Security Review Report
NM-0620 zkLighter Bridge

NETHERMIND
SECURITY

(September 22, 2025)

Contents

1 Executive Summary

This document presents the results of the security review conducted by Nethermind Security for zkLighter's bridge contracts. The **zkLighter bridge** contracts enable cross-chain deposits of USDC into the zkLighter protocol, leveraging **Circle's Cross-Chain Transfer Protocol (CCTP)** as the underlying bridging infrastructure.

The deposit flow begins on the `TokenBurnerSystemV2` contract, which pulls user funds from their corresponding `EphemeralTokenBurnerV2` contracts and deposits them into CCTP for burning on the source chain. Once the burn is confirmed, a corresponding mint is triggered on the destination chain. The minted tokens are then handled by the `FastCCTPV2` contract, which completes the process by depositing them into the `zklighterProxy` contract.

The audit comprises 295 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** five points of attention, where they are classified as `Informational`. 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.

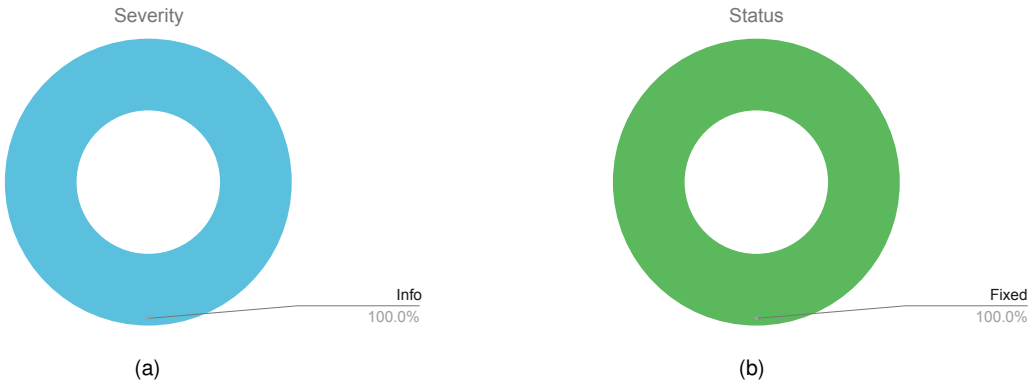

Fig. 1: Distribution of issues: Critical (0), **High** (0), **Medium** (0), **Low** (0), **Undetermined** (0), **Informational** (5), **Best Practices** (0). **Distribution of status: Fixed** (5), **Acknowledged** (0), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	August 31, 2025
Final Report	September 22, 2025
Initial commit	522de47bcab8dfd16c92522ede4aaa2e9a22a397
Final commit	7d6b776fff4eebcc5ade56f1350c1e487c325e9a
Documentation Assessment	Low
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	EphemeralTokenBurnerV2.sol	34	2	5.9%	9	45
2	TokenBurnerSystemV2.sol	160	9	5.6%	42	211
3	FastCCTPV2.sol	101	2	2.0%	28	131
	Total	**295**	**13**	**4.4%**	**79**	**387**

3 Summary of Issues

	Finding	Severity	Update
1	Excess contract balance will be included alongside user requests	Info	Fixed
2	Missing safeguards in `setGovernor` could lock governance	Info	Fixed
3	Possible fee overcharge for users	Info	Fixed
4	Unlimited USDC approval to CCTP `tokenMessenger`	Info	Fixed
5	`EphemeralTokenBurnerV2` cannot receive native tokens	Info	Fixed

4 System Overview

The `zkLighter Bridge` facilitates cross-chain deposits of USDC into the zkLighter protocol, utilizing Circle's Cross-Chain Transfer Protocol (CCTP) as the underlying bridging infrastructure. The system operates across two chains: a source chain where deposits are initiated and a destination chain where they are finalized.

The architecture is composed of three main contracts:

- **TokenBurnerSystemV2**: Deployed on the source chain, this contract manages the deposit and bridging initiation process.
- **FastCCTPV2**: Deployed on the destination chain, this contract verifies Circle attestations and finalizes deposits into `zkLighterProxy`.
- **EphemeralTokenBurnerV2**: A lightweight per-user contract deployed deterministically for each deposit intent. Users pre-fund their assigned ephemeral burner contract with USDC. The contract holds funds temporarily until they are swept into the `TokenBurnerSystemV2`. It includes a `rescueMoney` function as a safeguard, allowing recovery of tokens.

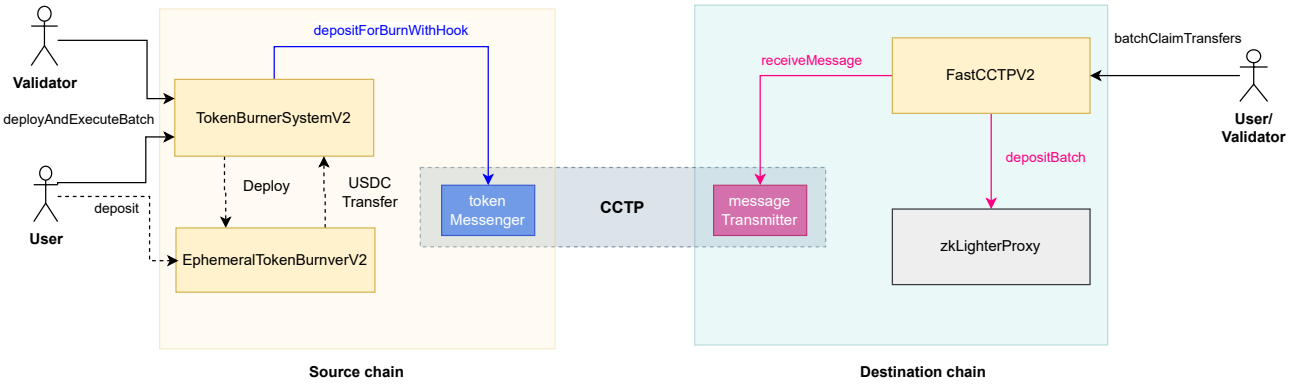

Fig. 2: zkLighter bridge overview

4.1 Source Chain Flow

On the source chain, users place their USDC into ephemeral contracts. The system then gathers these deposits and burns them via Circle's CCTP, preparing the transfer to the destination chain. The process unfolds as follows:

- Users pre-fund their deterministically computed `EphemeralTokenBurnerV2` address with USDC. The address is derived using the depositor's account, the claim contract, and recipient information.
- The `TokenBurnerSystemV2` contract's `deployAndExecuteBatch` function is then called. A validator (or the user) triggers this function, which:
  a. Deploys any necessary ephemeral burners.
  b. Collects USDC balances from all funded burners in the batch.
  c. Initiates a CCTP `depositForBurnWithHook` call in the `TokenMessenger` contract, which burns the USDC on the source chain and sends a cross-chain message to the destination chain, where the deposit details are encoded in the hook (recipients, amounts, and fee information).

The CCTP applies a bridging fee that depends on the transfer mode: standard or fast. In standard mode, no fee is charged, while fast mode applies a 0.01% fee on the transferred amount. In addition to Circle's fee, `zkLighter` applies its own execution fees, which are encoded in the message `hook`. All `zkLighter` fee parameters are configurable by the `governor` address.

4.2 Cross-Chain Transfer

Once the burn is initiated on the source chain, Circle's Cross-Chain Transfer Protocol takes over the bridging process. Circle's Attestation Service continuously monitors the burn events and issues a signed attestation authorizing the corresponding mint on the destination chain upon verification. This attestation acts as a cryptographic proof that the USDC burned on the source chain can be safely re-minted. Both the message containing the hook data and the attestation are required inputs for completing the transfer on the destination chain. In this way, CCTP guarantees that every minted USDC on the destination side is backed by an equivalent burn on the origin side, ensuring system solvency.

4.3 Destination Chain Flow

On the destination chain, the bridging process is completed through the `FastCCTPV2` contract. Users or validators invoke the `batchClaimTransfers` function, providing the Circle message alongside its attestation. The contract verifies the attestation through Circle's `MessageTransmitter`, ensuring authenticity of the transfer, and then decodes the embedded hook data to retrieve the list of depositors, balances, and fee parameters.

The contract computes the final credited deposits by applying zkLighter's fee logic, which combines Circle's base bridging fee (0.01% in fast mode, none in standard mode) with zkLighter's configurable fee parameters defined by the governor. These parameters consist of a proportional fee (`feeTicker`) and a fixed per-deposit fee (`feeAddition`). The net amounts are then deposited into zkLighter through a single batched call to `zkLighterProxy.depositBatch`, efficiently crediting the intended recipients.

If the attested USDC amount received from Circle is lower than the expected deposit sum due to fees, non-validator callers are required to provide the shortfall directly in USDC. Validators, however, are exempted from this requirement, allowing them to act as relayers without needing to pay for the CCTP 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 [Info] Excess contract balance will be included alongside user requests

File(s): TokenBurnerSystemV2.sol

Description: In the `TokenBurnerSystemV2` contract, the `deployAndExecuteBatch` function computes a `totalBalance` by summing the amounts of all burn requests in a batch. This `totalBalance` is intended to represent the exact number of tokens that should be burned on the source chain and minted on the destination chain.

Currently, the contract instead deposits its entire `USDC` balance when calling `_tokenMessenger.depositForBurnWithHook`. As a result, any additional tokens held by the contract, such as donations or other incoming funds, are inadvertently included in the burn operation, potentially exceeding the intended amount.

```
function _receiveTransfer(
    uint32 minFinalityThreshold,
    bytes memory hookData,
    uint256 totalBalance,
    uint256 cctpMaxFee,
    address claimContract
  ) internal {
    uint256 receivedAmount = _usdcToken.balanceOf(address(this));// @audit
    require(receivedAmount >= totalBalance, "FCCTP: insufficient balance received");

    bytes32 receiver = bytes32(uint256(uint160(claimContract)));

    _tokenMessenger.depositForBurnWithHook(
      receivedAmount,
      DESTINATION_DOMAIN,
      receiver,
      address(_usdcToken),
      receiver,
      cctpMaxFee,
      minFinalityThreshold,
      hookData
    );
  }
```

Recommendation(s): Revisit the intended behaviour and update the logic to deposit only the `totalBalance` instead of the full contract balance if necessary.

Status: Fixed

Update from the client: Updated as recommended. 005da7a5b94e824a145503b123788192778aa495

6.2 [Info] Missing safeguards in `setGovernor` could lock governance

File(s): TokenBurnerSystemV2.sol

Description: The `setGovernor` function replaces the current governor with a new address. Currently, this function does **not** include any validation checks. As a result, if an incorrect address (e.g., `address(0)`) is set, all functions restricted to the governor could become permanently inaccessible, potentially locking critical administrative operations.

Recommendation(s): Consider enforcing validation to prevent setting the governor to a zero address. Additionally, consider implementing a two-step governance update process to reduce the risk of accidental misconfiguration.

Status: Fixed

Update from the client: Added zero address check for setGovernor. 005da7a5b94e824a145503b123788192778aa495

6.3 [Info] Possible fee overcharge for users

File(s): TokenBurnerSystemV2.sol

Description: When sending a message through CCTP, the parameters `cctpMaxFee` and `minFinalityThreshold` determine whether the message is executed in fast mode or standard mode, with Circle charging fees accordingly. Lighter additionally applies its own fees based on the execution mode, which is indicated by the `isFast` flag.

A discrepancy arises when `isFast` is set to `true`, but the provided `cctpMaxFee` is insufficient to meet Circle's minimum requirement for fast execution. In such cases, CCTP automatically falls back to standard mode, but the Lighter contract still charges the higher fast-mode fee based solely on the `isFast` flag.

As a result, users may be overcharged: they pay for a fast execution fee by Lighter, while the message is actually processed in standard mode by CCTP. As a consequence, the user will be charged by Lighter fast mode fees for a standard executiion.

Recommendation(s): Consider introducing validation to ensure that the `isFast` flag and `cctpMaxFee` are consistent with the execution mode that CCTP will process.

Status: Fixed

Update from the client: Added minCCTPFee to make sure fee params are aligned. 005da7a5b94e824a145503b123788192778aa495

6.4 [Info] Unlimited USDC approval to CCTP `tokenMessenger`

File(s): TokenBurnerSystemV2.sol

Description: Upon deployment, the `TokenBurnerSystemV2` contract grants the CCTP `tokenMessenger` contract an unlimited USDC allowance (`type(uint256).max`). This allows `tokenMessenger` to transfer any amount of USDC held by the contract without requiring re-approval. In practice, the allowance is only consumed during calls to `depositForBurnWithHook`, which are executed whenever a burn operation is performed.

Although this design is functional under the current architecture, granting unlimited approvals introduces a systemic risk. If the `tokenMessenger` contract were ever upgraded, compromised, or misconfigured, it would have the ability to spend the entire USDC balance of the contract. This approach violates the principle of least privilege, as the contract unnecessarily exposes its funds to the external `tokenMessenger` logic.

Recommendation(s): Replace unlimited approvals with per-transaction approvals by approving only the `depositForBurnWithHook` amount before each call.

Status: Fixed

Update from the client: Updated as recommended. 005da7a5b94e824a145503b123788192778aa495

6.5 [Info] `EphemeralTokenBurnerV2` cannot receive native tokens

File(s): EphemeralTokenBurnerV2.sol

Description: The `rescueMoney` function is intended to transfer the contract's balance to the specified `_zklighterRecipient`. When the `token` parameter is passed as `address(0)`, the function attempts to transfer native tokens.

However, the `EphemeralTokenBurnerV2` contract does not implement any mechanism to receive native tokens. Consequently, the native token handling logic in `rescueMoney` is redundant and will never execute in practice.

```
function rescueMoney(IERC20 token) external {
  require(msg.sender == _creator, "FCCTP: only creator");
  if (address(token) == address(0)) {
    (bool success, ) = _zklighterRecipient.call{value: address(this).balance}("");
    require(success, "Transfer failed");
    return;
  }
  // ...
}
```

Recommendation(s): Consider removing the native token handling logic in `rescueMoney` to avoid confusion.

Status: Fixed

Update from the client: Removed the native token handling logic. 005da7a5b94e824a145503b123788192778aa495

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 zkLighter's documentation

The zkLighter team provided a clear technical overview of the contracts in scope during regular calls and responded thoroughly to all questions raised by the Nethermind Security team, offering valuable insights and a deeper understanding of the protocol's technical aspects.

However, the contracts lack formal written documentation and `Natspec` comments that would explicitly describe the system architecture, core components, and expected functionality. Adding these resources would significantly improve maintainability and make the protocol easier to audit and extend in the future.

8 Test Suite Evaluation

8.1 Compilation Output

```
> npx hardhat compile
[dotenv@17.2.1] injecting env (2) from .env -- tip:  version env with Radar: https://dotenvx.com/radar
Warning: Unused function parameter. Remove or comment out the variable name to silence this warning.
 --> contracts/mocks/MockZkLighter.sol:7:76:
  |
7 |   function depositBatch(uint64[] calldata _amount, address[] calldata _to, uint48[] calldata _accountIndexes)
↪  external {
  |                                                                            ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^


Warning: Unused function parameter. Remove or comment out the variable name to silence this warning.
  --> contracts/mocks/MockZkLighter.sol:15:28:
   |
15 |   function getAccountIndex(address _addr) public pure returns (uint64) {
   |                            ^^^^^^^^^^^^^


Generating typings for: 25 artifacts in dir: typechain-types for target: ethers-v6
Successfully generated 76 typings!
Compiled 25 Solidity files successfully (evm target: cancun).
```

8.2 Tests Output

```
> npx hardhat test
[dotenv@17.2.1] injecting env (2) from .env -- tip:  enable debug logging with { debug: true }


  FastCCTP.batchClaimTransfers
     should fail on invalid attestation length
     should successfully claim for single bridge (47ms)
     should successfully claim for two bridges - single deposit each - with fee (39ms)
     should successfully claim for single bridge - two deposits - with zero fee

  FastCCTPV2 governance and admin functions
    setGovernor
       allows governor to change governor
       reverts if non-governor tries to change governor
    setValidator / isValidator
       governor can add validator
       governor can remove validator
       non-governor cannot modify validators
    withdraw
       governor can withdraw tokens
       reverts if non-governor calls withdraw
       reverts if amount is zero
TokenBurnerSystemV2
     should revert if zklighterRecipient array is empty
     should deploy and execute single
     should deploy and execute batch of two with custom amount
     should deploy and execute batch of three

  TokenBurnerSystemV2 Governance
    setGovernor
       should allow current governor to update governor
       should revert if non-governor calls
    setValidator / isValidator
       should allow governor to add a validator
       should allow governor to remove a validator
       should revert if non-governor calls
```

```
    setFee
      should allow governor to update fees
      should revert if non-governor calls
    setCCTPMaxFee
      should allow governor to update CCTP max fees
      should revert if non-governor calls

  25 passing (727ms)

·-----------------------------------------------------------------------------------------------------------------·
|  Solidity and Network Configuration                                                                             |
·-----------------------------·---------------·---------------·---------------·-------------------------------·
|  Solidity: 0.8.25           ·  Optim: true  ·  Runs: 1000   ·  viaIR: true  ·     Block: 30,000,000 gas     |
·-----------------------------·---------------·---------------·---------------·-------------------------------·
|  Methods                                                                                                        |
·-----------------------------·---------------·---------------·---------------·---------------·---------------·
|  Contracts / Methods        ·  Min          ·  Max          ·  Avg          ·  # calls      ·  usd (avg)    |
·-----------------------------·---------------·---------------·---------------·---------------·---------------·
|  FastCCTPV2                 ·                                                                                   |
·-----------------------------·---------------·---------------·---------------·---------------·---------------·
|      batchClaimTransfers    ·         50,807  ·        72,843  ·        59,561  ·             6  ·           -  |
·-----------------------------·---------------·---------------·---------------·---------------·---------------·
|      setGovernor            ·              -  ·             -  ·        26,888  ·             7  ·           -  |
·-----------------------------·---------------·---------------·---------------·---------------·---------------·
|      setValidator           ·         24,349  ·        46,261  ·        38,957  ·             3  ·           -  |
·-----------------------------·---------------·---------------·---------------·---------------·---------------·
|      withdraw               ·              -  ·             -  ·        57,056  ·             3  ·           -  |
·-----------------------------·---------------·---------------·---------------·---------------·---------------·
|  MockERC20                  ·                                                                                   |
·-----------------------------·---------------·---------------·---------------·---------------·---------------·
|      mint                   ·         50,972  ·        50,984  ·        50,980  ·             9  ·           -  |
·-----------------------------·---------------·---------------·---------------·---------------·---------------·
|  TokenBurnerSystemV2        ·                                                                                   |
·-----------------------------·---------------·---------------·---------------·---------------·---------------·
|      deployAndExecuteBatch  ·        431,698  ·     1,156,147  ·       795,650  ·             6  ·           -  |
·-----------------------------·---------------·---------------·---------------·---------------·---------------·
|      setCCTPMaxFee          ·              -  ·             -  ·        61,573  ·             1  ·           -  |
·-----------------------------·---------------·---------------·---------------·---------------·---------------·
|      setFee                 ·              -  ·             -  ·       112,796  ·             1  ·           -  |
·-----------------------------·---------------·---------------·---------------·---------------·---------------·
|      setGovernor            ·              -  ·             -  ·        27,066  ·             8  ·           -  |
·-----------------------------·---------------·---------------·---------------·---------------·---------------·
|      setValidator           ·         24,394  ·        46,318  ·        43,183  ·             7  ·           -  |
·-----------------------------·---------------·---------------·---------------·---------------·---------------·
|  Deployments                                ·                                ·  % of limit   ·               |
·-----------------------------·---------------·---------------·---------------·---------------·---------------·
|  FastCCTPV2                 ·        900,240  ·       900,252  ·       900,249  ·           3 %  ·           -  |
·-----------------------------·---------------·---------------·---------------·---------------·---------------·
|  MockERC20                  ·              -  ·             -  ·       576,762  ·         1.9 %  ·           -  |
·-----------------------------·---------------·---------------·---------------·---------------·---------------·
|  MockMessageTransmitter     ·              -  ·             -  ·       125,887  ·         0.4 %  ·           -  |
·-----------------------------·---------------·---------------·---------------·---------------·---------------·
|  MockTokenMessengerV2       ·              -  ·             -  ·       134,703  ·         0.4 %  ·           -  |
·-----------------------------·---------------·---------------·---------------·---------------·---------------·
|  MockZkLighter              ·              -  ·             -  ·       175,563  ·         0.6 %  ·           -  |
·-----------------------------·---------------·---------------·---------------·---------------·---------------·
|  TokenBurnerSystemV2        ·      1,571,287  ·     1,571,299  ·     1,571,298  ·         5.2 %  ·           -  |
·-----------------------------·---------------·---------------·---------------·---------------·---------------·
|  Key                                                                                                            |
·-----------------------------------------------------------------------------------------------------------------·
|  ○  Execution gas for this method does not include intrinsic gas overhead                                       |
·-----------------------------------------------------------------------------------------------------------------·
|     Cost was non-zero but below the precision setting for the currency display (see options)                    |
·-----------------------------------------------------------------------------------------------------------------·
|  Toolchain:  hardhat                                                                                            |
·-----------------------------------------------------------------------------------------------------------------·
```

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.