Security Review Report NM-0372
Bebop Jam Settlement

NETHERMIND SECURITY

(December 9, 2024)

Contents

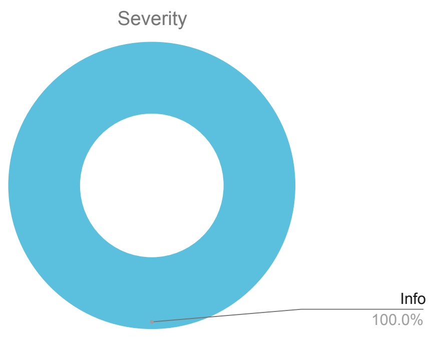
1 Executive Summary

This document outlines the security review conducted by Nethermind Security for the Bebop JAM (just-in-time aggregation model) contracts, a liquidity aggregator system allowing any token to any token swaps with efficient execution quality. Additionally, an interaction flow has been added to allow `Permit2` to work with the existing BebopBlend smart contracts.

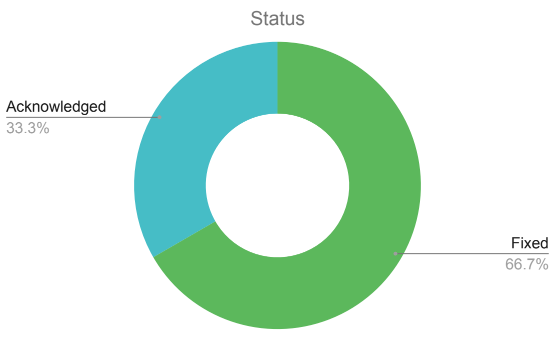
The audited code comprises of 1567 lines of code written in the Solidity language, and the audit was performed using (a) manual analysis of the codebase, (b) automated analysis tools, (c) simulation of the smart contract. At the end of the engagement after all fixes were applied, the class hashes for the base account and full account implementations are listed below:

Along this document, we report six points of attention, where one is classified as `Medium`, two are classified as `Low`, and three 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 describes the protocol overview. Section 5 discusses the risk rating methodology. Section 6 details the issues. Section 7 discusses protocol observations. Section 8 discusses the documentation provided by the client for this audit. Section 9 presents the compilation, tests, and automated tests. Section 10 concludes the document



(a) distribution of issues according to the severity



(b) distribution of issues according to the status

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

Summary of the Audit

Audit Type	Security Review
Initial Report	December 5, 2024
Final Report	December 9, 2024
Methods	Manual Review, Automated analysis
Repository	bebop-dex/bebop-jam-contracts
Commit Hash	`4ab5d579675e28cad503edd58a13e8d3c19833be`
Final Commit Hash	`b7043731d8699d34118a54e44b7ba5f54168e808`
Documentation Assessment	High
Test Suite Assessment	High

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/JamSettlement.sol	195	10	5.1%	14	219
2	src/JamBalanceManager.sol	101	11	10.9%	12	124
3	src/libraries/BlendMultiOrder.sol	60	9	15.0%	11	80
4	src/libraries/BlendAggregateOrder.sol	125	16	12.8%	12	153
5	src/libraries/JamInteraction.sol	29	2	6.9%	5	36
6	src/libraries/JamOrder.sol	52	6	11.5%	11	69
7	src/libraries/JamHooks.sol	15	4	26.7%	5	24
8	src/libraries/BlendSingleOrder.sol	36	7	19.4%	8	51
9	src/base/JamTransfer.sol	66	23	34.8%	7	96
10	src/base/JamPartner.sol	73	26	35.6%	11	110
11	src/base/JamValidation.sol	127	41	32.3%	16	184
12	src/base/Errors.sol	27	27	100.0%	27	81
13	src/interfaces/IBebopBlend.sol	51	28	54.9%	11	90
14	src/interfaces/IJamBalanceManager.sol	41	32	78.0%	8	81
15	src/interfaces/IJamSettlement.sol	44	30	68.2%	10	84
16	src/interfaces/IPermit2.sol	37	38	102.7%	10	85
	Total	**1079**	**310**	**28.7%**	**178**	**1567**

3 Summary of Issues

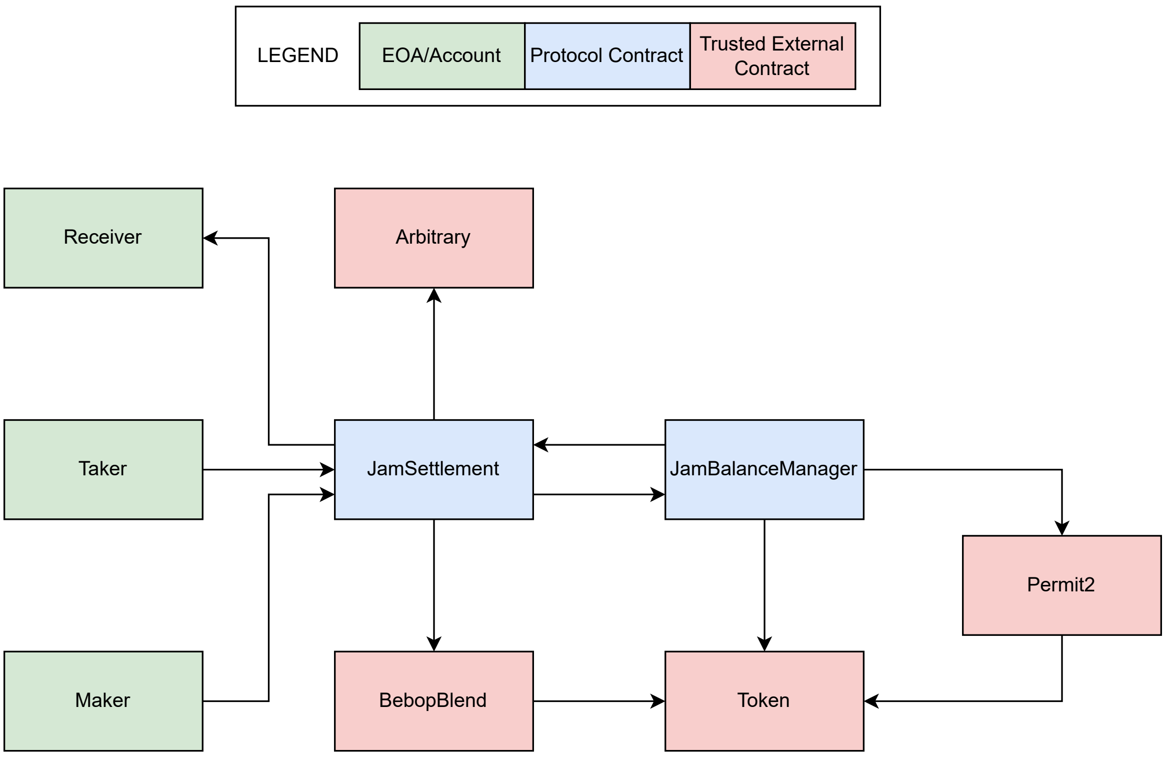
	Finding	Severity	Update
1	Explicitly specifying `partnerInfo` with zero fees causes transaction to revert	Info	Fixed
2	Fee distribution between protocol and partner can differ in `batchSettle`	Info	Acknowledged
3	Missing EIP-5267	Info	Fixed

4 Protocol Overview

Bebop JAM is a Just-in-Time aggregation model that allows independent solvers to compete for user orders to offer the best execution outcome. Once user accepts the solver's offer onchain, the solver is committed to deliver the execution providing accurate or better prices at superior execution cost.

On receiving order request from the user, Bebop JAM orchestrator will query solvers to provide the best solution for the order. User can accepts the solver's proposal by signing to confirm the trade. The orchestrator returns the accepted order to the winning solver for execution on-chain to complete the transaction.

The solvers can fulfill the order by interacting with external protocols or providing the liquidity directly. Bebop JAM offers an entry point to interact with BebopBlend contract using Permit2+witness functionality using single, multi and aggregate orders.



(c) protocol overview diagram

JamSettlement: The primary contract through which users interact to execute and settle orders. The contract supports four entrypoints, allowing for basic settlements, direct asset swap settlements, batch settlements and retroactive Permit2 support for BebopBlend. This contract inherits and/or utilizes the following:

- **JamValidation**: Handles validation of signatures and data passed for settlement.
- **JamPartner**: Handles partner info for fee distribution between partner and protocol for the order.
- **JamTransfer**: Moves ERC20 and native assets between accounts as part of settlement process.
- **JamInteractions**: Handles interactions with external contracts specified by the order maker to source desired tokens.

JamBalanceManager: All asset movement flows through this contract. Permit2 allowances specified by users are given to this address, as well as normal allowances. The exposed functions on this contract are only callable by the JamSettlement contract to ensure that funds cannot be accessed by any other contract.

BebopBlend: An exising Bebop token swap protocol, which has been integrated with JamSettlement to support Permit2 allowances.

5 Risk Rating Methodology

The risk rating methodology used by Nethermind 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 an attacker will uncover and exploit the finding. 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 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 formally pass to the client;

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] Explicitly specifying `partnerInfo` with no fees causes transaction to revert

File(s): `JamPartner.sol`

Description: In the function `settleInternal`, the field `partnerInfo` in the struct `order` is used to specify the fees that should be applied to the order. It is a `uint256` which contains the address of the partner which will recieve fees, as well as the protocol fee amount and partner fee amount in basis points. During an internal settlement, if the `partnerInfo` field is empty then no fees are applied, otherwise a separate logic branch for handling fees is entered, as shown below:

```
function settleInternal(...) external payable nonReentrant {
    // ...
    if (order.partnerInfo == 0){
        uint256[] calldata buyAmounts = validateFilledAmounts(filledAmounts, order.buyAmounts);
        balanceManager.transferTokens(order.buyTokens, buyAmounts, msg.sender, order.receiver);
        // ...
    } else {
        balanceManager.transferTokens(order.buyTokens, buyAmounts, msg.sender, order.receiver);
        if (protocolFees.length != 0){
            balanceManager.transferTokens(order.buyTokens, protocolFees, msg.sender, protocolFeeAddress);
        }
        if (partnerFees.length != 0){
            balanceManager.transferTokens(order.buyTokens, partnerFees, msg.sender, partner);
        }
    }
    // ...
}
```

However, it is possible to provide a `partnerInfo` field that specifies the partner address, but the fee amounts for the partner and protocol are still zero. Because the partner address is specified, `partnerInfo` field is nonzero so the fee branch is entered and the function `getUpdatedAmountsAndFees` is called. This function will create an array `newAmounts` which will contain the amounts that the user will recieve after fees. There are two `if` statements which only execute if the relevant fee type is nonzero, and they calculate the new user amount after these fees are subtracted, as shown below:

```
function getUpdatedAmountsAndFees(...) internal pure returns (...) {
    (address partnerAddress, uint16 partnerFee, uint16 protocolFee) = unpackPartnerInfo(partnerInfo);
    // ...
    newAmounts = new uint256[](tokensLength);
    if (protocolFee > 0) {
        protocolFees = new uint256[](tokensLength);
        for (uint256 i; i < tokensLength; ++i) {
            protocolFees[i] = amounts[i] * protocolFee / HUNDRED_PERCENT;
            newAmounts[i] = amounts[i] - protocolFees[i];
        }
    }
    if (partnerFee > 0) {
        partnerFees = new uint256[](tokensLength);
        for (uint256 i; i < tokensLength; ++i) {
            partnerFees[i] = amounts[i] * partnerFee / HUNDRED_PERCENT;
            newAmounts[i] = newAmounts[i] == 0 ? amounts[i] - partnerFees[i] : newAmounts[i] - partnerFees[i];
        }
    }
    for (uint256 i; i < tokensLength; ++i) {
        require(newAmounts[i] >= minAmounts[i], InvalidFilledAmounts(minAmounts[i], newAmounts[i]));
    }
    // ...
```

When both fees are zero, none of these `if` statements will execute, and the array `newAmounts` will never be written to, meaning all values in the array will be zero. At the end of the function a final validation is done to ensure that the `newAmounts` entries are not less than the `minAmounts`. In the case of both fees being zero however, each entry in `newAmounts` is zero, causing the transaction to revert on the `require(newAmounts[i] >= minAmounts[i])` comparison.

Recommendation(s): Consider updating the logic in the function `getUpdatedAmountsAndFees` to return the correct transferable amounts when both fees are zero.

Status: Fixed

Update from the client: Added verification to ensure that if partnerFee is zero, then partnerAddress is zero as well. Addressed in commit `c4ad578`.

6.2 [Info] Fee distribution between protocol and partner can differ in `batchSettle`

File(s): `JamTransfer.sol`

Description: In the function `batchSettle` after the interactions have executed, the new buy amounts for each order are calculated with fees being taken into account, before being transferred to the order recipient. The calculations are done in `calculateNewAmounts`, which includes a validation to ensure that that the total fees in the `partnerInfo` field are consistent between all orders in the batch. A snippet from the function is shown below:

```
function calculateNewAmounts(...) internal view returns (...) {
    JamOrder calldata curOrder = orders[curInd];
    // ...
    for (uint i; i < curOrder.buyTokens.length; ++i) {
        // ...
        for (uint j = curInd; j < orders.length; ++j) {
            for (uint k; k < orders[j].buyTokens.length; ++k) {
                if (orders[j].buyTokens[k] == curOrder.buyTokens[i]) {
                    // ...
                    // The total fees specified in `partnerInfo` must be same for all in batch
                    require(
                        getTotalFeesBps(curOrder.partnerInfo) == getTotalFeesBps(orders[j].partnerInfo),
                        DifferentFeesInBatch()
                    );
                }
            }
        }
        // ...
    }
    // ...
}
```

This check and the associated error message imply that the exact same fees must be applied between all orders in the batch. However, only the total fees for each order is compared. It is possible change the fee distribution between the protocol and partner, as long as the total amounts to the same consistent value across all orders. Also, the `partnerAddress` can be changed and this will not be detected either. Consider the following example:

```
OrderA{
    protocolFee: 10, partnerFee: 15, partnerAddress: 0xabcd
}
OrderB{
    protocolFee: 15, partnerFee: 10, partnerAddress: 0xabcd
}
OrderC{
    protocolFee: 5,  partnerFee: 25, partnerAddress: 0x1234
}
```

Recommendation(s): Consider adding a check to ensure that the fee distribution consistent between all orders.

Status: Acknowledged

Update from the client: Only the total fees are required to be equal, how the fees are distributed is not a concern.

6.3 [Info] Missing EIP-5267

File(s): `JamValidation.sol`

Description: The contract `JAMSettlement` allows solvers to execute orders on behalf of users by implementing the EIP-712 specification, where the user authorizes this by signing some structured data. Rather than importing EIP-712 from a stardard contract library such as Openzeppelin a custom implementation has been written instead. A difference between this custom implementation and the Openzeppelin implementation is the absence of EIP-5267. This EIP allows applications to retrieve and generate appropriate domain separators, helping to integrate EIP-712 signatures securely and scalably, and is commonly included alongside EIP-712.

Recommendation(s): Consider adding additional custom logic to support EIP-5267 or alternatively importing EIP-712 logic from the Openzeppelin contract library which already includes EIP-5267.

Status: Fixed

Update from the client: Added `eip712Domain` function to support ERC5267. Addressed in commit `ef58b97`.

7 Protocol Observations

This section aims to highlight observations made by the Nethermind team during the audit process that don't have any security implication and/or don't neccesarily need to be addressed, but should be noted nonetheless:

7.1 Nonces are shared between JamSettlement orders and Permit2

The `JamOrder` struct contains a field `nonce` which is used to ensure unique signatures across JamSettlement orders. This nonce that is used for the order is also the same nonce that is passed through to Permit2 for asset transfers. By having this nonce shared between JamSettlement and Permit2 some interesting and undocumented constraints are applied to order nonces.

On the JamSettlement contract there are two separate nonce invalidation storage maps, one for regular orders and one for limit orders. This effectively makes for three separate nonce management systems (regular order, limit order, Permit2) that must be satisfied when an order is processed. The off-chain logic that determines which nonce to use must take into account which nonces are unused in all three of these in order to generate a valid order.

Consider a regular order with a nonce of 15 and it requires a Permit2 asset transfer. This order is executed and the nonce of 15 is consumed both on the regular order nonce storage and the Permit2 nonce storage. Now the user would like to make a new order, this time being a limit order. The user intends to use nonce 15 which is unused on the limit order nonce storage (as it was only used on the regular order nonce storage). This would appear to be valid, but when execution reaches the Permit2 contract the nonce of 15 will have already been used and the transaction will revert.

Another case to consider is the user interacting with other protocols, and a given Permit2 nonce may have been used already. Even if an order is made where the nonce is unused both on the regular order and limit order nonce storage, it may still revert if it has been used on Permit2 by another protocol.

8 Documentation Evaluation

Software documentation refers to the written or visual information describing software's functionality, architecture, design, and implementation. 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 Bebop Jam documentation

The documentation for the Bebop Jam contracts is contained in the README of the project's Github.
It provides information on the following:

- General description of the protocol
- Core data structures (`Order` struct)
- Primary execution/interaction flows
- Integration with existing contracts (BebopBlend)

The information is presented in a very concise and technical manner, with well-written explanations and references where necessary. It also features a section explaining development dependencies and instructions on how to build and test the code.

Code comments are also of high quality, with explanations for some functions describing the purpose, context, and situations where the function will be called. Other than functions, some comments exist in other areas of the code where extra information is necessary, allowing readers to understand the codebase at a faster pace.

9 Test Suite Evaluation

9.1 Compilation Output

```
user@machine:~/NM-0372/report/bebop-jam-contracts$ forge build
[] Compiling...
[] Compiling 53 files with Solc 0.8.27
[] Solc 0.8.27 finished in 12.92s
Compiler run successful!
```

9.2 Tests Output

```
npm run test:jam

> bebop-jam-contracts@2.0.0 test:jam
> npx hardhat test ./test/hardhat/JamOrders.test.ts

Compiling 21 Solidity files
Generating typings for: 20 artifacts in dir: typechain-types for target: ethers-v5
Successfully generated 60 typings!
Successfully generated 38 typings for external artifacts!
Successfully compiled 21 Solidity files

  JamOrders
     JAM: settle - Simple with Permit2
     JAM: settle - Simple with standard approvals
     JAM: settle - Many-to-One with Permit2
     JAM: settle - Many-to-One with standard approvals
     JAM: settle - One-to-Many
     JAM: settle - Many-to-Many
     JAM: settle - One-to-Many with user excess
     JAM: settle - One-to-Many with user and solver excess
     JAM: settle - Without solver-contract, Permit2
     JAM: settle - Without solver-contract, standard approvals
     JAM: settle - SellNative, without solver contract
     JAM: settle - SellNative, taker=solver
     JAM: settle - BuyNative
     JAM: settle - BuyNativeAndWrapped
     JAM: settle - Protocol fee
     JAM: settle - Partner fee
     JAM: settle - Protocol fee & Partner fee
     JAM: settle - Protocol fee & Partner fee without solver contract
     JAM: settle - Protocol&Partner fee and userExcess
     JAM: settleInternal - Simple with Permit2
     JAM: settleInternal - Simple with standard approvals
     JAM: settleInternal - Many-to-One with Permit2
     JAM: settleInternal - One-to-Many with fees
     JAM: settleInternal - One-to-Many with increased amounts
     JAM: settleInternal - One-to-Many with increased amounts and fees
     JAM: settleBatch - Simple
     JAM: settleBatch - Simple with same user, same tokems
     JAM: settleBatch - Partner&Protocol fee
     JAM: settleBatch - One-to-Many with fees and excess
     JAM: beforeSettle hooks - ERC20-Permit
     JAM: beforeSettle hooks - DAI-Permit
     JAM: afterSettle hooks
     JAM: Invalid executor
     JAM: Limit order cancellation
     JAM: taker with smart wallet, standard approvals
     JAM: taker with smart wallet, permit2

  36 passing (14s)
```

```
npm run test:blend

> bebop-jam-contracts@2.0.0 test:blend
> npx hardhat test ./test/hardhat/BlendOrders.test.ts

Compiling 21 Solidity files
Generating typings for: 20 artifacts in dir: typechain-types for target: ethers-v5
Successfully generated 60 typings!
Successfully generated 38 typings for external artifacts!
Successfully compiled 21 Solidity files

  BlendOrders
     BebopBlend: SingleOrder
     BebopBlend: SingleOrder with sending better amounts to user
     BebopBlend: SingleOrder with keeping positive slippage
     BebopBlend: SingleOrder with worse amounts
     BebopBlend: SingleOrder with beforeSettle hooks
     BebopBlend: SingleOrder with afterSettle hooks
     BebopBlend: SingleOrder with another maker
     BebopBlend: MultiOrder - One-to-Many
     BebopBlend: MultiOrder - Many-to-One
     BebopBlend: MultiOrder - Many-to-One with sending better amounts to user
     BebopBlend: MultiOrder - Many-to-One with keeping positive slippage
     BebopBlend: MultiOrder - Many-to-One with worse amounts
     BebopBlend: AggregateOrder - One-to-One
     BebopBlend: AggregateOrder - One-to-One with partner
     BebopBlend: AggregateOrder - One-to-Many
     BebopBlend: AggregateOrder - One-to-Many with native token
     BebopBlend: AggregateOrder - Many-to-One
     BebopBlend: AggregateOrder - One-to-One with extra hop
     BebopBlend: AggregateOrder - One-to-One with 3 makers
     BebopBlend: AggregateOrder - Many-to-One 3 makers with hop
     BebopBlend: AggregateOrder - One-to-Many 3 makers with 2 hops
     BebopBlend: AggregateOrder - One-to-Many with sending better amounts to user
     BebopBlend: AggregateOrder - One-to-Many with keeping positive slippage
     BebopBlend: AggregateOrder - One-to-Many with worse amounts

  24 passing (13s)


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

Ran 3 tests for test/foundry/JamSolver.t.sol:JamSolverTest
[PASS] testOwnership() (gas: 19770)
[PASS] testWithdrawEth() (gas: 18850)
[PASS] testWithdrawTokens() (gas: 108144)
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 1.26ms (626.88µs CPU time)

Ran 3 tests for test/foundry/SignatureTests.t.sol:SignatureTests
[PASS] testEIP712InvalidSignature() (gas: 19281)
[PASS] testValidEIP1271Signature() (gas: 43837)
[PASS] testValidEIP712Signature() (gas: 17374)
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 2.76ms (3.60ms CPU time)

Ran 2 test suites in 6.93ms (4.03ms CPU time): 6 tests passed, 0 failed, 0 skipped (6 total tests)
Collapse
```

10 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development procehttps://www.overleaf.com/project/65c0e737f41a29601bda5c48ss, 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

Disclaimer