size	15h	6,910,964,941	0	report
bn254_g1_affine_deserialize_invalid_data	72h	29,176,052,496	0	report
bn254_g1_affine_deserialize_invalid_size	72h	40,058,083,578	0	report
bn254_g1_mul_valid_points_equation_solving	8h	516,948,318	0	report
bn254_g1_mul_valid_points_generator	72h	2,378,424,810	0	report
bn254_g1_mul_valid_size	72h	23,348,791,689	0	report
bn254_g1_projective_serialize_uncompressed_random	72h	15,756,384,088	0	report
bn254_g2_affine_deserialize_invalid_data_off_curve	8h	7,229,897,590	0	report
bn254_g2_affine_deserialize_invalid_data_outside_subgroup	8h	29,657,405	0	report
bn254_g2_affine_deserialize_invalid_size	8h	2,433,444,517	0	report
bn254_multi_pairing_check_invalid_vector_length	8h	847,686,524	0	report
bn254_multi_pairing_check_valid_points_generator	8h	8,571,610	0	report*
bn254_multi_pairing_check_valid_size	8h	1,963,612,954	0	report*

Empirical Metering Measurements Statistics

Host Function	Max. Relative Error	Max. Batch Size	Run Time	Samples	Notes
`bn254_g1_add`	0.12%	50	5h 0m 0s	9161	
`bn254_g1_mul`	0.11%	50	5h 0m 0s	20541	
`bn254_g1_mul`	0.15%	50	5h 0m 0s	8501	evenly distributed 1-bits
`bn254_multi_pairing_check`	0.06%	40	10h 15m 47s	3411	
`poseidon_permutation`	0.09%	50	5h 0m 0s	6652	
`poseidon2_permutation`	0.11%	50	5h 0m 0s	17724	

Benchmark Hardware

The benchmarks were run on a system with an environment described in more detail in Empirical Metering Harness for Soroban Host Functions. Specifically, the benchmarks were run on:

- The second core of an Intel Core i7-1355U set to 1.7 GHz
- Linux kernel 6.12.65