certora

Security Assessment & Formal Verification *DRAFT PRELIMINARY* Report

Uniswap v4 core

Date 07/24

Prepared for

Uniswap

Table of content

certora UNISWAP

Project Summary

Project Scope

Project Name	Repository (link)	Latest Commit Hash	Platform
Uniswap/v4-core	Uniswap V4 core repository	d5d4957	EVM/Solidity 0.8

Project Overview

This document describes the specification and verification of the **UniswapV4-core** using the Certora Prover and manual code review findings. The work was undertaken from **May 28th** to **July 2nd.**

The following contract list is included in our scope:

```
src/*
```

The Certora Prover demonstrated that the implementation of the Solidity contracts above is correct with respect to the formal rules written by the Certora team. In addition, the team performed a manual audit of all the Solidity contracts**.** During the verification process and the manual audit, the Certora team discovered bugs in the Solidity contracts code, as listed below.

Protocol Overview

Uniswap V4 is the fourth iteration of the Uniswap AMM protocol. It uses the CLMM logic from Uniswap V3 and creates novel features to be created on top of it through Hooks . Hooks allow flexible AMMs strategies and protocols to be created on top of Uniswap infrastructure, diminishing the effort and cost that it takes to create and innovate on AMM strategies and protocols.

The actors interacting with this protocol could be labeled as LPs, Swappers, Donators and Hooks. These roles are not mutually exclusive and the same actor could perform all of these actions.

Findings Summary

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

Severity	Discovered	Confirmed	Fixed
Critical	0	0	0
High	0	0	0
Medium	2	2	1
Low	3	3	2
Informational	3	3	3
Total	8	8	6

Severity Matrix

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

Detailed Findings

ID	Title	Severity	Status
M-01	Incorrect use of "memory-safe" annotation potentially causing executable code corruption	Medium	Acknowledged and Fixed
M-02	Tick may not correspond to the sqrtPrice	Medium	Acknowledged
L-01	Reachable state degeneration due to price change	Low	Acknowledged
L-02	Liquidity addition DoS by filling gross liquidity in initializable ticks	Low	Acknowledged and Fixed
L-03	Incorrect assumptions about most significant bits of narrow types	Low	Acknowledged and Fixed
I-01	Flash accounting cannot be used for actions that require untrusted calls	Informational	Acknowledged and Fixed
I-02	Same currency can generate multiple token ids	Informational	Acknowledged and Fixed
I-03	Array exttload() and extsload() functions may corrupt results due to improper ABI decoding	Informational	Acknowledged and Fixed

Medium Severity Issues

M-01 Incorrect use of `"memory-safe"` annotation potentially causing executable code corruption

Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: Several	Status: Reported	Violated Rule:

Description: In order for the optimizer to be able to perform some optimizations on bare assembly code, it needs certain properties of the code to be guaranteed. Authors can use the `"memory-safe"` dialect in order to guarantee that the assembly code fulfills certain assumptions around respecting the memory safety model. In the codebase, these annotations are used extensively. Unfortunately, some of them are used while the required assumptions are not fulfilled, and given that these assumptions need to be strictly adhered to because of the nature of optimizer's reasoning, it is likely to lead to incorrect and undefined behavior that cannot be easily discovered via testing.

Moreover, even if bytecode is verified, any small change in the code, a minor Solidity version update, or even a compiler configuration change can cause a new unexpected code corruption issue.

Recommendations: Make sure that any memory layout range, or allocated memory is not being corrupted in any assembly blocks. In particular, make sure to write any temporary data that may be longer than 64 bytes at the free memory pointer.

Customer's response: Acknowledged and fixed.

Fix Review: Fixed in commit https://github.com/Uniswap/v4-core/pull/759/files.

M-02 Tick may not correspond to the sqrtPrice

Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: Pool.sol	Status: Reported	Violated Rule: *TickSqrtPriceStrongCorrelation*

Description: At the end of the loop in the `swap()` function in `Pool.sol`, the tick information is being updated – if the next price was reached and if the tick is initialized, it is crossed. Aside from that, the new current tick is set to the value corresponding to the next price. Also, if `zeroForOne`, the tick is decreased by one to account for next swaps – they will be happening in the tick below.

But, an incorrect assumption is made that whenever the next price is achieved, the swap will continue. Such a loop iteration may happen to be the last (significant) one (because the price limit has been set to the next price, the amount specified has run out, or the remaining amount specified is insufficient to move the price any further in the next iteration of the loop). In these cases, the tick is decreased, despite the price exactly at the initializable tick at the next price corresponding to the tick larger by one.

Impact: Despite a crucial invariant being broken, ticks are crossed properly, so the only direct impact is that the `donate()` function in `PoolManager` may donate to the wrong tick (and hence a wrong set of positions) whenever the price is exactly at a price corresponding to an initializable tick. Hooks or other actors could be using the `donate()` function as a core part of their strategy to compensate LPs, therefore this error could heavily impact certain integrators.

Recommendations: Consider decrementing the tick only after the price has gone below the price at a given tick in `zeroForOne` swaps.

Customer's response: Acknowledged. Won't be fixed. Clarifying comments have been added to the code in PR #851.

Low Severity Issues

L-01 Reachable state degeneration due to price change		
Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: Pool.sol	Status: Reported	Violated Rule:

Description: The initialization can set the price to any value in the range `[MIN_PRICE, MAX_PRICE - 1]`. `MAX_PRICE` is not achievable by design, since it belongs to a tick with price range above the maximum considered price. But the pool can have the price at `MIN_PRICE`.

Despite that, due to a swap (which is the only way to change the price), the price may not go back to `MIN_PRICE`. Hence, after the pool is initialized with any price other than `MIN_PRICE`, the price will never be able to go back.

Impact: The liquidity in range `[MIN_PRICE, MIN_PRICE + 1]` can be utilized only once, at the very first swap (that changes the price) in the pool, if the price was initialized at `MIN_PRICE`. Afterward, this range will "contain" only `currency1`.

Recommendations: Change the boolean at line 325 to `<` instead of `<=` in order to allow the price limit to be exactly at the minimum price.

Customer's response: Acknowledged. Won't be fixed.

L-02 **Liquidity addition DoS by filling gross liquidity in initializable ticks**		
Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: [Pool.sol]()	Status: Reported	Violated Rule {Optional}:

Description: In order to make sure that the liquidity between any initializable ticks doesn't overflow without evaluating it in every interval possible, bounds on `liquidityGross` are introduced. This means that there is a constraint of about 2^{107} in liquidity starting and ending at any tick (for tick spacing of 1, the limit grows approximately linearly with the tick spacing).

A very narrow position around price 1 (which could be the current price) that utilizes this liquidity limit costs about 2^{93} token wei. So, it may be affordable for an attacker to deposit a position of this size to DoS a single liquidity addition operation of any other user.

The capital requirement of this attack shrinks as the position gets further from the current price. A single tick wide position at an extreme price costs only about $5.20 \cdot 10^8$ ($\approx 2^{29}$) token weis. So, an attacker can quite cheaply add positions that will DoS any full-range liquidity additions in the future. This griefing can be easily bypassed by LPs choosing the ends of their positions a number of ticks before the min or max tick ranges.

However, the attacker can add single-tick(-spacing) wide positions every second available tick (spacing), from most extreme prices towards the current price. For a full pool DoS, and assuming the price is close to 1, this will cost about 2^{106} of both tokens. But, if an attacker decides to DoS only ticks 1000x above and below the current price, the capital requirement also decreases about 1000x.

Attack capital requirements (except from the last one, which rises linearly) rise quadratically as the tick spacing increases. This is because increasing it increases both the maximum gross liquidity at any tick and the width of a minimal position.

Recommendations: Make sure that these properties (e.g. possible inability to add full-range positions) is well-documented.

Customer's response: Acknowledged and fixed.

Fix Review: Fixed

L-03 Incorrect assumptions about most significant bits of narrow types

Severity: **Low**	Impact: **High**	Likelihood: **Very Low**
Files: Several	Status: Reported	Violated Rule:

Description:

The Solidity language explicitly doesn't give any guarantees about bits that do not take part in a Type's encoding. This means, for example, the 232 most significant bits of an `int24` are neither guaranteed to be zero nor consistent with the sign (for the number to be treated as a valid `int256`). The potential impact could be very severe and lead to drainage of the protocol in case an attacker managed to leverage the higher (dirty) bits as exemplified in finding I-02. Despite not being able to identify an exploitable vector for this attack, we strongly recommend preventative measures to be taken in order to make this attack vector unexploitable.

Recommendations: Clear most significant bits whenever necessary, or use `signextend` to make sure a number can be interpreted as a higher-sized signed integer.

Customer's response: Acknowledged and fixed.

Fix Review: Fixed in commit https://github.com/Uniswap/v4-core/pull/780.

Informational Severity Issues

I-01. Flash accounting cannot be used for actions that require untrusted calls

Description: While it is usually possible to swap without making untrusted calls when the `PoolManager` is unlocked, this cannot be said for actions that, aside from swaps, require a flash loan to be executed.

The core reason is that anyone can accrue a nonzero `currencyDelta` that will cause the whole call to revert on the `unlock()` level.

An example could be a liquidation scenario, where the lending protocol makes an untrusted call to position's owner (and properly limits the gas consumed and doesn't revert on call failure). Normally, such an action cannot affect the EVM contexts executing above (unless through some reentrancy issues), but with the current design of flash accounting, the untrusted contract can create a nonzero delta for itself and let it remain unsettled, until the whole operation reverts on the `unlock()` level. In order to carry out a liquidation of such an account correctly, one can't use Uniswap v4's flash logic as it currently is.

Recommendation: Consider introducing an option to block any contracts from modifying the `currencyDelta`s, until the same actor unblocks this option.

Customer's response: "Not a complete fix, but an improvement"

Fix Review: Partially fixed in commit https://github.com/Uniswap/v4-core/pull/786/files

I-02. Same currency can generate multiple token ids

Description: `mint()` and `burn()` functions accept an `uint256` as a currency identifier, and this value can have 96 most significant bits dirty. They are discarded when converting the value to an address. It means that the same currency can have 2^{96} native variants of itself.

This works similarly to subaccounts when considering only a single actor using them.

Fix Review: Fixed in commit https://github.com/Uniswap/v4-core/pull/776/files

I-03. Array `exttload()` and `extsload()` functions may corrupt results due to improper ABI decoding

Description: The ABI encoding of a single array encodes the pointer in calldata to the array, and at that pointer, the length is stored. Right after the length, all array elements are encoded, one by one.

The pointer doesn't have to point to the nearest available place in the calldata. It may point to any place in the calldata, even if it's a part of the pointer itself, other data, or can be beyond the next free place. But, the ABI decoding of an array written in assembly in `exttload()` Exttload.sol and `extsload()` Extsload.sol assumes that the ABI encoding always puts the array right after the pointer to it. If that's not the case, the code misunderstands the length and/or the contents of the calldata array, and can return wrongly-sized output and/or corrupt its contents. Although, Solidity's default ABI encoding will never produce calldata that can be misinterpreted that way.

Impact: Integrators optimizing for gas, or using other languages that also produce valid ABI-encoded data could end up producing this non-default calldata, leading to `exttload()` and `extsload()` reading the wrong values.

Recommendations: Read the value of the array pointer in calldata and read values starting from there. Alternatively, we recommend this behavior to be clearly outlined in the docs for potential integrators.

Customer's response: Acknowledged and fixed.

Fix Review: Fixed in commit https://github.com/Uniswap/v4-core/pull/781

Formal Verification

Assumptions and Simplifications

Project General Assumptions

A. We used Solidity Compiler version 8.26 to verify the protocol.
B. All assembly blocks are memory-safe.
C. We assume mulDiv behaves correctly

Verification Notations

Formally Verified	The rule is verified for every state of the contract(s), under the assumptions of the scope/requirements in the rule.
Formally Verified After Fix	The rule was violated due to an issue in the code and was successfully verified after fixing the issue
Violated	A counter-example exists that violates one of the assertions of the rule.

Formal Verification Properties

Pool Manager

Module General Assumptions

- The following properties are proved for src/PoolManager.sol contract.
- We assume that all loops (especially swap loop) iterate at most once.
- All *extsload* accesses are safe and correct.
- Calculations of storage slots through *StateLibrary* are correct.

Contract Properties

P-01. Balance deltas are zero whenever the contract is locked	
Status: Verified	Property Assumptions:

Rule Name	Status	Description	Rule Assumptions	Link to rule report
must_terminate_with_zero_delta	Verified	*After a successful call for the unlock function, all deltas should be zero.*	*Caller is not the contract itself*	
nonZeroCorrect	Verified	*The amount of all non-zero deltas of all users and all currencies equals the value stored at NONZERO_DELTA_COUNT_SLOT.*	*The amount of all non-zero deltas of all users and all currencies is less than max uint256.*	
isLockedAndDeltaZero	Verified	*When the contract is locked, all deltas are zero.*	*hold before and after the outermost call to the contract.*	

P-02. Pool Initialization is Done Correctly	
Status: Verified	Property Assumptions:

Rule Name	Status	Description	Rule Assumptions	Link to rule report
InitializedPoolHasValidTickSpacing	Verified	*Tick spacing is greater than 0 always.*		
initializationSetsPriceCorrectly	Verified	*Pool not initialized before the call and is initialized after.* *Price is set as expected and is valid.*		
initializeIsSafe	Verified	*Initializing a new pool doesn't affect any currency deltas.*		
pool_sqrt_price_never_turns_zero	Verified	*The sqrtPrice at any initialized pool can never turn zero.*		

P-03. Integrity of swap()

Status: Verified	Property Assumptions:

Rule Name	Status	Description	Rule Assumptions	Link to rule report
net_liquidity_immutable_in_swap	Verified	*Net liquidity doesn't change after a call for swap.*		
swapCantIncreaseBothCurrencies	Verified	*swap can't increase currency deltas for the user in both tokens*	*Caller is not the contract itself*	
ValidSwapFee	Verified	*The liquidity provider fee is less than the MAX_LP_FEE. Both protocol fees are less than MAX_PROTOCOL_FEE.*		
swap_integrity	Verified	*swaps until specified amount is swapped or limit price is reached. Update Price and Tick Data in the right direction.*		

P-04. Modify Liquidity Accounting	
Status: Verified	Property Assumptions:

Rule Name	Status	Description	Rule Assumptions	Link to rule report
only_modify_liquidity_changes_position_liquidity	Verified	*The only function which can change a position's liquidity is modifyLiquidity.*		
modify_liquidity_position_changes_correctly	Verified	*modifyLiquidity properly changes the correct position only, based on function input and the msg.sender.*		
liquidity_changed_by_owner_only	Verified	*Only the msg.sender can change their own position liquidity.*		
change_of_liquidity_preserves_funds	Verified	*The currency deltas resulting from a change of position's liquidity match the change of position's value.*	*The relevant pool is initialized with valid ticks and price.*	
modify_liquidity_returns_position_funds	Verified	*The modifyLiquidity function returns deltas that match the position funds function.*	*The relevant pool is initialized with valid ticks, tick spacing and price.* *Assume no hooks delta (AFTER_REMOVE_LIQUIDITY_FLAG is off)*	

active_liquidity_is_updated_correctly_modifyLiquidity	Verified	*modifyLiquidity correctly updates the total active liquidity if the position is active or not.*		
funds_of_total_liquidity_exceeds_sum_of_position_funds	Verified	*The underlying funds of the entire liquidity in a tick range exceeds the sum of funds for all positions in a tick. By induction, this is true, for a sum of any number of arbitrary positions.*		
modifyLiquidity_doesnt_affect_others	Verified	*Modifying liquidity for one position doesn't affect the liquidity for other positions.*		[link to rule report]()

P-05. Swap Accounting

Status: Verified	Property Assumptions:

Rule Name	Status	Description	Rule Assumptions	Link to rule report
positions_to_the_left_dont_change_value	Verified	*When swapping to the right, inactive positions to the left don't change their value.*	*The relevant pool is initialized with valid ticks and price.* `params.zeroForOne` *is set to* `false`*.*	

positions_to_the_right_dont_change_value	Verified	*When swapping to the left, inactive positions to the right don't change their value.*	*The relevant pool is initialized with valid ticks and price.* `params.zeroForOne` *is set to* `true`.	
position_funds_change_upon_tick_slip_max_upper	Verified	*When a tick shifts to the left (*`zeroForOne = true`*), positions funds with* `tickUpper = MAX_TICK()` *should change by the amount deltas of the prices before and after the shift.*	*The relevant pool is initialized with valid ticks and price.* *The relevant position is active.*	
position_funds_change_upon_tick_slip_min_lower	Verified	*When a tick shifts to the right (*`zeroForOne = false`*), positions funds with* `tickLower = MIN_TICK()` *should change by the amount deltas of the prices before and after the shift.*	*The relevant pool is initialized with valid ticks and price.* *The relevant position is active.*	

P-06. Valid Liquidity State

Status: Verified	Property Assumptions:

Rule Name	Status	Description	Rule Assumptions	Link to rule report
liquidityGrossCorrect	Verified	*The `liquidityGross` of a tick is the sum of all liquidites from positions whose lower or upper tick is that tick.*		
liquidityNetCorrect	Verified	*The `liquidityNet` of a tick is the difference between the total liquidity from all positions whose lower tick is that tick and the total liquidites from all positions whose upper tick is that tick.*		
NoGrossLiquidityForUninitializedTick	Verified	*A tick has gross liquidity if and only if it's initialized.*		
OnlyAlignedTicksPositions	Verified	*Only positions with tick boundaries that are aligned with the tick spacing have liquidity.*		

P-07. Valid Pool State	
Status: Violated	Property Assumptions:

Rule Name	Status	Description	Rule Assumptions	Link to rule report
ValidTickAndPrice	Verified	*For every initialized pool the tick is valid (`MIN_TICK` <= tick <= `MAX_TICK`) and the price is valid as well (`MIN_SQRT_PRICE` <= sqrt price <= `MAX_SQRT_PRICE`).*	*The relevant pool is initialized.*	
TickSqrtPriceStrongCorrelation	Violated	*tick(poolSqrtPrice) = pool tick*	*The relevant pool is initialized.*	
TickSqrtPriceCorrelation	Verified	*The current pool tick corresponds to the current price of the pool and vice versa, or the price is exactly at a tick and the tick is one too low .* ***Weak version of* TickSqrtPriceStrongCorrelation**	*The relevant pool is initialized.*	
ValidSwapFee	Verified	*The liquidity provider fee is less than the `MAX_LP_FEE`. Both protocol fees are less than `MAX_PROTOCOL_FEE`.*		

Protocol Fee Library

Module General Assumptions

- The following properties are proved for src/libraries/LPFeeLibrary.sol and src/libraries/ProtocolFeeLibrary.sol contracts.
- All *extsload* accesses are safe and correct.

Contract Properties

P-01. Correct Fee Calculations	
Status: Verified	Property Assumptions:

Rule Name	Status	Description	Rule Assumptions	Link to rule report
test_getZeroForOneFee	Verified	*Verify ZeroForOne fee calculation with Maximum protocol fee.*		link to rule report
test_FV_getZeroForOneFee	Verified	*Verify correct Calculation of ZeroForOne Fee based on input fee.*		link to rule report
test_getOneForZeroFee	Verified	*Verify OneForZero Fee Calculation with Maximum Protocol Fee.*		link to rule report
test_FV_getOneForZeroFee	Verified	*Verify Correct Calculation of OneForZero Fee Based on Input Fee.*		link to rule report

certora UNISWAP

test_FV_isValidProtocolFee_fee	Verified	*Verify Correct Calculation of isValidProtocolFee Fee Based on Input Fee.*		[link to rule report]()
test_FV_calculateSwapFee	Verified	*Verify Swap Fee Calculation with Protocol and LP Fees*		[link to rule report]()

SqrtPrice Math

Module General Assumptions

- The following properties are proved for src/libraries/SqrtPriceMath.sol contract.

P-01. Library mathematical properties	
Status: Verified	Property Assumptions:

Rule Name	Status	Description	Rule Assumptions	Link to rule report
getAmount0Delta_zero_diff	Verified	*Verify that the getAmount0Delta function correctly returns zero when the square roots of the lower and upper price bounds are equal or when the liquidity is zero.*		[link to rule report]()
getAmount0Delta_symmetric	Verified	*Verify Symmetry in Amount0Delta Calculation.*		[link to rule report]()

getAmount0Delta_rounding_diff	Verified	*Verify that the difference in the getAmount0Delta function's output, when rounding up versus rounding down, is either zero or one, ensuring minimal discrepancy due to rounding.*		[link to rule report]()
getAmount0Delta_liquidity_monotonic	Verified	*Verify Monotonicity of Amount0Delta with Respect to Liquidity.*	*Prices are valid.*	[link to rule report]()
getAmount0Delta_sqrtPrice_monotonic	Verified	*Verify Monotonicity of Amount0Delta with Respect to Square Root Prices.*	*At least one of the prices is either `MIN_SQRT_RATIO` or `MAX_SQRT_RATIO`. Prices are valid.*	[link to rule report]()
getAmount0Delta_sqrtPrice_additivity	Verified	*Verify Additivity of Amount0Delta Across Price Intervals (up to rounding error of 1).*	*At Least one of the prices is either `MIN_SQRT_RATIO` or `MAX_SQRT_RATIO`. Prices are valid.*	[link to rule report]()
getAmount0Delta_liquidity_additivity	Verified	*Verify Additivity of Amount0Delta with Respect to Combined Liquidity (up to rounding error of 1).*	*At Least one of the prices is either `MIN_SQRT_RATIO` or `MAX_SQRT_RATIO`. Prices are valid.*	[link to rule report]()
getAmount1Delta_zero_diff	Verified	*Verify that the getAmount1Delta function correctly returns zero when the square roots of the lower and upper price bounds are equal or*		[link to rule report]()

		when the liquidity is zero.		
getAmount1Delta_symmetric	Verified	*Verify Symmetry in Amount1Delta Calculation.*		[link to rule report]()
getAmount1Delta_rounding_diff	Verified	*Verify that the difference in the getAmount1Delta function's output, when rounding up versus rounding down, is either zero or one, ensuring minimal discrepancy due to rounding.*		[link to rule report]()
getAmount1Delta_sqrtPrice_monotonic	Verified	*Verify Monotonicity of Amount1Delta with Respect to Square Root Prices.*	*Prices are valid.*	[link to rule report]()
getAmount1Delta_liquidity_monotonic	Verified	*Verify Monotonicity of Amount1Delta with Respect to Liquidity.*	*Prices are valid.*	[link to rule report]()
getAmount1Delta_sqrtPrice_additivity	Verified	*Verify Additivity of Amount1Delta Across Price Intervals (up to rounding error of 1).*	*Prices are valid.*	[link to rule report]()
getAmount1Delta_liquidity_additivity	Verified	*Verify Additivity of Amount1Delta with Respect to Combined Liquidity (up to rounding error of 1).*	*Prices are valid.*	[link to rule report]()
amountDelta_getNextSqrtPriceFromI	Verified	*Verify that the* `getNextSqrtPriceFromI`	*Prices are valid.*	[link to rule report]()

nput_bound		`nput` *function returns a new square root price within the correct bounds based on the input amount, validating the function's accuracy for both zero-for-one and one-for-zero scenarios.*		
amountDelta_getNextSqrtPriceFromOutput_bound	Verified	*Verify that the* `getNextSqrtPriceFromOutput` *function returns a new square root price within the correct bounds based on the input amount, validating the function's accuracy for both zero-for-one and one-for-zero scenarios.*	*Prices are valid.*	
getNextSqrtPriceFromInput_amountDelta_bound	Verified	*Verify Bound of Next Sqrt Price from Input Amount, ensuring the function correctly handles both zero-for-one and one-for-zero scenarios.*	*Prices are valid.*	*link to rule report*
getNextSqrtPriceFromOutput_amountDelta_bound	Verified	*Verify Bound of Next Sqrt Price from Output Amount and Delta, ensuring the function correctly handles both zero-for-one and one-for-zero scenarios.*	*Prices are valid.*	*link to rule report*
checkAxiomC	Verified	*This test ensures that the*	*Prices are valid.*	*link to rule report*

		`getNextSqrtPriceFromInput` and `getNextSqrtPriceFromOutput` functions correctly transition the square root price according to the expected directionality when given input and output amounts for both zero-for-one and one-for-zero swaps.		
getNextSqrtPriceFromInput_amountDelta_cannot_revert	Verified	*Ensure Non-Reversion of Amount Delta from Input Price Transition*		*link to rule report*
getNextSqrtPriceFromOutput_amountDelta_cannot_revert	Verified	*Ensure Non-Reversion of Amount Delta from Output Price Transition*		*link to rule report*

TickBitMap Library

Module General Assumptions

- The following properties are proven for src/libraries/TickBitmap.sol contract.

Contract Properties

P-01. Correctness of flipTick()	
Status: Verified	Property Assumptions:

Rule Name	Status	Description	Rule Assumptions	Link to rule report
flipTickEquivalence	Verified	*Equivalence of Solidity and assembly flipTick*	*Ticks and tick spacing are valid.*	
flipTickIntegrityTest	Verified	*Tick flipped after a call for flipTick.*		
flipTickAffectsOnlyTickTest	Verified	*Flipping a tick doesn't affect other ticks.*		

Disclaimer

The Certora Prover takes a contract and a specification as input and formally proves that the contract satisfies the specification in all scenarios. Notably, the guarantees of the Certora Prover are scoped to the provided specification and the Certora Prover does not check any cases not covered by the specification.

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.