Security Audit Report

Anoma Q4 2025: RISC Zero RM & EVM Protocol Adapter

Authors:

Manuel Bravo,

Ivan Gavran,

Aleksandar Stojanovic,

Carlos Rodriguez

Last Revised:

07.11.2025

Contents

Audit Overview

The Project

In October 2025, Anoma engaged with Informal Systems to conduct a security audit of the Anoma Resource Machine (ARM) protocol, its implementation with RISC Zero and the EVM protocol adapter.

The ARM is used to create, compose, and verify transactions. Anoma users submit their intents to the intent gossip network in the form of unbalanced ARM transactions, which are received and processed by solvers that output balanced ARM transactions. Balanced transactions are routed to settlement protocol adapters (such as the EVM protocol adapter for EVM-compatible chains) that validate and coordinate execution of resource machine transactions.

Scope of this report

The audit focused on evaluating the coherence and consistency of the state architecture protocol (for resource machine and shielded resource machine) and the correctness and security properties of the ARM RISC Zero implementation and the EVM protocol adapter.

Audit plan

The audit was conducted between October 13th, 2025 and October 31st, 2025 by the following personnel:

- Manuel Bravo
- Ivan Gavran
- Aleksandar Stojanovic
- Carlos Rodriguez

Conclusions

We praise the Anoma development team for the exceptional quality of their work on both the ARM RISC Zero implementation and the EVM protocol adapter. The codebases demonstrate strong cryptographic expertise, thoughtful architectural decisions, and careful attention to security best practices. The well-structured code and comprehensive edge case handling reflect a mature approach to building complex zero-knowledge proof systems.

Our audit identified 15 findings with no critical or high severity issues. Two medium severity findings were discovered in the initial tag (v0.8.1): the transaction verification function lacked double-spending checks and didn't validate that compliance units properly partition actions. Both were promptly resolved by the team in tag v0.8.2. Four low severity findings were identified: missing validation (before addition) of input points for elliptic curve membership, lack of automatic memory zeroing of sensitive cryptographic data (which could leave private keys and intermediate values exposed in memory after use), the delegation of nonce computation to applications (which increases the risk of nonce reuse), and missing size validation of `LogicVerifierInputs` `app_data`, which could bloat the public output of the aggregation circuit.

Nine informational findings provide recommendations for code quality improvements, including enhanced specification documentation to address discrepancies between specification and implementation, domain separation improvements for authorization signatures and Merkle tree construction, and various refinements to Merkle tree handling and general code quality. The development team's rapid resolution of medium severity issues and engagement throughout the audit process reinforces confidence in the project's strong security posture.

Audit Dashboard

Target Summary

- **Type:** Protocol and implementation
- **Platform:** Rust, Solidity
- **Artifacts:**
  - Specs for resource machine and shielded resource machine.
  - EVM Protocol Adapter (tag v1.0.0-rc.3)
    - All files in `contracts/src` **except** those in the folder `forwarders`
    - Additionally, the `ecAdd` function as used in `Delta.sol` file from the `@elliptic-curve-solidity` library
  - ARM Risc0 (tag v0.8.2)
    - The folder `arm`, but **excluding** the files listed below
      - `rustler_util.rs`
      - `tests.rs`
      - `test_logic.rs`
      - `evm.rs`
      - `hash.rs`
      - `aggregation/sequential.rs`
      - `aggregation/pcd.rs`
    - `arm-circuits/batch-aggregation` folder

Engagement Summary

- **Dates**: October 13th, 2025 → October 31st, 2025
- **Method**: Threat modelling, manual code review, testing

Severity Summary

Finding Severity	Number
Critical	0
High	0
Medium	2
Low	4
Informational	9
Total	**15**

Table 1: Identified Security Findings

System Overview

In the Anoma Protocol, users send information about the desired state change (intent) to solvers. By analyzing the desired outcomes, solvers determine the optimal execution paths, and the resource machine generates transactions with the necessary cryptographic proofs to validate the proposed resource transitions. The protocol adapter receives these fully-formed transactions and serves as the authoritative source of truth for resource state.

Following, we describe in some detail the two components in scope: the ARM risc0 resource machine and the EVM protocol adapter. The definitions and properties in the threat model define more precisely the desired behavior of the components under scope.

`arm`: Anoma Resource Machine

The ARM is used to create, compose, and verify transactions. It is stateless and run by every node that processes transactions. Anoma users submit their intents to the intent gossip network in the form of *unbalanced* ARM transactions with metadata, which are received and processed by solvers that output *balanced* ARM transactions. These transactions are then ordered and finally sent to the executor node, which verifies and executes the transactions in the determined order, updating the global state.

The resource machine generates three distinct types of proofs, each serving a specific purpose in the overall security and privacy model.

- Compliance proofs demonstrate that resource consumption and creation follow the fundamental protocol rules. These proofs verify that consumed resources actually exist in the commitment tree, that the user possesses the correct nullifier keys, that new resource commitments are properly formed, and that the balance changes are correctly computed.
- Logic proofs validate application-specific constraints for each resource involved in the transaction. Unlike compliance proofs, which focus on protocol-level rules, logic proofs ensure that resources behave according to their specific application logic.
- Delta proofs provide the critical guarantee that transactions maintain overall balance and conservation of value. In the abstract design, these proofs demonstrate that the sum of all resource changes across the entire transaction equals a specified expected value. For the current shielded implementation, this expected value is always zero, preventing any net value creation or destruction. The delta proof operates at the transaction level, aggregating the balance impacts of all individual resource transitions and proving that the net effect maintains system invariants according to the expected balance constraint.

Proof Type	Purpose	Scope	What It Validates
Compliance Proof	Protocol-level validation	Compliance unit	“Is this state transition allowed?”
Resource Logic Proof	Application-level validation	Action	“Does this action satisfy the rules specified by this resource?”
Delta Proof	Balance validation	Entire transaction	“Is this transaction balanced?”

Components architecture

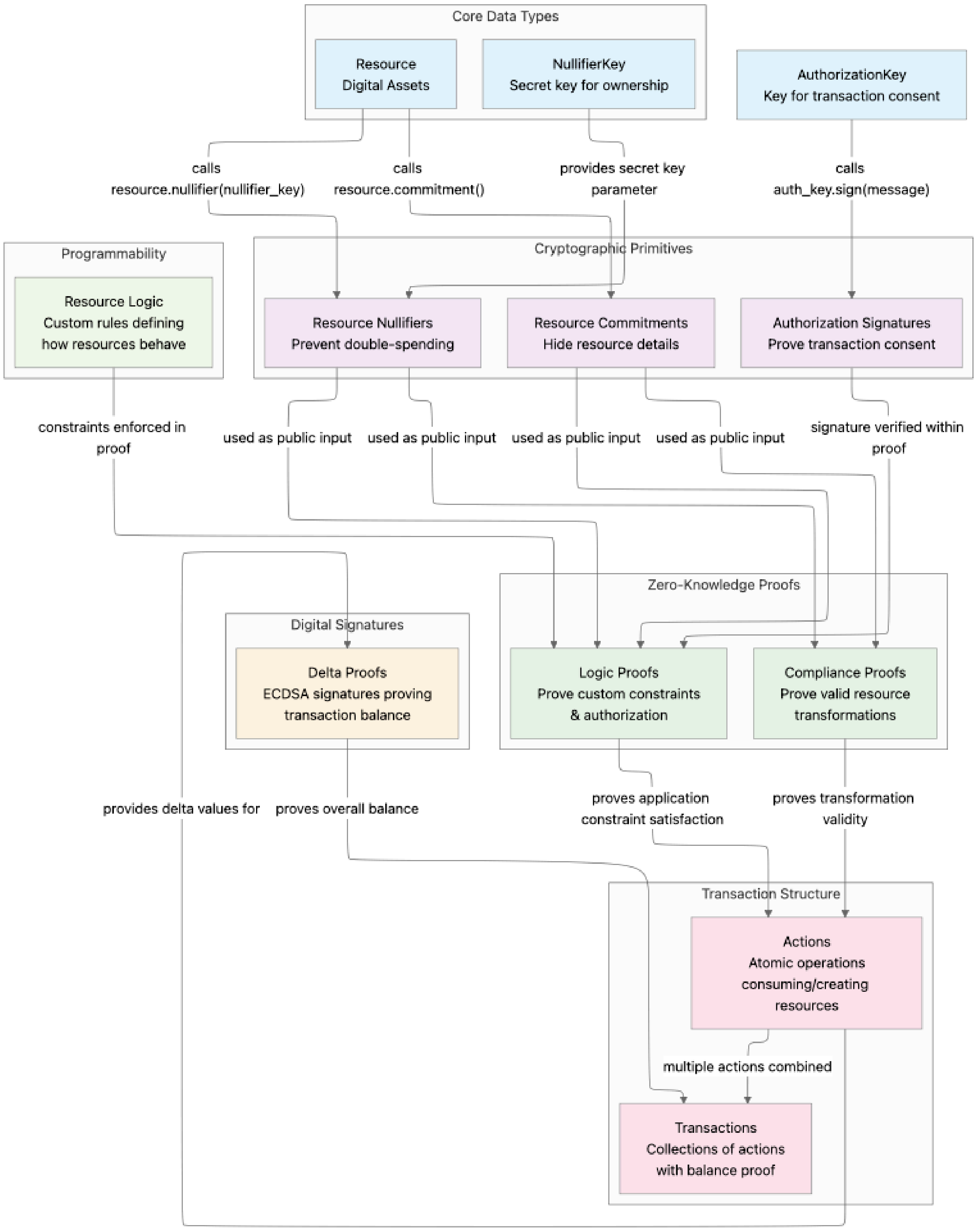

Figure 2: Components diagram

EVM protocol adapter

The EVM protocol adapter serves as a bridge, bringing Anoma's privacy-preserving, intent-centric resource machine model to the Ethereum ecosystem. The primary functionality revolves around executing resource machine transactions on Ethereum. The protocol adapter validates resource transitions through zero-knowledge proofs and coordinates the execution of corresponding operations on other smart contracts.

The adapter maintains two critical state structures: a commitment accumulator implemented as a Merkle tree that stores commitments to all created resources, and a nullifier set that prevents double-spending by tracking consumed resources. This dual-state system ensures both the availability of resources for future consumption and the prevention of replay attacks or double-spending scenarios.

The forwarder system enables interaction with other smart contracts. Forwarders act as trusted intermediaries that receive calls from the protocol adapter, validate them against resource specifications, and forward them to target contracts.

Threat Model

Definitions

A transaction is valid if:

- It is balanced: for any given resource kind, the transaction consumes the same amount that it creates, i.e., it includes a valid delta proof.
- It satisfies the RM constraints, i.e., it includes a valid compliance proof for each compliance unit.
- It satisfies the application constraints on resource consumption and creation, i.e., it includes a valid resource logic proof for each resource.
- All external calls produce the expected output.
- There are no duplicate resources in the transaction, i.e., the transaction attempts to consume or create a resource at most once
- For any non-ephemeral resource that it is consumed in the transaction, its commitment is in the accumulator by the time the transaction is verified
- No consumed resource in the transaction has been consumed by a previously executed transaction.
- For any given created resource, its commitment is not in the accumulator by the time the transaction is verified.

We say that a transaction has been successfully executed when a `TransactionExecuted` event is emitted after the transaction's execution, i.e. it passes all verification checks.

Property ARM-01: The RM implements both the ARM and the shielded specifications correctly (minus the documented discrepancies)

1. It implements all primitive types correctly

    1. The primitive types (Set, Map) do not seem to be relevant (are they outdated documentation?). Even where there is a pointer to where a primitive type is used (such as Map in `Action`'s `logic_verifier_inputs`, this is not reflected in the code, where a vector is used.

    2. For the fixed types:

        1. `NullifierKey` is of type `Vec<u8>`

        2. `Nonce` is indeed a fixed size, 32 bytes, e.g. here

        3. `NullifierKeyCommitment` is indeed a fixed size, a struct structconsisting of only `Digest` (of 32 bytes)

   4. The `Arithmetic` interface does not explicitly exist in the code. The instances of the interface, e.g., `Resource.quantity` field is of type `u128`, which satisfies the requirements of `Arithmetic` and `fixed` type.

2. It implements all main data structures correctly, including their computable components

   1. Commitment: Commitment is calculated as a hash of all fields of `Resource`, with `rand_seed` taken into a separate, inner hash domain (combined with separators and nonce), as calculated in the function `rcm`. To the best of our understanding, this follows good cryptographic practices. There is a non-consequential finding “When reserving commitment bytes, some fields treated as a wrong type”, associated with inspection of the commitment.

   2. Nullifier: Similar to commitment, it hashes the private key, nonce, commitment, and puts the `rand_seed` inside the psi function. The separator differs from the one for calculating commitment.

   3. Kind: The implementation follows the spec, except that it hashes `(logic_ref, label_ref)` , as compared to the spec saying it should hash `(label_ref, logic_ref)`. This does not constitute a problem inside the codebase, but may be confusing for somebody inspecting the values based on the spec.

   4. Delta

      1. compliance unit: implemented as in spec, as a difference between consumed resources and created resources, with added an independent generator point multiplied by a random scalar. One inconsequential difference to the spec: The spec is using old terminology on *input* and *output* resources, instead of consumed and created. Even with those, it has the signs the other way round than in the code (all of that shouldn’t matter since the check is for 0).

      2. action: the implementation follows the spec

      3. transaction: the implementation follows the spec

3. It implements all interfaces faithfully

   1. The spec defines the interface for `ProvingSystem.prove` to accept three arguments, `ProvingKey`, `Instance`, `Witness`. However, the implementation follows RISC Zero’s interfaces and accepts only two arguments `proving_key` and `witness`, while `instance` is returned (since it is a part of the guest computation). It may be useful to note that difference in the spec, even if in a footnote.

   2. The aggregation interface is faithfully implemented (though it goes from a vector of proofs into a proof, slightly differing from the spec).

Threats

- **Some required primitive types are not supported.**

- **The implementation of data structures differs from the specification, e.g, some fields are missing.**

- **The implementation of some methods is incorrect or missing according to the specification.**

Property ARM-02: Only valid transactions pass verification

Threats

- **A transaction passes verification without verifying all compliance proofs.**

  The threat doesn't hold.

  If the aggregation feature is turned off, for each action the function `verify` is called. Inside it, iterating over all compliance units, a `verify` is called for each.

  If the aggregation feature is turned on, there are two cases

  1. Sequential aggregation: in the function `transaction_transcript`, all compliance units are added to `step_instances` by calling `tx.get_batch_cu`, against which `tx.aggregation_proof` is verified.
  2. Batch aggregation: All compliance units are again fetched by calling `tx.get_batch_cu`, against which `tx.aggregation_proof` is verified.

- **A transaction passes verification without verifying the required resource logic proof for each consumed or created resource, e.g., the `get_logic_verifiers` computes the set of tags wrongly or does not guarantee that there is verifier in `logic_verifiers` for each computed tag.**

  The threat holds in the v0.8.1 (tag under review at the beginning of the review). Nevertheless, the issue is fixed in a later version (v0.8.2) as follows.

  First, all tags are extracted into a `tags` vector by iterating over `compliance_instances`, and for each, adding both `consumed_nullifier` and `created_commitment`. The same is done for corresponding logics, which are stored into `logics` variable. Since `compliance_instances` is a vector, we know that `tags` and `logics` have tag↔logic correspondence at each position.

  Then, we iterate over corresponding `(tag, logic)` and make sure that every `tag` has its match among `action.logic_verifier_inputs`, and that the logic to be checked is the correct one (as referred to in the compliance instance). If all those conditions hold, a logic verifier is created from logic input.

  The boolean `is_consumed` is handled correctly: the even positions of `tags` hold consumed tags because of always chaining `(consumed, created)`.

- **The `get_logic_verifiers` returns a `TagNotFound` for a tag that exists.**

  The threat doesn't hold.

As explained in the previous point, for each tag from the compliance unit, it searches `action.logic_verifier_inputs` for matching tags. Another case is when the lengths of `tags` and `self.logic_verifier_inputs` do not match. This matches the requirement that there should not be additional `logic_verifier_inputs`. It is furthermore impossible for an action to have multiple same tags because of the explicit check for duplicate nullifiers, and chaining of commitments and nullifiers in the proof.

- **A transaction passes verification without verifying the delta proof, e.g., an unbalanced transaction is executed successfully.**

  The threat doesn't hold.

  Whenever a transaction `tx` is verified, a delta proof `tx.delta_proof` is verified against `proof` and `instance`.

- **An unbalance transaction passes the delta verification, e.g., there is a bug in the delta's proving system verify function.**

  The threat doesn't hold.

  The verification checks the proof with respect to the transaction data `message` and the public verifying key `instance` using ECDSA and digest trusted functions. The key question is then how are the parameters calculated.

  1. delta message (`message`): A vector created by iterating over all actions in the transaction and then over all compliance units in each action, when finally the vector is extended by `(nullifier, commitment)`.
  2. verifying key that is supposed to be the overall delta (`instance`): For each action and for each compliance unit in it, the public values `delta_x` and `delta_y` are used to encode the point. All those points are collected in a vector and the delta instance is calculated by summing up all those points. At the point where the delta instance is created (in the proof), it is calculated correctly combining consumed and created kinds and quantities.

- **A transaction passes verification with duplicate resources, e.g., a resource is consumed twice within the transaction.**

  The threat holds in the v0.8.1 (tag under review at the beginning of the review). Nevertheless, the issue is fixed in a later version (v0.8.2) by adding a check for duplicate consumption. The check is based on inserting all seen nullifiers into a hash set and checking if they are being re-inserted.

Property ARM-03: If a transaction is valid, then it passes verification

Threats

- The action tree is computed incorrectly in `get_logic_verifiers`. This may lead to valid transactions failing verification, e.g., the root is wrongly computed.

  The threat doesn't hold.

  The action tree is created from the vector of `tags` (which we previously concluded was calculated correctly). Because of how the structure `MerkleTree` is implemented, the creation of the tree is nothing more than stating that the vector of leaves corresponds to the vector `tags`. Thus, the critical part to examine is the `root()` computation for the Merkle tree implementation. The `root()` function is implemented correctly, with `PADDING_LEAF` matching the `empty_leaf_hash` from the evm protocol adapter.

- A balanced transaction fails delta proof verification because of a bug in the delta prove function.

  The threat doesn't hold.

  Let us inspect all situations in which `delta()` may fail or produce a wrong result:

  1. If `compliance_witness.rcv` is not 32 bytes long: in all constructors, `rcv`s are created by `Scalar::random` or `Scalar::ONE` (used only for tests), which both produce 32 byte long scalars.
  2. If the `Scalar` type cannot be recovered from the byte representation. As above, since they all come from `Scalar`, turning back into `Scalar` always succeeds.
  3. problems with calculating `kind()`: the kind is implemented as specified.
  4. wrong calculation of `delta`: the expression is correct, and when summed over all compliance units it should cancel out matching kinds.
  5. problems converting into `EncodedPoint` and explicit `x` and `y` coordinate: the conversion is straightforward.

- A valid transaction fails verification because a prover does not generate valid proofs, i.e., there is a problem with the `prove` function of a proving system.

  The threat doesn't hold.

  Since the delta proof is checked in the previous point, here we analyze how a proof is created for a compliance unit. The proving happens in functions `prove` and `prove_inner`, that are agnostic about the code to be proven (the `proving_key` is passed as a parameter). Handling of data and calling the underlying RISC Zero function is done correctly.

Property ARM-04: The integration of risc0 follows best practices at the host-side code

1. Instance (what to make public) definitions do not reveal any private data
2. Minimize instance size by using hashes instead of full data whenever possible, and make public only what's strictly necessary for the verifier
3. Verifier must check the correct Image ID
4. RISC Zero padding issues are considered in the definition of the proving systems that use RISC Zero

Threats

- **Instances definitions reveal sensitive data.**

  The threat doesn't hold.

  The instance only includes hashes that are public (`consumed_nullifier`, `consumed_logic_ref`, `consumed_commitment_tree_root`, `created_commitment`, `created_logic_ref`) and computed delta. All these fields public.

- **Instances include unnecessary fields.**

  The threat doesn't hold.

  Hashes are used consistently in the instance. It is necessary to include `consumed_nullifier` and `created_commitment`, and `consumed_commitment_tree_root` to connect to updated state. It is necessary to include references to resource logics and delta to avoid checking resource logics which do not correspond to those that were used in the resource.

- **Verifiers are not using the correct circuit during proof verification.**

  The threat doesn't hold.

  The process of embedding the image is automated: the build script for compliance calls the `embed_methods` function. `Cargo.toml` sets the name of the subdirectory for RISC Zero to check to be "guest". In the `guest` folder, the name of the package is set to be "compliance-guest", and the ELF binary is generated at `compliance-guest.bin`. At the build time, the file `methods.rs` is built that defines `COMPLIANCE_GUEST_ELF` by calling `include_bytes!(compliance-guest.bin)`. This variable is then used for proving (and generating the receipt).

- **The proving systems type definitions do not deal well with risc0 padding.**

  The threat doesn't hold.

  1. For compliance instance, all fields of the struct are digests, of fixed size. The remaining two fields, `delta_x` and `delta_y`, are set to be fixed-size arrays.

2. For logic instance, the structure looks as if there could be padding problems. However, the structure is not directly serialized, but instead the structure is first serialized into a vector.

Property ARM-05: The aggregation-proof guest code follows best practices

1. Execution is deterministic; for example, it uses ordered collections such as `BTreeMap` instead of `HashMap`, and it does not utilize randomness or system time
2. Only commits public data
3. Uses the optimized crates
4. Validates inputs
5. Use constant-time ops for sensitive data to avoid revealing information about private data

Threats

- **There is an iteration where the order matters that it is non-deterministic.**

  The threat does not hold.

  For batch aggregation, the prover and the verifier rely on `get_batch_cu` and `get_batch_lp`. They both are based on iteration over vectors, which is deterministic. Afterwards, the processing continues by:

  1. Iterating over the vector of compliance instances in the exact same way.
  2. Iterating over the vector of LP instances in the exact same way.
  3. Finally, creating the environment by committing the data in the (same) order: compliance instances, `COMPLIANCE_VK`, logic instances, logic keys.

- **The batch aggregation circuit commits sensitive data.**

  The threat does not hold.

  The batch aggregation circuit only commits the data that is pre-processed and public: compliance instances, compliance key, logic instances, logic keys.

- **The batch aggregation circuit uses non-optimized crates unnecessarily.**

  The threat does not hold.

  Optimized crates are used (for compliance check, and in all examples).

- **The batch aggregation circuit does not validate inputs.**

  The threat holds partially.

There is no input validation in the batch aggregation circuit. Since the circuit is only used internally, this does not constitute a problem for the most part. However, there is also no validation of the size provided by (external) applications. Related to this, see the finding "Missing checks on logic app_data size".

Even though size equality of `logic_instances` and `logic_keys` is not checked in the circuit, they are of the same length because they both stem from the same, non-modified, vector `lps` (here and here).

- **The batch aggregation circuit does a comparison involving sensitive data that may return early in some cases.**

  The threat does not hold.

  For individual compliance proofs, this property holds:

  - Commitments are created by adding chunks of the pre-defined size.
  - There is a comparison of the two fields of the fixed size.
  - Calculation of delta combines a sequence of assembling fields of pre-defined size and arithmetic operations on those fields.

  The aggregation circuit then only takes those proofs together, thus not revealing any sensitive data.

Property ARM-06: The encryption module is secure

1. Uses well-vetted algorithms that are resistant to known attacks
2. Implements secure memory handling, e.g., clears sensitive data from memory after use
3. Uses cryptographically secure random number generators
4. Constant-time operations for all security-critical comparisons
5. Performs point validation, e.g., to prevent low-order point attacks

Threats

- **The encryption module uses cryptography libraries that are not secure.**

  The threat doesn't hold.

  The encryption module relies on the `aes-gcm` crate (ref) for AES-256-GCM authenticated encryption. For elliptic curve operations, it uses the `k256` crate (ref), which provides an implementation of `secp256k1`. Both libraries are actively maintained, follow constant-time implementation practices to prevent timing attacks, and are part of the RustCrypto project. The module uses `OsRng` for random number generation, which leverages the operating system's secure random source.

- **The encryption module does not clears sensitive data from memory (zero it) after using it.**

  The threat holds.

  The encryption module does not automatically zero sensitive data from memory. Neither `k256::Scalar` (used for private keys) nor `aes_gcm::Key` (used for AES encryption keys) implement `ZeroizeOnDrop`, meaning they do not automatically zero their memory when dropped. The `SecretKey` (code ref) and `InnerSecretKey` (code ref) wrapper types inherit this lack of protection, requiring manual implementation of both the `Zeroize` trait (providing a `.zeroize()` method) and the `ZeroizeOnDrop` marker trait (which auto-generates a `Drop` implementation that calls `.zeroize()`). Decrypted plaintexts are returned as `Vec<u8>`, which implements `Zeroize`, allowing manual zeroing via `.zeroize()`, but it does not implement `ZeroizeOnDrop`, so plaintexts will not automatically zero unless explicitly wrapped in a newtype with `ZeroizeOnDrop`. Intermediate computation buffers (such as concatenated bytes during key derivation) are `Vec<u8>` and can be manually zeroed with `.zeroize()`, but this must be done explicitly at each call site. We have documented a recommendation in finding "No automatic memory zeroing of sensitive cryptographic data".

- **The encryption module does not use secure random generators.**

  The threat doesn't hold.

  The encryption module correctly uses `OsRng`, which is a cryptographically secure random number generator appropriate for generating private keys and other cryptographic secrets. `OsRng` provides high-quality entropy from the operating system's secure random source.

- **The encryption module does a comparison involving sensitive data that may return early in some cases.**

  The threat doesn't hold.

  The encryption module does not perform any variable-time comparisons on sensitive cryptographic data.

- **The encryption module does not perform point validation, i.e., it does not prevent low-order point attacks.**

  The threat holds.

  The encryption module relies on `k256`'s point validation during deserialization, which validates that all points lie on the `secp256k1` curve and rejects invalid curve points. However, `k256` explicitly accepts the identity point during deserialization, and the encryption module provides no additional validation to reject it before use in ECDH operations. While this does not represent a cryptographic vulnerability in the traditional sense (as it requires user code to mistakenly provide the identity point), it creates a defense-in-depth gap that could lead to predictable shared secrets and complete privacy breaches if programming errors occur in application code. We have documented a recommendation in finding "Missing identity point validation in ECDH key exchange" to add identity point validation.

Property ARM-07: The authorization module is secure

1. Correct usage of the `k256` crate
2. Serialization logic, e.g., byte conversions, input validation
3. Errors don't leak sensitive information

Threats

- **There is a problem in the usage of the `k256` crate that makes the authorization module not secure**.

  The threat doesn't hold.

  The usage of the `k256` crate follows standard patterns correctly. The implementation uses `SigningKey::random(&mut OsRng)` (code ref) for secure key generation with a cryptographically secure random number generator, calls the standard `sign()` (code ref) method, properly derives verification keys via `verifying_key()` (code ref), and validates signatures through the standard `verify()` interface (code ref) with appropriate error handling. However, there is a design-level concern not specific to `k256` usage: the module signs raw messages without application-specific domain separation or context binding, which could theoretically allow signature reuse across different protocol contexts if the same key signs the same message bytes in different scenarios.

- **There is a bug in the serialization logic.**

  The threat doesn't hold.

  The serialization logic in the authorization module is correctly implemented and does not contain bugs. The module employs three serialization approaches: `AuthorizationSigningKey` serializes to a fixed 32-byte array (code ref) and properly validates through `k256`'s `SigningKey::from_bytes()` on deserialization (code ref); `AuthorizationVerifyingKey` relies on `k256`'s `AffinePoint` serialization using standard uncompressed point encoding (code ref); and `AuthorizationSignature` utilizes `k256`'s `Signature` serialization with IEEE P1363 format (ref), and `Signature::from_bytes()` on deserialization (code ref). All deserialization paths include proper validation and error handling, rejecting invalid inputs appropriately.

- **Error handling leaks sensitive information.**

  The threat doesn't hold.

  The authorization module's error handling does not leak sensitive information . All error conditions are properly abstracted into generic error types (`ArmError::InvalidSigningKey`, `ArmError::InvalidPublicKey`, `ArmError::InvalidSignature`) that reveal only the failure category without exposing underlying details. The implementation consistently uses the pattern `.map_err(|_| ArmError::...)` to discard `k256`'s detailed error information.

Property ARM-08: The action Merkle tree implementation is safe

1. Collision and pre-image resistance
2. Root computation correctness: the `root` function computes the tree's root by repeatedly pairing and hashing child nodes until a single root hash remains.
3. Given a Merkle tree `tree` and a digest `leaf`, the Merkle tree implementation guarantees that:
    1. If `leaf` is not in `tree`, then the `generate_path` function returns `None`.
    2. If `leaf` is in `tree`, the `generate_path` function returns a Merkle path `path`
    3. A verifier can verify the inclusion of `leaf` by providing the root from `tree` computed from the function `root` and the Merkle path generated by the `generate_path`.

Threats

- **The Merkle tree implementation does not rely on collision-resistant hash functions, i.e., it is feasible to find two inputs that hash to the same output.**

  The threat doesn't hold.

  The Merkle tree uses the function `hash_two()` (code ref), which relies on an implementation of SHA-256 from the `risc0/risc0` repository.

- **The Merkle tree implementation is not second image resistant.**

  The threat doesn't hold.

  The current protocol implementation makes second presage attacks not practically exploitable because leaf of the Merkle tree are cryptographic hashes (resource commitments and nullifiers). However, the Merkle tree implementation itself does not provide protection to second presage attacks and does not follow standard recommended security practices to prevent it. Even though the team was aware of this issue, we have documented a recommendation in finding "Lack of domain separation in Merkle tree implementation".

- **The `root` function computes a tree's root wrongly.**

  The threat doesn't hold.

  The `MerkleTree::root()` function (code ref) implements the standard bottom-up Merkle tree construction algorithm correctly. It pads the leaf array to the next power of two using a consistent padding value, then iteratively hashes adjacent pairs of nodes until a single root hash remains. The algorithm correctly maintains tree structure with proper left-right ordering during hash operations. However, the function has two edge case issues: it can panic when operating on an empty tree (accessing `cur_layer[0]` on an empty vector after resize) and theoretically when `next_power_of_two()`

overflows on extremely large trees (though this requires more memory than can be addressed). We have documented recommendation for both edge cases in findings “Invalid root and Merkle path for empty action trees” and “Miscellaneous code improvements”.

- **Given a tree and a data element that it is a leaf in the tree, the `generate_path` function may return `None`.**

  The threat doesn’t hold.

  The `MerkleTree::generate_path()` function (code ref) will always successfully return a path for a leaf that exists in `self.leaves`. The function searches for the leaf after padding the cloned leaf array, and since padding only appends to the end, any leaf originally in the tree remains findable at its original position. The search is performed via equality comparison, and if the leaf exists, `position()` returns `Some(index)`, causing the function to generate and return `Ok(MerklePath)` rather than `Err(ArmError::InvalidLeaf)`. However, if the tree contains duplicate leaf values, `generate_path()` will always return the path for the first occurrence since `position()` finds the first matching element, with no mechanism to specify which duplicate to target.

- **Given a tree and a data element that it is not leaf in the tree, the `generate_path` function may return a valid Merkle path, i.e., can be used to verify that the data element is in the tree using the `root` method of `MerklePath`.**

  The threat holds.

  The `MerkleTree::generate_path()` function (code ref) can return valid paths for elements that were never inserted into the tree in two scenarios. First, if the tree is empty, calling `generate_path(&PADDING_LEAF)` succeeds because the padded tree becomes `[PADDING_LEAF]`, and the function finds and generates a path for it even though no leaf was ever inserted. Second, for trees with non-power-of-two sizes, the function pads with `PADDING_LEAF` to the next power of two, so `generate_path(&PADDING_LEAF)` will succeed if the padding positions are searched, returning a path that verifies against the tree root despite `PADDING_LEAF` never being an actual inserted leaf. This allows false positives where non-member elements (particularly the padding value) can generate valid authentication paths. We have documented a recommendation to fix this issue in finding “Invalid root and Merkle path for empty action trees”.

- **Given a tree, a data element that it is a leaf in the tree and an invalid Merkle path, the `root` method of `MerklePath` returns the tree’s root.**

  The threat doesn’t hold.

  The `MerklePath::root()` function (code ref) deterministically computes a hash by starting with the leaf value and iteratively hashing it with each sibling in the path according to the specified order. For an invalid path (one with incorrect sibling values, wrong ordering flags, or incorrect depth) to produce the tree’s actual root when given a valid leaf, a hash collision would need to occur where different inputs to the hash function produce the same output. With SHA-256’s collision resistance, the probability of an invalid path accidentally producing the correct root is negligibly small.

- **Given a tree, a data element that it is a leaf in the tree and a valid Merkle path, the `root` method of `MerklePath` returns root that it is different from the tree's root.**

  The threat doesn't hold.

  The `MerkleTree::generate_path()` and `MerklePath::root()` functions use algorithms that guarantee reconstructing the same root. When `generate_path()` builds a path, it records each sibling hash along with a boolean flag indicating whether the sibling is on the left (`is_sibling_left = position % 2 != 0`). The `root()` method then applies these siblings in the same order: if `is_sibling_left` is false (even position, sibling on right), it computes `hash(current, sibling)`; if true (odd position, sibling on left), it computes `hash(sibling, current)`. Both functions apply the same padding strategy (resize to next power of two with `PADDING_LEAF`), ensuring consistent tree structures. The hash operations are deterministic (same inputs always produce same output), and the path construction records exactly the information needed for root reconstruction. As long as the path is generated from the same tree state and applied to the correct leaf value, the reconstructed root will match the tree's root.

Property EPA-01: The adapter implements the protocol adapter specification correctly

1. **It implements all primitive types correctly** - True except for few types that were not implemented / explicitly used
   1. **FixedSize**
      1. Requiring: `bit_size: Int`, `new(Arg) T`, `equal(T, T) Bool`
         1. Solidity uses fixed-size types: `bytes32` , `uint128` , `bool`
         2. Constructor/derivation is implicit in Solidity type system
         3. Equality is built-in to Solidity (`==` operator)
   2. **Arithmetic**
      1. Extends FixedSize
      2. Adds operations: `add(T, T) T`, `sub(T, T) T`
      3. Used for: `Quantity`, `Delta`
         1. Implemented via `uint128` for quantities (code ref)
         2. Addition/subtraction supported via Solidity's arithmetic operators
         3. Delta uses elliptic curve point addition (code ref)
   3. **Hash**

1. FixedSize with binding (if the input value of type `Arg` changed, the output value would change as well) and collision-resistant derivation function
2. Used as base for all cryptographic hash types
    1. All hash types implemented as `bytes32` (SHA256 output size)
    2. SHA256 used throughout (code ref)

4. **Set**
    1. Unordered data structure containing only distinct elements
    2. Required operations: `new()`, `new(List)`, `size()`, `contains()`, `insert()`, `union()`, `intersection()`, `difference()`, `disjointUnion()`
    3. Element uniqueness guaranteed
    4. **Ordered set -** Set that preserves insertion order
        1. Uses OpenZeppelin's `EnumerableSet.Bytes32Set` (code ref, code ref)
5. **Map**
    1. **Not explicitly used** as a primitive type in the Solidity implementation
6. Specific **primitive** types
    - **Nonce** (code ref)
        - FixedSize
        - Guarantees resource uniqueness
        - `bytes32`
    - **RandSeed** (code ref)
        - FixedSize
        - Randomness seed for cryptographic operations
        - `bytes32`
    - **NullifierKeyCommitment** (code ref)
        - FixedSize
        - Commitment to the nullifier key
        - `bytes32`

- **Quantity** (code ref)
  - Arithmetic
  - Resource amount
  - `uint128`
- **DeltaHash** (code ref)
  - Arithmetic + Hash
  - Additively homomorphic delta value
  - `uint256` (elliptic curve point x,y coordinates)
- **PS.VerifyingKey** (code ref, code ref)
  - Hash
  - RISC Zero proof system verifying key (ImageID)
  - `bytes32`
- **LabelHash** (code ref)
  - Hash
  - Hash of resource label
  - `bytes32`
- **ValueHash** (code ref)
  - Hash
  - Hash of resource value
  - `bytes32`
- **Commitment** (code ref)
  - Hash
  - Resource commitment
  - `bytes32`
- **Nullifier** (code ref)
  - Hash
  - Resource nullifier

    - `bytes32`
- **LogicVKOuterHash** (code ref, code ref)
    - Hash
    - Logic verifying key (logicRef)
    - `bytes32`
- **MerkleTreeNodeHash** (code ref)
    - Hash
    - Intermediate Merkle tree node hashes (calculated during MerkleTree operations)
    - `bytes32`
- **Types computed off-chain: Nullifier key, Kind**
- **Not implemented: Balance, LogicVKCompact, AppDataValueHash** (couldn't find explicit definition)

2. **It implements all main data structures correctly, including their computable components**
    1. **Resource**
        1. The Resource structure must contain:
            1. `logicRef`: Hash (verifying key of logic circuit)
            2. `labelRef`: Commitment (SHA256(label))
            3. `valueRef`: Commitment (SHA256(value))
            4. `quantity`: u128
            5. `isEphemeral`: Bool
            6. `nonce`: Hash
            7. `nullifierKeyCommitment`: Hash (SHA256(nk))
            8. `randSeed`: Hash
        2. All required fields are present in implementation (code ref)
        3. The specification defines **computable components and they are computed off-chain** in RISC Zero circuits :
            1. **kind**: SHA256(logicRef || labelRef)

2. **psi**: SHA256(“RISC0_ExpandSeed” || 0 || randSeed ‖ nonce)
3. **rcm**: SHA256(“RISC0_ExpandSeed” || 1 || randSeed ‖ nonce)
4. **cm (commitment)**: SHA256(logicRef, labelRef, valueRef, quantity, isEphemeral, nonce, nullifierKeyCommitment, rcm)
5. **nf (nullifier)**: SHA256(nk, nonce, psi, cm)

2. **Compliance unit**
   1. A compliance unit must contain:
      1. One consumed resource (with nullifier, logicRef, commitmentTreeRoot)
      2. One created resource (with commitment, logicRef)
      3. Nonce of created resource equals nullifier of consumed resource
      4. Delta value (elliptic curve point)
   2. Implementation (code ref) contains all the required components
3. **Action**
   1. It must contain:
      1. `logicVerifierInputs`: Map from resource tags to logic verification data
         - Each entry contains: `verifyingKey`, `applicationData`, `proof`
      2. `complianceUnits`: List of compliance units (one per consumed/created resource pair)
      3. `actionTreeSize`: Integer determining action tree depth
   2. Implementation (code ref) uses arrays instead of map:
      1. `logicVerifierInputs[]`: Logic.VerifierInput array
         - Logic.VerifierInput structure (code ref) contains required fields
      2. `complianceVerifierInputs[]`: Compliance.VerifierInput array
      3. `actionTreeSize` computed
4. **Transaction**
   1. It must contain:
      1. Actions array
      2. Delta proof (mandatory)

            3. Aggregation proof (optional)
            4. expectedBalance
        2. Implementation (code ref) contains all required fields except `exceptedBalance` which is currently not in scope of this adapter implementation :
            1. `actions[]`: array of actions
            2. `deltaProof`: balance verification proof
            3. `aggregationProof`: optional recursive proof (can be empty bytes)
        3. Optional aggregation is correctly handled:
            1. If `aggregationProof.length > 0`: compliance and logic instances aggregated (code ref)
            2. If absent: each compliance and logic proof verified individually (code ref, code ref)
    5. **Global State**
        1. The replicated global state must contain:
            1. **Commitment Accumulator**: Merkle tree containing commitments of all created resources
            2. **Nullifier Set**: Set containing nullifiers of all consumed resources
        2. **Commitment Tree** implementation (code ref):
            - Incremental Merkle tree (code ref)
            - Historical roots stored in `_roots` (EnumerableSet) for verification of past commitments
            - Variable depth (grows dynamically as required by spec)
            - Uses SHA256 for hashing
            - Provides `_addCommitment()` (code ref) and `verifyMerkleProof()` (code ref)
        3. **Nullifier Set** implementation (code ref):
            - Uses OpenZeppelin's `EnumerableSet.Bytes32Set`
            - Provides `_addNullifier()` with duplicate prevention (code ref)
            - Reverts on duplicate nullifiers (double-spend prevention)
3. **It implements all interfaces faithfully**
    1. **Transaction execution** (code ref)

2. **Logic proof verification** (code ref)
3. **Compliance proof verification** (code ref)
4. **Delta proof verification** (code ref)
5. **Aggregation proof verification** (code ref)
6. **State management**
7. **Additional methods beyond core specification:**
    1. Emergency stop mechanism (code ref)
    2. Application data blob emission (code ref)

Threats

- **Some required primitive type is not supported.**
    - Three primitive types mentioned in specification are not explicitly implemented: `Balance`, `LogicVKCompact`, `AppDataValueHash`.
    - `Map` type not used (arrays used instead).
    - All critical primitive types for transaction processing are supported.
- **The implementation of data structures differs from the specification, e.g, some fields are missing.**
    - `Action`: Uses arrays instead of Map for `logicVerifierInputs`
    - `Transaction`: Missing `expectedBalance` field (intentionally out of scope for this implementation)
    - All other data structures match specification completely with all required fields present
- **The implementation of some methods is incorrect or missing according to the specification.**
    - Threat doesn't hold. All core methods implemented correctly.

Property EPA-02: Type definitions and methods of the proving systems match those defined in the arm-risc0

Threats

- **The proving systems data structures, instances and methods do not match the definitions in `arm-risc0`.**

  Threat doesn't hold.

  All proving system data structures, instances and methods match `arm-risc0` definitions:

  - Logic Instance (`arm-risc0`, `evm-protocol-adapter`)
  - Compliance Instance (`arm-risc0`, `evm-protocol-adapter`)
  - Aggregation Instance (`arm-risc0`, `evm-protocol-adapter`)
  - Delta Proof (`arm-risc0`, `evm-protocol-adapter`)

- **Wrong image IDs in the risc0 proving systems are used for verification.**

  Threat doesn't hold.

  Image IDs match between `evm-protocol-adapter` and `arm-risc0` v0.8.1:

  - Compliance VK: `0x5e12ee6aeaa338ff28ae3efe563704e0439b358c75895e98fee8feb2acac697b` (`arm-risc0` v0.8.1, `evm-protocol-adapter`)
  - Aggregation VK: `0x5fa611e4c78bd728efdd65f356ac9a89b3beab18e5ecd414ea9ef6750595279a` (`arm-risc0` v0.8.1, `evm-protocol-adapter`)

  The `arm-risc0` repository has since moved to v0.8.2 with updated image IDs. The team confirmed they will update the adapter image IDs after audit completion to avoid tag churn during the audit process.

Property EPA-03: Only valid transactions are executed successfully

Threats

- **A transaction is successfully executed without verifying all compliance proofs. Both cases with and without aggregation proof should be considered.**

Threat doesn't hold.

All compliance proofs are verified in the execution loop (code ref):

- Without aggregation: Each compliance proof verified via RISC Zero (code ref)
- With aggregation: All compliance instances collected (code ref) and verified together in aggregation proof (code ref)

Commitment tree root existence checked for all consumed resources (code ref).

- **A transaction is successfully executed without verifying the required resource logic proof for each consumed or created resource. Both cases with and without aggregation proof should be considered.**

  Threat doesn't hold.

  All logic proofs are verified for both consumed and created resources (code ref):

  - Logic reference verified against compliance unit (code ref)
  - Without aggregation: Each logic proof verified via RISC Zero (code ref)
  - With aggregation: All logic instances collected (code ref) and verified together (code ref)

- **A transaction is successfully executed without verifying the delta proof, e.g., an unbalanced transaction is executed successfully.**

  Threat doesn't hold.

  Delta proof is always verified before state changes are committed (code ref):

  - Transaction delta accumulated from all unit deltas (code ref)
  - Delta proof verification is mandatory (code ref)
  - Verifying key computed from all tags (code ref)

- **An unbalance transaction passes the delta verification, e.g., there is a bug in the delta proving system verify function.**

  Threat doesn't hold.

  Delta verification uses standard ECDSA signature recovery (code ref):

  - Uses OpenZeppelin's `ECDSA.recover` (code ref)
  - Verifying key = Keccak256(all tags) binds proof to specific transaction (code ref)
  - Recovered address must match delta point converted to address (code ref)

- **A transaction is successfully executed without executing an external call or without checking its output correctly.**

  Threat doesn't hold.

  All external calls are executed and outputs verified (code ref):

  - External calls executed for each externalPayload blob (code ref)
  - Output checked against expected output via Keccak256 hash comparison (code ref)
  - Reverts if outputs don't match (code ref)

- **A transaction is successfully executed with duplicate resources, e.g., a resource is consumed twice within the transaction.**

  Threat doesn't hold.

  Duplicate consumption within transaction prevented by nullifier set (code ref):

  - `_addNullifier` called for each consumed resource (code ref)
  - EnumerableSet.add returns false for duplicates (code ref)
  - Transaction reverts on duplicate nullifier (code ref)

- **A transaction is successfully executed while consuming a resource that was consumed by a previously executed transaction (double-spend issue)**

  Threat doesn't hold.

  Double-spend prevented by persistent nullifier set (code ref):

  - Nullifier set is inside contract storage (persists across transactions)
  - `_addNullifier` reverts if nullifier already exists (code ref)
  - Nullifiers are never removed from set (append-only)

- **A transaction is successfully executed while consuming a non-ephemeral resource that has not been created before.**

  Threat doesn't hold.

  Compliance proof verifies commitment tree root existence (code ref):

  - Consumed resource references historical commitment tree root
  - Root existence check ensures resource was previously created

- **A transaction is successfully executed while creating a resource that was created by a previously executed transaction.**

Threat doesn't hold.

Commitment uniqueness guaranteed by compliance circuit assuming nullifier uniqueness (code ref):

- Compliance circuit enforces: created resource nonce = consumed resource nullifier
- Since nullifiers are unique, and nonce determines commitment, commitments are unique
- `_addCommitment` allows duplicate leaves but uniqueness guaranteed by circuit logic

Property EPA-04: Assume that the risc0 verifier is available and the adapter has not been paused. If a transaction is valid, it is then executed successfully.

Threats

- **The action tree is computed incorrectly. This may lead to valid transactions failing verification, e.g., the root is wrongly computed.**

  Threat doesn't hold.

  Action tree computation is correct. Tag collection (code ref) iterates through all compliance units and collects consumed nullifier at `index = i * 2` and created commitment at `index = i * 2 + 1`, maintaining correct order. Tree depth computation (code ref) correctly calculates minimal depth where `2^depth >= leavesCount` by doubling capacity until it fits all leaves. Root computation (code ref) uses standard bottom-up Merkle tree construction, padding leaves with `EMPTY_HASH` to fill capacity and building upward with `nodes[i] = hash(nodes[2*i], nodes[2*i+1])`. Tag count validation (code ref) ensures `logicVerifierInputs.length == complianceUnitCount * 2` to prevent mismatched counts.

- **There is a bug in the `VerifierInput` lookup function in `Logic.sol` such that it returns a `TagNotFound` error when the tag exists.**

  Threat doesn't hold.

  Lookup function (code ref) is correct. It performs a linear search through all elements, returning immediately upon finding a matching tag. The function only reverts with `TagNotFound` if the tag is not found after checking all elements. There are no off-by-one errors, early termination bugs, or hash collision issues since it uses direct `bytes32` equality comparison.

Property EPA-05: The emergency stop feature is implemented correctly

1. It can only be initiated by authorized users

   The `emergencyStop` function (code ref) uses the `onlyOwner` modifier, restricting access to the contract owner only. The function also requires `whenNotPaused` to prevent repeated calls after emergency stop is triggered.

2. It cannot be reverted, i.e., once the contract is stopped, its operation cannot be resumed.

   The contract inherits from OpenZeppelin's `Pausable` (code ref) which provides an internal `_unpause()` function. However, the ProtocolAdapter does not expose any public or external function that calls `_unpause()`. There is no `resume`, `unpause`, or similar function in the contract, making the emergency stop effectively irreversible.

3. No transaction is executed if the contract is in emergency stop.

   The `execute` function (code ref) uses the `whenNotPaused` modifier, which reverts if the contract is paused. This is the only external entry point for transaction execution, ensuring no transactions can be executed while emergency stopped.

Note: External RISC Zero pause is included in `isEmergencyStopped` but not in the `whenNotPaused` check. Verification calls will still revert while the verifier is paused, preventing successful execution.

Threats

Based on analysis above, none of the listed threat's hold.

- **Unauthorized users can stop the protocol adapter**
- **An unauthorized user can resume the protocol adapter**
- **Transactions are executed while the contract is paused.**

Property EPA-06: Given a transaction that passes verification, the commitments of the transaction's created resources are added to the commitment accumulator.

Threats

- **Given a successfully executed transaction, one ore more created resources are not added to the accumulator.**

  Threat doesn't hold.

  All created resources are added to the commitment accumulator through the following execution flow:

  1. **Iteration over all compliance units** (code ref): The execution loop processes every compliance unit in every action, with each compliance unit containing exactly one consumed and one created resource.
  2. **Processing created resource logic** (code ref): For each compliance unit, `_processLogic` is called with `isConsumed: false` for the created resource, passing the commitment from `complianceVerifierInput.instance.created.commitment`.
  3. **Adding commitment to tree** (code ref): Inside `_processLogic`, when `isConsumed` is false, `_addCommitment(tag)` is called where `tag` is the created resource's commitment. This updates `latestCommitmentTreeRoot` in the internal variables.
  4. **Storing final root** (code ref): After all actions are processed and proofs verified, the final commitment tree root (containing all added commitments) is stored via `_addCommitmentTreeRoot(vars.latestCommitmentTreeRoot)`.

  The execution flow guarantees that every created resource commitment is added because: (1) the loop processes all compliance units without skipping, (2) each compliance unit has exactly one created resource that is processed, (3) `_addCommitment` is unconditionally called for created resources, and the transaction only succeeds if all steps complete without reverting. If any commitment fails to be added, the entire transaction reverts.

Property EPA-07: Given a transaction that passes verification, the nullifiers of the transaction's consumed resources are revealed, i.e., added to the nullifier set.

Threats

- **Given a successfully executed transaction, the nullifiers of one or more consumed resources are not revealed.**

  Threat doesn't hold.

  All consumed resources have their nullifiers added to the nullifier set through the following execution flow:

  1. **Iteration over all compliance units** (code ref): The execution loop processes every compliance unit in every action, with each compliance unit containing exactly one consumed and one created resource.
  2. **Processing consumed resource logic** (code ref): For each compliance unit, `_processLogic` is called with `isConsumed: true` for the consumed resource, passing the nullifier from `complianceVerifierInput.instance.consumed.nullifier`.
  3. **Adding nullifier to set** (code ref): Inside `_processLogic`, when `isConsumed` is true, `_addNullifier(tag)` is called where `tag` is the consumed resource's nullifier. The function reverts if the nullifier already exists, preventing double-spends.
  4. **Atomicity guarantee**: The transaction only succeeds if all steps complete without reverting. If any nullifier fails to be added (e.g., duplicate), the entire transaction reverts via `PreExistingNullifier` error (code ref).

  The execution flow guarantees that every consumed resource nullifier is revealed because: (1) the loop processes all compliance units without skipping, (2) each compliance unit has exactly one consumed resource that is processed, (3) `_addNullifier` is unconditionally called for consumed resources, and the nullifier set is persistent contract storage that survives transaction completion.

Property EPA-08: The variable-size Merkle Tree implementation is safe

1. The tree maintains insertion order
2. Depth increases when capacity is reached (not before or later), at that point, capacity is doubled
3. Collision and pre-image resistance

4. Completeness: the ability to verify that a specific data element is a part of a Merkle tree by reconstructing the tree's root hash using only the element's hash and the necessary intermediate hashes (the "proof").
5. Soundness: if a claimed data element is not in the tree, a dishonest prover cannot convince a verifier that it is.

Threats

- **The Merkle tree implementation does not guarantee insertion order.**

  Threat doesn't hold.

  Insertion order is guaranteed by `_nextLeafIndex` counter (code ref). Each `push()` call assigns `index = self._nextLeafIndex++`, incrementing the counter atomically. Leaves are inserted sequentially at positions 0, 1, 2, … without gaps or reordering.

- **The Merkle tree implementation increases the tree's depth before its capacity is reached.**

  Threat doesn't hold.

  Depth increase occurs only after capacity is reached (code ref). The condition `if (self._nextLeafIndex == capacity(self))` executes after incrementing `_nextLeafIndex`, meaning depth increases only when the next insertion would exceed current capacity. The tree never increases depth prematurely.

- **The Merkle tree implementation does not increase the tree's depth once its capacity is reached.**

  Threat doesn't hold.

  See conclusion above.

- **The Merkle tree implementation does not double capacity when it increases its depth.**

  Threat doesn't hold.

  Capacity calculation (code ref) is `capacity = 2^depth`. When depth increases by 1 (from `d` to `d+1`), capacity changes from `2^d` to `2^(d+1) = 2 * 2^d`, exactly doubling the previous capacity.

- **The Merkle tree implementation does not rely on collision-resistant hash functions, i.e., it is feasible to find two inputs that hash to the same output.**

  Threat doesn't hold.

  The implementation uses SHA-256 (code ref) via Solidity's native `sha256()` precompile. SHA-256 provides collision resistance, making collision attacks computationally infeasible with current technology.

- **The Merkle tree implementation is not second preimage resistant**

Threat doesn't hold.

The tree uses SHA-256 for node hashing, which is second preimage resistant under standard assumptions. Leaves are already cryptographic hashes (nullifiers/commitments), so producing a different leaf that collides with an internal-node hash would require a SHA-256 (second-)preimage, which is computationally infeasible.

- **Given a tree, a data element that it is a leaf in the tree and a valid Merkle path, the `processProof` function returns a root that it is different to tree's root.**

  Threat doesn't hold.

  The `processProof` function (code ref) correctly reconstructs the root from a leaf and valid proof. Starting with `computedHash = leaf` , it iterates through siblings, applying `hash(sibling, current)` or `hash(current, sibling)` based on direction bits. This matches the tree construction in `push()` which uses the same hashing logic. A valid proof for an existing leaf must reproduce the tree's root.

- **Given a tree, a data element that it is not leaf in the tree and a Merkle path, the `processProof` function returns the tree's root.**

  Threat doesn't hold.

  For a non-leaf element to produce the tree's root requires finding a Merkle path (siblings and direction bits) such that hashing the non-leaf with those siblings yields the root. Without breaking SHA-256's cryptographic properties, a dishonest prover cannot construct a valid-looking proof for a non-existent leaf.

- **Given a tree, a data element that it is a leaf in the tree and a invalid Merkle path, the `processProof` function returns the tree's root.**

  Threat doesn't hold.

  An invalid Merkle path (wrong siblings or wrong direction bits) will produce a different root. The `processProof` function (code ref) deterministically computes hashes level by level. Changing any sibling or direction bit produces a different hash at that level, which cascades through remaining levels, yielding a different final root. The only way an invalid path could produce the correct root is through a SHA-256 collision (computationally infeasible).

Property EPA-09: Standard Solidity practices are followed

Property holds.

The codebase demonstrates strong adherence to standard Solidity practices across multiple dimensions:

1. Code organization and structure

Follows best practices:

- Clear separation of concerns: interfaces, libraries, state contracts, and main contract (code ref)
- Libraries for reusable logic (SHA256, MerkleTree, TagUtils, RiscZeroUtils)
- Abstract base contracts for forwarders (ForwarderBase, EmergencyMigratableForwarderBase)
- Dedicated proving system libraries (Compliance, Logic, Delta, Aggregation)
- State separation (CommitmentTree, NullifierSet as inherited contracts)

2. Access control and security patterns

Implements industry-standard patterns:

- Uses OpenZeppelin's `Ownable` for admin functions (code ref)
- `Pausable` for emergency stop mechanism (code ref)
- `ReentrancyGuardTransient` for reentrancy protection using transient storage (code ref)
- Appropriate modifier usage: `onlyOwner`, `whenNotPaused`, `nonReentrant`

3. Error handling

Follows modern error patterns:

- Custom errors with parameters (gas-efficient) instead of string revert messages
- Descriptive error names: `TagNotFound`, `PreExistingNullifier`, `DeltaMismatch`, `NonExistingRoot`
- Error parameters provide debugging context

4. Documentation

Comprehensive NatSpec documentation:

- Function documentation with `@notice`, `@param`, `@return`, `@dev`
- Contract-level documentation with `@title`, `@author`, `@notice`
- Security annotations: `@custom:security-contact security@anoma.foundation`
- Explicit slither disable comments where intentional: `// slither-disable-next-line calls-loop`

5. Gas optimization

Demonstrates gas-conscious design:

- Uses transient storage for reentrancy guard (EIP-1153, gas-efficient)
- OpenZeppelin's `Arrays.unsafeAccess` for performance (code ref)
- Memory variables cached to avoid repeated SLOAD: `uint256 treeDepth = depth(self)` (code ref)
- Calldata for read-only parameters: `Transaction calldata transaction`
- Pre-increment where applicable: `++treeDepth` instead of `treeDepth++`
- Packed struct layouts where possible

6. External dependencies

Uses audited, industry-standard libraries:

- OpenZeppelin Contracts (access control, utilities, cryptography)
- RISC Zero Ethereum libraries (proof verification)
- Solady (EfficientHashLib for gas optimization)
- Permit2 integration for token approvals
- No custom implementations of standard patterns

7. Immutability and constants

Proper use of storage modifiers:

- Constants for fixed values: `bytes32 internal constant _VERIFYING_KEY`
- Immutable for constructor-set values: `_TRUSTED_RISC_ZERO_VERIFIER_ROUTER`
- Clear naming convention: underscore prefix for internal/private state

8. Event emission

Comprehensive event logging:

- Events for all state changes: `TransactionExecuted`, `ActionExecuted`, `CommitmentTreeRootAdded`
- Events for external calls: `ForwarderCallExecuted`
- Events for payload data: `ResourcePayload`, `DiscoveryPayload`, `ExternalPayload`, `ApplicationPayload`
- Indexed parameters for efficient filtering where appropriate

9. Testing and verification

Code quality tooling integration:

- Slither static analysis integration (explicit disable comments for known patterns)
- Forge linting with `forge-lint` comments
- Test-friendly architecture with clear separation of concerns

Property EPA-10: The ecAdd function implementation is safe, as well as its usage

1. The implementation maintains group properties of addition:
   - Closure: If P and Q are on two points on the elliptic curve, then $P + Q$ is also on the curve
   - Associativity: The addition operation is associative (i.e., $(P + Q) + R = P + (Q + R)$)
   - Identity element: The point at infinity O acts as the additive identity (i.e., $P + O = O + P = P$)
   - Inverse element: For every point $P = (x, y)$, there exists $-P = (x, -y)$ such that $P + (-P) = O$
   - Commutativity: The order of addition does not matter (i.e., $P + Q = Q + P$)

   The properties are satisfied by fuzz testing. The tests are provided in Appendix "Testing of `ecAdd`".

2. The function correctly implements formulas for addition of two points on the elliptic curve over finite field F_p:
   - Point addition:

     Let $P_1 = (X_1, Y_1, Z_1)$ and $P_2 = (X_2, Y_2, Z_2)$ be two distinct points on the elliptic curve in Jacobian coordinates, their sum $P_3 = P_1 + P_2 = (X_3, Y_3, Z_3)$ is computed through the following sequence of operations:
     - $Z_1^2 \equiv Z_1 \cdot Z_1 \ (mod\ p)$ (ref)
     - $Z_1^3 \equiv Z_1 \cdot Z_1^2 \ (mod\ p)$ (ref)
     - $Z_2^2 \equiv Z_2 \cdot Z_2 \ (mod\ p)$ (ref)
     - $Z_2^3 \equiv Z_2 \cdot Z_2^2 \ (mod\ p)$ (ref)
     - $U_1 \equiv X_1 \cdot Z_2^2 \ (mod\ p)$ (ref)
     - $U_2 \equiv X_2 \cdot Z_1^2 \ (mod\ p)$ (ref)
     - $S_1 \equiv Y_1 \cdot Z_2^3 \ (mod\ p)$ (ref)
     - $S_2 \equiv Y_2 \cdot Z_1^3 \ (mod\ p)$ (ref)
     - $H \equiv U_2 - U_1 \ (mod\ p)$ (ref)

- $R \equiv S_2 - S_1 \ (mod\ p)$ (ref)

If $H \equiv 0 \ (mod\ p)$, the two points have the same x-coordinate. In this case, if $R \equiv 0 \ (mod\ p)$ as well, the points are identical and the point doubling formula should be used instead. (ref)

If $H \equiv 0 \ (mod\ p)$ but $R \not\equiv 0 \ (mod\ p)$, the points are inverses of each other and their sum is the point at infinity.

Assuming $H \not\equiv 0 \ (mod\ p)$, proceed with the addition. Calculate:

- $H^2 \equiv H \cdot H \ (mod\ p)$ (ref)
- $H^3 \equiv H \cdot H^2 \ (mod\ p)$ (ref)
- $T \equiv U_1 \cdot H^2 \ (mod\ p)$
- $X_3 \equiv R^2 - H^3 - 2 \cdot T \ (mod\ p)$ (ref)
- $Y_3 \equiv R \cdot (T - X_3) - S_1 \cdot H^3 \ (mod\ p)$ (ref)
- $Z_3 \equiv Z_1 \cdot Z_2 \cdot H \ (mod\ p)$ (ref)

The function `jacAdd` (ref) correctly implements the formulas.

- Point doubling:

  Let $P = (X, Y, Z)$ be a point on the elliptive curve in Jacobian coordinates, to obtain $2P = (X', Y', Z')$ is computed through the following sequence of operations:

  - $X^2 \equiv X \cdot X \ (mod\ p)$ (ref)
  - $Y^2 \equiv Y \cdot Y \ (mod\ p)$ (ref)
  - $Y^4 \equiv Y^2 \cdot Y^2 \ (mod\ p)$
  - $Z^2 \equiv Z \cdot Z \ (mod\ p)$ (ref)
  - $Z^4 \equiv Z^2 \cdot Z^2 \ (mod\ p)$
  - $S \equiv 4 \cdot X \cdot Y^2 \ (mod\ p)$ (ref)
  - $M \equiv 3 \cdot X^2 + a \cdot Z^4 \ (mod\ p)$ (ref)
  - $X' \equiv M^2 - 2 \cdot S \ (mod\ p)$ (ref)
  - $Y' \equiv M \cdot (S - X') - 8 \cdot Y^4 \ (mod\ p)$ (ref)
  - $Z' \equiv 2 \cdot Y \cdot Z \ (mod\ p)$ (ref)

  where a is the curve parameter from the Weierstrass equation $y^2 \equiv x^3 + ax + b \ (mod\ p)$.

  The function `jacDouble` (ref) correctly implements the formulas.

3. Let $P = (X, Y, Z)$ be a point on the elliptive curve over F_p in Jacobian coordinates, the function correctly converts from Jacobian to affine coordinates $P' = (X', Y')$:

   - Compute Z^{-1} (modular inverse of $Z \ (mod\ p)$) (ref)
   - $Z^{-2} \equiv (Z^{-1})^2 \ (mod\ p)$ (ref)
   - $Z^{-3} \equiv Z^{-2} \cdot Z^{-1} \ (mod\ p)$
   - $X' \equiv X \cdot Z^{-2} \ (mod\ p)$ (ref)
   - $Y' \equiv Y \cdot Z^{-3} \ (mod\ p)$ (ref)

   The function `toAffine` (ref) correctly implements the formulas.

Findings

Name	Type	Severity	Status
The transaction verification function does not check for double-spending	Implementation	2 - Medium	Resolved
The transaction verification function does not check that compliance units partition an action	Implementation	2 - Medium	Resolved
Missing validation of input points to ensure coordinate bounds and curve membership	Implementation	1 - Low	Patched Without Reaudit
No automatic memory zeroing of sensitive cryptographic data	Implementation	1 - Low	Patched Without Reaudit
The encryption module requires nonces to be computed by applications	Design	1 - Low	Patched Without Reaudit
Missing checks of logic app_data size	Implementation	1 - Low	Acknowledged
When reserving commitment bytes, some fields treated as a wrong type	Implementation	0 - Informational	Patched Without Reaudit
Lack of domain separation in authorization signatures	Implementation	0 - Informational	Patched Without Reaudit
Lack of domain separation in Merkle tree implementation	Design	0 - Informational	Acknowledged
Missing identity point validation in ECDH key exchange	Implementation	0 - Informational	Patched Without Reaudit
Invalid root and Merkle path for empty action trees	Implementation	0 - Informational	Patched Without Reaudit

Name	Type	Severity	Status
Empty Merkle paths provide no cryptographic authentication	Implementation	0 - Informational	Patched Without Reaudit
Inefficient Merkle tree depth computation	Implementation	0 - Informational	Patched Without Reaudit
Miscellaneous code improvements	Implementation	0 - Informational	Patched Without Reaudit
Feedback on specification	Documentation	0 - Informational	Acknowledged

Table 3: Identified Security Findings

The transaction verification function does not check for double-spending

Severity Medium **Exploitability** High **Status** Resolved

Type Implementation **Impact** Low

Involved artifacts

- `arm-risc0`: `arm/src/transaction.rs`

Description

The `verify` function (code ref) of `Transaction` in the `arm-risc0` repository does not check that there is no double-spending in a given transaction. This may result in an invalid transaction passing verification.

Problem scenarios

An invalid transaction passing verification may result in a safety violation. For instance, it may result in a transaction being able to consumed twice the same resource.

Given that the `verify` function is currently only used for testing purposes, we consider this as low impact. Nevertheless, it is been discussed with the team that this function may be used by other actors such as Anoma executors in the future. In such cases, the impact (hence the severity of the finding) should increase.

Recommendation

Add a check to the verify that there is no double-spending in a given transaction.

The transaction verification function does not check that compliance units partition an action

Severity Medium **Exploitability** High **Status** Resolved

Type Implementation **Impact** Low

Involved artifacts

- `arm-risc0`: `arm/src/transaction.rs`

Description

The `verify` function (code ref) of `Transaction` in the `arm-risc0` repository does not check that the compliance units partition an action correctly. This may result in an invalid transaction passing verification.

More concretely, the function only checks that for each logic verifier input there is a matching (tag, logic) of a consumed or created resource in a compliance unit of the action. Nevertheless, it does not check that for each (tag, logic) there is a matching logic verifier input that will be verified to assess the validity of the transaction.

Problem scenarios

An invalid transaction passing verification may result in a safety violation. For instance, it may be the case that the transaction passes verification but some resources are created or consumed without verifying their resource logics, i.e., application constraints.

Given that the `verify` function is currently only used for testing purposes, we consider this as low impact. Nevertheless, it is been discussed with the team that this function may be used by other actors such as Anoma executors in the future. In such cases, the impact (hence the severity of the finding) should increase.

Recommendation

Add a check to the `verify` function of the `Transaction` type.

Missing validation of input points to ensure coordinate bounds and curve membership

Severity Low **Exploitability** Low **Status** Patched Without Reaudit

Type Implementation **Impact** Medium

Involved artifacts

- `evm-protocol-adapter`: `contracts/src/libs/proving/Delta.sol`

Description

Consuming applications of `ecAdd` function (code ref) from `witnet/elliptic-curve-solidity` library are recommended to call `isOnCurve()` (code ref) to validate all input points before performing elliptic curve operations, as noted in the library's README.md: "This library **does not check whether the points passed as arguments to the library belong to the curve**. However, the library exposes a method called `isOnCurve` that can be utilized before using the library functions".

This validation serves two critical purposes:

1. Ensures coordinates are properly reduced: `0 < x, y < p`
2. Verifies curve membership: `y² = x³ + ax + b (mod p)`

Problem scenarios

Failure to perform this validation may result in:

- Invalid points being passed to `ecAdd`
- Unpredictable behavior or security vulnerabilities

As the following test shows, adding a point that is not on the curve to the generator point produce a meaningless result:

```solidity
/// @notice COUNTEREXAMPLE: Points NOT on curve produce incorrect results
/// @dev Demonstrates that skipping isOnCurve validation leads to wrong outputs
function test_CounterExample_InvalidPoints_WrongResults() pure public {
  // Using secp256k1 parameters
  uint256 pp = SECP256K1_PP;
  uint256 aa = SECP256K1_AA;
  uint256 bb = SECP256K1_BB;

  // Generator point G (valid, on curve)
  uint256 gx = SECP256K1_GX;
  uint256 gy = SECP256K1_GY;
  assertTrue(EllipticCurve.isOnCurve(gx, gy, aa, bb, pp), "G must be on curve");

  // Create an INVALID point by slightly modifying G's y-coordinate
  // (gx, gy+1) is NOT on the curve
  uint256 invalidX = addmod(gx, 10, pp);
  uint256 invalidY = addmod(gy, 10, pp);
  assertFalse(EllipticCurve.isOnCurve(invalidX, invalidY, aa, bb, pp),
    "Point must NOT be on curve");

  // Add valid point G to invalid point (gx, gy+1)
  (uint256 resultX, uint256 resultY) = EllipticCurve.ecAdd(
    gx, gy,             // Valid point G
    invalidX, invalidY, // INVALID point (not on curve)
    aa, pp
  );

  // The result is meaningless
  // It's not point at infinity, and it isn't on the curve either
  assertFalse(isPointAtInfinity(resultX, resultY),
    "Result should not be point at infinity for this case");

  // And it should not be on the curve either
  assertFalse(EllipticCurve.isOnCurve(resultX, resultY, aa, bb, pp),
    "Result should not be a valid point on the curve");
}
```

However, the severity of this issue is as low considering that the calculation of `unitDeltaX` and `unitDeltaY` correctly produces coordinates of points on the `secp256k1` finite field elliptic curve (ref).

Recommendation

Validate points before use:

```solidity
uint256 internal constant _BB = 7;

function add(CurvePoint memory p1, CurvePoint memory p2) internal pure returns
(CurvePoint memory sum) {
    require(
        EllipticCurve.isOnCurve({_x: p1.x, _y: p1.y, _aa: _AA, _bb: _BB, _pp: _PP}),
        "Point must be on curve"
    )
    require(
        EllipticCurve.isOnCurve({_x: p2.x, _y: p2.y, _aa: _AA, _bb: _BB, _pp: _PP}),
        "Point must be on curve"
    )

  (sum.x, sum.y) = EllipticCurve.ecAdd({_x1: p1.x, _y1: p1.y, _x2: p2.x, _y2: p2.y,
  _aa: _AA, _pp: _PP});
}
```

Status

The development team addressed this finding in PR #396.

No automatic memory zeroing of sensitive cryptographic data

Severity Low **Exploitability** Low **Status** Patched Without Reaudit

Type Implementation **Impact** Medium

Involved artifacts

- `arm-risc0`: `arm/src/encryption.rs`

Description

The encryption module does not automatically zero sensitive cryptographic data from memory after use. None of the cryptographic types used implement automatic memory zeroing, leaving private keys, encryption keys, plaintexts, and intermediate values in memory until overwritten or the process terminates.

Problem scenarios

The lack of memory zeroing creates attack vectors: if private keys and encryption keys remain in memory, they are vulnerable to extraction via memory dumps.

Recommendation

Add `zeroize` dependency (ref):

```toml
## Cargo.toml
[dependencies]
zeroize = { version = "1.8.2" }
```

Implement `Zeroize` and `ZeroizeOnDrop` for `SecretKey` and `InnerSecretKey`:

```rust
use zeroize::{Zeroize, ZeroizeOnDrop};

// SecretKey - implement Zeroize manually
pub struct SecretKey(Scalar);

impl Zeroize for SecretKey {
  fn zeroize(&mut self) {
    // Manually zero the Scalar's inner U256
    // Use volatile write to prevent compiler optimization
    unsafe {
      let ptr = &mut self.0 as *mut Scalar as *mut [u8; 32];
```

```rust
            core::ptr::write_volatile(ptr, [0u8; 32]);
        }
    }
}

// Mark as ZeroizeOnDrop - auto-generates Drop impl that calls zeroize()
impl ZeroizeOnDrop for SecretKey {}

// InnerSecretKey - implement Zeroize manually
struct InnerSecretKey(Key<Aes256Gcm>);

impl Zeroize for InnerSecretKey {
    fn zeroize(&mut self) {
        // Zero the Key's bytes
        // Key<Aes256Gcm> is GenericArray<u8, U32>
        self.0.as_mut_slice().zeroize();
    }
}

// Mark as ZeroizeOnDrop - auto-generates Drop impl that calls zeroize()
impl ZeroizeOnDrop for InnerSecretKey {}
```

Zero intermediate buffers:

```rust
fn generate_shared_key(shared_point: &ProjectivePoint, pk: &ProjectivePoint)
-> Self {
    let pk_bytes = pk.to_bytes();
    let key_bytes = shared_point.to_bytes();
    let mut concat = [&pk_bytes[..], &key_bytes[..]].concat();
    let hash = hash_bytes(&concat);

    // Zero intermediate concatenated bytes containing shared secret
    concat.zeroize();

    let shared_key = hash.as_bytes();
    let key = Key::<Aes256Gcm>::from_slice(&shared_key[..32]);
    InnerSecretKey(*key)
}
```

Provide secure plaintext wrapper:

```rust
use zeroize::{Zeroize, ZeroizeOnDrop};

/// Wrapper for plaintext that automatically zeros on drop
pub struct SecurePlaintext(Vec<u8>);
```

```rust

// Vec<u8> already implements Zeroize, so we can implement ZeroizeOnDrop
impl ZeroizeOnDrop for SecurePlaintext {}

impl SecurePlaintext {
  pub fn new(data: Vec<u8>) -> Self {
    SecurePlaintext(data)
  }

  pub fn as_bytes(&self) -> &[u8] {
    &self.0
  }
}

impl Ciphertext {
  /// Decrypt into SecurePlaintext (automatically zeroed on drop)
  pub fn decrypt(&self, sk: &SecretKey) -> Result<SecurePlaintext, ArmError> {
    // ... existing implementation ...

    let plaintext = aes_gcm
      .decrypt(&cipher.nonce.into(), cipher.cipher.as_ref())
      .map_err(|_| ArmError::DecryptionFailed)?;

    Ok(SecurePlaintext::new(plaintext))
  }
}
```

Status

The development team addressed this finding in PR #158.

The encryption module requires nonces to be computed by applications

Severity	Low	**Exploitability**	Low	**Status**	Patched Without Reaudit
Type	Design	**Impact**	Medium		

Involved artifacts

- `arm-risc0`: `arm/src/encryption.rs`

Description

Nonces for encryption are provided by callers instead of being computed internally. The nonce should ideally be generated randomly and internally to guarantee uniqueness. However, since the `Ciphertext::encrypt` API is also used in circuits—where random generators aren't available—the nonce must be passed in. The expectation is that applications (native and non-native), which also use the module, generate nonces securely. Nevertheless, this cannot be guaranteed.

Problem scenarios

Using non-unique nonces during encryption could allow attackers to recover the plaintext, violating confidentiality.

Recommendation

We recommend exposing two encrypt functions: `encrypt_with_nonce` for the circuits and `encrypt` for applications where the nonce is computed internally,

Status

The development team addressed this finding in PR #158.

Missing checks of logic app_data size

Severity Low **Exploitability** Low **Status** Acknowledged

Type Implementation **Impact** Low

Description

The struct `LogicVerifierInputs` contains the field `app_data`, which contains a number of vector fields. The input comes from applications, but there is no check on the size of `app_data`.

Problem scenarios

The `app_data` is a part of `LogicInstance` returned by the function `to_instance`. This becomes a part of the return value of the function `to_logic_verifier`. This is subsequently added to the vector output of the function `get_logic_verifiers`. From there, `get_batch_lp` adds it to its own `instances` field.

Finally, these instances are written to the `env`. In the aggregation circuit, they are committed as the output.

This bloats the public output. It does not affect the proving time, since the proofs are added as assumptions to the aggregation.

Recommendation

We recommend adding a check for the size of `app_data`. In general, it may make sense to add sanity checks for other fields of the `LogicVerifierInputs`, even though the other fields should not affect the computation as much.

When reserving commitment bytes, some fields treated as a wrong type

Severity Informational **Exploitability** None **Status** Patched Without Reaudit

Type Implementation **Impact** Low

Description

When creating a commitment, the resource field `label_ref` of type `Digest` had reserved `DEFAULT_BYTES = 32` number of bytes. The same is true when calculating `RESOURCE_BYTES = 3*DIGEST_BYTES + 3*DEFAULT_BYTES + 1*QUANTITY_BYTES + 1`, while the `Resource` struct contains `4*DIGEST_BYTES + 2*DEFAULT_BYTES + 1*QUANTITY_BYTES + 1`.

The same is true in the calculation of `rcm()`, when for `rand_seed` and `nonce` we reserve `2 * DIGEST_BYTES` (instead of `2 * DEFAULT_BYTES`).

Problem scenarios

There is no immediate problem, because `DEFAULT_BYTES = 32` , and `DIGEST_BYTES = DIGEST_WORDS * WORD_SIZE = 8 * 4 = 32`.

Flagging this nonetheless since it may cause problems if in the future the used default number of bytes is changed.

Recommendation

Treat `label_ref` explicitly as of type `Digest` when calculating the bytes size to be added. If there are particular reasons for why this is fully impossible, we suggest removing the distinction between `DEFAULT_BYTES` and `DIGEST_BYTES`.

Status

The development team addressed this finding in PR #158.

Lack of domain separation in authorization signatures

Severity Informational **Exploitability** None **Status** Patched Without Reaudit

Type Implementation **Impact** None

Involved artifacts

- `arm-risc0`: `arm/src/authorization.rs`

Description

The authorization module signs messages without domain separation, making signatures potentially reusable across different operation types and applications. The `sign()` method (code ref) accepts raw byte arrays without binding the signature to any specific context. Authorization signatures are used across multiple examples (`kudo_application`, `simple_transfer_application`) for operations that sign action tree roots using the same signature mechanism with no indication of which application generated the signature or what operation the signature authorizes. While the current protocol design provides strong protection through the nullifier mechanism (each resource can only be consumed once), adding domain separation would provide defense-in-depth and align with industry best practices.

Recommendation

Add domain separation as a defense measure and future-proofing enhancement. Use the application's `logic_ref` (verification key) as the unforgeable domain separator.

```rust
// In arm/src/authorization.rs

impl AuthorizationSigningKey {
  /// Sign a message with application context binding
  ///
  /// Binds the signature to a specific application using its verification key.
  /// This provides defense-in-depth alongside the protocol's nullifier mechanism.
  ///
  /// # Arguments
  /// * `app_logic_ref` - The application's logic verification key (unforgeable
  identifier)
  /// * `message` - The message to sign (typically action tree root)
  pub fn sign_for_application(
    &self,
    app_logic_ref: &Digest,
    message: &[u8],
  ) -> AuthorizationSignature {
    let mut domain_separated_msg = Vec::with_capacity(
      16 + app_logic_ref.as_bytes().len() + message.len()
    );

    // Protocol version prefix
    domain_separated_msg.extend_from_slice(b"ARM_AUTH_V1");

    // Application identity (unforgeable)
    domain_separated_msg.extend_from_slice(app_logic_ref.as_bytes());

    // Actual message
    domain_separated_msg.extend_from_slice(message);

    AuthorizationSignature(self.0.sign(&domain_separated_msg))
  }
}

impl AuthorizationVerifyingKey {
  /// Verify a signature with application context binding
  pub fn verify_for_application(
    &self,
    app_logic_ref: &Digest,
    message: &[u8],
    signature: &AuthorizationSignature,
  ) -> Result<(), ArmError> {
    let mut domain_separated_msg = Vec::with_capacity(
      16 + app_logic_ref.as_bytes().len() + message.len()
    );

    domain_separated_msg.extend_from_slice(b"ARM_AUTH_V1");
    domain_separated_msg.extend_from_slice(app_logic_ref.as_bytes());
    domain_separated_msg.extend_from_slice(message);

    VerifyingKey::from_affine(self.0)
      .map_err(|_| ArmError::InvalidPublicKey)?
      .verify(&domain_separated_msg, signature.inner())
      .map_err(|_| ArmError::InvalidSignature)
  }
}
```

```rust
// In application code (e.g., Kudo transfer)
let kudo_logic_ref = KudoMainInfo::verifying_key();
let action_tree_root = action_tree.root();

```

```
// Sign with application binding (new method)
let signature = owner_key.sign_for_application(
  &kudo_logic_ref,
  action_tree_root.as_bytes()
);

// Verify with same binding
owner_pk.verify_for_application(
  &kudo_logic_ref,
  action_tree_root.as_bytes(),
  &signature
)?;
```

Status

The development team addressed this finding in PR #158.

Lack of domain separation in Merkle tree implementation

Severity	Informational	**Exploitability**	None	**Status**	Acknowledged
Type	Design	**Impact**	None		

Involved artifacts

- `arm-risc0`: `arm/src/utils.rs`
- `arm-risc0`: `arm/src/action_tree.rs`
- `arm-risc0`: `arm/src/merkle_path.rs`
- `evm-protocol-adapter`: `contracts/src/libs/SHA256.sol`
- `evm-protocol-adapter`:`contracts/src/libs/MerkleTree.sol`
- `evm-protocol-adapter`: `contracts/src/state/CommitmentTree.sol`

Description

The Merkle tree implementation lacks domain separation between leaf and internal node hashing, violating RFC 6962 (Certificate Transparency) requirements for Merkle tree security.

Rust implementation: The `hash_two()` function (code ref) hashes both leaf pairs and internal node pairs identically using `H(left || right)` without any distinguishing prefix or marker.

Solidity implementation: The `SHA256.hash(bytes32 a, bytes32 b)` function (code ref) uses `sha256(abi.encode(a, b))` for both internal nodes (in `MerkleTree.push()` and `computeRoot()`) and leaf pairs (in action tree construction) without domain separation.

Without domain separation, a Merkle tree structure theoretically allows an attacker to present an internal node value as if it were a leaf value. However, the current implementation stores cryptographic hashes (resource commitments and nullifiers) as leaf values (not arbitrary data) making a second preimage attack not practically exploitable. For a second preimage attack to succeed, an attacker would need to:

1. Compute target: `target = H(commitment_A || commitment_B)`
2. Find resource where: `H(resource_fields) = target`
3. This is a preimage attack on SHA-256 (computationally infeasible)

Current security relies entirely on the protocol-level constraint that all leaf values must be hashed. The Merkle tree structure itself provides no protection.

Problem scenarios

- **Single point of failure**: Security depends solely on protocol-level hashing. The Merkle tree structure itself provides no protection. This violates defense-in-depth principles where multiple independent security layers should exist.
- **RFC 6962 non-compliance**: Industry standard mandates domain separation regardless of leaf content.

Recommendation

Despite the current preimage barrier making practical exploitation infeasible, domain separation is recommended to be implemented as a security enhancement following industry best practices and standards compliance.

Rust (`arm-risc0`)

```rust
// In arm/src/utils.rs

const LEAF_DOMAIN_SEPARATOR: u8 = 0x00;
const INTERNAL_DOMAIN_SEPARATOR: u8 = 0x01;

/// Hash a leaf value with domain separation
pub fn hash_leaf(leaf: &Digest) -> Digest {
  let mut bytes = Vec::with_capacity(33);
  bytes.push(LEAF_DOMAIN_SEPARATOR);
  bytes.extend_from_slice(leaf.as_bytes());
  *Impl::hash_bytes(&bytes)
}

/// Hash two internal nodes with domain separation
pub fn hash_internal(left: &Digest, right: &Digest) -> Digest {
  let mut bytes = Vec::with_capacity(65);
  bytes.push(INTERNAL_DOMAIN_SEPARATOR);
  bytes.extend_from_slice(left.as_bytes());
  bytes.extend_from_slice(right.as_bytes());
  *Impl::hash_bytes(&bytes)
    }
```

```rust
// In arm/src/action_tree.rs

impl MerkleTree {
  pub fn root(&self) -> Result<Digest, ArmError> {
    if self.leaves.is_empty() {
      return Err(ArmError::EmptyTree);
    }

    let len = self.leaves.len()
      .checked_next_power_of_two()
      .ok_or(ArmError::TreeTooLarge)?;

    // Hash leaves with LEAF domain separator
    let mut cur_layer: Vec<Digest> = self.leaves
      .iter()
      .map(|leaf| hash_leaf(leaf))
      .collect();

    // Pad with hashed padding leaf
    cur_layer.resize(len, hash_leaf(&PADDING_LEAF));

    // Hash internal nodes with INTERNAL domain separator
    while cur_layer.len() > 1 {
      cur_layer = cur_layer
        .chunks(2)
        .map(|pair| hash_internal(&pair[0], &pair[1]))
        .collect();
    }

    Ok(cur_layer[0])
  }
}
```

```rust
// In arm/src/merkle_path.rs

impl MerklePath {
  pub fn root(&self, leaf: &Digest) -> Result<Digest, ArmError> {
    if self.0.is_empty() {
      return Err(ArmError::EmptyPath);
    }

    // Apply internal node hashing for path traversal
    Ok(self.0.iter().fold(
      hash_leaf(leaf),  // Start with hashed leaf
      |current, (sibling, is_sibling_left)| match is_sibling_left {
        true => hash_internal(sibling, &current),   // Sibling on left
        false => hash_internal(&current, sibling),  // Sibling on right
      }
    ))
  }
}
```

Solidity (`evm-protocol-adapter`)

```solidity
// In contracts/src/libs/SHA256.sol

library SHA256 {
  /// @notice Domain separator for leaf hashing
  bytes1 private constant LEAF_DOMAIN_SEPARATOR = 0x00;

  /// @notice Domain separator for internal node hashing
  bytes1 private constant INTERNAL_DOMAIN_SEPARATOR = 0x01;

  /// @notice The hash of the string "EMPTY".
  bytes32 public constant EMPTY_HASH =
  0xcc1d2f838445db7aec431df9ee8a871f40e7aa5e064fc056633ef8c60fab7b06;

  /// @notice Hashes a single `bytes32` value.
  function hash(bytes32 a) internal pure returns (bytes32 ha) {
    ha = sha256(abi.encode(a));
  }

  /// @notice Hashes a leaf value with domain separation
  function hashLeaf(bytes32 leaf) internal pure returns (bytes32 hashedLeaf) {
    hashedLeaf = sha256(abi.encode(LEAF_DOMAIN_SEPARATOR, leaf));
  }

  /// @notice Hashes two internal nodes with domain separation
  function hashInternal(bytes32 a, bytes32 b) internal pure returns (bytes32
  hashedInternal) {
    hashedInternal = sha256(abi.encode(INTERNAL_DOMAIN_SEPARATOR, a, b));
  }
}
```

```solidity
// In contracts/src/libs/MerkleTree.sol

function setup(Tree storage self) internal returns (bytes32 initialRoot) {
  initialRoot = SHA256.hashLeaf(SHA256.EMPTY_HASH);
  Arrays.unsafeSetLength(self._zeros, 256);

  bytes32 currentZero = SHA256.hashLeaf(SHA256.EMPTY_HASH);
  for (uint256 i = 0; i < 256; ++i) {
    Arrays-unsafeAccess(self-_zeros, i)-value = currentZero;
    // Use internal node hashing with domain separator
    currentZero = SHA256-hashInternal(currentZero, currentZero);
  }

  self-_nextLeafIndex = 0;
```

```
}

function push(Tree storage self, bytes32 leaf) internal returns (uint256 index, bytes32 newRoot) {
  uint256 treeDepth = depth(self);
  index = self-_nextLeafIndex++;

  bytes32 currentLevelHash = SHA256-hashLeaf(leaf);
  uint256 currentIndex = index;
  for (uint256 i = 0; i < treeDepth; ++i) {
    if (isLeftChild(currentIndex)) {
      Arrays-unsafeAccess(self-_sides, i)-value = currentLevelHash;
      // Use internal node hashing with domain separator
      currentLevelHash = SHA256-hashInternal(currentLevelHash, Arrays-unsafeAccess(self-_zeros, i)-value);
    } else {
      // Use internal node hashing with domain separator
      currentLevelHash = SHA256-hashInternal(Arrays-unsafeAccess(self-_sides, i)-value, currentLevelHash);
    }
    currentIndex >>= 1;
  }

  if (self._nextLeafIndex == capacity(self)) {
    self._sides.push(currentLevelHash);
    // Use internal node hashing with domain separator
    currentLevelHash = SHA256.hashInternal(currentLevelHash, Arrays.unsafeAccess(self._zeros, treeDepth).value);
  }

  newRoot = currentLevelHash;
}

function processProof(bytes32[] memory siblings, uint256 directionBits, bytes32 leaf)
  internal pure returns (bytes32 root)
{
  bytes32 computedHash = SHA256.hashLeaf(leaf);

  uint256 treeDepth = siblings.length;
  for (uint256 i = 0; i < treeDepth; ++i) {
    // Use internal node hashing with domain separator
    if (isLeftSibling(directionBits, i)) {
      computedHash = SHA256-hashInternal(siblings[i], computedHash);
    } else {
      computedHash = SHA256-hashInternal(computedHash, siblings[i]);
    }
  }
  root = computedHash;
}

function computeRoot(bytes32[] memory leaves, uint8 treeDepth) internal pure returns (bytes32 root) {
  uint256 treeCapacity = uint256(1) << treeDepth;

  // Apply leaf domain separator to all leaves
  bytes32[] memory nodes = new bytes32[](treeCapacity);
  for (uint256 i = 0; i < treeCapacity; ++i) {
    if (i < leaves-length) {
      nodes[i] = SHA256-hashLeaf(leaves[i]);
    } else {
      nodes[i] = SHA256-hashLeaf(SHA256-EMPTY_HASH);
    }
  }

  // Build tree with internal node hashing
  uint256 currentLevelCapacity = treeCapacity;
  while (currentLevelCapacity > 1) {
    currentLevelCapacity /= 2;

    for (uint256 i = 0; i < currentLevelCapacity; ++i) {
      // Use internal node hashing with domain separator
      nodes[i] = SHA256.hashInternal(nodes[2 * i], nodes[2 * i + 1]);
    }
  }

  root = nodes[0];
}
```

Missing identity point validation in ECDH key exchange

Severity Informational **Exploitability** None **Status** Patched Without Reaudit

Type Implementation **Impact** None

Involved artifacts

- `arm-risc0`: `arm/src/encryption.rs`

Description

The encryption module does not validate that elliptic curve points used in ECDH key exchange are non-identity points. While `k256`'s `AffinePoint` deserialization validates that points lie on the `secp256k1` curve, it accepts the identity point. The identity point has the mathematical property that `IDENTITY * any_scalar = IDENTITY`, which makes ECDH operations with the identity point produce predictable shared secrets. This could lead to privacy breaches if user code mistakenly provides the identity point as a public key. As a defense-in-depth measure, the encryption module should validate and reject identity points at all ECDH entry points to prevent such misuse, even though correct usage should never provide identity points.

Problem scenarios

Application calls `Ciphertext::encrypt` with the identity point as the receiver public key:

```rust
// From simple_transfer_witness/src/lib.rs

let cipher = Ciphertext::encrypt(
  &self.resource.to_bytes()?,          // Plaintext resource data
  &encryption_info.encryption_pk,      // ← IDENTITY point (attacker-influenced)
  &encryption_info.sender_sk,          // Sender secret key
  encryption_info.encryption_nonce... // Random nonce
)?;
```

The receiver public key is passed to `InnerSecretKey::from_encryption` to derive the shared encryption key:

```rust
// From encryption.rs

pub fn encrypt(
  message: &Vec<u8>,
  receiver_pk: &AffinePoint,   // ← IDENTITY point
  sender_sk: &SecretKey,       // Sender secret key
```

```
    nonce: [u8; 12],
) -> Result<Self, ArmError> {
    // Derive encryption key via ECDH
    let inner_secret_key = InnerSecretKey::from_encryption(receiver_pk,
    sender_sk.inner());

      // ... rest of implementation ...
}
```

In `InnerSecretKey::from_encryption` the generated shared point is again the identity point:

```rust
// From encryption.rs

pub fn from_encryption(pk: &AffinePoint, sk: &Scalar) -> Self {
    // pk = receiver_pk (IDENTITY in attack)
    // sk = sender_sk (random)

    let pk = ProjectivePoint::from(*pk);  // Convert to projective

    // CRITICAL: ECDH computation
    let shared_point = pk * sk;
    // If pk is IDENTITY: shared_point = IDENTITY * sender_sk = IDENTITY
    // Mathematical property: IDENTITY * anything = IDENTITY

    // Pass to key derivation
    Self::generate_shared_key(&shared_point, &pk)
    //                          ↑             ↑
    //                          |             |
    //                       IDENTITY      IDENTITY (receiver_pk)
}
```

Using the identity point as the shared point makes the shared key a compatible value by an attacker:

```rust
// From encryption.rs

fn generate_shared_key(shared_point: &ProjectivePoint, pk: &ProjectivePoint) -> Self
{
    // shared_point = IDENTITY (from ECDH)
    // pk = IDENTITY (receiver_pk)

    let pk_bytes = pk.to_bytes();
    // pk_bytes = IDENTITY.to_bytes()
    // This is a KNOWN CONSTANT

    let key_bytes = shared_point.to_bytes();
```

```
  // key_bytes = IDENTITY.to_bytes()
  // This is the SAME KNOWN CONSTANT

  let hash = hash_bytes(&[&pk_bytes[..], &key_bytes[..]].concat());
  // hash = Hash(IDENTITY_bytes || IDENTITY_bytes)
  // BOTH inputs are known constants!
  // ATTACKER CAN COMPUTE THIS EXACT HASH

  let shared_key = hash.as_bytes();
  let key = Key::<Aes256Gcm>::from_slice(&shared_key[..32]);
  InnerSecretKey(*key)
  // Returns AES key derived from known values
}
```

As a result any information (e.g. resources) encrypted with the predictable shared key could be decrypted by anyone.

Recommendation

Add identity point validation at all ECDH entry points:

```rust
// From encryption.rs

pub fn from_encryption(pk: &AffinePoint, sk: &Scalar) -> Result<Self, ArmError> {
  // Validate: Reject identity point
  if bool::from(pk.is_identity()) {
    return Err(ArmError::InvalidPublicKey);
  }

  // ... rest of the implementation ...
}

pub fn from_decryption(pk: &AffinePoint, sk: &Scalar) -> Result<Self, ArmError> {
  // Validate: Reject identity point
  if bool::from(pk.is_identity()) {
    return Err(ArmError::InvalidPublicKey);
  }

  // ... rest of the implementation ...
}

fn generate_shared_key_checked(
  shared_point: &ProjectivePoint,
  pk: &ProjectivePoint
) -> Result<Self, ArmError> {
  // Validate: Reject identity as shared secret (defense in depth)
  if bool::from(shared_point.is_identity()) {
    return Err(ArmError::InvalidSharedSecret);
  }

  // Validate: Reject identity as public key (defense in depth)
  if bool::from(pk.is_identity()) {
    return Err(ArmError::InvalidPublicKey);
  }

  // ... rest of the implementation ...
}
```

Status

The development team addressed this finding in PR #158.

Invalid root and Merkle path for empty action trees

Severity Informational **Exploitability** None **Status** Patched Without Reaudit

Type Implementation **Impact** None

Involved artifacts

- `arm-risc0`: `arm/src/action_tree.rs`
- `evm-protocol-adapter`: `contracts/src/libs/MerkleTree.sol`

Description

The `generate_path()` (code ref) and `root()` (code ref) functions of `MerkleTree` incorrectly handle empty Merkle trees, leading to semantically invalid operations. When `self.leaves` is empty:

1. In `generate_path()`: The function pads the tree to size 1 with `PADDING_LEAF`, then successfully generates an authentication path if the caller provides `PADDING_LEAF` as the target leaf, returning an empty path that verifies against the padding leaf as root.
2. In `root()`: The function returns `PADDING_LEAF` as the root of an empty tree, creating a well-known, predictable root value for empty trees.

Both behaviors violate the semantic invariant that padding leaves are placeholder values for tree structure, not actual data elements representing valid resources.

Additionally, in `evm-protocol-adapter`'s `MerkleTree` library when `leaves.length == 0:`

The `computeRoot(bytes32[] memory leaves)` function (code ref) exhibits the same issue:

1. Calls `computeMinimalTreeDepth(0)` which returns `0`
2. Calls `computeRoot(leaves, 0)` with depth 0
3. Creates tree capacity `2^0 = 1`
4. Fills array with one `SHA256.EMPTY_HASH` (code ref)
5. Returns `EMPTY_HASH` as root without hashing (code ref)

This violates the same semantic invariant that `EMPTY_HASH` is a placeholder value, not a valid root representing actual resources.

Problem scenarios

An attacker could exploit this to claim a padding leaf represents a valid resource in an empty tree:

```rust
let empty_tree = MerkleTree::new(vec![]);
let fake_path = empty_tree.generate_path(&PADDING_LEAF)?;  // Succeeds!
let root = empty_tree.root();  // Returns PADDING_LEAF
assert_eq!(fake_path.root(&PADDING_LEAF), root);  // Verification passes!
```

An adversary could potentially construct a proof that a non-existent resource exists in an empty action tree.

In `evm-protocol-adapter`, an adversary could construct an action with zero compliance units (empty leaves array), which would produce a predictable `EMPTY_HASH` root:

```solidity
bytes32[] memory emptyLeaves = new bytes32[](0);
bytes32 root = MerkleTree.computeRoot(emptyLeaves);  // Returns EMPTY_HASH
// Attacker knows the root without seeing the transaction
```

Recommendation

Rust (`arm-risc0`)

Add explicit validation at the beginning of both functions:

```rust
pub fn root(&self) -> Result<Digest, ArmError> {
  // Reject empty trees
  if self.leaves.is_empty() {
    return Err(ArmError::EmptyTree);
  }

  // ... rest of implementation
}

pub fn generate_path(&self, cur_leave: &Digest) -> Result<MerklePath, ArmError> {
  // Reject empty trees
  if self.leaves.is_empty() {
    return Err(ArmError::EmptyTree);
  }

  // ... rest of implementation
}
```

Additionally, in `generate_path()`, explicitly reject padding leaves as search targets:

```rust
pub fn generate_path(&self, cur_leave: &Digest) -> Result<MerklePath,
ArmError> {
  if self.leaves.is_empty() {
    return Err(ArmError::EmptyTree);
  }

  // Reject padding leaf as a valid search target
  if cur_leave == &*PADDING_LEAF {
    return Err(ArmError::InvalidLeaf);
  }

  // ... rest of implementation
}
```

Solidity (`evm-protocol-adapter`)

For `evm-protocol-adapter`, add validation to reject empty leaf arrays:

```solidity
function computeRoot(bytes32[] memory leaves, uint8 treeDepth) internal
pure returns (bytes32 root) {
  // Reject empty trees
  if (leaves.length == 0) {
    revert EmptyTree();
  }

  // ... rest of implementation
}

function computeRoot(bytes32[] memory leaves) internal pure returns (bytes32 root) {
  // Reject empty trees
  if (leaves.length == 0) {
    revert EmptyTree();
  }

  root = MerkleTree.computeRoot({leaves: leaves, treeDepth:
  computeMinimalTreeDepth(leaves.length)});
}
```

Status

The development team addressed this finding in PR #158 for `arm-risc0`.

Empty Merkle paths provide no cryptographic authentication

Severity Informational **Exploitability** None **Status** Patched Without Reaudit

Type Implementation **Impact** None

Involved artifacts

- `arm-risc0`: `arm/src/merkle_path.rs`

Description

A `MerklePath` can be constructed with empty authentication paths (code ref), and then the `root()` function (code ref) returns the input leaf unchanged, providing zero cryptographic authentication. When `self.0` is empty, the fold operation never executes its closure and simply returns `*leaf` unchanged (code ref). This means:

- No cryptographic hashing is performed
- No binding between the leaf and any specific tree
- The function degenerates to an identity function: `f(x) = x`

A proper Merkle authentication path should cryptographically bind a specific leaf to a specific root through a chain of hash operations. Empty paths provide no such binding and thus offer no authentication.

Problem scenarios

For legitimate single-element trees, the empty path provides no authentication:

```rust
let tree = MerkleTree::new(vec![legitimate_commitment]);
let path = tree.generate_path(&legitimate_commitment)?;  // Empty path

// Attacker can now "verify" ANY value with it
let fake_root1 = path.root(&attacker_value_1);
let fake_root2 = path.root(&attacker_value_2);
// Both succeed - path authenticates nothing!
```

For scenarios where single-element action trees might occur, empty paths provide no cryptographic guarantee. An attacker could, for example, claim arbitrary commitments are in the tree. The only protection is comparison with `tree.root()`, not the path itself.

Recommendation

There is no structural way to distinguish a legitimate empty path (from a single-element tree) from a forged empty path one. Security relies entirely on external root comparison, not on the path itself. Attackers can trivially forge empty paths that are indistinguishable from legitimate ones.

Our recommendation is for the development team to decide how to handle single-element trees based on protocol and security requirements (e.g., determine if single-element action trees are expected to occur in normal operation or only in edge cases and understand what guarantees Merkle paths are expected to provide).

Status

The development team addressed this finding in PR #158.

Inefficient Merkle tree depth computation

Severity Informational **Exploitability** None **Status** Patched Without Reaudit

Type Implementation **Impact** None

Involved artifacts

- `evm-protocol-adapter`: `contracts/src/libs/MerkleTree.sol`

Description

The `computeMinimalTreeDepth` function (code ref) uses a loop-based approach to calculate the minimal tree depth for a given number of leaves. This can be optimized using a logarithmic formula.

This function is called in `MerkleTree.computeRoot()` (code ref), which is invoked during action tree construction (code ref).

Gas measurements from fuzz testing (10,000 runs):

- Original implementation: μ: 6,163 gas, ~: 6,644 gas
- Optimized implementation: μ: 4,024 gas, ~: 4,213 gas
- Savings: ~2,000 gas average, ~2,400 gas median

Problem scenarios

- Gas inefficiency.
- Theoretical overflow risk: While impractical due to gas limits, the loop-based approach could theoretically overflow the `uint8` return type with malicious inputs (e.g., `leavesCount > 2^255`).

Recommendation

Replace the loop-based implementation with a logarithmic formula using OpenZeppelin's `Math.log2`:

```solidity
/// @notice Optimized implementation using log2
function computeMinimalTreeDepth_Optimized(uint256 leavesCount) internal pure returns
(uint8 treeDepth) {
  return leavesCount <= 1 ? 0 : uint8(Math-log2(leavesCount - 1) + 1);
}
```

The optimization was validated with a comprehensive test suite-

```solidity
 // SPDX-License-Identifier: MIT
 pragma solidity ^0-8-30;

```

```
import {Test} from "forge-std/Test-sol";
import {Math} from "@openzeppelin-contracts/utils/math/Math-sol";

/// @title ComputeMinimalTreeDepth Comparison Test
/// @notice Tests that compare the original loop-based implementation with the optimized log2 formula
contract ComputeMinimalTreeDepthTest is Test {
  /// @notice Original implementation using loop
  function computeMinimalTreeDepth_Original(uint256 leavesCount) internal pure returns (uint8 treeDepth) {
    uint256 treeCapacity = 1;
    treeDepth = 0;

    while (treeCapacity < leavesCount) {
      treeCapacity *= 2;
      ++treeDepth;
    }

    return treeDepth;
  }

  /// @notice Optimized implementation using log2
  function computeMinimalTreeDepth_Optimized(uint256 leavesCount) internal pure returns (uint8 treeDepth) {
    return leavesCount <= 1 ? 0 : uint8(Math-log2(leavesCount - 1) + 1);
  }

  // ========================================
  // Standard Tests
  // ========================================

  function test_compare_zero_leaves() public pure {
    assertEq(computeMinimalTreeDepth_Original(0), computeMinimalTreeDepth_Optimized(0));
    assertEq(computeMinimalTreeDepth_Optimized(0), 0);
  }

  function test_compare_one_leaf() public pure {
    assertEq(computeMinimalTreeDepth_Original(1), computeMinimalTreeDepth_Optimized(1));
    assertEq(computeMinimalTreeDepth_Optimized(1), 0);
  }

  function test_compare_two_leaves() public pure {
    assertEq(computeMinimalTreeDepth_Original(2), computeMinimalTreeDepth_Optimized(2));
```

```
        assertEq(computeMinimalTreeDepth_Optimized(2), 1);
    }

    function test_compare_three_leaves() public pure {
        assertEq(computeMinimalTreeDepth_Original(3),
        computeMinimalTreeDepth_Optimized(3));
        assertEq(computeMinimalTreeDepth_Optimized(3), 2);
    }

    function test_compare_four_leaves() public pure {
        assertEq(computeMinimalTreeDepth_Original(4),
        computeMinimalTreeDepth_Optimized(4));
        assertEq(computeMinimalTreeDepth_Optimized(4), 2);
    }

    function test_compare_five_leaves() public pure {
        assertEq(computeMinimalTreeDepth_Original(5),
        computeMinimalTreeDepth_Optimized(5));
        assertEq(computeMinimalTreeDepth_Optimized(5), 3);
    }

    function test_compare_eight_leaves() public pure {
        assertEq(computeMinimalTreeDepth_Original(8),
        computeMinimalTreeDepth_Optimized(8));
        assertEq(computeMinimalTreeDepth_Optimized(8), 3);
    }

    function test_compare_nine_leaves() public pure {
        assertEq(computeMinimalTreeDepth_Original(9),
        computeMinimalTreeDepth_Optimized(9));
        assertEq(computeMinimalTreeDepth_Optimized(9), 4);
    }

    function test_compare_sixteen_leaves() public pure {
        assertEq(computeMinimalTreeDepth_Original(16),
        computeMinimalTreeDepth_Optimized(16));
        assertEq(computeMinimalTreeDepth_Optimized(16), 4);
    }

    function test_compare_seventeen_leaves() public pure {
        assertEq(computeMinimalTreeDepth_Original(17),
        computeMinimalTreeDepth_Optimized(17));
        assertEq(computeMinimalTreeDepth_Optimized(17), 5);
    }

    function test_compare_power_of_two_boundaries() public pure {
```

```
    // Test powers of 2 and their neighbors
    uint256[10] memory testCases = [uint256(1), 2, 4, 8, 16, 32, 64, 128, 256, 512];

    for (uint256 i = 0; i < testCases-length; i++) {
      uint256 powerOfTwo = testCases[i];

      assertEq(
        computeMinimalTreeDepth_Original(powerOfTwo - 1), computeMinimalTreeDepth_Optimized(powerOfTwo - 1)
      );

      // Test power of 2
      assertEq(computeMinimalTreeDepth_Original(powerOfTwo), computeMinimalTreeDepth_Optimized(powerOfTwo));

      // Test power of 2 + 1
      assertEq(
        computeMinimalTreeDepth_Original(powerOfTwo + 1), computeMinimalTreeDepth_Optimized(powerOfTwo + 1)
      );
    }
  }

  function test_compare_large_values() public pure {
    // Test some large but realistic values
    uint256[5] memory testCases = [uint256(1000), 10000, 100000, 1000000, 10000000];

    for (uint256 i = 0; i < testCases-length; i++) {
      assertEq(computeMinimalTreeDepth_Original(testCases[i]), computeMinimalTreeDepth_Optimized(testCases[i]));
    }
  }

  // ========================================
  // Fuzz Tests
  // ========================================

  /// @notice Fuzz test with unbounded uint256
  function testFuzz_compare_implementations_unbounded(uint256 leavesCount) public pure {
    leavesCount = bound(leavesCount, 0, type(uint128)-max);

    uint8 original = computeMinimalTreeDepth_Original(leavesCount);
    uint8 optimized = computeMinimalTreeDepth_Optimized(leavesCount);

    assertEq(original, optimized, "Implementations must match");
  }

  /// @notice Fuzz test with small values (0-1000)
  function testFuzz_compare_implementations_small(uint256 leavesCount) public pure {
    leavesCount = bound(leavesCount, 0, 1000);

    uint8 original = computeMinimalTreeDepth_Original(leavesCount);
    uint8 optimized = computeMinimalTreeDepth_Optimized(leavesCount);

    assertEq(original, optimized, "Implementations must match");
  }

  /// @notice Fuzz test with medium values (1000-1000000)
  function testFuzz_compare_implementations_medium(uint256 leavesCount) public pure
  {
    leavesCount = bound(leavesCount, 1000, 1000000);

    uint8 original = computeMinimalTreeDepth_Original(leavesCount);
    uint8 optimized = computeMinimalTreeDepth_Optimized(leavesCount);

    assertEq(original, optimized, "Implementations must match");
  }

  // ========================================
  // Gas Measurement Tests - Fuzz
  // ========================================

  /// @notice Fuzz test - Original implementation only (to measure gas)
  function testFuzz_gas_original(uint256 leavesCount) public pure {
    leavesCount = bound(leavesCount, 0, 1000000);
    computeMinimalTreeDepth_Original(leavesCount);
  }

  /// @notice Fuzz test - Optimized implementation only (to measure gas)
  function testFuzz_gas_optimized(uint256 leavesCount) public pure {
    leavesCount = bound(leavesCount, 0, 1000000);
    computeMinimalTreeDepth_Optimized(leavesCount);
  }
}
```

Status

The development team addressed this finding in PR #397.

Miscellaneous code improvements

Severity Informational **Exploitability** None **Status** Patched Without Reaudit

Type Implementation **Impact** None

`arm-risc0`

- The `root()` and `generate_path()` methods use `next_power_of_two()` (here and here) to round up the leaf count for tree construction. For an N-bit `usize` type, when `self.leaves.len()` exceeds 2^(N-1), `next_power_of_two()` would need to return 2^N to provide the next power of two, but 2^N doesn't fit in an N-bit unsigned integer. This causes integer overflow, wrapping the result to 0. The subsequent `resize(0, ...)` operation empties te vector, leading to an out-of-bounds panic at `cur_layer[0]` (code ref) with an unclear error message. While this scenario is practically impossible, handling it gracefully with `checked_next_power_of_two()` provides better error diagnostics.

  Recommendation: Use `checked_next_power_of_two()` to detect overflow and return a proper error instead of allowing a panic with an unclear message.

```rust
// In arm/src/error.rs, add a new error variant: TreeTooLarge

  pub fn root(&self) -> Result<Digest, ArmError> {
    let len = self.leaves.len()
      .checked_next_power_of_two()
      .ok_or(ArmError::TreeTooLarge)?;

    let mut cur_layer = self.leaves.clone();
    cur_layer.resize(len, *PADDING_LEAF);

    while cur_layer.len() > 1 {
      cur_layer = cur_layer
        .chunks(2)
        .map(|pair| hash_two(&pair[0], &pair[1]))
        .collect();
    }

    Ok(cur_layer[0])
  }

  pub fn generate_path(&self, cur_leave: &Digest) -> Result<MerklePath, ArmError> {
    let len = self.leaves.len()
      .checked_next_power_of_two()
      .ok_or(ArmError::TreeTooLarge)?;

    let mut cur_layer = self.leaves.clone();
    cur_layer.resize(len, *PADDING_LEAF);

    // ... rest of the implementation
  }
```

`evm-protocol-adapter`

- The codebase relies heavily on the assumption that compliance units consist of 1 consumed and 1 created resource, leading to hardcoded `* 2` and `/ 2` operations and sensitive index accessing:

  ```solidity
  // RiscZeroUtils.sol
  bytes memory consumedJournal = instance.logicInstances[(i * 2)].toJournal();
  bytes memory createdJournal = instance.logicInstances[(i * 2) + 1].toJournal();

  // TagUtils.sol - checkedActionTagCount
  if (actionTagCount != complianceUnitCount * 2) {
      revert TagCountMismatch({expected: actionTagCount, actual: complianceUnitCount * 2});
  }

  // ProtocolAdapter.sol
  updatedVars.complianceInstances[vars.tagCounter / 2] = input.instance;
  ```

  RISC Zero encoding constants are also hardcoded:

  ```solidity
  // RiscZeroUtils.sol
  input.isConsumed ? uint32(0x01000000) : uint32(0x00000000)  // Little-endian bool
  ```

  Recommendation: Extract into named constants (e.g., `RESOURCES_PER_COMPLIANCE_UNIT = 2`, `RISC0_BOOL_TRUE_LE = 0x01000000`) to improve readability and maintainability.

- `public` functions not used internally could be marked `external`

```solidity
function getProtocolAdapterVersion() public pure override returns (bytes32 version)
```

Status

The development team addressed this finding in PR #398 for `evm-protocol-adapter`.

Feedback on specification

Severity Informational **Exploitability** None **Status** Acknowledged

Type Documentation **Impact** None

In this finding, we provide feedback on the specification. As a side note, we would like to highlight that the navigation bar on the side is a bit awkward to use because it refreshes the page each time you expand the menu.

General feedback

- The abstract specification is not very approachable.
  - It assumes that the reader already knows most details of the overall architecture, i.e., what's a resource machine, its purpose, the context in which resource machine may be used and how it would interact with other components/actors. Paragraphs like "The role of the ARM" definitely help but it is not enough. We recommend providing more context. If this context is provided elsewhere (could not find it), we recommend adding a couple short paragraphs still and links to the elaborated descriptions.
  - Initially, the specification was quite abstract. After reading a concrete implementation (the shielded one), it became much easier to understand. We recommend interleaving examples within the abstract specification to make the concepts more tangible. For instance, the shielded specification could serve as a running example throughout.
  - The specification is not always self-contained and assumes that the reader is an expert or has more context. For instance, I could not parse this on my first read:

    "Each time a commitment is added to the accumulator, the accumulator and all witnesses of the already accumulated commitments are updated. For a commitment that existed in the accumulator before a new one was added, both the old witness and the new witness (with the corresponding accumulated value parameter) can be used to prove membership. However, the older the witness (and, consequently, the accumulator) that is used in the proof, the more information about the resource it reveals (an older accumulator gives more concrete boundaries on the resource's creation time). For that reason, it is recommended to use fresher parameters when proving membership."

    There are more examples as such. For instance, the spec about proving systems does not really explains what witnesses or instances are. Are those part of any proving system or of a subset of them?

    We recommend trying to make things more self-contained.
- We recommend using notes only to clarify things or provide concrete examples and avoid using them to state requirements and constraints. For instance in the Nullifier page, a note is used to state a requirement:

"Every time a resource is consumed, it has to be checked that the resource existed before (the resource's commitment is in the commitment tree) and has not been consumed yet (the resource's nullifier is not in the nullifier set)."

- There is no clear motivation for having multiple compliance units per action. It would be relevant if a transaction may be built using multiple RMs such that each CU may impose different constraints. But as confirmed by the team, this is not a direction being considered at the moment. Removing compliance units completely and assuming that each transaction still includes a RM compliance proofs would certainly simplify the design.

Comments

- In Proving Systems, there is curried notation `f x w` that is never introduced, and is mixed with tuple-style notation (e.g., in `Verify(vk, x, pi)` . It would be less confusing to introduce/mention the use of curried notation upfront.
- In Proving Systems, there is a confusing sentence "A proof π for which `Verify(pr) = True` is considered valid"
  - what is `pr` here?
  - Shouldn't `Verify` (following the previous text) accept 3 arguments?
- In the specification of kind component, the hashing function should be applied to the pair `(label_ref, logic_ref)`. However, in the code, it is applied to `(logic_ref, label_ref)`.
- The delta spec is using the old terminology on *input* and *output* resources, instead of consumed and created. Even with those, it has the signs the other way round than in the code (all of that shouldn't matter since the check is for 0).
- There is a clash between the proving system's witnesses and the accumulator's witnesses, which makes things confusing.
- The note in the Nullifier Set section is confusing. It is unclear if this is a requirement currently and it is just a TODO, or it is not a requirement but it could be in the future. We recommend either removing it if it is not a requirement currently or adding it as a requirement if it is (or may be for some implementations) with a proper motivation.
- The resource pages are somewhat difficult to navigate, with many pages containing very little text. It might be better to consolidate everything into a single page.

Appendix: Testing of ecAdd

Tests to verify the group properties of addition of function `ecAdd` (ref).

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "forge-std/Test.sol";
import "../contracts/EllipticCurve.sol";

/**
 * @title EllipticCurveProperties
 * @dev Property-based tests for EllipticCurve.ecAdd using Foundry fuzzing
 *
 * These tests verify GROUP PROPERTIES with points VERIFIED to be on the curve.
 * Points are generated via scalar multiplication: P = k × G
 * This guarantees all test points satisfy the curve equation y² = x³ + ax + b
 *
 * Run with: forge test -vv
 * Run with more fuzz runs: forge test --fuzz-runs 10000 -vv
 */
contract EllipticCurvePropertiesTest is Test {
  using EllipticCurve for *;

  // secp256k1 parameters
  uint256 constant SECP256K1_PP =
  0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2F;
  uint256 constant SECP256K1_AA = 0;
  uint256 constant SECP256K1_BB = 7;
  uint256 constant SECP256K1_GX =
  0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798;
  uint256 constant SECP256K1_GY =
  0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8;

  // Helper: Check if point is at infinity
  function isPointAtInfinity(uint256 x, uint256 y) internal pure returns (bool) {
    return x == 0 && y == 0;
  }

  /*//////////////////////////////////////////////////////////////
      GROUP PROPERTY TESTS (Points Verified On Curve)
  //////////////////////////////////////////////////////////////*/

  /// @notice GROUP PROPERTY: Commutativity - P + Q = Q + P
  /// @dev Uses secp256k1 with points VERIFIED to be on the curve
  function testGroupProperty_Commutativity_Secp256k1(uint256 scalar1, uint256 scalar2) pure public {
    // Generate two points on secp256k1 by scalar multiplication of G
    scalar1 = bound(scalar1, 1, 20); // Keep small for performance
    scalar2 = bound(scalar2, 1, 20);

    // P1 = scalar1 * G
    (uint256 x1, uint256 y1) = EllipticCurve.ecMul(
      scalar1, SECP256K1_GX, SECP256K1_GY, SECP256K1_AA, SECP256K1_PP
    );

    // P2 = scalar2 * G
    (uint256 x2, uint256 y2) = EllipticCurve.ecMul(
      scalar2, SECP256K1_GX, SECP256K1_GY, SECP256K1_AA, SECP256K1_PP
    );

    // Verify both points are on curve
    assertTrue(
      EllipticCurve.isOnCurve(x1, y1, SECP256K1_AA, SECP256K1_BB, SECP256K1_PP),
      "P1 must be on curve"
    );
    assertTrue(
      EllipticCurve.isOnCurve(x2, y2, SECP256K1_AA, SECP256K1_BB, SECP256K1_PP),
      "P2 must be on curve"
    );

    // Test commutativity: P1 + P2 = P2 + P1
    (uint256 resultX1, uint256 resultY1) = EllipticCurve.ecAdd(
      x1, y1, x2, y2, SECP256K1_AA, SECP256K1_PP
    );
    (uint256 resultX2, uint256 resultY2) = EllipticCurve.ecAdd(
      x2, y2, x1, y1, SECP256K1_AA, SECP256K1_PP
    );

    assertEq(resultX1, resultX2, "GROUP PROPERTY: P + Q must equal Q + P");
    assertEq(resultY1, resultY2, "GROUP PROPERTY: P + Q must equal Q + P");
    }

  /// @notice GROUP PROPERTY: Identity - P + O = P
  function testGroupProperty_Identity_Secp256k1(uint256 scalar) pure public {
    scalar = bound(scalar, 1, 20);

    // P = scalar * G (guaranteed on curve)
    (uint256 x, uint256 y) = EllipticCurve.ecMul(
      scalar, SECP256K1_GX, SECP256K1_GY, SECP256K1_AA, SECP256K1_PP
    );

    // Verify P is on curve
    assertTrue(
      EllipticCurve.isOnCurve(x, y, SECP256K1_AA, SECP256K1_BB, SECP256K1_PP),
      "P must be on curve"
    );

    // P + O should equal P
    (uint256 resultX, uint256 resultY) = EllipticCurve.ecAdd(
      x, y, 0, 0, SECP256K1_AA, SECP256K1_PP
    );

    assertEq(resultX, x, "GROUP PROPERTY: P + O must equal P");
    assertEq(resultY, y, "GROUP PROPERTY: P + O must equal P");
  }

  /// @notice GROUP PROPERTY: Inverse - P + (-P) = O
  function testGroupProperty_Inverse_Secp256k1(uint256 scalar) pure public {
    scalar = bound(scalar, 1, 20);

    // P = scalar * G (guaranteed on curve)
    (uint256 x, uint256 y) = EllipticCurve.ecMul(
    scalar, SECP256K1_GX, SECP256K1_GY, SECP256K1_AA, SECP256K1_PP
    );

    // Verify P is on curve
    assertTrue(
        EllipticCurve.isOnCurve(x, y, SECP256K1_AA, SECP256K1_BB, SECP256K1_PP),
        "P must be on curve"
    );

    // Get -P
    (uint256 invX, uint256 invY) = EllipticCurve.ecInv(x, y, SECP256K1_PP);

    // Verify -P is on curve
    assertTrue(
      EllipticCurve.isOnCurve(invX, invY, SECP256K1_AA, SECP256K1_BB, SECP256K1_PP),
      "-P must be on curve"
    );

    // P + (-P) should equal O
    (uint256 resultX, uint256 resultY) = EllipticCurve.ecAdd(
      x, y, invX, invY, SECP256K1_AA, SECP256K1_PP
    );

    assertTrue(
      isPointAtInfinity(resultX, resultY),
      "GROUP PROPERTY: P + (-P) must equal point at infinity"
    );
  }

  /// @notice GROUP PROPERTY: Associativity - (P + Q) + R = P + (Q + R)
  function testGroupProperty_Associativity_Secp256k1(
    uint256 scalar1,
    uint256 scalar2,
    uint256 scalar3
  ) pure public {
    scalar1 = bound(scalar1, 1, 10); // Keep small for performance
    scalar2 = bound(scalar2, 1, 10);
    scalar3 = bound(scalar3, 1, 10);

    // Generate three points on curve
    (uint256 x1, uint256 y1) = EllipticCurve.ecMul(
      scalar1, SECP256K1_GX, SECP256K1_GY, SECP256K1_AA, SECP256K1_PP
    );
    (uint256 x2, uint256 y2) = EllipticCurve.ecMul(
      scalar2, SECP256K1_GX, SECP256K1_GY, SECP256K1_AA, SECP256K1_PP
    );
    (uint256 x3, uint256 y3) = EllipticCurve.ecMul(
      scalar3, SECP256K1_GX, SECP256K1_GY, SECP256K1_AA, SECP256K1_PP
    );

    // Verify all points are on curve
    assertTrue(EllipticCurve.isOnCurve(x1, y1, SECP256K1_AA, SECP256K1_BB,
    SECP256K1_PP));
    assertTrue(EllipticCurve.isOnCurve(x2, y2, SECP256K1_AA, SECP256K1_BB,
    SECP256K1_PP));
    assertTrue(EllipticCurve.isOnCurve(x3, y3, SECP256K1_AA, SECP256K1_BB,
    SECP256K1_PP));

    // Compute (P + Q) + R
    (uint256 pqX, uint256 pqY) = EllipticCurve.ecAdd(x1, y1, x2, y2, SECP256K1_AA,
    SECP256K1_PP);
    (uint256 resultX1, uint256 resultY1) = EllipticCurve.ecAdd(
      pqX, pqY, x3, y3, SECP256K1_AA, SECP256K1_PP
    );

    // Compute P + (Q + R)
    (uint256 qrX, uint256 qrY) = EllipticCurve.ecAdd(x2, y2, x3, y3, SECP256K1_AA,
    SECP256K1_PP);
    (uint256 resultX2, uint256 resultY2) = EllipticCurve.ecAdd(
      x1, y1, qrX, qrY, SECP256K1_AA, SECP256K1_PP
    );

    assertEq(resultX1, resultX2, "GROUP PROPERTY: Associativity must hold");
    assertEq(resultY1, resultY2, "GROUP PROPERTY: Associativity must hold");
  }

  /// @notice GROUP PROPERTY: Closure - If P, Q on curve, then P+Q on curve or at infinity
  function testGroupProperty_Closure_Secp256k1(uint256 scalar1, uint256 scalar2) pure public {
    scalar1 = bound(scalar1, 1, 20);
    scalar2 = bound(scalar2, 1, 20);

    // Generate two points on curve
    (uint256 x1, uint256 y1) = EllipticCurve.ecMul(
      scalar1, SECP256K1_GX, SECP256K1_GY, SECP256K1_AA, SECP256K1_PP
    );
    (uint256 x2, uint256 y2) = EllipticCurve.ecMul(
      scalar2, SECP256K1_GX, SECP256K1_GY, SECP256K1_AA, SECP256K1_PP
    );

    // Verify inputs are on curve
    assertTrue(EllipticCurve.isOnCurve(x1, y1, SECP256K1_AA, SECP256K1_BB, SECP256K1_PP));
    assertTrue(EllipticCurve.isOnCurve(x2, y2, SECP256K1_AA, SECP256K1_BB, SECP256K1_PP));

    // Add points
    (uint256 resultX, uint256 resultY) = EllipticCurve.ecAdd(
      x1, y1, x2, y2, SECP256K1_AA, SECP256K1_PP
    );

    // Result must be on curve or at infinity
    bool atInfinity = isPointAtInfinity(resultX, resultY);
    bool onCurve = EllipticCurve.isOnCurve(resultX, resultY, SECP256K1_AA, SECP256K1_BB, SECP256K1_PP);

    assertTrue(
      atInfinity || onCurve,
      "GROUP PROPERTY: Closure - result must be on curve or at infinity"
    );
  }

  /*//////////////////////////////////////////////////////////////
```

```
                PRECONDITION TESTS (UNREDUCED COORDINATES)
//////////////////////////////////////////////////////////////*/

/// @notice PRECONDITION TEST: Demonstrates correct behavior with reduced
coordinates
function test_Precondition_ReducedCoordinates() pure public {
  // Use a small prime: p = 7, curve parameter a = 2
  uint256 pp = 7;
  uint256 aa = 2;

  // Point P: (3, 5), Point Q: (3, 2) where Q is inverse of P since 2 + 5 = 7 ≡ 0
  (mod 7)
  (uint256 rx, uint256 ry) = EllipticCurve.ecAdd(3, 5, 3, 2, aa, pp);

  // P + (-P) should equal point at infinity
  assertTrue(isPointAtInfinity(rx, ry),
    "With REDUCED coordinates: (3,5) + (3,2) correctly gives point at infinity");
}

/*//////////////////////////////////////////////////////////////
                    CONCRETE TESTS (SECP256K1)
//////////////////////////////////////////////////////////////*/

/// @notice Concrete: G + G = 2G
function test_Concrete_Doubling_Secp256k1() pure public {
  (uint256 rx, uint256 ry) = EllipticCurve.ecAdd(
    SECP256K1_GX, SECP256K1_GY, SECP256K1_GX, SECP256K1_GY, SECP256K1_AA,
    SECP256K1_PP
  );

  // Expected 2G
  assertEq(rx,
  0xc6047f9441ed7d6d3045406e95c07cd85c778e4b8cef3ca7abac09b95c709ee5);
  assertEq(ry,
  0x1ae168fea63dc339a3c58419466ceaeef7f632653266d0e1236431a950cfe52a);

  // Verify result is on curve
  assertTrue(EllipticCurve.isOnCurve(rx, ry, SECP256K1_AA, SECP256K1_BB,
  SECP256K1_PP));
}

/// @notice Concrete: G + O = G
function test_Concrete_Identity_Secp256k1() pure public {
  (uint256 rx, uint256 ry) = EllipticCurve.ecAdd(
    SECP256K1_GX, SECP256K1_GY, 0, 0, SECP256K1_AA, SECP256K1_PP
  );
```

```

    assertEq(rx, SECP256K1_GX);
    assertEq(ry, SECP256K1_GY);
  }

  /// @notice Concrete: G + (-G) = 0
  function test_Concrete_Inverse_Secp256k1() pure public {
    (uint256 invX, uint256 invY) = EllipticCurve.ecInv(SECP256K1_GX, SECP256K1_GY,
    SECP256K1_PP);

    (uint256 rx, uint256 ry) = EllipticCurve.ecAdd(
      SECP256K1_GX, SECP256K1_GY, invX, invY, SECP256K1_AA, SECP256K1_PP
    );

    assertTrue(isPointAtInfinity(rx, ry), "G + (-G) should be point at infinity");
  }

  /// @notice Concrete: (G + 2G) + 3G = G + (2G + 3G) = 6G
  function test_Concrete_Associativity_Secp256k1() pure public {
    // Compute 2G, 3G
    (uint256 g2x, uint256 g2y) = EllipticCurve.ecMul(2, SECP256K1_GX, SECP256K1_GY,
    SECP256K1_AA, SECP256K1_PP);
    (uint256 g3x, uint256 g3y) = EllipticCurve.ecMul(3, SECP256K1_GX, SECP256K1_GY,
    SECP256K1_AA, SECP256K1_PP);

    // Compute (G + 2G) + 3G
    (uint256 g_2g_x, uint256 g_2g_y) = EllipticCurve.ecAdd(
      SECP256K1_GX, SECP256K1_GY, g2x, g2y, SECP256K1_AA, SECP256K1_PP
    );
    (uint256 result1X, uint256 result1Y) = EllipticCurve.ecAdd(
      g_2g_x, g_2g_y, g3x, g3y, SECP256K1_AA, SECP256K1_PP
    );

    // Compute G + (2G + 3G)
    (uint256 g2_3g_x, uint256 g2_3g_y) = EllipticCurve.ecAdd(
      g2x, g2y, g3x, g3y, SECP256K1_AA, SECP256K1_PP
    );
    (uint256 result2X, uint256 result2Y) = EllipticCurve.ecAdd(
      SECP256K1_GX, SECP256K1_GY, g2_3g_x, g2_3g_y, SECP256K1_AA, SECP256K1_PP
    );

    // Both should equal 6G
    (uint256 g6x, uint256 g6y) = EllipticCurve.ecMul(6, SECP256K1_GX, SECP256K1_GY,
    SECP256K1_AA, SECP256K1_PP);

    assertEq(result1X, result2X, "Associativity: (G+2G)+3G = G+(2G+3G)");
    assertEq(result1Y, result2Y, "Associativity: (G+2G)+3G = G+(2G+3G)");
    assertEq(result1X, g6x, "Result should be 6G");
    assertEq(result1Y, g6y, "Result should be 6G");
  }
}
```

Appendix: Vulnerability Classification

For classifying vulnerabilities identified in the findings of this report, we employ the simplified version of Common Vulnerability Scoring System (CVSS) v3.1, which is an industry standard vulnerability metric. For each identified vulnerability we assess the scores from the *Base Metric Group*, the Impact score, and the Exploitability score. The *Exploitability score* reflects the ease and technical means by which the vulnerability can be exploited. That is, it represents characteristics of the *thing that is vulnerable*, which we refer to formally as the *vulnerable component*. The *Impact score* reflects the direct consequence of a successful exploit, and represents the consequence to the *thing that suffers the impact*, which we refer to formally as the *impacted component*. In order to ease score understanding, we employ CVSS Qualitative Severity Rating Scale, and abstract numerical scores into the textual representation; we construct the final *Severity score* based on the combination of the Impact and Exploitability sub-scores.

As blockchains are a fast evolving field, we evaluate the scores not only for the present state of the system, but also for the state that deems achievable within 1 year of projected system evolution. E.g., if at present the system interacts with 1-2 other blockchains, but plans to expand interaction to 10-20 within the next year, we evaluate the impact, exploitability, and severity scores wrt. the latter state, in order to give the system designers better understanding of the vulnerabilities that need to be addressed in the near future.

Impact Score

The Impact score captures the effects of a successfully exploited vulnerability on the component that suffers the worst outcome that is most directly and predictably associated with the attack.

Impact Score	Examples
🟠 **High**	Halting of the chain; loss, locking, or unauthorized withdrawal of funds of many users; arbitrary transaction execution; forging of user messages / circumvention of authorization logic
🟡 **Medium**	Temporary denial of service / substantial unexpected delays in processing user requests (e.g. many hours/days); loss, locking, or unauthorized withdrawal of funds of a single user / few users; failures during transaction execution (e.g. out of gas errors); substantial increase in node computational requirements (e.g. 10x)
🟢 **Low**	Transient unexpected delays in processing user requests (e.g. minutes/a few hours); Medium increase in node computational requirements (e.g. 2x); any kind of problem that affects end users, but can be repaired by manual intervention (e.g. a special transaction)
🔵 **None**	Small increase in node computational requirements (e.g. 20%); code inefficiencies; bad code practices; lack/incompleteness of tests; lack/incompleteness of documentation

Exploitability Score

The Exploitability score reflects the ease and technical means by which the vulnerability can be exploited; it represents the characteristics of the vulnerable component. In the below table we list, for each category, examples of actions by actors that are enough to trigger the exploit. In the examples below:

- *Actors* can be any entity that interacts with the system: other blockchains, system users, validators, relayers, but also uncontrollable phenomena (e.g. network delays or partitions).
- *Actions* can be
  - *legitimate*, e.g. submission of a transaction that follows protocol rules by a user; delegation/redelegation/bonding/unbonding; validator downtime; validator voting on a single, but alternative block; delays in relaying certain messages, or speeding up relaying other messages;
  - *illegitimate*, e.g. submission of a specially crafted transaction (not following the protocol, or e.g. with large/incorrect values); voting on two different alternative blocks; alteration of relayed messages.
- We employ also a *qualitative measure* representing the amount of certain class of power (e.g. possessed tokens, validator power, relayed messages): *small* for < 3%; *medium* for 3-10%; *large* for

10-33%, *all* for >33%. We further quantify this qualitative measure as relative to the largest of the system components. (e.g. when two blockchains are interacting, one with a large capitalization, and another with a small capitalization, we employ *small* wrt. the number of tokens held, if it is small wrt. the large blockchain, even if it is large wrt. the small blockchain)

Exploitability Score	Examples
🟠 **High**	illegitimate actions taken by a small group of actors; possibly coordinated with legitimate actions taken by a medium group of actors
🟡 **Medium**	illegitimate actions taken by a medium group of actors; possibly coordinated with legitimate actions taken by a large group of actors
🟢 **Low**	illegitimate actions taken by a large group of actors; possibly coordinated with legitimate actions taken by all actors
🔵 **None**	illegitimate actions taken in a coordinated fashion by all actors

Severity Score

The severity score combines the above two sub-scores into a single value, and roughly represents the probability of the system suffering a severe impact with time; thus it also represents the measure of the urgency or order in which vulnerabilities need to be addressed. We assess the severity according to the combination scheme represented graphically below.

As can be seen from the image above, only a combination of high impact with high exploitability results in a Critical severity score; such vulnerabilities need to be addressed ASAP. Accordingly, High severity score receive vulnerabilities with the combination of high impact and medium exploitability, or medium impact, but high exploitability.

Severity Score	Examples
🔴 Critical	Halting of chain via a submission of a specially crafted transaction
🟠 High	Permanent loss of user funds via a combination of submitting a specially crafted transaction with delaying of certain messages by a large portion of relayers
🟡 Medium	Substantial unexpected delays in processing user requests via a combination of delaying of certain messages by a large group of relayers with coordinated withdrawal of funds by a large group of users
🟢 Low	2x increase in node computational requirements via coordinated withdrawal of all user tokens
🔵 Informational	Code inefficiencies; bad code practices; lack/incompleteness of tests; lack/incompleteness of documentation; any exploit for which a coordinated illegitimate action of all actors is necessary

Disclaimer