# Security Review Report
# NM-0205 TWAMM
# (Ekubo Extension)

NETHERMIND
SECURITY

(April 4, 2024)

# Contents

# 1 Executive Summary

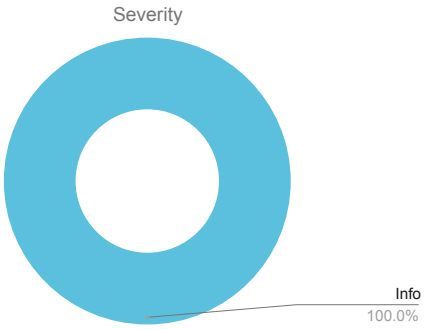
This document outlines the security review conducted by Nethermind Security for a Time-Weighted Average Market Maker (TWAMM implemented as an Ekubo extension. *Ekubo extensions* allow third-party developers to create new kinds of pools on *Ekubo* that integrate into the same ecosystem of aggregators and interfaces built on top of *Ekubo*.

Through the use of the *TWAMM* extension, traders can submit long-term orders, which are orders to sell a fixed amount of one of the assets over a fixed number of blocks. The *TWAMM* works by breaking long-term orders into infinitely many infinitely small pieces and executing them against an embedded constant-product AMM smoothly over time. The embedded AMM in this context is the *Ekubo* pool, which operates using the *TWAMM* as an extension.

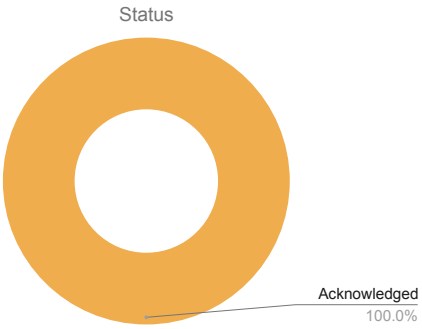
The team conducted a six-engineer-week review of 1,681 lines of code. The `Ekubo` team has provided documentation explaining the formulas used for the *TWAMM* implementation and how the extensions integrate with the rest of the protocol. Aside from the provided documentation, the `Ekubo` and `Nethermind Security` engaged in multiple calls to clarify any remaining questions about the expected behavior of the protocol.

**The audit was performed using**: (a) manual analysis of the codebase, and (b) simulation of the smart contracts. **Along this document, we report** one point of attention classified as `Informational`. The issues are summarized in Fig. 1.

**This document is organized as follows.** Section 2 presents the files in the scope of this audit. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology adopted for this audit. Section 6 details the issues. Section 7 describes complementary validations done during the assessment. 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** (1), **Best Practices** (0). **(b) Distribution of status: Fixed** (0), **Acknowledged** (1), **Mitigated** (0), **Unresolved** (0)

### Summary of the Audit

| | |
|---|---|
| **Audit Type** | Security Review |
| **Initial Report** | Mar 15, 2024 |
| **Final Report** | Apr 4, 2024 |
| **Methods** | Manual Review, Tests |
| **Repository** | EkuboProtocol |
| **Commit Hash** | 0050c6a14e6838d13b09c56089422fabec59c941 |
| **Final Commit Hash** | 7564ce718b02c43d957ac66b30503a023e7972ae |
| **Documentation** | Documentation |
| **Documentation Assessment** | High |
| **Test Suite Assessment** | High |

## 2 Audited Files

| | Contract | LoC | Comments | Ratio | Blank | Total |
|---|---|---|---|---|---|---|
| 1 | src/positions.cairo | 464 | 30 | 6.5% | 81 | 575 |
| 2 | src/extensions/twamm.cairo | 842 | 50 | 5.9% | 146 | 1038 |
| 3 | src/extensions/twamm/math.cairo | 95 | 19 | 20.0% | 24 | 138 |
| 4 | src/extensions/twamm/math/time.cairo | 19 | 5 | 26.3% | 6 | 30 |
| 5 | src/extensions/twamm/math/exp.cairo | 217 | 2 | 0.9% | 6 | 225 |
| 6 | src/extensions/interfaces/twamm.cairo | 44 | 14 | 31.8% | 13 | 71 |
| | **Total** | **1681** | **120** | **7.1%** | **276** | **2077** |

## 3 Summary of Issues

| | Finding | Severity | Update |
|---|---|---|---|
| 1 | Users may lose tokens in pools with low liquidity | Info | Acknowledged |

# 4 System Overview

This section presents basic concepts on *TWAMM* and an overview of the *TWAMM* extension.

## 4.1 TWAMM Concepts

*TWAMM* (Time-Weighted Average Market Maker), is a new type of AMM. It was established with the aim of assisting traders in executing large orders with minimal slippage and low gas fees, while ensuring minimal impact on the price.

The *TWAMM* provides the on-chain equivalent of the *TWAP* order. *TWAP* (Time-Weighted Average Price) is a popular type of algorithmic trade. In traditional finance, traders often use brokers to execute large orders via *TWAP* order. This strategy aims to mitigate market impact by breaking down large orders into smaller quantities and executing them at consistent intervals over time.

The *TWAMM* has specialized order-splitting logic and maintains a direct link to an integrated exchange, ensuring smooth execution with minimal gas fees. Arbitrageurs help align prices on the *TWAMM*'s exchange with market rates, enabling execution close to the *TWAP* of the asset. A *TWAMM* instance eases trading between a specific pair of assets. Specifically, it contains an embedded AMM for assets.

**General usage flow**. Users submit long-term orders to the *TWAMM*, specifying the sale of a fixed amount of one asset over a fixed number of blocks. These orders are fragmented into a large number of small virtual orders, which trade consistently against the embedded AMM over time.

## 4.2 Extensions

Ekubo implements the concept of Extensions that enables users to add custom logic at certain pool lifecycle events. These customized pools can implement different features such as, limit orders or *TWAMM* orders. Ekubo allows Extensions re-enter core contracts to set their own swaps, updates, etc. before/after the external contract interacts with a pool.

***TWAMM* extension flow**. Similarly to the general flow described in the previous section, users submit long-term orders to the *TWAMM* extension and it breaks these orders into infinitely many small virtual orders and executes them against the embedded AMM over time. In this case, the Ekubo pool serves as the embedded AMM.

As presented in the struct below, extensions are defined as part of the pool key in an immutable setting of a pool. When a pool is initialized and an extension contract exists, the extension is always called before the initialization.

```
pub struct PoolKey {
    pub token0: ContractAddress,
    pub token1: ContractAddress,
    pub fee: u128,
    pub tick_spacing: u128,
    pub extension: ContractAddress,
}
```

As users choose pools to interact with, it's crucial to carefully evaluate the extensions involved, as they significantly impact the pool's lifecycle. Errors in extension implementation may lead to users losing funds.

## 4.3 Main contracts

## 4.4 TWAMM

This contract implements the TWAMM extension to enable users place their long-term orders. The contract implements `ITWAMM` and `IExtension` interfaces. The `ITWAMM` consists of functions to perform TWAMM operations while `IExtension` has functions related to Ekubo's extensions.

1. `ITWAMM` **Interface** - The `TWAMM` contract implements this interface that provides the core operations:

A. `execute_virtual_orders(...)` - executes virtual orders. The function receives the struct `StateKey` listed below:

```
pub struct StateKey {
    pub token0: ContractAddress,
    pub token1: ContractAddress,
    pub fee: u128,
}
```

Then, it calls the function `lock(...)` in the `Core` contract to create a locked context for the caller where multiple swaps and liquidity management operations can be done. The passed data is sent back to the caller (TWAMM contract) through the `locked(...)` hook. Finally, `locked(...)` checks if the caller is the core contract before proceeding with the virtual orders execution.

Ⓑ. `update_order(...)` - updates an existing TWAMM order. The function receives the `salt`, `sale_rate_delta` and the struct `OrderKey` presented below:

```
pub struct OrderKey {
    pub sell_token: ContractAddress,
    pub buy_token: ContractAddress,
    pub fee: u128,
    pub start_time: u64,
    pub end_time: u64
}
```

Similarly to `execute_virtual_orders(...)`, the function `update_order(...)` calls `lock(...)` in the `Core` contract to create a locked context for the caller. The parameters received related to the order of the owner to be updated are sent back to the `TWAMM` contract through the `locked(...)` hook.

Then, all the steps to update an order are executed, e.g., the checking if the order is not ended, virtual order executions, save the order state, save rate state, etc. At the end, this function also may interact with the `Core` contract again to load the stored balance, check the liquidity pool, withdraw the specified token from the `Core` contract, pay a given token into the `Core`, and save tokens in the `Core` to be used for later operations.

Ⓒ. `collect_proceeds(...)` - collects proceeds from a `TWAMM` order. This function receives the salt and the struct `OrderKey` presented previously.

The flow of this function is similar to `update_order(...)`. It sends the owner address and the received parameters to `lock(...)` function in the `Core` contract that sends all them back to `TWAMM` contract to collect proceeds. This function executes the virtual orders and saves the reward rate to enable to know that the proceeds of the order have been withdrawn at the current time. At the end, this function also may interact with the `Core` contract again to load the stored balance and withdraw the specified token from the `Core` contract.

Ⓗ. The contract also implements several functions to get distinct information about sale rate and order state, e.g.: `get_sale_rate_and_last_virtual_order_time(...)` is used to return the current sale rates and the latest time orders were executed for a particular pool.

❷. `IExtension` **Interface** - the `TWAMM` extension also implements this interface that consists of three functions:

Ⓓ. `before_initialize_pool(...)` - this function is called by `Core` contract during the pool initialization process. Before the pool is initialized, this function receives the struct `PoolKey` (presented in Extension section) and checks if the `tick_spacing` is valid and sets the `StateKey` and `SaleRateState` in the storage. The struct `StateKey` is listed in Ⓐ and `SaleRateState` consists of the following information:

```
pub struct SaleRateState {
    pub token0_sale_rate: u128,
    pub token1_sale_rate: u128,
    pub last_virtual_order_time: u64
}
```

Ⓔ. `before_swap(...)` - this function is also called by `Core` contract before the exchange of tokens within a specific pool is executed. It receives `StateKey` data and passes it to the `execute_virtual_orders(...)`, presented in Ⓐ.

Ⓕ. `before_update_position(...)` - the `Core` contract calls this function before updating a position. This function receives `StateKey` data and information to update a position (struct `UpdatePositionParameters`):

```
pub struct UpdatePositionParameters {
    pub salt: felt252,
    pub bounds: Bounds,
    pub liquidity_delta: i129,
}
```

The function asserts the bounds are valid and then calls `execute_virtual_orders(...)`, presented in Ⓐ, passing the `StateKey` data.

## 4.5 Positions

The `TWAMM` contract maintains the state of all the created orders. The *Positions* contract enhances features for traders by tokenizing their orders' ownership as Non-Fungible Tokens (NFTs), enabling them to optimize capital efficiency. This tokenization enhances capital efficiency and introduces functionalities like allowing authorized accounts, other than the owner, to manage positions.

Users interact with *Positions* contract to submit long-term orders to the TWAMM (by minting a TWAMM position), update orders, and withdraw proceeds from a TWAMM position.

1. **`IPositions` Interface** The `Positions` contract implements this interface that provides several position operations management. The following functions are those which interact with the `TWAMM` contract.

Ⓖ. `mint_and_increase_sell_amount(...)` - this function mints a TWAMM position and sets the sale rate. This function receives the struct `OrderKey` (referenced in Ⓑ) and the amount to be deposited. Only authorized callers can call this function. For a given amount and the end time of order, the function calculates the sale rate and updates the order calling the `TWAMM.update_order(...)` (outlined in Ⓑ).

Ⓖ. `decrease_sale_rate(...)` - only authorized callers are permitted to invoke this function to decrease the sale rate. The function requires the struct `OrderKey` (in Ⓑ) and the sale rate delta to update the order by calling `TWAMM.update_order(...)` (in Ⓑ).

Ⓖ. `withdraw_proceeds_from_sale(...)` - only authorized callers are permitted to withdraw proceeds from a TWAMM position. This function receives an `OrderKey` and calls the function `collect_proceeds(...)` (outlined in Ⓒ) in the `TWAMM` contract to withdraw proceeds.

# 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] Users may lose tokens in pools with low liquidity

**File(s)**: twamm.cairo

**Description**: The TWAMM executes the existing orders against a pool. When all the existing orders in the TWAMM sell the same token, the TWAMM will execute a swap in the pool, using the total amount of tokens being sold for an exact input swap. This part of the execution can be appreciated in the following snippet of code.

```
...
let (amount, is_token1, sqrt_ratio_limit) = if token0_amount
    .is_non_zero() {
    (token0_amount, false, min_sqrt_ratio())
} else {
    (token1_amount, true, max_sqrt_ratio())
};

if sqrt_ratio_limit != sqrt_ratio {
    // @audit - This will try to swap all the tokens from the users
    delta = core
        .swap(
            pool_key,
            SwapParameters {
                amount: i129 { mag: amount, sign: false },
                is_token1,
                sqrt_ratio_limit,
                skip_ahead: 0
            }
        );
}
...
```

However, a problem arises when the pool lacks liquidity to receive all the tokens, and only part of them are swapped. The rest of the tokens will remain in the balance saved by the extension in the Ekubo contract. Users cannot access this residual because it has already been accounted for as sold.

**Recommendation(s)**: Consider accounting for the residual tokens so users can retrieve them. Optionally advise users not to set orders for pools with an amount of liquidity relatively close to the amount of tokens being traded.

**Status**: Acknowledged.

**Update from the client**: This is known and intentional. TWAMM orders will only be placed on pools with sufficient liquidity to absorb the trade, and in most cases orders will exist on both sides of the TWAMM, so in practice, this will never happen when selling tokens of non-zero value.

# 7 Complementary Validations

## 7.1 *exp_inner(...)* function

The *exp_inner(...)* function calculates $e^x$ for an input $x$, which is a *fixed-point 64.64* number less than 2, the result will be given as a *fixed-point 128.128* number. To assess this function's precision, the Nethermind Security team utilized the starknet foundry toolkit to generate ten million random pairs $(x, y)$, where $0 \leq x < 36893488147419103232$ (corresponding to the interval $[0, 2)$ in *fixed-point 64.64* format) and $y$ = *exp_inner*$(x)$ in *fixed-point 128.128* format.

Using SageMath, for each generated pair, we took the value $x$ and computed the value of $e^x$ using fixed-point numbers with a 256 bits mantissa. Using the result of this computation, which we called *real value*, we computed the relative error for value received from the *exp_inner(...)* function. The following expression was used to compute the relative error.

$$\frac{|real\_value - exp\_inner(x)|}{real\_value} * 100$$

The following code was used for executing this check

```
from math import e

R256 = RealField(256, rnd='RNDZ')
BASE = R256(e)

def parse_line(line):
    [exp, result] = line.split(':')
    return [int(exp.strip()), int(result.strip())]

def compute_real_value(value_64):
    exp = R256(value_64/2**64)
    return BASE ** exp


def compute_absolute_error(test_case):
    real_value = compute_real_value(test_case[0])
    real_value_u128 = real_value * 2**128
    return real_value_u128 - test_case[1]

def compute_relative_error(test_case):
    real_value = compute_real_value(test_case[0])
    real_value_u128 = real_value * 2**128
    return (abs(real_value_u128 - test_case[1]) / real_value_u128) * 100


with open("test_cases.txt", 'r') as file:
    lines = file.readlines()


test_cases = [parse_line(x) for x in lines]
errors = [compute_relative_error(x) for x in test_cases]
maximum_error = max(errors))

```

The maximum relative error obtained from all the pairs checked was approximately **1.063647541321l8e-14**, which we consider an acceptable precision for this function.

## 7.2 Execution when orders exist for both tokens

When there are active orders in the *TWAMM* for both tokens from the pool, the new price of the pool is calculated using the following formula:

$$\sqrt{sell\_price} * \frac{e^{\frac{2*t*\sqrt{X_{in}*Y_{in}}}{l}} - \frac{\sqrt{sell\_price}-\sqrt{price}}{\sqrt{sell\_price}+\sqrt{price}}}{e^{\frac{2*t*\sqrt{X_{in}*Y_{in}}}{l}} + \frac{\sqrt{sell\_price}-\sqrt{price}}{\sqrt{sell\_price}+\sqrt{price}}}$$

Where:

- $X_{in}$ is the total amount of *token0* being sold from the active orders.
- $Y_{in}$ is the total amount of *token1* being sold from the active orders.
- $sell\_price = \frac{Y_{in}}{X_{in}}$.
- $price$ is the current price from the pool.

After the new price is computed, a swap is executed so that the pool reaches the expected price. The swap is demonstrated in the following code snippet:

```
...
let twamm_delta = if (token0_amount.is_non_zero()
    && token1_amount.is_non_zero()) {
    next_sqrt_ratio =
        Option::Some(
            calculate_next_sqrt_ratio(
                sqrt_ratio,
                core.get_pool_liquidity(pool_key),
                token0_sale_rate,
                token1_sale_rate,
                time_elapsed
            )
        );
    // @audit - This swap is done such that the pool reaches the expected price
    delta = core
        .swap(
            pool_key,
            SwapParameters {
            // @audit - The amount to send is unlimited.
                amount: i129 {
                    mag: 0xffffffffffffffffffffffffffffffff, sign: false
                },
                is_token1: sqrt_ratio < next_sqrt_ratio.unwrap(),
                sqrt_ratio_limit: next_sqrt_ratio.unwrap(),
                skip_ahead: 0
            }
        );

    // both sides are swapping, twamm delta is the swap amounts needed to reach
    // the target price minus amounts in the twamm
    delta
        - Delta {
            amount0: i129 { mag: token0_amount, sign: false },
            amount1: i129 { mag: token1_amount, sign: false }
        }
...
```

The code snippet demonstrates that the *TWAMM* imposes no limit on the amount sent to the pool. If the necessary amount for executing the swap exceeds the amount sold by the active orders, the rest of the tokens will be taken from the balances saved by the *TWAMMs* for other pools.

The Nethermind Security team tried to determine how this could be exploited by an evil actor to steal assets from legit traders. Using the starknet foundry toolkit, multiple cases were checked to find if it was possible for the swap amount to exceed the amount sold by the active order. It was found that this could happen in multiple cases. The following table shows some of the parameters that cause this to happen:

| *current sqrt price* | *new sqrt price* | *liquidity* | *token0 sold* | *token1 sold* | *token0 swapped* | *token1 swapped* |
|---|---|---|---|---|---|---|
| 1844674843714833906100000 | 18446748437148339060999998 | 10702546923282819942115045000 | **319346** | 262175817 | **21405083720140812** ↑ | 0 |
| 1844674843714833906100000 | 18446748437148339060999999 | 1070254692328281994211 50 | **318** | 94 | **107025418601** ↑ | 0 |
| 8992152344810963542384217 | 899215234481096354238 4216 | 27317737700269151837427992243736 | **548056250** | 471471094 | **114962675770617605742** ↑ | 0 |
| 861702605361360308185377660163292 | 861702605361360308185377660163291 | 4878112417830401368044884336271 9866 | **152** | 126 | **223551** ↑ | 0 |
| 19499155677397362396269182716 | 19499155677397362396269182714 | 1287360344157664590500 | **4** | 9 | **2305** ↑ | 0 |

- ***current sqrt price*** -> Indicates the price of the pool at the moment of execution.
- ***new sqrt price*** -> Indicates the price limit used in the swap.
- ***liquidity*** -> Indicates the current liquidity in the pool.
- ***token{0, 1} sold*** -> Indicates the token{0, 1} amount being sold by the *TWAMM*.
- ***token{0, 1} swapped*** -> Indicates the resulting delta of the swap. ↑ Indicates tokens are swapped into the pool and ↓ Indicates tokens are swapped out of the pool.

The error occurs when the *new sqrt price* value is lower than what it should be due to rounding errors, and this small movement in the price requires an amount larger than the one being sold. As can be seen from the table, this happens when the *current sqrt price* is very small and/or liquidity is very high. For getting a pool to a state like this, a large number of *token0* is needed, sometimes the needed amount is even larger than $2^{128}$.

From the tests done, an attacker would need a large amount of *token0* to carry out an attack successfully. Note that this amount could not be obtained through a flash loan because orders are executed after time passes. If legit pools trigger this issue, it could lead to the *TWAMM* becoming insolvent, halting swaps and liquidity management actions by putting some pools in an invalid state.

**Final Thoughts:** Even though executing this attack requires substantial economic resources and may seem improbable, the Nethermind Security team recommends mitigating this risk by ensuring that the swapped amount never exceeds the amount sold in active orders.

# 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 Ekubo documentation**

The Ekubo team has provided documentation about how extensions integrate with the core protocol. Information on the derivation of the used formulas and core concepts of the *TWAMM* concept was also provided. The documentation about extensions is public as part of the developers documentation. Moreover, the Ekubo team was available to address any inquiries or concerns raised by Nethermind Security.

# 9 Test Suite Evaluation

## 9.1 Compilation Output

```
> scarb build
Compiling ekubo v0.1.0 (.../EkuboProtocol/contracts/Scarb.toml)
 Finished release target(s) in 10 seconds
```

## 9.2 Tests Output

```
> scarb test
    Running cairo-test ekubo
   Compiling test(ekubo_unittest) ekubo v0.1.0 (.../EkuboProtocol/contracts/Scarb.toml)
    Finished release target(s) in 19 seconds
testing ekubo ...
running 56 tests
test math_test::SaleRateTest::test_place_order_sale_rate ... ok (gas usage est.: 3333080)
test math::time_test::test_is_time_valid_future_times_near_second_boundary ... ok (gas usage est.: 1278340)
test math_test::TWAMMMathTest::test_calculate_next_sqrt_ratio ... ok (gas usage est.: 2943880)
test math_test::TWAMMMathTest::test_zero_c ... ok (gas usage est.: 1371080)
test math::exp_test::test_exp ... ok (gas usage est.: 81797500)
test math_test::SaleRateTest::test_calculate_amount_from_sale_rate ... ok (gas usage est.: 712250)
test math_test::RewardRateTest::test_largest_reward_amount_no_overflow ... ok (gas usage est.: 374560)
test math_test::test_calculate_reward_rate ... ok (gas usage est.: 676800)
test math_test::TWAMMMathTest::test_c_min_values ... ok (gas usage est.: 1828140)
test math::time_test::test_is_time_valid_past_or_close_time ... ok (gas usage est.: 1234410)
test math_test::SaleRateTest::test_sale_rates_smallest_amount ... ok (gas usage est.: 849670)
test math_test::TWAMMMathTest::test_c_max_values ... ok (gas usage est.: 456960)
test math::time_test::test_is_time_valid_future_times_near ... ok (gas usage est.: 1887760)
test math_test::SaleRateTest::test_sale_rates_overflow ... ok (gas usage est.: 31340)
test math_test::TWAMMMathTest::test_c_range ... ok (gas usage est.: 8226980)
test twamm_test::UpgradableTest::test_replace_class_hash_can_be_called_by_owner ... ok (gas usage est.: 720670)
test twamm_test::BitmapTest::test_time_spacing_sixteen ... ok (gas usage est.: 1606900)
test twamm_test::BitmapTest::test_next_initialized_time ... ok (gas usage est.: 31891088)
test twamm_test::CancelOrderTests::test_place_order_and_cancel_after_end_time ... ok (gas usage est.: 31957464)
test twamm_test::PlaceOrdersAndUpdateSaleRate::test_update_after_order_expiry ... ok (gas usage est.: 32024664)
test twamm_test::PlaceOrdersAndUpdateSaleRate::test_update_invalid_sale_rate ... ok (gas usage est.: 32054094)
test twamm_test::PoolTests::test_before_initialize_pool_invalid_tick_spacing ... ok (gas usage est.: 1901078)
test twamm_test::PoolTests::test_before_initialize_pool_valid_tick_spacing ... ok (gas usage est.: 1923078)
test twamm_test::PlaceOrderAndCheckExecutionTimesAndRates::test_place_orders_3 ... ok (gas usage est.: 90681802)
test twamm_test::CancelOrderTests::test_place_order_and_cancel_before_order_execution ... ok (gas usage est.: 34765904)
test twamm_test::PlaceOrdersAndUpdateSaleRate::test_decrease_order_sale_rate_before_order_starts_and_pay_fee_token0 ...
↪  ok (gas usage est.: 36140688)
test twamm_test::PoolTests::test_before_update_position_valid_bounds ... ok (gas usage est.: 32940078)
test twamm_test::PlaceOrdersAndUpdateSaleRate::test_decrease_order_sale_rate_before_order_starts_token1 ... ok (gas
↪  usage est.: 37953078)
test twamm_test::PlaceOrdersAndUpdateSaleRate::test_decrease_order_sale_rate_before_order_starts_token0 ... ok (gas
↪  usage est.: 37953078)
test twamm_test::PlaceOrderAndCheckExecutionTimesAndRates::test_place_orders_2 ... ok (gas usage est.: 90528078)
test twamm_test::PlaceOrdersAndUpdateSaleRate::test_increase_order_sale_rate_before_order_starts_token1 ... ok (gas
↪  usage est.: 38330158)
test twamm_test::PoolTests::test_before_update_position_invalid_bounds ... ok (gas usage est.: 27208108)
test twamm_test::PlaceOrdersAndUpdateSaleRate::test_update_at_order_expiry ... ok (gas usage est.: 32015764)
test twamm_test::CancelOrderTests::test_place_order_and_cancel_during_order_execution_without_withdrawing_proceeds ...
↪  ok (gas usage est.: 38754292)
test twamm_test::CancelOrderTests::test_place_order_and_cancel_before_full_order_execution ... ok (gas usage est.:
↪  43973102)
test twamm_test::PlaceOrdersAndUpdateSaleRate::test_decrease_order_sale_rate_before_order_starts_and_pay_fee_token1 ...
↪  ok (gas usage est.: 36140688)
test twamm_test::PlaceOrdersAndUpdateSaleRate::test_increase_order_sale_rate_before_order_starts_token0 ... ok (gas
↪  usage est.: 38330158)
test twamm_test::PlaceOrdersCheckDeltaAndNet::test_place_orders_0 ... ok (gas usage est.: 51482736)
test twamm_test::PlaceOrderDurationTooLong::test_order_duration_too_long_positions ... ok (gas usage est.: 24085800)
test twamm_test::PlaceOrdersAndUpdateSaleRate::test_decrease_order_sale_rate_after_order_starts_token0 ... ok (gas
↪  usage est.: 48233252)
```

```
test twamm_test::PlaceOrdersCheckDeltaAndNet::test_place_orders_2 ... ok (gas usage est.: 44687144)
test twamm_test::PlaceOrderOnBothSides::test_place_orders_and_swap ... ok (gas usage est.: 61501198)
test twamm_test::PlaceOrderDurationTooLong::test_order_duration_too_long_twamm ... ok (gas usage est.: 26272434)
test twamm_test::PlaceOrdersCheckDeltaAndNet::test_place_orders_1 ... ok (gas usage est.: 51448488)
test twamm_test::PlaceOrderOnBothSides::test_place_orders_2 ... ok (gas usage est.: 72737880)
test twamm_test::PlaceOrderOnBothSides::test_place_orders_1 ... ok (gas usage est.: 84071700)
test twamm_test::PlaceOrderOnOneSideAndWithdrawProceeds::test_place_orders_0 ... ok (gas usage est.: 56883840)
test twamm_test::PlaceOrderAndCheckExecutionTimesAndRates::test_place_orders_0 ... ok (gas usage est.: 41964808)
test twamm_test::PlaceOrderOnOneSideAndWithdrawProceeds::test_place_orders_1 ... ok (gas usage est.: 53208200)
test twamm_test::PlaceOrderAndCheckExecutionTimesAndRates::test_place_orders_1 ... ok (gas usage est.: 42863632)
test twamm_test::PlaceOrderOnBothSides::test_place_orders_and_update_position ... ok (gas usage est.: 70024642)
test twamm_test::PlaceOrderOnBothSides::test_place_orders_3 ... ok (gas usage est.: 73759476)
test twamm_test::PlaceOrderOnOneSideAndWithdrawProceeds::test_place_orders_3 ... ok (gas usage est.: 53130822)
test twamm_test::PlaceOrderOnOneSideAndWithdrawProceeds::test_place_orders_2 ... ok (gas usage est.: 56806462)
test twamm_test::PlaceOrdersCheckDeltaAndNet::test_place_orders_3 ... ok (gas usage est.: 44687144)
test twamm_test::PlaceOrderOnBothSides::test_place_orders_0 ... ok (gas usage est.: 72659540)
test result: ok. 56 passed; 0 failed; 0 ignored; 462 filtered out;
```

# 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 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