Security Audit Report

Gyroscope Protocol's Mathematical Model Implementation

Delivered: April 27th, 2022

Prepared for Gyro Finance by

runtime verification

Summary

Runtime Verification, Inc. conducted a security audit of the Gyroscope protocol contracts, with a specific focus on their mathematical model implementation. The audit was conducted from February 14–18, and March 21 to April 22, 2022.

The Gyroscope team arranged multiple audits running in parallel that are complementary to each other. Among those, we were specifically focusing on the implementation of the core math models of the protocol. The Gyroscope math models involve nontrivial real arithmetic, e.g., solving quadratic and cubic equations, and introducing nontrivial curves such as ellipses. It is thus challenging to faithfully and efficiently implement such a math model using fixed-point arithmetic in a way that is gas-efficient and high-precision. Moreover, numerical stability is critical for security of the system, since numerical errors may lead to malfunctions or (potentially exploitable) vulnerabilities when they are propagated and amplified further during the complex and lengthy arithmetic computations.

Considering the importance of the numerical implementation security, our audit in this engagement were particularly focusing on the following core critical components of the protocol:

- Gyroscope Primary Algorithmic Market Maker (P-AMM)'s math model implementation
- Constant-Ellipse Market Maker (CEMM)'s math model implementation
- LP share pricing math implementation
- Motherboard and Safety Checks
    - *Please note these components only received a single review. The issues identified can be found in this report under "Other Findings". As such, this component should be the subject of a future audit at a date and time to be determined by the code's authors.*

As a result of this audit, we revealed a number of critical flaws in the implementation, as well as various precision loss and numerical issues, and recommended potential improvements.

We note that, however, because of the nature of this focused audit, we had to assume the functional correctness and security of the other parts of the system and their integration with the core math implementation. *Thus, it is integral to check carefully whether the excluded parts are covered by the other independent audits, and as a result, our assumptions are effectively validated by them.* We leave this to the Gyroscope team.

Scope and Result

Gyroscope is an algorithmic stablecoin. Unlike Maker or other collateral-based pegging systems, Gyroscope does *not* require (over-)collateralization to mint Gyro tokens. Users may freely mint tokens at the price of 1 USD[1], or buy them at a discount on external markets if available. Users may also redeem the tokens at the price of 1 USD as long as its pegging is maintained or the overall redemption rate is not too high. This mechanism thus provides arbitrage opportunities. Maintaining its pegging relies on this arbitrage opportunity as well as the reserve-generated yields.

The redemption price is algorithmically determined by the Gyroscope primary algorithmic market maker (P-AMM), which depends on the current redemption rate (which is time-exponentially-decreasing sum of historical redemption amounts) and the current reserve value (against the total supply of Gyro tokens). In the beginning of each block, given the current state, the P-AMM induces a new redemption curve, and this curve determines the redemption price throughout the block.

On the other hand, the reserve consists of diversified (yield-bearing) assets to reduce the effect of market volatility. To this end, the Gyroscope team also introduced two new automatic market makers (AMMs), constant-ellipse market maker (CEMM) and constant-product market maker with virtual reserves (CPMMv), designed particularly for stablecoins or relatively stable asset pairs. Compared to the existing CPMMs, both the new AMMs enjoy better capital efficiency and less slippage, thus higher yields in the end for their use cases.

The USD value of the reserve is computed using the price oracles of underlying assets. Chainlink price feeds (with extra adjustments for better security) are used for popular high-volume assets. For LP share tokens, the protocol adopts a new method of computing the price of LP tokens, which is argued to be harder to manipulate by adversaries. Specifically, the new method relies on the oracle prices of individual pool assets, instead of the pool reserve balance of the assets, where the former is (assumed to be) harder to manipulate than the latter.

The major innovations described above, i.e., the P-AMM, the CEMM, the CPMMv, and the new LP pricing, enjoy the said benefits at the cost of higher computation complexity. Operations on the new AMMs involve solving quadratic or cubic equations, which in turn requires calculating square roots or running iterative methods that are inherently gas-expensive. Moreover, all these computations are performed using fixed-point arithmetic, where numerical stability is non-trivial to achieve or analyze.

Below we describe the detailed scope and result for each target component mentioned earlier.

[1] In the original design presented in the whie paper, users may need to pay a premium for minting when the market price is too high, but the current version doesn't implement the premium yet.

Primary Algorithmic Market Maker (P-AMM)'s math model implementation

The Gyroscope primary algorithmic market maker (P-AMM) provides a redemption price curve for the Gyro tokens. The price is nontrivial when the reserve value per Gyro token is less than 1, but greater than a minimum that is externally given.[2] In that case, the price remains at 1 up to a certain amount, and then linearly decreases until it is equal to the reserve ratio. Specifically, the price curve is piecewise linear with three pieces, where the first and third pieces are constant, and the second piece is decreasing with a slope α. An important property of this curve is that the reserve ratio decreases over the first and second pieces, and then stays constant in the third piece, where the final ratio is equal to the constant of the third piece. There may exist multiple curves that satisfy this property, and the P-AMM chooses one that minimizes the price decay speed. The P-AMM paper presents an algorithm that constructs such a curve given the current state. The algorithm is implemented in the PrimaryAMMV1 contract, and executed in every call to the redeem function..

Scope

In this audit, we checked whether the PrimaryAMMV1 contract faithfully implements the P-AMM math (ultimately Algorithm 4), and analyzed the robustness of the implementation against numerical corner cases. Specifically, we checked the following functions:

Functions in the PrimaryAMMV1 contract	Corresponding parts in the P-AMM paper
computeAlphaHat()	Proposition 3
computeReserveFixedParams()	Proposition 1
computeXl()	Proposition 2
computeXuHat()	Proposition 4
computeBa()	Lemma 4
createDerivedParams()	Algorithm 1
computeReserve(), computeRedeemAmount()	Algorithm 4
isInFirstRegion(), isInSecondRegion(), isInSecondRegionHigh(), isInThirdRegionHigh(), computeReserveValueRegion()	Algorithm 2

[2] The price is fixed at 1 when the reserve ratio is greater than 1. When the reserve ratio is less than the minimum, the price is set to the ratio.

computeAnchoredReserveValue()	Algorithm 3

Checked functions

The target source file in the scope of this audit is:

- protocol/contracts/PrimaryAMMV1.sol at git-commit-id 03f74bd.

The reference papers given for this audit are:

- research/P-AMM_design/pamm-technical-paper/main.pdf at git-commit-id 89bdccc.
- research/P-AMM_design/pamm-summary/PAMM summary.pdf at git-commit-id b9bc50e.

NOTE: We did *not* check the correctness of the curve construction algorithm itself. Also, the numerical stability of Algorithm 2 (and in turn Algorithm 3) was *not* verified — we did *not* check whether a different case analysis result due to numerical errors near the boundary of the conditions in Algorithm 2 does not lead to a significant difference in the result of Algorithm 3 as well as Algorithm 4.

Findings

As a result of this audit, we found discrepancies between the math and the implementation (Issues #45–#47, #51). We also identified several places that may employ extra numerical error handling or sanity checks (Issues #49, #50, #52, #53).

	Issue	Category	Status
#45	PrimaryAMMV1: inconsistency between createDerivedParams() and Algorithm 1	Bug	Addressed by client in this commit
#46	PrimaryAMMV1: inconsistency between isInThirdRegionHigh() and Algorithm 2	Bug	Addressed by client in this commit
#47	PrimaryAMMV1: computeReserveValueRegion(): condition for CASE_I_ii	Bug	Addressed by client in this commit
#49	PrimaryAMMV1: handling corner cases in computeXl()	Sanity Check	Addressed by client in this commit

#50	PrimaryAMMV1: potential subtraction overflow in computeXuHat()	Sanity Check	Addressed[3] by client in this commit
#51	PrimaryAMMV1: issues in outflow tracking	Bug	Addressed by client in this commit
#52	PrimaryAMMV1: logically infeasible path in computeXl()	Sanity Check	Addressed by client in this commit
#53	PrimaryAMMV1: additional sanity checks for computeRedeemAmount()	Sanity Check	Addressed[4] by client in this commit
#62	PrimaryAMMV1: sanity check for setting systemParams	Sanity Check	Addressed by client in this commit

Findings in P-AMM

Status: The Gyroscope team stated that the issues in the above table have been reviewed and the code has been subject to a subsequent audit by a different auditing company.

Constant-Ellipse Market Maker (CEMM)'s math model implementation

The constant-ellipse market maker (CEMM) is an automated market maker where the price curve is an ellipse, more specifically part of an ellipse that is rotated and shifted. Similar to the constant-product market maker with virtual reserve, the CEMM achieves the capital efficiency where the reserve can be fully utilized within the given price bounds. The ellipse curve of the CEMM is developed as a stretched circle, and the radius of the original circle is the invariant *L*.

[3] **The client states:** Mathematically, `\hat x_u(\alpha)` is always non-negative when α is chosen according to Proposition 3. We have added a proof of this statement in a new appendix in the PAMM technical paper (Lemma 7 in Appendix B). We understand that this property may be violated due to numerical errors and otherwise agree with the comments as well, and we have changed the code accordingly.

[4] **The client states:** We need this statement for the *strict* version of this statement, i.e., we need to show that `\bar\theta < b_0/y_0 < 1` implies that `\bar\theta < b_a/y_a < 1`. This is because the algorithm catches `b_0/y_0 \le \bar\theta` and `b_0/y_0 \ge 1` as special cases at the beginning of `computeRedeemAmount()`. We have added a proof of the implication for strict inequalities in a new appendix in the PAMM technical paper (Lemma 8 in Appendix B).
Independently of this, we agree with the statement regarding numerical errors and we have modified the code accordingly.

However, the invariant increases after each swap, because the swap fees are compounded for better yields. Thus, given the reserve state *(x,y)* and the swap request *dx*, it first computes *L* from *(x,y)* by solving a quadratic equation for *L*, and then it computes *dy* from *(x,y)* and *L* by solving another quadratic equation for (*y*+*dy*). The two quadratic equations are given in Propositions 13 and 14 in the CEMM paper, respectively.

Scope

In this audit, we checked whether the GyroCEMMMath contract faithfully implements the CEMM math formulae, and analyzed the numerical error bounds of the implementation. Specifically, we checked the following functions against the math:

Functions in the GyroCEMMMath contract	Corresponding parts in the CEMM paper
scalarProdUp(), scalarProdDown()	
mulAinv(), mulA()	Section 2.2
virtualOffsets(), maxBalances()	Proposition 7
chi()	Proposition 8
calculateInvariant(), _calculateInvariant(), solveQuadraticPlus()	Proposition 13
calcOutGivenIn(), calcInGivenOut(), calcYGivenX(), calcXGivenY(), solveQuadraticMinus()	Proposition 14

Checked functions

NOTE: We did *not* check the other functions of the GyroCEMMMath contract, that is, validateNormed(), validateDerivedParams(), zeta(), tau(), eta(), calculatePrice(), checkAssetBounds(), and liquidityInvariantUpdate().

The target source file in the scope of this audit is:

- vaults/contracts/cemm/GyroCEMMMath.sol at git-commit-id c955ecd.

The reference paper given for this audit is:

- research/AMM technical papers/CEMM notes clean.pdf at git-commit-id 89bdccc.

NOTE: We have ***not*** audited the other CEMM pool contracts (e.g., GyroCEMMPool.sol and GyroCEMMOracleMath.sol). The functional correctness and security of the overall CEMM pool implementation needs to be audited separately. Moreover, the other two pools, the CPMMv pools for 2- and 3-assets, are *not* in the scope of this audit, which also needs a separate audit.

Findings

As a result of this audit, we found a critical mistake in their pool integration (Issue #15). We also identified several precision loss issues (Issues #16–#24), and recommended potential improvements. The numeral error analysis result is available here.

	Issue	Category
#15	_onSwapGivenIn() calls GyroCEMMMath.calcInGivenOut()	Bug
#16	SignedFixedPoint rounding	Precision
#17	Non-optimal precision of mulA()	Precision
#18	Non-optimal precision of mulAinv()	Precision
#19	Non-optimal precision of virtualOffsets()	Precision
#20	Non-optimal precision of maxBalances()	Precision
#21	Non-optimal precision of calcYGivenX() and calcXGivenY()	Precision
#23	Unnecessary precision loss in fee addition	Precision
#24	Unnecessary constant factors in solving quadratic equations	Precision

Findings in CEMM

Status: The Gyroscope team stated that the issues in the above table have been reviewed, the code has been subject to a major revision, and the code has been subject to a subsequent audit by a different auditing company.

LP share pricing math implementation

The LP share price of the underlying pools is used to compute the Gyroscope reserve value as well as the amount of proceeds for mint and redeem. To make it harder to manipulate the price, the LP share price is determined by only the oracle prices of pool assets and the pool invariant value per LP share. Since the pool invariant per share changes only by compounding fees, the security of the LP price should mainly depend on the oracle prices of assets. This LP pricing scheme is implemented in the BalancerLPSharePricing contract that supports only the Balancer pools at this moment.

Scope

Assuming that the oracle prices are trusted, we reviewed the security of the LP pricing logic and its use. We also checked whether the BalancerLPSharePricing contract faithfully implements the pricing formulae, and analyzed the numerical errors. Specifically, we checked the following functions:

Functions in the BalancerLPSharePricing contract	Corresponding parts in the Oracles paper
priceBptCPMM(), priceBptTwoAssetCPMM(), priceBptCPMMEqualWeights()	Section 4.1
priceBptCPMMv2()	Section 4.2
priceBptCPMMv3(), relativeEquilibriumPricesCPMMv3()	Section 4.3
priceBptCEMM(), scalarProdDown(), mulAinv(), mulA(), zeta(), tau(), eta()	Section 4.4

Checked functions

The target source file in the scope of this audit is:

- protocol/contracts/oracles/balancer/BalancerLPSharePricing.sol at git-commit-id 03f74bd.

The reference papers given for this audit are:

- oracle-design/write-up-working/main.pdf at git-commit-id f8856e2.
- research/AMM technical papers/CPMMv 2d and 3d.pdf at git-commit-id 89bdccc.

Findings

As a result of this audit, we found a critical mistake in their vault integration (Issue #54). We also identified several precision loss issues (Issues #56).

	Issue	Category	Status
#54	BaseVaultPriceOracle: incorrect getPriceUSD()	Bug	Addressed by client in this commit

#55	BalancerLPSharePricing: minor gas-optimization for priceBptCPMMv2()	Gas Optimization	Addressed by client in this commit
#56	BalancerLPSharePricing: precision bias in priceBptCPMMv2()	Precision	Acknowledged by client
#57	BalancerLPSharePricing: rounding direction in relativeEquilibriumPricesCPMMv3()	Precision	Acknowledged by client
#58	BalancerLPSharePricing: rounding direction in priceBptCPMMv3()	Precision	Acknowledged by client
#59	SignedFixedPoint: incorrect divUp()	Bug	Addressed by client in this PR
#60	LP share pricing: regarding price manipulation	Security	Acknowledged by client
#61	LP share pricing: temporary gap between the spot price and the oracle-based (equilibrium) price	Security	Acknowledged by client

Findings in LP share pricing

Status: The Gyroscope team stated that the issues in the above table have been reviewed and the code has been subject to a subsequent audit by a different auditing company.
For further information on the status of the findings, we refer the reader to the findings' links.

Future Directions

We have *not* verified the correctness of the library contracts. Some of the libraries, e.g., SignedFixedPoint and Arrays, are newly written by the client, and their correctness is nontrivial. Thorough reviews are highly recommended.

Some of the LP pricing functions in the BalancerLPSharePricing contract require that the underlying asset prices are given in the correct order. It may need to be validated that the current Balancer integration satisfies the requirements.

In this audit, we assumed that the ratio between the underlying asset prices is *not* extremely large or small. It is desired to analyze the numerical stability of the LP pricing under the potentially extreme price ratios, e.g., one of the asset prices crashing to zero.

Other findings

To better understand the context in which the aforementioned math implementations are used, we had to refer to the other related contracts such as Motherboard, which led to revealing several issues as below.

NOTE: We have *not* thoroughly reviewed those contracts, and thus certain critical issues may have been missed while we skimmed through. A separate in-depth audit is needed for them.

	Issue	Category	Status
#38	Motherboard: use of safeApprove() in _convertMintInputAssetToVaultToken()	Best Practice	Addressed by client in this commit
#40	Motherboard: flaws in _getAssetAmountMint()	Bug	Addressed by client in this commit
#41	RootSafetyCheck: flaws in checkAndPersistMint()	Bug	Addressed by client in this PR
#42	Motherboard: unchecked result of the safety check	Bug	Addressed by client in this PR
#43	VaultRegistry: deregisterVault() not recycle vault metadata	Best Practice	Addressed by client in this commit
#63	LPTokenExchangerRegistry: sanity checks for register and deregister	Sanity Check	Addressed by client in this commit
#64	Motherboard: double-spending exploits for redeem()	Bug	Addressed by client in this commit
#65	BaseVault: precision issue in depositFor()	Precision	Addressed by client in this commit
#66	BaseVault: handling exceptional cases in exchangeRate()	Security	Addressed by client in this commit

#67	VaultSafetyMode: assertion never violated in checkAndPersistMint()	Code Readability	Addressed by client in this PR
#68	Flow: sanity check for updateFlow()	Sanity Check	Addressed by client in this commit

Findings in Motherboard

Status: The Gyroscope team stated that the issues in the above table have been reviewed and the code has been subject to a subsequent audit by a different auditing company.

Future Directions

We found that the Motherboard architecture is somewhat monolithic. It may be refactored so that mint()/redeem() in multiple assets can be constructed from several mint()/redeem() in a single asset. This will simplify the logic with less use of loops, avoiding potential mistakes like Issues #40 and #64. It will also improve the gas consumption, since it will no longer need to iterate all vaults everytime.

We have *not* systematically verified that the normalization and denormalization of the number of decimals are performed flawlessly in every required place. A separate review would be recommended.

Assumptions

The audit is based on the following assumptions and trust model.

- The PAMM and CEMM mathematical models are sound both mathematically and economically. We trusted their mathematical development and proofs. We have *not* performed the economic analysis of the model, e.g., how robust it is against black swan events, or whether it is game-theoretically safe. We also assumed that the desired properties specified in the PAMM summary paper hold.
- The price oracles and their integration are correct and secure. The price oracle mechanism was *not* in the scope of this audit. An additional audit for the oracles is highly recommended to detect price manipulation vulnerabilities, if any, related to the oracles.
- The exchanger implementation and integration is correct and secure. The exchanger code was *not* in the scope of this audit, thus any system behaviors that involve the exchanger (e.g., depositing or redeeming in the underlying tokens of vaults for mint() and redeem()) were not reviewed. An additional audit for the exchange code is recommended.

- We assumed that the Balancer's fixed-point math implementation, especially their approximated exponential function implementation, pow(), is correct and precise enough.
- External token contracts conform to the ERC20 standard. Also, they do *not* allow any callbacks (e.g., ERC721 or ERC777) or any external contract calls.
- For the authorizable contracts, their owners and authorized users are trusted to behave correctly and honestly.
- The contracts will be deployed and integrated correctly.

This audit is limited in scope within the boundary of the Solidity contract only. Off-chain and client-side portions of the codebase as well as deployment and upgrade scripts are *not* in the scope of this engagement, but are assumed to be correct.

Disclaimer