Security Review Report
NM-0502 iBTC-Cairo

NETHERMIND
SECURITY

(June 2, 2025)

Contents

1 Executive Summary

This document presents the security review performed by Nethermind Security for iBTC Network's cairo contract. iBTC offers a bridgeless wrapped Bitcoin for decentralized finance by securing BTC directly in the Bitcoin Layer 1 using a 2-of-2 multisig vault system. iBTC uses Discrete Log Contracts (DLCs) on the Bitcoin network that enable conditional move and lock of Bitcoin.

Using Discrete Log Contracts (DLCs), the iBTC Network mints iBTC token that represents Bitcoin locked in the DLCs and burns the iBTC token when the DLC is settled.

The audit focused on the iBTC Cairo contracts, consisting of 1,122 lines of Cairo code, which enable minting and burning of the iBTC token on the Starknet blockchain once Bitcoin deposits are confirmed on Bitcoin Layer 1. **The audit was performed using** (a) manual analysis of the codebase, (b) automated analysis tools, and (c) creation of test cases. **Along this document, we report seven points of attention**, where one is classified as `High`, and six are classified as `Informational` and `Best Practices` as shown 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.

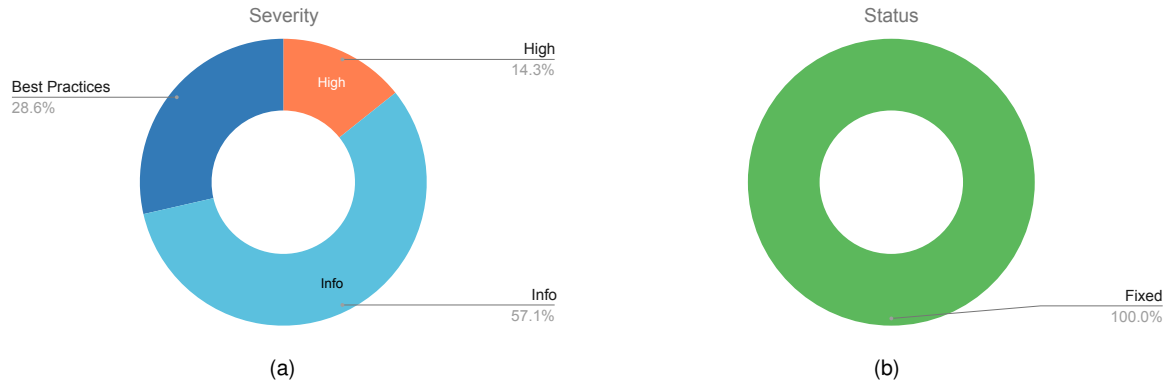


Fig. 1: Distribution of issues: Critical (0), **High** (1), **Medium** (0), **Low** (0), **Undetermined** (0), **Informational** (4), **Best Practices** (2). **Distribution of status: Fixed** (7), **Acknowledged** (0), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Final Report	June 2, 2025
Repository	ibtc-cairo
Commit	f50901f
Final Commit	3434907
Documentation	iBTC documentation
Documentation Assessment	Medium
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/utils.cairo	76	6	7%	11	93
2	src/ibtc_token.cairo	105	31	29%	24	160
3	src/interface.cairo	107	25	23%	20	152
4	src/ibtc_manager.cairo	652	86	13%	111	849
5	src/ibtc_library.cairo	63	3	4%	4	70
6	src/lib.cairo	9	0	0%	1	10
7	src/timelock_controller.cairo	31	13	41%	5	49
8	src/event.cairo	79	21	26%	21	121
	Total	**1122**	**185**	**16.5%**	**197**	**1504**

3 Summary of Issues

	Finding	Severity	Update
1	Potential for double minting via multiple signature sets	High	Fixed
2	The `ATTESTOR_MULTISIG_STRUCT_TYPE_HASH` includes the `u256` struct	Info	Fixed
3	The `ibtc_manager` does not have a function to upgrade `ibtc_token` contract	Info	Fixed
4	The `set_approved_signers(...)` override `signer_count` rather than increasing it	Info	Fixed
5	The `set_status_funded(...)` can be executed for a different transaction ID than that of `set_status_pending(...)`	Info	Fixed
6	The `_check_por(...)` does not need to return bool	Best Practices	Fixed
7	The `get_ssf_message(...)` and `get_ssp_message(...)` takes mutable reference of the contract state	Best Practices	Fixed

4 System Overview

The iBTC cairo contracts provide a mechanism for merchants to receive iBTC on Starknet in a 1:1 ratio once their BTC deposit has been attested and confirmed by Attestors using Flexible Round-Optimized Schnorr Threshold Signatures (FROST) on Bitcoin Layer 1. Merchants are entities such as trading firms, asset managers, and market makers who provide liquidity.

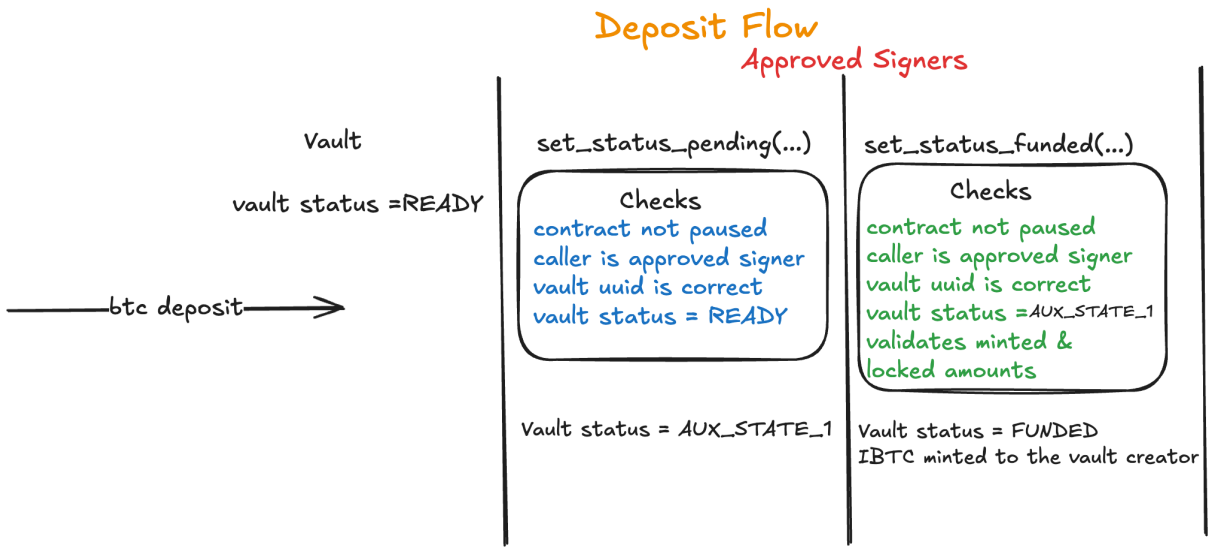
4.1 Vault Creation

Merchants set up an iBTCVault on Starknet that contains the necessary information to mint and burn iBTC. Every vault is identified by a unique identifier `uuid` and contains information such as the amount of BTC locked and iBTC minted, plus additional information including the status of the vault, the creator of the vault, and fee-related information.

```
    pub struct IBTCVault {
        uuid: u256,
        protocol_contract: ContractAddress,
        timestamp: u64,
        value_locked: u256,
        creator: ContractAddress,
        status: u8,
        funding_tx_id: u256,
        closing_tx_id: u256,
        btc_fee_recipient: ByteArray,
        btc_mint_fee_basis_points: u64,
        btc_redeem_fee_basis_points: u64,
        taproot_pubkey: ByteArray,
        value_minted: u256,
        wd_tx_id: u256,
}
```

4.2 Deposit Flow

Once BTC deposit on Bitcoin Layer 1 has been attested and confirmed, **Approved signers** will call `set_status_pending (...)`, which will perform vault validation checks and update the status of the vault to `AUC_STATE_1`. After six block confirmations, the **Approved signers** will call the `set_status_funded (...)` function, which will mint iBTC to the `iBTC Vault Creator` after performing further vault validations.



(c)

4.3 Withdraw Flow

User can redeem their BTC through merchants with vault in Funded Status who can then execute the `withdraw (...)` function on the `iBTCManager` contract, which will effectively burn the iBTC token on Starknet. Attestors will then confirm and attest to the burn action. **Approved signers** will then call `set_status_pending (...)`, wait for 6 block confirmation and then call `set_status_funded (...)` on the `iBTCManager` contract. The locked BTC will eventually be released, and the redemption will be completed in 30-60 minutes.

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 [High] Potential for double minting via multiple signature sets

File(s): ibtc_manager.cairo

Description: When a user deposits on the Bitcoin chain, an approved signer needs to call `set_status_pending(...)`, and after some block confirmations, they'll need to call `set_status_funded(...)`. Both of these functions take signatures of attestors as arguments. These signatures are validated using `_attestor_multisig_is_valid(...)`. It ensures that the number of signatures is greater than or equal to `threshold` and signatures from the same attestors are not included more than once.

```
fn _attestor_multisig_is_valid(
    ref self: ContractState,
    uuid: u256,
    btc_tx_id: u256,
    tx_type: felt252,
    amount: u256,
    signatures: Span<(ContractAddress, Array<felt252>)>
) {
    let threshold = self.threshold.read();
    assert(signatures.len() >= threshold.into(), Errors::NOT_ENOUGH_SIGNATURES);
    for (attestor, signature) in signatures {
        // ...
        let is_valid_signature = IAccountDispatcher { contract_address: *attestor }
            .is_valid_signature(message, signature.clone());
        assert(is_valid_signature == 'VALID', Errors::INVALID_SIGNATURE);
        assert(self.accesscontrol.has_role(APPROVED_SIGNER, *attestor), Errors::INVALID_SIGNER);
        self._check_signer_unique(*attestor, message);
    }
}

fn _check_signer_unique(
    ref self: ContractState,
    attestor_pub_key: ContractAddress,
    message_hash: felt252,
) {
    assert(!self.seen_signers.read((attestor_pub_key, message_hash)), Errors::DUPLICATE_SIGNATURE);
    self.seen_signers.write((attestor_pub_key, message_hash), true);
}
```

However, if there are more attestors than `threshold`, then a malicious approved signer can execute `set_status_pending(...)` and `set_status_funded(...)` more than once by passing different sets of signatures that are at least equal to the threshold.

For example, if `threshold` is `3` and there are `6` attestors, then a malicious approved signer can execute `set_status_pending(...)` and `set_status_funded(...)` via signatures from exactly `3` attestors and then execute both of these again after withdrawing via signatures from `3` different attestors.

Recommendation(s): Consider adding more checks to ensure that a user's deposit can't be processed more than once. This can be done by adding more validation to the Bitcoin transaction ID.

Status: Fixed

Update from the client: Fixed in 524727a

6.2 [Info] The ATTESTOR_MULTISIG_STRUCT_TYPE_HASH includes the u256 struct

File(s): utils.cairo

Description: The hash of struct like StarknetDomain, AttestorMultisigTx and u256 is calculated using `selector` macro.

```
struct StarknetDomain {
    name: felt252,
    version: felt252,
    chain_id: felt252,
    revision: felt252,
}
const STARKNET_DOMAIN_TYPE_HASH: felt252 =
    selector!(
        "\"StarknetDomain\"(\"name\":\"shortstring\",\"version\":\"shortstring\",\"chainId\":\"shortstring\",
        \"revision\":\"shortstring\")"
    );

pub struct AttestorMultisigTx {
    uuid: u256,
    btc_tx_id: u256,
    tx_type: felt252,
    amount: u256,
}
// @audit it also contains fields of u256
const ATTESTOR_MULTISIG_STRUCT_TYPE_HASH: felt252 =
    selector!("\"AttestorMultisigTx\"(\"uuid\":\"u256\",\"btc_tx_id\":\"u256\",\"tx_type\":\"felt\",\"amount\":
    \"u256\")\"u256\"(\"low\":\"u128\",\"high\":\"u128\")");


const U256_TYPE_HASH: felt252 = selector!("\"u256\"(\"low\":\"u128\",\"high\":\"u128\")");
```

However, the `ATTESTOR_MULTISIG_STRUCT_TYPE_HASH` incorrectly contains a `u256` struct at the end.

Recommendation(s): Consider removing the `u256` struct in the hash of the `AttestorMultisigTx` struct.

Status: Fixed

Update from the client: Fixed in 4d8656e

6.3 [Info] The ibtc_manager does not have a function to upgrade ibtc_token contract

File(s): ibtc_token.cairo

Description: The `ibtc_token` contract has an `upgrade(...)` function that can only be executed by the `owner`. The owner for the `ibtc_token` contract is the `ibtc_manager`. However, there's no function in the `ibtc_manager` contract to execute `upgrade(...)` in the `ibtc_token` contract.

Although `transfer_token_contract_ownership(...)` can be used to transfer ownership, the new owner can then execute `upgrade(...)` in the `ibtc_token` contract.

Recommendation(s): Consider adding a function in the `ibtc_manager` contract to execute `upgrade(...)` in the `ibtc_token` contract.

Status: Fixed

Update from the client: Fixed in 850353e

6.4 [Info] The `set_approved_signers(...)` override `signer_count` rather than increasing it

File(s): ibtc_manager.cairo

Description: When an account is granted `APPROVED_SIGNER` role via `grant_role(...)`, the `signer_count` is increase by `1`.

```
fn grant_role(ref self: ContractState, role: felt252, account: ContractAddress) {
    if (self._has_any_roles(account)) {
        panic_with_felt252(Errors::INCOMPATIBLE_ROLE);
    }
    self.accesscontrol.grant_role(role, account);
    if (role == APPROVED_SIGNER) {
        self.signer_count.write(self.signer_count.read() + 1);
    }
}
```

However, an account can be granted the `APPROVED_SIGNER` role using `set_approved_signers(...)`, too. However, this doesn't increase the `signer_count` but rather overrides it.

```
fn set_approved_signers(ref self: ContractState, signers: Array<ContractAddress>) {
    self.accesscontrol.assert_only_role(IBTC_ADMIN_ROLE);

    let signers_len = signers.len();
    let mut i: usize = 0;
    loop {
        if i >= signers_len {
            break;
        }
        let signer = signers.at(i);
        self.accesscontrol._grant_role(APPROVED_SIGNER, *signer);
        i += 1;
    };
    // @audit signer_count is overriden
    self.signer_count.write(signers_len.try_into().unwrap());
    self.emit(SetApprovedSigners{signers});
}
```

If `set_approved_signers(...)` is ever used after executing `grant_role(...)` to grant an account the `APPROVED_SIGNER` role, the `signer_count` would have incorrect accounting.

Recommendation(s): Consider increasing `signer_count` rather than overriding it in `set_approved_signers(...)`.

Status: Fixed

Update from the client: Fixed in a73f925

Update from the Nethermind Security: The `set_approved_signers(...)` can only be executed by `IBTC_ADMIN_ROLE`. Previously, it was directly using `_grant_role(...)` which doesn't have any access control so the caller of `set_approved_signers(...)` needs to have `IBTC_ADMIN_ROLE` only. However, now, it further invokes `grant_role(...)` which can only be executed by admin of that role i.e. admin of `APPROVED_SIGNER` role i.e. `DEFAULT_ADMIN_ROLE`. So the caller not only needs to have `IBTC_ADMIN_ROLE` but also `DEFAULT_ADMIN_ROLE`.

Update from the client: Fixed in 64a72a1

6.5 [Info] The `set_status_funded(...)` can be executed for a different transaction ID than that of `set_status_pending(...)`

File(s): ibtc_manager.cairo

Description: When a user deposits on the Bitcoin chain, an approved signer needs to call `set_status_pending(...)`, and after some block confirmations, they'll need to call `set_status_funded(...)`. Both of these functions take a transaction ID as an argument. The transaction ID is expected to be the same in both cases. However, the code doesn't ensure that the transaction ID passed as an argument in `set_status_funded(...)` is the same as the one passed in `set_status_pending(...)`.

Recommendation(s): Consider adding a check to ensure that the transaction ID passed in `set_status_funded(...)` is the same as the one passed in `set_status_pending(...)`.

Status: Fixed

Update from the client: Fixed in 524727a

6.6 [Best practices] The `_check_por(...)` does not need to return bool

File(s): ibtc_manager.cairo

Description: The `_check_por(...)` is not completely implemented yet, but it's expected to either always return true or revert. If it returns false, then the `iBTC` won't be minted to the user for their deposits, and their deposit on the Bitcoin chain would be stuck. So, it should revert rather than return false, as also done in the Solidity implementation. As the function will never return false, there's no need to return a boolean.

Recommendation(s): Consider updating the function signature for `_check_por(...)`.

Status: Fixed

Update from the client: Fixed in a53ca2d

Update from the Nethermind Security: The `_check_mint(...)` is also modified and now it doesn't return the bool which completely defeats its purpose. It was previously used in if check to ensure that tokens are only minted if it returns true, whereas now as it doesn't return a bool it's executed and not used in if check and tokens are minted irrespectively. Thus, now executing `_check_mint(...)` does nothing.

Update from the client: Fixed in e7e7e3f

6.7 [Best practices] The `get_ssf_message(...)` and `get_ssp_message(...)` takes mutable reference of the contract state

File(s): ibtc_manager.cairo

Description: The `get_ssf_message(...)` and `get_ssp_message(...)` functions return the hash of the message that can be used by attestors to sign the message. It doesn't update the contract state but just returns the calculated hash.

```
#[external(v0)]
fn get_ssf_message(ref self: ContractState, attestor: ContractAddress, uuid: u256, btc_tx_id: u256, new_value_locked:
↪  u256) -> felt252 {
    let message = AttestorMultisigTx {
        uuid,
        btc_tx_id,
        tx_type: 'set-status-funded',
        amount: new_value_locked,
    };

    message.get_message_hash(attestor)
}

#[external(v0)]
fn get_ssp_message(ref self: ContractState, attestor: ContractAddress, uuid: u256, wdtx_id: u256, new_value_locked:
↪  u256) -> felt252 {
    let message = AttestorMultisigTx {
        uuid,
        btc_tx_id: wdtx_id,
        tx_type: 'set-status-pending',
        amount: new_value_locked,
    };

    let message_hash = message.get_message_hash(attestor);
    message_hash
}
```

However, both functions take a mutable reference of `self`, i.e, the contract state.

Recommendation(s): Consider taking a snapshot of the contract state (`self: @ContractState`) rather than a mutable reference (`ref self: ContractState`).

Status: Fixed

Update from the client: Fixed in ba418ce

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 the iBTC contract documentation

The `iBTC` team has provided a comprehensive walkthrough of the project in the kick-off call. The documentation includes the explanation of the intended functionalities. Moreover, the team addressed the questions and concerns raised by the Nethermind Security team, providing valuable insights and a comprehensive understanding of the project's technical aspects.

8 Complementary Checks

8.1 Compilation Output

```
scarb build
   Compiling snforge_scarb_plugin v0.34.0
    Finished `release` profile [optimized] target(s) in 0.29s
   Compiling lib(ibtc_cairo) ibtc_cairo v0.1.0 (/.../ibtc-cairo/contracts/Scarb.toml)
   Compiling starknet-contract(ibtc_cairo) ibtc_cairo v0.1.0 (/.../ibtc-cairo/contracts/Scarb.toml)
    Finished `dev` profile target(s) in 2 minutes
```

8.2 Tests Output

```
snforge test
[WARNING] Package snforge_std version does not meet the recommended version requirement =0.37.0, it might result in
↪  unexpected behaviour
   Compiling snforge_scarb_plugin v0.34.0
    Finished `release` profile [optimized] target(s) in 0.35s
   Compiling test(ibtc_cairo_unittest) ibtc_cairo v0.1.0 (/.../ibtc-cairo/contracts/Scarb.toml)
   Compiling test(ibtc_cairo_integrationtest) ibtc_cairo_integrationtest v0.1.0 (/.../ibtc-cairo/contracts/Scarb.toml)
    Finished `dev` profile target(s) in 3 minutes


Collected 30 test(s) from ibtc_cairo package
Running 0 test(s) from src/
Running 30 test(s) from tests/
[PASS] ibtc_cairo_integrationtest::test_ibtc_manager::test_contracts_are_deployed_correctly (gas: ~1703)
[PASS] ibtc_cairo_integrationtest::test_ibtc_manager::test_contract_is_pausable (gas: ~2635)
[PASS] ibtc_cairo_integrationtest::test_ibtc_manager::test_multiple_vault_setup (gas: ~3438)
[PASS] ibtc_cairo_integrationtest::test_ibtc_manager::test_revoke_approved_signer (gas: ~1927)
[PASS] ibtc_cairo_integrationtest::test_ibtc_manager::test_get_vault_by_index (gas: ~4816)
[PASS] ibtc_cairo_integrationtest::test_ibtc_manager::test_get_ibtc_vault (gas: ~4817)
[PASS] ibtc_cairo_integrationtest::test_ibtc_manager::test_set_minimum_deposit (gas: ~1713)
[PASS] ibtc_cairo_integrationtest::test_ibtc_manager::test_set_maximum_deposit (gas: ~1713)
[PASS] ibtc_cairo_integrationtest::test_ibtc_manager::test_deposit_too_much_bitcoin (gas: ~6309)
[PASS] ibtc_cairo_integrationtest::test_ibtc_manager::test_deposit_more_bitcoin (gas: ~6216)
[PASS] ibtc_cairo_integrationtest::test_ibtc_manager::test_tss_commitment (gas: ~1771)
[PASS] ibtc_cairo_integrationtest::test_ibtc_manager::test_set_status_funded_duplicate_signers (gas: ~3178)
[PASS] ibtc_cairo_integrationtest::test_ibtc_manager::test_set_threshold (gas: ~1717)
[PASS] ibtc_cairo_integrationtest::test_ibtc_manager::test_setup_vault (gas: ~2641)
[PASS] ibtc_cairo_integrationtest::test_ibtc_token::test_constructor (gas: ~433)
[PASS] ibtc_cairo_integrationtest::test_ibtc_token::test_burn_by_burner (gas: ~513)
[PASS] ibtc_cairo_integrationtest::test_ibtc_token::test_burn_by_unauthorized (gas: ~429)
[PASS] ibtc_cairo_integrationtest::test_ibtc_token::test_mint_by_minter (gas: ~633)
[PASS] ibtc_cairo_integrationtest::test_ibtc_manager::test_set_status_funded_validations (gas: ~4879)
[PASS] ibtc_cairo_integrationtest::test_ibtc_token::test_mint_by_owner (gas: ~565)
[PASS] ibtc_cairo_integrationtest::test_ibtc_token::test_mint_by_unauthorized (gas: ~429)
[PASS] ibtc_cairo_integrationtest::test_ibtc_token::test_set_burner (gas: ~495)
[PASS] ibtc_cairo_integrationtest::test_ibtc_token::test_set_minter (gas: ~495)
[PASS] ibtc_cairo_integrationtest::test_ibtc_token::test_set_minter_by_unauthorized (gas: ~429)
[PASS] ibtc_cairo_integrationtest::test_ibtc_manager::test_withdraw_half_locked_tokens (gas: ~4850)
[PASS] ibtc_cairo_integrationtest::test_ibtc_token::test_set_burner_by_unauthorized (gas: ~429)
[PASS] ibtc_cairo_integrationtest::test_ibtc_token::test_burn_from_by_unauthorized (gas: ~429)
[PASS] ibtc_cairo_integrationtest::test_ibtc_token::test_burn_from_by_owner (gas: ~512)
[PASS] ibtc_cairo_integrationtest::test_ibtc_manager::test_withdraw_and_redeem_bitcoin (gas: ~6247)
[PASS] ibtc_cairo_integrationtest::test_ibtc_manager::test_withdraw_redeem_too_much_bitcoin (gas: ~6348)
Tests: 30 passed, 0 failed, 0 skipped, 0 ignored, 0 filtered out
```

8.3 Automated Tools

8.3.1 AuditAgent

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

9 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development process, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

Blockchain Security: At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

Blockchain Core Development: Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

DevOps and Infrastructure Management: Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

Cryptography Research: At Nethermind, our Cryptography Research team 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

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.