CERTORA

Formal Verification Report of Popsicle V3 Optimizer

Summary

This document describes the specification and verification of `PopsicleV3Optimizer` using the Certora Prover. The work was undertaken from September 22, 2021 to October 25, 2021.

The scope of our verification was the Popsicle V3 Optimizer.

The Certora Prover proved the implementation of the `PopsicleV3Optimizer` is correct with respect to the formal rules written by the Popsicle and the Certora teams. During the verification process, the Certora Prover discovered bugs in the code listed in the table below. All issues were promptly corrected, and the fixes were verified to satisfy the specifications up to the limitations of the Certora Prover. The Certora development team is currently handling these limitations. The next section formally defines high level specifications of `PopsicleV3Optimizer`. All the rules are publically available in a public Github https://github.com/Popsicle-Finance/PopsicleV3Optimizer/tree/certora.

Scope

Verification of the main contract: `PopsicleV3Optimizer.sol`

Out of Scope

Verification of all math libraries, optimization algorithms, and interaction with `UniswapV3Pool` were omitted.

Simplified Version of UniswapV3pool

Since including `UniswapV3Pool` code base into the project would essentially mean verification of that code as well, a simplified model of that code was required, and was developed specifically for this project.

The functionality of our simplified model is updating the following global variables:

```
uint256 public liquidity;
uint256 public ratio; // the price
address public immutable override token0;
address public immutable override token1;
uint128 public owed0;
uint128 public owed1;
```

- `SymbolicUniswapV3Pool.sol` is a symbolic representation of UniswapV3 pool , with the following functions:
  `mint(), collect(), burn(), swap(), observe(), tickSpacing(), slot0(), positions()`

- and the following basic assumptions:
  - `liquidity == balanced0 - owed0`
  - The price (`token1 / token0`) fluctuates between three possible values

List of Main Issues Discovered

Severity: High

Issue:	**Theft of yield**
Description:	When calling rebalance, the share price goes up as the imbalanced token is being invested. Thus anyone buying shares in deposit right before rebalance is called and withdrawing right after rebalance, will make no-risk profit at the expense of the investors.
Mitigation/Fix:	Include the imbalanced token amount in share price calculation.

Severity: Medium

Issue:	**Loss of yield**
Description:	When a user withdraws his shares a long time after the last rebalance call, he won't get his part of the imbalanced token.
Mitigation/Fix:	At withdrawal, give the user his part also in the imbalanced token.

Severity: Low

Issue:	**Possible to deposit when tick is out of range**
Description:	When the tick is out of range, users can still deposit, and that's not lucrative.
Mitigation/Fix:	In deposit, check that tick is in range.

Severity: Low

Issue:	**Minor loss of shares**
Description:	When withdrawing, due to significant rounding down, some shares might be lost.
Mitigation/Fix:	Compute back the shares to burn.

Severity: Recommendation

Issue:	**Commingling investors assets with governance assets**
Description:	After collecting fees the governance would invest it together with its clients.
Mitigation/Fix:	Fixed

Severity: Recommendation

Issue:	**Collecting fees as a share of investors income**
Description:	Collecting fees as a share of investors income regardless of the incurred expenses, may raise the issue of conflict of interest.
Mitigation/Fix:	Taken into consideration

Disclaimer

The Certora Prover takes as input a contract and a specification and formally proves that the contract satisfies the specification in all scenarios. Importantly, 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.

We hope that this information is useful, but 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.

Notations

✔️ indicates the rule is formally verified on the latest reviewed commit. We write ✔️* when the rule was verified on a simplified version of the code (or under some assumptions).

❌ indicates the rule was violated under one of the tested versions of the code.

✍ indicates the rule is not yet formally specified.

🔁 indicates the rule is postponed (<due to other issues, low priority>) .

Verification of PopsicleV3 Optimizer

Assumptions/Simplifications:

- Using a simplified UniswapV3 pool
- Simplified math

Functions

```
deposit(uint256 amount0Desired, uint256 amount1Desired, address to) :
uint256,uint256,uint256
```

Deposit the desired amount of token0 and token1 . Returns the actual amounts deposited and the number of shares issued for these amounts.

```
withdraw(uint256 shares, address to) : uint256,uint256
```

Withdrawing with accordance to the number of shares and sent to the recipient “to”. Returns the actual amounts of token0 and token1 recieved.

```
rerange()
```

An operator approved function that sets new position tick ranges.

```
rebalance()
```

An operator approved function that corrects the balance between token0 and token1 in the pool.

```
_earnFees()
```

Collects the fees reserved by UniswapV3 pool.

```
_compoundFees() : uint256,uint256
```

Investing the colected fees in the pool, in addition to existing ones. Returns the acual amount invested.

```
collectProtocolFees(uint256 amount0,uint256 amount1)
```

Governance collecting the desired amounts of token0 and token1 from the accumulated protocol fees.

Properties

1. Rule `zeroCharacteristicOfWithdraw`

Verifies that if withdraw returns `amount0 == 0` and `amount1 == 0` then necessarily `shares == 0`.

```
amount0 == 0 && amount1 == 0 => shares == 0
```

2. Rule `more_shares_more_amounts_to_withdraw` ✔️

Verifies that with larger number of shares one will withdraw a larger amount of assets.

This rule passes only when the following line added to `burnLiquidityShares()`:

`require (share == liquidity * totalSupply/uint256(liquidityInPool));`

```
amount0X,amount1X =  withdraw(sharesX, to);
amount0Y,amount1Y =  withdraw(sharesY, to) at init;
assert amount0X >= amount0Y && amount1X >= amount1Y;
```

3. Rule `totalSupply_vs_positionAmounts` ✔️

Verifies that the total supply before applying an arbitrary public function `f` is greater than the total supply afterwards, implies position liquidity before is greater than position liquidity afterwards minus last compound liquidity.

```
totalSupplyAfter < totalSupplyBefore =>
            posLiquidityAfter - compoundAfter < posLiquidityBefore;
```

4. Rule `protocolFees_state` ✔️

Verifies that balance of governance before applying an arbitrary public function `f` + the change in protocol fees is greater or equal balance of governance after applying `f`.

Excluded functions:

`uniswapV3SwapCallback()` - filtered, meaningless outside of the swap context.

`uniswapV3MintCallback()` - filtered, meaningless outside of the mint context.

`acceptGovernance()` - filtered, breaks the rule when governance changes.

```
balanceGovAfter <= balanceGovBefore + protocolFees_Change;
```

5. Rule `empty_pool_zero_totalSupply` ✔️

Verifies that the pool is empty if and only if the total supply is zero.

Excluded functions:

`uniswapV3MintCallback()` - filtered, meaningless outside of the mint context.

```
(pool.balance0() - pool.owed0() == 0 && pool.balance1() -
pool.owed1() == 0 ) <=> totalSupply() == 0
```

6. Rule `withdraw_amount` ✔️

Verifies that the amount of token0 (token1 is omitted since by symmetry) the user withdraws are no greater than the amount he was supposed to get.

```
amount0 <= amount0_calculated
```

7. Invariant `balance_vs_protocol_liquidity` ✔️

Verifies that if total supply is zero than all the assets of the system is the owned to governance.

`uniswapV3SwapCallback()` - filtered, meaningless outside of the swap context.

`uniswapV3MintCallback()` - filtered, meaningless outside of the mint context.

```
(totalSupply() == 0) => token0.balanceOf(currentContract) ==
protocolFees0()
```

8. Invariant `balance_contract_GE_protocolFees` ✔️

Verifies that balance of the contract is greater than the protocol fees.

Used to establish the initial valid state of the system.

`uniswapV3SwapCallback()` - filtered, meaningless outside of the swap context.

`uniswapV3MintCallback()` - filtered, meaningless outside of the mint context.

```
token0.balanceOf(currentContract) >= protocolFees0()
```

9. Invariant `empty_pool_state` ✔️

Verifies that pool liquidity is zero if and only total supply is zero.

`uniswapV3MintCallback()` - filtered, meaningless outside of the mint context.

```
pool.liquidity() == 0 <=> totalSupply() == 0
```

10. Invariant `empty_pool_state_reverse` ✔️

Verifies that pool liquidity is zero if and only if pool balance minus pool owed is zero.

`uniswapV3MintCallback()` - filtered, meaningless outside of the mint context.

```
pool.liquidity() == 0 <=> (pool.balance0() - pool.owed0() == 0 &&
pool.balance1() - pool.owed1() == 0)
```

11. Invariant `zero_totalSupply_zero_owed` ✔️

Verifies that all assets can be properly withdrawn - if total supply is zero, then no owed assets are left in the pool.

```
totalSupply() == 0 => (pool.owed0() == 0 && pool.owed1() == 0)
```

12. Invariant `pool_balance_vs_owed` ✔️

Verifies that pool balance is greater or equal to pool owed.

```
pool.balance0() >= pool.owed0() && pool.balance1() >= pool.owed1()
```

13. Invariant `zero_pool_balance_zero_owed` ✔️

Verifies that pool balance equals to zero implies pool owed equals to zero.

`uniswapV3MintCallback()` - filtered, meaningless outside of the mint context.

```
(pool.balance0() == 0 => pool.owed0() == 0) &&
(pool.balance1() == 0 => pool.owed1() == 0)
```

14. Invariant `total_vs_protocol_Fees` ✔️

Verifies that total fees are greater than protocol fees.

Used to establish the initial valid state of the system.

```
totalFees0() > protocolFees0() ||
totalFees0() == 0 &&  protocolFees0() == 0
```

15. Invariant `liquidity_GE_poolBalance0` ✔️

Verifies that pool liquidity equals to pool balance minus pool owed.

Used to establish the initial valid state of the system.

`uniswapV3MintCallback()` - filtered, meaningless outside of the mint context.

```
pool.liquidity() == pool.balance0() - pool.owed0()
```