Version
v 1.0

Customer
Ekubo

15 April
2025

Invariant Analysis

ABDK Consulting

Your guide in the world of Crypto

Contents

1 Changelog

#	Date	Author	Description
0.1	12.04.25	M. Vladimirov	Initial Draft
0.2	15.04.25	D. Khovratovich	Minor revision
1.0	15.04.25	A. Zveryanskaya	Release

2 Introduction

Author: Mikhail Vladimirov

Scope of Analysis. ABDK Consulting analyzed the Ekubo protocol claims by the Ekubo team, given in the private communication. The results of the analysis are collected in this document. The materials provided include:

[1] Ekubo Desmos note

[2] Ekubo Protocol source code

During the analysis, ABDK Consulting continuously communicated with the Ekubo team, promptly clarifying any unclear matters.

3 Protocol summary

Ekubo is an AMM, featuring 100x concentrated liquidity, extensibility, and highly optimized smart contracts to provide the best execution and LP returns.

4 TWAMM Math Analysis

Ekubo team asked whether the `computeNextSqrtRatio` function matches the formula in the following Desmos note [1].
We've carefully reconstructed math from the code of the `computeNextSqrtRatio` function, as well as from other functions this function depends on. The results are provided below.

4.1 Notation

x : amount (unsigned integer)
d : duration (unsiged integer)
S : sale rate (unsigned 80.32)
ΔS : sale rate delta (signed 80.32)
R : reward rate (unsigned 160.96)
ρ : Square root ratio
ρ_s : Square root sale ratio (unsigned 128.128)
c : The c parameter (signed 128.128)
L : Liquidity amount

4.2 Functions

We have reconstructed the implemented functions as follows:
`computeSaleRate`
For an amount x and a duration d, compute the sale rate S as:

$$S = \frac{x}{d}$$

`addSaleRateDelta`
For a sale rate S and a salte rate delta ΔS, compute the new sale rate S' as:

$$S' = S + \Delta S$$

`computeAmountFromSaleRate`
For a sale rate S and a duration d, compute the amount x as:

$$x = \lfloor Sd \rfloor$$

or

$$x = \lceil Sd \rceil$$

depending on the desired rounding.
`computeRewardAmount`
For a reward rate R and a sale rate S, compute the reward amount y as

$$y = \lfloor RS \rfloor$$

`computeC`

For a square root ratio ρ, and a square root sale ratio ρ_s, compute the c parameter as:

$$c = \frac{\rho_s - \rho}{\rho_s + \rho_s}$$

`computeSqrtSaleRatio`

For a token 0 sale rate S_0 and a token 1 sale rate S_1, compute the square root sale ratio ρ_s as:

$$\rho_s = \sqrt{\frac{S_1}{S_0}}$$

`computeNextSqrtRatio`

For a square root ratio ρ, liquidity amount L, token 0 sale rate S_0, token 1 sale rate S_1, an elapsed time t, and a fee ϕ, compute the next square root ratio ρ' as:

$$\begin{aligned}
\rho_s &= \sqrt{\frac{S_1}{S_0}} \\
c &= \frac{\rho_s - \rho}{\rho_s + \rho_s} \\
\text{roundUp} &\equiv \rho > \rho_s \\
\epsilon &= 12392656037 \left(\sqrt{S_0 S_1} - \sqrt{S_0 S_1} \cdot \phi \right) \frac{t}{L}
\end{aligned}$$

So we have

$$\rho' = \begin{cases}
\left\lceil \sqrt{\frac{S_1}{S_0}} \right\rceil, \text{ if } (c = 0 \vee L = 0 \vee \epsilon \geqslant 64) \wedge \rho > \sqrt{\frac{S_1}{S_0}} \\
\left\lfloor \sqrt{\frac{S_1}{S_0}} \right\rfloor, \text{ if } (c = 0 \vee L = 0 \vee \epsilon \geqslant 64) \wedge \rho \leqslant \sqrt{\frac{S_1}{S_0}} \\
\max\left(\sqrt{\frac{S_1}{S_0}}, \sqrt{\frac{S_1}{S_0}} \frac{|2^\epsilon - c|}{|2^\epsilon + c|} \right), \text{ if } \neg(c = 0 \vee L = 0 \vee \epsilon \geqslant 64) \wedge \rho > \sqrt{\frac{S_1}{S_0}} \\
\min\left(\sqrt{\frac{S_1}{S_0}}, \sqrt{\frac{S_1}{S_0}} \frac{|2^\epsilon - c|}{|2^\epsilon + c|} \right), \text{ if } \neg(c = 0 \vee L = 0 \vee \epsilon \geqslant 64) \wedge \rho \leqslant \sqrt{\frac{S_1}{S_0}}
\end{cases}$$

4.3 Comparison with the formulas in [1]

The note [1] basically matches the main scenario implemented in the `computeNextSqrtRatio` function. However, [1] doesn't cover all the nuances implemented in the code. They don't match exactly, so as [1] describes a simplified version of what is actually implemented. Without proper analysis of the protocol math, which was not in the scope of this review, it is hard to tell whether the differences are insignificant or actually break the math. Here is the list of differences:

1. In case $c = 0$, $L = 0$, or exponent is too high ($\geqslant 2^{64}$), the code sets the next sqrt ratio equal to the sqrt sell ratio. [1] agrees with this approach only for the $c = 0$ case. The case $L = 0$ leads to division by zero in [1], and [1] doesn't have any specific high exponent threshold nor any special handling for high exponent values.

2. When c is negative, the code explicitly guarantees, that the next sqrt ratio may not be less than the sqrt sale ratio. For non-negative c, the code guarantees that the next sqrt ratio may not be greater than the sqrt sale ratio. While these constraints should hold for formulas in [1], they could be violated due to rounding, and [1] does not explicitly require these constraints to be strictly preserved.

3. The code uses several different number formats with different precisions, and without proper math analysis it is hard to tell whether these precisions are sufficient, as [1] does not provide any specific precision requirements.

4. The code uses different rounding directions in different cases, and without proper math analysis it is hard to tell whether these rounding directions are chosen correctly, as [1] doesn't provider any specific rounding requirements.

In order to make a final decision whether the code behaves as intended, there should be a more comprehensive specification, which justifies precisions, rounding directions, high exponent threshold, and special handling of edge cases in the code.
The most concerning difference is the special handling of the $\mathrm{exponent} \geqslant 2^{64}$. It seems that the threshold was chosen to prevent low-level overflow in the code, rather than according to some business-level requirements. Consider revisiting the threshold value.

5 Invariants

Also, the Ekubo team asked us to check the following invariants regarding the protocol:

Under the following assumptions:

- Total supply of any ERC20 token is less than 2^{128}
- Virtual orders for a pool are executed at least once every 2^{32} - 1 seconds after pool creation

These TWAMM extension invariants should be checked:

- The pool cannot be ”frozen”:
  - `beforeSwap`, `beforeUpdatePosition` and `beforeCollectFees` will never revert, i.e.:
    - Executing virtual orders `_executeVirtualOrdersFromWithinLock` will never throw an error
    - Executing virtual orders always costs less than 5 million gas
  - Therefore:
    - Liquidity positions in the TWAMM pool can *always* be withdrawn
    - Given valid swap arguments, the pool can always be swapped against
    - TWAMM orders ending in the future may be stopped
    - Non-zero proceeds can always be collected without error for a TWAMM order.
- The TWAMM extension will always be ”solvent”:
  - All orders can be stopped in any state, refunding the remaining sell amount minus the fee paid to liquidity providers
  - Non-zero proceeds can be collected for all orders in any state
- The orders receive approximately the time weighted average price:
  - The price received for every order must be between the ratio of sales and the pool ratio over the time period.

We checked these requirements. The results are summarized below.

5.1 The pool cannot be ”frozen”

We rephrase this as it shouldn’t be possible to put the pool in a non-operational state. There is no explicit “freeze” functionality explicitly coded in the protocol, but a few requirements should be satisfied so that one can not abuse to “freeze” the protocol:

`beforeSwap`, `beforeUpdatePosition` **and** `beforeCollectFees` **must never revert**

We have found that all these functions basically delegate to the

`_executeVirtualOrdersFromWithinLock` function, so these checks reduce to the following.

Executing virtual orders `_executeVirtualOrdersFromWithinLock` will never throw an error

We have found that the code can revert only when the pool is not initialized or is in an invalid state. So the protocol may freeze but only if the pool is non-operational, which would anyway prevent the protocol from working. Apart form this, there is no explicit logic that reverts or in another way “freezes” the protocol.

Executing virtual orders always costs less than 5 million gas

We have figured out that in order to execute virtual order, the code needs to check only a limited amount of valid times. This effectively limits the maximum gas amount spent while executing virtual orders. Exact gas usage could be measured in test environment by artificially constructing the worst possible scenario. We did not try to find it; but it should not be a problem for the code authors.
However, even if for now it is guaranteed that executing virtual orders always fits into the block gas limit, in the future both the block gas limit and gas costs of particular EVM operations could change. As an additional protection measure, one may implement a function that executes virtual orders up to certain time, so that virtual orders execution could be spread among several transactions.

Liquidity positions in the TWAMM pool can always be withdrawn

We have found that withdrawal is possible as long as virtual orders can be executed, so that we reduce to the case above. However, as a defensive measure, a special emergency mode could be implemented, when all trading is stopped, including virtual order execution, and liquidity could be withdrawn as is.

Given valid swap arguments, the pool can always be swapped against

We have found that a swap is possible as long as virtual orders can be executed, so that we again reduce to the case above. If the protocol is screwed up to a point where virtual orders cannot be executed, then normal swaps are probably shouldn’t be executed either.

TWAMM orders ending in the future may be stopped

We have found that a TWAMM order requires the virtual order functionality. In addition, ending an order executes a complex logic that involves updating the code pool state, TWAMM extension state, transferring tokens etc. This logic doesn’t contain control paths that explicitly prevent ending a TWAMM order, but its complexity necessitates an additional control mechanism. We recommend a special emergency mode, when all trading is stopped, including virtual order execution, but all virtual orders can be ended and refunded as is.

Non-zero proceeds can always be collected without error for a TWAMM order

We have found that a proceeds collection requires the virtual order functionality, but it is non-trivial to guarantee that there is no other way to block this functionality. Again, we recommend a defensive measure similar to that for TWAMM: a special emergency mode, when all trading is stopped, including virtual order execution, but proceeds can be collected as is.

5.2 The TWAMM extension will always be ″solvent″:

These invariants basically duplicate the invariants described above.

All orders can be stopped in any state, refunding the remaining sell amount minus the fee paid to liquidity providers

See above

Non-zero proceeds can be collected for all orders in any state

See above

5.3 The orders receive approximately the time weighted average price:

This invariant is not rigorous enough to be enforced. However, the protocol logic executes order at a price that could be called the time weights average price, as it averages the pool price over a time period.

The price received for every order must be between the ratio of sales and the pool ratio over the time period

We have found that both the ratio of sales and the pool ratio could change during a time period, so it is unclear how this condition should be checked. A protocol math analysis would be required for a formal answer.

6 Recommendations

We provide some recommendations which could increase confidence in the invariants above.

1. Revisit the $\mathrm{exponent} \geqslant 2^{64}$ threshold to ensure it is valid from the business point of view.
2. Revisit the precisions and rounding in math implementation to ensure they are valid from the business point of view. It is unclear, whether precisions and rounding in the code are based on some math analysis, or were chosen just for coding convenience. If there is no sufficient math analysis behind the code, then revisiting precisions and rounding could mean performing math analysis to determine what precisions and rounding should be used from business point of view. and then ensuring that code satisfies these requirements.
3. Reconstruct in a test environment the worst case virtual orders execution scenario and measure the exact gas usage for it. Ensure it is well below the block gas limit.
4. Implement an ability to execute virtual orders up to a certain time in the past, to make it possible spreading virtual orders execution across several blocks. This could help in emergency scenarios, such as a drastic decrease in the block gas limit.
5. Implement an “escape hatch” functionality, which would allow taking money out of the protocol in the case when the main protocol logic stopped working. This functionality should become available only when normal operations of of the protocol are broken.

ABDK
Consulting

About us

Established in 2016, is a leading service provider in the space of blockchain development and audit. It has contributed to numerous blockchain projects, and co-authored some widely known blockchain primitives like Poseidon hash function.

The ABDK Audit Team, led by Mikhail Vladimirov and Dmitry Khovratovich, has conducted over 40 audits of blockchain projects in Solidity, Rust, Circom, C++, JavaScript, and other languages.

Contact

Email

dmitry@abdkconsulting.com

Website

abdk.consulting

Twitter

twitter.com/ABDKconsulting

LinkedIn

linkedin.com/company/abdk-consulting