Security Assessment & Formal Verification Final Report

Beefy Finance

June / 24

Prepared for
Beefy Finance

certora

Table of content

Project Summary **3**
Project Scope 3
Project Overview 3
Protocol Overview 3
Findings Summary 5
Severity Matrix 5
Detailed Findings **6**
Medium Severity Issues **7**
M-01. Withdrawals Are Disabled When a Global Pause Is In Effect If the Individual Strategy Is Not Paused 7
M-02. Fee Income Can Be Lost Due to Lack of Slippage Protection on Swaps of Fees to Native Tokens 8
M-03. MEV Extraction Via Deposit Rebalancing and Withdraw Logic 8
Low Severity Issues **10**
L-01. Vaults and Strategies Can Be Maliciously Initialized Due to Lack of Atomic Initialization 10
L-02. The Alt Position May Not Always Lie Within the Main Range 10
L-03. getTokensRequired() might return smaller tokens required with bigger deposits 11
L-04. liquidity can get left behind in old positions when strategy moves to a new position 14
Informational Severity Issues 14
I-01. Events Are Emitted For Path Updates Even When No Update Occurs 14
I-02. Keepers Can Circumvent The Retirement of a Strategy 15
Formal Verification **16**
Assumptions and Simplifications 16
Verification Notations 16
Formal Verification Properties 17
BeefyVaultConcLiq.sol 17
Properties 17
StrategyPassiveManagerUniswap.sol 20
Properties 21
Disclaimer **23**
About Certora **23**

certora

Project Summary

Project Scope

Project Name	Repository	Last Commit Hash	Platform
Cowcentrated Contracts	https://github.com/beefyfinance/cowcentrated-contracts	18c36e	EVM/Solidity 0.8.23

Project Overview

This document describes the specification and verification of the **Beefy Vault & Strategy** using the Certora Prover and manual code review findings. The work was undertaken from **April 30, 2024** to **May 31, 2024**.

The following contract list is included in our scope:

- **`BeefyVaultConcLiq.sol`**
- **`BeefyVaultConcLiqFactory.sol`**
- **`StrategyPassiveManagerUniswap.sol`**
- **`StrategyFactory.sol`**

The Certora Prover demonstrated that the implementation of the **`BeefyVaultConcLiq` and `StrategyPassiveManagerUniswap`** contracts above is correct with respect to most of the formal rules written by the Certora team. The formal verification also indicated some issues as detailed out in the findings section. In addition, the team performed a manual audit of all the Solidity contracts. During the verification process and the manual audit, the Certora team discovered bugs in the Solidity contracts code, as listed below.

Protocol Overview

Beefy's cowcentrated contracts provide an automated yield optimizer based on Uniswap V3's concentrated liquidity pools. Concentrated liquidity pools allow users to define a price range for which they want to provide liquidity. As a consequence Uniswap V3 liquidity pools enhance control over capital allocation as well as appropriate compensation for various degrees of risk taking. However, they come with a limitation in auto-compounding: price ranges are statically chosen and pool rewards are not auto-compounded by LPs but need to be manually claimed and redeposited.

Beefy Finance bridges the gap between automated yield optimization and concentrated LPs by providing concentrated liquidity management for Uniswap V3 pools. That is, automating and aggregating complex on-chain activities to reduce costs and enhance rewards on provided liquidity without the effort and risk of errors of manually managing concentrated positions.

To do so, Beefy's Cowcentrated Liquidity Management provides Beefy vaults and strategies to manage user funds. Vaults automatically invest and manage user funds such that pool rewards are auto-compounded and concentrated liquidity positions are optimized according to current prices. Additionally, pool rewards are automatically harvested and reinvested by Beefy's strategies in an aggregated way, optimizing gas costs and saving on transaction fees.

certora

Findings Summary

The table below summarizes the findings of the review, including type and severity details.

Severity	Discovered	Confirmed	Fixed
Medium	3	3	2
Low	4	4	1
Informational	2	2	2
Total	9	9	5

Severity Matrix

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
		Probability		

certora

Detailed Findings

ID	Title	Severity	Status
M-01	Withdrawals are disabled when a global pause is in effect if the individual strategy is not paused	Medium	Fixed
M-02	Fee income can be lost due to lack of slippage protection on swaps of fees to native tokens	Medium	Fixed
M-03	MEV Extraction via deposit rebalancing and withdraw logic	Medium	Fixed
L-01	Vaults and Strategies can be maliciously initialized due to lack of atomic initialization	Low	Acknowledged
L-02	The alt position may not always lie within the main range	Low	Acknowledged
L-03	getTokensRequired() might return smaller tokens required with bigger deposits	Low	Fixed
L-04	Liquidity can be left behind when strategy moves to a new position	Low	Acknowledged
I-01	Events are emitted for path updates even when no update occurs	Informational	Fixed

I-02	Keepers can circumvent the retirement of a strategy	Informational	Fixed

Medium Severity Issues

M-01. Withdrawals Are Disabled When a Global Pause Is In Effect If the Individual Strategy Is Not Paused

Impact: Low
Probability: High

Description: In the `withdraw()` function of the StrategyPassiveManagerUniswap contract, the `_addLiquidity()` function is called only if the strategy itself is not paused:

```
Unset
        if (!paused()) _addLiquidity();
```

Because the `paused()` function only returns the truth value of the per-strategy pause, if a global pause (as determined by the `StrategyFactory` contract associated with the strategy instance) is in effect, `_addLiquidity()` will be called anyway. Because `_addLiquidity()` calls the `_whenStrategyNotPaused()` function, which reverts if either the individual strategy is paused or if a global pause is in effect, any attempt to withdraw from an unpaused strategy when there is a global pause in effect will revert. This could block users from withdrawing funds at critical moments when they expected to be able to, in particular when there is an ongoing exploit and their funds may be at risk.

Recommendations: Modify the check in `withdraw()` so that `_addLiquidity()` is only called if the strategy is not paused and the global pause is not in effect.

Customer's response: fix commit

Fix Review: Mitigation Verified; `withdraw()` will now succeed under any configuration of the strategy-specific and global pauses.

M-02. Fee Income Can Be Lost Due to Lack of Slippage Protection on Swaps of Fees to Native Tokens

Impact: Medium
Probability: Medium

Description: The strategy distributes a portion of its earnings to 1) the caller of `harvest()`, 2) the strategist, and 3) the Beefy protocol itself. This portion is converted to native tokens, if necessary, before being distributed. The swaps to native tokens are done without any slippage protection, allowing them to be sandwiched by MEV bots to extract value away from the intended recipients. A sandwich attacker can even sandwich their own call `harvest()` to additionally receive the call fee, making the activity even more likely to be profitable and simpler to execute. Frequently harvesting the pool and ensuring the routes used to swap to native tokens have deep liquidity can minimize attack profitability, but are not necessarily sufficient to protect against loss of revenue.

Recommendations: Consider adding slippage protection to the swaps to native tokens.

Customer's response: Fixed in commit

M-03. MEV Extraction Via Deposit Rebalancing and Withdraw Logic

Impact: Medium
Probability: Medium

Description: The Vault logic requires that if it is not holding equal value in both tokens (as defined by the price of the underlying Uniswap pool) when a deposit is made, the depositor must first provide the token of which there is a deficit until the values are equalized, and then must provide tokens at the ratio determined by the pool price. This is intended to mitigate impermanent loss for prior LPs and ensure that there is liquidity on both sides of the price to capture more fees. When a withdrawal is made, it is always done on a proportional basis to the

LP shares redeemed for both assets (so if, say, 10% of shares are redeemed, the withdrawer receives 10% of the balance of each token held). These mechanics combined allow swapping between the two tokens at the pool price in the opposite direction of the Vault's value imbalance without paying fees or causing price impact. This is done by making a deposit in only one token, and then immediately withdrawing the same value (based on the pool price) in a combination of both tokens.

Depending on the external market price relative to the Uniswap pool price, this may be favorable or unfavorable to depositors. When it is unfavorable, depositors have the option to simply wait for better conditions. When it is favorable, they may execute either atomic arbitrage against other DEXes or statistical arbitrage against CEXes. Due to this adverse selection, over time value will be leaked to MEV extraction once the market becomes sufficiently aware of this weakness. Vault LPs are not compensated for this risk via fees, nor does the pool price move to reverse impermanent loss (making the implemented mechanism ineffective at one of its goals). The requirement that deposits must occur during a calm period may lessen the opportunities to exploit this but does not (and cannot) eliminate it.

Recommendations: There are a number of possible solutions. Requiring deposits to always be done proportionally to the asset ratio in the Vault would work, but defeats the goal of balancing liquidity to maximize fee capture. A simple fee on deposit or withdrawal would work, but may discourage intended usage. A locking period (withdrawal only possible after a certain amount of time has passed) or a decaying fee (e.g. a withdrawal fee that goes to zero a fixed amount of time post-deposit), though more complex to implement, would allow preserving the rebalancing logic and avoid imposing a penalty on non-extractive depositors.

Customer's response: Fixed in pull request

Fix Review: By charging a fee on the "swapped" portion of a deposit, a cost hurdle is introduced for value extractors and preexisting LPs are compensated for the risk of offering a fixed-price swap. This mostly mitigates the issue, and is likely sufficient in practice, although in some situations value extraction can still be feasible. The team is encouraged to write thorough unit tests for the new logic, and to consider that on chains with very cheap transaction costs, asset tokens with low decimals may be vulnerable to "deposit splitting" to avoid the fee, as it is calculated with truncating division so that sufficiently small deposits do not incur a fee.
Given the extent of this fix, we reran and verified the formal specifications of

BeefyVaultConcLiq on the fix. The issue L-03 persists in this current fix. All other rules were proved correct.

Low Severity Issues

L-01. Vaults and Strategies Can Be Maliciously Initialized Due to Lack of Atomic Initialization

Impact: Low
Probability: Low

Description: The factories for both the vaults and strategies permit a minor griefing vector as a consequence of not allowing atomic initialization of new proxy instances--a call to `cloneVault()` or `createStrategy()` could be backrun with an attacker-initiated transaction that calls `initialize()`, forcing re-doing the cloning/creation until the deployer successfully calls `initialize()` themselves. There is relatively little incentive to do this, as the attacker is unlikely to gain anything (unless someone fails to notice that the vault or strategy was maliciously initialized).

Recommendations: Consider allowing atomic initialization of new proxy instances from the factory methods.

Customer's response: Acknowledged

L-02. The Alt Position May Not Always Lie Within the Main Range

Impact: Low
Probability: Low

Violated Rule: `altSubsetOfMainPosition`

Description: The Beefy protocol deploys liquidity to two ranges: a main range that is intended to remain roughly centered on the current pool price with liquidity provided above and below the current price and an alternate ("alt") position that deploys single-sided liquidity in

whichever token is leftover after deploying balanced liquidity to the main range. The alt position is intended to extend from one tick above or below the current price to the upper or lower tick of the main range, respectively. The `_setAltTicks()` function will fail to set or move the alt range in the edge case that the balance of `token1` is exactly equal to the balance of `token0` multiplied by the price. If this occurs on the very first deposit into a strategy, then the transaction will revert due to division by zero in the `LiquidityAmounts` library (as the upper and lower alt ticks will both be uninitialized, and hence zero). This can be compensated for by making the first deposit slightly unbalanced. If the value of the two token balances are equal on any other invocation of `_setAltTicks()`, then the range will simply not move from where it is already set. This will lead to the alt position collecting fewer fees, as it will either be further from the center of the main range, or it will cross the current tick and receive no allocation (as all attempts to add liquidity to the alt range use a single token only). This can be corrected at any time by a call from a keeper to `rebalance()`, however.

Recommendations: While no action is deemed strictly necessary, it would be safe and effective to check where the alt range is currently located in relation to the main range before deploying liquidity to it, and then adjusting it based on the current main range (*not* the current tick) in `_addLiquidity()` if needed. This would also prevent the edge-case reversion in the event of the first deposit having balanced value.

Customer's response: Acknowledged

L-03. getTokensRequired() might return smaller tokens required with bigger deposits

Impact: Low
Probability: Low

Violated Rule: `depositMonotonicity0 and depositMonotonicity1`

Description: In the `BeefyVaultConcLiq` contract, the function `_getTokensRequired()` returns the amounts of token0 and token1 to be deposited on calling `deposit()` based on the current strategy's balances to optimize the liquidity ratio of the vault.
Our two formal verification rules `depositMonotonicity0` and `depositMonotonicity1` have shown

that given two calls to the `deposit()` function one with a higher and one with a lower amount of assets in one of the tokens might lead to a lower amount of shares in the call with the higher amount of assets. That is, we have counterexamples to the monotonicity of deposits. However, our investigation has shown that the culprit of this unexpected behavior lies in the calculation of `_getTokensRequired()`: the result of `_getTokensRequired()` can get lower with a bigger deposit due to the precision loss occurring during the conversion of token0 to token1 and vice versa. That is, the number of shares issued are correct with regards to the actual amounts deposited. Nevertheless, this might negatively impact user experience as the amount of shares issued from a bigger deposit amount entered by the user might lead to a smaller real deposit than a smaller deposit amount due to asset imbalances and precision loss.
Note that this happens when the balances are somewhat imbalanced already, that is when there is significantly more liquidity in one token than the other.
The following lines are affected:

Unset

```
    if (_bal1 < bal0InBal1) {
           uint256 finalBalanceForAmount1 = _bal1 + _amount1;
           uint256 owedAmount0 = finalBalanceForAmount1 > bal0InBal1
               ? (finalBalanceForAmount1 - bal0InBal1) * PRECISION / _price
               : 0;
           if (owedAmount0 > _amount0) {
               depositAmount0 = _amount0;
               depositAmount1 = _amount1 - ( (owedAmount0 - _amount0) * _price / PRECISION );
           } else {
               depositAmount0 = owedAmount0;
               depositAmount1 = _amount1;
           }
    } else {
           uint256 finalBalanceForAmount0 = bal0InBal1 + ( _amount0 * _price / PRECISION );
           uint256 owedAmount1 = finalBalanceForAmount0 > _bal1
               ? finalBalanceForAmount0 - _bal1
               : 0;
           if (owedAmount1 > _amount1) {
               depositAmount0 = _amount0 - ( (owedAmount1 - _amount1) * PRECISION / _price );
               depositAmount1 = _amount1;
           } else {
               depositAmount0 = _amount0;
               depositAmount1 = owedAmount1;
           }
```

```
    }
```

Recommendation: We propose the following fix to mitigate precision loss during the calculation of the amounts. The new version has been verified against deposit monotonicity properties here. Other vault properties could also be established with the fixed version, see here.

```
Unset
    if (_bal1 < bal0InBal1) {
            uint256 finalBalanceForAmount1 = _bal1 + _amount1;
            uint256 owedAmount0 = finalBalanceForAmount1 * PRECISION > bal0InBal1 * PRECISION
                ? (finalBalanceForAmount1 - bal0InBal1)
                : 0;
            if (owedAmount0 > (_amount0 * _price / PRECISION)) {
                depositAmount0 = _amount0;
                depositAmount1 = _amount1 - (owedAmount0 - (_amount0 * _price / PRECISION));
            } else {
                depositAmount0 = owedAmount0 * PRECISION / _price;
                depositAmount1 = _amount1;
            }
    } else {
            uint256 finalBalanceForAmount0 = bal0InBal1 * PRECISION + _amount0 * _price;
            uint256 owedAmount1 = finalBalanceForAmount0 > _bal1 * PRECISION
                ? finalBalanceForAmount0 - _bal1 * PRECISION
                : 0;
            if (owedAmount1 > _amount1 * PRECISION) {
                depositAmount0 = _amount0 - (owedAmount1 - _amount1 * PRECISION) / _price;
                depositAmount1 = _amount1;
            } else {
                depositAmount0 = _amount0;
                depositAmount1 = (owedAmount1 + PRECISION - 1) / PRECISION;
            }
    }
```

Customer's response: fixed in commit according to proposal

Fix Review: We provide a formal review of our proposed solution against deposit monotonicity properties here and other vault properties here.

L-04. liquidity can get left behind in old positions when strategy moves to a new position

Impact: Low
Probability: Low

Violated Rule: `oldPositionsZeroLiquidityOwedFees`

if someone donates some token0 and/or token 1 to the strategy contract and calls one of `withdraw`, `harvest`, `rebalance` or `setPositionWidth` functions before the first call to the `deposit` function, the donated amount will get deposited as liquidity in the uniswap pool at positions determined by the price at that time. Later, when the 1st `deposit` call happens, if the price has moved since the previous liquidity addition, the `deposit` function will first reset the position based on the new price and then add liquidity to the new position while the liquidity deposited in the previous position continues to be held in that position. This will be recoverable only if the strategy positions at some point get reset to exactly the previous positions. Till then the liquidity deposited in the first step would be locked in the uniswap pool. It's only users who transfer directly to the contract instead of depositing through the `vault` contract, who lose the tokens. If the position of the strategy gets reset to exactly the previous position, the liquidity deposited in the first step would be available to all the shareholders of the `vault` contract.

Customer's response: Acknowledged

Informational Severity Issues

I-01. Events Are Emitted For Path Updates Even When No Update Occurs

Description: In the `StrategyPassiveManagerUniswap` contract, the functions `setLpToken0ToNativePath()` and `setLpToken1ToNativePath()` both only update the path if the provided array is non-empty, but emit an event with the provided argument in any case. For example:

```
Unset
    function setLpToken1ToNativePath(bytes calldata _path) public onlyOwner {
        if (_path.length > 0) {
           // update logic omitted for brevity
        }
        emit SetLpToken1ToNativePath(_path);
    }
```

This could lead to confusion for any consumers of the event if the function is ever called with a zero-length path.

Recommendation: Move the event emissions inside the if statements, or consider reverting if `_path.length == 0`.

Customer's response: fix commit

I-02. Keepers Can Circumvent The Retirement of a Strategy

Description: Only the owner of a strategy can call `retireStrategy()`, which is intended to prevent the strategy from ever being used again (the owner is even set to zero in the function). However, the `unpause()` function can be called by a keeper as well as the owner (as it has the `onlyManager` modifier), and this undoes the "disabling" actions of `retireStrategy()` by unpausing the contract. The vault of the reactivated strategy will be in an inconsistent state, as the assets originally backing the locked shares will have been transferred out of the strategy. There is no clear harm from this sort of reactivation, but it does violate the apparent intent of the code.

Recommendation: Consider preventing `unpause()` from being called by a keeper after a strategy has been retired, for example by reverting in `unpause()` if the owner is zero.

Customer's response: fix commit

certora

Formal Verification

Assumptions and Simplifications

General Assumptions

A. We are using a compositional reasoning approach that performs verification on the contracts individually while assuming lightweight symbolic representations of dependencies.
B. We summarized the Uniswap V3 pools symbolically to prove properties on the strategy contract. More details can be found here.
C. We abstracted the strategy's internal behavior to prove correctness of vault properties. More details can be found here.

Specification files:

The following contracts were formally verified against prior agreed upon specifications:

A. contracts/vault/BeefyVaultConcLiq.sol
B. contracts/strategies/uniswap/StrategyPassiveManagerUniswap.sol

Verification Notations

Formally Verified	The rule is verified for every state of the contract(s), under the assumptions of the scope/requirements in the rule.
Violated	A counter-example exists that violates one of the assertions of the rule.

certora

Formal Verification Properties

BeefyVaultConcLiq.sol

Assumptions

- To verify properties on the vault contract, we assumed a lightweight symbolic version of the strategy contract that provides the expected values of called functions and stores simple balances of underlying assets.

Properties

Rule Name	Description		Link to rule report
depositMonotonicity0	Depositing a larger amount of token0 and the same amount of token1 leads to at least as many shares as a smaller deposit of token0 and an equal deposit of token1. {link to issue page} **Violation reason: Due to the imprecision of rounding in the else-branch of getTokensRequired() (lines 165-177) this property cannot be verified for the current version of the contract.** A verification of the fix is provided here: depositMonotonicity0 fixed	**Violated, see Issue L-03**	Certora Counterexample
depositMonotonicity1	Depositing a larger amount of token1 and the same amount of token0 leads to at least as many shares as a smaller deposit of token1 and an equal deposit of token0. **Violation reason: see above** A verification of the fix is provided here: depositMonotonicity1	**Violated, see Issue L-03**	Certora Counterexample

depositIntegrity	For each underlying token, the amount by which a user's balance decreases and the strategy's underlying assets increase by calling deposit() is the same	**Verified**	Certora Proof
depositedSharesAtLeasePreview	The amount of shares returned by previewDeposit() is the minimum amount of shares minted during deposit().	**Verified**	Certora Proof
withdrawMonotonicity	Withdrawing a larger amount of shares results in at least the same amount of underlying tokens as a smaller withdraw.	**Verified**	Certora Proof
withdrawIntegrity	For each underlying token, the amount by which a user's balance increases by withdraw(), and the strategy's assets decrease is the same.	**Verified**	Certora Proof
previewWithdrawShouldNotRevertAfterDeposit	previewWithdraw() should not revert when calling with the same amount that was just deposited to the vault.	**Verified**	Certora Proof
balanceOfShouldNotRevert	Querying the amount of shares of a user does not revert.	**Verified**	Certora Proof
getTokensRequiredMinAmount	The amounts returned by _getTokensRequired() should be less than or equal to the amounts it's called with. Note that we adjusted the modifier to public to be able to prove the private function used from munged/BeefyVaultConcLiq.sol.	**Verified**	Certora Proof

sharesAndBalanceConsistency	An decrease in a user's shares signals an increase in at least one of the underlying assets and vice versa	**Verified**	Certora Proof
withdrawAfterDeposit	Depositing and immediately withdrawing does not increase both of the user's underlying balances.	**Verified**	Certora Proof
dustFavorsTheHouse	Depositing and immediately withdrawing does not decrease both underlying balances.	**Verified**	Certora Proof
splitWithdrawFavorsTheContract	Splitting withdraws into multiple withdraw calls is not advantageous to a user, that is it does not produce more assets than a single withdraw.	**Verified**	Certora Proof
totalSupplyIsSumOfShares	The sum of balances stored in the vault is equal to the total supply.	**Verified**	Certora Proof
totalSupplyHasMinShares	The vault's total supply is zero or has at least the minimum amount of shares deposited with the first deposit to the dead address.	**Verified**	Certora Proof
totalSupplyHasAtLeastMinShares	The vault's total supply at least covers as many shares as are minted to the dead address.	**Verified**	Certora Proof
vaultSolvency	The vault's total supply is covered by the strategy's underlying assets.	**Verified**	Certora Proof
totalSharesIsZeroWithUnderlyingBalances	The vault's total supply is zero when no deposits have been made to the strategy, i.e. the strategy assets are zero.	**Verified**	Certora Proof

StrategyPassiveManagerUniswap.sol

We assume the following symbolic representation of *Uniswap V3 Concentrated Liquidity Pools* that facilitates formal reasoning without loss of soundness:

For the mint, burn and collect flows, we track the owed amounts for each token with the ghost mappings `owed0` and `owed1`

```
ghost mapping(bytes32 => uint128) owed0
ghost mapping(bytes32 => uint128) owed1
```

We track positions with the ghost mapping `positionOfUser`

```
ghost mapping(bytes32 => mathint) positionOfUser
```

A. `CVLSlot0` summarizes `slot0` where:
   a. sqrtPriceX96 is a ghost mapping that returns an arbitrary but consistent value for each tick
   b. currentTick is a ghost mapping that returns an arbitrary but consistent value for each block timestamp
   c. Everything else is non-deterministic
B. `CVLMint` summarises the `mint` function and does the following:
   a. Uses a fixed ratio of 5/3 and 7/3 multiplied by the liquidity amount to calculate the amount of tokens to be pulled from the LP (strategy contract in this case)
   b. Adds the liquidity amount to `positionOfUser` mapping for the key
   c. Adds an arbitrary amount of fee growth to `owed0` and owed1 if the `lastUpdated` for the key is less than current block time
C. `CVLBurn` summarises the `burn` function and does the following:
   a. Uses a fixed ratio of 5/3 and 7/3 multiplied by the liquidity amount to calculate the amount of tokens to be add to `owed0` and `owed1`
   b. Deducts the liquidity amount from `positionOfUser` mapping for the key
   c. Adds an arbitrary amount of fee growth to `owed0` and `owed1` if the `lastUpdated` for the key is less than current block time
D. `CVLCollect` summarises the `collect` function and it
   a. transfers the requested amount of tokens to the strategy contract and
   b. deducts the amounts from `owed0` and `owed1`

certora

Properties

Rule Name	Description		Link to rule report
oldPositionsZeroLiquidityOwedFees	When ticks bounds are changed, there shouldn't be any liquidity or fees owed left in the old positions {link to issue page}	**Violated**	Certora counter example
altSubsetOfMainPosition	The alternative position is a subset of the main position, that is, upper and lower tick bounds lie within the main position. {link to issue page}	**Violated**	Certora counter example
ticksUninitializedPoolBalancesZero	When initTicks is false, the balancesOfPool should be 0 {link to issue page}	**Violated**	Certora counter example
rebalanceDoesntDecreaseBalance	Strategy balances() should not decrease due to rebalancing, assuming that the main and alt positions are different.	**Verified**	Certora Proof
addingAndRemovingLiquidityDoesntReduceBalance	Calling addLiquidity() and removeLiquidity() in the same block should not reduce the total assets of the strategy.	**Verified**	Certora Proof
feesMonotonicity	Fees cannot decrease unless harvest() is called.	**Verified**	Certora Proof
balanceOnlyDecreaseIfPaused	Strategy assets cannot increase while the strategy is paused.	**Verified**	Certora Proof

lockedProfitsDontIncreaseWithDepositWithdraw	Deposit and withdraw functions should not increase the totalLocked amounts	**Verified**	Certora Proof
whenPausedPoolBalancesMustBe0	When the strategy is paused, the pool balances must be 0, that is no liquidity remains in the pool.	**Verified**	Certora Proof
addingLiquidityReducesAssets	A call to addLiquidity() reduces the internal asset balances of the strategy and increases the pool balances.	**Verified**	Certora Proof
removingLiquidityIncreasesAssets	A call to removeLiquidity() increases the internal asset balances of the strategy and decreases the pool balances.	**Verified**	Certora Proof
- rebalanceShouldRevertNotAuthorized - depositShouldRevertNotAuthorized - withdrawShouldRevertNotAuthorized - beforeActionShouldRevertNotAuthorized - panicShouldRevertNotAuthorized - unpauseShouldRevertNotAuthorized	Authorization: methods must revert when not executed by the correct roles. Note that rebalance() can only be executed by a rebalancer. deposit(), withdraw() and beforeAction() can only be called by the vault. panic() and unpause() can only be executed by the owner() or a keeper().	**Verified**	Certora Proofs

Disclaimer

The Certora Prover takes a contract and a specification as input and formally proves that the contract satisfies the specification in all scenarios. Notably, the guarantees of the Certora Prover are scoped to the provided specification and the Certora Prover does not check any cases not covered by the specification.

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.