Triton VM Code Audit Final Report by Hridam Basu

Overview

Background

Triton VM has requested Hridam Basu for a cryptographic code audit of the Triton VM project based on ZK-STARKs. Triton VM is used in STARK-based private payment system called Neptune Cash.

Project Dates

- 30th September - Audit Starts
- 21st October - Delivery of Initial Audit Report
- 11th November - Delivery of Final Audit Report

Review Team

Hridam Basu, Senior Cryptography Researcher and Engineer
Past Experience: Ethereum Foundation - Privacy and Scaling Explorations (PSE), Polygon Technology, Aztec Network, Findora, Bolt Labs.
Short Stints: AT&T Research Labs New York City, NTT Research Labs Tokyo, TCS Innovations Labs India.
Education: Masters degree from Northeastern University, USA and Undergrad from Jadavpur University, India.

Coverage

Target Code and Revision

The audit covers repository triton-vm on https://github.com/TritonVM/triton-vm branch master at the time of commencement of the audit, namely **v0.42.1** and commit id **d2c6fc7**. The following folders were audited from the Triton-VM repository:

- **Triton ISA**: https://github.com/TritonVM/triton-vm/tree/master/triton-isa
- **Triton VM**: https://github.com/TritonVM/triton-vm/tree/master/triton-vm
- **Triton AIR**: https://github.com/TritonVM/triton-vm/tree/master/triton-air

Supported Documentation

The Triton VM book is found at https://triton-vm.org/spec/. This book contains detailed theoretical and mathematical descriptions of each of the constituent components of the Triton VM.

Areas of Concern

Our investigation focussed on the following areas:

- Correctness of the implementation
- Soundness of the Stark Verify function
- Vulnerabilities in the Code leading to attacks
- Cryptographic errors leading to basic mathematical vulnerabilities
- Code Quality issues including documentation

Findings

We have examined the Triton-AIR in detail for the arithmetization procedure. In particular, we have carefully verified all the 4 types of constraints defined in the protocol specification and code base corresponding to each of the trace tables mentioned in detail in the rest of the document. All of the constraints in the codebase match exactly with the constraints mentioned in the specification of Triton VM. We have not identified any cryptographic soundness error in the entire code.

Detailed Analysis of the Cryptographic Code

This document provides an in-depth analysis of the cryptographic components of Triton-VM as utilized in Neptune Cash, focusing on core components within the Triton-AIR, Triton-VM, and Triton-ISA folders. These elements enable verifiable computation based on STARKs (Scalable Transparent Arguments of Knowledge), an efficient and secure proof system. Below is a refined breakdown of how these components interact, along with an examination of the tables and data structures.

Overview of Triton-VM Architecture

Triton-VM is a virtual machine designed to execute and verify cryptographic computations using the STARK framework. This framework provides post-quantum security, scalability, and transparency without requiring trusted setups. Triton-VM comprises three primary components:

- **Triton-VM**: The virtual machine that manages state transitions and computation.
- **Triton-AIR**: The Algebraic Intermediate Representation (AIR) component, defining the constraints that describe the evolution of the VM's state.

- **Triton-ISA**: The instruction set architecture (ISA), which defines the operations the VM supports.

These components work in tandem to facilitate efficient and secure verifiable computations.

Triton-VM: Execution Model

State Transition and Execution Trace

Triton-VM operates on a series of state transitions captured in an execution trace, with each step in the trace representing the VM's state at a specific point in the computation. This execution trace, termed the Algebraic Execution Trace (AET), is the sequence of states that (in an honest scenario) satisfy the AIR constraints. Each state includes key components:

- **Program Counter (PC)**: Tracks the current instruction being executed.
- **Stack**: Holds intermediate values and maintains the state of the VM's stack.
- **Memory**: Stores data loaded during computation, with elements sorted by memory address rather than cycle count.
- **Input/Output Buffers**: Manage data inputs and outputs during execution.
- **Registers**: Store critical values, such as hash accumulators or jump targets.

The VM operates in cycles, with each cycle processing one instruction and updating the VM's state accordingly. The execution trace grows with each step, representing the VM's state evolution as constrained by Triton-AIR.

Triton-AIR: Algebraic Intermediate Representation

Triton-AIR provides the algebraic foundation for the STARK proof system, translating the VM's state evolution into polynomial constraints. The AET satisfies these constraints, forming the core of the proof.

Trace Tables

Triton-AIR organizes the execution trace into several tables, each serving different purposes and indexed by different criteria:

- **Program Table**: Contains the program's instructions at each step in the trace, enforcing the program counter's correct progression.
- **Processor Table**: Tracks the processor's internal state, including stack and register values, and enforces valid state transitions.
- **Op Stack Table**: Contains the operand stack's contents, sorted by the stack pointer, and ensures correct stack operations (push, pop).
- **RAM Table**: Represents memory content, sorted by memory address, and enforces consistency for memory read/write operations.

- **Jump Stack Table**: Tracks jump operations (such as calls), ensuring correct control flow without native loop support.
- **Hash Table**: Contains intermediate hash values, verifying consistency for cryptographic operations using Tip5 Hash.
- **Cascade Table**: Manages cascading constraints in the execution trace, enforcing consistent, multi-step dependencies for computations with nested or sequential operations, ensuring values propagate accurately.
- **Lookup Table**: Facilitates efficient, verifiable lookups, validating data dependencies and cross-references between tables (e.g., Processor, RAM) to maintain computation integrity.
- **U32 Table**: Verifies 32-bit unsigned integer operations, enforcing constraints for arithmetic tasks, handling overflow, and ensuring secure state transitions within the 32-bit domain.

Each of these tables is represented as polynomials in Triton-AIR, with constraints ensuring valid execution. These constraints do not enforce behavior directly; instead, the verifier only accepts a proof if the AET satisfies the AIR constraints.

Constraints

- **Initial Constraints**: Set the initial conditions for the VM's state, such as stack and register values.
- **Consistency Constraints**: Ensure that values across cycles remain consistent where required.
- **Transition Constraints**: Enforce valid state transitions, such as ensuring stack and memory operations behave as expected.
- **Terminal Constraints**: Ensure the final state meets the required conditions, checking outputs and final register values.

Triton-ISA: Instruction Set Architecture

The Triton-ISA defines the instructions Triton-VM can execute, supporting fundamental operations without conditional jumps, which would disrupt the zero-knowledge property. The ISA includes the following key instructions:

- **Arithmetic Operations**: Basic operations like addition, subtraction, multiplication, and division.
- **Memory Operations**: Load and store instructions for accessing memory.
- **Stack Operations**: Push and pop operations for managing the operand stack.
- **Control Flow**: A single `call` instruction supports function-like jumps without native loops or conditional jumps.

- **Hash Operations**: Cryptographic operations like Tip5 hashes are zk-STARK friendly and verified using the Hash Table in Triton-AIR.

Interplay Between Triton-VM, Triton-AIR, and Triton-ISA

The three components of Triton work together to create a robust verifiable computation model:

1. **Triton-VM** generates an execution trace by processing instructions defined in Triton-ISA.
2. **Triton-AIR** converts this trace into polynomial constraints that validate the VM's state transitions, stack behavior, and memory accesses.
3. **Triton-ISA** defines the VM's instruction semantics, shaping the AIR constraints in Triton-AIR.

Through this collaboration, the execution trace, tables, and constraints ensure that the VM's execution can be verified via STARK proofs, providing Neptune Cash with scalable, post-quantum secure guarantees on computation correctness.

Tables and Their Interactions

Tables defined in previous section interact to ensure the verifiability of computations without native loop or conditional jump support, preserving zero-knowledge properties and scalability.

Conclusion

The Triton-VM architecture, combined with Triton-AIR and Triton-ISA, creates a secure, efficient environment for verifiable computation using STARKs. This setup allows privacy-focused applications like Neptune Cash to cryptographically prove computation correctness, leveraging trace tables, polynomial constraints, and a minimal instruction set. The careful alignment of execution traces, data tables, and algebraic constraints ensures robust verifiability, aligning with Neptune Cash's commitment to secure, privacy-preserving computation.

General Comments

Code Quality

We performed a manual review of the repositories in scope and found the codebase to be generally organised and well-written. However, we found that the implementation uses assertions at several places in AIR, ISA and VM to verify the correctness of input data or computations while we recommend using error messages instead to allow the user to benefit from a more graceful handling of such cases.

Documentation And Code Comments

Website:
In addition, this audit report was prepared with the following documents as a reference:

- Triton VM Specification
- The Tip5 Hash Function for Recursive STARKs by Alan Szepieniec, Alexander Lemmens, Jan Ferdinand Sauer, Bobbin Threadbare, and Al-Kindi
- Stark Anatomy by Alan Szepieniec
- Multivariate lookups based on logarithmic derivatives: by Ulrich Haböck
- DEEP-FRI: Sampling Outside the Box Improves Soundness: by Eli Ben-Sasson, Lior Goldberg, Swastik Kopparty, Shubhangi Saraf

Scope

The audit does not include code from the following folders:

- **Triton Constraint Builder**: Link
- **Triton Constraint Circuit**: Link

The audit also does not include the following:

- Tests
- Benchmarks
- Code related to the naked operation of the VM (without proving and verifying anything) and code written in tasm

Dependencies

We did not identify any vulnerabilities in the implementation's use of dependencies.

Specific Issues And Suggestions

Issue A: Error propagation instead of unwrap

Location

- Triton VM Stark
- Jump Stack Table
- ISA - OP Stack
- Lookup Table
- Master Table (1)
- Master Table (2)
- Master Table (3)

- OP Stack Table
- Processor Table (1)
- Processor Table (2)
- Processor Table (3)

Synopsis

The code currently uses `unwrap()` to handle potential errors during operations involving array creation and type conversions. While `unwrap()` works in certain situations, it causes the program to panic if an error occurs, which can be problematic in production environments. Instead of using `unwrap()`, error propagation should be applied to handle these potential failures gracefully.

Impact

- Current Behavior: When an error occurs during the creation of an array or the conversion of a vector, the `unwrap()` call will cause the program to panic, leading to an abrupt termination of the application. This approach is risky in systems where robustness is critical (e.g., cryptographic proving systems, blockchain environments).
- Proposed Change: Error propagation will replace `unwrap()`, allowing errors to be returned and handled appropriately without crashing the program.
- Impact on Security and Stability: Removing `unwrap()` in favor of error propagation improves the reliability of the code, ensuring that unexpected errors do not cause the system to crash. This is crucial for systems where continuous uptime and resilience are essential.

Feasibility

- Replacing `unwrap()` with proper error propagation is straightforward. Both cases can be modified to return a Result and propagate the error upwards.
- Rust's error-handling model makes it easy to propagate errors using Result and the ? operator. No significant changes in logic are required, making this update highly feasible without disrupting the existing code.

Technical Details

We describe the technical details with respect to two out of the eleven locations mentioned above:

Current Code:

- Line 1:

```
Array2::from_shape_vec((main_table.nrows(), 1), auxiliary_column).unwrap()
```

- Line 2:

```
let out_of_domain_curr_row_quot_segments = quotient_segment_polynomials
    .map(|poly| poly.evaluate(out_of_domain_point_curr_row_pow_num_segments))
    .to_vec()
    .try_into()
    .unwrap();
```

Proposed Code:

- Line 1 Replacement:

```
let array = Array2::from_shape_vec((main_table.nrows(), 1), auxiliary_column)
    .map_err(|e| ProvingError::ArrayCreationError(e.to_string()))?;
```

This replaces `unwrap()` with proper error propagation by returning a Result if array creation fails, and an appropriate error message is passed upward.

- Line 2 Replacement:

```
let out_of_domain_curr_row_quot_segments: Result<_, ProvingError>
= quotient_segment_polynomials
    .map(|poly| poly.evaluate(out_of_domain_point_curr_row_pow_num_segments))
    .to_vec()
    .try_into()
    .map_err(|_| ProvingError::ConversionError
        ("Failed to convert polynomial segments".to_string()))?;
```

This replaces `unwrap()` with error propagation in the conversion logic, returning an error with a custom message if the `try_into()` conversion fails.

- Error Types: New error variants (e.g., ArrayCreationError, ConversionError) can be added to the custom ProvingError enum to handle these cases gracefully.

Remediation

- Remove `unwrap()` Calls: Replace all instances of `unwrap()` in the code with proper error handling using Result types, ensuring that errors are propagated upwards instead of causing a panic.
- Use `map_err()` for Custom Error Messages: Convert errors into meaningful messages that can help developers understand the cause of failure. This can be done by leveraging the `map_err()` function.
- Error Documentation: Clearly document the types of errors that could occur in the function signatures and ensure developers know how to handle these errors effectively.

Status

The Triton VM team partially acknowledged our issue and argued that they should document the cases where the unwraps cannot fail. For the others, they would consider returning an error.

Verification

Partially resolved.

Issue B: Integer Overflow error: Use Saturated Add instead of incrementing +=

Location

- Program (1)
- Program (2)
- AET
- VM (1)
- VM (2)
- VM (3)
- VM (4)
- VM (5)
- VM (6)
- VM (7)
- VM (8)
- VM (9)
- VM (10)
- VM (11)
- VM (12)

Synopsis

An integer overflow vulnerability exists where the code increments a variable using the += operator. If the variable exceeds the maximum limit for its type, it will wrap around to the minimum value, leading to potential errors or unintended behavior. Implementing Saturated Add will ensure that when the maximum value is reached, it will no longer increment, thus preventing overflow.

Impact

If not addressed, this vulnerability could cause incorrect calculations, data corruption, or crashes. In certain systems, this overflow could be exploited to cause denial of service (DoS) or escalate privileges.

Feasibility

Mitigating this issue is feasible with minimal changes in the code. Replacing += with a Saturated Add function in the affected places can eliminate the overflow risk without affecting the overall logic of the application.

Technical Details

The vulnerable code uses `+=` to increment an integer value without bounds checking. If the variable exceeds the maximum value for its type (e.g., `u32 or u64`), it will wrap around. By using Saturated Add (e.g., `u32::saturating_add()` in Rust), the value will remain at the maximum limit if an overflow would have occurred.

Remediation

Replace all instances of `+=` that are prone to overflow with a Saturated Add operation, such as `saturating_add()` in Rust. This will ensure that the value is clamped at the type's maximum instead of wrapping around to the minimum.

Status

The Triton VM team has again partially acknowledged the issue and argued that in some cases, it is better to fail on overflow.

Verification

Partially resolved.

Issue C: Iteration Improvement in OP Stack

Location

OP Stack

Synopsis

The IntoIterator implementation for OpStack is currently designed to reverse the stack in place before returning an iterator. The proposed change leverages the `iter()` method in conjunction with `rev()` and `cloned()` to iterate over the elements in reverse without modifying the original stack.

Impact

- Current Behavior: The original implementation mutates the internal state of OpStack by reversing the stack, which could lead to unexpected behavior if the original stack is needed afterward. This can be problematic in contexts where multiple iterations or accesses to the original order are required.

- Proposed Behavior: The improved implementation safely iterates over the stack in reverse order without altering its original state. This change ensures that the stack remains intact for subsequent operations, enhancing the robustness of the code.

Feasibility

- Effort to Fix: The transition to the improved method requires minimal changes, primarily modifying the into_iter function.
- Applicability: This solution is applicable in scenarios where immutability of the original stack is essential, such as in concurrent contexts or when the stack is accessed in multiple places.

Technical Details

- Original Implementation:

```
fn into_iter(self) -> Self::IntoIter {
    let mut stack = self.stack;
    stack.reverse(); // Mutates the stack
    stack.into_iter() // Returns an iterator over the reversed stack
}
```

This code mutates the stack, resulting in potential side effects if the stack is accessed elsewhere after iteration.

- Improved Implementation:

```
fn into_iter(self) -> Self::IntoIter {
    self.stack.iter().rev().cloned().collect::<Vec<_>>().into_iter()
}
```

This revised approach creates an iterator over the stack in reverse order without changing its internal structure. It clones the elements into a new vector, ensuring that the original OpStack remains unaltered.

Remediation

- Adopt the Improved Method: Transition to the new implementation that prevents mutation of the OpStack. This change increases safety and predictability in the code's behavior.
- Monitor for Performance Impacts: Although the improved method enhances safety, it introduces a cloning overhead. It is advisable to profile the performance to ensure it meets requirements in high-performance scenarios.

Status

The Triton VM team has refuted our claim and said that "self" cannot be used afterwards. We acknowledge their point.

Verification

Resolved.

Issue D: Uniform Sequence Check Enhancement

Location

OP Stack

Synopsis

The function `is_uniform_sequence()` checks if all elements in a given sequence are of the same type by comparing them to the first element. However, if the sequence is empty, accessing `sequence[0]` will lead to a panic due to out-of-bounds indexing. This presents a risk of runtime errors and instability in systems using this function.

Impact

If called with an empty sequence, the current implementation will panic, potentially crashing the application or causing unintended behavior. This issue could arise in scenarios where user input, dynamically generated data, or external factors result in an empty sequence being passed to this function. The impact ranges from minor to severe, depending on how the function is used in the application.

Feasibility

The issue is easy to reproduce by passing an empty sequence `(&[])` to the function, triggering a panic. Fixing the issue is straightforward and involves adding a guard clause to check for an empty sequence before performing the comparison logic.

Technical Details

- Current behavior: The function does not account for the case where sequence is empty. When an empty sequence is passed, accessing `sequence[0]` results in a runtime error.
- Proposed change: Introduce a condition to check if the sequence is empty `(sequence.is_empty())` at the beginning of the function. If it is empty, return true or handle the case as per the desired application behavior (e.g., returning false if a uniform sequence cannot exist in an empty set).

Remediation

Modify the function to handle empty sequences gracefully. Example fix:

```
pub fn is_uniform_sequence(sequence: &[Self]) -> bool {
    if sequence.is_empty() {
        return true;
        // Or false, depending on desired behavior for empty sequences
    }
    sequence.iter().all(|io| io.is_same_type_as(&sequence[0]))
}
```

This will prevent the function from panicking and provide a predictable response when the sequence is empty.

Status

The Triton VM team has argued that no out-of-bounds access will realistically occur and has also added a test to ensure this. We acknowledge their point.

Verification

Resolved.

Issue E: Integer Overflow Error - Saturated Mul instead of *

Location

Stark

Synopsis

An integer overflow issue occurs in the expression `trace_domain_generator * out_of_domain_point_curr_row`. If the result of the multiplication exceeds the maximum allowable value for the integer type, an overflow will occur, potentially leading to undefined behavior. The issue can be resolved by using `saturated_mul()`, which ensures that if an overflow would occur, the result is set to the maximum possible value for that type.

Impact

If the overflow is not handled, the computation of out_of_domain_point_next_row may result in incorrect values, which would lead to incorrect execution of the cryptographic protocol. This could compromise the integrity of the proof generation process, potentially leading to incorrect or unverifiable proofs. In a worst-case scenario, this could be exploited to bypass security checks or corrupt the proof system.

Feasibility

The fix is highly feasible and requires minimal code changes. By replacing the simple multiplication operator `(*)` with a saturated multiplication method (such as

`saturating_mul()` in Rust), the integer overflow will be prevented. This modification can be applied locally without affecting other parts of the codebase.

Technical Details

The overflow occurs in the following line:

```
let out_of_domain_point_next_row
= trace_domain_generator * out_of_domain_point_curr_row;
```

If the multiplication of trace_domain_generator and out_of_domain_point_curr_row exceeds the type's maximum limit (e.g., u32 or u64), the result wraps around, causing an overflow.

To resolve this, Rust's `saturating_mul()` method can be used, which ensures that the result will stay within the bounds of the type, saturating at the maximum value instead of wrapping around.

Example of the corrected line:

```
let out_of_domain_point_next_row
= trace_domain_generator.saturating_mul(out_of_domain_point_curr_row);
```

Remediation

Replace the standard multiplication operation with Saturated Mul by using `saturating_mul()` in Rust. This ensures that the multiplication will not overflow, safeguarding the integrity of the proof generation process.

Status

The Triton VM team has argued that multiplication only occurs on Field elements. We totally acknowledge their claim.

Verification

Resolved.

Issue F: Use debug macro or propagate error upwards instead of assert statements

Location

- Stark
- Jump Stack
- RAM
- Arithmetic Domain

- Hash Table
- VM

Synopsis

The code currently uses assert! and assert_ne! macros to enforce conditions during runtime. These assertions are used to ensure that the processor table has at least one row and that the FRI expansion factor is greater than one. In production code, assertions like these can cause abrupt termination (panic) of the program, which is undesirable for robust applications. Instead, proper error handling should be implemented by either propagating the errors upward or using the debug_assert! macro for less critical, debug-only checks.

Impact

- Current Behavior: When the conditions in the assertions fail, the program will panic and terminate unexpectedly. This could lead to data loss or service unavailability, especially in environments where uninterrupted service is critical (e.g., cryptographic proving systems or blockchain nodes).
- Proposed Change: Using Result for error handling will allow the program to gracefully handle these errors and propagate them upwards, providing the caller an opportunity to handle the failure.
- Impact on Security and Stability: Replacing assertions with proper error handling ensures the system remains stable under unexpected conditions and improves the overall resilience of the application.

Feasibility

- The replacement of assertions with error handling is straightforward. Both cases can use the Result type to propagate errors upwards. The assert! statements can be replaced with conditional checks that return an error.
- For the development and debugging phase, debug_assert! can still be used to catch potential errors during testing without affecting production stability.
- Rust has built-in mechanisms for error propagation (e.g., the ? operator and Result types), making it feasible to implement without significant refactoring.

Technical Details

We describe the technical details with respect to two out of the total six locations that are mentioned above:

Original Code:

- Assertion 1:

```
assert!(table_len > 0,
    "Processor Table must have at least 1 row.");
```

- Assertion 2:

```
assert_ne!(0, log2_of_fri_expansion_factor,
    "FRI expansion factor must be greater than one.");
```

Proposed Code:

- Assertion 1 Replacement:

```
if table_len == 0 {
    return Err(ProvingError::InvalidTableLength
        ("Processor Table must have at least 1 row."));
}
```

This propagates an error if the table length is invalid, allowing upstream logic to handle it instead of panicking.

- Assertion 2 Replacement:

```
if log2_of_fri_expansion_factor == 0 {
    return Err(ProvingError::InvalidFRIExpansionFactor
        ("FRI expansion factor must be greater than one."));
}
```

This checks the condition and propagates an error if the expansion factor is invalid.

- Alternatively, during debugging, debug_assert! can be used:

```
debug_assert!(table_len > 0, "Processor Table must have at least 1 row.");
```

Remediation

- Replace Assertions: Replace the use of assert! and assert_ne! macros with error propagation via Result types, ensuring that errors are reported back to the caller for proper handling.
- Use Debug Assertions: Where applicable, use debug_assert! to catch potential errors during testing or debugging phases, without affecting production performance or stability.
- Documentation: Document the potential errors returned by these functions, making it clear to developers that certain invalid states (e.g., zero-length tables or invalid FRI expansion factors) will return errors.

Status

The Triton VM team has totally agreed to the issue that we raised here.

Verification

Unresolved.

Issue G: Cascade Table and Lookup Table Multiplicities

Location

Algebraic Execution Trace

Synopsis

The current implementation lacks adequate handling and validation of multiplicities for the Cascade and Lookup tables. This can lead to inaccuracies in tracking the number of times entries in these tables are accessed, potentially compromising the integrity of the cryptographic proofs generated.

Impact

Inaccurate multiplicity tracking can result in:

- Faulty cryptographic proofs that may be accepted as valid despite being based on incorrect data.
- Increased vulnerability to attacks exploiting these inaccuracies, potentially leading to compromised security.
- Difficulty in debugging and verifying the correctness of operations due to inconsistent state information.

Feasibility

The proposed fixes for the multiplicity handling are technically feasible, as they involve implementing additional checks and validations during the entry updates. Modifications can be made with minimal disruption to existing functionality, and it is straightforward to integrate these improvements into the current system architecture.

Technical Details

The implementation currently increases multiplicities based on the observed state elements during hash operations. However, it does not validate whether the updates align with expected counts. This could lead to incorrect multiplicities being recorded in the cascade_table_lookup_multiplicities and lookup_table_lookup_multiplicities. The logic should ensure that multiplicities are updated in a consistent manner, particularly when entries are added or modified.

Remediation

In order to mitigate this issue:

- Introduce validation checks when increasing multiplicities to ensure that they accurately reflect the number of lookups performed.
- Implement unit tests to verify that the multiplicities reflect the expected counts after various operations are executed.
- Document and review the logic surrounding multiplicity updates to ensure clarity and correctness in the implementation, providing a basis for future audits and assessments.

Status

The Triton VM team has again acknowledged our claim and said that they owuld take a closer look at the code and perhaps add unit tests and make the code cleaner to resolved the issue.

Verification

Resolved.

Issue H: Potential Inefficiencies in Iterations

Location

- Cross Table Argument (1)
- Cross Table Argument (2)
- Cross Table Argument (3)

Synopsis

The compute_terminal methods utilize .iter().map() followed by .fold() to process the symbols array. This chaining can lead to inefficiencies, especially when the array is large, as it may result in unnecessary intermediate collections.

Impact

In scenarios with a large symbols array, the current implementation may cause increased memory usage and slower performance due to the creation of intermediate structures during iteration. This could affect the overall throughput of the cryptographic computations being performed.

Feasibility

Refactoring the iteration approach is feasible and can be achieved with minimal changes to the existing logic. The impact on the rest of the codebase should be minimal as long as care is taken to ensure the correctness of the new implementation.

Technical Details

The current implementations of the compute_terminal methods are as follows:

- PermArg:

```
symbols
    .iter()
    .map(|&symbol| challenge - symbol)
    .fold(initial, XFieldElement::mul)
```

- EvalArg:

```
symbols.iter().fold(initial, |running_evaluation, &symbol| {
    challenge * running_evaluation + symbol
})
```

- LookupArg:

```
symbols
    .iter()
    .map(|symbol| (challenge - symbol.lift()).inverse())
    .fold(initial, XFieldElement::add)
```

Each of these methods can be refactored to use a single loop to accumulate results, thereby eliminating the intermediate collections.

Remediation

Replace the use of .iter().map().fold() with a single loop construct to accumulate results directly. For example, in PermArg, you could modify the implementation to:

```
let mut result = initial;
for &symbol in symbols {
    result = XFieldElement::mul(result, challenge - symbol);
}
result
```

Similarly, adjust the other methods to avoid intermediate mappings and folding, thus optimizing performance.

Status

The Triton VM team has refuted our claim and we acknowledge their point.

Verification

Resolved.

Suggestion A: Club All Imports into a Nested Format

Location

Affects all three sections: Triton-ISA, Triton-VM, Triton-AIR.

Synopsis

The current import structure across the three sections is fragmented, making the codebase harder to manage and understand. Consolidating imports into a more organized, nested format would improve code readability, maintainability, and reduce potential for import conflicts.

Mitigation

Refactor the import statements into a nested structure, grouping related modules under a common namespace. This will create a cleaner, more intuitive structure, ensuring that imports are logically organized and easier to trace across the codebase.

Status

The Triton VM team disagreed with our suggestion and argued that readability is a matter of taste. They instead pointed us to a different Rust formatting standard which would help them in maintaining the codebase further. While we understand their viewpoint, we would like to agree to disagree here with them and leave the suggestion here.

Verification

Resolved.

Suggestion B: Improve Documentation for Functions

Location

Affects all three sections: Triton-ISA, Triton-VM, Triton-AIR.

Synopsis

Many functions across the codebase lack sufficient documentation, which hampers understanding, especially for new developers or auditors. This affects the clarity of code logic and could lead to misinterpretation or mistakes.

Mitigation

Ensure that all functions are properly documented with clear, concise descriptions of their purpose, parameters, and expected behavior. This will enhance code readability, ease future maintenance, and improve collaboration.

Status

The Triton VM team acknowledged and agreed with our suggestion completely.

Verification

Partially resolved.

Suggestion C: Reduce Code Duplication with a Common Trait or Abstraction

Location

- Triton VM
- Triton AIR Table Column

Synopsis

There is significant code duplication across various functions, with hardcoded elements that could be potentially abstracted out. While not strictly required to use a trait, consolidating these functions into a single abstraction would result in cleaner, more maintainable, and efficient code.

Mitigation

Refactor the duplicated functions into a common trait or abstraction that can be implemented across the relevant areas. This will reduce redundancy, make the code more modular, and enhance overall maintainability without relying on hardcoded logic.

Status

The Triton VM team acknowledged the suggestion but failed to see the problem in vm.rs. We acknowledge their point.

Verification

Partially resolved.

Suggestion D: Provide User-Facing Messages for Unmatched Instructions

Location

Instruction Link

Synopsis

The current implementation does not offer any feedback to users when an instruction does not match any defined arms in the instruction set. This lack of communication may lead to

confusion regarding the system's behavior and can hinder debugging or troubleshooting efforts.

Mitigation

Implement user-facing messages to indicate when an instruction falls outside the matched arms. For example, when an unmatched instruction occurs, log a message such as, "Warning: The instruction provided is not recognized. Please check your input or consult the documentation for valid instructions." This will enhance user experience by providing clarity on system behavior and improve overall communication regarding operational status.

Status

The Triton VM team refuted our claim and said that user facing messages were provided wherever necessary. We agree with their conclusion after taking a closer look.

Verification

Resolved.

Suggestion E: Introduce Constants to a Separate File

Location

Processor Table

Synopsis

The current implementation contains scattered constants and magic numbers across various sections of the code, which can lead to confusion and make maintenance difficult. This practice undermines code readability and increases the risk of errors, as the significance of these values may not be immediately clear.

Mitigation

Refactor the code to move all constants into a separate file named Constants in a suitable module. Define an enum or structured constants to provide meaningful names for these values, which will enhance code clarity and maintainability. This consolidation will help eliminate magic numbers and centralize configuration, making it easier to update and manage constants in the future.

Status

The Triton VM team has accepted this suggestion as a good idea.
We are specifically relating to the readability of the AIR constraints. We feel they should have a placeholder for constants.

Verification

Partially resolved.

Suggestion F: Reduce Number of Clones

Location

Tables in Triton-AIR and Triton-VM. For example:
Cascade Table

Synopsis

The current implementation exhibits a high number of clones, which can lead to unnecessary memory usage and impact performance. Frequent cloning of variables, especially within the constraints, may indicate that the scope of variables is not being managed effectively, potentially leading to redundant computations and inefficiencies.

Mitigation

Refactor the code to minimize the number of clones by introducing proper scoping for variables. Consider using mutable references or other techniques to manage data ownership and lifetimes more effectively. For instance, variables that are cloned multiple times could be defined in a more limited scope or reused where appropriate. This will help optimize memory usage and enhance overall performance while maintaining clarity and functionality in the constraints logic.

Status

The Triton VM will be resolving this but will be on their lower priority in their to-do-list.

Verification

Partially resolved.

Suggestion G: Theoretical Proof of Arithmetization

Location

Arithmetization Link

Synopsis

The current implementation relies on arithmetization methods, but there is a lack of formal theoretical proofs demonstrating that the mathematical foundations and transformations used are sound and valid. Without rigorous validation, the integrity and reliability of the cryptographic functions may be called into question, potentially undermining the system's security.

Mitigation

Conduct a comprehensive theoretical proof of the arithmetization process used in the implementation and document the findings in an academic paper. This paper should detail the mathematical principles involved, the transformations applied, and the conditions under which the arithmetization holds. By providing a solid theoretical foundation, the proof will enhance confidence in the correctness of the cryptographic implementation and serve as a valuable resource for future research and development in this area.

Status

The triton VM team has accepted our suggestion and acknowldged the fact that there should be theoretical rigour behind the protocol mentioned in the specification in the form of a technical/research paper.

Verification

Unresolved.

NB: For some of the issues, we have mentioned few of the exact code locations where the issue is present. We have not provided an exhaustive list of all the locations where the issue could be present simply because the list will be too long. Please check the entire codebase accordingly searching for similar occurrences of the issue and correct all such places.

Our Methodology

Manual Code Review

In our manual review of the code, we examine potential concerns related to code logic, error handling, protocol details, cryptographic implementations, and the handling of random number generators. We also identify areas where implementing more defensive programming practices could help mitigate risks and simplify future audits. While the primary focus is on the code within the audit scope, we also analyze dependencies and their behavior when relevant to the investigation.

Vulnerability Analysis

Our audit process involves manual code analysis, user interface testing, and whitebox penetration testing. We begin by reviewing the project's website to gain a high-level understanding of the software's functionality, followed by discussions with the developers to grasp their vision for the project. After installing and using the software, we explore its user interactions and roles while brainstorming potential threat models and attack surfaces. We review design documents, similar projects, source code dependencies, and open issue tickets to gather additional context beyond the implementation itself. From there, we hypothesize potential vulnerabilities and proceed with our Issue Investigation and Remediation process for each identified concern.

Documenting Results

We employed a conservative and transparent approach to identifying and addressing potential security vulnerabilities. As soon as a possible issue is identified, we create an Issue entry in this document, even before verifying its feasibility or impact. This conservative method ensures that we document suspicions early, even if they are later determined to be non-exploitable. Typically, we first record the suspicion along with any open questions, and then validate the issue through code analysis, live testing, or automated tests. Code analysis is often preliminary, so we aim to provide supporting evidence such as test code, logs, or screenshots to confirm findings. Once confirmed, we assess the practicality of exploiting the issue in a live system.

Suggested Solutions

We seek immediate and comprehensive mitigations that can be implemented in live deployments, and we also provide recommendations for long-term remediation in future releases. These suggestions should be carefully reviewed by the developers and deployment teams, as successful mitigation and remediation are ongoing collaborative efforts following the delivery of our Initial Audit Report and leading up to a verification review.

Before sharing our findings and solutions, we prefer to work closely with your team to identify practical resolutions that can be implemented promptly without significantly disrupting existing plans. While each issue requires a tailored approach, we aim to establish a timeline for resolution that balances user impact with the needs of your project team.

Resolutions and Publishing

After the findings have been fully addressed, we conduct a verification review to ensure that the issues and recommendations have been properly resolved. Once this analysis is complete, we update the report and issue a Final Audit Report, which can be published in its entirety.