NM-0555 - Surge Rollup

(August 4, 2025)

Contents

1 Executive Summary

This document presents the security review performed by Nethermind Security for the Surge Rollup Inbox smart contracts from Jul. 21, 2025 to Aug. 1, 2025. Surge is a based rollup that extends Taiko Alethia, designed to be maximally aligned with Ethereum's decentralisation principles by relying on Ethereum's transaction ordering. It aims to meet Stage 2 rollup requirements by enforcing a time-lock and mandating a liveness streak for protocol upgrades.

Surge uses Sp1 and Risc0 ZK provers along with SGX and TDX Trusted Executed Environments to avoid a single point of failure when proving transactions that provide finality to the L2.

Specifically, within the reviewed scope, Surge handles proposition of transactions in batches of blocks, submission of proofs for batches, verification, handling of conflicting state transitions and provide Rollup Stage 2 requirements.

The audited code comprises 1733 lines of code written in the Solidity language.

The audit was performed using (a) manual analysis of the codebase and (b) automated analysis tools.

Along this document, we report six points of attention, where one is classified as `High`, three are classified as `Medium`, and two are classified as `Low`. **Fig. 1** summarizes the issues.

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 concludes the document.

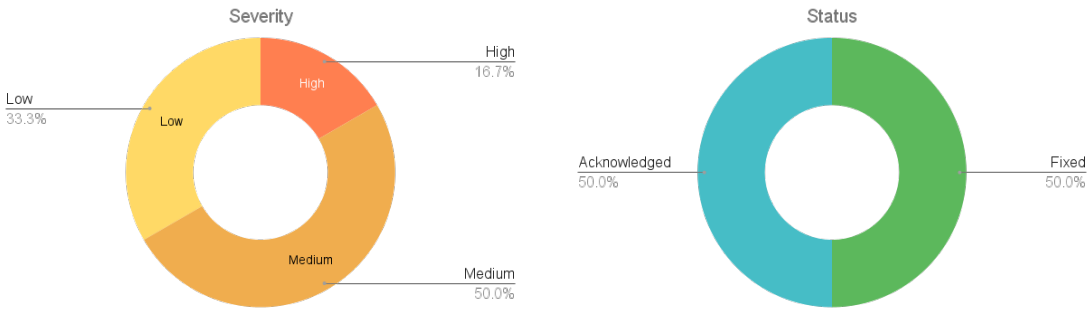

Fig. 1: Distribution of issues: Critical (0), **High** (1), **Medium** (3), **Low** (2), **Undetermined** (0), **Informational** (0), **Best Practices** (0). **Distribution of status: Fixed** (3), **Acknowledged** (3), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security review
Initial Report	July 21, 2025
Final Report	October 7, 2025
Response from Client	Regular responses during audit engagement
Repository	surge-taiko-mono
Audit Commit	`b23d6371abaa9271053d26e0278f54168b150a93`
Final Commit	`95e7567a8b621618c1f2cc25dd86a4f8cc6fdd2c`
Documentation	Communication documents and diagrams
Documentation Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	.../layer1/based/TaikoInbox.sol	566	147	25.97%	136	848
2	.../layer1/based/ITaikoInbox.sol	223	175	78.47%	42	440
3	.../layer1/based/LibVerifying.sol	164	32	19.51%	43	239
4	.../shared/common/EssentialContract.sol	153	67	43.79%	41	261
5	.../layer1/surge/common/SurgeTimelockController.sol	123	29	23.58%	29	181
6	.../layer1/surge/verifiers/SurgeVerifier.sol	107	15	14.01%	16	138
7	.../layer1/surge/SurgeHoodiInbox.sol	63	9	14.28%	7	79
8	.../layer1/surge/verifiers/LibProofType.sol	51	17	33..33%	16	84
9	.../shared/libs/LibAddress.sol	49	24	48.98%	6	79
10	.../shared/libs/LibNetwork.sol	39	22	56.41%	9	70
11	.../shared/common/DefaultResolver.sol	34	18	52.94%	10	62
12	.../shared/libs/LibStrings.sol	33	3	9.09%	1	37
13	.../shared/libs/LibBytes.sol	28	13	46.43%	4	45
14	.../layer1/surge/verifiers/ISurgeVerifier.sol	24	18	46.43%	8	50
15	.../shared/common/ResolverBase.sol	17	3	17.65%	3	23
16	.../layer1/mainnet/libs/LibFasterReentryLock.sol	15	8	53.33%	3	26
17	.../shared/common/IResolver.sol	12	11	91.67%	2	25
18	.../shared/libs/LibMath.sol	12	12	100.00%	3	27
19	.../layer1/based/IProposeBatch.sol	10	11	110.00%	2	23
20	.../layer1/verifiers/IVerifier.sol	10	7	70%	3	30
	Total	**1733**	**638**	**36.81%**	**383**	**2767**

3 Summary of Issues

The table below presents a summary of issues detected in the contracts and circuits.

	Finding	Severity	Status
1	Front-running can steal deposited funds	High	Acknowledged
2	The verification streak check can be bypassed with incorrect batch	Medium	Acknowledged
3	Lack of setting finalisingTransitionIndex to synchronised batch	Medium	Fixed
4	Lack of resolving ZK+TEE conflicting transitions	Medium	Acknowledged
5	Finalisation logic does not match the specification	Low	Fixed
6	Unused pausing logic	Low	Fixed

4 System Overview

Surge is a rollup protocol based on the Taiko architecture, adapted and extended by Nethermind to emphasize decentralization, multi-prover support, and Ethereum-native alignment. In this section, we describe the core smart contract components that make up the protocol, particularly focusing on the scoped smart contracts of Layer 1 and Shared directories.

The main differences between Taiko and Surge come down to the minimization of trust and decentralization. Surge takes on the Taiko approach to leverage Ethereum security with Stage 2 security in mind, while reducing trust in provers by verifying multiple proofs for the same state transition, i.e., two ZK provers and two Trusted Executed Environment provers.

The Layer 1 contracts represent the core of the protocol. They handle proposals of blocks and verifications, as well as the finality gadget for Layer 2. For more information, please check the Taiko Alethia and Surge docs.

4.1 Contract

The main contract is called `TaikoInbox.sol` — this is the Layer 1 entry-point for proposing blocks of transactions in batches to be executed in the Layer 2 rollup.

Through this contract, anyone can permissionlessly propose a batch by submitting well-formed transaction data and providing a bond to assert the validity of the resulting state transition.

Secondly, anyone can submit proofs for previously proposed batches by specifying the proof type and invoking the corresponding verifier contract associated with that prover. Once batches accumulate at least two different proof types, the block can be finalized through the finality gadget.

The finality gadget is executed within the `verifyBatches(...)` function, specifically through the `_tryFinalising(...)` function defined in the `LibVerifying.sol` file. This mechanism runs during batch proposal or batch proving and represents the core update to the Surge protocol. It aims to:

- Check how many state transitions there are; if there is only one state transition and it lacks both proof types, and a waiting period of 7 days has passed, it returns as a finalized block
- If both proofs are found, this is considered a finalized block
- If multiple transitions are found, the contract iterates through them. The last transition with both proofs is finalized, and any intermediate transitions lacking both proofs are flagged for potential verifier contract upgrades
- Following these, a `bondReceiver` is assigned to the last proof submitter
- For a single transition, if only one proof type is submitted and the second proof type is not submitted in 7 days and no challenging transition is provided, such transition is considered finalising. In such a case, the proposer's bond is assigned to the DAO.

To meet Stage 2 upgradability requirements, Surge introduces a safety mechanism based on batch verification activity. If no batch has been finalized in the past 7 days, either the next verification or the proposal of a new batch will trigger a 45-day cooldown period. This cooldown gives users time to safely exit before any protocol upgrade — except for verifier contract upgrades, which are managed separately.

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 exploitation.

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.

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

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 and circuit development best practices;

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

6 Issues

6.1 [High] Front-running can steal deposited funds

File(s): TaikoInbox.sol

Description: Verifying the batch of Surge blocks is made on-chain in two steps: the batch is proposed, and then the transition with the corresponding proof is provided. The proposed batch needs to deposit funds that act as a bond. If the proof of correct transition is submitted in the proving window (24 hours), the bond is returned to the proposer. However, if that time has passed, any prover can submit the proof, and such a prover would collect the funds deposited by the proposer. This is done by storing the `msg.sender` address of the prover in the TaikoInbox contract:

```
1 __ts.bondReceiver = inProvingWindow ? meta.proposer : msg.sender;
```

There are also other cases where the prover challenges the previously submitted proof, in which case such a prover would collect the bond funds as well:

```
1 transitions[mti].bondReceiver = msg.sender;
2 ...
3 _ts.bondReceiver = msg.sender;
```

However, such transactions can be front-run by anyone with the exact same data. This allows any entity to steal the funds from the prover that actually constructed the correct proof. Note however, that the risk of front-running can be avoided if the proof is provided in the proving window, since in that case the bond goes back to the proposer.

Recommendation(s): Consider including the bond receiver address into the proof, which would allow only the proof creator to safely submit the proof without the risk of losing the reward.

Status: Acknowledged

Update from the client: We acknowledge this issue.

We intend to fix this in a future fork, and not immediately for the following reasons:

- The effect of the issue is highly restrictive:
  - Given the high cost of running a prover infrastructure and considering Surge's positioning of "a not-for-profit rollup", Nethermind is likely the only entity that is going to run the prover for Surge mainnet.
  - We are keeping a proving window of 24 hours at launch, and setting up a sufficiently fault-tolerant infrastructure so as to not miss the initial proving window.
- The solution requires effort from both the smart contract and prover engineers, which may cause potential delays in launch.

6.2 [Medium] The verification streak check can be bypassed with incorrect batch

File(s): TaikoInbox.sol

Description: One of the requirements for a Stage 2 rollup implemented by Surge is a 45-day upgrade timelock, with execution allowed only if there has not been any period of a week or more in the past 45 days during which no batches were verified. The verification delay checkpoint is registered in `verifyBatches(...)` function:

```
if (block.timestamp - batch.lastBlockTimestamp > _config.maxVerificationDelay) {
    _state.stats1.verificationStreakStartedAt = uint64(block.timestamp);
}
```

However, a verification delay detection can be avoided by proposing and proving the incorrect batches. Since proposing the batch with `proposeBatch(...)` is permissionless, any caller can propose a batch that is incorrect. Such an incorrect batch is treated as empty when proving the transition. Such transition is accepted on-chain with the `batch.lastBlockTimestamp` set to an arbitrary timestamp (less than the current timestamp). As a result, the L2 chain could be inactive for over a week, but the on-chain verification delay check would not detect that. Note, however, that this scenario can only happen if no batches are proposed at all.

Recommendation(s): Consider performing the verification delay on the last correct batch.

Status: Acknowledged

Update from the client: This issue can happen only if the chain completely halts for over a week (no batches are proposed). In that case, the described avoidance of the check is not seen as a problem. The verification streak check exists for the case where the batches are proposed, but the verification is delayed. Moreover, clarification on incorrect and unprovable batches is described below:

- An "incorrect batch" is not the same as an "unprovable batch".
- A batch is deemed "incorrect" if the transactions listed in the blob cannot be decoded into a list of valid RLP encoded transactions.
  - In this case, the prover overwrites the faulty transaction list with an empty transaction list.
  - Prover reference: raiko
  - **This is a provable batch and cannot halt the chain.**
- A batch is deemed "unprovable" when the prover panics, or is not able to generate the proof at all.
  - This is possible due to (but not limited to):
    - A bug in the prover software or in the underlying execution client.
    - A bug or a limitation in the underlying zkVM suite.
  - **Such a batch is unprovable, not verifiable on-chain, and can halt the chain.**
  - Currently, extensive research is ongoing to identify and bypass prover killer blocks.
  - We believe there is no on-chain strategy to immediately mitigate this.

6.3 [Medium] Lack of setting finalisingTransitionIndex to synchronised batch

File(s): TaikoInbox.sol

Description: When the batch is verified, the synchronisation happens in verifyBatches(...). The state synchronisation happens for every other block. Only the last batch in the group of proven batches is updated in the verifiedTransitionId and finalisingTransitionIndex fields. A situation can occur where the synchronized batch is not the last verified batch. In such a situation, this batch's verifiedTransitionId field also is updated:

```
function verifyBatches(...)
    public
{
    ...
        for (++ls.batchId; ls.batchId < ls.stopBatchId; ++ls.batchId) {
            ...

            if (ls.batchId % _config.stateRootSyncInternal == 0) {
                synced.batchId = ls.batchId;
                synced.blockId = batch.lastBlockId;
                synced.tid = ls.tid;
                synced.stateRoot = transitions[ls.fti].stateRoot;
            }
        }

        unchecked {
            --ls.batchId;
        }

        if (_stats2.lastVerifiedBatchId != ls.batchId) {
            _stats2.lastVerifiedBatchId = ls.batchId;

            batch = _state.batches[_stats2.lastVerifiedBatchId % _config.batchRingBufferSize];
            // @audit: update of the verified batch
            batch.verifiedTransitionId = ls.tid;
            // Surge: update the finalising transition index for the batch
            batch.finalisingTransitionIndex = uint8(ls.fti);

            emit ITaikoInbox.BatchesVerified(_stats2.lastVerifiedBatchId, ls.blockHash);

            if (synced.batchId != 0) {
                if (synced.batchId != _stats2.lastVerifiedBatchId) {
                    // We write the synced batch's verifiedTransitionId to storage
                    batch = _state.batches[synced.batchId % _config.batchRingBufferSize];
                    // @audit: update of the synchronised
                    batch.verifiedTransitionId = synced.tid;
                    // @audit: lack of update of
                    // finalisingTransitionIndex for sync batch
                }

                ITaikoInbox.Stats1 memory stats1 = _state.stats1;
                stats1.lastSyncedBatchId = batch.batchId;
                stats1.lastSyncedAt = uint64(block.timestamp);
                _state.stats1 = stats1;

                emit ITaikoInbox.Stats1Updated(stats1);

                // Ask signal service to write cross chain signal
                _signalService.syncChainData(
                    _config.chainId, LibStrings.H_STATE_ROOT, synced.blockId, synced.stateRoot
                );
            }
        }
    ...
}
```

However, the synchronised batch does not update the `finalisingTransitionIndex` field. This creates the issue because the function `getBatchVerifyingTransition(...)` would return the first conflicting transition, which might not to be the correct transition that proved the batch.

```
function getBatchVerifyingTransition(uint64 _batchId)
    public
    view
    returns (TransitionState memory ts_)
{
    Config memory config = pacayaConfig();

    uint64 slot = _batchId % config.batchRingBufferSize;
    Batch storage batch = state.batches[slot];
    require(batch.batchId == _batchId, BatchNotFound());

    if (batch.verifiedTransitionId != 0) {
        // @audit: since finalisingTransitionIndex is not set
        // it would be 0 and would return the first transition
        // but it may not be the correct transition if there are conflicting
        ts_ =
            state.transitions[slot][batch.verifiedTransitionId][batch.finalisingTransitionIndex];
    }
}

```

Recommendation(s): Consider updating the finalisingTransitionIndex for the synchronised batch.

Status: Fixed

Update from the client: Fixed in the commit 35663b74b13ac517593366909a81bfc9c2cc1a8b

6.4 [Medium] Lack of resolving ZK+TEE conflicting transitions

File(s): TaikoInbox.sol

Description: The conflicting transitions are resolved in the finality gadget. However, currently only the conflicting transitions with a single proof type are resolved. The transitions with full proof containing both ZK and TEE proofs are not handled, and the transition provided as last is used as the finilising.

```
 function _tryFinalising(...)
    internal
    returns (uint256)
{
    ...
    if (numTransitions == 1) {
        // If the first transition is just ZK or TEE proven
        if (!_transitions[0].proofType.isZkTeeProof()) {
            if (_transitions[0].createdAt + _config.cooldownWindow > block.timestamp) {
                return fti;
            }
        }

        // The first transition itself is the finalising transition
        fti = 0;
    } else {
        // Proof type(s) to upgrade
        LibProofType.ProofType ptToUpgrade;

        // Try to find a finalising proof
        for (uint256 i; i < numTransitions; ++i) {
            //@audit: the last conflicting ZK+TEE proof
            //        is chosen as the finalising
            if (_transitions[i].proofType.isZkTeeProof()) {
                fti = i;
            } else {
                ptToUpgrade = ptToUpgrade.combine(_transitions[i].proofType);
            }
        }
        ...
}
```

Recommendation(s): Consider handling the conflicting transitions with ZK and TEE proofs.

Status: Acknowledged

Update from the client: We believe this does not require an immediate fix:

- While not impossible, it is improbable that a transition proven by both ZK and TEE will receive a conflict.
- Another strategy to mitigate the risk immediately on our end would be to incorporate different implementations of the execution client for ZK and TEE. For instance, Reth (Rust client) with ZK and Geth (Golang client) with TEE.

6.5 [Low] Finalization logic does not match the specification

File(s): TaikoInbox.sol

Description: The specification provided by the developers specifies that the ZK proof has the priority over the TEE proof. This means that if there are conflicting ZK and TEE proofs, the challenging TEE proof is ignored and the batch can be finalised by the ZK proof alone (after the cooldown of 7 days). However, the logic in the `_tryFinalising(...)` treats both ZK and TEE proofs as equal, meaning that if we challenge the ZK proof with the TEE proof, the protocol waits for a complementary proof or a new full (ZK+TEE) proof.

```
function _tryFinalising(...)
    internal
    returns (uint256)
{
    ...
    if (numTransitions == 1) {
        ...
    } else {
        // @audit: this branch is executed if
        // conflicting transitions are recorded

        // Proof type(s) to upgrade
        LibProofType.ProofType ptToUpgrade;

        // Try to find a finalising proof
        for (uint256 i; i < numTransitions; ++i) {
            // @audit: the transition is only considered
            // as finalising if both ZK+TEE are submitted
            if (_transitions[i].proofType.isZkTeeProof()) {
                fti = i;
            } else {
                ptToUpgrade = ptToUpgrade.combine(_transitions[i].proofType);
            }
        }

        // If no finalising transition is found, we return
        if (fti == type(uint256).max) {
            return fti;
        } else {
            // Mark non finalising verifiers for upgrade
            ISurgeVerifier(_verifier).markUpgradeable(ptToUpgrade);
        }
    }
    ...
}
```

This logic is different from the one specified in the documentation.

Recommendation(s): Consider aligning the implemented logic with the documentation.

Status: Fixed

Update from the client: The documentation (flow diagram) precedes the code implementation. Based on later decisions, we made the implementation more generic compared to the diagram. We believe updating the diagram and polishing the documentation would be the correct approach here.

6.6 [Low] Unused pausing logic

TaikoInbox.sol

Description: The pausing logic is implemented in the `TaikoInbox`, however there is no possibility of pausing the protocol. Below we list the pausing logic in the `TaikoInbox`:

- `_pause(...)` `_unpause(...)` functions
- `paused(...)` view function
- `require(!stats2.paused, ContractPaused());` in `proveBatches(...)`
- `lastUnpausedAt` in `inProvingWindow = block.timestamp <= uint256(meta.proposedAt).max(stats2.lastUnpausedAt) + config.provingWindow;` in the `proveBatches(...)`
- the `whenNotPaused(...)` modifier

Recommendation(s): Consider removing unused code that handles pausing the contract.

Status: Fixed

Update from the client: Fixed in the commit a99ef204c12d55df664867ea535fe9a33140725f.

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 documentation

The client provided an extensive explanation of the project during the calls. Also, additional documents and diagrams were provided.

8 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 is dedicated to continuous internal research while fostering close collaboration with external partners. The team has expertise across a wide range of domains, including cryptography protocols, consensus design, decentralized identity, verifiable credentials, Sybil resistance, oracles, and credentials, distributed validator technology (DVT), and Zero-knowledge proofs. This diverse skill set, combined with strong collaboration between our engineering teams, enables us to deliver cutting-edge solutions to our 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.

Learn more about us at nethermind.io.

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