Report

v. 3.0

Customer

Algebra

Smart Contract Audit

Algebra Protocol

25th April 2023

Report prepared by

ABDK Consulting

Contents

1 Changelog

#	Date	Author	Description
0.1	02.04.23	A. Zveryanskaya	Initial Draft
0.2	02.04.23	A. Zveryanskaya	Minor revision
1.0	02.04.23	A. Zveryanskaya	Release
1.1	03.04.23	A. Zveryanskaya	Layout revision
2.0	03.04.23	A. Zveryanskaya	Release
2.1	25.04.23	A. Zveryanskaya	Project scope updated
2.2	25.04.23	A. Zveryanskaya	CVF-199-125 are added
3.0	25.04.23	A. Zveryanskaya	Release

2 Introduction

The following document provides the result of the audit performed by ABDK Consulting (Mikhail Vladimirov and Dmitry Khovratovich) at the customer request. The audit goal is a general review of the smart contracts structure, critical/major bugs detection and issuing the general recommendations.

Algebra is a breakthrough AMM, and a concentrated liquidity protocol for decentralized exchanges, running on adaptive fees. Providing projects with more user-friendly, fresh DeFi solutions and implementing the most efficient technologies, it reforms the DeFi field as we know it.

3 Project scope

We were asked to review:

- Original Code
- Code with Fixes

Then were asked to review:

- Additional Code
- Fixed additional Code

Files:

base/common/			
	Timestamp.sol		
base/			
	AlgebraPoolBase.sol	DerivedState.sol	LimitOrderPositions.sol
	Positions.sol	ReentrancyGuard.sol	ReservesManager.sol
	SwapCalculation.sol	TickStructure.sol	
interfaces/callback/			
	IAlgebraFlash Callback.sol	IAlgebraMintCallback.sol	IAlgebraSwap Callback.sol
interfaces/pool/			
	IAlgebraPoolActions.sol	IAlgebraPool DerivedState.sol	IAlgebraPoolEvents.sol
	IAlgebraPool Immutables.sol	IAlgebraPool PermissionedActions.sol	IAlgebraPoolState.sol
interfaces/			
	IAlgebraFactory.sol	IAlgebraFee Configuration.sol	IAlgebraPool.sol
	IAlgebraPool Deployer.sol	IAlgebraPoolErrors.sol	IAlgebraVirtualPool.sol
	IDataStorage Operator.sol	IERC20Minimal.sol	
libraries/			
	AdaptiveFee.sol	Constants.sol	DataStorage.sol
	FullMath.sol	LimitOrderManager.sol	LiquidityMath.sol
	LowGasSafeMath.sol	PriceMovementMath.sol	SafeCast.sol
	SafeTransfer.sol	TickManager.sol	TickMath.sol
	TickTree.sol	TokenDeltaMath.sol	
/			
	AlgebraCommunity Vault.sol	AlgebraFactory.sol	AlgebraPool.sol
	AlgebraPoolDeployer.sol	DataStorage Operator.sol	

Additional Files:

core/contracts/libraries/	
LimitOrderManage-ment.sol	Constants.sol
core/contracts/base/	
LimitOrderPositions.sol	

4 Methodology

The methodology is not a strict formal procedure, but rather a selection of methods and tactics combined differently and tuned for each particular project, depending on the project structure and technologies used, as well as on client expectations from the audit.

- **General Code Assessment**. The code is reviewed for clarity, consistency, style, and for whether it follows best code practices applicable to the particular programming language used. We check indentation, naming convention, commented code blocks, code duplication, confusing names, confusing, irrelevant, or missing comments etc. At this phase we also understand overall code structure.
- **Entity Usage Analysis**. Usages of various entities defined in the code are analysed. This includes both: internal usages from other parts of the code as well as potential external usages. We check that entities are defined in proper places as well as their visibility scopes and access levels are relevant. At this phase, we understand overall system architecture and how different parts of the code are related to each other.
- **Access Control Analysis**. For those entities, that could be accessed externally, access control measures are analysed. We check that access control is relevant and done properly. At this phase, we understand user roles and permissions, as well as what assets the system ought to protect.
- **Code Logic Analysis**. The code logic of particular functions is analysed for correctness and efficiency. We check if code actually does what it is supposed to do, if that algorithms are optimal and correct, and if proper data types are used. We also make sure that external libraries used in the code are up to date and relevant to the tasks they solve in the code. At this phase we also understand data structures used and the purposes they are used for.

We classify issues by the following severity levels:

- **Critical issue** directly affects the smart contract functionality and may cause a significant loss.
- **Major issue** is either a solid performance problem or a sign of misuse: a slight code modification or environment change may lead to loss of funds or data. Sometimes it is an abuse of unclear code behaviour which should be double checked.
- **Moderate issue** is not an immediate problem, but rather suboptimal performance in edge cases, an obviously bad code practice, or a situation where the code is correct only in certain business flows.
- **Minor issues** contain code style, best practices and other recommendations.

5 Our findings

We found 1 critical, 12 major, and a few less important issues. All identified Critical issues have been fixed.

Issues	Active	Fixed
Severity		
Critical	**0**	**1**
Major	**7**	**5**
Moderate	**21**	**14**
Minor	**42**	**35**

Fixed 55 out of 125 issues

6 Critical Issues

CVF-1. FIXED

- **Category** Flaw
- **Source** AlgebraFactory.sol

Description This condition will most probably evaluate to true in case ”renounceOwnershipStartTimestamp” is zero, i.e. if the ”startRenounceOwnership” function wasn’t called.

Client Comment *Fixed. We do not agree that this is a critical issue in our case, rather major.*

```
118 require(block.timestamp - renounceOwnershipStartTimestamp >=
      ↪ RENOUNCE_OWNERSHIP_DELAY);
```

7 Major Issues

CVF-4. FIXED

- **Category** Documentation
- **Source** DataStorage.sol

Description The ”oldestIndex” returned value is not documented.

Recommendation Consider documenting.

Client Comment *We do not agree that this is a major issue in our case.*

```
48 ) internal returns (uint16 indexUpdated, uint16 oldestIndex) {
```

CVF-5. INFO

- **Category** Readability
- **Source** AdaptiveFee.sol

Description Here overflow checks are skipped based on complicated business logic properties. This is a bad practice, as business logic implementation may have bugs, may be changed in the future, or calculations may produce rounding errors.

Recommendation Consider using checked math and safe type conversions anyway.

Client Comment *Considered. We prefer not to spend gas for checks here. We do not agree that this is a major issue in our case.*

```
38 uint256 sumOfSigmoids = sigmoid(volatility, config.gamma1, config.
      ↪ alpha1, config.beta1) +
   sigmoid(volatility, config.gamma2, config.alpha2, config.beta2);
```

```
43 return uint16(config.baseFee + sumOfSigmoids); // safe since alpha1
      ↪ + alpha2 + baseFee _must_ be <= type(uint16).max
```

CVF-7. FIXED

- **Category** Unclear behavior
- **Source** SafeTransfer.sol

Description In case there are more than 32 bytes returned, all extra bytes are silently ignored.

Recommendation Consider reverting when more than 32 bytes were returned.

Client Comment *According to the standard, the token must return exactly one bool, we do not consider it a major.*

```
26 or(and(eq(mload(0), 1), gt(returndatasize(), 31)), iszero(
      ↪ returndatasize())),
```

CVF-8. FIXED

- **Category** Suboptimal
- **Source** SafeTransfer.sol

Description Relying on execution order in such way makes code error-prone and very hard to read.

Recommendation Consider splitting the expression into two separate sentences.

Client Comment *Splitted. We do not agree that this is a major issue in our case.*

```
28 // because and() evaluates its arguments from right to left
```

CVF-9. INFO

- **Category** Suboptimal
- **Source** PriceMovementMath.sol

Description While the "mulDiv" function is very efficient in general case, for specific cases more efficient approaches do exist. For example, when denominator is a compile-time constant, its multiplicative modular inverse could be precomputed.

Client Comment *Adding a separate implementation with binary shifts gives minimal gas savings in specific rare cases (for very large numbers). At the same time, this entails a significant increase in the size of the contract bytecode. We prefer to use a "mulDiv" and use higher compiler optimization settings due to the saved bytecode size. We do not agree that this is a major issue in our case.*

```
123 uint256 amountAvailableAfterFee = FullMath.mulDiv(uint256(
    ↪ amountAvailable), 1e6 - fee, 1e6);
```

CVF-10. INFO

- **Category** Suboptimal
- **Source** LimitOrderPositions.sol

Description While the "mulDiv" function is very efficient in general case, for specific cases better approaches do exist. For example, when a denominator is a power of two, the division could be replaced with a shift of a 512-bit number.

Client Comment *Adding a separate implementation with binary shifts gives minimal gas savings in specific rare cases (for very large numbers). At the same time, this entails a significant increase in the size of the contract bytecode. We prefer to use a "mulDiv" and use higher compiler optimization settings due to the saved bytecode size. We do not agree that this is a major issue in our case.*

```
92 boughtAmount = FullMath.mulDiv(_cumulativeDelta, amountToSellInitial
   ↪ , Constants.Q128);
```

```
94 uint256 price = FullMath.mulDiv(sqrtPrice, sqrtPrice, Constants.Q96)
   ↪ ;
```

CVF-11. FIXED

- **Category** Readability
- **Source** LimitOrderPositions.sol

Description This line relies of the fact that ”amountToSell” and ”amountToSellInitial” both fit into 128 bits. This fact is guaranteed by a code located in a different file that could be changed in the future. Such hidden relationships make code more error-prone and harder to read.

Recommendation Consider explicitly requiring these values to fit into 128 bits.

Client Comment *We do not agree that this is a major issue in our case. Added check.*

```
131 (position.liquidity) = ((amountToSell << 128) | amountToSellInitial)
      ↪ ; // tightly pack data
```

CVF-14. INFO

- **Category** Suboptimal
- **Source** LimitOrderManager.sol

Description While the ”mulDiv” function is very efficient in general case, for specific cases more efficient approaches do exist. For example, when the denominator is a power of two, division could be replaced with shift of a 512-bit number. When the denominator is a compile-time constant, the modular multiplicative inverse of it could be precomputed.

Client Comment *Adding a separate implementation with binary shifts gives minimal gas savings in specific rare cases (for very large numbers). At the same time, this entails a significant increase in the size of the contract bytecode. We prefer to use a ”mulDiv” and use higher compiler optimization settings due to the saved bytecode size. We do not agree that this is a major issue in our case.*

```
77  uint256 price = FullMath.mulDiv(tickSqrtPrice, tickSqrtPrice,
      ↪ Constants.Q96);
```

```
80    ? FullMath.mulDiv(uint256(amountA), price, Constants.Q96) //
        ↪ tokenA is token0
```

```
91    amountOut = FullMath.mulDiv(amountOut, 1e6 - fee, 1e6);
```

```
103 amountIn = zeroToOne ? FullMath.mulDivRoundingUp(amountOut,
      ↪ Constants.Q96, price) : FullMath.mulDivRoundingUp(amountOut,
      ↪ price, Constants.Q96);
```

```
106     feeAmount = FullMath.mulDivRoundingUp(amountIn, fee, 1e6);
```

CVF-15. INFO

- **Category** Suboptimal
- **Source** TokenDeltaMath.sol

Description While in general case the "mulDiv" function is very efficient, for specific cases better approaches do exist. For example, when the denominator is a power of 2 known at compile time, the division could be replaced with a right shift of a 512-bit value.

Client Comment *Adding a separate implementation with binary shifts gives minimal gas savings in specific rare cases (for very large numbers). At the same time, this entails a significant increase in the size of the contract bytecode. We prefer to use a "mulDiv" and use higher compiler optimization settings due to the saved bytecode size. We do not agree that this is a major issue in our case.*

```
43 token1Delta = roundUp ? FullMath.mulDivRoundingUp(priceDelta,
   ↪ liquidity, Constants.Q96) : FullMath.mulDiv(priceDelta,
   ↪ liquidity, Constants.Q96);
```

CVF-16. INFO

- **Category** Flaw
- **Source** FullMath.sol

Description This makes the function quite dangerous to use.

Recommendation Consider making it safer by performing a division by zero check inside.

Client Comment *The function has been renamed to show its dangerous nature. We do not agree that this is a major issue in our case.*

```
124 /// @dev division by 0 has unspecified behavior, and must be checked
   ↪  externally
```

CVF-122. INFO

- **Category** Suboptimal
- **Source** LimitOrderManager.sol

Recommendation While the ”mulDiv” function is very efficient in general case, for specific cases better approaches do exist. For example, when a numerator or a denominator is a power of two,the ”mulDiv” function could be replaced with more efficient shift+div or mul+shift function.

Client Comment *Adding a separate implementation with binary shifts gives minimal gas savings in specific rare cases (for very large numbers). At the same time, this entails a significant increase in the size of the contract bytecode. We prefer to use a ”mulDiv” and use higher compiler optimization settings due to the saved bytecode size. We do not agree that this is a major issue in our case.*

```
52 +  ? FullMath.mulDivRoundingUp(_amountToSell, Constants.Q144,
      ↪ priceX144)
   +  : FullMath.mulDivRoundingUp(_amountToSell, priceX144, Constants.
      ↪ Q144);
```

```
119 +? FullMath.mulDiv(uint256(amountA), priceX144, Constants.Q144) //
       ↪ tokenA is token0
120 +: FullMath.mulDiv(uint256(amountA), Constants.Q144, priceX144); //
       ↪ tokenA is token1
```

```
147 +? FullMath.mulDivRoundingUp(amountOut, Constants.Q144, priceX144)
    +: FullMath.mulDivRoundingUp(amountOut, priceX144, Constants.Q144);
```

CVF-123. FIXED

- **Category** Unclear behavior
- **Source** LimitOrderPositions.sol

Description This logic is unclear and the comment doesn’t help much.

Recommendation Consider elaborating more.

Client Comment *Added more comments.*

```
97 +// initial value isn't zero, but accumulators can overflow
   +if (position.innerFeeGrowth0Token == 0) position.
      ↪ innerFeeGrowth0Token = _limitOrder.boughtAmount0Cumulative;
   +if (position.innerFeeGrowth1Token == 0) position.
      ↪ innerFeeGrowth1Token = _limitOrder.boughtAmount1Cumulative;
```

8 Moderate Issues

CVF-2. INFO

- **Category** Readability
- **Source** AlgebraPool.sol

Description Using a business-level field as a low-level technical flag is a bad practice, as business logic may have bugs of the behavior of the business-level field may change in the future.

Recommendation Consider using a separate dedicated flag.

Client Comment *The fact that the price cannot become zero is the key to the whole logic of the pool. Adding a new flag costs gas, so we prefer not to.*

```
40 if (globalState.price != 0) revert alreadyInitialized(); // after
   ↪ initialization, the price can never become zero
```

CVF-3. INFO

- **Category** Suboptimal
- **Source** AlgebraFactory.sol

Recommendation The value for this constant could be obtained as: keccak256(type(AlgebraPool).creationCode)

Client Comment *With this approach, the size of the factory bytecode becomes too large.*

```
134 bytes32 private constant POOL_INIT_CODE_HASH =
    0x9a6810113806533f58ba03fd4242aeacec87dbc6b15d932f991a4b43ef5dd546;
```

CVF-6. INFO

- **Category** Unclear behavior
- **Source** SwapCalculation.sol

Description Setting the limit price to the min or max available price could be used to execute a ”market” order.

Recommendation Consider not forbidding these values.

Client Comment *For these purposes are used accordingly: MIN_SQRT_RATIO + 1 and MAX_SQRT_RATIO - 1*

```
68 if (limitSqrtPrice >= currentPrice || limitSqrtPrice <= TickMath.
   ↪ MIN_SQRT_RATIO) revert invalidLimitSqrtPrice();
```

```
71 if (limitSqrtPrice <= currentPrice || limitSqrtPrice >= TickMath.
   ↪ MAX_SQRT_RATIO) revert invalidLimitSqrtPrice();
```

CVF-12. INFO

- **Category** Suboptimal
- **Source** TickManager.sol

Recommendation It would be more efficient to assume that all growth happened ”inside”, so no ”outer” counter would need to be updated.

Client Comment *At the moment we don’t have a way to effectively define the ”inside” area. In addition, even if such an approach could be implemented efficiently, it would require a major change in logic in many places in the code. For now, we prefer to leave it as it is.*

```
110 // by convention, we assume that all growth before a tick was
    ↪ initialized happened _below_ the tick
```

CVF-13. INFO

- **Category** Unclear behavior
- **Source** TickManager.sol

Description These assignments do nothing, as the values being assigned were just read from the same storage slots.

Client Comment *The line above removes all data from this storage slot, but since we need to save the previous and next tick for the min/max ticks, this assignment occurs.*

```
165 (self[tick].prevTick, self[tick].nextTick) = (prevTick, nextTick);
```

CVF-17. INFO

- **Category** Suboptimal
- **Source** IAlgebraPoolEvents.sol

Description Indexing tick range boundaries seems useless, as indexed only support exact value lookups.

Recommendation Consider not indexing the tick range boundaries.

Client Comment *Indexing by tick number can be used to obtain information about a particular tick. So we leave it as it is. We do not agree that this is an issue at all in our case.*

```
26   int24 indexed bottomTick,
     int24 indexed topTick,
```

```
41 event Collect(address indexed owner, address recipient, int24
     ↪ indexed bottomTick, int24 indexed topTick, uint128 amount0,
     ↪ uint128 amount1);
```

```
51 event Burn(address indexed owner, int24 indexed bottomTick, int24
     ↪ indexed topTick, uint128 liquidityAmount, uint256 amount0,
     ↪ uint256 amount1);
```

CVF-18. FIXED

- **Category** Suboptimal
- **Source** AlgebraPool.sol

Recommendation Should be ”liquidityDesired” instead of ”liquidityActual”. Currently, the desired liquidity could be decreased twice making it less than is should be. For example, if ”receivedAmount0” is 2 times less than ”amount0” and ”receivedAmount1” is two times less than ”amount1”, then ”liquidityActual” will be four times less than ”liquidityDesired”, while it should be only two time less.

Client Comment *In our opinion this issue is closer to ”major”.*

```
89 uint128 liquidityForRA1 = uint128(FullMath.mulDiv(uint256(
      ↪ liquidityActual), receivedAmount1, amount1));
```

CVF-19. FIXED

- **Category** Procedural
- **Source** AlgebraPool.sol

Recommendation This duplicated code should be removed.

```
102 liquidityActual = liquidityDesired;
    if (receivedAmount0 < amount0) {
      liquidityActual = uint128(FullMath.mulDiv(uint256(liquidityActual)
        ↪ , receivedAmount0, amount0));
    }
    if (receivedAmount1 < amount1) {
      uint128 liquidityForRA1 = uint128(FullMath.mulDiv(uint256(
        ↪ liquidityActual), receivedAmount1, amount1));
      if (liquidityForRA1 < liquidityActual) liquidityActual =
        ↪ liquidityForRA1;
    }
110 if (liquidityActual == 0) revert zeroLiquidityActual();
```

CVF-20. FIXED

- **Category** Overflow/Underflow
- **Source** AlgebraPool.sol

Description Overflow is possible here.

Recommendation Consider using safe convesion.

Client Comment *Added check.*

```
141 int128 liquidityDelta = -int128(amount);
```

CVF-21. INFO

- **Category** Suboptimal
- **Source** AlgebraPool.sol

Description Transferring several different tokens in a single transaction is a bad practice, as if one of the transfers fail for some reason, no transfers will be performed.

Recommendation Consider implementing a two-step withdrawal schema, when the contract count how many tokens each user is eligible to get, and user may initiate token transfers later. This also could be more gas efficient in some cases, as such approach could allow combining several small transfers into one.

Client Comment *When calling the function, the user can choose to request one of the tokens.*

```
176 if (amount0 > 0) SafeTransfer.safeTransfer(token0, recipient,
    ↪ amount0);
    if (amount1 > 0) SafeTransfer.safeTransfer(token1, recipient,
    ↪ amount1);
```

CVF-22. FIXED

- **Category** Overflow/Underflow
- **Source** AlgebraPool.sol

Description Overflow is possible here.

Recommendation Consider using checked math.

Client Comment *Added check.*

```
227 if (amountRequired < 0) amountRequired = -amountRequired; // we
    ↪ support only exactInput here
```

CVF-23. FIXED

- **Category** Overflow/Underflow
- **Source** AlgebraPool.sol

Description Overflow is possible here.

Recommendation Consider using safe conversion.

Client Comment *Added checks.*

```
238 amountReceived = int256(_balanceToken0() - balance0Before);
```

```
241 amountReceived = int256(_balanceToken1() - balance1Before);
```

CVF-24. FIXED

- **Category** Procedural
- **Source** DataStorage.sol

Description While the ”oldestIndex” value assigned here is correct at this time moment, it will become obsolete after creating a new time point few lines below. Returning an obsolete value seems weird.

Recommendation Consider returning an up-to-date value.

Client Comment *Fixed.*

```
59 if (self[indexUpdated].initialized) oldestIndex = indexUpdated;
```

CVF-25. INFO

- **Category** Overflow/Underflow
- **Source** DataStorage.sol

Description Over-/underflow is possible here.

Recommendation Consider using checked math and safe conversions.

Client Comment *With correct parameters, overflows/underflows are impossible, and the result always fits into 88 bits.*

```
258 int256 K = (tick1 - tick0) - (avgTick1 - avgTick0); // (k - p)*dt
    int256 B = (tick0 - avgTick0) * dt; // (b - q)*dt
260 int256 sumOfSequence = dt * (dt + 1); // sumOfSequence * 2
    int256 sumOfSquares = sumOfSequence * (2 * dt + 1); // sumOfSquares
       ↪ * 6
    volatility = uint256((K ** 2 * sumOfSquares + 6 * B * K *
       ↪ sumOfSequence + 6 * dt * B ** 2) / (6 * dt ** 2));
```

CVF-26. INFO

- **Category** Overflow/Underflow
- **Source** DataStorage.sol

Description Over-/underflow is possible here.

Recommendation Consider using checked math.

Client Comment *Over-/underflows are desired here since volatilityCumulative is a relative accumulator and all timestamp interactions in DataStorage allow one underflow or overflow.*

```
352 uint32 target = time - secondsAgo;
```

```
359   return (beforeOrAt.volatilityCumulative +
360     uint88(_volatilityOnRange(int256(uint256(target - beforeOrAt.
          ↪ blockTimestamp)), beforeOrAt.tick, tick, beforeOrAt.
          ↪ averageTick, avgTick)));
```

```
367 (uint32 timepointTimeDelta, uint32 targetDelta) = (timestampAfter -
      ↪ beforeOrAt.blockTimestamp, target - beforeOrAt.blockTimestamp)
      ↪ ;
```

```
369 return beforeOrAt.volatilityCumulative + ((volatilityCumulativeAfter
      ↪  - beforeOrAt.volatilityCumulative) / timepointTimeDelta) *
      ↪ targetDelta;
```

CVF-27. INFO

- **Category** Overflow/Underflow
- **Source** DataStorage.sol

Description Over-/underflow is possible here.

Recommendation Consider using checked math.

Client Comment *Over-/underflows are desired here since tickCumulative is relative accumulator and all timestamp interactions in DataStorage allow one underflow or overflow.*

386
```
uint32 target = time - secondsAgo;
```

398
```
if (samePoint) return ((tickCumulativeBefore + int56(tick) * int56(
   ↪ uint56(target - timestampBefore))), _indexBeforeOrAt); // if
   ↪ target is newer than last timepoint
```

404
```
(uint32 timepointTimeDelta, uint32 targetDelta) = (timestampAfter -
   ↪ timestampBefore, target - timestampBefore);
```

406
```
  tickCumulativeBefore + ((tickCumulativeAfter -
     ↪ tickCumulativeBefore) / int56(uint56(timepointTimeDelta))) *
     ↪  int56(uint56(targetDelta)),
```

CVF-28. INFO

- **Category** Overflow/Underflow
- **Source** DataStorage.sol

Description Underflow is possible here.

Recommendation Consider using checked math.

Client Comment *Underflow is desired since "time" can overflow.*

435
```
if (time - target <= WINDOW) {
```

CVF-29. FIXED

- **Category** Unclear behavior
- **Source** AdaptiveFee.sol

Description Its unclear why this value always fits into 155 bits.

Recommendation Consider explaining and/or using checked math below to no rely of this.

Client Comment *Added comments.*

```
56 uint256 ex = expXg4(x, g, g4); // < 155 bits
```

CVF-30. FIXED

- **Category** Unclear behavior
- **Source** AdaptiveFee.sol

Description Its unclear why this value always fits into 156 bits.

Recommendation Consider explaining and/or using checked math below to no rely of this.

Client Comment *Added comments.*

```
63 uint256 ex = g4 + expXg4(x, g, g4); // < 156 bits
```

CVF-31. INFO

- **Category** Flaw
- **Source** AdaptiveFee.sol

Description This assumes that ”xdg” doesn’t exceed 5, which is not actually checked.

Recommendation Consider adding an explicit check for this fact.

Client Comment *Added comment that this function has good accuracy only if x/g < 6.*

```
93 default {
     closestValue := 14841315910257660342111 // ~= e^5
   }
```

CVF-32. FIXED

- **Category** Suboptimal
- **Source** SwapCalculation.sol

Description The reasoning behind this line is unclear. It seems that the semantics of the ”step.inLimitOrder” field is different in different cases.

Recommendation Consider refactoring or at least clearly documenting the logic.

Client Comment *Renamed.*

```
117 step.inLimitOrder = !step.inLimitOrder;
```

CVF-33. INFO

- **Category** Overflow/Underflow
- **Source** SwapCalculation.sol

Description Over-/underflow is possible here.

Recommendation Consider using checked math.

Client Comment *Business logic guarantees that overflows/underflows can not occur.*

```
132 amountRequired -= (step.input + step.feeAmount).toInt256(); //
    ↪ decrease remaining input amount
```

```
135 amountRequired += step.output.toInt256(); // increase remaining
    ↪ output amount (since its negative)
```

```
141 step.feeAmount -= delta;
    communityFeeAmount += delta;
```

```
159     currentTick = zeroToOne ? step.nextTick : step.nextTick - 1;
```

```
177     liquidityDelta = -ticks.cross(
```

```
199   ? (step.nextTick - 1, cache.prevInitializedTick)
```

CVF-34. INFO

- **Category** Suboptimal
- **Source** SwapCalculation.sol

Description Just catching possible errors is not enough to protect from a failed nested call as it may consume too much gas or return to long result.

Recommendation Consider using an assembly block instead and limiting the gas amount.

Client Comment *A wrap in a try-catch is used just in case, errors in the called contract must be excluded. The address of the contract used can be changed after the deployment of the pools, and if there are problems described in this contract (too much gas or return too long result), users will be able to withdraw liquidity from the pools without any problems. In this case, even a malicious actor that has seized the rights to set the contract address will not be able to block user' funds, only to stop swaps.*

```
166 try IAlgebraVirtualPool(cache.activeIncentive).cross(step.nextTick,
      ↪ zeroToOne) returns (bool success) {

168 } catch {}
```

CVF-35. INFO

- **Category** Overflow/Underflow
- **Source** AlgebraPoolBase.sol

Description Overflow is possible here.

Recommendation Consider using checked math and safe conversion.

Client Comment *Overflows and underflows are desired here since it is relative values. Overflow in cast to uint160 is impossible. Added comments.*

```
119 secondsPerLiquidityCumulative += ((uint160(blockTimestamp - _lastTs)
      ↪  << 128) / (currentLiquidity > 0 ? currentLiquidity : 1));

138   _secPerLiqCumulative += ((uint160(blockTimestamp - _lastTs) <<
        ↪ 128) / (currentLiquidity > 0 ? currentLiquidity : 1));
```

CVF-36. INFO

- **Category** Suboptimal
- **Source** AlgebraPoolBase.sol

Description Simply catching error is not enough to reliably isolate nested call failure, as a nested call may still consume too much gas and also may return too long value.

Recommendation Consider using assembly block instead of a "try/catch" block and limiting gas usage.

Client Comment *The correctness of the size of the returned data is checked at compile time. A wrap in a try-catch is used just in case, errors in the called contract must be excluded.*

```
123 // failure should not occur. But in case of failure, the pool will
      ↪ remain operational
```

CVF-37. FIXED

- **Category** Overflow/Underflow
- **Source** ReservesManager.sol

Description Underflow and overflow are possible in calculations and type conversions.

Recommendation Consider using checked math and safe conversions.

Client Comment *Added comments and checks.*

```
59 (uint256 _cfPending0, uint256 _cfPending1) = (communityFeePending0 +
     ↪  communityFee0, communityFeePending1 + communityFee1);
```

```
70   (deltaR0, deltaR1) = (deltaR0 - int256(_cfPending0), deltaR1 -
       ↪ int256(_cfPending1));
```

```
73 (communityFeePending0, communityFeePending1) = (uint128(_cfPending0)
     ↪ , uint128(_cfPending1));
```

CVF-38. FIXED

- **Category** Overflow/Underflow
- **Source** ReservesManager.sol

Description Overflow and underflow are possible here.

Recommendation Consider using safe conversions.

Client Comment *Added checks.*

```
79 if (deltaR0 != 0) _reserve0 = uint128(int128(_reserve0) + int128(
   ↪ deltaR0));
80 if (deltaR1 != 0) _reserve1 = uint128(int128(_reserve1) + int128(
   ↪ deltaR1));
```

CVF-39. FIXED

- **Category** Flaw
- **Source** LimitOrderPositions.sol

Description Here, the "_limitOrder.boughtAmount0Cumulative" value is overwritten without checking that it is indeed zero. It is hard to tell whether it is possible for it to be non-zero.

Recommendation Consider adding an explicit assert for this fact.

Client Comment *Added "initialized" flag.*

```
81 (_limitOrder.boughtAmount0Cumulative, _limitOrder.
   ↪ boughtAmount1Cumulative) = (1, 1); // maker pays for storage
   ↪ slots
```

CVF-40. FIXED

- **Category** Overflow/Underflow
- **Source** LimitOrderPositions.sol

Description Over-/underflow is possible when converting types.

Recommendation Consider using safe conversions.

Client Comment *Added checks and comments. Other conversions are safe by business logic.*

```
107   if (boughtAmount > 0) position.fees1 = position.fees1.add128(
          ↪ uint128(boughtAmount));
```

```
110   if (boughtAmount > 0) position.fees0 = position.fees0.add128(
          ↪ uint128(boughtAmount));
```

```
121     ? -int256(FullMath.mulDiv(uint128(-amountToSellDelta),
            ↪ amountToSellInitial, amountToSell)).toInt128()
        : int256(FullMath.mulDiv(uint128(amountToSellDelta),
            ↪ amountToSellInitial, amountToSell)).toInt128();
```

```
126 amountToSell = LiquidityMath.addDelta(uint128(amountToSell),
        ↪ amountToSellDelta);
    amountToSellInitial = LiquidityMath.addDelta(uint128(
        ↪ amountToSellInitial), amountToSellInitialDelta);
```

CVF-41. FIXED

- **Category** Overflow/Underflow
- **Source** LimitOrderManager.sol

Description Underflow is possible here.

Recommendation Consider using checked math.

Client Comment *Added checks.*

```
74 if (!exactIn) amountA = -amountA;
```

CVF-42. INFO

- **Category** Overflow/Underflow
- **Source** LimitOrderManager.sol

Description Over/underflow is possible here.

Recommendation Consider using checked math.

Client Comment *Business logic guarantees that overflows/underflows can not occur or desired. Added comments.*

```
88  uint256 unsoldAmount = amountToSell - soldAmount;
```

```
91    amountOut = FullMath.mulDiv(amountOut, 1e6 - fee, 1e6);
```

```
100   data.soldAmount = soldAmount + uint128(amountOut);
```

```
111   feeAmount = FullMath.mulDivRoundingUp(amountIn, fee, 1e6 - fee);
```

```
115   data.boughtAmount0Cumulative += FullMath.mulDivRoundingUp(amountIn
        ↪ , Constants.Q128, amountToSell);
```

```
117   data.boughtAmount1Cumulative += FullMath.mulDivRoundingUp(amountIn
        ↪ , Constants.Q128, amountToSell);
```

CVF-43. INFO

- **Category** Overflow/Underflow
- **Source** LimitOrderManager.sol

Description Overflow is possible when converting to ”uint128”.

Recommendation Consider using safe conversion.

Client Comment *Added comment.*

```
100 data.soldAmount = soldAmount + uint128(amountOut);
```

CVF-44. INFO

- **Category** Suboptimal
- **Source** IAlgebraPoolActions.sol

Description It seems unfair that flash fees go only to the in-range liquidity providers, while assets provided by out of range providers may also be flash loaned.

Recommendation Consider distributing the flash fees among all the liquidity providers. Otherwise one can provide liquidity into the current tick, then flash loan much more liquidity from the protocol, and thus pay majority of the flash loan fees to himself.

Client Comment *At the moment, there is no satisfactory way to distribute liquidity among all providers. Current implementation incentivizes providers to manage liquidity more actively.*

```
106 /// @dev All excess tokens paid in the callback are distributed to
    ↪ currently in-range liquidity providers as an additional fee.
```

CVF-121. INFO

- **Category** Unclear behavior
- **Source** LimitOrderManager.sol

Description This assumption relies on a business logic constraints enforced elsewhere, which introduces a hidden relationship inside the code.

Recommendation Consider checking this assumptions with an explicit ”assert” statement.

Client Comment *We prefer to avoid ”assert” here to reduce bytecode size and gas costs*

```
49 +  // MAX_LIMIT_ORDER_TICK check guarantees that this value does not
   ↪  overflow
```

```
113 +// MAX_LIMIT_ORDER_TICK check guarantees that this value does not
    ↪ overflow
```

CVF-124. INFO

- **Category** Unclear behavior
- **Source** LimitOrderPositions.sol

Description These assumptions rely on business logic constraints enforced elsewhere, which introduces hidden relationships inside the code.

Recommendation Consider checking these assumptions with explicit "assert" statements.

Client Comment *We prefer to avoid "assert" here to reduce bytecode size and gas costs*

114
```
+  // MAX_LIMIT_ORDER_TICK check guarantees that this value does not
   ↪  overflow
```

126
```
+// casts aren't checked since boughtAmount must be <= type(uint128)
   ↪ .max (we are not supporting tokens with totalSupply > type(
   ↪ uint128).max)
```

9 Minor Issues

CVF-45. INFO

- **Category** Procedural
- **Source** Timestamp.sol

Description Specifying a particular compiler version makes it harder to upgrade to newer versions.

Recommendation Consider specifying as "^0.8.0". Also relevant for: AlgebraPool.sol, DataStorageOperator.sol, AlgebraFactory.sol, AlgebraCommunityVault.sol, DataStorage.sol, AdaptiveFee.sol, TickStructure.sol, SwapCalculation.sol, AlgebraPoolBase.sol, PriceMovementMath.sol, ReservesManager.sol, ReentrancyGuard.sol, Positions.sol, LimitOrderPositions.sol, DerivedState.sol, TickManager.sol, LimitOrderManager.sol, TokenDeltaMath.sol, Constants.sol.

Client Comment *We do not mean compiling with other versions of Solidity.*

```
2 pragma solidity =0.8.17;
```

CVF-46. INFO

- **Category** Procedural
- **Source** Timestamp.sol

Recommendation In Ethereum, timestamps are 256-bits long, thus for a generic function like this one, consider returning a 256-bits number and truncating in a calling code if necessary.

Client Comment *Everywhere in our code, a 32-bit number is used as a timestamp*

```
6 /// @return A timestamp converted to uint32
```

CVF-47. FIXED

- **Category** Suboptimal
- **Source** Timestamp.sol

Recommendation The "unchecked" block is redundant, as there are no operations inside that could be checked.

Client Comment *Removed.*

```
8 unchecked {
```

CVF-48. INFO

- **Category** Bad datatype
- **Source** AlgebraPoolDeployer.sol

Recommendation The type of this variable should be ”IAlgebraFactory”.

Client Comment *We prefer to leave it as it is*

```
14 address private immutable factory;
```

CVF-49. INFO

- **Category** Bad datatype
- **Source** AlgebraPoolDeployer.sol

Recommendation The type of the ”_factory” argument should be ”IAlgebraFactory”.

Client Comment *We prefer to leave it as it is*

```
17 constructor(address _factory, address _communityVault) {
```

CVF-50. FIXED

- **Category** Bad naming
- **Source** AlgebraPoolDeployer.sol

Description The semantics of the returned values is unclear.

Recommendation Consider giving descriptive names to the returned values and/or describing in the documentation comment.

Client Comment *Added names.*

```
23 function getDeployParameters() external view override returns (
     ↪ address, address, address, address, address) {
```

CVF-51. FIXED

- **Category** Bad datatype
- **Source** AlgebraPool.sol

Recommendation The value 500 should be a named constant.

```
324 if (newTickSpacing <= 0 || newTickSpacing > 500 || tickSpacing ==
    ↪ newTickSpacing) revert invalidNewTickSpacing();
```

CVF-52. INFO

- **Category** Bad datatype
- **Source** DataStorageOperator.sol

Recommendation The type of this variable should be ”IAlgebraPool”.

Client Comment *We prefer to leave it as it is.*

```
25 address private immutable pool;
```

CVF-53. INFO

- **Category** Bad datatype
- **Source** DataStorageOperator.sol

Recommendation The type of this variable should be ”IAlgebraFactory”.

Client Comment *We prefer to leave it as it is.*

```
26 address private immutable factory;
```

CVF-54. INFO

- **Category** Suboptimal
- **Source** DataStorageOperator.sol

Recommendation The argument type should be ”IAlgebraPool”.

Client Comment *We prefer to leave it as it is.*

```
33 constructor(address _pool) {
```

CVF-55. INFO

- **Category** Bad datatype
- **Source** AlgebraFactory.sol

Recommendation The type of this variable should be ”IAlgebraPoolDeployer”.

Client Comment *We prefer to leave it as it is.*

```
23 address public immutable override poolDeployer;
```

CVF-56. INFO

- **Category** Bad datatype
- **Source** AlgebraFactory.sol

Recommendation The type of this mapping should be: mapping(IERC20 ⇒ mapping(IERC20 ⇒ IAlgebraPool))

Client Comment *We prefer to leave it as it is.*

```
44 mapping(address => mapping(address => address)) public override
   ↪ poolByPair;
```

CVF-57. INFO

- **Category** Bad datatype
- **Source** AlgebraFactory.sol

Recommendation The type of the arguments should be”IERC20”.

Client Comment *We prefer to leave it as it is.*

```
64 function createPool(address tokenA, address tokenB) external
   ↪ override returns (address pool) {
```

```
140 function _computeAddress(address token0, address token1) private
    ↪ view returns (address pool) {
```

CVF-58. INFO

- **Category** Bad datatype
- **Source** AlgebraFactory.sol

Recommendation The return type should be ”IAlgebraPool”.

Client Comment *We prefer to leave it as it is.*

```
64 function createPool(address tokenA, address tokenB) external
      ↪ override returns (address pool) {
```

```
140 function _computeAddress(address token0, address token1) private
      ↪ view returns (address pool) {
```

CVF-59. FIXED

- **Category** Unclear behavior
- **Source** AlgebraFactory.sol

Description This event is emitted even if nothing actually changed.

```
91 emit DefaultCommunityFee(newDefaultCommunityFee);
```

CVF-60. FIXED

- **Category** Bad naming
- **Source** AlgebraCommunityVault.sol

Recommendation Events are usually named via nouns, such as ”Withdrawal”.

```
10 event TokensWithdrawn(address indexed token, address indexed to,
      ↪ uint256 amount);
```

CVF-61. INFO

- **Category** Bad datatype
- **Source** AlgebraCommunityVault.sol

Recommendation The type of the ”token” parameter should be ”IERC20”.

Client Comment *We prefer to leave it as it is.*

```
10 event TokensWithdrawn(address indexed token, address indexed to,
   ↪ uint256 amount);
```

CVF-62. INFO

- **Category** Bad datatype
- **Source** AlgebraCommunityVault.sol

Recommendation The type of this variable should be ”IAlgebraFactory”.

Client Comment *We prefer to leave it as it is.*

```
14 address private immutable factory;
```

CVF-63. INFO

- **Category** Bad datatype
- **Source** AlgebraCommunityVault.sol

Recommendation The type of the ”token” argument should be ”IERC20”.

Client Comment *We prefer to leave it as it is.*

```
25 function withdraw(address token, address to, uint256 amount)
   ↪ external onlyWithdrawer {
```

CVF-64. INFO

- **Category** Bad datatype
- **Source** AlgebraCommunityVault.sol

Recommendation The type of the ”tokens” argument should be ”IERC20 []”.

Client Comment *We prefer to leave it as it is.*

```
29 function withdrawTokens(address[] calldata tokens, address[]
   ↪ calldata tos, uint256[] calldata amounts) external
   ↪ onlyWithdrawer {
```

CVF-65. FIXED

- **Category** Suboptimal
- **Source** AlgebraCommunityVault.sol

Recommendation It would be more efficient to pass a single array of structs with three fields, rather than three parallel arrays. This would also make the length checks unnecessary.

```
29 function withdrawTokens(address[] calldata tokens, address[]
   ↪ calldata tos, uint256[] calldata amounts) external
   ↪ onlyWithdrawer {
```

CVF-66. INFO

- **Category** Readability
- **Source** DataStorage.sol

Description Here an argument is used as a local variable.

Recommendation Consider avoiding such practice and it makes code harder to read.

Client Comment *Considered.*

```
316 lastIndex = oldestIndex + 1;
```

CVF-67. INFO

- **Category** Procedural
- **Source** DataStorage.sol

Description These blocks are never executes for correct arguments.

Recommendation Consider reverting in them rather then returning something.

Client Comment *In case of an emergency, the worst result right now is an incorrect fee value and disruption of the oracle. Adding assertions can potentially result in the user's funds being blocked.*

492
```
    // beforeOrAt is initialized and <= target, and next timepoint
       ↪ is uninitialized
    // should be impossible if initial boundaries and `target` are
       ↪ correct
    return (beforeOrAt, beforeOrAt, indexBeforeOrAt);
```

500
```
// we've landed on an uninitialized timepoint, keep searching higher
// should be impossible if initial boundaries and `target` are
   ↪ correct
left = indexBeforeOrAt + 1;
```

CVF-68. FIXED

- **Category** Documentation
- **Source** AdaptiveFee.sol

Recommendation It would be good to explain the actual decimal values of the constants here.

Client Comment *Added comments.*

14
```
3000 - Constants.BASE_FEE, // alpha1
15000 - 3000, // alpha2
360, // beta1
60000, // beta2
59, // gamma1
8500, // gamma2
```

CVF-69. INFO

- **Category** Suboptimal
- **Source** AdaptiveFee.sol

Recommendation For a binary fixed-point value x, e^x could be calculated quite efficiently and very precisely as described here:https://hackmd.io/@abdk/B1pqT9yl2

Client Comment *Noted.*

```
70 /// @notice calculates e^(x/g) * g^4 in a series, since (around zero
      ↪ ):
   /// e^x = 1 + x + x^2/2 + ... + x^n/n! + ...
   /// e^(x/g) = 1 + x/g + x^2/(2*g^2) + ... + x^(n)/(g^n * n!) + ...
```

CVF-70. INFO

- **Category** Suboptimal
- **Source** TickStructure.sol

Description This condition is always true.

Recommendation Consider either removing the conditional statement, or replacing it with an assert.

Client Comment *This condition will be true only if the subtrees become active or vice versa after the tick toggle.*

```
58 if (newTickTreeRoot != oldTickTreeRoot) tickTreeRoot =
      ↪ newTickTreeRoot;
```

CVF-71. FIXED

- **Category** Suboptimal
- **Source** TickTree.sol

Recommendation A single conversion to ”uint24” would be sufficient.

```
76 if (treeRoot & (1 << uint256(int256(nodeNumber))) != 0) {
```

CVF-72. FIXED

- **Category** Procedural
- **Source** TickTree.sol

Recommendation The conversion to ”int24” is redundant as Solidity compiler could do it automatically.

```
87 (nextTick, initialized) = nextTickInTheSameNode(treeRoot, int24(++
   ↪ nodeNumber));
```

CVF-73. FIXED

- **Category** Bad naming
- **Source** TickTree.sol

Description These function names are misleading as the functions are used on various tree levels, not only on the ticks level.

Recommendation Consider renaming to avoid the ”tick” word.

Client Comment *Renamed*.

```
101 function _getNextTickInSameNode(
```

```
115 function _getFirstTickInNode(mapping(int16 => uint256) storage row,
    ↪ int24 nodeNumber) private view returns (int24 nextTick) {
```

```
128 function nextTickInTheSameNode(uint256 word, int24 tick) private
    ↪ pure returns (int24 nextTick, bool initialized) {
```

CVF-74. FIXED

- **Category** Procedural
- **Source** TickTree.sol

Description Unlike other private functions, this function doesn’t have the underscore (”_”) prefix in its name.

Recommendation Consider using consistent naming.

```
128 function nextTickInTheSameNode(uint256 word, int24 tick) private
    ↪ pure returns (int24 nextTick, bool initialized) {
```

CVF-75. FIXED

- **Category** Procedural
- **Source** SwapCalculation.sol

Recommendation This check should be done earlier.

```
62 if (amountRequired == 0) revert zeroAmountRequired();
```

CVF-76. FIXED

- **Category** Procedural
- **Source** SwapCalculation.sol

Recommendation It is a good practice to put a comment into an empty block to explain why the block is empty.

```
168 } catch {}
```

CVF-77. INFO

- **Category** Bad datatype
- **Source** AlgebraPoolBase.sol

Recommendation The type of this variable should be ”IAlgebraFactory”.

Client Comment *We prefer to leave it as it is*

```
33 address public immutable override factory;
```

CVF-78. INFO

- **Category** Bad datatype
- **Source** AlgebraPoolBase.sol

Recommendation The type of these variables should be ”IERC20”.

Client Comment *We prefer to leave it as it is*

```
35 address public immutable override token0;
```

```
37 address public immutable override token1;
```

CVF-79. INFO

- **Category** Suboptimal
- **Source** AlgebraPoolBase.sol

Recommendation It would be more efficient to merge these the mappings into a single mapping whose keys are ticks and values are structs of two fields encapsulating the values of the original mappings.

Client Comment *Considered. Combining structures can increase the amount of irrelevant data returned to the function caller.*

```
70 mapping(int24 => TickManager.Tick) public override ticks;
```

```
72 mapping(int24 => LimitOrderManager.LimitOrder) public override
   ↪ limitOrders;
```

CVF-80. FIXED

- **Category** Bad naming
- **Source** AlgebraPoolBase.sol

Description The semantics of the returned values is unclear.

Recommendation Consider giving descriptive names to the returned values and/or describing in the documentation comment.

Client Comment *Added names.*

```
114 function _writeTimepoint(uint16 timepointIndex, uint32
    ↪ blockTimestamp, int24 tick, uint128 currentLiquidity) internal
    ↪  returns (uint16, uint16) {
```

CVF-81. FIXED

- **Category** Documentation
- **Source** AlgebraPoolBase.sol

Description The reasoning behind replacing zero liquidity with liquidity 1 is unclear.

Recommendation Consider explaining it.

Client Comment *Added comments.*

```
119 secondsPerLiquidityCumulative += ((uint160(blockTimestamp - _lastTs)
      ↪  << 128) / (currentLiquidity > 0 ? currentLiquidity : 1));
```

```
138   _secPerLiqCumulative += ((uint160(blockTimestamp - _lastTs) <<
        ↪ 128) / (currentLiquidity > 0 ? currentLiquidity : 1));
```

CVF-82. FIXED

- **Category** Procedural
- **Source** SafeTransfer.sol

Description Specifying an unbounded compiler version range is a bad practice, as one cannot guarantee compatibility with future major releases.

Recommendation Consider specifying as "^0.8.0". Also relevant for: LowGasSafeMath.sol, TickMath.sol, LiquidityMath.sol, FullMath.sol, IAlgebraPoolErrors.sol.

Client Comment *These libraries use custom errors introduced in 0.8.4, so we can't use the suggested specifying. Limited the version.*

```
2 pragma solidity >=0.8.4;
```

CVF-83. INFO

- **Category** Bad datatype
- **Source** SafeTransfer.sol

Recommendation The type of the "token" argument should be "IERC20".

Client Comment *We prefer to leave it as it is.*

```
16 function safeTransfer(address token, address to, uint256 amount)
     ↪ internal {
```

CVF-84. INFO

- **Category** Readability
- **Source** PriceMovementMath.sol

Recommendation Should be ”else return”.

Client Comment *We prefer the current version.*

```
55 return uint160(FullMath.divRoundingUp(liquidityShifted, (
   ↪ liquidityShifted / price).add(amount)));
```

CVF-85. FIXED

- **Category** Bad datatype
- **Source** PriceMovementMath.sol

Recommendation The value ”1e6” should be a named constant.

```
123 uint256 amountAvailableAfterFee = FullMath.mulDiv(uint256(
   ↪ amountAvailable), 1e6 - fee, 1e6);
```

```
127   feeAmount = FullMath.mulDivRoundingUp(input, fee, 1e6 - fee);
```

```
136     feeAmount = FullMath.mulDivRoundingUp(input, fee, 1e6 - fee);
```

```
161 feeAmount = FullMath.mulDivRoundingUp(input, fee, 1e6 - fee);
```

CVF-86. FIXED

- **Category** Suboptimal
- **Source** ReservesManager.sol

Description This logic looks weird, as it sends to the community vault all the tokens above 2^128.

Recommendation Consider clearly explaining the reasoning behind this logic. Currently the logic seems unfair for liquidity providers.

Client Comment *Added comments.*

```
22 if (balance0 > type(uint128).max) {
     SafeTransfer.safeTransfer(token0, communityVault, balance0 - type(
        ↪ uint128).max);
     balance0 = type(uint128).max;
   }
   if (balance1 > type(uint128).max) {
     SafeTransfer.safeTransfer(token1, communityVault, balance1 - type(
        ↪ uint128).max);
     balance1 = type(uint128).max;
   }
```

CVF-87. FIXED

- **Category** Procedural
- **Source** ReservesManager.sol

Description The expressions ”balance0 > _reserve0” and ”balance1 > _reserve1” are calculated twice.

Recommendation Consider calculating once and reusing.

```
36 if (balance0 > _reserve0 || balance1 > _reserve1) {
```

```
38     if (balance0 > _reserve0) {
```

```
41     if (balance1 > _reserve1) {
```

CVF-88. INFO

- **Category** Procedural
- **Source** Positions.sol

Description The expression ”liquidityDelta < 0” is calculated twice.

Recommendation Consider calculating once and reusing.

Client Comment *It looks like the change will use more gas than it saves.*

```
115 previousTick = _insertOrRemoveTick(bottomTick, currentTick,
       ↪ previousTick, liquidityDelta < 0);
```

```
118 previousTick = _insertOrRemoveTick(topTick, currentTick,
       ↪ previousTick, liquidityDelta < 0);
```

CVF-89. INFO

- **Category** Readability
- **Source** DerivedState.sol

Recommendation Should be ”else if”.

Client Comment *We prefer the current version.*

```
27 if (currentTick < topTick) {
```

CVF-90. FIXED

- **Category** Bad datatype
- **Source** DerivedState.sol

Recommendation Should be ”else return”.

```
35 return (upperOuterSecondPerLiquidity - lowerOuterSecondPerLiquidity,
       ↪  upperOuterSecondsSpent - lowerOuterSecondsSpent);
```

CVF-91. FIXED

- **Category** Bad naming
- **Source** TickManager.sol

Description The word ”manager” sounds odd in a library name.

Recommendation Consider renaming.

Client Comment *Renamed*.

```
12 library TickManager {
```

CVF-92. FIXED

- **Category** Suboptimal
- **Source** TickManager.sol

Description This is performed in all branches.

Recommendation Consider coding in one place outside of conditional statements.

```
164   delete self[tick]; // MIN_TICK and MAX_TICK cannot be removed from
        ↪  tick list
```

```
173 delete self[tick];
```

CVF-93. FIXED

- **Category** Bad naming
- **Source** LimitOrderManager.sol

Description The word ”manager” sounds odd in a library name.

Recommendation Consider renaming.

Client Comment *Renamed*.

```
9 library LimitOrderManager {
```

CVF-94. INFO

- **Category** Suboptimal
- **Source** LimitOrderManager.sol

Description Inside an unchecked block these two lines basically do the same.

Recommendation Consider merging them into one line.

Client Comment *Yes, it is, but we prefer to keep it more readable.*

```
28 _amountToSell += uint128(amount);

30 _amountToSell -= uint128(-amount);
```

CVF-95. INFO

- **Category** Suboptimal
- **Source** LimitOrderManager.sol

Description Inside an unchecked block these two branches basically do the same.

Recommendation Consider merging them into one line.

Client Comment *Yes, it is, but we prefer to keep it more readable.*

```
45 if (amount > 0) {
     data.amountToSell += uint128(amount);
     data.soldAmount += uint128(amount);
   } else {
     data.amountToSell -= uint128(-amount);
50   data.soldAmount -= uint128(-amount);
   }
```

CVF-96. FIXED

- **Category** Procedural
- **Source** TickMath.sol

Recommendation The conversion to ”uint256” is redundant, as Solidity compiler may do it automatically.

```
31 uint256 absTick = uint256(uint24((tick ^ mask) - mask));
   if (absTick > uint256(uint24(MAX_TICK))) revert IAlgebraPoolErrors.
      ↪ tickOutOfRange();
```

CVF-97. FIXED

- **Category** Suboptimal
- **Source** TickMath.sol

Recommendation This could be simplified as: price = uint160(ratio + 0xFFFFFFFF » 32);

Client Comment *Simplified.*

```
60 price = uint160((ratio >> 32) + (ratio % (1 << 32) == 0 ? 0 : 1));
```

CVF-98. FIXED

- **Category** Procedural
- **Source** SafeCast.sol

Description Specifying an unbounded compiler version range is a bad practice, as one cannot guarantee compatibility with future major releases.

Recommendation Consider specifying as "^0.5.0 || ^0.6.0 || ^0.7.0 || ^0.8.0". Also revelant for: IAlgebraMintCallback.sol, IAlgebraFlashCallback.sol, IAlgebraFactory.sol, IAlgebraVirtualPool.sol, IAlgebraPoolPermissionedActions.sol, IAlgebraPoolEvents.sol, IAlgebraPoolState.sol, IAlgebraPoolActions.sol, IAlgebraPoolDerivedState.sol, IERC20Minimal.sol, IDataStorageOperator.sol, IAlgebraFeeConfiguration.sol, IAlgebraPoolDeployer.sol, IAlgebraPoolImmutables.sol, IAlgebraSwapCallback.sol, IAlgebraPool.sol.

Client Comment *Library versions were bounded. Interface versions are left as is.*

```
2 pragma solidity >=0.5.0;
```

CVF-99. FIXED

- **Category** Suboptimal
- **Source** SafeCast.sol

Recommendation This could be simplified as: require ((z = int256(y)) >= 0);

Client Comment *Simplified.*

```
34 require(y < 2 ** 255);
   z = int256(y);
```

CVF-100. INFO

- **Category** Suboptimal
- **Source** LiquidityMath.sol

Recommendation Here ”x - uint128(-y)” could be replaced with ”x + uint128(y)”.

Client Comment *Yes, it is, but we prefer to keep it more readable.*

```
19 if ((z = x - uint128(-y)) >= x) revert IAlgebraPoolErrors.
      ↪ liquiditySub();
```

CVF-101. INFO

- **Category** Procedural
- **Source** Constants.sol

Description This library contains only constants.

Recommendation Consider removing the library and moving the constant to the top level.

Client Comment *We prefer to leave it as it is to make it clear that these are constants.*

```
4 library Constants {
```

CVF-102. FIXED

- **Category** Readability
- **Source** Constants.sol

Recommendation Consider rendering the value as ”0.0001e6”.

Client Comment *Changed.*

```
11 uint16 internal constant BASE_FEE = 100; // init minimum fee value
      ↪ in hundredths of a bip (0.01%)
```

CVF-103. FIXED

- **Category** Readability
- **Source** Constants.sol

Recommendation Consider rendering the value as ”0.25e3”.

Client Comment *Changed.*

```
20 uint8 internal constant MAX_COMMUNITY_FEE = 250;
```

CVF-104. FIXED

- **Category** Bad naming
- **Source** IAlgebraFactory.sol

Recommendation Events are usually named via nouns, such as ”RenounceOwnership-Start”, ”RenounceOwnershipStop”, etc.

```
14 event renounceOwnershipStarted(uint256 timestamp, uint256
   ↪ finishTimestamp);

18 event renounceOwnershipStopped(uint256 timestamp);

22 event renounceOwnershipFinished(uint256 timestamp);
```

CVF-105. FIXED

- **Category** Bad naming
- **Source** IAlgebraFactory.sol

Recommendation Event names are usually start with capital letters.

```
14 event renounceOwnershipStarted(uint256 timestamp, uint256
   ↪ finishTimestamp);

18 event renounceOwnershipStopped(uint256 timestamp);

22 event renounceOwnershipFinished(uint256 timestamp);
```

CVF-106. INFO

- **Category** Bad datatype
- **Source** IAlgebraFactory.sol

Recommendation The type of the token parameters should be ”IERC20”.

Client Comment *We prefer to leave it as it is.*

```
28 event Pool(address indexed token0, address indexed token1, address
   ↪ pool);
```

CVF-107. INFO

- **Category** Bad datatype
- **Source** IAlgebraFactory.sol

Recommendation The type of the arguments should be ”IERC20”.

Client Comment *We prefer to leave it as it is.*

```
75 function poolByPair(address tokenA, address tokenB) external view
   ↪ returns (address pool);
```

```
86 function createPool(address tokenA, address tokenB) external returns
   ↪  (address pool);
```

CVF-108. INFO

- **Category** Bad datatype
- **Source** IAlgebraFactory.sol

Recommendation The return type should be ”IAlgebraPool”.

Client Comment *We prefer to leave it as it is.*

```
86 function createPool(address tokenA, address tokenB) external returns
   ↪  (address pool);
```

CVF-109. INFO

- **Category** Procedural
- **Source** IAlgebraPoolState.sol

Description In Ethereum, timestamps are usually represented as 256-bit numbers.

Recommendation Consider changing the return type to ”uint256”.

Client Comment *Everywhere in our code, a 32-bit number is used as a timestamp.*

```
50 function communityFeeLastTimestamp() external view returns (uint32);
```

CVF-110. INFO

- **Category** Bad naming
- **Source** IAlgebraPoolState.sol

Description Despite the name, this function returns information about a single tick.

Recommendation Consider renaming.

Client Comment *This function is actually a mapping.*

```
77 function ticks(
```

CVF-111. INFO

- **Category** Suboptimal
- **Source** IAlgebraPoolErrors.sol

Recommendation These errors could be made more useful by adding some parameters to them.

Client Comment *Noted. At the moment, we do not consider it necessary to use arguments.*

```
43 error invalidNewTickSpacing();
   error invalidNewCommunityFee();
```

```
64 error tickOutOfRange();
   error priceOutOfRange();
```

CVF-112. FIXED

- **Category** Procedural
- **Source** IAlgebraFeeConfiguration.sol

Description This interface contains only a struct.

Recommendation Consider moving the struct definition to the top level and removing the interface.

Client Comment *Moved.*

```
6 interface IAlgebraFeeConfiguration {
```

CVF-113. INFO

- **Category** Documentation
- **Source** IAlgebraFeeConfiguration.sol

Description The number format of these fields is unclear.

Recommendation Consider documenting.

```
10 uint16 alpha1; // max value of the first sigmoid
   uint16 alpha2; // max value of the second sigmoid
   uint32 beta1; // shift along the x-axis for the first sigmoid
   uint32 beta2; // shift along the x-axis for the second sigmoid
   uint16 gamma1; // horizontal stretch factor for the first sigmoid
   uint16 gamma2; // horizontal stretch factor for the second sigmoid
   uint16 baseFee; // minimum possible fee
```

CVF-114. INFO

- **Category** Bad datatype
- **Source** IAlgebraPoolDeployer.sol

Recommendation The type of the ”dataStorage” argument should be ”IDataStorageOperator”.

Client Comment *We prefer to leave it as it is.*

```
18 function getDeployParameters() external view returns (address
   ↪ dataStorage, address factory, address communityVault, address
   ↪ token0, address token1);
```

```
25 function deploy(address dataStorage, address token0, address token1)
   ↪  external returns (address pool);
```

CVF-115. INFO

- **Category** Bad datatype
- **Source** IAlgebraPoolDeployer.sol

Recommendation The type of the ”factory” argument should be ”IAlgebraFactory”.

Client Comment *We prefer to leave it as it is.*

```
18 function getDeployParameters() external view returns (address
   ↪ dataStorage, address factory, address communityVault, address
   ↪ token0, address token1);
```

CVF-116. INFO

- **Category** Bad datatype
- **Source** IAlgebraPoolDeployer.sol

Recommendation The type of the ”token0” and ”token1” arguments should be ”IERC20”.

Client Comment *We prefer to leave it as it is.*

```
18 function getDeployParameters() external view returns (address
   ↪ dataStorage, address factory, address communityVault, address
   ↪ token0, address token1);
```

```
25 function deploy(address dataStorage, address token0, address token1)
   ↪  external returns (address pool);
```

CVF-117. INFO

- **Category** Bad datatype
- **Source** IAlgebraPoolImmutables.sol

Recommendation The return type should be ”IAlgebraFactory”.

Client Comment *We prefer to leave it as it is.*

```
14 function factory() external view returns (address);
```

CVF-118. INFO

- **Category** Bad datatype
- **Source** IAlgebraPoolImmutables.sol

Recommendation The return type should be ”IERC20”.

Client Comment *We prefer to leave it as it is.*

```
18 function token0() external view returns (address);
```

```
22 function token1() external view returns (address);
```

CVF-119. FIXED

- **Category** Documentation
- **Source** Constants.sol

Recommendation Consider adding a comment explaining how this number was calculated and what it guarantees about limit order prices.

Client Comment *Added comments.*

```
20 +int24 constant MAX_LIMIT_ORDER_TICK = 776363;
```

CVF-120. FIXED

- **Category** Bad naming
- **Source** LimitOrderManager.sol

Recommendation The argument name should be ”currentTick”.

```
26 +/// @param tick The current tick in pool
```

CVF-125. FIXED

- **Category** Procedural
- **Source** LimitOrderPositions.sol

Recommendation Usually, ”assert” is used rather then ”require” for checks that should never fail.

Client Comment *Changed to ”assert”*

```
155 +require(amountToSell <= type(uint128).max && amountToSellInitial <=
        ↪  type(uint128).max); // should never fail, just in case
```

ABDK
Consulting

About us

Established in 2016, is a leading service provider in the space of blockchain development and audit. It has contributed to numerous blockchain projects, and co-authored some widely known blockchain primitives like Poseidon hash function.

The ABDK Audit Team, led by Mikhail Vladimirov and Dmitry Khovratovich, has conducted over 40 audits of blockchain projects in Solidity, Rust, Circom, C++, JavaScript, and other languages.

Contact

Email

dmitry@abdkconsulting.com

Website

abdk.consulting

Twitter

twitter.com/ABDKconsulting

LinkedIn

linkedin.com/company/abdk-consulting