certora

Security Assessment & Formal Verification Final Report

Likwid Protocol

March 2025

Prepared for Likwid.fi

Table of content

Project Summary..4

Project Scope..4

Project Overview..4

Protocol Overview..4

Findings Summary..5

Severity Matrix..5

Detailed Findings..6

Critical Severity Issues..10

C-01 onlyStatus, onlyStatusManager, and onlyPairPoolManager modifiers can be circumvented..............10

C-02 TWAP oracle can be manipulated due to post-state-change observation writes................................11

High Severity Issues..13

H-01 Anyone can spend another user's allowance on LendingPoolManager and PairPoolManager...........13

H-02 If user donates their retain LP tokens, they may DoS the pool..15

H-03 Closing a margin position may be DoS-ed if the underlying tokens are borrowed..........................17

H-04 MarginOracle.observeNow() bypasses TWAP by querying a manipulable last observation...............19

H-05 TruncatedOracle.Observation stores reserves as a snapshot instead of a cumulative value, breaking TWAP logic..20

H-06 PoolStatusManager::_updateInterest accrue wrong ratio for the pair pool... 22

H-07 LendingPoolManager::handleWithdraw uses wrong amount for _burnReturn....................................24

H-08 Exploiter can grief liquidity removal by depositing dust amounts.. 25

Medium Severity Issues..27

M-01 User assets deposited to be lent for pool X may be borrowed from different pool.............................27

M-02 Unsafe unchecked block in PoolStatusManager may result in withdrawing user's liquidity................28

M-03 Users can withdraw 100% of their collateral to avoid any risk... 30

M-04 Updating hooks in PairPoolManager can DoS existing pools...31

M-05 dynamicFee() may exceed 100%, causing denial-of-service in margin operations............................33

Low Severity Issues..36

L-01 LendingPoolManager::handleDeposit uses wrong address as sender in Deposit event......................36

L-02 Incorrect caller in ERC-6909 Transfer events when minting and burning...37

L-03 PairPoolManager::addLiquidity - check msg.value when the currency is address(0)..........................38

L-04 If someone transfers his position to another user, receiver's _marginPositionIds is overridden...........39

L-05 Margin position NFT is not burned when position is closed...40

L-06 MarginLiquidity::addLiquidity mint X2 levelId tokens to address(this)..41

L-07 Follow CEI pattern and implement nonReentrant modifier for the complex functions......................... 42

L-08 MarginChecker::checkValidity is redundant and only waste gas where it is called..............................43

L-09 Incorrect max sliding upper bound calculation when adding liquidity..44

L-10 Opening margin positions for third parties could be griefed...46

L-11 ERC6909.allowance() returns raw values, breaking ERC-6909 specification compliance.................. 48
L-12 Incorrect fee currency reported in Fees event during swaps.. 49
L-13 Native currency refund sent to msg.sender instead of sender...51
Informational Severity Issues... 52
I-01. Redundant zero totalSupply check in PairPoolManager.addLiquidity()... 52
I-02. Position managers in PairPoolManager cannot be revoked.. 52
I-03. Sliding range validation when adding liquidity is non-inclusive.. 52
I-04. Ambiguous margin level naming may lead to confusion.. 53
I-05. Unnecessarily complex logic for calculating reserves and cumulative price in TruncatedOracle.observeSingle()..54
I-06. Incorrect function names: getReverse0() and getReverse1()... 54
Formal Verification...55
Verification Notations...55
General Assumptions and Simplifications.. 55
Formal Verification Properties... 57
PairPoolManager...57
P-01. No Non-Zero Currency Deltas after Locking of PoolManager.. 57
P-02. Invariant: PairPoolManager methods do not decrease rateCumulativeLast values........................... 58
P-03. Integrity of addLiquidity and removeLiquidity..58
LendingPoolManager.. 60
P-04. No Non-Zero Currency Deltas after Locking of PoolManager.. 60
P-05. Integrity of deposit.. 60
PoolStatusManager...62
P-05. Verify getAmounts and dynamicFee summaries.. 62
P-06. Integrity properties PoolStatusManager methods...62
MarginHook.. 64
P-07. Round Trip Swap cannot make profit..64
Disclaimer.. 65
About Certora...65

Project Summary

Project Scope

Project Name	Repository (link)	Latest Commit Hash	Platform
Likwid.fi	https://github.com/likwid-fi/likwid-margin	9c8a04d	EVM

Project Overview

This document describes the specification and verification of **Likwid.fi** using the Certora Prover and manual code review findings. The work was undertaken from **March 17th, 2025** to **April 7th, 2025**.

The Certora Prover demonstrated that the implementation of the Solidity contracts above is correct with respect to the formal rules written by the Certora team. 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 on the following page.

Protocol Overview

Likwid.fi is a decentralized derivatives platform built on Uniswap V4. It enables users to long, short, and trade ERC-20 token pairs without oracles or direct counterparties, using pooled liquidity and collateral-backed positions. Lenders can provide liquidity to the protocol to earn yield from traders and borrowers. Unlike Uniswap V4's concentrated liquidity model, Likwid.fi uses a traditional constant product AMM (x*y=k) to price assets. The protocol leverages Uniswap V4's infrastructure—such as hooks and singleton pools—to implement custom trading logic and dynamic fees while maintaining full decentralization and capital efficiency.

Findings Summary

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

Severity	Discovered	Confirmed
Critical	2	2
High	8	8
Medium	5	5
Low	13	13
Informational	6	6
Total	34	34

Severity Matrix

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
		Likelihood		

Detailed Findings

ID	Title	Severity
C-01	onlyStatus, onlyStatusManager, and onlyPairPoolManager modifiers can be circumvented	Critical
C-02	TWAP oracle can be manipulated due to post-state-change observation writes	Critical
H-01	Anyone can spend another user's allowance on LendingPoolManager and PairPoolManager	High
H-02	If user donates their retain LP tokens, they may DoS the pool	High
H-03	Closing a margin position may be DoS-ed if the the underlying tokens are borrowed	High
H-04	MarginOracle.observeNow() bypasses TWAP by querying a manipulable last observation	High
H-05	TruncatedOracle.Observation stores reserves as a snapshot instead of a cumulative value, breaking TWAP logic	High

H-06	PoolStatusManager::_updateInterest accrue wrong ratio for the pair pool	High
H-07	LendingPoolManager::handleWithdraw uses wrong amount for _burnReturn	High
H-08	Exploiter can grief liquidity removal by depositing dust amounts	High
M-01	User assets deposited to be lent for pool X may be borrowed from different pool	Medium
M-02	Unsafe unchecked block in PoolStatusManager may result in withdrawing user's liquidity	Medium
M-03	Users can withdraw 100% of their collateral to avoid any risk	Medium
M-04	Updating hooks in PairPoolManager can DoS existing pools	Medium
M-05	dynamicFee() may exceed 100%, causing denial-of-service in margin operations	Medium
L-01	LendingPoolManager::handleDeposit uses wrong address as sender in Deposit event	Low

L-02	Incorrect caller in ERC-6909 Transfer events when minting and burning	Low
L-03	PairPoolManager::addLiquidity - check msg.value when the currency is address(0)	Low
L-04	If someone transfers his position to another user, receiver's _marginPositionIds is overridden	Low
L-05	Margin position NFT is not burned when position is closed	Low
L-06	MarginLiquidity::addLiquidity mint X2 levelId tokens to address(this)	Low
L-07	Follow CEI pattern and implement nonReentrant modifier for the complex functions	Low
L-08	MarginChecker::checkValidity is redundant and only waste gas where it is called	Low
L-09	Incorrect max sliding upper bound calculation when adding liquidity	Low
L-10	Opening margin positions for third parties could be griefed	Low

L-11	ERC6909.allowance() returns raw values, breaking ERC-6909 specification compliance	Low
L-12	Incorrect fee currency reported in Fees event during swaps	Low
L-13	Native currency refund sent to msg.sender instead of sender	Low

Critical Severity Issues

C-01 onlyStatus, onlyStatusManager, and onlyPairPoolManager modifiers can be circumvented

Severity: **Critical**	Impact: **High**	Likelihood: **High**
Files: MarginOracle.sol#L26-L29 MirrorTokenManager.sol#L31 MarginLiquidity.sol#L38		

Description: The `MirrorTokenManager` contract includes a `mintInStatus()` function that checks if `IStatusBase(msg.sender).pairPoolManager()` is a whitelisted address. However, since `msg.sender` can be a malicious contract that implements the `IStatusBase` interface and returns a valid `pairPoolManager`, this check can be bypassed. This allows malicious contracts to mint an arbitrary amount of mirror tokens to any address.

A similar issue exists in the `MarginLiquidity.addInterests` function, where the `onlyStatus` modifier is used, and in `MarginOracle.write()` which relies on the `onlyPairPoolManager` modifier.

Exploit Scenario: An attacker can mint unlimited mirror tokens to any address, which can manipulate pool token prices, cause denial of service across the protocol, and more.

Additionally, the attacker can manipulate the `MarginOracle` data for any pool at any time.

Recommendations: Store a trusted caller address as a state variable in each affected contract and use it for verification.

C-02 TWAP oracle can be manipulated due to post-state-change observation writes

Severity: **Critical**	Impact: **High**	Likelihood: **High**
Files: PoolStatusManager.sol#L445		

Description: The protocol's TWAP oracle writes observations **after** reserve-changing operations such as swaps, margin positions, adding/removing liquidity, etc. This is done at the end of the `PoolStatusManager.update()` function which is called after reserve-changing operations in the protocol are executed. This opens the door for price manipulation as a malicious actor can momentarily skew the spot price (e.g., via a flashloan or large trade) which will immediately trigger an observation write that captures the manipulated state.

Unlike the original Uniswap V3 design—which updates oracle observations **before** the first trade in a block to ensure that each price snapshot reflects a period of real market price—the current implementation records observations **after** a reserve change but once per block. This leads to two critical issues:

1. **Manipulable TWAP** An attacker can manipulate the TWAP oracle price used in borrowing, liquidation, or margin calculations.
2. **Broken TWAP math** The cumulative time component is incorrectly attributed to the current (post-manipulation) price rather than the previous one. This results in incorrect average pricing and undermines the time-weighted nature of the oracle.

Exploit Scenario: A malicious actor could use this flaw to manipulate the result of `MarginChecker.getMaxDecrease()`, a function that determines how much of their margin a user is allowed to withdraw. By artificially inflating the TWAP in their favor, the attacker could make their position appear overcollateralized and withdraw the majority (or entirety) of their margin without triggering liquidation. This could directly lead to protocol insolvency.

Recommendations: Write TWAP observations before any state-changing operation occurs, ensuring the recorded price reflects actual market conditions prior to manipulation risk. This will also ensure that the cumulative time component is correctly attributed to the TWAP oracle observations.

High Severity Issues

H-01 Anyone can spend another user's allowance on LendingPoolManager and PairPoolManager		
Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: LendingPoolManager.sol#L223-L224		Violated Property: P-05 depositWithdrawsOthersBalance

Description: The `deposit()` function allows any caller to arbitrarily set both the `sender` and `recipient` parameters:

```
Unset
   function deposit(address sender, address recipient, PoolId poolId, Currency currency,
uint256 amount)
        public
        payable
        returns (uint256 originalAmount)
    {
        uint256 sendAmount = currency.checkAmount(amount);
        bytes memory result =
            poolManager.unlock(abi.encodeCall(this.handleDeposit, (sender, recipient, poolId,
currency, amount)));
        originalAmount = abi.decode(result, (uint256));
        if (msg.value > sendAmount) transferNative(msg.sender, msg.value - sendAmount);
        pairPoolManager.statusManager().updateLendingPoolStatus(poolId);
    }
```

Since `sender` is user-controlled and not validated against `msg.sender`, an attacker can exploit users who have approved an allowance for `LendingPoolManager` (or `PairPoolManager`). They can simply call `deposit()` with the victim as the `sender` and themselves as the `recipient`, transferring tokens from the victim's balance.

This is especially dangerous because approvals and contract interactions are often done in two separate transactions. After the user approves the allowance, but before their own `deposit()` call is mined, an attacker can frontrun and steal their tokens.

Exploit Scenario: A user approves `LendingPoolManager` to spend tokens. Before they call `deposit()`, an attacker frontruns their transaction, specifying the user as `sender` and themselves as `recipient`, stealing the approved tokens.

Recommendations: Make the vulnerable `deposit()` function to be internal and introduce an access controlled function which can specify a depositor (because this is the flow in [PositionMarginManager::modify]()).

H-02 If user donates their retain LP tokens, they may DoS the pool		
Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: MarginLiquidity.sol#L282-L284		Violated Property: P-06 integrityOfTotalSupply

Description: The retain LP token supply is tracked using `balanceOf(poolManager, lPoolId)`. This value can be manipulated if a user transfers their retain-level LP tokens directly to the `poolManager` address. Doing so inflates the tracked retain supply without increasing the actual `totalSupply`.

This can lead to a state where `retainSupply0` or `retainSupply1` becomes larger than `totalSupply`, causing underflows in reserve calculations and breaking the protocol's flow logic. Specifically, the issue propagates through:

`PairPoolManager.setBalances()` → `_updateInterests` → `marginLiquidity.getInterestReserves`

```
Unset
function getInterestReserves(address pairPoolManager, PoolId poolId, PoolStatus memory
status)
    public
    view
    returns (uint256 reserve0, uint256 reserve1)
{
    uint256 uPoolId = _getPoolId(poolId);
    (uint256 totalSupply, uint256 retainSupply0, uint256 retainSupply1) =
_getPoolSupplies(pairPoolManager, uPoolId);
    if (totalSupply > 0) {
        reserve0 = Math.mulDiv(totalSupply - retainSupply0, status.reserve0(), totalSupply);
...
```

Exploit Scenario:

Bob provides liquidity to pool `x` with margin level = 0 and receives 100 LP tokens. At this point, `totalSupply = retainSupply0 = retainSupply1` because none of the liquidity has been lent.

Bob then transfers his 100 LP tokens to `PairPoolManager` , making `retainSupply0 = 200` while `totalSupply` remains 100. Now, every call to `PairPoolManager.setBalances()` will revert due to an underflow, effectively DoSing the pool.

Recommendations: Disallow users from transferring ERC6909 tokens of `MarginLiquidity` directly to the `PairPoolManager` contract

H-03 Closing a margin position may be DoS-ed if the underlying tokens are borrowed

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: PairPoolManager.sol#L454-L455		

Description: The current system design allows margin tokens to be lent out and borrowed by other margin positions. This may result in inability to close a position which is near liquidation. Closing a position is a time sensitive operation and must be reliably available. However, when a user calls `close()`, it leads to the following call chain:

```
Unset
PairPoolManager.release() → handleRelease → LendingPoolManager.realOut()
```

The `realOut()` function does not check if there is enough available balance or if the tokens are currently borrowed. If the required funds are not available due to being lent out, the token transfer will underflow and revert, effectively preventing the user from closing their position.

Note: A coded PoC is available here.

Recommendations: Consider introducing a check similar to the logic used in `handleWithdraw()`. Also, consider adding a user-configurable flag to indicate whether they allow their margin to be lent out.

Unset
```
   function handleWithdraw(address sender, address recipient, PoolId poolId, Currency
currency, uint256 amount)
        external
        selfOnly
    {
        uint256 id = currency.toTokenId(poolId);
        uint256 balance = poolManager.balanceOf(address(this), currency.toId());
        uint256 realAmount = amount;
        if (balance < amount) {
            uint256 mirrorBalance = mirrorTokenManager.balanceOf(address(this), id);
            uint256 exchangeAmount = Math.min(amount - balance, mirrorBalance);
            bool success = pairPoolManager.mirrorInRealOut(poolId, currency, exchangeAmount);
            require(success, "NOT_ENOUGH_RESERVE");
            realAmount = balance + exchangeAmount;
        }
```

H-04 MarginOracle.observeNow() bypasses TWAP by querying a manipulable last observation

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: [MarginOracle.sol#L73]		

Description: The `MarginOracle.observeNow()` function uses `observeSingle(block.timestamp, 0 /** secondsAgo **/, ...)` to fetch the oracle value. However, passing `0` as `secondsAgo` simply returns the current cumulative price and reserves snapshot, rather than a time-weighted average. As a result, `observeNow()` does not use TWAP at all.

This is dangerous because the returned price and reserves reflect only the latest reserve state, which can be easily manipulated by an attacker through a flashloan or a short-term price spike. Since the protocol uses this value for margin-related logic, such as determining collateralization. This introduces an exploit vector where the attacker only needs to manipulate one observation, making the exploit cheap and reliable.

Recommendations: Use a proper TWAP by passing a non-zero `secondsAgo` value (e.g., 1800 for 30 minutes) to `observeSingle()`. This ensures that the returned price and reserves reflects a time-weighted average over a meaningful window, which is significantly harder to manipulate.

H-05 TruncatedOracle.Observation stores reserves as a snapshot instead of a cumulative value, breaking TWAP logic

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: TruncatedOracle.sol#L22-L27 MarginOracle.sol#L62-L80 MarginOracle.sol#L82-L95		

Description: The `Observation` struct used by `TruncatedOracle` stores the `reserves` field as a snapshot of the reserves at a given timestamp, rather than a cumulative value over time. This design choice fundamentally breaks the intended purpose of implementing a TWAP oracle.

A TWAP requires cumulative values (e.g., `tickCumulative` in Uniswap V3) so that an average over a time interval can be calculated by taking the difference between two cumulative observations and dividing by the elapsed time. Without cumulative reserve data, the system cannot determine how long a given price persisted, which invalidates any attempt to compute a time-weighted average.

For example, the current implementation of `MarginOracle.observeNow()` simply returns the latest snapshot of reserves. Since these reserves are not cumulative, there's no way to calculate how long the reserves were at a given level, which makes TWAP calculations impossible. To implement a proper TWAP mechanism, the protocol must follow a design similar to Uniswap's `OracleLibrary.consult()` function. That implementation uses historical cumulative values to compute an average over a specified time window:

```
Unset
averageTick = (tickCumulativeNow - tickCumulativePast) / timeElapsed;
```

A similar mechanism would require reserves to be stored in cumulative form to allow valid averaging logic.

Recommendations: Update the `Observation` struct to store cumulative reserves rather than snapshots. Then, implement time-weighted average calculations in `observeNow()` based on differences between cumulative observations over a specified time window, similar to how Uniswap's `OracleLibrary` works.

H-06 PoolStatusManager::_updateInterest accrue wrong ratio for the pair pool

Severity: **High**	Impact: **Medium**	Likelihood: **High**
Files: PoolStatusManager.sol#L349-L355		

Description: PoolStatusManager::_updateInterest is calculating and updating interest for the lending parties. For the lenders in the pairPoolManager to withdraw interest, we should update accruesRatioX112Of of MarginLiquidity contract. This is done in the end of PoolStatusManager::_updateInterest:

```
Unset
                            if (interestStatus0.pairInterest + interestStatus1.pairInterest >
        0) {
                                marginLiquidity.addInterests(
                                    poolId,
                                    status.reserve0(),
                                    status.reserve1(),
                                    interestStatus0.pairInterest,
                                    interestStatus1.pairInterest
                                );
                            }
```

The problem is that earlier in the same function we update status.mirrorReserve0/1 with the accrued interest, but marginLiquidity.addInterests is internally calculating the new reserves (adding the interest again):

```
Unset

    function _addInterests(
        address pairPoolManager,   PoolId poolId,
        uint256 _reserve0,
        uint256 _reserve1,
        uint256 interest0,
        uint256 interest1
    ) internal returns (uint256 liquidity) {
        uint256 rootKLast = Math.sqrt(_reserve0 * _reserve1);
        uint256 rootK = Math.sqrt((_reserve0 + interest0) * (_reserve1 + interest1));
        if (rootK > rootKLast) {
            uint256 uPoolId = _getPoolId(poolId);
            uint256 _totalSupply = balanceOf(pairPoolManager, uPoolId);
            uint256 numerator = _totalSupply * (rootK - rootKLast);
            uint256 denominator = rootK + rootKLast;
            liquidity = numerator / denominator;
```

This will result in a compromised `accruesRatioX112Of` interest update for pairPool lenders => resulting in smaller interest than intended.

Recommendations: Subtract the already added interest on this func invocation:

```
Unset
        if (interestStatus0.pairInterest + interestStatus1.pairInterest > 0) {
            marginLiquidity.addInterests(
                poolId,
                status.reserve0() - interestStatus0.pairInterest,
                status.reserve1() - interestStatus1.pairInterest,
                interestStatus0.pairInterest,
                interestStatus1.pairInterest
            );
        }
```

H-07 LendingPoolManager::handleWithdraw uses wrong amount for _burnReturn		
Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: [LendingPoolManager.sol#L258-L259]		

Description: In `LendingPoolManager::handleWithdraw` we pass the real amount of tokens we want to withdraw. There is an `if` branch, which override provided amount with a lower value, if currently the contract don't have the underlying tokens liquid:

```JavaScript
...
        uint256 realAmount = amount;
        if (balance < amount) {
            uint256 mirrorBalance = mirrorTokenManager.balanceOf(address(this), id);
            uint256 exchangeAmount = Math.min(amount - balance, mirrorBalance);
            bool success = pairPoolManager.mirrorInRealOut(poolId, currency, exchangeAmount);
            require(success, "NOT_ENOUGH_RESERVE");
            realAmount = balance + exchangeAmount;
        }
        currency.settle(poolManager, address(this), realAmount, true);
        currency.take(poolManager, recipient, realAmount, false);
        uint256 originalAmount = _burnReturn(sender, id, amount);
...
```

The problem is that when we override `amount` (realAmount), we still pass user provided `amount` to `_burnReturn` , which would burn shares worth more underlying tokens than needed.

This scenario is possible because currently the protocol allows deposits for one `poolId` to be borrowed from another `poolId` as long as the borrowed currency matches.

Recommendations: Provide `realAmount` to `_burnReturn` function.

H-08 Exploiter can grief liquidity removal by depositing dust amounts		
Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: [MarginLiquidity.sol#L155]()		

Description: The current design of the pool doesn't allow to remove liquidity for a period of 30 seconds after some liquidity provision for the same `poolId`. This can easily be exploited by a malicious actor and grief someone who wants to remove his liquidity:

```
Unset
{Code Snipped}

       function addLiquidity(address receiver, uint256 id, uint8 level, uint256 amount)
        external
        onlyPoolManager
        returns (uint256 liquidity)
    {
        datetimeStore[id] = uint32(block.timestamp % 2 ** 32);
        ...

    function removeLiquidity(address sender, uint256 id, uint8 level, uint256 amount)
        external
        onlyPoolManager
        returns (uint256 liquidity)
    {
        if (datetimeStore[id].getTimeElapsed() < 30) {
            revert NotAllowed();
        }
        ...
```

An exploiter can indefinitely DoS liquidity removal if he calls `addLiquidity` every 30 seconds with dust amounts, which results in blocked funds for the victims.

Recommendations: Make `datetimeStore[id]` per user (caller)

Medium Severity Issues

M-01 User assets deposited to be lent for pool X may be borrowed from different pool		
Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: [LendingPoolManager.sol#L176-L182]()		

Description: When a user is lending assets, he should call `LendingPoolManager::deposit` with corresponding `PoolId poolId` and `Currency currency`. For example, a user may want to lend only to the most stable and liquid pool X, which is `{eth : usdc}`. But his deposited `eth` may be borrowed from margin position from the low liquid pool Y `{eth: memeCoin}`. `mirrorInRealOut` function, which is called from `PairPoolManager` when someone opens a margin position for pool don't take into account the `poolId` provided, but only the currency:

```
Unset
    function mirrorInRealOut(PoolId poolId, Currency currency, uint256 amount)
        external
        onlyPairManager
        returns (uint256 exchangeAmount)
    {
        uint256 id = currency.toId();
        uint256 balance = poolManager.balanceOf(address(this), id);
        exchangeAmount = Math.min(balance, amount);
```

Impact is that victims are exposed to the risk of all pools, which use that currency.

Recommendations: In `mirrorInRealOut` first check if we have deposits for that pool.

M-02 Unsafe unchecked block in PoolStatusManager may result in withdrawing user's liquidity		
Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: [PoolStatusManager.sol#L514-L518]()	Status: Waiting for customer response	

Description: When protocol owners want to collect fees, they call `collectProtocolFees` which accept `Currency currency` and `uint256 amount`:

```
Unset
   function collectProtocolFees(Currency currency, uint256 amount)
        external
        onlyPoolManager
        returns (uint256 amountCollected)
    {
        amountCollected = (amount == 0) ? protocolFeesAccrued[currency] : amount;
        unchecked {
            protocolFeesAccrued[currency] -= amountCollected;
        }
    }
```

If `amount = 0` we will use `protocolFeesAccrued[currency]`. But if the caller has provided value `> protocolFeesAccrued[currency]` value will silently underflow to a value close to `type(uint256).max`. This may result in a DoS of the pool when new protocol fees are accrued and hit the overflow limit:

```
Unset
   function updateProtocolFees(Currency currency, uint256 amount)
        external
        onlyPoolManager
        returns (uint256 restAmount)
    {
        uint256 protocolFees = marginFees.getProtocolFeeAmount(amount);
        protocolFeesAccrued[currency] += protocolFees;
        restAmount = amount - protocolFees;
    }
```

Recommendations: Remove `unchecked` block

M-03 Users can withdraw 100% of their collateral to avoid any risk		
Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: [MarginChecker.sol#L265-L290]()		

Description: The protocol implements a safety mechanism to check for a `minMarginLevel` when opening and modifying a position.

The initial `minMarginLevel` is set to 117%, which means that `totalMargin(leveraged margin) + marginAmount (user collateral)` should be `> 117% borrowedAmount` (when normalized to one asset).

The problem is that if the user leverage position is large enough, he can withdraw 100% of his collateral (marginAmount) and pass all checks (`_position.marginTotal is worth > 117% positon.borrowAmount`).

This drastically affects the stability of the protocol because users can take risk-free loans with leverage.

Recommendations: Implement a minimum ratio enforcement between `marginAmount` and `totalMargin`.

Customer's response: Waiting for customer response

Fix Review:

M-04 Updating hooks in PairPoolManager can DoS existing pools		
Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: PairPoolManager.sol#L297		

Description: The `MarginHook` contract is initialized with a `pairPoolManager` upon construction, making it immutable. Additionally, when a new pool is created via Uniswap V4's `PoolManager.initialize()` and a hook is configured for it, the hook is also immutable and cannot be updated.

However, the `PairPoolManager` contract allows updating its `hooks` contract via the `setHooks()` function. This introduces a risk where updating the `hooks` contract in `PairPoolManager` will render all existing pools that rely on it non-functional. This is due to the `onlyHooks` modifier, which enforces access control over critical functions such as:

- `initialize()`

- `setBalances()`

- `updateBalances()`

- `swap()`

Since the old hook remains assigned to existing pools, but `PairPoolManager` will only recognize the new one, all function calls from existing pools will fail, leading to a Denial of Service (DoS) for all users interacting with those pools.

Recommendations: To prevent this issue, restrict any updates to the `hooks` contract once it has been set. Modify the `setHooks()` function as follows to ensure it can only be set once:

```
Unset
function setHooks(IHooks _hooks) external onlyOwner {
    if (address(hooks) != address(0)) revert HookAlreadySet();
    hooks = _hooks;
}
```

This ensures that once the hooks contract is configured, it cannot be updated, preventing a DoS scenario for existing pools.

M-05 dynamicFee() may exceed 100%, causing denial-of-service in margin operations		
Severity: **Medium**	Impact: **High**	Likelihood: **Medium**
Files: MarginFees.sol#L100-L105		

Description: The `dynamicFee()` function includes logic for applying a one-block penalty fee, which multiplies the base fee by 20 when multiple margin-related actions occur in the same block:

```
Unset
uint256 timeElapsed = status.marginTimestampLast.getTimeElapsed();
// ...
if (timeElapsed == 0) {
    uint24 oneBlockFee = 20 * status.key.fee;
    if (oneBlockFee > _fee) {
        _fee = oneBlockFee;
    }
}
```

If `status.key.fee >= 50_000` (i.e., 5%), this logic will produce a value greater than the defined `MAX_LP_FEE` of `1_000_000` (100%). This violates the assumption that fees are always below 100% and leads to an underflow in `FeeLibrary.attachFrom()`:

```
Unset
function attachFrom(uint24 fee, uint256 amount) internal pure returns (uint256 amountWithFee)
{
    uint256 ratio = PerLibrary.ONE_MILLION - fee; // underflows if fee >= 1_000_000
    amountWithFee = Math.mulDiv(amount, PerLibrary.ONE_MILLION, ratio);
}
```

This causes the transaction to revert in any context where `dynamicFee()` is used—including sensitive operations like margin position closure or repayment. An attacker can frontrun a user's transaction and perform a trivial margin action (e.g., opening a small position in the same pool). This sets the pool's `marginTimestampLast` to the current block timestamp and causes the fee multiplier to activate. If the configured base fee is 5% or more, this will cause the user's transaction to revert, potentially blocking them from repaying or closing a position in time and exposing them to liquidation.

Exploit Scenario:

1. Alice has an open margin position and is close to liquidation.
2. She submits a transaction to repay or close her position.
3. Bob frontruns Alice's transaction with a minimal margin action in the same pool.
4. This triggers the one-block penalty fee for Alice and sets `dynamicFee = 20 * status.key.fee`.
5. If `status.key.fee >= 50_000`, `dynamicFee()` returns a value `>= 1_000_000`, causing an underflow in `FeeLibrary.attachFrom()` and reverting Alice's transaction.
6. Alice is unable to close her position and may be liquidated.

Recommendations: Move the following overflow-protection logic to the end of the `dynamicFee`() function, so it applies to all scenarios::

```
Unset
function dynamicFee(address _poolManager, PoolStatus memory status) public view returns
(uint24 _fee) {
    // ...
    if (dFee >= PerLibrary.ONE_MILLION) {
        _fee = uint24(PerLibrary.ONE_MILLION) - 1;
    } else {
        _fee = uint24(dFee);
    }
}
```

This ensures that the dynamic fee never exceeds 100%, even in penalty scenarios and prevents downstream underflow reverts.

Low Severity Issues

L-01 LendingPoolManager::handleDeposit uses wrong address as sender in Deposit event		
Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: LendingPoolManager.sol#L234 LendingPoolManager.sol#L259		

Description: `handleDeposit` will always be called by `address(this)` (LendingPoolManager itself), because it is called in `_unlockCallback` function which means that the `msg.sender` of `handleDeposit` will always be `LendingPoolManager` address.

At the end of the function the `Deposit` event is emitted and `msg.sender` is provided to the `sender` event param, which is wrong because the sender that initiated the deposit is different.

Note: The same problem exists in `handleWithdraw` function

Recommendations: Pass `sender` address in the corresponding event in the functions `handleDeposit` and `handleWithdraw`

L-02 Incorrect caller in ERC-6909 Transfer events when minting and burning

Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: [ERC6909Accrues.sol#L92-L112]()		

Description: The `_mint()` and `_burn()` functions in `ERC6909Accrues` emit `Transfer` events every time a token is minted or burned as per the ERC-6909 specification. However, in the current implementation these functions are not directly called by the user but instead by other contracts such as `PairPoolManager`, `PoolStatusManager` etc. But the implementation of these functions uses `msg.sender` as the `caller` regardless of the context. This leads to incorrect `caller` being logged as part of the `Transfer` events emitted during minting and burning tokens as the calling contract is logged as the caller instead of the user who mints or burns the tokens.

Recommendations: Add additional param `originalCaller` to the functions

L-03 PairPoolManager::addLiquidity - check msg.value when the currency is address(0)		
Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: [PairPoolManager.sol#L235-L236]()		

Description: If the pool `currency0` is native eth (address(0)) and there are idle funds in `PairPoolManager` contract, msg.sender can use them to provide liquidity because we the function doesn't check `msg.value`

Recommendations: Check if `currency0 == address(0)`. If so - check that `params.amount0 == msg.value`

certora

L-04 If someone transfers his position to another user, receiver's _marginPositionIds is overridden

Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: [MarginPositionManager.sol#L126-L131]()		

Description: `_marginPositionIds/_borrowPositionIds` should store a position for {poolId, marginForOne, userAddress}. An invariant is that the user may have only one position per those params and he passes the same params to the margin, it will fetch his existing position and modify it. This invariant is violated when the NFT of a position is sent to a user who has active position for the same margin params.

If user A has opened a margin position with keys = {poolId, marginForOne, userAddress} and user B has NFTfor the same pool and margin token, and transfer it to user A, corresponding `_marginPositionIds/_borrowPositionIds` for the receiver (user A) will be overridden and will now point to the received NFT, while user A has another position for the same params.

Now `getPositionId` will return partially wrong data.

Recommendations: {Suggest a rough idea of how to solve the issue}

L-05 Margin position NFT is not burned when position is closed		
Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: MarginPositionManager.sol#L110		

Description: When a margin position is closed, repaid, or liquidated, the corresponding NFT that represents the position remains in existence and is not burned. While internal accounting within the protocol correctly tracks the status of the position, retaining the NFT after it has become inactive can lead to unintended consequences in composability scenarios.

For instance, if a third-party protocol integrates this NFT for collateralization or access control without verifying whether the position is still active, a user could potentially misuse a closed or liquidated NFT to gain unintended benefits. This becomes more problematic if such integrations rely solely on `ownerOf()` checks rather than explicitly querying position status.

This design choice expands the protocol's external attack surface and may result in integration bugs in downstream protocols.

Recommendations: Burn the NFT when the associated position is closed, repaid, or liquidated by calling `_burn(positionId)` in `MarginPositionManager._burnPosition()`.

L-06 MarginLiquidity::addLiquidity mint X2 levelId tokens to address(this)

Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: [MarginLiquidity.sol#L215-L217]()		

Description: `MarginLiquidity` extends `ERC6909Accrues` , which `_mint` function always mints `amount` to `balanceOriginal[address(this)][id]` apart from original receiver.

When calling `addLiquidity` in `MarginLiquidity` contract, we mint the corresponding `amount` for given `levelId` to both `receiver` and `poolManager` . This will result in minting X2 tokens to `MarginLiquidity`(address(this)) for that id inside `_mint`. If `MarginLiquidity::balanceOf(MarginLiquidity)` is used, it will return a wrong and confusing value.

Recommendations: Consider refactoring the code in a clear format for tracking `totalSupply` per id in `MarginLiquidity`

L-07 Follow CEI pattern and implement nonReentrant modifier for the complex functions		
Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: MarginPositionManager.sol#L478-L479		

Description: CEI pattern is violated in many places in the code, which may result in complex exploits using unupdated state vars.

Also consider adding a `nonReentrant` modifier to complex functions such as `liquidateCall`, `liquidateBurn`.

Recommendations: Implement CEI pattern where it is possible.

L-08 MarginChecker::checkValidity is redundant and only waste gas where it is called		
Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: [MarginChecker.sol#L78-L81]()		

Description: `MarginChecker::checkValidity` is called in `MarginPositionManager` and will only waste gas for the external call made, but it is useless because always return true:

```
Unset
    function liquidateCall(uint256 positionId) external payable returns (uint256 profit) {
        require(checker.checkValidity(msg.sender, positionId), "AUTH_ERROR");
}
```

Recommendations: Remove `checker.checkValidity`

L-09 Incorrect max sliding upper bound calculation when adding liquidity		
Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: PoolStatusLibrary.sol#L76		

Description: When computing the maximum sliding upper bound for `amount1` during liquidity addition, the implementation incorrectly applies input-style fee logic, leading to a slight overestimation of the upper bound.

In `PoolStatusLibrary.computeLiquidity()`, the following line is used:

```
Unset
(uint256 amount1Lower, uint256 amount1Upper) = maxSliding.bound(amount1Exactly);
```

This delegates to `FeeLibrary.bound()`, which uses `FeeLibrary.attachFrom()` to compute the upper bound:

```
Unset
function attachFrom(uint24 fee, uint256 amount) internal pure returns (uint256 amountWithFee)
{
    uint256 ratio = PerLibrary.ONE_MILLION - fee;
    amountWithFee = Math.mulDiv(amount, PerLibrary.ONE_MILLION, ratio);
}
```

This calculation is appropriate when determining how much input is needed to receive a specific output (i.e., *grossing up* by fee), but inappropriate for sliding bounds. Sliding bounds should apply the fee as a simple percentage change.

For example, if `amount1Exactly = 1_000_000` and `maxSiding = 0.5%`, the expected upper bound is:

$$1,000,000 \times (1 + 0.005) = 1,005,000$$

But the current implementation computes:

$$1,000,000 \div (1 - 0.005) \approx 1,005,025.13$$

This results in a slightly inflated upper bound due to incorrect base assumptions. While unlikely to cause critical issues, it may reduce the precision of bounds validation or open up small inefficiencies.

Recommendations: Avoid reusing `attachFrom()` for non-fee-related calculations. Instead, calculate the upper bound using:

```
Unset
uint256 upperBound = Math.mulDiv(amount, PerLibrary.ONE_MILLION + sliding,
PerLibrary.ONE_MILLION);
```

L-10 Opening margin positions for third parties could be griefed

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: MarginPositionManager.sol #L201		

Description: The `margin()` function in `MarginPositionManager` is used for both opening new margin positions and modifying existing ones. When opening a position with `params.recipient` set to a third party (e.g. Alice opening a position for Bob), a griefing attack vector is possible.

An attacker (e.g. Charlie) can frontrun Alice's transaction and open a dust-sized margin position (e.g. 1 wei) for the same recipient (Bob). When Alice's transaction executes, the following check fails:

```
Unset
if (positionId == 0) {
    // create new position
    // ...
} else {
    require(ownerOf(positionId) == msg.sender, "AUTH_ERROR"); // @audit this will revert
}
```

Since a position already exists for Bob, `positionId` is non-zero, and Alice is not the owner — her transaction reverts. This allows any actor to grief users attempting to open margin positions for others.

Recommendations: Split the `margin()` function into two explicit functions: `openMarginPosition()` and `modifyMarginPosition()`. This separation would eliminate ambiguity between creation and modification paths and prevent this type of griefing attack vector.

L-11 ERC6909.allowance() returns raw values, breaking ERC-6909 specification compliance		
Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: ERC6909Accrues.sol#L19		

Description: The `ERC6909Accrues` implementation exposes scaled values via external functions such as `balanceOf()` and `transfer()`, applying the `accruesRatioX112Of(id)` multiplier to raw internal values. However, the `allowance()` function returns the raw, unscaled value as stored in `_allowances`, without applying the scaling factor.

According to the ERC-6909 specification, the `allowance()` function returns:

> The total `amount` of a token `id` that a `spender` is permitted to transfer on behalf of an `owner`.

By returning the raw internal value, the current implementation of `ERC6909Accrues` breaks compliance with the specification. From the perspective of external users and integrators — who only interact with scaled values elsewhere — this introduces a unit mismatch and can lead to subtle bugs in wallets or dApps that rely on `allowance()` for UI or transaction logic.

Recommendations: To comply with ERC-6909 and ensure unit consistency, implement an `allowance()` function which would return scaled values, applying the same `accruesRatioX112Of(id)` transformation used in `balanceOf()`.

L-12 Incorrect fee currency reported in Fees event during swaps		
Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: PairPoolManager.sol#L190		

Description: The `Fees` event is emitted using `specified` as the fee currency:

```
Unset
    function liquidateCall(uint256 positionId) external payable returns (uint256 profit) {
        require(checker.checkValidity(msg.sender, positionId), "AUTH_ERROR");
}
```

However, this is inaccurate. The protocol always deducts fees from the input token, which may not match the `specified` currency.

In swap operations:

- If `zeroForOne` is `true`, the input is `key.currency0` and fees are taken in `currency0`
- If `zeroForOne` is false, the input is `key.currency1` and fees are taken in `currency1`

Using `specified` as the fee currency can mislead downstream systems or analytics tools that rely on accurate event logs to track fee flows per token.

Recommendations: Update the event emission to use the correct input token as the fee currency:

```
Unset
address feeCurrency = zeroForOne ? key.currency0 : key.currency1;
emit Fees(key.toId(), feeCurrency, sender, uint8(FeeType.SWAP), feeAmount);
```

certora

L-13 Native currency refund sent to msg.sender instead of sender		
Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: [LendingPoolManager.sol#L204-L215]()		

Description: The `deposit()` function allows an external operator to deposit funds on behalf of a user by specifying a different `sender` address. However, when excess native currency (e.g. ETH) is sent, the refund is transferred to `msg.sender` instead of `sender`:

```
Unset
if (msg.value > sendAmount) transferNative(msg.sender, msg.value - sendAmount);
```

This design assumes that the operator (i.e. `msg.sender`) will properly handle refunding the `sender` but this cannot be guaranteed and may result in the `sender` losing excess funds. This is especially relevant in cases where `msg.sender != sender` which is a supported use case in this function.

Recommendations: Consider refunding the excess native currency to the `sender` address or adding an additional parameter to explicitly specify the address that must be refunded.

Informational Severity Issues

I-01. Redundant zero totalSupply check in PairPoolManager.addLiquidity()

Description: The `PairPoolManager.addLiquidity()` function checks if the liquidity `_totalSupply` is `0`. This check is redundant since the `PoolStatusLibrary.computeLiquidity()` function that is used already covers this scenario. Thus, the check in `addLiquidity()` is redundant and leads to an unnecessarily complex implementation.

Recommendation: Remove the `_totalSupply` zero check in `PairPoolManager.addLiquidity()`.

I-02. Position managers in PairPoolManager cannot be revoked

Description: The `PairPoolManager` allows the contract owner to whitelist new position managers via `PairPoolManager.addPositionManager()`. However the contract does not allow removing old/inactive/malicious position managers.

Recommendation: Add `removePosition()` manager to `PairPoolManager` to allow the contract owner to remove inactive/legacy/malicious position managers.

I-03. Sliding range validation when adding liquidity is non-inclusive

Description: The sliding range validation in `PoolStatusLibrary.computeLiquidity()` when adding liquidity is non-inclusive:

```
Unset
require(amount1 > amount1Lower && amount1 < amount1Upper, "OUT_OF_RANGE");
```

This means that if `amount1` is equal to the lower or upper bound, the operation will fail.

Recommendation: Make the sliding range validation inclusive:

```
Unset
require(amount1 >= amount1Lower && amount1 =< amount1Upper, "OUT_OF_RANGE");
```

I-04. Ambiguous margin level naming may lead to confusion

Description: The current naming of margin levels such as `NO_MARGIN`, `ONE_MARGIN`, and `ZERO_MARGIN` is ambiguous and potentially misleading. These names do not clearly communicate whether they refer to the borrowable token or the retained token, and may be misinterpreted as referring to collateral rather than borrow permissions. This could result in implementation errors or incorrect assumptions when using or integrating with the protocol.

Recommendation: Rename the constants to more intuitive and self-explanatory identifiers, such as:

- `RETAIN_BOTH` for no margin on either token
- `BORROW_TOKEN0` for allowing only `token0` borrowing
- `BORROW_TOKEN1` for allowing only `token1` borrowing
- `BORROW_BOTH` for allowing borrowing of both tokens

This naming better reflects the intent and behavior of each level.

I-05. Unnecessarily complex logic for calculating reserves and cumulative price in TruncatedOracle.observeSingle()

Description: The `observeSingle()` function uses overly complex logic when the target timestamp falls between two observations. It manually interpolates `reserve1` using conditional arithmetic and then derives `reserve0` from the difference in `price1CumulativeLast` values. This mix of reserve-based and price-based math makes the code harder to understand and maintain. While the logic may work as intended, it could be simplified significantly without changing functionality.

Recommendation: Use standard linear interpolation for both `reserve0` and `reserve1` directly as well as `price1CumulativeLast`. This would make the code easier to read, audit, and reason about.

I-06. Incorrect function names: getReverse0() and getReverse1()

Description: The `PriceMath` library contains two functions named `getReverse0()` and `getReverse1()`. These names are incorrect and intended to be `getReserve0()` and `getReserve1()`, matching the terminology used throughout the rest of the codebase.

Recommendation: Rename `getReverse0()` and `getReverse1()` to `getReserve0()` and `getReserve1()` for clarity and consistency.

Formal Verification

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.
Formally Verified After Fix	The rule was violated due to an issue in the code and was successfully verified after fixing the issue
Violated	A counter-example exists that violates one of the assertions of the rule.

General Assumptions and Simplifications

1. Prover Configuration:
   - We use Solidity version 0.8.26
   - Loops are inherently difficult for formal verification. We handle loops by unrolling them a specific number of times. For the LikwidFi protocol, all loops are unrolled at most 2 times.
2. Code changes:
   - We added a `getPoolKey` method to the `PoolStatusManager` contract, it allows extracting `PoolKey` given a `PoolID`.
3. Mocks:
   - We substitute the Uniswap implementation of the `IPoolManager` contract with Certora's implementation. Certora's implementation has been formally verified independently of the current project, hence it is safely used.
   - Similarly, we substitute OpenZeppelin's implementation of `IERC20` with Certora's formally verified CVL-based `CVLERC20` implementation.

- We constrain `PoolStatusManager` additionally such that at most two distinct poolIds are accessed in the `PoolStatusManager`.`statusStore` storage.

4. Code Summarization:
   - We summarize `PoolStatusManager`.`getAmountIn` and `PoolStatusManager`.`getAmountsOut` with simpler functions in order to reduce nonlinear arithmetic. The summaries are proven correct as part of property P-05.
   - Similarly, we substitute math functions `mulDiv`, `average` and `sqrt` with summaries that use standard axiomatizations of these functions.
   - All rules in this project check properties over a single block of transaction. Since interest can only be accumulated across blocks, we ignore the interest calculation. In particular, we assume that `PoolStatusManager`.`_updateInterest0`, `PoolStatusManager`.`_updateInterest1`, and `PoolStatusManager`.`_updateInterests` have no effect.
   - Finally, we summarize certain functions to return non-deterministic values. This results in over-approximation of program behaviors, meaning if a rule can be verified assuming nondeterministic values, then the rule holds true for the original function too. These functions are:
     + `MarginFees`.`getBorrowRateCumulativeLast`
     + `MarginFees`.`getBorrowMaxAmount`
     + `MarginFees`.`getMarginBorrow`
     + `MarginOracleReader`.`observeNow`
     + `MarginOracleWrite`.`write`
     + `MarginLiquidity`.`getInterestReserves`
     + `MarginLiquidity`.`getMarginReserves`
     + `MarginLiquidity`.`getInterestReserves`

Formal Verification Properties

PairPoolManager

P-01. No Non-Zero Currency Deltas after Locking of PoolManager	
Status: Verified after fix	The PoolManager must be locked after the PairPoolManager method. If the method results in non-zero deltas, then it reverts, potential for DOSing the protocol. The following rules ensure this does not happen.

Rule Name	Status	Description	Link to rule report
addLiquidityEndsWithZeroVirtualAccounting	Verified after fix	*This rule verifies that virtual accounting by the PairPoolManager.addLiquidity method is zeroed-out at the end.*	Report
removeLiquidityEndsWithZeroVirtualAccounting	Verified	*This rule verifies that virtual accounting by the PairPoolManager.removeLiquidity method is zeroed-out at the end.*	
releaseEndsWithZeroVirtualAccounting	Verified	*This rule verifies that virtual accounting by the PairPoolManager.release method is zeroed-out at the end.*	
collectProtocolFeesEndsWithZeroVirtualAccounting	Verified	*This rule verifies that virtual accounting by the PairPoolManager.collectProtocolFees method is zeroed-out at the end.*	
swapMirrorEndsWithZeroVirtualAccounting	Verified	*This rule verifies that virtual accounting by the PairPoolManager.swapMirror method is zeroed-out at the end.*	

marginEndsWithZeroVirtualAccounting	Verified	*This rule verifies that virtual accounting by the PairPoolManager.margin method is zeroed-out at the end.*	

P-02. Invariant: PairPoolManager methods do not decrease rateCumulativeLast values

Status: Verified	The protocol stores the cumulative interest rate as rateCumulativeLast. Since these values are cumulative, all methods must ensure these values do not decrease.

Rule Name	Status	Description	Link to rule report
rateCumulativeCannotDecrease	Verified	*This rule verifies that rateCumulativeLast values cannot be decreased by methods in the PairPoolManager contract.*	[Report]()

P-03. Integrity of addLiquidity and removeLiquidity

Status: Verified	

Rule Name	Status	Description	Link to rule report
addingLiquidityHasZeroEffectOnOtherPools	Verified	*This rule verifies that the addLiquidity method does not affect the getAmountIn values of other pools.*	[Report]()
removingLiquidityHasZeroEffectOnOtherPools	Verified	*This rule verifies that the removeLiquidity method does not affect the getAmountIn values of other pools.*	[Report]()

integrityOfAddLiquidity	Verified	*This rule verifies that addLiquidity does not decrease the real reserves and does not change mirror reserves.*	[Report]()
integrityOfRemoveLiquidity	Verified	*This rule verifies that removeLiquidity does not increase the real reserves and does not change mirror reserves.*	[Report]()

LendingPoolManager

P-04. No Non-Zero Currency Deltas after Locking of PoolManager	
Status: Verified	The PoolManager must be locked after the LendingPoolManager method. If the method results in Non-zero deltas, then it reverts, potentially DOSing the protocol. The following rules ensure this does not happen.

Rule Name	Status	Description	Link to rule report
depositEndsWithZeroVirtualAccounting	Verified	*This rule verifies that virtual accounting by the LendingPoolManager.deposit method is zeroed-out at the end.*	[Report]
withdrawEndsWithZeroVirtualAccounting	Verified	*This rule verifies that virtual accounting by the LendingPoolManager.withdraw method is zeroed-out at the end.*	
balanceMirrorEndsWithZeroVirtualAccounting	Verified	*This rule verifies that virtual accounting by the LendingPoolManager.balanceMirror method is zeroed-out at the end.*	

P-05. Integrity of deposit	
Status: Violated	

Rule Name	Status	Description	Link to rule report

depositWithdrawsOthersBalance	Violated	*This rule verifies that the LendingPoolManager.deposit method does not change the balance of any user other than the sender and the receiver. The rule is violated due to H-01.*	Report

PoolStatusManager

P-05. Verify getAmounts and dynamicFee summaries	
Status: Verified	We summarize dynamicFee, getAmountIn and getAmountOut in order to simplify the arithmetic. The rules below verify that the summary is correct for the original code of above functions.

Rule Name	Status	Description	Link to rule report
checkAxioms_dynamicFee	Verified	*This rule verifies the summaries for dynamicFee function.*	[Report]()
checkFeeAxioms_getAmountIn	Verified	*This rule verifies the Fee axioms for the getAmountIn function.*	
checkFeeAxioms_getAmountOut	Verified	*This rule verifies the Fee axioms for the getAmountOut function. For this rule, we assume that status.key.fee < MAX_FEE_UNITS() / 20.*	
checkAmountOutBoundAxiom	Verified	*This rule verifies the Bound axioms for the getAmountIn function.*	
checkAmountInBoundAxiom	Verified	*This rule verifies the Bound axioms for the getAmountOut function.*	

P-06. Integrity properties PoolStatusManager methods	
Status: Violated	

Rule Name	Status	Description	Link to rule report
integrityOfSetAndUpdateBalances	Verified after fix	*This rule verifies that the setBalance() followed updateBalance() does not change the balances.*	*Report*
integrityOfTotalSupply	Violated	*This rule demonstrates how retainSupply can grow bigger than the totalSupply as described in H-02.*	*Report*
StatusStorePoolKeyMatch	Verified	*This invariant verifies that for every initialized pool stored in PoolStatusManager.statusStore, its poolId and poolKey are consistent with each other.*	*Report*

P-07. Round Trip Swap cannot make profit	
Status: Verified	

Rule Name	Status	Description	Link to rule report
roundTripSwapCannotMakeProfit	Verified	*This rule verifies that performing a swap and then a reverse swap in the same transaction block cannot result in a profit.*	[Report]()

Disclaimer

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.

The review was conducted over a limited timeframe and may not have uncovered all relevant issues or vulnerabilities. This report does not include a review of the fixes.

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.