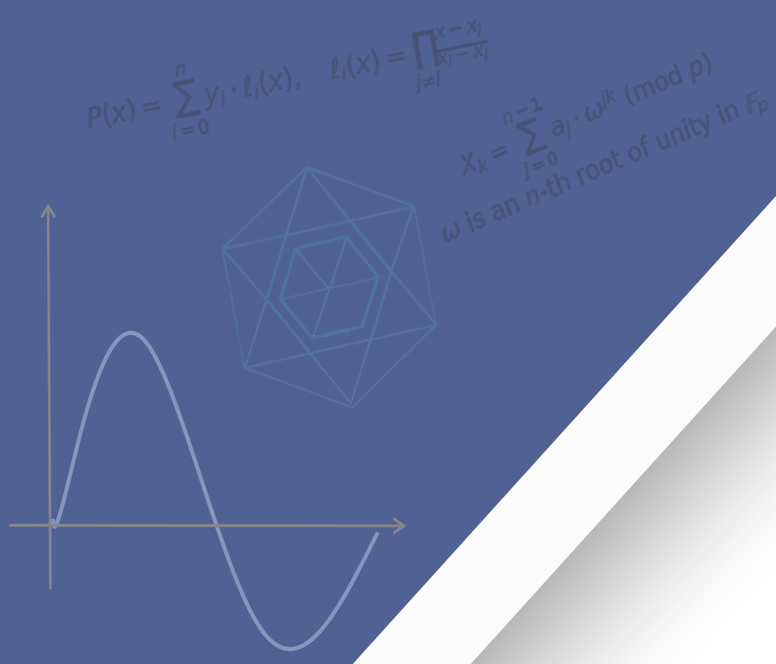


Security Review Report

NM-0677 - Anoma Resource State Machine

November 13, 2025

NETHERMIND SECURITY

Contents

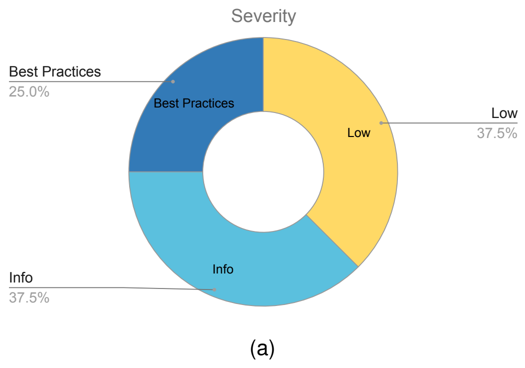
1 Executive Summary

This document presents the results of the security review conducted by Nethermind Security for Anoma, including Resource Machine and EVM Protocol Adapter.

This audit focused on the end-to-end review of the above-described changes. **The audit comprises 789** lines of Solidity and **2085** Rust code. The audit was performed using (a) manual analysis of the codebase, and (b) automated analysis tools.

Along this document, we report eight points of attention, where three are classified as `Low` and five are classified as `Informational` or `Best Practices`. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology. Section 6 details the security approach. Section 7 details the issues. Section 8 discusses the documentation provided by the client for this audit. Section 9 presents the test suite evaluation and automated tools used. Section 10 concludes the document.



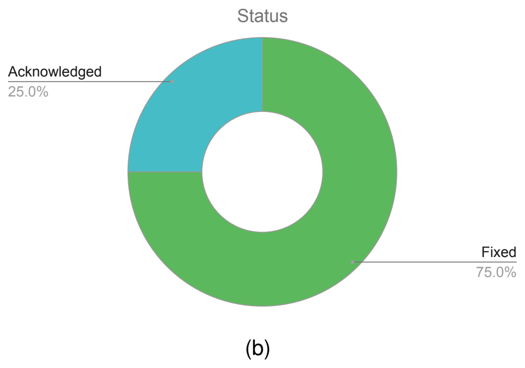


Fig. 1: Distribution of issues: Critical (0), **High** (0), **Medium** (0), **Low** (3), **Undetermined** (0), **Informational** (3), **Best Practices** (2).
Distribution of status: Fixed (6), **Acknowledged** (2), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	November 10, 2025
Final Report	November 13, 2025
Initial Protocol Adapter Commit Hash	`e6cfdf8fabe003c727c7c85dd99a993ac4111744`
Initial ARM Commit Hash	`a0cca9cdc8e87508b97f6afc65a3b7582aa3e59d`
Final Protocol Adapter Commit Hash	`fee4f47050689b82473e9a3198e7a2065becb3fb`
Final ARM Commit Hash	`087e7d05f6b7f5a961ea4197d6be5615aea85343`
Documentation Assessment	High
Test Suite Assessment	High

2 Audited Files

2.1 EVM Protocol Adapter

	Contract	LoC	Comments	Ratio	Blank	Total
1	ProtocolAdapter.sol	268	117	43.66%	57	442
2	libs/MerkleTree.sol	103	68	66.02%	32	203
3	state/CommitmentTree.sol	88	33	37.50%	23	144
4	libs/RiscZeroUtils.sol	78	50	64.10%	14	142
5	libs/proving/Logic.sol	51	35	68.63%	8	94
6	libs/proving/Delta.sol	34	33	97.06%	14	81
7	interfaces/IProtocolAdapter.sol	16	43	268.75%	13	72
8	libs/TagUtils.sol	30	17	56.67%	10	57
9	interfaces/ICommitmentTree.sol	17	28	164.71%	9	54
10	libs/proving/Compliance.sol	23	26	113.04%	5	54
11	state/NullifierSet.sol	23	12	52.17%	9	44
12	Types.sol	22	18	81.82%	4	44
13	libs/SHA256.sol	10	14	140.00%	3	27
14	libs/proving/Aggregation.sol	11	12	109.09%	3	26
15	interfaces/INullifierSet.sol	6	13	216.67%	3	22
16	interfaces/IForwarder.sol	4	11	275.00%	1	16
17	libs/Versioning.sol	5	7	140.00%	2	14
Total		**789**	**537**	**68.06%**	**210**	**1536**

The scope includes a function EllipticCurve.ecAdd() from the external library elliptic-curve-solidity, which was not written by the Anoma team. This function was included in the security review since the entire library was never audited.

2.2 Anoma Resource Machine Circuits

	Contract	LoC	Comments	Ratio	Blank	Total
1	resource.rs	213	38	17.84%	22	273
2	compliance.rs	220	15	6.82%	31	266
3	encryption.rs	150	16	10.67%	35	201
4	logic_proof.rs	173	0	0.00%	26	199
5	delta_proof.rs	164	4	2.44%	26	194
6	transaction.rs	149	15	10.07%	15	179
7	aggregation/mod.rs	132	8	6.06%	20	160
8	aggregation/batch.rs	123	7	5.69%	20	150
9	authorization.rs	111	2	1.80%	22	135
10	action.rs	95	4	4.21%	15	114
11	action_tree.rs	70	15	21.43%	8	93
12	proving_system.rs	74	5	6.76%	11	90
13	error.rs	70	0	0.00%	1	71
14	nullifier_key.rs	52	3	5.77%	11	66
15	resource_logic.rs	51	5	9.80%	8	64
16	utils.rs	48	0	0.00%	7	55
17	logic_instance.rs	47	0	0.00%	8	55
18	merkle_path.rs	43	3	6.98%	7	53
19	compliance_unit.rs	39	3	7.69%	5	47
20	lib.rs	37	0	0.00%	1	38
21	constants.rs	13	4	30.77%	3	20
22	aggregation/constants.rs	11	4	36.36%	3	18
Total		**2085**	**151**	**7.24%**	**305**	**2541**

3 Summary of Issues

	Finding	Severity	Update
1	Check a point is on the curve	Low	Fixed
2	ECDSA signature creation and verification	Low	Fixed
3	Update `rand_seed` to stronger randomness guarantees	Low	Fixed
4	Privacy Leakage through imageID usage	Info	Acknowledged
5	Differences between the specification and implementation	Info	Acknowledged
6	The function `verifyMerkleProof()` does not distinguish between the leaves and nodes	Info	Fixed
7	Should constrain logic instances' length to match the number of keys in the batch circuit	Best Practices	Fixed
8	Remove debug trait for secret key	Best Practices	Fixed

4 System Overview

4.1 General

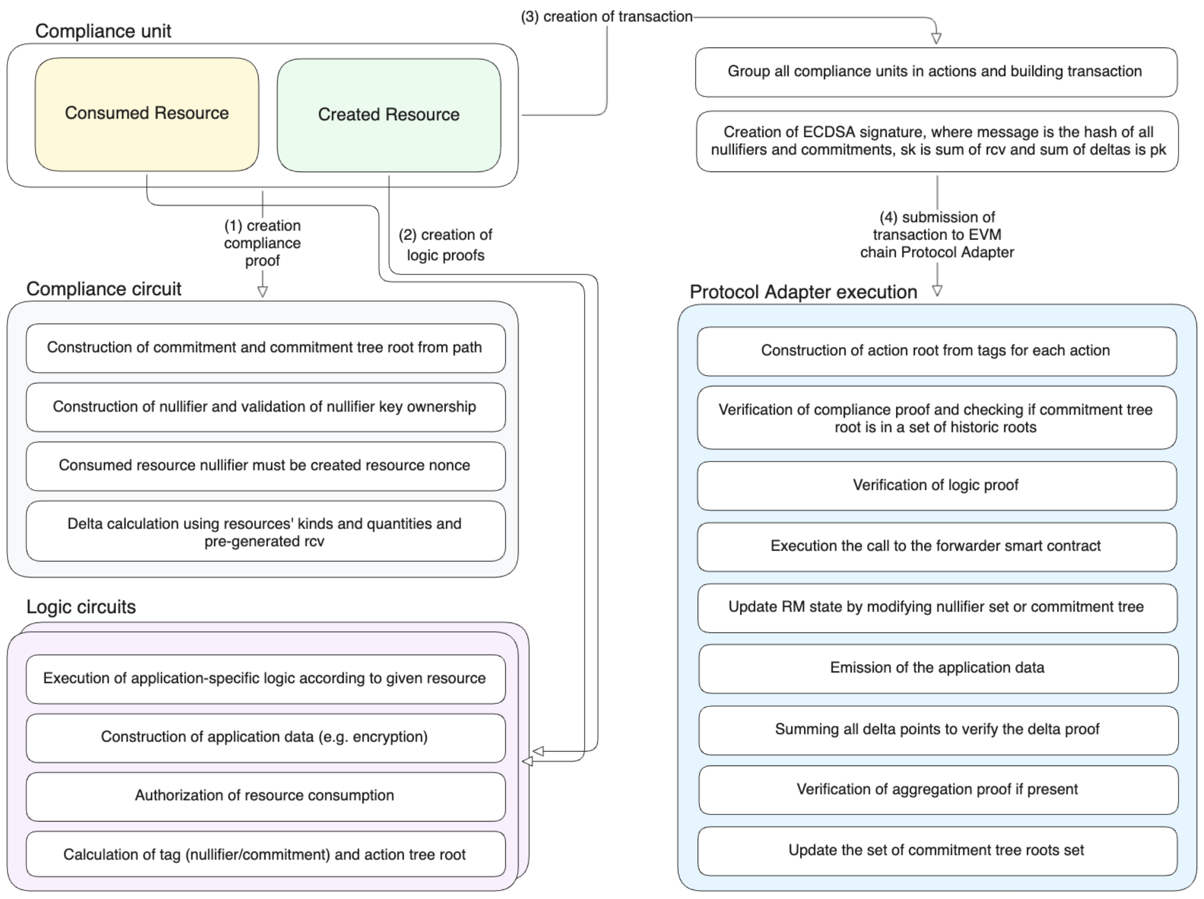
Anoma's Resource Machine (ARM) defines what resources and transactions are, how they compose, and how validity is proven — supporting an intent-centric, privacy-preserving model where users express goals and settlement proves correctness without revealing sensitive data. The ARM specification frames intents, transactions, and proofs, and places the Resource Machine as a core protocol component for programmable, shielded state updates.

The arm-risc0 codebase is a concrete implementation of this Resource Machine that runs ARM logic inside a zkVM. It provides full transaction processing, plus circuits and helpers for resources, transactions, Merkle paths, nullifier keys, compliance checks, and batching/aggregation.

On the settlement side, the EVM Protocol Adapter is a Solidity suite that lets EVM chains verify ARM executions and update on-chain state via commitments. The adapter's purpose is to enable ARM transaction settlement on EVM-compatible chains.

Proof verification is designed to rely on the RISC Zero Ethereum verifier contracts, so an ARM execution proven in the RISC0 zkVM can be checked on EVM.

Taking all into account, users produce intents that compile to ARM transactions; the ARM circuits compute state transitions and generate (optionally aggregated) ZK proofs; the EVM Protocol Adapter receives the transaction and verifies proofs on-chain. Fig. 2 shows the flow of execution.



(3) creation of transaction
Compliance unit
Consumed Resource
Created Resource
Group all compliance units in actions and building transaction
Creation of ECDSA signature, where message is the hash of all nullifiers and commitments, sk is sum of rcv and sum of deltas is pk
(1) creation compliance proof
(2) creation of logic proofs
(4) submission of transaction to EVM chain Protocol Adapter
Compliance circuit
Construction of commitment and commitment tree root from path
Construction of nullifier and validation of nullifier key ownership
Consumed resource nullifier must be created resource nonce
Delta calculation using resources' kinds and quantities and pre-generated rcv
Logic circuits
Execution of application-specific logic according to given resource
Construction of application data (e.g. encryption)
Authorization of resource consumption
Calculation of tag (nullifier/commitment) and action tree root
Protocol Adapter execution
Construction of action root from tags for each action
Verification of compliance proof and checking if commitment tree root is in a set of historic roots
Verification of logic proof
Execution the call to the forwarder smart contract
Update RM state by modifying nullifier set or commitment tree
Emission of the application data
Summing all delta points to verify the delta proof
Verification of aggregation proof if present
Update the set of commitment tree roots set

Fig. 2: Flow of execution. Presents actions performed to create and execute a valid Anoma transaction on an EVM-compatible chain.

4.2 Resources

Resources can be thought of as "units of state", they represent objects that can be created and consumed during state transitions — e.g., tokens, NFTs, rights, or any other application-specific object. A Resource carries both data and its own rules for how it may be consumed. Each Resource specifies the logic that governs its life-cycle via a reference to a resource-logic program. A Resource has the following components:

- `logic_ref` — hash of the logic program that defines how this resource may be consumed or created

- label_ref — identifies the kind/class of resource
- quantity — amount of this resource
- value_ref — application-specific payload
- is_ephemeral — marks resources that do not need nullifier balancing checks
- nonce — uniqueness salt for otherwise identical resources
- nk_commitment — commitment derived from the nullifier key used to prevent double-consumption
- rand_seed — randomness used for commitment and nullifier generation

Each Resource, therefore, contains both what it is (quantity) and how it behaves (logic). Through its logic_ref, the Resource enforces constraints on whether it may be consumed, and under which conditions new Resources may be created.

4.3 Compliance Unit

A compliance unit is the part of the transaction logic that proves that a consumed resource was valid and that it is correctly linked to a newly created resource. It serves as the bridge between the previous state and the new state, and it is enforced through a zero-knowledge proof.

A compliance unit contains

- proof the ZK proof attesting correctness
- instance the public inputs bound to that proof

The witness inputs to this ZK proof are:

- comsumed_resource
- merkle_path
- ephemeral_root
- nf_key
- created_resource
- rcv

The proof checks that the consumed resource was present in the commitment tree (via its Merkle path or ephemeral root), that the correct nullifier key is used to consume it, and that the newly created resource is consistent with the referenced logic program. It also calculates the “delta” curve point: consumed kind times its quantity, minus created kind times its quantity, plus a generator multiple of the random scalar. That delta captures the net change in fungible balance.

4.4 Action

The Action groups of compliance units. This allows for gathering consumed and created resources that participate in the same application, but can contain different logics. The action is represented as a Merkle Tree, where leaves are nullifiers and commitments. The action root is recreated during the execution in the Protocol Adapter to ensure the correct content of the action.

```
struct Action {
    Logic.VerifierInput[] logicVerifierInputs;
    Compliance.VerifierInput[] complianceVerifierInputs;
}
```

4.5 Transaction

Transaction consists of Actions, delta proof and aggregation proof (if used). The delta proof guarantees that all the resources in the transaction are balanced. The transaction is submitted to the protocol adapter, as illustrated in Fig. 3.

```
struct Transaction {
    Action[] actions;
    bytes deltaProof;
    bytes aggregationProof;
}
```

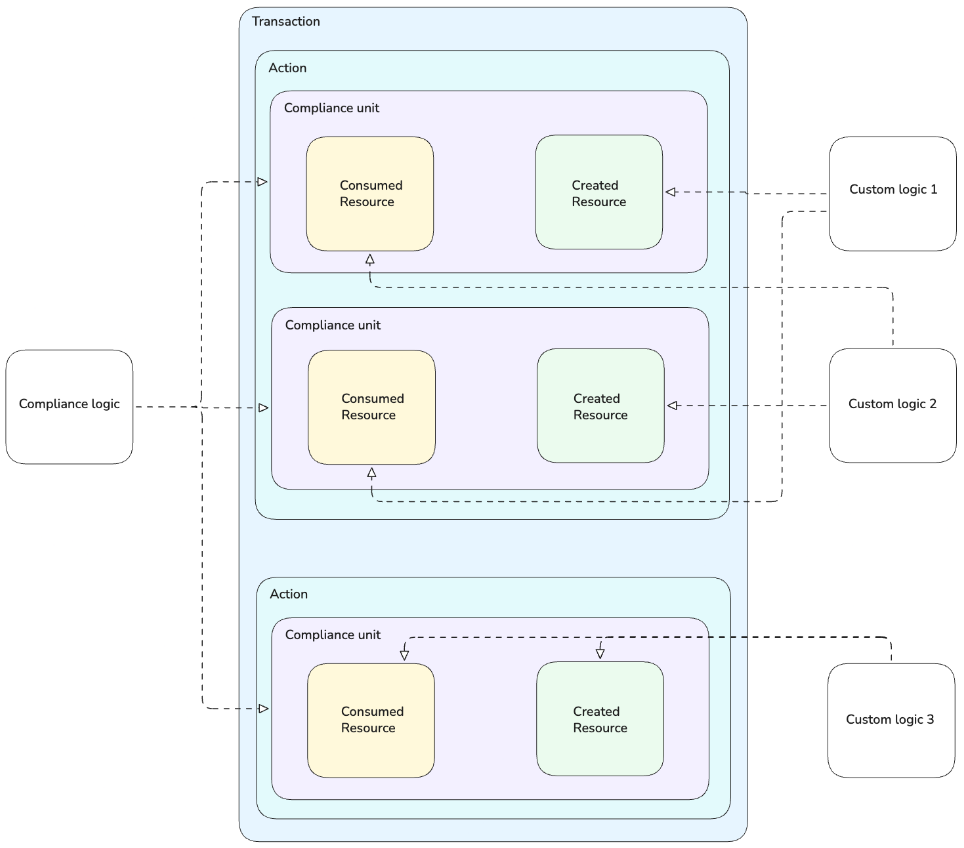


Fig. 3: Transaction. Transaction's structure with example logic association.

4.6 ZK proofs

Built using the Risc-Zero zkVM, Anoma uses two different ZK proofs to prove state transitions to establish the correct consumption and creation of resources, called the compliance proofs and logic proofs. Additionally, for efficiency reasons, it also provides an in-built aggregation method to verify all the proofs in one.

Compliance proofs are a fixed proof defined by Anoma that proves that the nullifier of the resource consumed is equal to the nonce of the created resource. Additionally, it computes the delta value of the created and computed resources, following the style of a Pedersen commitment, whereby Δ = consumed kind$*$consumed quantity$-$created kind$*$created quantity$+G*$rcv where G is the generator on the curve and rcv is a random scalar generated. A resource kind is the hash to curve(logic reference||label reference) where the hash used is SHA256 and the curve is Secp256k1. The public values of the circuit are:

- Consumed resource's nullifier.
- Consumed resource's logic reference.
- Consumed commitment resource's tree root (if the resource is ephemeral, this value is arbitrary).
- Created resource's commitment.
- Created resource's logic reference.
- The delta as (x, y) coordinates.

These proofs convince the verifier that a user has correctly consumed and created a new resource.

Logic Proofs are user-defined proofs that outline how specific resources can or cannot be consumed or created. More specific details about writing these circuits will be discussed in Section 4.10.

The final proof circuit is the **Batch Aggregation Proof**, which, as the name suggests, proves that all these provided proofs verify under these provided keys. Split between compliance and logic proofs, it takes as input the journal instances and the keys/imageIDs (for compliance, it is a fixed key as the circuit does not change) to verify. Calling the in-built verify function, the proofs can be verified in sequence: first, all the compliance proofs are verified, then all the logic proofs are verified. Finally, all the instances and keys input are committed as public values.

4.7 Delta proof

The delta proof is the signature (and recovery ID) of all the deltas in a transaction. Formally, it can be expressed as Sign$(H(\text{Vec}(\Delta))_{\text{rcv}} \rightarrow \sigma = (r, s)$, recID where the signature attests to the integrity and authenticity of the aggregated deltas.

To enforce the transaction being balanced, meaning the total amount created of a resource must equal the total amount consumed, the sum of all delta values must correspond to the aggregate randomness rcv. When a transaction is submitted on-chain, this condition is verified by recovering the public key from the signature rcv and comparing it to the expected public key derived from the summed delta values. Should the value not recover to the aggregate sum of the rcv's in the transaction, it means there is an imbalance.

4.8 EVM Protocol Adapter

The Protocol Adapter (PA) allows the Anoma applications to be run on existing chains. It processes the transactions created in the Anoma Resource Machine (ARM) and updates the state according to the provided resources. In this iteration of the protocol, the PA is implemented as a Solidity smart contract and can be utilised as a connector to any EVM-compatible chain. The entrypoint is the function `execute()` with input being the `Transaction`. The `execute()` function iterates over provided `Action` objects, computes the `Action` root from collected nullifiers and commitments (tags), and for each `Compliance Unit` performs the following actions:

- Checking if the provided root corresponding to the consumed resource commitment exists in the set of historic commitment tree roots.
- Verification of the compliance proof or collecting instance for aggregation proof.
- Verification of the logic proof or collecting an instance for aggregation proof of both consumed and created resources.
- Execution external call to the `Forwarder` smart contract with provided calldata.
- Addition of the nullifier/commitment to the set of tags.
- Addition of the logic reference to the set of verification keys, required for aggregation proofs.
- Updating the accumulator with nullifier/commitment.
- Emission of the application data.
- Addition delta points to the delta accumulator.

Next, the delta proof is verified. The signature is utilised to recover the `Keccak-256` hash of tags, which is compared against the hash of the provided public key (instance). If used, the aggregation proof is checked. If all proofs are verified correctly and the external call was executed successfully with the expected output, the new commitment root is added to the set of historic roots.

Commitment accumulator is a Merkle Tree with variable size. The leaves contain the resource commitment. The tree is expanded with empty leaves if the full capacity is reached. Roots of the tree are stored in the `_roots` mapping, which allows for proving commitment against historic tree state. This accumulator allows for duplicated values of leaves.

Nullifier set contains the resource nullifier. It prevents duplicates.

4.9 Fowarder

The `Forwarder` is a smart contract created by the application developer. It should be callable only from Anoma's protocol adapter and only with the calldata included in the resource linked to particular logic. The forwarder can contain arbitrary logic and is used to interact with an external contract.

4.10 Applications on Anoma

Users create applications on Anoma by developing logic circuits and deploying a forwarder contract on the desired blockchain. Logic consists of constraints that must be satisfied during proof creation. Logic constraints allow for defining:

- Application logic - all the flows and requirements of the application must be included in the logic circuit. This should handle different resource types' consumption and creation or validation of calldata used to interact with the forwarder.
- Resources behavior - the resources can be ephemeral and non-ephemeral. The ephemeral resources do not require a valid Merkle proof to be consumed, therefore, they should be handled properly. The logic of one resource may constrain another type of resource, with particular logic must be present in a particular action.
- Ownership of the resource - the basic ownership check is enforced by the nullifier commitment, to consume a resource, the creator of a proof needs to hold the nullifier key. If the application needs delegation of proof creation, additional mechanisms, like the owner's signature, can be introduced in the logic.
- Migration strategy - in case of pausing Anoma protocol, the applications developer can migrate to the new instance by augmenting the logic with migration operations.

5 Risk Rating Methodology

The risk rating methodology used by Nethermind Security follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood measures how likely the finding is to be uncovered and exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding, other factors are also considered. These can include but are not limited to motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if an attacker exploits the finding. This factor will be one of the following values:

a) **High**: The issue can cause significant damage, such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage, such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage, such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding, other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind Security also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to pass to the client formally;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

6 Security Approach

This section outlines the approach the Nethermind Security team performed to evaluate the security of the Anoma Resource Machine protocol, outlining the properties checked.

Additionally, the assumptions about the system are provided, taking into account when assessing the protocol and the limitations of the security review.

6.1 Assumptions

The assumptions used for the security audit were as follows:

- All external, out-of-scope components, including cryptographic libraries and the Risc Zero framework, were verified to function correctly.
- The secure implementation of resource logic for all resource types (except padding) is the responsibility of the Anoma application builders.
- Fee handling mechanisms and any associated risks are managed by the application builders.
- The implementation of third-party solvers and executors follows secure and fair design principles.
- Forwarders have been implemented correctly, ensuring reliable message and data transmission.
- zkVM proofs are correctly and securely processed and return valid cryptographic proofs, maintaining integrity and confidentiality throughout the proof lifecycle.

6.2 Limitations

During this security review, we were unable to assess:

- The impact of potential MEV scenarios available for solvers, transaction builders, and executors.
- Estimate the probability of pairing the nullifier and commitments.

6.3 Resources

The following properties were assessed with regard to the security and validity of the resource objects:

- The computation of commitment (Resource::commit()) and nullifier (Resource::nullifier_from_commitment()) is done in a secure way and results in unique values. The nullifier hash correctly encapsulates the resource commitment such that it does not allow two values to map to the same hash.
- The Resource::kind(...) is correctly calculated from the resource's logic reference and label. It is also correctly represented on the curve.
- Neither the commitment nor the nullifier reveals any information about the contents of a resource.
- The nullifier and commitment cannot be linked (assuming lack of access to global state).
- A resource can only be consumed by the owner of the nullifier key to create a nullifier.
- Evaluated `rand_seed` value generation using `thread_rng()`, which is used in both nullifier and commitment creation.

6.4 Compliance Unit

The following properties were assessed with regard to the security and validity of the resource objects:

- Correctly computes consumed resource's commitment and nullifier and created resource's commitment.
- The nullifier can only be computed if the prover holds the nullifier key.
- The created resource and consumed resource are linked by the nullifier used as the nonce, which enforces creation of each resource by consumption of a single unique resource.
- The delta is correctly calculated (ComplianceWitness::delta()) by correctly using the resources' kinds, quantities, and random scalar.
- The commitment tree root is correctly constructed from the path and consumed resource's commitment (ComplianceWitness::consumed_commitment_tree_root()) and the ephemeral resource contains any root.
- The logic references for each resource are included in the instance.
- The instance does not reveal any additional information, e.g., information about the connection between nullifier and commitment.

- Verification of compliance proof is performed correctly both in ComplianceUnit::verify() and ProtocolAdapter._process-Compliance(). Only case in which such proof won't be verified is if the Risc0 verifying router stops supporting the version for which the proof was generated. Such a scenario is handled by Anoma.

6.5 Action

The properties checked for actions were:

- Correct root generation with MerkleTree::root().
- Correct path generation for each leaf with MerkleTree::generate_path().

6.6 Transaction

The properties checked for transactions were:

- Verified that all transactions are balanced, ensuring proper enforcement of this constraint.
- Confirmed that the composition of two transactions correctly aggregates their respective delta proofs.
- Validated that the batch aggregation proof appropriately constrains and verifies the input proofs, and correctly commits to the intended public values.
- Assessed on-chain transaction execution to ensure:
  - Correct verification of the delta proof.
  - Accurate validation of the action root against the submitted actions.
  - Proper distinction between batch and non-batch proofs for verification.
  - Prevention of replay attacks by disallowing resubmission of old transactions or nullifiers.

6.7 Delta Proofs

For the delta proof, the following was checked:

- Verified that `Resource::kind()` (consisting of the hash of the logic reference and label) is correctly projected onto the curve.
- Confirmed that all performed calculations are consistent with elliptic curve (EC) operations.
- Verified that each transaction is balanced by ensuring equality between quantities of the same kind within the transaction.
- Ensured that it is not possible to craft a special quantity value that could artificially balance a transaction.
- Similarly confirmed that crafting a special `rcv` value to artificially balance a transaction is not possible.
- Validated correct delta accumulation in `Action::delta()`, `Transaction::delta()`, and in the Protocol Adapter at `vars.transactionDelta.add()`.
- Reviewed signature creation in `DeltaProof::prove()`, verifying that the public key is computed as the sum of deltas, the secret key as the sum of `rcv` values, and the signed message as the hash of all nullifiers and commitments.
- Verified signature validation in `DeltaProof::verify()` and in the Protocol Adapter's `Delta.verify()` implementation.
- Confirmed that the set of signed tags cannot be modified in a way that would produce a different or unauthorized set of resources.
- The signature is created correctly and is compatible between RM and PA.

6.8 Protocol Adapter

The following properties were assessed in the Protocol Adapter:

- Correct invocation of proof verification: `_processCompliance()` and `_processLogic()` call into the Risc Zero router with the exact public instance values reconstructed from calldata.
- Correctness of delta accumulation via transactionDelta.add() and that the delta used for Delta.verify is identical to the delta enforced when updating state.
- State-transition side effects (nullifier storage, commitment insertion) only occur after a successful verification step
- Confirm emit logic only exposes intended payloads, respects deletion criteria, and doesn't leak sensitive state.
- The execution is only correct if all proofs are valid and the on-chain call is successful.

6.9 Attack Scenarios Considered

In addition to ensuring the above properties of each component of Anoma, the review considered the following potential attack vectors:

- Resources
  - Can produce two different commitments or nullifiers for the same resource.
  - Two different commitments map to the same hash
  - A deterministic RNG could potentially break the security of a secret.
- Compliance Unit
  - Taking advantage of a badly designed delta to forge quantities of resources.
  - Proving the existence of an insecure Merkle tree implementation to include fake resources.
- Actions:
  - Providing an incorrect root in order to prove the inclusion of a non-existent leaf in the action tree.
  - Providing a resource outside of action.
- Transactions:
  - Resubmitting a transaction to inflate the number of commitments.
  - Providing an unbalanced transaction enabling malicious growth of the resource number owned by a user.
  - Anonymity of the user submitting the transaction is revealed.
  - Resubmitting a transaction to inflate the number of commitments.
  - Providing an unbalanced transaction enabling malicious growth of the resource number owned by a user.
  - Anonymity of the user submitting the transaction.
- Batch Transactions:
  - Submit two different proofs for the same transaction.
  - Provide mismatching proof instances & logic verifying keys.
- Delta Proofs:
  - Could reuse the same rcv value across multiple proofs, leaking the corresponding private key material.
  - Could create a malleable signature that passes verification but represents a different message hash.
  - Could reorder actions or transactions to exploit non-commutative accumulation, producing divergent final deltas.
- Protocol Adapter:
  - Updating state without all proofs verification.
  - Front-runs and reentrancy.
  - Finalising execution without expected external call output.
  - Updating incorrectly the commitment and nullifier accumulators.
  - Updating historic roots set with incorrect root.
  - Using an incorrect verifier.
  - Using migration to process to create incorrect state updates.
- Privacy Properties
  - Revealing the user's identity with available data.
  - Revealing data in the resource.
  - Detecting a connection between the nullifier. and commitment to the same resource, given the public data.

7 Issues

7.1 [Low] Check a point is on the curve

File(s): `contracts/src/libs/proving/Delta.sol`

Description: The function `ecAdd(...)` uses functions `jacAdd(...)` and `jacDouble(...)` to add points on the elliptic curve, secp256k1 specifically. However, inputs to these functions are of type `uint256`, which do not assert the validity that these points are initially on the curve and Jacobian coordinates mapping, as well as the additional formulas that assume that the points are already on the curve.

Moreover, `isOnCurve(...)` function won't validate the point at infinity " `(0, 0)`", so this case must be handled separately.

Recommendation(s): Check that points are on the Curve in `add(...)`.

Status: Fixed

Update from the client: Added in protocol adapter PR #396.

Update from Nethermind team: We note that this fix assumes the first elliptic-curve point used is already on the curve, as is also mentioned in the function's comment. Since the resulting point would—with high probability—fall outside the curve.

7.2 [Low] ECDSA signature creation and verification

File(s): `arm/src/delta_proof.rs`, `contracts/src/libs/proving/Delta.sol`

Description: The Anoma Resource Machine implements the `DeltaProof::verify()` function, which differs from the `Delta.verify()` in the Ethereum Protocol Adapter. Below we list the differences:

- The ARM `DeltaProof::verify()` accepts the malleable signatures, where the `s > secp256k1n ÷ 2 + 1`, while the Protocol Adapter `Delta.verify()` uses the OpenZeppelin library that prevents malleability by ensuring the `s` is less than half of the curve order;
- The ARM `DeltaProof::verify()` handles the cases where `recovery ID` is equal to `2` or `3`, which represent the overflow of `r`. Such cases are unlikely to happen, but the on-chain ECDSA verification `ecrecover()` does not handle such signatures, since EVM treats them as invalid (Ethereum Yellow Paper Appendix F);

Moreover, the ARM's `DeltaProof::prove()` function utilises the `k256` crate, which may generate a signature with the `recid` equal to `2` or `3`. Those signatures are not supported in on-chain verification and should be handled during generation. Note, however, that there is a low probability of generating such signatures.

Recommendation(s): Consider checking if the generated signature is compatible with the on-chain verification algorithm. Also consider unifying the verification function in ARM and the Ethereum Protocol Adapter.

Status: Fixed

Update from the client: Added in PRs arm-risc0 #162 and #161.

7.3 [Low] Update `rand_seed` to stronger randomness guarantees

File(s): `arm/src/resource.rs`

Description: When a resource is created, `rand::thread_rng().gen()` is used as a seed in the `psi()` function. Although thread_-rng() is initially seeded from OS entropy and is generally secure, for stronger guarantees, it's recommended to use `OsRng` directly.

Recomendations: Replace `rand_seed` initialization using `thread_rng()` with `OsRng()`.

Status: Fixed

Update from the client: Added in PR arm-risc0 #159.

7.4 [Info] Privacy Leakage through imageID usage

File(s): `contracts/src/ProtocolAdapter.sol`

Description: The imageID of a circuit is its identifier, but also its verifying key. In Anoma, users can construct their own Risc-0 circuits, known as the logic, which defines a resource's ability to be consumed, created, and how it should deal with being ephemeral.

Though the circuit itself is not necessarily public, and is up to the app how it is communicated, when the Anoma transaction is sent on-chain to add the commitments to the tree and nullify consumed resources, the imageID is used to verify the logic zk proof for each resource. This is shown in the code below:

```
// snippet from execute(...) in ProtocolAdapter.sol
// @audit the logic ref is the same as the imageID
vars = _processLogic({
    isConsumed: true,
    // The `lookup` function reverts if the nullifier is not part of the logic verifier inputs.
    input: action.logicVerifierInputs.lookup(complianceVerifierInput.instance.consumed.nullifier),
    logicRefFromComplianceUnit: complianceVerifierInput.instance.consumed.logicRef,
    actionTreeRoot: actionTreeRoot,
    vars: vars
});

// Process the logic proof of the created resource.
vars = _processLogic({
    isConsumed: false,
    // The `lookup` function reverts if the commitment is not part of the logic verifier inputs.
    input: action.logicVerifierInputs.lookup(complianceVerifierInput.instance.created.commitment),
    logicRefFromComplianceUnit: complianceVerifierInput.instance.created.logicRef,
    actionTreeRoot: actionTreeRoot,
    vars: vars
});
```

Ultimately this calls, in the function `_processLogic(...)` the following:

```
// snippet from _processLogic(...) in ProtocolAdapter.sol
// Verify the logic proof.
//slither-disable-next-line calls-loop
_TRUSTED_RISC_ZERO_VERIFIER_ROUTER.verify({
    seal: input.proof,
    imageId: logicRef,
    journalDigest: sha256(instance.toJournal())
});
```

The nullifier and its commitment should not be linkable given a large enough privacy pool, however, the usage of this imageID enables them to be linkable under certain conditions. Since it is possible for an observer to monitor the transactions on Ethereum, they can create a mapping of each imageID to its commitment/nullifier. This ultimately shrinks the privacy to pool from one with all commitments, to a subset with only the specific imageID.

This means the privacy pool can be shrunk by the other public information available and the "rules" that Anoma transactions and actions work with. More specifically:

- The Merkle root that needed to be provided for the compliance proof reveals that some nullifier corresponds to a commitment that is in this tree, not to any later, and since the tree and the commitments/leaves are public, it is possible to read them all;
- The commitment and nullifier of the same resource can't be in the same transaction, so those pairs are excluded as potential pairs given a non-ephemeral resource;
- The initial creation of resources for a particular logic does not increase the commitment privacy pool, because they are ephemeral and made for initial creation to balance the transaction;
- Knowing the circuit behind the imageID can also reveal additional information, for example, if only ephemeral resources with zero quantity perform a certain type of on-chain call or store a particular field from application data, the observer would know that those won't have nullifiers;

The simplest example is given, an existing commitment tree with a large number of commitments and a long list of nullifiers in the nullifier accumulator. If a user creates a new resource with a unique image ID. This means that an observer watching this will see a unique image ID be submitted on-chain to verify a commitment's logic zk proof. If no other resource of the same imageID is created, then an observer will know for certain which commitment corresponds to which nullifier, as the imageID will need to be used again to nullify the resource and verify its zk proof. Thus, the resources commitment and nullifier are linked.

In summary, the usage of imageID to verify the resource logic zk proof can thus be used to link together commitments and their corresponding nullifiers.

Recommendation(s): Add explicit explanation for developers that, on first usage of a specific imageID, it can cause the aforementioned problem. The solution currently, without function privacy, is to, on first usage, create many of the resources to initialise the privacy pool.

Status: Acknowledged

Update from the client: Planning on updating specification.

7.5 [Info] Differences between the specification and implementation

File(s): `*`

Description: The specification that describes the Anoma RM components differs from the implementation. Below we list the identified differences:

Delta Proving System:

- Described proof aggregation is not implemented
- The description of proving does not match the implementation, since the proving function takes a message and a witness

Commitment accumulator:

- The described types are not implemented: `Accumulator`, `AccumulatorWitness`, `CommitmentIdentifier`, and `AccumulatedValue`
- Adding the resource commitment to the tree does not return the path
- The "witness" functionality, which finds the resource commitment in the tree and returns the path to it, is not implemented
- Names of the functions differ: `add()` is `_addCommitment()`, `verify()` is `verifyMerkleProof()` and `value()` is `latestCommitmentTreeRoot()`

Nullifier set:

- Function names are different from the description. Adding nullifier is done with `_addNullifier()` not `insert()` and checking nullifier existence is done with `isNullifierContained()` not `contains()`

Compliance Unit:

- The definition of the `ComplianceUnit` in the specification contains `VerifyingKey`, which is not present in the implementation
- The definition of `Instance` is different in the specification, since it does not include logic references
- `create()` function implementation takes only the witness, while the specification states that parameters are `ProvingKey`, `VerifyingKey`, `Instance`, `Witness`
- `verify()` does in specification returns a boolean, while the implemented function throws an error on incorrect proof
- The `created()` and `consumed()` functions are not implemented
- The `delta()` function does not return data of type `DeltaHash`

Compliance proof:

- Instance definition in the specification is different from the implementation. There are no lists of consumed and created resources. There are also no `logicVKOuter` fields. Implemented `ComplianceInstance` contains a single nullifier and commitment, logic references, consumed resource commitment tree root, and delta
- specification states that witnesses objects are defined with different fields depending on whether the resource is consumed or not, but in implementation, the witness structure is identical
- Compliance constraints section describes the CMT path validity check, but the comparison between the expected root and the one generated from the path is not done in the implementation.
- Specification describes constraints that are not implemented `logicVKOuter = logicVKOuterHash(r.logicRef, ...)`

Action:

- The implemented Action structure does not contain `actionTreeSize`
- The implemented LogicVerifierInputs structure contains `tag`, which is not present in the specification
- The above structures are implemented with different types
- There is no `create()` function that generates compliance proofs and resource logic proofs. Proofs are generated first, and an action is created with `new()`
- Lack of `to_instance()` function which is in specification

Transaction:

- Transaction structure does not contain `delta_vk`, and the types are different except for the `actions` field
- The implemented function `create()` takes `actions` and `delta` as parameters, which is different from the description. Creation of a delta proof is not done in `create()` but in separate function `generate_delta_proof()`

- The `compose()` function in the specification states that two transactions with generated delta proofs can be aggregated using their delta proofs and verifying keys. However, in the implementation, transactions are composed using delta witnesses before proof generation, using the signing key (rcv) and the set of tags
- The implementation of the `verify()` does not contain checks that do not require access to global structures described in point 2 (actions partition the state change induced by the transaction). This is addressed in v0.8.2
- Delta proof instance structure does not match implemented delta proof structures `DeltaProof` and `DeltInstance`

Transaction With Payment and TransactionFunction - it is not implemented;

Action in Protocol Adapter:

- The specification describes that the `actionTreeRoot` is computed as the root of a Merkle tree of depth 4, but in the implementation, the action tree does not have a defined depth.

Recommendation(s): Consider unifying the specification and the implementation.

Status: Acknowledged

Update from the client: Will update documentation.

7.6 [Info] The function `verifyMerkleProof()` does not distinguish between the leaves and nodes

File(s): `contracts/src/state/CommitmentTree.sol`

Description: The function `verifyMerkleProof()` in `CommitmentTree` verifies that a Merkle path and a commitment leaf reproduce a given root. However, if this function received the node as a commitment parameter and a shorter path, the proof would still be valid. This is possible, since there is no lower limit on the path size and no distinction between generating the leaf and the node during the proof check.

Recommendation(s): Consider introducing the domain separation in the commitment tree to protect against providing valid proofs for the nodes.

Status: Fixed

Update from the client: The `verifyMerkleProof()` is not used in production or in any way affects state change. However, due to the fact that this function can be misinterpreted by external parties to possess usual path-verification semantics, we have decided to delete it from production code in #392.

7.7 [Best Practice] Should constrain logic instances' length to match the number of keys in the batch circuit

File(s): `arm_circuits/batch_aggregation/methods/guest/src/main.rs`

Description: In the batch aggregation circuit, the inputs to the zkVM are read, but there is a lack of checking that the same number of keys and instances exist before beginning to verify the proofs.

Recommendation(s): It is best practice to add a constraint that logic instances length == logic keys length to ensure this is the case, and so it does not rely on the panic.

Status: Fixed

Update from the client: Added in arm risc0 PR #160.

7.8 [Best Practice] Remove debug trait for secret key

File(s): `encryption.rs`

Description: The `Debug` trait is being derived for the `SecretKey`, which makes it possible to log and therefore retain in memory the secret key of a user, potentially leaking the value. This issue arises mainly in two scenarios:

a. It could be accidentally printed during a panic crash log.
b. Any future print statements could inadvertently leak this value.

Enabling logging of this value increases the risk of exposing sensitive information such as the user's secret key. As this function may be used by the users of Anoma, it is possible that they will not run in release mode when inheriting the struct and could inadvertently leak the secret key.

Recommendation(s): Remove the `Debug` trait derivation from the `SecretKey` implementation.

Status: Fixed

Update from the client: Added in arm risc0 PR #164.

8 Documentation Evaluation

Software documentation refers to the written or visual information that describes the functionality, architecture, design, and implementation of software. It provides a comprehensive overview of the software system and helps users, developers, and stakeholders understand how the software works, how to use it, and how to maintain it. Software documentation can take different forms, such as user manuals, system manuals, technical specifications, requirements documents, design documents, and code comments. Software documentation is critical in software development, enabling effective communication between developers, testers, users, and other stakeholders. It helps to ensure that everyone involved in the development process has a shared understanding of the software system and its functionality. Moreover, software documentation can improve software maintenance by providing a clear and complete understanding of the software system, making it easier for developers to maintain, modify, and update the software over time. SDK's can use various types of software documentation. Some of the most common types include:

- Technical whitepaper: A technical whitepaper is a comprehensive document describing the SDK's design and technical details. It includes information about the purpose of the contract, its architecture, its components, and how they interact with each other;
- User manual: A user manual is a document that provides information about how to use the SDK. It includes step-by-step instructions on how to perform various tasks and explains the different features and functionalities of the contract;
- Code documentation: Code documentation is a document that provides details about the code of the SDK. It includes information about the functions, variables, and classes used in the code, as well as explanations of how they work;
- API documentation: API documentation is a document that provides information about the API (Application Programming Interface) of the SDK. It includes details about the methods, parameters, and responses that can be used to interact with the contract;
- Testing documentation: Testing documentation is a document that provides information about how the SDK was tested. It includes details about the test cases that were used, the results of the tests, and any issues that were identified during testing;
- Audit documentation: Audit documentation includes reports, notes, and other materials related to the security audit of the SDK. This type of documentation is critical in ensuring that the SDK is secure and free from vulnerabilities.

These types of documentation are essential for SDK development and maintenance. They help ensure that the contract is properly designed, implemented, and tested, and they provide a reference for developers who need to modify or maintain the contract in the future.

Remarks about Anoma Resource Machine documentation

The **Anoma Resource Machine** protocol provided detailed specification: Resource Machine and Protocol Adapter. In addition, the team provided audit-specific documentation, detailing the core components and outlining the flow step by step. Furthermore, the **Anoma** team thoroughly addressed all questions and concerns raised by the **Nethermind Security** team, providing valuable insights and a comprehensive understanding of the project's technical aspects.

9 Test Suite Evaluation

9.1 Resource Machine Tests

```
RISC0_DEV_MODE=1 cargo test
   Compiling arm v0.8.1 ($HOME/.../NM-0677-Security-Review-Anoma-ZK/arm)
    Finished `test` profile [optimized + debuginfo] target(s) in 3.45s
     Running unittests src/lib.rs ($HOME/.../NM-0677-Security-Review-Anoma-ZK/target/debug/deps/
     arm-a07de974769cf83b)

running 9 tests
test evm::encode_permit_witness_transfer_from_test ... ok
test evm::forward_call_data_test ... ok
test evm::evm_resource_test ... ok
test utils::test_bytes_to_words ... ok
test utils::test_words_to_bytes ... ok
test encryption::test_encryption ... ok
test authorization::test_authorization ... ok
test delta_proof::test_delta_proof ... ok
test logic_proof::test_padding_logic_prover ... ok

test result: ok. 9 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.04s

   Doc-tests arm

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

cargo test --release
   Compiling arm v0.8.1 ($HOME/.../NM-0677-Security-Review-Anoma-ZK/
   arm)
   Compiling counter-witness v0.1.0 ($HOME/.../NM-0677-Security-Review-Anoma-ZK/
   examples/simple_counter_application/counter_witness)
   Compiling kudo-logic-witness v0.1.0 ($HOME/.../NM-0677-Security-Review-Anoma-ZK/
   examples/kudo_application/logic_witness)
   Compiling simple-transfer-witness v0.1.0 ($HOME/.../NM-0677-Security-Review-Anoma-ZK/
   examples/simple_transfer_application/simple_transfer_witness)
   Compiling kudo-traits v0.1.0 ($HOME/.../NM-0677-Security-Review-Anoma-ZK/
   examples/kudo_application/kudo_traits)
   Compiling counter-app v0.1.0 ($HOME/.../NM-0677-Security-Review-Anoma-ZK/
   examples/simple_counter_application/app)
   Compiling simple-transfer-app v0.1.0 ($HOME/.../NM-0677-Security-Review-Anoma-ZK/
   examples/simple_transfer_application/app)
   Compiling kudo-app v0.1.0 ($HOME/.../NM-0677-Security-Review-Anoma-ZK/
   examples/kudo_application/app)
    Finished `release` profile [optimized] target(s) in 1m 02s
     Running unittests src/lib.rs (target/release/deps/arm-7940f7f3ade0ad2f)

running 9 tests
test evm::encode_permit_witness_transfer_from_test ... ok
test evm::evm_resource_test ... ok
test evm::forward_call_data_test ... ok
test utils::test_bytes_to_words ... ok
test utils::test_words_to_bytes ... ok
test encryption::test_encryption ... ok
test authorization::test_authorization ... ok
test delta_proof::test_delta_proof ... ok
test logic_proof::test_padding_logic_prover ... ok

test result: ok. 9 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 15.13s

     Running unittests src/lib.rs (target/release/deps/counter_app-de8cd10f0a82148b)
```

```
running 2 tests
test increment::test_create_increment_tx has been running for over 60 seconds
test init::test_create_init_counter_tx has been running for over 60 seconds
test init::test_create_init_counter_tx ... ok
test increment::test_create_increment_tx ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 4774.17s

     Running unittests src/lib.rs (target/release/deps/counter_witness-2928b1fae750cf82)

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

     Running unittests src/lib.rs (target/release/deps/kudo_app-92fd8e26be4427ce)

running 4 tests
test burn_tx::generate_a_burn_tx has been running for over 60 seconds
test issue_tx::generate_an_issue_tx has been running for over 60 seconds
test swap_tx::generate_a_swap_tx has been running for over 60 seconds
test transfer_tx::generate_a_transfer_tx has been running for over 60 seconds
error: test failed, to rerun pass `-p kudo-app --lib`

Caused by:
  process didn't exit successfully: `$HOME/.../NM-0677-Security-Review-Anoma-ZK/target/release/deps/
  kudo_app-92fd8e26be4427ce` (signal: 9, SIGKILL: kill)
```

Because the tests perform local proving, the Kudo app test fails due to an OOM signal. However, the remaining tests successfully confirm the correct creation of proofs.

9.2 EVM Protocol Adapter Tests

```
forge test
[] Compiling...
No files changed, compilation skipped

Ran 5 tests for test/forwarders/bases/ForwarderBase.t.sol:ForwarderBaseTest
[PASS] test_forwardCall_calls_the_function_in_the_target_contract() (gas: 22778)
[PASS] test_forwardCall_emits_the_CallForwarded_event() (gas: 22520)
[PASS] test_forwardCall_forwards_calls_if_the_pa_is_the_caller() (gas: 19443)
[PASS] test_forwardCall_reverts_if_the_logic_ref_mismatches() (gas: 13868)
[PASS] test_forwardCall_reverts_if_the_pa_is_not_the_caller() (gas: 13755)
Suite result: ok. 5 passed; 0 failed; 0 skipped; finished in 31.41ms (13.54ms CPU time)

Ran 3 tests for test/proofs/LogicProof.t.sol:LogicProofTest
[PASS] testFuzz_different_empty_payloads_produce_different_digest(bytes32,bool,bytes) (runs: 1000, : 15667, ~:
↪  15636)
[PASS] test_verify_example_logic_proof_consumed() (gas: 253096)
[PASS] test_verify_example_logic_proof_created() (gas: 252938)
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 201.43ms (200.67ms CPU time)

Ran 14 tests for test/Benchmark.t.sol:Benchmark
[PASS] test_execute_00() (gas: 20127)
[PASS] test_execute_01_01_agg() (gas: 688681)
[PASS] test_execute_01_01_reg() (gas: 1178433)
[PASS] test_execute_02_02_agg() (gas: 1631663)
[PASS] test_execute_02_02_reg() (gas: 4327642)
[PASS] test_execute_05_01_agg() (gas: 1951210)
[PASS] test_execute_05_01_reg() (gas: 5381206)
[PASS] test_execute_10_01_agg() (gas: 3549833)
[PASS] test_execute_10_01_reg() (gas: 10618977)
[PASS] test_execute_15_01_agg() (gas: 5203202)
[PASS] test_execute_15_01_reg() (gas: 15837656)
[PASS] test_execute_20_01_agg() (gas: 7011428)
[PASS] test_execute_20_01_reg() (gas: 21091814)
[PASS] test_print_calldata_reg() (gas: 11171296)
Suite result: ok. 14 passed; 0 failed; 0 skipped; finished in 1.75s (1.69s CPU time)

Ran 4 tests for test/forwarders/BlockTimeForwarder.t.sol:BlockTimeForwarderTest
[PASS] testFuzz_forwardCall_accepts_arbitrary_logic(bytes32) (runs: 1000, : 6396, ~: 6396)
[PASS] test_forwardCall_returns_EQ_if_timestamp_is_in_the_present() (gas: 6727)
[PASS] test_forwardCall_returns_GT_if_timestamp_is_in_the_future() (gas: 6758)
[PASS] test_forwardCall_returns_LT_if_timestamp_is_in_the_past() (gas: 6868)
Suite result: ok. 4 passed; 0 failed; 0 skipped; finished in 55.55ms (49.94ms CPU time)

Ran 20 tests for test/state/CommitmentTree.t.sol:CommitmentTreeTest
[PASS] test_addCommitmentTreeRoot_emits_the_CommitmentTreeRootAdded_event_on_store_() (gas: 60410)
[PASS] test_addCommitmentTreeRoot_reverts_on_pre_existing_root() (gas: 61306)
[PASS] test_addCommitmentTreeRoot_stores_the_root() (gas: 64227)
[PASS] test_addCommitment_allows_adding_the_same_commitment_multiple_times() (gas: 102790)
[PASS] test_addCommitment_returns_correct_roots() (gas: 198951)
[PASS] test_addCommitment_should_add_commitments() (gas: 182664)
[PASS] test_commitmentTreeRootAtIndex_returns_the_right_index() (gas: 532673)
[PASS] test_constructor_emits_the_CommitmentTreeRootAdded_event() (gas: 6150302)
[PASS] test_constructor_initializes_the_tree_with_0_leaves() (gas: 6147807)
[PASS] test_constructor_initializes_the_tree_with_capacity_1() (gas: 6149515)
[PASS] test_constructor_initializes_the_tree_with_depth_0() (gas: 6149461)
[PASS] test_constructor_stores_the_initial_root_being_the_empty_leaf_hash() (gas: 6148433)
[PASS] test_should_produce_an_invalid_root_for_a_non_existent_leaf() (gas: 555465)
[PASS] test_should_produce_an_invalid_root_for_a_non_existent_leaf_in_the_empty_tree() (gas: 10714)
[PASS] test_verifyMerkleProof_reverts_on_non_existent_commitment() (gas: 133559)
[PASS] test_verifyMerkleProof_reverts_on_non_existent_root() (gas: 14267)
[PASS] test_verifyMerkleProof_reverts_on_wrong_path() (gas: 134946)
[PASS] test_verifyMerkleProof_reverts_on_wrong_path_length() (gas: 112754)
[PASS] test_verifyMerkleProof_verifies_path_for_roots() (gas: 135334)
[PASS] test_verifyMerkleProof_verifies_the_empty_tree_with_depth_zero() (gas: 16072)
Suite result: ok. 20 passed; 0 failed; 0 skipped; finished in 15.56ms (13.03ms CPU time)
```

```
Ran 2 tests for test/proofs/ComplianceProof.t.sol:ComplianceProofTest
[PASS] test_compliance_instance_encoding() (gas: 5213)
[PASS] test_verify_example_compliance_proof() (gas: 243092)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 19.30ms (18.57ms CPU time)

Ran 8 tests for test/proofs/DeltaProof.t.sol:DeltaProofTest
[PASS] testFuzz_add_delta_correctness(uint256,(uint256,uint128,bool),(uint256,uint128,bool)) (runs: 1000, :
↪  76743, ~: 76660)
[PASS] testFuzz_verify_balanced_delta_succeeds(uint256,(uint256,uint128,bool)[],bytes32) (runs: 1000, : 357755,
↪  ~: 356348)
[PASS] testFuzz_verify_delta_succeeds(uint256,uint256,bool,bytes32) (runs: 1000, : 18382, ~: 18404)
[PASS]
↪  testFuzz_verify_fails_if_verifying_keys_of_instance_and_proof_mismatch(uint256,uint256,bool,bytes32,bytes32)
↪  (runs: 1000, : 19523, ~: 19542)
[PASS] testFuzz_verify_imbalanced_delta_fails(uint256,(uint256,uint128,bool)[],bytes32) (runs: 1000, : 354773,
↪  ~: 330898)
[PASS] testFuzz_verify_inconsistent_delta_fails1((uint256,uint256,uint128,bool),bytes32) (runs: 1000, : 19800,
↪  ~: 19846)
[PASS] testFuzz_verify_inconsistent_delta_fails2(uint256,bool,bytes32,uint256,uint256) (runs: 1000, : 20245, ~:
↪  20244)
[PASS] test_verify_example_delta_proof() (gas: 25331)
Suite result: ok. 8 passed; 0 failed; 0 skipped; finished in 11.17s (11.17s CPU time)

Ran 6 tests for test/state/MerkleTree.t.sol:MerkleTreeTest
[PASS] testFuzz_push_returns_the_same_roots() (gas: 5837971)
[PASS] testFuzz_push_returns_the_same_roots(bytes32[1]) (runs: 1000, : 6008200, ~: 6008200)
[PASS] testFuzz_push_returns_the_same_roots(bytes32[2]) (runs: 1000, : 6080874, ~: 6080874)
[PASS] testFuzz_push_returns_the_same_roots(bytes32[]) (runs: 1000, : 8034783, ~: 7901282)
[PASS] test_push_expands_the_tree_depth_if_the_capacity_is_reached() (gas: 5977086)
[PASS] test_setup_returns_the_expected_initial_root() (gas: 5828583)
Suite result: ok. 6 passed; 0 failed; 0 skipped; finished in 12.98s (12.98s CPU time)

Ran 6 tests for test/state/NullifierSet.t.sol:NullifierSetTest
[PASS] test_addNullifier_adds_nullifier() (gas: 73374)
[PASS] test_addNullifier_reverts_on_duplicate() (gas: 76817)
[PASS] test_atIndex_returns_the_nullifier_at_the_give_index() (gas: 472638)
[PASS] test_contains_returns_false_if_the_nullifier_is_not_contained() (gas: 7717)
[PASS] test_contains_returns_true_if_the_nullifier_is_contained() (gas: 73188)
[PASS] test_length_returns_the_length() (gas: 445247)
Suite result: ok. 6 passed; 0 failed; 0 skipped; finished in 455.85µs (253.36µs CPU time)

Ran 1 test for test/forwarders/Permit2.t.sol:DeployPermit2Test
[PASS] test_deploys_Permit2_to_the_canonical_address() (gas: 2388)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 360.25µs (10.43µs CPU time)

Ran 11 tests for test/ProtocolAdapter.t.sol:ProtocolAdapterTest
[PASS] test_constructor_reverts_on_address_zero_router() (gas: 5957959)
[PASS] test_constructor_reverts_on_vulnerable_risc_zero_verifier() (gas: 5998969)
[PASS] test_emergencyStop_emits_the_Paused_event() (gas: 16628)
[PASS] test_emergencyStop_pauses_the_protocol_adapter() (gas: 17010)
[PASS] test_emergencyStop_reverts_if_the_caller_is_not_the_owner() (gas: 11460)
[PASS] test_execute() (gas: 984994)
[PASS] test_execute_does_not_emit_the_CommitmentTreeRootAdded_event_for_the_empty_transaction() (gas: 19482)
[PASS] test_execute_executes_the_empty_transaction() (gas: 19479)
[PASS] test_execute_reverts_if_the_pa_has_been_stopped() (gas: 38118)
[PASS] test_execute_reverts_on_vulnerable_risc_zero_verifier() (gas: 81336)
[PASS] test_getProtocolAdapterVersion_returns_a_semantic_version() (gas: 6927)
Suite result: ok. 11 passed; 0 failed; 0 skipped; finished in 13.34ms (12.04ms CPU time)

Ran 3 tests for test/DeployProtocolAdapter.t.sol:DeployProtocolAdapterTest
[PASS] test_DeployProtocolAdapter_deploys_on_arbitrum_sepolia() (gas: 12399910)
[PASS] test_DeployProtocolAdapter_deploys_on_base_sepolia() (gas: 12399822)
[PASS] test_DeployProtocolAdapter_deploys_on_sepolia() (gas: 12399751)
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 1.11s (1.11s CPU time)

Ran 11 tests for test/forwarders/ERC20Forwarder.t.sol:ERC20ForwarderTest
[PASS] testFuzz_enum_panics(uint8) (runs: 1000, : 3146, ~: 3957)
[PASS] test_unwrap_emits_the_Unwrapped_event() (gas: 75931)
[PASS] test_unwrap_sends_funds_to_the_user() (gas: 77677)
[PASS] test_witness_structHash_complies_with_eip712() (gas: 4657)
[PASS] test_witness_typeHash_complies_with_eip712() (gas: 3868)
```

```
[PASS] test_wrap_emits_the_Wrapped_event() (gas: 152259)
[PASS] test_wrap_pulls_funds_from_user() (gas: 154198)
[PASS] test_wrap_reverts_if_the_amount_to_be_wrapped_overflows() (gas: 105818)
[PASS] test_wrap_reverts_if_the_signature_expired() (gas: 120708)
[PASS] test_wrap_reverts_if_the_signature_was_already_used() (gas: 180930)
[PASS] test_wrap_reverts_if_user_did_not_approve_permit2() (gas: 126173)
Suite result: ok. 11 passed; 0 failed; 0 skipped; finished in 25.30ms (21.84ms CPU time)

Ran 14 tests for test/forwarders/ERC20ForwarderV2.t.sol:ERC20ForwarderV2Test
[PASS] testFuzz_enum_panics(uint8) (runs: 1000, : 3295, ~: 4059)
[PASS] test_migrate_reverts_if_the_resource_has_already_been_migrated() (gas: 204699)
[PASS] test_migrate_reverts_if_the_resource_to_migrate_has_already_been_consumed() (gas: 1038825)
[PASS] test_migrate_transfers_funds_from_V1_to_V2_forwarder() (gas: 210538)
[PASS] test_unwrap_emits_the_Unwrapped_event() (gas: 76059)
[PASS] test_unwrap_sends_funds_to_the_user() (gas: 77794)
[PASS] test_witness_structHash_complies_with_eip712() (gas: 4727)
[PASS] test_witness_typeHash_complies_with_eip712() (gas: 3934)
[PASS] test_wrap_emits_the_Wrapped_event() (gas: 152396)
[PASS] test_wrap_pulls_funds_from_user() (gas: 154289)
[PASS] test_wrap_reverts_if_the_amount_to_be_wrapped_overflows() (gas: 105988)
[PASS] test_wrap_reverts_if_the_signature_expired() (gas: 120850)
[PASS] test_wrap_reverts_if_the_signature_was_already_used() (gas: 181065)
[PASS] test_wrap_reverts_if_user_did_not_approve_permit2() (gas: 126297)
Suite result: ok. 14 passed; 0 failed; 0 skipped; finished in 58.94ms (55.60ms CPU time)

Ran 17 tests for
↪ test/forwarders/bases/EmergencyMigratableForwarderBase.t.sol:EmergencyMigratableForwarderBaseTest
[PASS] test_emergencyCaller_returns_the_emergency_caller_after_it_has_been_set() (gas: 76969)
[PASS] test_emergencyCaller_returns_zero_if_the_emergency_caller_has_not_been_set() (gas: 7887)
[PASS] test_forwardCall_calls_the_function_in_the_target_contract() (gas: 23086)
[PASS] test_forwardCall_emits_the_CallForwarded_event() (gas: 22838)
[PASS] test_forwardCall_forwards_calls_if_the_pa_is_the_caller() (gas: 19923)
[PASS] test_forwardCall_reverts_if_the_logic_ref_mismatches() (gas: 14165)
[PASS] test_forwardCall_reverts_if_the_pa_is_not_the_caller() (gas: 14021)
[PASS] test_forwardEmergencyCall_calls_the_function_in_the_target_contract() (gas: 92172)
[PASS] test_forwardEmergencyCall_emits_the_EmergencyCallForwarded_event() (gas: 92662)
[PASS] test_forwardEmergencyCall_forwards_calls_if_the_pa_is_stopped_and_the_caller_is_the_emergency_caller()
↪ (gas: 89302)
[PASS] test_forwardEmergencyCall_reverts_if_the_pa_is_stopped_but_the_caller_is_not_the_emergency_caller()
↪ (gas: 80359)
[PASS] test_forwardEmergencyCall_reverts_if_the_pa_is_stopped_but_the_emergency_caller_is_not_set() (gas:
↪ 46704)
[PASS] test_setEmergencyCaller_reverts_if_the_caller_is_not_the_emergency_committee() (gas: 11696)
[PASS] test_setEmergencyCaller_reverts_if_the_emergency_caller_has_already_been_set() (gas: 80143)
[PASS] test_setEmergencyCaller_reverts_if_the_new_emergency_caller_is_the_zero_address() (gas: 11228)
[PASS] test_setEmergencyCaller_reverts_if_the_pa_is_not_stopped() (gas: 29654)
[PASS] test_setEmergencyCaller_sets_the_emergency_caller() (gas: 76947)
Suite result: ok. 17 passed; 0 failed; 0 skipped; finished in 3.99ms (1.66ms CPU time)

Ran 2 tests for test/SHA256.t.sol:SHA256Test
[PASS] test_hash2_reproduces_the_EfficientHashLib_hash(bytes32,bytes32) (runs: 1000, : 1431, ~: 1614)
[PASS] test_hash_reproduces_the_EfficientHashLib_hash(bytes32) (runs: 1000, : 987, ~: 987)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 45.79ms (45.67ms CPU time)

Ran 3 tests for test/proofs/TagLookup.t.sol:TagLookupTest
[PASS] test_lookup_returns_the_verifier_input_if_it_finds_the_lookup_tag() (gas: 658634)
[PASS] test_lookup_reverts_if_the_tag_is_not_found() (gas: 191758)
[PASS] test_lookup_reverts_on_empty_list() (gas: 9332)
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 5.92ms (5.61ms CPU time)

Ran 1 test for test/libs/Versioning.t.sol:VersioningTest
[PASS] test_check_that_the_current_version_is_a_pre_release() (gas: 1695)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.18ms (26.24µs CPU time)

Ran 25 tests for test/ProtocolAdapterMock.t.sol:ProtocolAdapterMockVerifierTest
[PASS] testFuzz_execute_1_txn_with_2_action_with_1_and_0_cus(bool) (runs: 1000, : 383933, ~: 377714)
[PASS] testFuzz_execute_1_txn_with_n_actions_and_n_cus(uint8,uint8,bool) (runs: 1000, : 1126442, ~: 386408)
[PASS] testFuzz_execute_1_txn_with_up_to_3_empty_actions(bool[3],bool) (runs: 1000, : 487346, ~: 401207)
[PASS] testFuzz_execute_2_txns_with_n_actions_and_n_cus(uint8,uint8,bool) (runs: 1000, : 2175832, ~: 1123767)
[PASS] testFuzz_execute_emits_ActionExecuted_events_for_each_action(uint8,uint8,bool) (runs: 1000, : 5338000,
↪ ~: 2058421)
[PASS] testFuzz_execute_emits_all_ForwarderCallExecuted_events(bool) (runs: 1000, : 764061, ~: 756866)
```

```
[PASS] testFuzz_execute_emits_the_ForwarderCallExecuted_event_on_consumed_carrier_resource(bool) (runs: 1000, :
↪  407990, ~: 414403)
[PASS] testFuzz_execute_emits_the_TransactionExecuted_event(uint8,uint8,bool) (runs: 1000, : 5306338, ~:
↪  1613948)
[PASS] testFuzz_execute_reverts_if_commitment_from_compliance_inputs_cannot_be_found_in_logic_inputs(
uint8,uint8,uint8,uint8,bytes32,bool) (runs: 1000, : 1285223, ~: 855559)
[PASS] testFuzz_execute_reverts_if_nullifier_from_compliance_inputs_cannot_be_found_in_logic_inputs(
uint8,uint8,uint8,uint8,bytes32,bool) (runs: 1000, : 1188112, ~: 783259)
[PASS] testFuzz_execute_reverts_if_proofs_start_with_an_unknown_verifier_selector(
uint8,uint8,uint8,uint8,bytes4,bytes,bool) (runs: 1000, : 1491203, ~: 1058377)
[PASS]
↪  testFuzz_execute_reverts_on_compliance_and_logic_verifier_logic_reference_mismatch_for_consumed_resource(
uint8,uint8,uint8,uint8,bytes32,bool) (runs: 1000, : 1150191, ~: 765236)
[PASS] testFuzz_execute_reverts_on_compliance_and_logic_verifier_logic_reference_mismatch_for_created_resource(
uint8,uint8,uint8,uint8,bytes32,bool) (runs: 1000, : 1228701, ~: 774148)
[PASS]
↪  testFuzz_execute_reverts_on_compliance_and_logic_verifier_tag_count_mismatch(uint8,uint8,uint8,uint8,bool)
↪  (runs: 1000, : 889116, ~: 615535)
[PASS]
↪  testFuzz_execute_reverts_on_logic_and_compliance_verifier_tag_count_mismatch(uint8,uint8,uint8,uint8,bool)
↪  (runs: 1000, : 951203, ~: 671216)
[PASS] testFuzz_execute_reverts_on_non_existing_root(uint8,uint8,uint8,uint8,bytes32,bool) (runs: 1000, :
↪  1145079, ~: 725394)
[PASS] testFuzz_execute_reverts_on_pre_existing_nullifier(bool) (runs: 1000, : 402981, ~: 401832)
[PASS] testFuzz_execute_reverts_on_resource_count_mismatch(uint8,bool) (runs: 1000, : 291848, ~: 252136)
[PASS] testFuzz_execute_reverts_on_ubalanced_delta(uint128,uint128,bool) (runs: 1000, : 308556, ~: 308375)
[PASS] testFuzz_execute_reverts_on_unexpected_forwarder_call_output(bytes,bool) (runs: 1000, : 254992, ~:
↪  250549)
[PASS] testFuzz_execute_updates_commitment_root_exactly_with_desired_commitments(uint8,uint8,bool) (runs: 1000,
↪  : 8316676, ~: 7779204)
[PASS] testFuzz_execute_updates_nullifier_set_exactly_with_desired_nullifiers(uint8,uint8,bool) (runs: 1000, :
↪  1806720, ~: 1350460)
[PASS] testFuzz_execute_updates_root(uint8,uint8,bool) (runs: 1000, : 1856266, ~: 1339569)
[PASS] test_execute_emits_the_ForwarderCallExecuted_event_on_created_carrier_resource(bool) (runs: 1000, :
↪  408141, ~: 414621)
[PASS] test_execute_skips_risc_zero_proofs_if_aggregation_proof_is_present() (gas: 384088)
Suite result: ok. 25 passed; 0 failed; 0 skipped; finished in 96.09s (170.58s CPU time)

Ran 19 test suites in 109.35s (123.59s CPU time): 156 tests passed, 0 failed, 0 skipped (156 total tests)
```

10 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development process, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

Blockchain Security: At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

Blockchain Core Development: Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

DevOps and Infrastructure Management: Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

Cryptography Research: At Nethermind, our cryptography Research team conducts cutting-edge internal research and collaborates closely with external partners on cryptographic protocols, consensus design, succinct arguments and folding schemes, elliptic curve-based STARK protocols, post-quantum security and zero-knowledge proofs (ZKPs). Our research has led to influential contributions, including Zinc (Crypto '25), Mova, FLI (Asiacrypt '24), and foundational results in Fiat-Shamir security and STARK proof batching. Complementing this theoretical work, our engineering expertise is demonstrated through implementations such as the Latticefold aggregation scheme, the Labrador proof system, zkvm-benchmarks, and Plonk Verifier in Cairo. This combined strength in theory and engineering enables us to deliver cutting-edge cryptographic solutions to partners and clients.

Smart Contract Development & DeFi Research: Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

Our suite of L2 tooling: Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;
- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;

– **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

General Advisory to Clients

As auditors, we recommend that any changes or updates made to the audited codebase undergo a re-audit or security review to address potential vulnerabilities or risks introduced by the modifications. By conducting a re-audit or security review of the modified codebase, you can significantly enhance the overall security of your system and reduce the likelihood of exploitation. However, we do not possess the authority or right to impose obligations or restrictions on our clients regarding codebase updates, modifications, or subsequent audits. Accordingly, the decision to seek a re-audit or security review lies solely with you.

Disclaimer