Security Review Report
NM-0626 - Kintsu

NETHERMIND
SECURITY

(November 4, 2025)

Contents

1 Executive Summary

This document presents the results of a security review conducted by Nethermind Security for the Kintsu protocol.

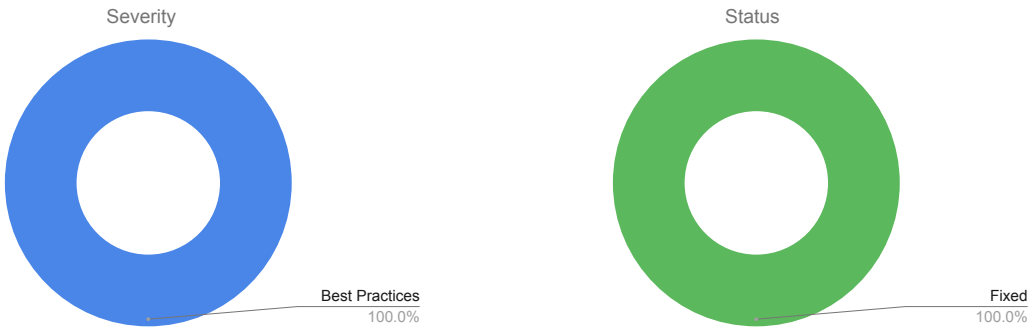
Kintsu is a liquid staking protocol designed for the Monad network. It enables users to deposit native `MON` tokens into a common pool in exchange for `sMON`, an ERC20 liquid staking token (LST). This allows users to retain liquidity and participate in DeFi activities while their underlying assets earn staking rewards.

The protocol's architecture is closely integrated with Monad's native consensus and staking mechanisms. It pools user deposits and delegates them across a set of validators defined in an on-chain `Registry`. A core, permissionless batching system aligns with Monad's epoch structure to process net deposits or withdrawals, automatically rebalancing the protocol's stake to match DAO-governed validator weights. To accommodate Monad's native unstaking period, withdrawals are a two-step process: users first `requestUnlock` their `sMON` and then, after a delay, `redeem` them for native `MON`.

The scope of this security review encompassed the core `StakedMonad` smart contract, which includes the logic for deposits, the two-step withdrawal queue, batch processing, reward compounding, fee collection, and all interactions with the Monad staking precompile via its wrapper.

The audit comprises 711 lines of Solidity code. **The audit was performed using** (a) manual analysis of the codebase, and (b) automated analysis tools. **Along this document, we report** three points of attention, all of which are classified as 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 issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 presents the test suite evaluation and automated tools used. Section 9 concludes the document.



(a) (b)

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

Summary of the Audit

Audit Type	Security Review
Initial Report	November 3, 2025
Final Report	November 4, 2025
Initial Commit	0d3c9f49ee98d067d5e5a2d97666dfc6f94b70b4
Final Commit	5d0a0f84699cd825df9d7cb9ec75fd97207422d8
Documentation Assessment	Medium
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/CustomErrors.sol	20	23	115.0%	17	60
2	src/Registry.sol	110	52	47.3%	42	204
3	src/StakedMonad.sol	446	206	46.2%	125	777
4	src/precompile/StakerUpgradeable.sol	135	91	67.4%	36	262
	Total	**711**	**372**	**52.3%**	**220**	**1303**

3 Summary of Issues

	Finding	Severity	Update
1	Fee calculations round down instead of rounding up in favor of the protocol	Best Practices	Fixed
2	Phase 2 logic in bonding and unbonding functions is unnecessary and adds complexity	Best Practices	Fixed
3	The `updateWeights(...)` function reverts with an incorrect error message	Best Practices	Fixed

4 System Overview

Kintsu is a liquid staking protocol for the Monad network. It allows users to deposit native MON tokens into a common pool in exchange for sMON, a liquid ERC20 token representing their share of the pooled, staked assets. The protocol automatically delegates these MON tokens across a set of validators defined in an on-chain Registry, compounding staking rewards and managing the undelegation process. This provides users with a tradable and DeFi-compatible asset (sMON) while their underlying MON tokens secure the Monad network.

4.1 Deposits and sMON Minting

Users initiate the staking process by calling the deposit function, sending MON tokens as msg.value. The protocol calculates the amount of sMON shares to mint based on the current conversion ratio. This ratio is not fixed but is derived from the contract's total assets and total shares by the convertToShares function. The totalPooled variable tracks all MON managed by the protocol (staked, batched, or liquid), while the totalShares() function represents the sum of all minted sMON held by users plus any outstanding protocol fees. New deposits increase the totalPooled and the total sMON supply, and the deposited MON is added to the batchDepositRequests for the currentBatchId.

4.2 Withdrawal Lifecycle (Two-Step)

Withdrawing from the protocol is a two-step process that spans multiple epochs due to the native unstaking delay on the Monad network.

- **1. Request Unlock**: A user calls requestUnlock, specifying the amount of sMON they wish to redeem. The protocol calculates the corresponding asset spotValue based on the current convertToAssets ratio. An exit fee is calculated and subtracted from the user's shares. The user's full sMON (including the fee portion) is transferred to the StakedMonad contract, and an UnlockRequest struct is created, tying the request to the currentBatchId. Users can reverse this action by calling cancelUnlockRequest, which returns their full sMON amount, provided the batch (currentBatchId) has not yet been submitted.
- **2. Redeem**: Once the batch containing the user's request has been processed by submitBatch and the Monad network's native unstaking period (getWithdrawDelay) has passed, the user can call redeem. This function deletes the UnlockRequest and transfers the native MON (spotValue) to the user.

The redeem function requires the StakedMonad contract to have a sufficient liquid MON balance. This balance is typically funded by a permissionless sweep function, which withdraws completed undelegations from the Monad staking precompile. However, funds from new deposits or compound calls are also held in this contract. This can create a "race condition" where a user's redeem call may be front-run or succeed using funds intended for the next deposit batch. While funds are fully accounted for, this mechanic means that a successful redemption relies on funds being available in the contract, and user redemptions could cause a subsequent submitBatch call to revert if it relies on those same funds.

4.3 Batch Processing and Rebalancing

The core logic of the protocol is managed by the submitBatch function, which is permissionless and intended to be called by a keeper. This function is designed around Monad's epoch structure. It first calls _getActivityEpoch to determine the current Monad epoch and the future epoch when any new stake changes will become active. The protocol enforces a MinimumBatchDelay, preventing a new batch from being submitted until the previous batch's activationEpoch has passed.

When called, submitBatch processes all queued deposits and withdrawals for the currentBatchId.

- If deposits are greater than withdrawals (net ingress), it calls _doBonding.
- If withdrawals are greater than deposits (net egress), it calls _doUnbonding.

Both functions rebalance the protocol's total stake by delegating or undelegating MON to/from the nodes listed in the Registry. This is done to align the staked amount of each node with its designated weight, ensuring the protocol's stake distribution matches the DAO-governed targets. Any rounding "dust" from these operations is carried over to the next batch.

4.4 Reward Compounding and Fees

Staking rewards are not distributed directly to users. Instead, a keeper calls the compound function, which claims all pending rewards from the Monad staking precompile for all registered nodes. These claimed MON rewards are sent to the StakedMonad contract, increasing the totalPooled (assets). Since no new sMON shares are minted, this action automatically increases the value of all existing sMON shares.

The protocol has two fees:

- **Exit Fee**: A small percentage fee, specified in exitFee.bips, taken from a user's shares during requestUnlock. These shares are held in escrow and, after the batch is submitted, become protocolShares that can be claimed.
- **Management Fee**: An annualized fee calculated in _updateFees. This function calculates fee shares as a virtualSharesSnapshot. Uniquely, these virtual shares are not minted but are immediately included in the totalShares() view function. This design isolates protocol revenue from user-held shares, as the totalShares() (the denominator of the sMON price) grows proportionally with the totalPooled (the numerator). This allows for clean accounting and revenue projection.

Both collected fees are claimed by a ROLE_FEE_CLAIMER by calling claimProtocolFees. This mints the accumulated virtualSharesSnapshot to the fee recipient and transfers the collected protocolShares from the exit fees.

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 Issues

6.1 [Best Practices] Fee calculations round down instead of rounding up in favor of the protocol

File(s): src/StakedMonad.sol

Description: The protocol implements two types of fees: exit fees charged when users request to unlock their shares, and management fees that accrue continuously as virtual shares. Both fee calculations use integer division which rounds down toward zero, causing the protocol to collect less fees than intended, especially for small amounts.

The exit fee calculation in `requestUnlock(...)` multiplies the user's shares by the exit fee basis points and divides by `BIPS`, using integer division that rounds down:

```
// @audit - should round in favor of the protocol
sharesToFee = uint96(uint256(shares) * exitFeeInBips / BIPS);
```

Similarly, the management fee calculation in `_updateFees()` uses integer division for both the fee percentage and time-weighted calculations:

```
uint256 feeShares = (ERC20Upgradeable.totalSupply() +
    _managementFee.virtualSharesSnapshot) * _managementFee.bips / BIPS;
uint256 feeSharesTimeWeighted = feeShares * time / YEAR;
// @audit - Both divisions round down, losing fees for the protocol
_managementFee.virtualSharesSnapshot = uint96(_managementFee.virtualSharesSnapshot
    + feeSharesTimeWeighted);
```

When dealing with small amounts, integer division can result in fees rounding down to zero, meaning the protocol receives no fee at all for those transactions. For example, with an exit fee of `5` basis points (0.05%), a user unlocking `1999` wei worth of shares would result in `1999 * 5 / 10000 = 0` wei fee after integer division.

Recommendation(s): Consider implementing rounding-up logic for fee calculations to ensure the protocol always collects fees in its favor. This is a common best practice in DeFi protocols to prevent gradual fee losses from rounding down.

Status: Fixed

Update from the client: We changed to a round-up approach when calculating protocol fees to round in the protocol's favor. When all fees are set to zero, no rounding takes place and the protocol will still accumulate zero fees as before.

We added these changes and associated tests in commit 31b9c1f0aafddf5c17106807162241d63bfe4474.

6.2 [Best Practices] Phase 2 logic in bonding and unbonding functions is unnecessary and adds complexity

File(s): src/StakedMonad.sol

The `_doBonding(...)` and `_doUnbonding(...)` functions implement a two-phase distribution mechanism where Phase 1 handles rebalancing towards optimal allocations and Phase 2 distributes any remaining funds proportionally. Phase 1 calculates and fills under-allocations (bonding) or drains over-allocations (unbonding) based on the imbalance between current stake distribution and target distribution derived from node weights. Any funds remaining after Phase 1 would be distributed in Phase 2 proportionally by weight or remaining stake.

However, Phase 2 can only contain at most `(number_of_nodes - 1)` wei because `underAllocation` is calculated using `newTotalStaked`, which already includes the `requestedBonding` amount (`newTotalStaked = totalPooled - batchWithdrawals`). Since optimal allocations are computed based on this future total that accounts for the new funds, `underAllocation` will always be approximately equal to `requestedBonding`. This means Phase 1 distributes essentially all of `requestedBonding` to fill the deficits, leaving only negligible rounding dust (at most `(number_of_nodes - 1)` wei from integer division) for Phase 2.

```
// Amount to distribute to under-allocated nodes
uint256 phase1 = requestedBonding < underAllocation ? requestedBonding : underAllocation;

// Remaining amount to distribute equitably to all nodes
uint256 phase2 = requestedBonding - phase1;

// ... Phase 1 distribution ...

// @audit-issue Phase 2 distribution handles only dust amounts that round to zero
uint256 phase2Amount = phase2 > 0 ? phase2 * _nodes[i].weight / _totalWeight : 0;
```

The dust created from Phase 2 (and Phase 1 rounding) is already handled by the batch submission mechanism in `submitBatch(...)`, which calculates the difference between requested amounts and actual amounts bonded/unbonded, then carries forward any dust to the next batch:

```
if (batchDepositRequest.assets > batchWithdrawRequest.assets) {
    uint96 bondedAmountEvm = _doBonding(...);
    uint96 dust = batchDepositRequest.assets - batchWithdrawRequest.assets
        - bondedAmountEvm;
    // @audit Dust from Phase 2 (and Phase 1 rounding) is handled here
    if (dust > 0) {
        batchDepositRequests[_currentBatchId + 1].assets = dust;
    }
}
```

This means the Phase 2 logic adds unnecessary complexity to the codebase without providing meaningful functionality, as any funds that would go through Phase 2 simply become dust that gets processed in the next batch submission anyway.

Recommendation(s): Consider removing the Phase 2 distribution logic from both `_doBonding(...)` and `_doUnbonding(...)` functions, as it only handles negligible dust amounts that are already properly managed by the existing batch dust handling mechanism. The Phase 1 logic is sufficient to handle all meaningful rebalancing operations, and any rounding losses will naturally become dust that gets carried forward to subsequent batches where they can accumulate to meaningful amounts before being processed.

Status: Fixed

Update from the client: After mimicking the logical flow in excel sheets, we confirmed that phase2 is indeed only handling dust. We simplified the algo in both `_doBonding()` and `_doUnbonding()` to use a single phase approach. We also refactored the helper function to better explain that it is doing the heavy lifting of pro rata weight based syncing.

Addressed in commit e99d4511bdb771a76f88b8182b9e5e9844e30889.

6.3 [Best Practices] The `updateWeights(...)` function reverts with an incorrect error message

File(s): src/Registry.sol

Description: The `updateWeights(...)` function in the `Registry` contract is used to modify the stake allocation weights for nodes. The logic correctly prevents increasing the weight of a node that has already been disabled.

However, when this check fails (i.e., an attempt is made to increase the weight of a disabled node), the function reverts with the `ActiveNode()` error. This error message is misleading, as it implies the node is active when the check is specifically for disabled nodes. This can cause confusion for users or off-chain tooling monitoring for failed transactions, as the revert reason does not match the actual condition that failed.

```
function updateWeights(WeightDelta[] calldata weightDeltas) external {
    // ...
    uint256 len = weightDeltas.length;
    for (uint256 i; i < len; ++i) {
        if (weightDeltas[i].isIncreasing) {
            Node storage node = getNodeByNodeId(weightDeltas[i].nodeId);
            // @audit-issue This check reverts with ActiveNode() error when
            // the node is disabled.
            if (isNodeDisabled[node.id]) revert ActiveNode();
            // ...
        } else {
            // ...
        }
    }
    // ...
}
```

Recommendation(s): Consider changing the error thrown in this scenario to one that more accurately reflects the reason for the revert, such as `NodeDisabled()`. This will improve the clarity of the function's behavior when it reverts.

Status: Fixed

Update from the client: We see how this could be confusing.

We changed the error message in commit af0a8346cc7bfaff1ae65aa02f0701f4ba9dff75.

7 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. Smart contracts 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 smart contract'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 smart contract. 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 smart contract. 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 smart contract. 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 smart contract 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 smart contract. This type of documentation is critical in ensuring that the smart contract is secure and free from vulnerabilities.

These types of documentation are essential for smart contract 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 Kintsu documentation

The Kintsu team provided a clear and detailed code walkthrough during the audit kick-off call. They professionally and effectively addressed all questions raised by the Nethermind Security team throughout the engagement.
Furthermore, the codebase itself is well-documented, featuring comprehensive NatSpec comments and inline explanations that clarify the reasoning behind key design decisions.

8 Test Suite Evaluation

8.1 Mutation Analysis

The Kintsu protocol's test suite is robust and clearly outlines the contract specifications. The test suite catches any change to the business logic. While core flows are well-tested, certain areas could benefit from higher test coverage, particularly the stake distribution logic in `_doBonding` and `_doUnbonding` (including the Phase 1 and Phase 2 rebalancing) and the associated dust handling mechanisms. These areas were subject to refactoring as a result of the audits performed on the codebase.

To assess the test coverage for these files, the Nethermind Security team applied a technique called mutation testing to uncover the untested paths. This technique introduces slight modifications to the code called "mutations" or "mutants." An example of a mutation is, for example, changing the operator in an expression `(a + b)` to `(a - b)`, or removing a `require(a > b)` statement entirely from the code. With these changes, the code no longer follows the expected business logic of the application, and the test suite should reflect that by failing.

Evaluation of the test suite with mutation testing consists of two phases:

a. Generating the modified version of each contract, called "mutants."

b. Inserting the mutant into the original codebase and running the test suite.

Only one modification can be tested at a time. If the contract has ten mutations, the test suite must run ten times (once for every mutation). If any of the tests fail, it means that the test suite caught the change in the code. Whenever that happens, the particular mutant is considered "slain" or "killed" and is removed from the mutant's set. If that does not occur, a new test case can be added to cover the code branch to "kill" the mutant.

The following table outlines the mutation analysis results performed on Kintsu's core smart contracts. Each row corresponds to a contract tested, showing the number of mutants generated, how many were "slain" (i.e., caught by the test suite), and the corresponding mutation score. A higher mutation score reflects stronger test coverage and better bug-detection capabilities.

Contract	Slain / Total	Mutation Score
`src/Registry.sol`	131 / 140	93.57%
`src/StakedMonad.sol`	545 / 704	77.41%
`src/precompile/StakerUpgradeable.sol`	48 / 65	73.85%
Total	**724 / 909**	**79.65%**

The following is a list of code modifications not caught by the existing test suite. Each point highlights a scenario that could benefit from higher test coverage to enhance the protocol's overall security and resilience in the next code change iterations as the project evolves.

src/Registry.sol

- In the `removeNode(...)` function, the `if` statement condition: `offsetIndex == 0` can be hardcoded to `false` without affecting the test suite. Consider adding test cases for the revert with the `InvalidNode` error.
- In the `removeNode(...)` function, the assignment `_nodeIdToOffsetIndex[lastNode.id] = offsetIndex;` can be changed to `_nodeIdToOffsetIndex[lastNode.id] = 0;` with no effect on the test suite. Consider adding test cases that verify the correct updating of mapping entries when nodes are removed and replaced in the array, particularly testing that the mapping is properly updated for the last node after a removal operation.
- In the `removeNode(...)` function, the line `_nodeIdToOffsetIndex[lastNode.id] = offsetIndex` can be replaced with `assert(true)` (which is effectively a no-op) without affecting the test suite. Consider adding test cases to check that the mapping `_nodeIdToOffsetIndex` is properly updated when nodes are removed and replaced, ensuring the correct index mapping for the last node in the array after removal.
- In the `removeNode(...)` function, the subtraction in the array indexing `nodes[offsetIndex - 1]` can be changed to modulo operation without affecting the test suite. Consider adding test cases that verify the correct node is being accessed and removed from the storage array.
- In the `updateWeights(...)` function, the `if` statement condition: `oldWeight == 0` can be hardcoded to `false` without affecting the test suite. Consider adding test cases for the path when certain nodes are skipped.
- In the `updateWeights(...)` function, the expression `nodes[offsetIndex - 1]` can be mutated to `nodes[1 - offsetIndex]` without affecting the test suite. Consider adding test cases that verify the correct array indexing behavior when updating node weights.
- In the `updateWeights(...)` function, the subtraction in the totalWeight calculation `_totalWeight = _totalWeight - oldWeight + newWeight` can be changed to modulo operation without affecting the test suite. Consider adding test cases that verify the correct accumulation of weights when updating node weights.

src/StakedMonad.sol

- In the `_authorizeRemoveNode(...)` function, the `if` statement condition: `StakerUpgradeable.isForceWithdrawPending(nodeId)` can be hardcoded to `false` without affecting the test suite. Consider adding test cases for the revert with the `PendingWithdrawals` error.

- In the `_deleteUnlockRequest(...)` function, the `array.pop()` statement can be replaced with `assert(true)` (which is effectively a no-op) without affecting the test suite. Consider adding test cases that verify the correct removal of array elements and ensure the array length is properly decremented after calling `_deleteUnlockRequest`.
- In the `_deleteUnlockRequest(...)` function, the `if` statement condition: `index != finalIndex` can be hardcoded to `true` without affecting the test suite. Consider adding tests for cases when the logic from the `if` branch of this statement is NOT executed.
- In the `_doBonding(...)` function, the `StakerUpgradeable.delegate(_nodes[i].id, bondAmount)` call can be replaced with `assert(true)` (which is effectively a no-op) without affecting the test suite. Consider adding test cases to check the expected effects of the original `StakerUpgradeable.delegate(...)` call, particularly verifying that delegation occurs correctly and that the staking system properly tracks delegated amounts.
- In the `_doBonding(...)` function, the `if` statement condition: `_totalWeight == 0` can be hardcoded to `false` without affecting the test suite. Consider adding test cases for the revert with the `ZeroTotalWeight` error.
- In the `_doBonding(...)` function, the `if` statement condition: `bondAmount > 0` can be hardcoded to `true` without affecting the test suite. Consider adding tests for cases when the logic from the `if` branch of this statement is NOT executed.
- In the `_doBonding(...)` function, the addition in the calculation `uint96 bondAmount = uint96(phase1Amount + phase2Amount);` can be changed to subtraction without affecting the test suite. Consider adding test cases that verify the correct computation of bond amounts when both phase1 and phase2 amounts are non-zero, particularly focusing on scenarios where the subtraction would produce different results than the original addition.
- In the `_doBonding(...)` function, the condition `phase2 > 0` in the ternary expression can be changed to `0 > phase2` without affecting the test suite. Consider adding test cases that specifically verify the distribution logic when `phase2` is zero, ensuring that there are no state changes coming from the 2nd phase.
- In the `_doBonding(...)` function, the multiplication in the phase2Amount calculation `phase2 * _nodes[i].weight / _totalWeight` can be changed to division without affecting the test suite. Consider adding test cases that verify the correct distribution of funds during the bonding process, particularly focusing on how the `phase2Amount` is calculated when distributing remaining funds equitably based on node weights.
- In the `_doBonding(...)` function, the subtraction in the calculation `_totalPooled - batchWithdrawals` can be changed to addition without affecting the test suite. Consider adding test cases that verify the correct computation of `newTotalStaked` and its impact on the allocation logic when batch withdrawals are involved.
- In the `_doBonding(...)` function, the subtraction in the calculation `uint256 phase2 = requestedBonding - phase1;` can be changed to modulo operation without affecting the test suite. Consider adding test cases that verify the correct distribution of funds in both phases of the bonding process.
- In the `_doUnbonding(...)` function, the addition in the calculation `uint96 unbondAmount = uint96(phase1Amount + phase2Amount);` can be changed to subtraction without affecting the test suite. Consider adding test cases that verify the correct computation of unbond amounts under different phase1 and phase2 scenarios.
- In the `_doUnbonding(...)` function, the division operation in the calculation `phase2 * (_nodes[i].staked - phase1Amount) / totalStakedAfterPhase1` can be changed to multiplication without affecting the test suite. Consider adding test cases that verify the correct distribution of unbonding amounts across nodes, particularly focusing on scenarios where the phase2 calculation would produce different results with multiplication versus division.
- In the `_doUnbonding(...)` function, the division operation in the calculation of `phase2Amount` can be changed to addition without affecting the test suite. Consider adding test cases that verify the correct distribution of unbonding amounts.
- In the `_doUnbonding(...)` function, the expression `uint256 phase2 = requestedUnbonding - phase1;` can be mutated to `phase2 = phase1 - requestedUnbonding;` without affecting the test suite. Consider adding test cases that verify the correct calculation of `phase2` under different scenarios where the order of subtraction matters.
- In the `_doUnbonding(...)` function, the subtraction in the `_getImbalances` function call `_totalPooled - batchWithdrawals` can be changed to addition without affecting the test suite. Consider adding test cases that verify the correct calculation of `newTotalStaked` and its impact on the imbalances calculation.
- In the `_doUnbonding(...)` function, the subtraction in the calculation `uint256 requestedUnbonding = batchWithdrawals - batchDeposits` can be changed to addition without affecting the test suite. Consider adding test cases that verify the correct computation of the `requestedUnbonding` value under different scenarios involving batch withdrawals and deposits.
- In the `_doUnbonding(...)` function, the subtraction in the calculation `uint256 totalStakedAfterPhase1 = (totalPooled - batchDeposits) - phase1` can be changed to multiplication without affecting the test suite. Consider adding test cases that verify the correct computation of `totalStakedAfterPhase1` and its usage in the subsequent `phase2Amount` calculation.
- In the `_getImbalances(...)` function, the `if` statement condition: `stakedAmountOptimal > stakedAmountCurrent` can be hardcoded to `true` without affecting the test suite. Consider adding test cases for the logic executed in the `else if` branch of this statement.
- In the `_updateFees()` function, the `if` statement condition: `time > 0` can be hardcoded to `true` without affecting the test suite. Consider adding tests for cases when the logic from the `if` branch of this statement is not executed.
- In the `_updateFees()` function, the addition in the calculation `ERC20Upgradeable.totalSupply() + _managementFee.virtualSharesSnapshot` can be changed to subtraction without affecting the test suite. Consider adding test cases that verify the correct computation of management fee shares based on the total supply and virtual shares snapshot.

- In the `cancelUnlockRequest(...)` function, the `if` statement condition: `sharesToFee > 0` can be hardcoded to `false` without affecting the test suite. Consider adding test cases for the logic executed in the `if` branch of this statement.
- In the `cancelUnlockRequest(...)` function, the `if` statement condition: `sharesToFee > 0` can be hardcoded to `true` without affecting the test suite. Consider adding test cases for scenarios where the exit fee is zero and results in a value that would make the condition evaluate to `false`.
- In the `cancelUnlockRequest(...)` function, the `if` statement condition: `userUnlockRequest.batchId != currentBatchId` can be hardcoded to `false` without affecting the test suite. Consider adding test cases for the revert with the `CancellationWindowClosed` error.
- In the `cancelUnlockRequest(...)` function, the assignment `_batchWithdrawRequest.assets -= userUnlockRequest.spotValue;` can be changed to `_batchWithdrawRequest.assets -= 0;` with no effect on the test suite. Consider adding test cases that verify the correct updating of batch asset amounts when unlock requests are cancelled.
- In the `cancelUnlockRequest(...)` function, the assignment `_batchWithdrawRequest.shares -= sharesToUnlock;` can be changed to `_batchWithdrawRequest.shares -= 0;` with no effect on the test suite. Consider adding test cases that verify the correct reduction of shares in the batch withdraw request when an unlock request is cancelled.
- In the `cancelUnlockRequest(...)` function, the assignment `exitFee.escrowShares -= sharesToFee;` can be changed to `exitFee.escrowShares -= 0;` with no effect on the test suite. Consider adding test cases that verify the correct updating of `exitFee.escrowShares` when unlock requests are cancelled.
- In the `cancelUnlockRequest(...)` function, the call to `_deleteUnlockRequest(userUnlockRequestArray, unlockIndex)` can be replaced with `assert(true)` (which is effectively a no-op) without affecting the test suite. Consider adding test cases to verify that the unlock request is properly removed from the `userUnlockRequests` array.
- In the `cancelUnlockRequest(...)` function, the line `_batchWithdrawRequest.assets -= userUnlockRequest.spotValue` can be replaced with `assert(true)` (which is effectively a no-op) without affecting the test suite. Consider adding test cases to check the expected effects of the asset amount deduction in the batch withdrawal request when an unlock request is cancelled.
- In the `cancelUnlockRequest(...)` function, the line `exitFee.escrowShares -= sharesToFee` can be replaced with `assert(true)` (which is effectively a no-op) without affecting the test suite. Consider adding test cases to check the expected effects of the `exitFee.escrowShares -= sharesToFee` expression.
- In the `claimProtocolFees(...)` function, the `if` statement condition: `exitFeeShares > 0` can be hardcoded to `true` without affecting the test suite. Consider adding test cases for the logic when the `if` branch is not executed.
- In the `claimProtocolFees(...)` function, the `if` statement condition: `managementFeeShares > 0` can be hardcoded to `true` without affecting the test suite. Consider adding test cases for the logic when the `if` branch is not executed.
- In the `claimProtocolFees(...)` function, the assignment `exitFee.protocolShares = 0;` can be changed to `exitFee.protocolShares = 1;` with no effect on the test suite. Consider adding test cases that verify the correct behavior of fee distribution and ensure that `protocolShares` is properly reset to zero after being withdrawn.
- In the `claimProtocolFees(...)` function, the assignment `managementFee.virtualSharesSnapshot = 0;` can be changed to `managementFee.virtualSharesSnapshot = 1;` with no effect on the test suite. Consider adding test cases that depend on the exact value of `managementFee.virtualSharesSnapshot` being reset to zero after claiming fees.
- In the `compound(...)` function, the `if` statement condition: `claimedRewards == 0` can be hardcoded to `true` without affecting the test suite. Consider adding test cases for the revert with the `NoChange` error.
- In the `compound(...)` function, the assignment `batchDepositRequests[currentBatchId].assets += uint96(claimedRewards);` can be changed to `batchDepositRequests[currentBatchId].assets += 0;` with no effect on the test suite. Consider adding test cases that verify the correct accumulation of claimed rewards into the batch deposit requests.
- In the `compound(...)` function, the assignment `totalPooled += uint96(claimedRewards);` can be changed to `totalPooled += 0;` with no effect on the test suite. Consider adding test cases that verify the correct updating of `totalPooled` when rewards are claimed, ensuring the balance reflects the actual rewards added to the pool.
- In the `compound(...)` function, the call to `StakerUpgradeable.claimRewards(nodeIds[i])` can be replaced with `assert(true)` (which is effectively a no-op) without affecting the test suite. Consider adding test cases that verify the actual reward claiming behavior and ensure the claimed rewards are properly added to the batch deposit requests and total pooled amount.
- In the `compound(...)` function, the expression `address(this).balance - balanceBefore` can be swapped to `balanceBefore - address(this).balance` without affecting the test suite. Consider adding test cases that verify the correct calculation of claimed rewards in relation to the existing contract balance.
- In the `compound(...)` function, the loop condition `i < len` can be changed to `len < i` without affecting the test suite. Consider adding test cases that ensure the loop performs the correct number of iterations when processing node IDs for reward claiming.
- In the `constructor()` function, the call to `Initializable._disableInitializers()` can be replaced with `assert(true)` (which is effectively a no-op) without affecting the test suite. Consider adding test cases to verify that the initialization guard mechanism is properly disabled and that subsequent re-initialization attempts are correctly prevented.
- In the `contributeToPool(...)` function, the assignment `batchDepositRequests[currentBatchId].assets += uint96(msg.value);` can be changed to `batchDepositRequests[currentBatchId].assets += 0;` with no effect on the test suite. Consider adding test cases that verify the correct accumulation of assets in the batch deposit requests structure.

- In the `convertToAssets(...)` function, the `if` statement condition: `_totalShares == 0` can be hardcoded to `false` without affecting the test suite. Consider adding test cases for the logic executed in the `if` branch of this statement.
- In the `convertToShares(...)` function, the `if` statement condition: `_totalPooled == 0` can be hardcoded to `true` without affecting the test suite. Consider adding tests for cases when the logic from the `if` branch of this statement is not executed.
- In the `deposit(...)` function, the `_updateFees()` call can be replaced with `assert(true)` (which is effectively a no-op) without affecting the test suite. Consider adding test cases to check the expected effects of the fee updates that should occur in `_updateFees()`, particularly around management fee its impact on share conversions and protocol accounting.
- In the `deposit(...)` function, the `if` statement condition: `shares == 0` can be hardcoded to `false` without affecting the test suite. Consider adding test cases for the revert with the `NoChange` error.
- In the `getMintableProtocolShares()` function, the addition in the calculation `ERC20Upgradeable.totalSupply() + shares` can be changed to subtraction without affecting the test suite. Consider adding test cases that verify the correct computation of mintable protocol shares under different scenarios involving the total supply and virtual shares snapshot.
- In the `getMintableProtocolShares(...)` function, the `if` statement condition: `time > 0` can be hardcoded to `true` without affecting the test suite. Consider adding tests for cases when the logic from the `if` branch of this statement is NOT executed.
- In the `getMintableProtocolShares(...)` function, the assignment `shares = _managementFee.virtualSharesSnapshot;` can be changed to `shares = 0;` with no effect on the test suite. Consider adding test cases that depend on the value of `shares` and verify the correct calculation of mintable protocol shares based on the virtual shares snapshot.
- In the `initialize(...)` function, the `ERC20Upgradeable.__ERC20_init("Kintsu Staked Monad", "sMON")` call can be replaced with `assert(true)` (which is effectively a no-op) without affecting the test suite. Consider adding test cases to check that the ERC20 token is properly initialized with the correct name and symbol.
- In the `initialize(...)` function, the `PausableUpgradeable.__Pausable_init()` call can be replaced with `assert(true)` (which is effectively a no-op) without affecting the test suite. Consider adding test cases to check the expected effects of the original `PausableUpgradeable.__Pausable_init()` call, particularly verifying that pause functionality is properly initialized and that the contract can be paused/unpaused as intended.
- In the `redeem(...)` function, the `if` statement condition: `!success` can be hardcoded to `false` without affecting the test suite. Consider adding test cases for the revert with the `TransferFailed` error.
- In the `redeem(...)` function, the addition in the condition `currentEpoch < batchSubmission.activationEpoch + StakerUpgradeable.getWithdrawDelay()` can be changed to exponentiation without affecting the test suite. Consider adding test cases that verify the correct behavior when the withdrawal delay is supposed to make the transaction revert/pass.
- In the `requestUnlock(...)` function, the `_updateFees()` call can be replaced with `assert(true)` (which is effectively a no-op) without affecting the test suite. Consider adding test cases to check the expected effects of the original `_updateFees()` call, particularly how fee updates impact the management fees stored in the contract state.
- In the `requestUnlock(...)` function, the `if` statement condition: `!isExitFeeExempt[msg.sender]` can be hardcoded to `true` without affecting the test suite. Consider adding test cases for scenarios where the exit fee logic is NOT executed, particularly when callers are exempt from exit fees.
- In the `requestUnlock(...)` function, the `if` statement condition: `spotValue < minSpotValue` can be hardcoded to `false` without affecting the test suite. Consider adding test cases for the revert with the `MinimumUnlock` error.
- In the `requestUnlock(...)` function, the `if` statement condition: `spotValue == 0` can be hardcoded to `false` without affecting the test suite. Consider adding test cases for the revert with the `NoChange` error.
- In the `setExitFeeExemption(...)` function, the assignment `isExitFeeExempt[user] = isExempt;` can be changed to `isExitFeeExempt[user] = false;` with no effect on the test suite. Consider adding test cases that verify the actual value being set for `isExitFeeExempt[user]` and ensure the function properly respects the `isExempt` parameter.
- In the `setManagementFee(...)` function, the `_updateFees()` call can be replaced with `assert(true)` (which is effectively a no-op) without affecting the test suite. Consider adding test cases to check the expected effects of the original `_updateFees()` call, particularly how it impacts fee distribution and protocol revenue calculations.
- In the `submitBatch(...)` function, the `ERC20Upgradeable._burn(address(this), batchWithdrawRequest.shares)` call can be replaced with `assert(true)` (which is effectively a no-op) without affecting the test suite. Consider adding test cases to check the expected effects of the token burning operation, particularly how it affects the contract's total supply and the shares balance of the contract address when withdrawal requests are processed in batches.
- In the `submitBatch(...)` function, the `_updateFees()` call can be replaced with `assert(true)` (which is effectively a no-op) without affecting the test suite. Consider adding test cases to check the expected effects of the `_updateFees()` call, particularly how fee updates impact protocol shares and management fees during batch submission operations.
- In the `submitBatch(...)` function, the `if` statement condition: `batchWithdrawRequest.assets > 0` can be hardcoded to `true` without affecting the test suite. Consider adding test cases for the logic that adjusts the `totalPooled` value.
- In the `submitBatch(...)` function, the `if` statement condition: `batchWithdrawRequest.shares > 0` can be hardcoded to `true` without affecting the test suite. Consider adding test cases for scenarios where the shares are NOT burned.

- In the `submitBatch(...)` function, the `if` statement condition: `dust > 0` can be hardcoded to `true` without affecting the test suite. Consider adding test cases around state changes when dust is NOT expected.
- In the `submitBatch(...)` function, the assignment `_exitFee.escrowShares = 0;` can be changed to `_exitFee.escrowShares = 1;` with no effect on the test suite. Consider adding test cases that verify the correct handling of escrow shares during batch submission, particularly focusing on scenarios where exit fees are processed and ensure the protocol shares are properly incremented while escrow shares are reset to zero.
- In the `submitBatch(...)` function, the assignment `_totalPooled += dust;` can be changed to `_totalPooled += 0;` with no effect on the test suite. Consider adding test cases that verify the correct updating of `totalPooled` when there is dust remaining from withdrawal requests, particularly focusing on scenarios where dust is present and should contribute to the total pooled amount.
- In the `submitBatch(...)` function, the assignment `batchDepositRequests[_currentBatchId + 1].assets = dust;` can be changed to `batchDepositRequests[_currentBatchId + 1].assets = 1;` with no effect on the test suite. Consider adding test cases that depend on the exact value of the dust amount being properly carried forward to the next batch request, particularly focusing on scenarios where the dust value impacts subsequent batch processing or total pooled calculations.
- In the `submitBatch(...)` function, the assignment `batchSubmission.activationEpoch = activityEpoch;` can be changed to `batchSubmission.activationEpoch = 0;` with no effect on the test suite. Consider adding test cases that depend on the `activationEpoch` value being correctly set to ensure proper batch processing and withdrawal delay enforcement.
- In the `submitBatch(...)` function, the assignment `batchSubmission.submissionEpoch = currentEpoch;` can be changed to `batchSubmission.submissionEpoch = 1;` with no effect on the test suite. Consider adding test cases that depend on the value of `batchSubmission.submissionEpoch` to ensure the correct epoch is being set and that the batch submission logic properly utilizes this timestamp value.
- In the `submitBatch(...)` function, the subtraction in the calculation of `dust = batchDepositRequest.assets - batchWithdrawRequest.assets - bondedAmountEvm` can be changed to addition without affecting the test suite. Consider adding test cases that verify the correct computation of the dust amount and ensure that the subsequent handling of dust (storing it in the next batch's assets) behaves as expected under different arithmetic scenarios.
- In the `sweepForced(...)` function, the `if` statement condition: `amountWithdrawn > 0` can be hardcoded to `false` without affecting the test suite. Consider adding test cases for scenarios where funds are actually withdrawn and the logic within the `if` branch is executed, particularly verifying that `batchDepositRequests[currentBatchId].assets` is correctly incremented when funds are available.
- In the `sweepForced(...)` function, the expression `address(this).balance - balanceSnapshot` can be swapped to `balanceSnapshot - address(this).balance` without affecting the test suite. Consider adding test cases that verify the correct calculation of `amountWithdrawn` when funds are withdrawn from forced node sweeps.
- In the `unbondDisableNode(...)` function, the assignment `node.staked = 0;` can be changed to `node.staked = 1;` with no effect on the test suite. Consider adding test cases that verify the correct updating of node staked amounts and validate that the staked value is properly set to zero after unbonding a disabled node.

src/precompile/StakerUpgradeable.sol

- In the `_undelegate(...)` function, the `if` statement condition: `!success` can be hardcoded to `false` without affecting the test suite. Consider adding test cases for the revert with the `PrecompileCallFailed` error.
- In the `claimRewards(...)` function, the `if` statement condition: `!success` can be hardcoded to `false` without affecting the test suite. Consider adding test cases for the revert with the `PrecompileCallFailed` error.
- In the `delegate(...)` function, the `if` statement condition: `!success` can be hardcoded to `false` without affecting the test suite. Consider adding test cases for the revert with the `PrecompileCallFailed` error.
- In the `undelegate(...)` function, the call to `_undelegate(val_id, amount, withdrawId)` can be replaced with `assert(true)` (which is effectively a no-op) without affecting the test suite. Consider adding test cases that verify the correct execution of the `_undelegate` function call and ensure that the precompile undelegation operation is properly invoked during normal undelegation flows.
- In the `undelegateForced(...)` function, the `_undelegate(val_id, amount, withdrawId)` call can be replaced with `assert(true)` (which is effectively a no-op) without affecting the test suite. Consider adding test cases to check the expected effects of the original `_undelegate(val_id, amount, withdrawId)` call, particularly verifying that the precompile undelegation function is properly invoked.
- In the `undelegateForced(...)` function, the `if` statement condition: `$.isForceWithdrawPending[val_id]` can be hardcoded to `false` without affecting the test suite. Consider adding test cases for the revert with the `MaxPendingWithdrawals` error.
- In the `withdraw(...)` function, the `if` statement condition: `!success` can be hardcoded to `false` without affecting the test suite. Consider adding test cases for the revert with the `PrecompileCallFailed` error.
- In the `withdraw(...)` function, the `if` statement condition: `withdrawCount == 0` can be hardcoded to `false` without affecting the test suite. Consider adding test cases for when `withdrawCount` equals zero, ensuring the early return logic is properly covered.
- In the `withdraw(...)` function, the loop condition `i < withdrawCount` can be changed to `withdrawCount < i` without affecting the test suite. Consider adding test cases that ensure the loop performs the correct number of iterations when processing pending withdrawals.

- In the `withdrawForced(...)` function, the `if` statement condition: `!$.isForceWithdrawPending[val_id]` can be hardcoded to `false` without affecting the test suite. Consider adding test cases for the revert with the `InsufficientPendingWithdrawals` error.
- In the `withdrawForced(...)` function, the `if` statement condition: `!success` can be hardcoded to `false` without affecting the test suite. Consider adding test cases for the revert with the `PrecompileCallFailed` error.

8.2 Automated Tools

8.2.1 AuditAgent

All the relevant issues raised by the AuditAgent have been incorporated into this report. The AuditAgent is an AI-powered smart contract auditing tool that analyses code, detects vulnerabilities, and provides actionable fixes. It accelerates the security analysis process, complementing human expertise with advanced AI models to deliver efficient and comprehensive smart contract audits. Available at https://app.auditagent.nethermind.io.

9 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

This report is based on the scope of materials and documentation provided by you to Nethermind in order that Nethermind could conduct the security review outlined in **1. Executive Summary** and **2. Audited Files**. The results set out in this report may not be complete nor inclusive of all vulnerabilities. Nethermind has provided the review and this report on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Blockchain technology remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. This report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on this report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.