Security Review Report
NM-0259 - Starknet Nova

NETHERMIND SECURITY

(October 10, 2024)

Contents

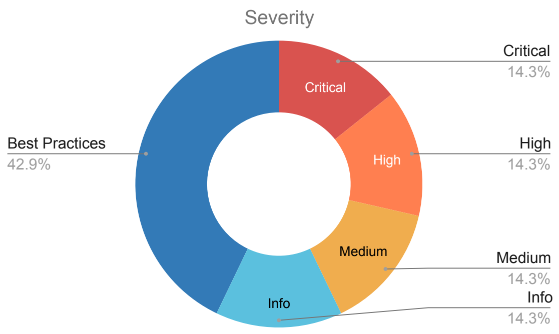
1 Executive Summary

This document outlines the security review conducted by Nethermind Security for Starknet Nova. Starknet Nova is a permissioned launchpad where users looking for investors can create token sales. Users who want to buy tokens must stake their supported ERC20 assets to gain a weighting that accumulates over time, which can be committed to a token sale. Committing these weights allows the users to gain an allocation in the token sale. Token sales run for a given time window, and then the assets can be released to the users either through a linear or cliff vesting approach. This review also included the SuperNovaToken, which is an ERC20 implementation that supports wrapping of the NovaToken.

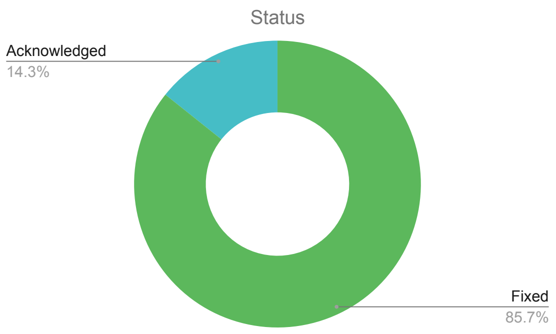
The audited code comprises of 1723 lines of code written in the Cairo language, and the audit was performed using (a) manual analysis of the codebase, (b) automated analysis tools, (c) simulation of the smart contract.

Along this document, we report seven points of attention, where one is classified as `Critical`, one is classified as `High`, one is classified as `Medium`, and four are classified as `Informational` or `Best Practices` . The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 presents the compilation, tests, and automated tests. Section 9 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 (1), **High** (1), **Medium** (1), **Low** (0), **Undetermined** (0), **Informational** (1), **Best Practices** (3). **(b) Distribution of status: Fixed** (6), **Acknowledged** (1), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	August 22, 2024
Final Report	October 10, 2024
Methods	Manual Review, Automated analysis
Repository	`Starknet-Nova/core-contracts`
Commit Hash	`b83e636758ff33ed65116e9fa2ac528db9bc88f6`
Final Commit Hash	`4af4a3cef1d442b1d9d178a532823eb421395271`
Documentation	Specs
Documentation Assessment	Medium
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/utils.cairo	2	0	0.0%	0	2
2	src/interfaces.cairo	8	0	0.0%	0	8
3	src/models.cairo	4	0	0.0%	0	4
4	src/core.cairo	5	0	0.0%	0	5
5	src/lib.cairo	4	0	0.0%	0	4
6	src/core/novawhitelister.cairo	48	11	22.9%	9	68
7	src/core/novatoken.cairo	25	9	36.0%	6	40
8	src/core/novasale.cairo	715	36	5.0%	50	801
9	src/core/novastaking.cairo	305	34	11.1%	23	362
10	src/core/supernovatoken.cairo	167	22	13.2%	22	211
11	src/utils/storeu64span.cairo	50	9	18.0%	10	69
12	src/utils/mockerc20.cairo	19	8	42.1%	5	32
13	src/models/sale.cairo	19	1	5.3%	1	21
14	src/models/usercheckpoint.cairo	38	1	2.6%	8	47
15	src/models/stakingtoken.cairo	4	1	25.0%	0	5
16	src/models/supernovauser.cairo	5	1	20.0%	0	6
17	src/interfaces/fundable.cairo	35	47	134.3%	5	87
18	src/interfaces/purchasable.cairo	112	170	151.8%	19	301
19	src/interfaces/versionable.cairo	3	1	33.3%	0	4
20	src/interfaces/novasale.cairo	23	49	213.0%	8	80
21	src/interfaces/stakeable.cairo	74	103	139.2%	18	195
22	src/interfaces/supernovatoken.cairo	23	75	326.1%	16	114
23	src/interfaces/vestable.cairo	20	48	240.0%	9	77
24	src/interfaces/whitelistable.cairo	15	17	113.3%	4	36
	Total	**1723**	**643**	**37.3%**	**213**	**2579**

3 Summary of Issues

	Finding	Severity	Update
1	Users can withdraw more than their purchased token amount	Critical	Fixed
2	The `supernovatoken` cannot be unwrapped by transfer recipients	High	Fixed
3	`if_user_uncommitted_after_purchase_start_in_sale_id` is not related to specific tokens	Medium	Fixed
4	Sale stage is not well defined at start and end timestamp	Info	Fixed
5	Basis points field `bips` in `supernovatoken` should be fixed	Best Practices	Fixed
6	Return values from `ERC20` transfer functions are not checked	Best Practices	Fixed
7	The `_assert_before_commit_start(...)` has multiple responsibilities	Best Practices	Acknowledged

4 System Overview

The audited core contracts in the Starknet Nova protocol implementation are shown below, each contract is described as follows:

4.1 `novasale.cairo`

Contains functions for managing sales. This contract allows the creation and configuration of new sales. Users looking to buy tokens will also interact with this contract through the `purchase(...)` and `purchase_whitelist(...)` functions. Whitelisted sales will use a `novawhitelister` contract to verify users that can participate in purchasing tokens; non-whitelisted sales will use the committed weights by users to define the token allocation associated with each user.

After a sale has finished, the sale's owner can use the `cashout(...)` function to claim the payment tokens and any surplus of sale tokens that were not bought by the users. Users can also claim the tokens they bought after the sale has ended. However, they depend on the sale configuration to decide how to claim the tokens. It is possible for the sales owner to set two different ways of claiming the tokens: one is through linear vesting, and the other is by a sequence of releases of different amounts of tokens.

4.2 `novastaking.cairo`

This contract manages all the functionalities related to staking. Staking is a crucial part of the NovaSale protocol. Users are required to stake tokens in order to gain weights that they can use to receive allocations in the different token sales that exist. The contract's owner can define which tokens can be staked at any moment and how much weight each token will generate per second.

Users can stake their tokens and will earn weights for each second they are staked. These weights can be committed to a sale during the commitment period to receive an allocation of tokens to buy. Users can also uncommit their committed weights at any time; however, they need to have them committed when the commitment period of the sale ends in order to get the token allocation. Because of how allocations are computed, users can commit the same weights in multiple sales and receive allocation if those sales do not finish their commitment phase simultaneously. It is important to note that weights are linked to the tokens that produced them.

The total weights a user has for a specific token are evenly distributed between the amount of that token the user has staked; this can produce different situations where staking and unstaking a certain amount of tokens can vary the amount of weights a user has in unexpected ways. To unstake a certain amount of tokens, the users must have a proportional amount of uncommitted weights associated with that token.

4.3 `novawhitelister.cairo`

This contract holds a merkle tree where each leaf is the result of hashing a `ContractAddress` and a `u256`, representing a whitelisted user's address and the amount of tokens they are eligible to purchase respectively. The merkle root is set by the owner, and a `verify(...)` function is exposed which is used by the NovaSale contract for verifying whitelisted purchases.

4.4 `novatoken.cairo`

The Nova Launchpad token, a standard ERC20 contract importing from the OpenZeppelin implementation.

4.5 `supernovatoken.cairo`

An ERC20 contract imported from the OpenZeppelin implementation with an additional feature allowing the Nova Launchpad token to be wrapped. Tokens must remain wrapped for some set amount of time, otherwise a penalty is applied.

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 [Critical] Users can withdraw more than their purchased token amount

File(s): src/core/novasale.cairo

Description: The function withdraw_claimable_tokens is called by users to transfer assets once a cliff time has been reached or when a reasonable amount of time has passed for linear vesting. The number of claimable tokens is calculated first, and then vesting_claimed_sale_token_amount_for_user is updated. This storage variable should track the cumulative amount of tokens claimed, but it only tracks the amount of the most recent claim.

```
fn withdraw_claimable_tokens(ref self: ContractState, sale_id: u64) {
    // ...
    let withdraw_amount = self.get_unclaimed_tokens(user, sale_id);
    self.vesting_claimed_sale_token_amount_for_user.write((user, sale_id), withdraw_amount);
    self.sale_token.read(sale_id).transfer(user, withdraw_amount);
    // ...
}
```

The function get_unclaimed_tokens then uses this storage variable with an incorrect assumption that it tracks the cumulative claim, and allows for more assets to be claimed than expected. Consider the following scenario for cliff vesting:

```
formula = total_token * released_bips / total_bips - vesting_claimed

total_tokens = 1000
cliff1 = 2500 bips
cliff2 = 2500 bips
cliff3 = 2500 bips
cliff4 = 2500 bips

1st claim
vesting_claimed = 0
released_bips = 2500
1000 * 2500 / 10000 - 0 = 250
250 claimed, set vesting_claimed to 250

2nd claim
vesting_claimed = 250
released_bips = 5000
1000 * 5000 / 10000 - 250 = 250
250 claimed, set vesting_claimed to 250 (should be 500, cumulative)

3rd claim
vesting_claimed = 250
released_bips = 7500
1000 / 7500 / 10000 - 250 = 500
500 claimed, set vesting_claimed to 500 (should be 750, cumulative)

4th claim
vesting_claimed = 500
released_bips = 10000
1000 * 10000 / 10000 - 500 = 500
500 claimed, set vesting_claimed to 500 (should be 1000, cumulative)
```

A user could call this function repeatedly after the fourth claim and the contract logic will determine every time that they should get 500 tokens. This can be used to drain the token balance from the contract.

Recommendation(s): Consider changing the function withdraw_claimable_tokens to add onto the existing value for vesting_claimed_sale_token_amount_for_user rather than directly overwriting.

Status: Fixed.

Update from the client: Resolved with 5a5dc1d5e06013498f35e7660dfc3b68baa53e02.

6.2 [High] The supernovatoken cannot be unwrapped by transfer recipients

File(s): src/core/supernovatoken.cairo

Description: When a user first wraps some amount of novatoken into supernovatoken, the storage mapping users will be updated. Particularly, the wrapped_amount field will be incremented to reflect the new amount of wrapped tokens, as shown below:

```
fn wrap(ref self: ContractState, amount: u256) {
    //...
    user.wrapped_amount += amount;
    self.users.write(user_address, user);
    self.erc20._mint(user_address, amount);
}
```

When unwrapping assets, this wrapped_amount field is decreased to reflect the change after some number of assets have been unwrapped. This operation will work correctly only when the caller has wrapped an equivalent number of tokens themselves. If a user did not ever wrap tokens themselves, but received them via transfer or transferFrom instead, then their entry in the users mapping will be empty. When subtracting from the wrapped_amount field it will result in an underflow and cause the call to revert.

```
fn unwrap(ref self: ContractState, amount: u256) {
    // ...
    let user_address = get_caller_address();
    let mut user = self.users.read(user_address);
    // @audit If user received tokens via transfer this will be zero
    //        Attempting to subtract will result in the call reverting
    user.wrapped_amount -= amount;
    // ...
}
```

Recommendation(s): Consider using the user's token balance to track the user wrapped amount instead of tracking it separately in storage with user.wrapped_amount, as they are both updated in the same functions with the same values.

Status: Fixed.

Update from the client: Resolved with 3ef6ee4db90de2ab933afb397d0f6b56b3b6f17d.

6.3 [Medium] if_user_uncommitted_after_purchase_start_in_sale_id is not related to specific tokens

File(s): src/core/novastaking.cairo

Description: Users can uncommit their voting power through the uncommit(...) function. This function contains a check to ensure that users do not uncommit power related to the same more than once.

```
fn uncommit(ref self: ContractState, sale_id: u64, token: ContractAddress) {
    ...
    let past_commit_period = !IPurchasableDispatcher {
        contract_address: self.nova_sale_contract.read()
    }
        .is_currently_committable(sale_id);
    assert(
        !(past_commit_period
            && self
                .if_user_uncommitted_after_purchase_start_in_sale_id
                .read((get_caller_address(), sale_id))),
        StakeableError::ALREADY_UNCOMMITTED
    );
    ...
}
```

For this, the if_user_uncommitted_after_purchase_start_in_sale_id mapping is used. However, these mapping entries are linked to a specific user and sales ID; they are not related to specific tokens. In case a user has committed voting power from multiple tokens until the end of the committable period, after they uncommit the power related to one token, they won't be able to uncommit the power related to the rest of the tokens.

Recommendation(s): Consider adding one more key to this mapping in order to have entries per the tuple (user, sale_id, token).

Status: Fixed.

Update from the client: Overhauled staking and committing logic in commits f1f9ebf58b50fcc2cf9186097cf5dba478518c9e and 9c24ab9ef4a089a87859f2bdece2a79c27e35fa7.

6.4 [Info] Sale stage is not well defined at start and end timestamp

File(s): src/core/novasale.cairo

Description: The _assert_before_sale_start(...), _assert_during_sale(...), and _assert_after_sale_end(...) functions are used to verify that a sale is at a specific stage.

```
fn _assert_before_sale_start(self: @ContractState, sale_id: u64) {
    let (start, _) = self.sale_start_end_timestamp.read(sale_id);
    assert(start > get_block_timestamp(), FundableError::NOT_BEFORE_SALE_START);
}

fn _assert_during_sale(self: @ContractState, sale_id: u64) {
    let (start, end) = self.sale_start_end_timestamp.read(sale_id);
    let current_timestamp = get_block_timestamp();
    assert(
        start < current_timestamp && end > current_timestamp, FundableError::NOT_DURING_SALE
     );
}

fn _assert_after_sale_end(self: @ContractState, sale_id: u64) {
    let (_, end) = self.sale_start_end_timestamp.read(sale_id);
    assert(end < get_block_timestamp(), FundableError::SALE_IN_PROGRESS);
}
```

The _assert_during_sale(...) checks that the current time is strictly lower than the end time and strictly bigger than the start time. The _assert_after_sale_end(...) checks that the current time is strictly greater than the end time. The _assert_before_sale_start(...) function checks that the current time is strictly lower than the start time. When the current timestamp is equal to the start or end timestamp, all these functions will fail

Recommendation(s): Consider defining the stage of a sale at the start and end timestamps and modifying the functions accordingly.

Status: Fixed.

Update from the client: Resolved with 57fba1b40448c553dfc923f2c3a340bf8a96772e.

6.5 [Best Practices] Basis points field bips in supernovatoken should be fixed

File(s): src/core/supernovatoken.cairo

Description: The contract supernovatoken has a penalty mechanism which is applied to users that unwrap their tokens before the specified wait time is over. The penalty fee is measured in basis points (bips), a financial term where each bip is one hundredth of one percentage point (1/10000). The supernovatoken contract does not follow the proper definition of basis points, as the denominator should always be 10000, but instead there is a function set_bips that allows it to be set to an arbitrary value.

```
// @audit The term "bips" implies it should always be 10000
//        No need for this to be set, can be a constant
fn set_bips(ref self: ContractState, bips: u256) {
    self.ownable.assert_only_owner();
    self.bips.write(bips);
}

// @audit This could be simplified to `amount * penalty / 10000`
fn get_penalty_amount(self: @ContractState, amount: u256) -> u256 {
    amount * self.force_claim_penalty_bips.read() / self.bips.read()
}
```

Recommendation(s): If the term "bips" is to be followed correctly, consider setting the bips as a fixed value of 10000. This will also remove a storage read when applying penalty amounts. If the denominator needs to change to values other than 10000, consider using a term other than "bips" to make it clear to any developers reading the codebase.

Status: Fixed.

Update from the client: Resolved in 5f9a8f6651309c896848aed6250aea0df124e016.

6.6 [Best Practices] Return values from `ERC20` transfer functions are not checked

File(s): `src/*`

Description: When using `transfer(...)` and `transfer_from(...)` functions from ERC20 contracts the returned value is not checked. These functions are expected to return a the value `true` when they are successful and revert when that are not. However, there are widely use tokens in networks like Ethereum mainnet that do not follow this flow and return `false` instead of reverting when a transfer cannot occur. It is possible that this behavior will be replicated in Starknet by some tokens, in this case contracts interacting with this kind of tokens could work in unintended ways.

Recommendation(s): Consider checking that the `transfer(...)` and `transfer_from(...)` functions return `true` indicating a successful execution.

Status: Fixed.

Update from the client: Resolved with 802dec21991cf43cbbba2bf881ac6c3b6f4837fa.

6.7 [Best Practices] The `_assert_before_commit_start(...)` has multiple responsibilities

File(s): `src/core/novasale.cairo`

Description: The function `_assert_before_commit_start(...)` is used to check if at least one of the following conditions is true:

- Commit time for the sale has not started;
- Commit time for the sale has not been set;

These conditions are expected to check for different flows; however, the same function is used in all the flows.

Recommendation(s): It is a good practice to keep a clear and unique responsibility to each function to avoid bugs in future development.

Status: Acknowledged.

Update from the client: Acknowledged, but skipping the fix.

7 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 StarknetNova Protocol documentation

The **Starknet Nova** provided a specification for some of the main functionalities of the contracts; besides this, inline comments within the codebase are informative and describe expected functionality. Additionally, the **Starknet Nova** team was available to address any questions or concerns from the Nethermind Security team.

8 Test Suite Evaluation

8.1 Compilation Output

```
> scarb build -v
     Running git clone --local --verbose --config core.autocrlf=false --recurse-submodules
        .cache/scarb/registry/git/db/alexandria-oen34onnk9gr0.git
        ↪  .cache/scarb/registry/git/checkouts/alexandria-oen34onnk9gr0/bbc010b
     Running git reset --hard bbc010b922d9ca6f807ef01b0b255e34331b5eac
     Running git clone --local --verbose --config core.autocrlf=false --recurse-submodules
        .cache/scarb/registry/git/db/cairo-contracts-9cboa8jg3jldq.git
        ↪  .cache/scarb/registry/git/checkouts/cairo-contracts-9cboa8jg3jldq/978b4e7
     Running git reset --hard 978b4e75209da355667d8954d2450e32bd71fe49
     Running git clone --local --verbose --config core.autocrlf=false --recurse-submodules
        .cache/scarb/registry/git/db/starknet-foundry-v6206fshfutuo.git
        ↪  .cache/scarb/registry/git/checkouts/starknet-foundry-v6206fshfutuo/95e9fb09
     Running git reset --hard 95e9fb09cb91b3c05295915179ee1b55bf923653
   Compiling nova v0.0.1 (NM-0259/NM-0259-Security-Review-StarknetNova/Scarb.toml)
    Finished release target(s) in 17 seconds
```

8.2 Tests Output

8.2.1 Staker and ERC20 tests

```
> snforge test
   Compiling nova v0.0.1 (.../core-contracts/Scarb.toml)
    Finished release target(s) in 13 seconds


Collected 16 test(s) from nova package
Running 0 test(s) from src/
Running 16 test(s) from tests/
[PASS] tests::supernovatoken::pause (gas: ~1059)
[PASS] tests::novasale::new_sale (gas: ~302)
[PASS] tests::novastaking::stake_hardcoded_0 (gas: ~1041)
[PASS] tests::novastaking::stake_fuzzed (runs: 256, gas: {max: ~1297, min: ~1041, mean: ~1204.00, std deviation:
 ↪  ~95.95})
[PASS] tests::supernovatoken::wrap (runs: 256, gas: {max: ~1654, min: ~1, mean: ~1522.00, std deviation: ~95.61})
[PASS] tests::supernovatoken::force_unwrap (runs: 256, gas: {max: ~1314, min: ~1, mean: ~672.00, std deviation:
 ↪  ~655.65})
[PASS] tests::novasale::vesting_linear_and_withdraw (runs: 256, gas: {max: ~5632, min: ~1, mean: ~411.00, std
 ↪  deviation: ~1322.90})
[PASS] tests::novastaking::unstake_no_commit_fuzzed (runs: 256, gas: {max: ~939, min: ~1, mean: ~934.00, std deviation:
 ↪  ~58.76})
[PASS] tests::novasale::funding (runs: 256, gas: {max: ~1541, min: ~1, mean: ~1408.00, std deviation: ~91.68})
[PASS] tests::supernovatoken::unwrap (runs: 256, gas: {max: ~1365, min: ~1, mean: ~1229.00, std deviation: ~109.33})
[PASS] tests::novastaking::setters (runs: 256, gas: {max: ~449, min: ~1, mean: ~238.00, std deviation: ~223.36})
[PASS] tests::novasale::commit_purchase_cashout (runs: 256, gas: {max: ~4648, min: ~1, mean: ~766.00, std deviation:
 ↪  ~1704.30})
[PASS] tests::novatoken::deployment (runs: 256, gas: {max: ~620, min: ~364, mean: ~619.00, std deviation: ~15.97})
[PASS] tests::novasale::sale_setters (runs: 256, gas: {max: ~1251, min: ~1, mean: ~49.00, std deviation: ~242.18})
[PASS] tests::supernovatoken::setters (runs: 256, gas: {max: ~1441, min: ~1185, mean: ~1438.00, std deviation: ~19.89})
[PASS] tests::novasale::vesting_cliff_and_withdraw (runs: 256, gas: {max: ~4302, min: ~1, mean: ~505.00, std deviation:
 ↪  ~1383.39})
Tests: 16 passed, 0 failed, 0 skipped, 0 ignored, 0 filtered out
Fuzzer seed: 1104218690715758810
```

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 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

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