

# Formal Verification Report: Angstrom L2

---

- Repository: <https://github.com/Cyfrin/audit-2025-09-sorella-l2-angstrom>
  - Latest Commit Hash: [253ec6e](#)
  - Date: September 2025
  - Author: [@alexzoid\\_eth](#) ([@CyfrinAudits](#) private formal verification engagement)
  - Certora Prover version: 8.1.1
- 

## Table of Contents

---

1. [About Angstrom L2](#)
2. [Formal Verification Approach](#)
  - [Assumptions](#)
3. [Formal Verification Methodology](#)
  - [Types of Properties](#)
  - [Verification Process](#)
4. [Verification Properties](#)
  - [Valid State](#)
  - [State Transitions](#)
5. [Real Issues Properties](#)
6. [Setup and Execution Instructions](#)
  - [Required source code modifications](#)
  - [Verification Execution](#)
  - [Running Verifications](#)
  - [Advanced Options](#)

## About Angstrom L2

---

Angstrom L2 is a Uniswap V4 hook-based MEV protection system designed specifically for L2 with priority fee ordering.

## Formal Verification Approach

---

Complexity is the primary challenge in formal verification. Various techniques can simplify verification without reducing coverage. The main strategy involves decomposing functionality based on verification objectives and analyzing each component separately. For Angstrom L2, I've identified two distinct verification targets: delta management by the hook in Pool Manager, and reward distribution mechanisms.

For delta neutrality verification, pool internals and reward distributions are completely abstracted. The sole objective is verifying that all delta movements through the hook flow in the correct direction. The single property [hookDeltaNeutrality](#) comprehensively validates this critical requirement.

Reward distribution verification presents additional challenges. While time constraints prevented full implementation of this verification component, the strategies outlined below provide a roadmap for future verification efforts. In Certora Prover bitwise operations are overapproximated by default (resulting in imprecise results), loops are restricted to 3 iterations, and non-linear math operations pose significant difficulties for SMT solvers ([as discussed](#) in [Certora Discord](#)). Based on experience with similar tick-based protocol, several effective strategies can enable successful formal verification:

- Summarizing bitwise-dependent internal functions into CVL mappings
- Abstracting non-linear math operations into CVL functions
- Limiting the input range of supported ticks and tick spacing through storage hook constraints (using `require` statements or ghost variable axioms to bound read/write values), so only a limited range of values (e.g., 3 ticks with one tick spacing) can be stored in state

## Assumptions

Assumptions are constraints applied during verification to make the problem tractable for the prover. They are classified as **Safe** (no impact on security guarantees) or **Unsafe** (may limit coverage).

### Safe Assumptions

These assumptions reflect real-world constraints or simplify non-critical aspects without compromising verification validity:

- Core hook callbacks (`beforeSwap()`, `afterSwap()`, `afterAddLiquidity()`, `afterRemoveLiquidity()`) fully verified with harness simulating operations inside an active unlock callback
- ERC20 tokens implemented in CVL, limited to 5 users per contract for tractability
- Block timestamp bounded to realistic values, block number non-zero
- Message sender assumed distinct from contracts under verification

### Unsafe Assumptions

These assumptions reduce verification scope to avoid prover timeouts but potentially may miss edge cases:

- System assumed to maintain only one pool throughout verification
- Loop unrolling capped at 3 iterations, bitwise operations overapproximated
- Tax distribution functions (`_zeroForOneDistributeTax()`, `_oneForZeroDistributeTax()`) and reward library modeled as empty stubs

## Formal Verification Methodology

---

Certora Formal Verification (FV) provides mathematical proofs of smart contract correctness by verifying code against a formal specification. It complements techniques like testing and fuzzing, which can only sometimes detect bugs based on predefined properties. In contrast, Certora FV examines all possible states and execution paths in a contract.

Simply put, the formal verification process involves crafting properties (similar to writing tests) in CVL language and submitting them alongside compiled Solidity smart contracts to a remote prover. This prover essentially transforms the contract bytecode and rules into a mathematical model and determines the validity of rules.

# Types of Properties

When constructing properties in formal verification, we mainly deal with two types: **Invariants** and **Rules**.

## Invariants

- Conditions that **MUST always remain true** throughout the contract's lifecycle.
- Process:
  1. Define an initial condition for the contract's state.
  2. Execute an external function.
  3. Confirm the invariant still holds after execution.
- Example: "Hook must maintain zero balance deltas for all currencies (delta neutral)."
- Use Case: Ensures **Valid State** properties - critical state constraints that **MUST** never be violated.
- Feature: Proven invariants can be reused in other properties with the `requireInvariant` keyword.

## Rules

- Flexible checks for specific behaviors or conditions.
- Structure:
  1. Setup: Set assumptions (e.g., "Assume a valid transaction environment is set").
  2. Execution: Simulate contract behavior by calling external functions.
  3. Verification:
    - Use `assert` to check if a condition is **always true** (e.g., "votes never exceed voting power").
    - Use `satisfy` to verify a condition is **reachable** (e.g., "a user can successfully vote").
- Example: "Only creator can extract tokens from AngstromL2"
- Use Case: Verifies a broad range of properties, from simple state changes to complex business logic.

# Verification Process

The process is divided into two stages: **Setup** and **Crafting Properties**.

## Setup

This stage prepares the contract and prover for verification:

- Resolve external contract calls and dependencies.
- Simplify complex operations (e.g., math or bitwise calculations) for prover compatibility.
- Install storage hooks to monitor state changes.
- Address prover limitations (e.g., timeouts or incompatibilities).

## Crafting Properties

This stage defines and implements the properties:




- Write properties in plain English for clarity.
- Categorize properties by purpose (e.g., Valid State, Variable Transition).
- Prove valid state invariants as a foundation for further rules

## Verification Properties

The verification properties are categorized into two distinct types:




1. **Valid State (VS):** System-wide invariants that **MUST** always hold true. These properties define the fundamental constraints of the protocol, such as accounting consistency and structural integrity. Once proven, these invariants serve as trusted assumptions in other properties via `requireInvariant`, reducing verification complexity.
2. **State Transition (ST):** Properties that verify the correctness of transitions between valid states. Building upon the valid state invariants, these properties ensure the protocol's state machine operates correctly and that state changes are both authorized and sequentially valid.

Links to specific Certora Prover runs are provided for each property, with status indicators:

-  Verified successfully
-  Timeout (property holds but requires optimization)
-  Violated (indicates a potential issue)


## Valid State

Valid State properties define the fundamental invariants that must always hold true throughout the protocol's lifecycle. These properties are organized by contract and proven as invariants, meaning they are checked to hold after every possible function execution from any valid initial state.

Property	Name	Description	Status	Notes
<a href="#">VS-01</a>	<code>hookDeltaNeutrality</code>	Hook must maintain zero balance deltas for all currencies (delta neutral)		Issue: <a href="#">All swaps other than the top-of-block swap will revert</a> .  Not violated in latest version after the fix. <a href="#">Mutation test</a> showcasing the class of bugs that could be caught.
<a href="#">VS-02</a>	<code>totalSwapFeesWithinBounds</code>	Total swap fees (protocol + creator) must be less than 100%		DoS issue in <code>FACTOR_E6</code> - <code>totalSwapFeeRateE6</code> , since <code>totalSwapFeeRateE6</code> can exceed <code>FACTOR_E6</code> since there's no longer a check when setting <code>protocolSwapFeeE6</code>

## State Transitions

State Transition properties verify the correctness of transitions between valid states. These properties ensure that state changes occur only under the right conditions, such as calls to specific functions or time elapsing.

Property	Name	Description	Status	Notes
<a href="#">ST-01</a>	<code>onlyCreatorCanExtractTokens</code>	Only creator can extract tokens from AngstromL2		

## Real Issues Properties

This section documents vulnerabilities discovered during the manual audit (including issues by all participants) and formal verification process. Each issue demonstrates how formal properties detected the vulnerability and confirmed its resolution after applying the fix.

### [MEDIUM] All swaps other than the top-of-block swap will revert (#20)

For swaps that are not top-of-block, `AngstromL2::afterSwap` short-circuits; however, this occurs too late in the execution with an incorrect `hookDeltaUnspecified` after a debt is erroneously created in the invocation of `_computeAndCollectProtocolSwapFee()`. This happens because `_getSwapTaxAmount()` is not top-of-block context dependent, so a non-zero `taxInEther` is passed even though the tax has already been taken from the swap with the highest priority fee.

✗ A property is violated: <https://prover.certora.com/output/52567/2ff4b86b481c42c9b70ca9c7b5d08995/?anonymousKey=b11e81aae9967377cf8f3273d88c841812aa11a>

```
// VS-01 Hook must maintain zero balance deltas for all currencies (delta neutral)
invariant hookDeltaNeutrality(env e)
  forall PoolManager.Currency currency.
    ghostCurrencyDeltas[_AngstromL2][currency] == 0
```

✓ Passed after the fix: <https://prover.certora.com/output/52567/61528817ccbc4f83a2b6ccac6e12058e/?anonymousKey=f3063485b95f5dc7d548fe11fd26a8e895e04a5b>

Also passed with the latest sources. I've executed a manual mutation test within the Certora Mutation Engine, showcasing the class of bugs that could be caught: <https://mutation-testing.certora.com/?id=446dee87-b1d2-4a40-8456-32c9b3e6f2c1&anonymousKey=d077f0f6-b90b-4f07-a8b4-34594e254d8d>

## Setup and Execution Instructions

For step-by-step installation steps refer to this setup [tutorial](#).

### Required source code modifications

No modifications are required to the source code for Certora verification. The verification setup uses harness contracts that wrap the original contracts to enable formal verification without modifying production code.

# Verification Execution

## Running Verifications

### 1. Core Delta Verification:

```
# Run delta specifications (VS-01, VS-02, VS-03, ST-01)
certoraRun certora/confs/UnlockCallback_deltas.conf
```

### 2. Mutation Testing for Hook Delta Neutrality:

```
# Run mutation tests for VS-01 (hookDeltaNeutrality)
certoraMutate certora/confs/UnlockCallback_mutations_hookDeltaNeutrality.conf
```

## Advanced Options

To optimize verification time or debug issues, you can run specific rules:

### 1. Run with specific rule:

```
certoraRun certora/confs/UnlockCallback_deltas.conf --rule hookDeltaNeutrality
certoraRun certora/confs/UnlockCallback_deltas.conf --rule onlyCreatorCanExtractTokens
```

### 2. Run with specific external function:

```
certoraRun certora/confs/UnlockCallback_deltas.conf --method
"swapZeroForOne((address,address,uint24,int24,address),int256,uint160)"
```