certora

Security Assessment & Formal Verification Draft Report

Angstrom

June 2025

Prepared for Sorella Labs

certora

Table of Contents

Project Summary

Project Scope

Project Name	Repository (link)	Commit Hash	Platform
Angstrom	https://github.com/SorellaLabs/angstrom	a847640	EVM

Project Overview

This document describes the specification and verification of **Angstrom** using the Certora Prover. The work was undertaken from May 26th, 2025 to June 23rd, 2025.

The following contract list is included in our scope:

```
contracts/modules/GrowthOutsideUpdater.sol
contracts/modules/PoolUpdates.sol
contracts/modules/PoolRewards.sol
```

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. During the verification process, the Certora team discovered bugs in the Solidity contracts code, as listed on the following page.

Protocol Overview

Angstrom is a decentralized exchange that combines order book functionality with automated market making. The protocol processes trades in batches, where each batch contains multiple orders that are executed atomically. This batch processing approach ensures that all orders within a batch are executed at the same price, preventing front-running and providing fair execution for all participants.

The protocol maintains a sophisticated reward distribution system for liquidity providers. When trades occur, the protocol tracks rewards using a global accumulator and per-tick tracking system. This allows the protocol to accurately distribute fees to liquidity providers based on their position's tick range. The reward tracking is implemented through the `PoolRewards` contract, which maintains a fixed-size array of reward growth values for each possible tick.

Orders in the system are processed in two phases. First, high-priority orders are executed at the beginning of each block, followed by standard user orders. Each order must be signed by the user, and the protocol verifies these signatures before execution. The protocol also handles gas fees by charging them in the trading asset rather than requiring users to hold ETH, making the system more accessible.

The protocol maintains solvency through a delta tracking system that ensures all token balances are properly accounted for. Pool management is handled through a gas-optimized storage system.

The protocol integrates with Uniswap V4's core functionality while adding its own order matching and execution layer. This integration allows the protocol to leverage Uniswap's AMM implementation while providing additional features for order book trading. The integration is implemented through Uniswap's hook system, which allows the protocol to execute custom logic during pool operations.

certora

Findings Summary

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

Severity	Discovered	Confirmed	Fixed
Critical	-	-	-
High	-	-	-
Medium	-	-	-
Low	1	-	-
Informational	3	-	-
Total	**4**	-	-

Severity Matrix

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

certora

Detailed Findings

ID	Title	Severity	Status
L-01	Solvency violation due to rounding	Low	Not yet fixed
I-01	Solvency violation when distributing rewards to zero active liquidity	Informational	Not yet fixed
I-02	Solvency violation due to reward accumulator overflow leading to decreased rewards	Informational	Not yet fixed
I-03	Rounding errors violate reward preservation invariant in liquidity addition	Informational	Not yet fixed

Low Severity Issues

L-01. Solvency violation due to rounding

Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: PoolUpdates.sol and GrowthOutsideUpdater.sol	Status:	Violated Property: Solvency

Description: A subtle rounding issue in the reward distribution mechanism can cause the total rewards claimable by liquidity providers to exceed the amount deducted from the protocol's bundle deltas. This occurs due to discrete threshold effects in X128 fixed-point arithmetic operations.

Both flatDivX128 and fullMulX128 use floor division (rounding down), which typically favors the protocol. However, when multiple liquidity providers are positioned near rounding thresholds, a small increase in globalGrowth can cause multiple LPs to simultaneously cross from 0 rewards to 1+ rewards.

Example: consider a state with two active liquidity providers, both with liquidity 1, total pool liquidity is 2 and the reward accumulator growthInside-lastGrowthInside has value 2^127 and so the rewards are currently 0 for both: ((2^127)*1)/2**128 = 0.
Now distribute a reward of 1 to the current tick, this decreases delta by 1, but global growth by (1*2^128)/2=2^127 causing an increase of 1 in each reward.
In total delta decreases by 1 and sum of rewards increases by 2.

Customer's response:

Fix Review:

Informational Issues

I-01. Solvency violation when distributing rewards to zero active liquidity

Severity: **Informational**	Impact: **Low**	Likelihood: **Low**
Files: modules/GrowthOutsideUpdater.sol	Status:	Violated Property: Solvency

Description: The _decodeAndReward function with currentOnly=true can violate solvency when expectedLiquidity = 0, poolLiquidity != 0, and amount != 0. The function returns early without updating reward accumulators but still subtracts the reward amount from bundle deltas, creating an accounting mismatch where tokens are deducted but no rewards are distributed.

```javascript
// File: contracts/src/modules/GrowthOutsideUpdater.sol
if (amount == 0 || expectedLiquidity == 0) {
    return (reader, amount);
}

// File: contracts/src/modules/PoolUpdates.sol
bundleDeltas.sub(swapCall.asset0, rewardTotal);  // Subtracts returned amount from deltas
```

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

Customer's response: {Customer feedback}

Fix Review: {Comments about the fix}

I-02. Solvency violation due to reward accumulator overflow leading to decreased rewards

Severity: **Informational**	Impact: **Low**	Likelihood: **Low**
Files: modules/GrowthOutsideUpdater.sol	Status:	Violated Property: Solvency

Description: When reward accumulators overflow and wrap around to smaller values, previously calculated rewards can appear to decrease while bundle deltas continue to be decremented. This creates a solvency violation where the protocol tracks more distributed rewards in its accounting than users can actually claim, potentially leading to insufficient funds for legitimate reward claims.

```javascript
// File: contracts/src/modules/GrowthOutsideUpdater.sol
unchecked {
    poolRewards_.globalGrowth += X128MathLib.flatDivX128(amount, pooLiquidity);  // Can overflow
}

// File: contracts/src/modules/PoolUpdates.sol
bundleDeltas.sub(swapCall.asset0, rewardTotal);  // Delta decremented regardless of overflow
```

Customer's response:

Fix Review:

I-03. Rounding errors violate reward preservation invariant in liquidity addition

Severity: **Informational**	Impact: **Low**	Likelihood: **Low**
Files: modules/GrowthOutsideUpdater.sol	Status:	Violated Property: Rewards Invariance

Description: The protocol implements a mathematical formula to preserve rewards when liquidity is added to existing positions, as documented in the code comments. However, due to rounding errors in fixed-point arithmetic operations, the actual implementation may not perfectly preserve rewards, creating small discrepancies that could accumulate over time or be exploited.

The intended invariant is:

```
JavaScript
rewards' == rewards
(growth_inside - last') * L' = (growth_inside - last) * L
```

However, the implementation uses FixedPointMathLib.fullMulDiv which introduces rounding errors that can cause the new lastGrowthInside value to deviate from the mathematically exact solution.

```
JavaScript
// File: contracts/src/modules/PoolUpdates.sol, lines 106-116
// We want to update `lastGrowthInside` such that any previously accrued rewards are
// preserved:
// rewards' == rewards
// (growth_inside - last') * L' = (growth_inside - last) * L
//  growth_inside - last' = (growth_inside - last) * L / L'
// last' = growth_inside - (growth_inside - last) * L / L'
unchecked {
```

```
    uint256 lastGrowthAdjustment = FixedPointMathLib.fullMulDiv(
        growthInside - position.lastGrowthInside, lastLiquidity, newLiquidity  // ← Rounding occurs here
    );
    position.lastGrowthInside = growthInside - lastGrowthAdjustment;  // ← May not preserve exact rewards
}
```

Customer's response:

Fix Review:

Formal Verification

Verification Methodology

We performed verification of the Angstrom protocol using the Certora verification tool which is based on Satisfiability Modulo Theories (SMT). In short, the Certora verification tool works by compiling formal specifications written in the Certora Verification Language (CVL) and Angstrom's implementation source code written in Solidity.
More information about Certora's tooling can be found in the Certora Technology Whitepaper.

If a property is verified with this methodology it means the specification in CVL holds for all possible inputs. However specifications must introduce assumptions to rule out situations which are impossible in realistic scenarios (e.g. to specify the valid range for an input parameter). Additionally, SMT-based verification is notoriously computationally difficult. As a result, we introduce overapproximations (replacing real computations with broader ranges of values) and underapproximations (replacing real computations with fewer values) to make verification feasible.

Rules: A rule is a verification task possibly containing assumptions, calls to the relevant functionality that is symbolically executed and assertions that are verified on any resulting states from the computation.

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. The Certora Prover takes in different arguments as part of the configuration, in this project, we notably used a loop iteration of k=1-4 (in different rules) which unrolls all loops k times and considers only program executions that have at most this many iterations.

2. To simplify formal verification, we apply patches to the original code to simplify functions that are hard to analyze, and are not essential to the functionality we are trying to analyze. The full list of patches we applied appears in certora/patches.
   For example, in SwapCall.sol we applied the patch:

```
JavaScript
    function getId(SwapCall memory self) internal pure returns (PoolId id) {
-       assembly ("memory-safe") {
-           id := keccak256(add(self, POOL_KEY_OFFSET), POOL_KEY_SIZE)
-       }
+       return PoolId.wrap(
+           keccak256(
+               abi.encode(
+                   self.asset0,
+                   self.asset1,
+                   self.fee,
+                   self.tickSpacing,
+                   self.hook
+               )
+           )
+       );
    }
```

which replaces the assembly code with equivalent, non-assembly code, that calculates the Id of the pool that relates to a swapCall.

3. To further simplify the rules to be verified we use summarizations. Summarizations replace every function call to some function call with a simplified version of the function. Summarizations are a form of approximation that allows us to use only the relevant information about the function at hand for our proofs.
   The full list of summaries we applied appears in certora/specs/summaries.
   For example, when analyzing PoolRewards.sol we applied the following summary to the IUniV4.getNextTickGt function:

```
JavaScript
function getNextTickGtCVL(address self, PoolUpdatesHarness.PoolId id, int24 tick,
int24 tickSpacing) returns (bool,int24) {
   // naive summarization
   bool initialized = isInitializedCVL(self,id,tick,tickSpacing);
   int24 nextTick;
   require nextTick > tick, "nextTick increases";
   require nextTick <= tick + tickSpacing, "nextTick increases at most tickSpacing";
}
```

   The summary says that the return value of getNextTickGt can be arbitrary, except it must satisfy the requirements listed above. This does not fully characterise the behavior of the original function, but it suffices for our proofs. Similar summaries were applied to mathematical functions.

4. To reason about internal functions we use “harnesses” that expose internal functions as external. Some harnesses replicate only some functionality of the internal function considered. For example, updateRewardsAndDelta harnesses the updates for rewards and delta from _updatePool and simulateRewardsCalculation harnesses the calculation of rewards from beforeRemoveLiquidity, without distributing rewards.
   The full list of harnesses we used appears in certora/harnesses.

Formal Verification Properties Overview

ID	Title	Impact	Status
P-01	GrowthInside Invariant	GrowthInside is unchanged when reward accumulators are updated.	Verified
P-02	Solvency in UpdatePools	Sum of Rewards and Delta is unchanged when calling decodeAndReward.	Violated
P-03	Rewards Invariance	Rewards are unchanged when liquidity is added.	Violated
P-04	No Rewards for New Position	Adding and immediately removing liquidity gives no rewards.	Verified
P-05	No Rewards After Liquidity Removal	Removing liquidity twice gives no further rewards.	Verified

Detailed Properties

PoolRewards.sol

Module Properties

P-01. growthInside is unchanged when updateAfterTickMove is called	
Status: Verified	**GrowthInside is unchanged when reward accumulators are updated.**

Rule Name	Status	Description	Link to rule report
growthInsideUnchangedInTickMove	Verified	The rule verifies that when updateAfterTickMove is called the values of growthInside for any input is unchanged. This guarantees rewards do not change due to calls to this function.	rule report

GrowthOutsideUpdater.sol

Module Properties

P-02. Solvency in updatePools	
Status: Violated	**Sum of Rewards and Delta is unchanged when calling decodeAndReward.** The property does not hold with equality: see issues I-01, I-02. The property is verified with inequality for limited scenarios, but violated even with inequality for 2 liquidity providers - see Issue L-01. Only 1 loop iteration is considered in all rules.

Rule Name	Status	Description	Link to rule report
updatePoolSolvencyEqCurrentOnly1LP	Violated	The sum of delta and rewards **does not change** when calling decodeAndReward and updating delta, in the case where **donating only to the current tick** with **one** liquidity provider.	rule report
updatePoolSolvencyLeqCurrentOnly1LP	Verified	The sum of delta and rewards **does not increase** when calling decodeAndReward and updating delta, in the case where **donating only to the current tick** with **one** liquidity provider.	
updatePoolSolvencyLeqCurrentOnly2LPs	Violated	The sum of delta and rewards **does not increase** when calling decodeAndReward and updating delta, in the case where **donating only to the current tick** with **two** liquidity provider.	
updatePoolSolvencyLeqNotCurrentOnly1LP	Verified	The sum of delta and rewards **does not increase** when calling decodeAndReward and updating delta, in the case where **above/below the current tick** with **one** liquidity provider.	

Module Properties

P-03. Rewards Invariance

Status: Verified	**Rewards are unchanged when liquidity is added.** The property is violated in that rewards may decrease (see issue I-03), but rewards cannot increase (which is more essential). The rules call beforeAddLiquidity and modify the liquidity to the appropriate values, and calculate the change in rewards.

Rule Name	Status	Description	Link to rule report
addLiquidityDoesNotIncreaseRewards	Verified	Rewards after call to the hook and modification of liquidity can only **decrease**.	rule report
addLiquidityDoesNotDecreaseRewards	Violated	Rewards after call to the hook and modification of liquidity can only **increase**.	

P-04. No Rewards for New Position

Status: Verified	**Adding and immediately removing liquidity gives no rewards.**

Rule Name	Status	Description	Link to rule report
NoRewardsForNewPosition	Verified	Starting with an empty position, adding liquidity and calling beforeAddLiquidity then calculating the rewards gives zero rewards.	rule report

P-05. No Rewards After Liquidity Removal

Status: Verified	**Removing liquidity twice gives no further rewards.**

Rule Name	Status	Description	Link to rule report
NoRewardsAfterRemoval	Verified	After calling beforeRemoveLiquidity the calculated rewards are zero.	rule 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.

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.