Uniswap v4-core Security Review

Auditors

Desmond Ho, Lead Security Researcher

Kurt Barry, Lead Security Researcher

Saw-Mon and Natalie, Lead Security Researcher

Jeiwan, Security Researcher

David Chaparro, Junior Security Researcher

Report prepared by: Lucas Goiriz

September 5, 2024

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Uniswap is an open source decentralized exchange that facilitates automated transactions between ERC20 token tokens on various EVM-based chains through the use of liquidity pools and automatic market makers (AMM).

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of v4-core according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Disclaimer: *The current report is a* ***draft****. Fix review is still in progress for many issues and nothing in this report should be considered finalized.*

Over the course of 10 days in total, Uniswap engaged with Spearbit to review the v4-core protocol. In this period of time a total of **36** issues were found.

Summary

Project Name	Uniswap
Repository	v4-core
Commit	7a7203...a2c037
Type of Project	DeFi, AMM
Audit Timeline	Jul 15 to Aug 26
Two week fix period	Aug 26 - Sep 10

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	1	1	0
Low Risk	5	2	1
Gas Optimizations	12	9	2
Informational	18	12	1
Total	**36**	**24**	**4**

5 Findings

5.1 Medium Risk

5.1.1 Donations can be stolen by providing just-in-time liquidity

Severity: Medium Risk

Context: PoolManager.sol#L252, Pool.sol#L463-L468

Description: The PoolManager.donate() function allows to donate tokens to liquidity providers. Donations are counted as swap fees and immediately added to the global swap fees trackers (Pool.sol#L463-L468):

```
if (amount0 > 0) {
    state.feeGrowthGlobal0X128 += FullMath.mulDiv(amount0, FixedPoint128.Q128, liquidity);
}
if (amount1 > 0) {
    state.feeGrowthGlobal1X128 += FullMath.mulDiv(amount1, FixedPoint128.Q128, liquidity);
}
```

This increases the earned swap fees of all liquidity positions that include the current price.

Since donation amounts can be arbitrary (specifically, they can be significantly bigger than swap fees), this opens up an attack vector that allows anyone to steal a portion of donations by providing just-in-time liquidity. This can be exploited via a sandwich attack that wraps the donating transaction in two transactions:

1. In the preceding transaction, some amount of liquidity is added around the current price.
2. The donating transaction rewards LPs, including the position added in 1.
3. In the following transaction, the liquidity added in 1 is removed and a portion of the donation is withdrawn.

In this scenario, the attacker earns a portion of the donation while not providing useful liquidity to the pool.

Recommendation: Given that the core contracts strive to remain as simple and basic as possible, we recommend removing the `PoolManager.donate()` function and letting integrators implement their own donations solution via the hooks. Alternatively, consider keeping `PoolManager.donate()` and warning users that it should only be used for donating insignificant amounts (users would need to determine their size by themselves, ensuring their donations are not profitable for MEV bots). For bigger amounts, however, integrators will still need to implement a more robust solution using the hooks. E.g. donations can be vested (i.e. distributed over time), or LPs can be required to keep their liquidity for a minimum amount of time.

Uniswap: Comments have been added in PR 851.

Spearbit: Verified.

5.2 Low Risk

5.2.1 `tickSpacingToMaxLiquidityPerTick`'s calculation is not completely accurate

Severity: Low Risk

Context: Pool.sol#L574

Description: In the above context when `minTick` is calculated one compresses the `MIN_TICK` such that it would round towards `0` and not negative infinity. Where as one needs to apply the compression towards negative infinity.

And thus the result can be off by `1` in the denominator.

Also see the related issue "Incorrect tick compression for negative ticks in `countInitializedTicksLoaded`" for 'v4-periphery'.

Recommendation: For better estimate make sure the tick compressions are preformed correctly so they would round toward negative infinity.

Uniswap: Fixed in PR 870.

Spearbit: Verified.

5.2.2 Mixed use of rounding direction and inaccurate constants in `getSqrtPriceAtTick`

Severity: Low Risk

Context: TickMath.sol#L54-L108

Description: Let i be the tick provided, and below to be the binary represnetation of $|\ i\ |$:

$$|\ i\ | = b_{19} \cdots b_2 b_1 b_0$$

Note that 20 binary digits is enough since in the min and max range of the ticks we know that $|\ i\ | < 2^{20}$.

Let $h_i(b)$ be (where $b \in \{0, 1\}$):

$$h_i(b) = \left\lceil \frac{2^{128}}{\sqrt{1.0001}^{2^i \cdot b}} \right\rceil$$

$$h_0(1) = \left\lceil \frac{2^{128}}{\sqrt{1.0001}} \right\rceil = 340265354078544963557816517032075149314 = \texttt{0xfffcb933bd6fad37aa2d162d1a594002}$$

Also we know $h_i(0) = 2^{128}$. Let's define the $\otimes$ operator as the multiplication in `Q...x128` type:

$$a \otimes b = \left\lfloor \frac{a \cdot b}{2^{128}} \right\rfloor$$

Then we have:

$$h_i(0) \otimes a = a \otimes h_i(0) = a$$

and up to TickMath.sol#L96 the `price` p calculated becomes (order of applying the $\otimes$ operator matters below):

$$p_{19} = (h_{19}(b_{19}) \otimes \cdots (h_2(b_2) \otimes (h_1(b_1) \otimes p_0)) \cdots) = \bigotimes_{i=0}^{19} h_i(b_i)$$

$$\texttt{getSqrtPriceAtTick}(i) = \begin{cases} \left\lceil \frac{\left\lfloor \frac{2^{256}-1}{p_{19}} \right\rfloor}{2^{32}} \right\rceil, & \text{if } i > 0 \\ \left\lceil \frac{p_{19}}{2^{32}} \right\rceil, & \text{otherwise} \end{cases}$$

1. Note that we have:

$$\frac{1}{\sqrt{1.0001}^{|i|}} = \frac{1}{1.0001^{2^{19-1} \cdot b_{19}}} \times \cdots \times \frac{1}{1.0001^{2^{1-1} \cdot b_1}} \times \frac{1}{1.0001^{2^{0-1} \cdot b_0}}$$

   and thus p_{19} should be the above multiplication in `Q128x128` with `128` bits of precision and then at the end lowered to `Q128x96`.

2. The multiplactions $\otimes$ are rounded down although the (most) of the constants $h_i(1)$ used are rounded up.

3. The inversion for the postive ticks i is rounded down although besides multiplications $\otimes$ everything else is rounded up.

4. The inversion for positive ticks $i > 0$ is not accurate in `Q128x128` the inversion should have been (also rounded up if possible):

$$\frac{2^{128}}{p_{19}} \cdot 2^{128} = \frac{2^{256}}{p_{19}}$$

   But since one cannot use 2^{256} that is probably why the constant `not(0)` $2^{256} - 1$ is used instead.

   Let's assess the accurary of the constants used $h_i(1)$:

formula	wolfram value	value used in the code	used - actual
$h_0(1)$	`0xfffcb933bd6fad37aa2d162d1a594002`	`0xfffcb933bd6fad37aa2d162d1a594001`	-1
$h_1(1)$	`0xfff97272373d413259a46990580e213a`	`0xfff97272373d413259a46990580e213a`	0
$h_2(1)$	`0xfff2e50f5f656932ef12357cf3c7fdcc`	`0xfff2e50f5f656932ef12357cf3c7fdcc`	0
$h_3(1)$	`0xffe5caca7e10e4e61c3624eaa0941cd0`	`0xffe5caca7e10e4e61c3624eaa0941cd0`	0
$h_4(1)$	`0xffcb9843d60f6159c9db58835c926644`	`0xffcb9843d60f6159c9db58835c926644`	0
$h_5(1)$	`0xff973b41fa98c081472e6896dfb254c0`	`0xff973b41fa98c081472e6896dfb254c0`	0
$h_6(1)$	`0xff2ea16466c96a3843ec78b326b52861`	`0xff2ea16466c96a3843ec78b326b52861`	0
$h_7(1)$	`0xfe5dee046a99a2a811c461f1969c3053`	`0xfe5dee046a99a2a811c461f1969c3053`	0
$h_8(1)$	`0xfcbe86c7900a88aedcffc83b479aa3a4`	`0xfcbe86c7900a88aedcffc83b479aa3a4`	0
$h_9(1)$	`0xf987a7253ac413176f2b074cf7815e54`	`0xf987a7253ac413176f2b074cf7815e54`	0
$h_{10}(1)$	`0xf3392b0822b70005940c7a398e4b70f3`	`0xf3392b0822b70005940c7a398e4b70f3`	0
$h_{11}(1)$	`0xe7159475a2c29b7443b29c7fa6e889d9`	`0xe7159475a2c29b7443b29c7fa6e889d9`	0
$h_{12}(1)$	`0xd097f3bdfd2022b8845ad8f792aa5826`	`0xd097f3bdfd2022b8845ad8f792aa5825`	-1
$h_{13}(1)$	`0xa9f746462d870fdf8a65dc1f90e061e5`	`0xa9f746462d870fdf8a65dc1f90e061e5`	0
$h_{14}(1)$	`0x70d869a156d2a1b890bb3df62baf32f7`	`0x70d869a156d2a1b890bb3df62baf32f7`	0
$h_{15}(1)$	`0x31be135f97d08fd981231505542fcfa6`	`0x31be135f97d08fd981231505542fcfa6`	0
$h_{16}(1)$	`0x9aa508b5b7a84e1c677de54f3e99bc9`	`0x9aa508b5b7a84e1c677de54f3e99bc9`	0
$h_{17}(1)$	`0x5d6af8dedb81196699c329225ee605`	`0x5d6af8dedb81196699c329225ee604`	-1

formula	wolfram value	value used in the code	used - actual
$h_{18}(1)$	0x2216e584f5fa1ea926041bedfe97 (inaccurate)	0x2216e584f5fa1ea926041bedfe98	1
$h_{19}(1)$	0x48a170391f7dc42444e8fa2 (inaccurate)	0x48a170391f7dc42444e8fa2	0

formula	Sympy value	value used in the code	used - actual
$h_0(1)$	0xfffcb933bd6fad37aa2d162d1a594002	0xfffcb933bd6fad37aa2d162d1a594001	-1
$h_1(1)$	0xfff97272373d413259a46990580e213a	0xfff97272373d413259a46990580e213a	0
$h_2(1)$	0xfff2e50f5f656932ef12357cf3c7fdcc	0xfff2e50f5f656932ef12357cf3c7fdcc	0
$h_3(1)$	0xffe5caca7e10e4e61c3624eaa0941cd0	0xffe5caca7e10e4e61c3624eaa0941cd0	0
$h_4(1)$	0xffcb9843d60f6159c9db58835c926644	0xffcb9843d60f6159c9db58835c926644	0
$h_5(1)$	0xff973b41fa98c081472e6896dfb254c0	0xff973b41fa98c081472e6896dfb254c0	0
$h_6(1)$	0xff2ea16466c96a3843ec78b326b52861	0xff2ea16466c96a3843ec78b326b52861	0
$h_7(1)$	0xfe5dee046a99a2a811c461f1969c3053	0xfe5dee046a99a2a811c461f1969c3053	0
$h_8(1)$	0xfcbe86c7900a88aedcffc83b479aa3a4	0xfcbe86c7900a88aedcffc83b479aa3a4	0
$h_9(1)$	0xf987a7253ac413176f2b074cf7815e54	0xf987a7253ac413176f2b074cf7815e54	0
$h_{10}(1)$	0xf3392b0822b70005940c7a398e4b70f3	0xf3392b0822b70005940c7a398e4b70f3	0
$h_{11}(1)$	0xe7159475a2c29b7443b29c7fa6e889d9	0xe7159475a2c29b7443b29c7fa6e889d9	0
$h_{12}(1)$	0xd097f3bdfd2022b8845ad8f792aa5826	0xd097f3bdfd2022b8845ad8f792aa5825	-1
$h_{13}(1)$	0xa9f746462d870fdf8a65dc1f90e061e5	0xa9f746462d870fdf8a65dc1f90e061e5	0
$h_{14}(1)$	0x70d869a156d2a1b890bb3df62baf32f7	0x70d869a156d2a1b890bb3df62baf32f7	0
$h_{15}(1)$	0x31be135f97d08fd981231505542fcfa6	0x31be135f97d08fd981231505542fcfa6	0
$h_{16}(1)$	0x9aa508b5b7a84e1c677de54f3e99bc9	0x9aa508b5b7a84e1c677de54f3e99bc9	0
$h_{17}(1)$	0x5d6af8dedb81196699c329225ee605	0x5d6af8dedb81196699c329225ee604	-1
$h_{18}(1)$	0x2216e584f5fa1ea926041bedfe98	0x2216e584f5fa1ea926041bedfe98	0
$h_{19}(1)$	0x48a170391f7dc42444e8fa3	0x48a170391f7dc42444e8fa2	-1

- See below the `sympy` code to calculate the constants:

```
import sympy
from math import ceil

# values from the codebase
u = [
    0xfffcb933bd6fad37aa2d162d1a594001,
    0xfff97272373d413259a46990580e213a,
    0xfff2e50f5f656932ef12357cf3c7fdcc,
    0xffe5caca7e10e4e61c3624eaa0941cd0,
    0xffcb9843d60f6159c9db58835c926644,
    0xff973b41fa98c081472e6896dfb254c0,
    0xff2ea16466c96a3843ec78b326b52861,
    0xfe5dee046a99a2a811c461f1969c3053,
    0xfcbe86c7900a88aedcffc83b479aa3a4,
    0xf987a7253ac413176f2b074cf7815e54,
```

```
    0xf3392b0822b70005940c7a398e4b70f3,
    0xe7159475a2c29b7443b29c7fa6e889d9,
    0xd097f3bdfd2022b8845ad8f792aa5825,
    0xa9f746462d870fdf8a65dc1f90e061e5,
    0x70d869a156d2a1b890bb3df62baf32f7,
    0x31be135f97d08fd981231505542fcfa6,
    0x9aa508b5b7a84e1c677de54f3e99bc9,
    0x5d6af8dedb81196699c329225ee604,
    0x2216e584f5fa1ea926041bedfe98,
    0x48a170391f7dc42444e8fa2,
]

x = sympy.symbols("x")
g = (
    sympy.S('340282366920938463463374607431768211456') # 2 ** 128
    / (sympy.S('10001/10000') ** (2 ** (x - 1)))
)

a = [0 for _ in range(21)]

PREC = 1000

a[0] = g.evalf(PREC, subs={x: sympy.S(' 0.0')})
a[1] = g.evalf(PREC, subs={x: sympy.S(' 1.0')})
a[2] = g.evalf(PREC, subs={x: sympy.S(' 2.0')})
a[3] = g.evalf(PREC, subs={x: sympy.S(' 3.0')})
a[4] = g.evalf(PREC, subs={x: sympy.S(' 4.0')})
a[5] = g.evalf(PREC, subs={x: sympy.S(' 5.0')})
a[6] = g.evalf(PREC, subs={x: sympy.S(' 6.0')})
a[7] = g.evalf(PREC, subs={x: sympy.S(' 7.0')})
a[8] = g.evalf(PREC, subs={x: sympy.S(' 8.0')})
a[9] = g.evalf(PREC, subs={x: sympy.S(' 9.0')})
a[10] = g.evalf(PREC, subs={x: sympy.S('10.0')})
a[11] = g.evalf(PREC, subs={x: sympy.S('11.0')})
a[12] = g.evalf(PREC, subs={x: sympy.S('12.0')})
a[13] = g.evalf(PREC, subs={x: sympy.S('13.0')})
a[14] = g.evalf(PREC, subs={x: sympy.S('14.0')})
a[15] = g.evalf(PREC, subs={x: sympy.S('15.0')})
a[16] = g.evalf(PREC, subs={x: sympy.S('16.0')})
a[17] = g.evalf(PREC, subs={x: sympy.S('17.0')})
a[18] = g.evalf(PREC, subs={x: sympy.S('18.0')})
a[19] = g.evalf(PREC, subs={x: sympy.S('19.0')})
a[20] = g.evalf(PREC, subs={x: sympy.S('20.0')})

for i in range(20):
    b = int(ceil(a[i]))

    print("| $h_{{{3}}}(1)$ | `0x{0:x}` | `0x{1:x}` | ${2:d}$|".format(
        b,
        u[i],
        u[i] - b,
        i
    ))
```

5. and so the values $h_0(1), h_{12}(1), h_{17}(1), h_{19}(1)$ are off by 1.

Recommendations:

1. Fix or document why a mixed use of rounding down and up is used in this function. This could have been due to gas saving since one could just use right shifts for multiplication.
2. Adjust the constants used for $h_0(1), h_{12}(1), h_{17}(1), h_{19}(1)$. Note that with the adjusted constants (from the

Sympy table) the test suite still passes.

3. Add code comments like Aperture-Finance/uni-v3-lib
4. Provide details/proof as why the final value fits in uint160 (Q64x96).

   Warning: If 2. is applied the invariants should be checked again. Mainly that getSqrtPriceAtTick is sticktly increasing and also close to the actual value. And also its related invariants in relashionship to getTickAtSqrtPrice is also preserved.

Uniswap: Regarding 2. Some comments have been added to explain the rounding direction for $h_i(1)$ to the nearest integer value in PR 867.

Spearbit: Partially fixed and verified.

5.2.3 The used constants representing the min and max of the errors in getTickAtSqrtPrice are not accurate

Severity: Low Risk

Context: TickMath.sol#L259-L262, Logarithm Approximation Precision by ABDK

Description: In the above context we have:

```
int256 log_sqrt10001 = log_2 * 255738958999603826347141; // 128.128 number

int24 tickLow = int24((log_sqrt10001 - 3402992956809132418596140100660247210) >> 128);
int24 tickHi = int24((log_sqrt10001 + 291339464771989622907027621153398088495) >> 128);
```

Let:

$$\psi = \frac{255738958999603826347141}{2^{64}}$$

$$\psi - \frac{1}{\log_2 \sqrt{1.0001}} = 1.08830\cdots 10^{-20}$$

Let's calculate the rounded-down maximum error:

$$\lfloor 2^{128} \cdot \max(\epsilon_i) \rfloor = \left\lfloor 2^{128} \cdot \left(64 \left(\psi - \frac{1}{\log_2 \sqrt{1.0001}} \right) + log_{\sqrt{1.0001}} 1.0000005 \right) \right\rfloor$$

$$\lfloor 2^{128} \cdot \max(\epsilon_i) \rfloor = 340299295680913241859614010066024720 9$$

The above 340299295680913241859614010066024720 9 is derived by wolframalpha. The value used in the codebase is $\lceil 2^{128} \cdot \max(\epsilon_i) \rceil$ which differs by the correct value only by 1:

$$\lceil 2^{128} \cdot \max(\epsilon_i) \rceil = 340299295680913241859614010066024721 0$$

Let's calculate the rounded-down minimium error:

$$\lfloor 2^{128} \cdot \min(\epsilon_i) \rfloor = \left\lfloor 2^{128} \cdot \left(-96 \left(\psi - \frac{1}{\log_2 \sqrt{1.0001}} \right) + \psi \left(\frac{-1}{2^i} + \frac{3}{2} \left(2 - \frac{1}{2^{i-1}} \right) \log_2 \left(1 - \frac{1}{2^{127}} \right) \right) + log_{\sqrt{1.0001}} 0.9999995 \right) \right\rfloor$$

We are interested in $\lfloor 2^{128} \cdot \min(\epsilon_{14}) \rfloor$ since only 14 approximated terms are used:

$$\lfloor 2^{128} \cdot \min(\epsilon_{14}) \rfloor = -291339464771989623025533689748046440464$$

The above `-291339464771989623025533689748046440464` is derived by wolframalpha. The value used in the codebase is instead the following:

$$\left\lfloor 2^{128} \cdot \left(-64 \left(\psi - \frac{1}{\log_2 \sqrt{1.0001}} \right) + \psi \left(\frac{-1}{2^i} + \frac{3}{2} \left(2 - \frac{1}{2^{i-1}} \right) \log_2 \left(1 - \frac{1}{2^{127}} \right) \right) + log_{\sqrt{1.0001}} 0.9999995 \right) \right\rfloor$$

which equals to `-291339464771989622907027621153398088495`. The difference is, one should have used:

$$\lfloor 2^{128} \cdot (-96(\psi - \cdots) + \cdots) \rfloor$$

but instead the following is calculated:

$$\lfloor 2^{128} \cdot (-64(\psi - \cdots) + \cdots) \rfloor$$

This is error in using `64` instead of `96` comes from the Logarithm Approximation Precision by `ABDK` where in the calculations it is assumed that $x \in [2^{-64}, 2^{64})$, ie it is of type `Q64x64`. Note that x in that document correponds to `price` (P) which is:

```
uint256 price = uint256(sqrtPriceX96) << 32;
```

We know that `sqrtPriceX96` is of the type `Q64x96` and thus `price` is of the type `Q64x128` but since it is merely been multiplided by 2^{32} its range remains as $[2^{-96}, 2^{64})$. And this is why 64 needs to be used in the formula for $\max(\epsilon_i)$ and -96 for $\min(\epsilon_i)$.

`unchecked` **block safety:**

No overflow shoud occur in calculation of `log_sqrt10001` since `log_2` at the very end would be smaller than $65 \cdot 2^{64}$ and:

$$65 \cdot 2^{64} \cdot 255738958999603826347141 = 65 \cdot 2^{128} \cdot \psi < 2^{148}$$

and no underflow shold occur since `log_2`:

$$-96 \cdot 2^{64} \cdot 255738958999603826347141 = -96 \cdot 2^{128} \cdot \psi > -2^{149}$$

No overflow or unsafe casting should occur for `tickHi` since (with the old or new constant):

$$\frac{2^{148} + 291339464771989622907027621153398088495}{2^{128}} < 2^{21}$$

No underflow or unsafe casting should occur for `tickLow` since (with the old or new constant):

$$\frac{-2^{149} - 340299295680913241859614010066024721 0}{2^{128}} > -2^{22}$$

Recommendation: Apply the following patch:

```
diff --git a/src/libraries/TickMath.sol b/src/libraries/TickMath.sol
index 6e5f8417..7a1f58ca 100644
--- a/src/libraries/TickMath.sol
+++ b/src/libraries/TickMath.sol
@@ -107,11 +107,11 @@ library TickMath {
         }
     }

-    /// @notice Calculates the greatest tick value such that getPriceAtTick(tick) <= price
-    /// @dev Throws in case sqrtPriceX96 < MIN_SQRT_PRICE, as MIN_SQRT_PRICE is the lowest value
↪  getPriceAtTick may
+    /// @notice Calculates the greatest tick value such that getSqrtPriceAtTick(tick) <= sqrtPriceX96
+    /// @dev Throws in case sqrtPriceX96 < MIN_SQRT_PRICE, as MIN_SQRT_PRICE is the lowest value
↪  getSqrtPriceAtTick may
     /// ever return.
     /// @param sqrtPriceX96 The sqrt price for which to compute the tick as a Q64.96
-    /// @return tick The greatest tick for which the price is less than or equal to the input price
+    /// @return tick The greatest tick for which the getSqrtPriceAtTick(tick) is less than or equal to
↪  the input sqrtPriceX96
     function getTickAtSqrtPrice(uint160 sqrtPriceX96) internal pure returns (int24 tick) {
         unchecked {
             // Equivalent: if (sqrtPriceX96 < MIN_SQRT_PRICE || sqrtPriceX96 >= MAX_SQRT_PRICE) revert
↪  InvalidSqrtPrice();
@@ -256,10 +256,10 @@ library TickMath {
                 log_2 := or(log_2, shl(50, f))
             }

-            int256 log_sqrt10001 = log_2 * 255738958999603826347141; // 128.128 number
+            int256 log_sqrt10001 = log_2 * 255738958999603826347141; // Q22.128 number

-            int24 tickLow = int24((log_sqrt10001 - 3402992956809132418596140100660247210) >> 128);
-            int24 tickHi = int24((log_sqrt10001 + 291339464771989622907027621153398088495) >> 128);
+            int24 tickLow = int24((log_sqrt10001 - 3402992956809132418596140100660247209) >> 128);
+            int24 tickHi = int24((log_sqrt10001 + 291339464771989623025533689748046440464) >> 128);

             tick = tickLow == tickHi ? tickLow : getSqrtPriceAtTick(tickHi) <= sqrtPriceX96 ? tickHi :
↪  tickLow;
         }
```


Warning: The intervals provided by both the old and the new constant overlap almost entirely and measure around $0.8661\cdots$ in length. But on low side the old internal hangs out as much as $\frac{1}{2^{128}}$ and the new internal on the high side hangs out as much as $\frac{3.482\cdots}{10^{19}}$ and thus the result is that in some edge cases the current and the new implementation using the new constant might be off by **one tick**. Note that the current tests all pass with the new constants so these edge cases are not tested throughly.

Note: Moreover, one can use the borrowed `msb` calculation from `Solady` to replace the current calcution to save some gas:

```
assembly ("memory-safe") {
    let f := shl(7, gt(r, 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF))
    msb := or(msb, f)
    r := shr(f, r)
}
assembly ("memory-safe") {
    let f := shl(6, gt(r, 0xFFFFFFFFFFFFFFFF))
    msb := or(msb, f)
    r := shr(f, r)
}
assembly ("memory-safe") {
    let f := shl(5, gt(r, 0xFFFFFFFF))
```

```
    msb := or(msb, f)
    r := shr(f, r)
}
assembly ("memory-safe") {
    let f := shl(4, gt(r, 0xFFFF))
    msb := or(msb, f)
    r := shr(f, r)
}
assembly ("memory-safe") {
    let f := shl(3, gt(r, 0xFF))
    msb := or(msb, f)
    r := shr(f, r)
}
assembly ("memory-safe") {
    let f := shl(2, gt(r, 0xF))
    msb := or(msb, f)
    r := shr(f, r)
}
assembly ("memory-safe") {
    let f := shl(1, gt(r, 0x3))
    msb := or(msb, f)
    r := shr(f, r)
}
assembly ("memory-safe") {
    let f := gt(r, 0x1)
    msb := or(msb, f)
}
```

Appendix: `getTickAtSqrtPrice` works as following note that $\sqrt{p}$ is a symbolic value representing `sqrtPriceX96` which is of the type `Q64x96`:

1. Check $\sqrt{p} \in [\sqrt{p_{min}}, \sqrt{p_{max}})$.
2. Then $P = \sqrt{p} \cdot 2^{32}$ and thus it is of the type `Q64x128` and in the range $[2^{-96}, 2^{64})$.
3. Find the most significant bit of P and let's name it $n = \lfloor log_2 P \rfloor$.
4. r is taken to be:

$$r = r_0 = \left\lfloor \frac{P}{2^{\lfloor log_2 P \rfloor}} \cdot 2^{127} \right\rfloor \in [2^{127}, 2^{128})$$

and that is why multiplying r by itselft does not overflow (this also applies to the other iterations). We then have:

$$\left\lfloor \frac{r_0^2}{2^{127}} \right\rfloor \in [2^{127}, 2^{129})$$

I've marked the assembly block below so that we can follow the variable naming with subscripts:

```
assembly ("memory-safe") {
    r := shr(127, mul(r, r))            // r_{i-1} = r before the multiplication
    let f := shr(128, r)                // f_i     = f
    log_2 := or(log_2, shl(64 - i, f))  // L_i(P)  = L_{i-1}(P) | f_i * 2^{64-i}
    r := shr(f, r)                      // r_i     = the assginment value
}
```

and thus:

$$f = f_1 = \left\lfloor \log_2 \left(\frac{\left\lfloor \frac{r_0^2}{2^{127}} \right\rfloor}{2^{127}} \right) \right\rfloor = \lfloor \log_2 g(P) \rfloor \in \{0, 1\}$$

In the above the function $g(x)$ is defined as:

$$g(x) = \left\lfloor \left(\frac{\left\lfloor \frac{x}{2^{\lfloor log_2 x \rfloor}} \cdot 2^{127} \right\rfloor}{2^{127}} \right)^2 \cdot 2^{127} \right\rfloor 2^{-127}$$

and so r_1 is calculated as:

$$r_1 = \left\lfloor \frac{g(P)}{2^{\lfloor log_2 g(P) \rfloor}} \cdot 2^{127} \right\rfloor \in \left[2^{127}, 2^{128}\right)$$

and thus f_2 ends up being:

$$f_2 = \left\lfloor \log_2 \left(\frac{\left\lfloor \frac{r_1^2}{2^{127}} \right\rfloor}{2^{127}} \right) \right\rfloor = \lfloor \log_2 g(g(P)) \rfloor \in \{0, 1\}$$

and so the ith approximation of `log_2` using the $L_i(P)$ notation with 64 binary precision ends up being:

$$L_i(P) = \left\lfloor \log_2 \frac{P}{2^{128}} \right\rfloor \cdot 2^{64} + \sum_{k=1}^{i} f_k \cdot 2^{64-k} = \left\lfloor \log_2 \frac{P}{2^{128}} \right\rfloor \cdot 2^{64} \vee \left(\bigvee_{k=1}^{i} f_k \cdot 2^{64-k} \right)$$

Above one can do + or $\vee$ (bitwise or) since $f_k \in \{0, 1\}$.

Note that the approximation provided by the `ABDK` document matches with the above formula not taking into the consideration the precision factor 2^{64}:

$$L_i^{ABDK}(P) = \left\lfloor \log_2 \frac{P}{2^{128}} \right\rfloor + \sum_{k=1}^{i} \frac{1}{2^k} \lfloor \log_2 g(g(\cdots g(P))) \rfloor$$

where in the above summation the g function is composed k times.

For `getTickAtSqrtPrice`, $i = 14$ and so $L_{14}(P)$ is calculated. Also all approximations in this case $L_i(P)$ are of the type `Q8x64`. And so:

$$\texttt{logSqrt10001} = L_{14}(P) \cdot \psi \cdot 2^{64}$$

$\psi \cdot 2^{64}$ is of the type `Q14x64`, thus the above `logSqrt10001` is of the type `Q22x128`

5.2.4 `PoolManager.updateDynamicLPFee()` doesn't emit an event

Severity: Low Risk

Context: PoolManager.sol#L324

Description: The PoolManager.updateDynamicLPFee() function allows the hook contract to update the LP fee when it's dynamic. The fee is recorded in the contract storage, however, there's no even emitted to allow monitoring applications to detect the change.

Recommendation: Consider emitting an event in `PoolManager.updateDynamicLPFee()` to allow off-chain applications to track LP fee changes.

Uniswap: Decided against emitting an event when the dynamic fee is updated. This is because the override possibility for individual swaps would make it hard to track all of them off chain

Spearbit: Acknowledged.

5.2.5 `bubbleUpAndRevertWith` is prone to returndata bombing and some other minor issues

Severity: Low Risk

Context: CustomRevert.sol#L88, CustomRevert.sol#L91

Description/Recommendation:

- ☐ CustomRevert.sol#L88: copying the `returndata` to memory is prone to return data bombing and can revert with out of gas here. It would be best to first estimate to see if such an operation can happen with the current leftover gas and if so perform the copy or otherwise throw with a different generic error. For reference, please look at this implementation from `Seaport`.
- ☐ CustomRevert.sol#L91: use `shr` and `shl` instead of `div` and `mul` since the right hand side operands are `32`. The `solc` compiler might at some step in the optimisation do the replacement but it would be best to enforce it in the code.
- ☐ CustomRevert.sol#L91: allocating more than copied memory in the `revert` statement might use portion of the memory space which has already been filled by other data. If the `size` is being aligned to multiples of `32`. If this operation is necessary it would be best to also make sure the extra allocated memory space is cleaned.

5.3 Gas Optimization

5.3.1 A simple upcasting operation can be performed

Severity: Gas Optimization

Context: SqrtPriceMath.sol#L241-L245

Description: The contract uses inline assembly to perform a bitwise `AND` operation to restrict the `liquidity` value to 128 bits. However, this approach is unnecessarily complex and less readable compared to a simple upcasting operation.

```
uint256 _liquidity;
assembly ("memory-safe") {
    // avoid implicit upcasting
    _liquidity := and(liquidity, 0xffffffffffffffffffffffffffffffff)
}
```

Recommendation: Replace the assembly code with a simple upcasting operation in order to simplifies the code and provide a small gas optimization.

```
uint256 _liquidity = uint256(liquidity);
```

Uniswap: Fixed in PR 857.

Spearbit: Fixed.

5.3.2 `toId` performs an unnecesary length calculation

Severity: Gas Optimization

Context: PoolIdLibrary.sol#L12-L16

Description: `toId` function currently calculates the size of the `poolKey` struct in memory using the expression `mul(32, 5)`. While this is correct, it performs an unnecessary multiplication operation every time the function is called. Replacing this with a hardcoded value can save gas and make the intention clearer.

Recommendation: Replace the calculation `mul(32, 5)` with the hardcoded hexadecimal value `0xa0`, which is equivalent to 160 bytes (5×32). Additionally, add a comment explaining the memory layout of `PoolKey` structure.

Uniswap: Fixed in PR 857.

Spearbit: Verified.

5.3.3 `state.sqrtPriceX96` can be used instead of `slot0Start.sqrtPriceX96()` in `Pool.swap`

Severity: Gas Optimization

Context: Pool.sol#L319-L328

Description: In this context when the `params.sqrtPriceLimitX96` bounds are checked against `slot0Start.sqrtPriceX96()`, the storage slots are reread again. Although they also have been cached in memory in `state.sqrtPriceX96`.

Recommendation: Reuse `state.sqrtPriceX96` instead of reading from storage again:


```
diff --git a/src/libraries/Pool.sol b/src/libraries/Pool.sol
index 1a376354..7625e1f5 100644
--- a/src/libraries/Pool.sol
+++ b/src/libraries/Pool.sol
@@ -316,15 +316,15 @@ library Pool {
         if (params.amountSpecified == 0) return (BalanceDeltaLibrary.ZERO_DELTA, 0, swapFee, state);

         if (zeroForOne) {
-            if (params.sqrtPriceLimitX96 >= slot0Start.sqrtPriceX96()) {
-                PriceLimitAlreadyExceeded.selector.revertWith(slot0Start.sqrtPriceX96(),
↪  params.sqrtPriceLimitX96);
+            if (params.sqrtPriceLimitX96 >= state.sqrtPriceX96) {
+                PriceLimitAlreadyExceeded.selector.revertWith(state.sqrtPriceX96,
↪  params.sqrtPriceLimitX96);
             }
             if (params.sqrtPriceLimitX96 < TickMath.MIN_SQRT_PRICE) {
                 PriceLimitOutOfBounds.selector.revertWith(params.sqrtPriceLimitX96);
             }
         } else {
-            if (params.sqrtPriceLimitX96 <= slot0Start.sqrtPriceX96()) {
-                PriceLimitAlreadyExceeded.selector.revertWith(slot0Start.sqrtPriceX96(),
↪  params.sqrtPriceLimitX96);
+            if (params.sqrtPriceLimitX96 <= state.sqrtPriceX96) {
+                PriceLimitAlreadyExceeded.selector.revertWith(state.sqrtPriceX96,
↪  params.sqrtPriceLimitX96);
             }
             if (params.sqrtPriceLimitX96 >= TickMath.MAX_SQRT_PRICE) {
                 PriceLimitOutOfBounds.selector.revertWith(params.sqrtPriceLimitX96);
```

```
forge s --diff
```

```
test_swap_beforeSwapNoOpsSwap_exactInput() (gas: -2 (-0.000%))
test_swap_beforeSwapNoOpsSwap_exactOutput() (gas: -2 (-0.000%))
test_addLiquidity_succeedsWithHooksIfInitialized(uint160) (gas: 4 (0.000%))
test_removeLiquidity_succeedsWithHooksIfInitialized(uint160) (gas: 4 (0.001%))
test_swap_succeedsWithCorrectSelectors() (gas: 21 (0.001%))
test_swap_failsWithIncorrectSelectors() (gas: 21 (0.001%))
test_swap_withHooks_gas() (gas: 42 (0.001%))
test_swap_afterSwapFeeOnUnspecified_exactInput() (gas: 21 (0.002%))
test_swap_afterSwapFeeOnUnspecified_exactOutput() (gas: 21 (0.002%))
test_shouldSwapEqual(uint24,int24,int24,int24,int256,int256,int128,bool) (gas: 115 (0.002%))
test_swap_succeedsWithHooksIfInitialized() (gas: 21 (0.002%))
test_getFeeGrowthInside() (gas: 21 (0.003%))
test_fuzz_getTickLiquidity((int24,int24,int256,bytes32)) (gas: 9 (0.003%))
test_fuzz_getTickBitmap((int24,int24,int256,bytes32)) (gas: 9 (0.004%))
test_getTickInfo() (gas: 21 (0.004%))
test_getTickFeeGrowthOutside() (gas: 21 (0.004%))
test_getSlot0() (gas: 21 (0.004%))
test_getPositionInfo() (gas: 21 (0.005%))
test_swap_withDynamicFee_gas() (gas: 21 (0.005%))
test_dynamicReturnSwapFee_notStored() (gas: 21 (0.005%))
test_dynamicReturnSwapFee_notUsedIfPoolIsStaticFee() (gas: 21 (0.005%))
test_getFeeGrowthGlobals0() (gas: 21 (0.005%))
test_fuzz_nonZeroDeltaCount(uint256) (gas: 12 (0.006%))
test_getFeeGrowthGlobals1() (gas: 21 (0.006%))
test_swap_succeedsWithHook() (gas: 21 (0.009%))
test_nestedSwap() (gas: 21 (0.010%))
test_collectProtocolFees_ERC20_accumulateFees_gas() (gas: 21 (0.011%))
test_swap_99PercentFee_AmountOut_WithProtocol() (gas: 21 (0.011%))
test_collectProtocolFees_nativeToken_accumulateFees_gas() (gas: 21 (0.011%))
test_collectProtocolFees_ERC20_accumulateFees_exactOutput() (gas: 21 (0.011%))
test_collectProtocolFees_nativeToken_returnsAllFeesIf0IsProvidedAsParameter() (gas: 21 (0.011%))
test_collectProtocolFees_ERC20_returnsAllFeesIf0IsProvidedAsParameter() (gas: 21 (0.011%))
test_afterDonate_skipIfCalledByHook() (gas: 3000 (0.012%))
test_beforeDonate_skipIfCalledByHook() (gas: 3000 (0.012%))
test_swap_100PercentFee_AmountIn_WithProtocol() (gas: 21 (0.012%))
test_afterRemoveLiquidity_skipIfCalledByHook() (gas: 3000 (0.012%))
test_afterAddLiquidity_skipIfCalledByHook() (gas: 3000 (0.012%))
test_beforeAddLiquidity_skipIfCalledByHook() (gas: 3000 (0.012%))
test_beforeRemoveLiquidity_skipIfCalledByHook() (gas: 3000 (0.012%))
test_gas_beforeSwap_skipIfCalledByHook() (gas: 3042 (0.012%))
test_afterInitialize_skipIfCalledByHook() (gas: 3000 (0.012%))
test_beforeInitialize_skipIfCalledByHook() (gas: 3000 (0.012%))
test_emitsSwapFee() (gas: 21 (0.012%))
test_afterSwap_skipIfCalledByHook() (gas: 3084 (0.012%))
test_beforeSwap_skipIfCalledByHook() (gas: 3084 (0.012%))
test_swap_mint6909IfOutputNotTaken_gas() (gas: 21 (0.012%))
test_updateDynamicLPFee_beforeSwap_succeeds_gas() (gas: 21 (0.013%))
test_returnDynamicSwapFee_beforeSwap_succeeds_gas() (gas: 21 (0.013%))
test_swap_50PercentLPFee_AmountIn_NoProtocol() (gas: 21 (0.013%))
test_fuzz_getPositionInfo((int24,int24,int256,bytes32),uint256,bool) (gas: -79 (-0.013%))
test_swap_succeedsIfInitialized() (gas: 21 (0.013%))
test_swap_50PercentLPFee_AmountOut_NoProtocol() (gas: 21 (0.013%))
test_settle_withStartingBalance() (gas: 21 (0.014%))
test_swap_100PercentLPFee_AmountIn_NoProtocol() (gas: 21 (0.014%))
test_swap_succeedsWithNativeTokensIfInitialized() (gas: 21 (0.014%))
test_swap_helper_zeroForOne_exactInput() (gas: 21 (0.014%))
test_swap_helper_zeroForOne_exactOutput() (gas: 21 (0.014%))
test_fuzz_dynamicReturnSwapFee(uint24) (gas: 21 (0.014%))
test_swap_mint6909IfNativeOutputNotTaken_gas() (gas: 21 (0.014%))
test_swapNativeInput_helper_zeroForOne_exactOutput() (gas: 21 (0.015%))
test_swap_helper_oneForZero_exactOutput() (gas: 21 (0.015%))
```

```
test_swap_helper_oneForZero_exactInput() (gas: 21 (0.015%))
test_fuzz_getLiquidity((int24,int24,int256,bytes32)) (gas: 38 (0.015%))
test_ffi_fuzz_addLiquidity_defaultPool_ReturnsCorrectLiquidityDelta((int24,int24,int256,bytes32)) (gas:
↪  40 (0.015%))
test_swap_helper_native_zeroForOne_exactInput() (gas: 21 (0.015%))
test_swapNativeInput_helper_zeroForOne_exactInput() (gas: 21 (0.015%))
test_swap_succeeds() (gas: 21 (0.015%))
test_take_failsWithNoLiquidity() (gas: 3000 (0.015%))
test_swap_burn6909AsInput_gas() (gas: 42 (0.016%))
test_swapNativeInput_helper_oneForZero_exactOutput() (gas: 21 (0.016%))
test_swapNativeInput_helper_oneForZero_exactInput() (gas: 21 (0.016%))
test_swap_helper_native_oneForZero_exactOutput() (gas: 21 (0.016%))
test_swap_helper_native_oneForZero_exactInput() (gas: 21 (0.016%))
test_swap_gas() (gas: 21 (0.016%))
test_afterSwap_invalidReturn() (gas: 21 (0.017%))
test_swap_withNative_succeeds() (gas: 21 (0.017%))
test_swap_burnNative6909AsInput_gas() (gas: 42 (0.017%))
test_swap_withNative_gas() (gas: 21 (0.018%))
test_swap_againstLiqWithNative_gas() (gas: 42 (0.021%))
test_swap_againstLiquidity_gas() (gas: 42 (0.021%))
test_fuzz_getFeeGrowthInside((int24,int24,int256,bytes32),bool) (gas: 405 (0.067%))
test_fuzz_ProtocolAndLPFee(uint24,uint16,uint16,int256) (gas: 162 (0.081%))
test_fuzz_swap(uint160,uint24,uint16,uint16,(int24,bool,int256,uint160,uint24)) (gas: 26 (0.159%))
test_fuzz_getTickLiquidity_two_positions((int24,int24,int256,bytes32),(int24,int24,int256,bytes32))
↪  (gas: -763 (-0.182%))
test_fuzz_consecutiveExtsload(uint256,uint256,uint256) (gas: 2014 (0.221%))
test_fuzz_getPositionLiquidity((int24,int24,int256,bytes32),(int24,int24,int256,bytes32)) (gas: -1104
↪  (-0.253%))
test_shouldSwapEqualMultipleLP(uint24,int24,(int24,int24,int256)[],int256,int128,bool) (gas: -39552
↪  (-0.460%))
test_fuzz_extsload(uint256,uint256,bytes) (gas: 14346 (1.126%))
test_swap_accruesProtocolFees(uint16,uint16,int256) (gas: -11043 (-1.553%))
test_fuzz_collectProtocolFees(address,uint256,uint256) (gas: -9403 (-10.907%))
Overall gas change: -7252 (-0.002%)
```


Uniswap: Acknowledged. Recommendation not applied.

Spearbit: Acknowledged.

5.3.4 Unnecessary operations in `tickSpacingToMaxLiquidityPerTick` can be removed

Severity: Gas Optimization

Context: Pool.sol#L574-L577

Description/Recommendation: The calculation in this context can be simplified by removing the unnecessary multiplication and then division by `tickSpacing`:

```
let minTick := sdiv(MIN_TICK, tickSpacing)
let maxTick := sdiv(MAX_TICK, tickSpacing)
let numTicks := add(sub(maxTick, minTick), 1)
result := div(0xffffffffffffffffffffffffffffffff, numTicks)
```

```
forge s --diff
```

```
test_swap_withHooks_gas() (gas: -21 (-0.001%))
test_swap_succeedsWithCorrectSelectors() (gas: -21 (-0.001%))
test_donate_succeedsWithCorrectSelectors() (gas: -21 (-0.001%))
test_donate_failsWithIncorrectSelectors() (gas: -21 (-0.001%))
test_swap_failsWithIncorrectSelectors() (gas: -21 (-0.001%))
test_removeLiquidity_failsWithIncorrectSelectors() (gas: -21 (-0.001%))
test_addLiquidity_succeedsWithCorrectSelectors() (gas: -21 (-0.001%))
```

```
test_addLiquidity_withHooks_gas() (gas: -21 (-0.001%))
test_addLiquidity_failsWithIncorrectSelectors() (gas: -21 (-0.001%))
test_removeLiquidity_succeedsWithCorrectSelectors() (gas: -21 (-0.001%))
test_removeLiquidity_withHooks_gas() (gas: -21 (-0.001%))
test_swap_afterSwapFeeOnUnspecified_exactInput() (gas: -21 (-0.002%))
test_swap_afterSwapFeeOnUnspecified_exactOutput() (gas: -21 (-0.002%))
test_removeLiquidity_withFeeTakingHook() (gas: -21 (-0.002%))
test_fuzz_swap_beforeSwap_returnsDeltaSpecified(int128,int256,bool) (gas: -21 (-0.002%))
test_swap_beforeSwapNoOpsSwap_exactInput() (gas: -21 (-0.002%))
test_swap_beforeSwapNoOpsSwap_exactOutput() (gas: -21 (-0.002%))
test_shouldSwapEqual(uint24,int24,int24,int24,int256,int256,int128,bool) (gas: -113 (-0.002%))
test_swap_succeedsWithHooksIfInitialized() (gas: -21 (-0.002%))
test_addLiquidity_succeedsWithHooksIfInitialized(uint160) (gas: -18 (-0.002%))
test_removeLiquidity_succeedsWithHooksIfInitialized(uint160) (gas: -18 (-0.002%))
test_modifyLiquidity_sameSalt_differentLiquidityRouters_doNotEditSamePosition() (gas: -42 (-0.002%))
test_take_failsWithInvalidTokensThatDoNotReturnTrueOnTransfer() (gas: -21 (-0.002%))
test_addLiquidity_withFeeTakingHook() (gas: -42 (-0.003%))
test_afterInitialize_skipIfCalledByHook() (gas: -1013 (-0.004%))
test_beforeInitialize_skipIfCalledByHook() (gas: -1013 (-0.004%))
test_afterSwap_skipIfCalledByHook() (gas: -1034 (-0.004%))
test_beforeSwap_skipIfCalledByHook() (gas: -1034 (-0.004%))
test_afterDonate_skipIfCalledByHook() (gas: -1034 (-0.004%))
test_beforeDonate_skipIfCalledByHook() (gas: -1034 (-0.004%))
test_gas_beforeSwap_skipIfCalledByHook() (gas: -1034 (-0.004%))
test_afterRemoveLiquidity_skipIfCalledByHook() (gas: -1097 (-0.004%))
test_afterAddLiquidity_skipIfCalledByHook() (gas: -1097 (-0.004%))
test_beforeAddLiquidity_skipIfCalledByHook() (gas: -1097 (-0.004%))
test_beforeRemoveLiquidity_skipIfCalledByHook() (gas: -1097 (-0.004%))
test_getPositionInfo() (gas: -21 (-0.005%))
test_swap_withDynamicFee_gas() (gas: -21 (-0.005%))
test_beforeAfterRemoveLiquidity_calledWithZeroLiquidityDelta() (gas: -21 (-0.005%))
test_fuzz_getLiquidity((int24,int24,int256,bytes32)) (gas: -13 (-0.005%))
test_take_failsWithNoLiquidity() (gas: -1011 (-0.005%))
test_dynamicReturnSwapFee_notStored() (gas: -21 (-0.005%))
test_dynamicReturnSwapFee_notUsedIfPoolIsStaticFee() (gas: -21 (-0.005%))
test_getFeeGrowthGlobals0() (gas: -21 (-0.005%))
test_getFeeGrowthGlobals1() (gas: -21 (-0.006%))
test_beforeAfterAddLiquidity_beforeAfterRemoveLiquidity_succeedsWithHook() (gas: -21 (-0.006%))
test_ffi_addLiqudity_weirdPool_0_returnsCorrectLiquidityDelta() (gas: -21 (-0.006%))
test_beforeAfterRemoveLiquidity_calledWithPositiveLiquidityDelta() (gas: -21 (-0.007%))
test_settle_withNoStartingBalance() (gas: -21 (-0.007%))
test_getFeeGrowthInside() (gas: -42 (-0.007%))
test_getTickLiquidity() (gas: -21 (-0.007%))
test_getTickBitmap() (gas: -21 (-0.007%))
test_getPositionLiquidity() (gas: -21 (-0.007%))
test_gas_modifyLiquidity_newPosition() (gas: -21 (-0.007%))
test_getTickInfo() (gas: -42 (-0.008%))
test_getTickFeeGrowthOutside() (gas: -42 (-0.008%))
test_beforeAfterAddLiquidity_calledWithPositiveLiquidityDelta() (gas: -21 (-0.008%))
test_getSlot0() (gas: -42 (-0.008%))
test_addLiquidity_6909() (gas: -21 (-0.008%))
test_nestedRemoveLiquidity() (gas: -21 (-0.008%))
test_removeLiquidity_6909() (gas: -21 (-0.008%))
test_ffi_addLiqudity_weirdPool_1_returnsCorrectLiquidityDelta() (gas: -21 (-0.008%))
test_afterRemoveLiquidity_invalidReturn() (gas: -21 (-0.009%))
test_nestedAddLiquidity() (gas: -21 (-0.009%))
test_beforeRemoveLiquidity_invalidReturn() (gas: -21 (-0.009%))
test_getLiquidity() (gas: -42 (-0.010%))
test_removeLiquidity_someLiquidityRemains_gas() (gas: -21 (-0.011%))
test_modifyLiquidity_samePosition_withSalt_isUpdated() (gas: -42 (-0.012%))
test_modifyLiquidity_samePosition_zeroSalt_isUpdated() (gas: -42 (-0.012%))
test_removeLiquidity_gas() (gas: -17 (-0.012%))
```

```
test_gas_modifyLiquidity_updateSamePosition_withSalt() (gas: -42 (-0.012%))
test_ffi_fuzz_addLiquidity_defaultPool_ReturnsCorrectLiquidityDelta((int24,int24,int256,bytes32)) (gas:
↪  -33 (-0.013%))
test_fuzz_getTickLiquidity((int24,int24,int256,bytes32)) (gas: -33 (-0.013%))
test_modifyLiquidity_sameTicks_withDifferentSalt_isNotUpdated() (gas: -60 (-0.013%))
test_fuzz_getTickBitmap((int24,int24,int256,bytes32)) (gas: -33 (-0.013%))
test_addLiquidity_gas() (gas: -21 (-0.013%))
test_addLiquidity_succeedsIfInitialized(uint160) (gas: -21 (-0.014%))
test_addLiquidity_succeedsForNativeTokensIfInitialized(uint160) (gas: -21 (-0.014%))
test_addLiquidity_withNative_gas() (gas: -21 (-0.014%))
test_afterAddLiquidity_invalidReturn() (gas: -21 (-0.014%))
test_addLiquidity_succeeds() (gas: -21 (-0.015%))
test_shouldSwapEqualMultipleLP(uint24,int24,(int24,int24,int256)[],int256,int128,bool) (gas: 1767
↪  (0.021%))
test_addLiquidity_secondAdditionSameRange_gas() (gas: -42 (-0.022%))
test_fuzz_getTickLiquidity_two_positions((int24,int24,int256,bytes32),(int24,int24,int256,bytes32))
↪  (gas: -135 (-0.032%))
test_fuzz_ProtocolAndLPFee(uint24,uint16,uint16,int256) (gas: 141 (0.070%))
test_fuzz_getFeeGrowthInside((int24,int24,int256,bytes32),bool) (gas: -462 (-0.076%))
test_fuzz_getPositionLiquidity((int24,int24,int256,bytes32),(int24,int24,int256,bytes32)) (gas: -364
↪  (-0.083%))
testTick_tickSpacingToParametersInvariants_fuzz(int24) (gas: -24 (-0.224%))
test_fuzz_tickSpacingToMaxLiquidityPerTick(int24) (gas: -21 (-0.240%))
test_fuzz_initialize((address,address,uint24,int24,address),uint160) (gas: 45 (0.275%))
test_fuzz_getPositionInfo((int24,int24,int256,bytes32),uint256,bool) (gas: 2642 (0.443%))
test_swap_accruesProtocolFees(uint16,uint16,int256) (gas: -11106 (-1.562%))
Overall gas change: -21941 (-0.005%)
```

Uniswap: Fixed in PR 823.

Spearbit: Verified.

5.3.5 Deriving `liquidityGrossBefore` can be optimised

Severity: Gas Optimization

Context: Pool.sol#L523

Description/Recommendation: It is cheaper to mask a value by using `and` than shifting left then right:

```
uint256 internal constant LIQUIDITY_GROSS_MASK = 0xffffffffffffffffffffffffffffffff;
// ...
liquidityGrossBefore := and(liquidity, LIQUIDITY_GROSS_MASK)
```

Uniswap: Usage of the assembly block has been removed in PR 827.

Spearbit: Verified since the optimisation does not apply anymore.

5.3.6 `msg.sender` can be inlined in `_burnFrom` to save gas

Severity: Gas Optimization

Context: ERC6909Claims.sol#L14-L19

Description/Recommendation: `msg.sender` can be inlined in `_burnFrom` to save gas to avoid using the `sender` stack variable:

```
function _burnFrom(address from, uint256 id, uint256 amount) internal {
    if (from != msg.sender && !isOperator[from][msg.sender]) {
        uint256 senderAllowance = allowance[from][msg.sender][id];
        if (senderAllowance != type(uint256).max) {
            allowance[from][msg.sender][id] = senderAllowance - amount;
        }
    }
    _burn(from, id, amount);
}
```

```
forge snapshot --diff
```

```
test_addLiquidity_succeedsWithHooksIfInitialized(uint160) (gas: 5 (0.001%))
test_removeLiquidity_succeedsWithHooksIfInitialized(uint160) (gas: 5 (0.001%))
test_fuzz_getTickLiquidity((int24,int24,int256,bytes32)) (gas: 9 (0.003%))
test_fuzz_getTickBitmap((int24,int24,int256,bytes32)) (gas: 9 (0.004%))
test_ffi_fuzz_addLiquidity_defaultPool_ReturnsCorrectLiquidityDelta((int24,int24,int256,bytes32)) (gas:
↪  10 (0.004%))
test_fuzz_getPositionLiquidity((int24,int24,int256,bytes32),(int24,int24,int256,bytes32)) (gas: 17
↪  (0.004%))
test_shouldSwapEqual(uint24,int24,int24,int24,int256,int256,int128,bool) (gas: 287 (0.005%))
test_fuzz_getLiquidity((int24,int24,int256,bytes32)) (gas: 29 (0.012%))
test_fuzz_getTickLiquidity_two_positions((int24,int24,int256,bytes32),(int24,int24,int256,bytes32))
↪  (gas: -79 (-0.019%))
test_fuzz_getFeeGrowthInside((int24,int24,int256,bytes32),bool) (gas: 232 (0.038%))
test_shouldSwapEqualMultipleLP(uint24,int24,(int24,int24,int256)[],int256,int128,bool) (gas: -4712
↪  (-0.055%))
test_fuzz_nextInitializedTickWithinOneWord(int24,bool) (gas: -75 (-0.108%))
test_fuzz_extsload(uint256,uint256,bytes) (gas: 7173 (0.563%))
test_fuzz_getPositionInfo((int24,int24,int256,bytes32),uint256,bool) (gas: 7118 (1.194%))
test_swap_accruesProtocolFees(uint16,uint16,int256) (gas: -11064 (-1.556%))
Overall gas change: -1036 (-0.000%)
```


Uniswap: We don't think this approach would improve gas costs.

Spearbit: Acknowledged.

5.3.7 `_fetchProtocolFee` can be optimised by using the scratch space

Severity: Gas Optimization

Context: ProtocolFees.sol#L88-L93

Description: If `success` is `true` then we know that the `returndatasize()` should be `32` so we can copy the returned value to the first memory slot in the scratch space to save on gas cost.

Recommendation: Avoid using the free memory point and instead use the scratch space to copy and use the returned value:

```
if success {
    returndatacopy(0, 0, 32)
    returnData := mload(0)
}
```

```
forge snapshot --diff
```

```
test_swap_withHooks_gas() (gas: -11 (-0.000%))
test_swap_succeedsWithCorrectSelectors() (gas: -11 (-0.000%))
test_donate_succeedsWithCorrectSelectors() (gas: -11 (-0.000%))
test_donate_failsWithIncorrectSelectors() (gas: -11 (-0.000%))
test_swap_failsWithIncorrectSelectors() (gas: -11 (-0.000%))
```

```
test_removeLiquidity_failsWithIncorrectSelectors() (gas: -11 (-0.000%))
test_addLiquidity_succeedsWithCorrectSelectors() (gas: -11 (-0.000%))
test_addLiquidity_withHooks_gas() (gas: -11 (-0.000%))
test_addLiquidity_failsWithIncorrectSelectors() (gas: -11 (-0.000%))
test_removeLiquidity_succeedsWithCorrectSelectors() (gas: -11 (-0.000%))
test_removeLiquidity_withHooks_gas() (gas: -11 (-0.000%))
test_initialize_failsWithIncorrectSelectors() (gas: -11 (-0.000%))
test_initialize_succeedsWithCorrectSelectors() (gas: -11 (-0.000%))
test_initialize_succeedsWithEmptyHooks(uint160) (gas: -11 (-0.000%))
test_swap_afterSwapFeeOnUnspecified_exactInput() (gas: -11 (-0.001%))
test_swap_afterSwapFeeOnUnspecified_exactOutput() (gas: -11 (-0.001%))
test_addLiquidity_withFeeTakingHook() (gas: -11 (-0.001%))
test_removeLiquidity_withFeeTakingHook() (gas: -11 (-0.001%))
test_fuzz_swap_beforeSwap_returnsDeltaSpecified(int128,int256,bool) (gas: -11 (-0.001%))
test_swap_beforeSwapNoOpsSwap_exactInput() (gas: -11 (-0.001%))
test_swap_beforeSwapNoOpsSwap_exactOutput() (gas: -11 (-0.001%))
test_swap_succeedsWithHooksIfInitialized() (gas: -11 (-0.001%))
test_take_failsWithInvalidTokensThatDoNotReturnTrueOnTransfer() (gas: -11 (-0.001%))
test_addLiquidity_succeedsWithHooksIfInitialized(uint160) (gas: -11 (-0.001%))
test_removeLiquidity_succeedsWithHooksIfInitialized(uint160) (gas: -11 (-0.001%))
test_initialize_succeedsWithHooks(uint160) (gas: -11 (-0.002%))
test_swap_withDynamicFee_gas() (gas: -11 (-0.003%))
test_dynamicReturnSwapFee_notStored() (gas: -11 (-0.003%))
test_dynamicReturnSwapFee_notUsedIfPoolIsStaticFee() (gas: -11 (-0.003%))
test_afterSwap_skipIfCalledByHook() (gas: -824 (-0.003%))
test_beforeSwap_skipIfCalledByHook() (gas: -824 (-0.003%))
test_afterDonate_skipIfCalledByHook() (gas: -824 (-0.003%))
test_beforeDonate_skipIfCalledByHook() (gas: -824 (-0.003%))
test_afterRemoveLiquidity_skipIfCalledByHook() (gas: -824 (-0.003%))
test_afterAddLiquidity_skipIfCalledByHook() (gas: -824 (-0.003%))
test_beforeAddLiquidity_skipIfCalledByHook() (gas: -824 (-0.003%))
test_beforeRemoveLiquidity_skipIfCalledByHook() (gas: -824 (-0.003%))
test_gas_beforeSwap_skipIfCalledByHook() (gas: -824 (-0.003%))
test_ffi_addLiqudity_weirdPool_0_returnsCorrectLiquidityDelta() (gas: -11 (-0.003%))
test_afterInitialize_skipIfCalledByHook() (gas: -835 (-0.003%))
test_beforeInitialize_skipIfCalledByHook() (gas: -835 (-0.003%))
test_fuzz_getTickLiquidity((int24,int24,int256,bytes32)) (gas: 9 (0.003%))
test_settle_withNoStartingBalance() (gas: -11 (-0.003%))
test_fuzz_getTickBitmap((int24,int24,int256,bytes32)) (gas: 9 (0.004%))
test_take_failsWithNoLiquidity() (gas: -811 (-0.004%))
test_ffi_addLiqudity_weirdPool_1_returnsCorrectLiquidityDelta() (gas: -11 (-0.004%))
test_shouldSwapEqual(uint24,int24,int24,int24,int256,int256,int128,bool) (gas: 305 (0.006%))
test_fetchProtocolFee_outOfBounds() (gas: -11 (-0.006%))
test_fetchProtocolFee_overflowFee() (gas: -11 (-0.007%))
test_initialize_succeedsWithHook() (gas: -11 (-0.008%))
test_callHook_revertsWithInternalErrorFailedHookCall() (gas: -11 (-0.008%))
test_nestedInitialize() (gas: -11 (-0.009%))
test_initialize_forNativeTokens(uint160) (gas: -6 (-0.010%))
test_donate_failsIfNoLiquidity(uint160) (gas: -11 (-0.011%))
test_callHook_revertsWithBubbleUp() (gas: -11 (-0.012%))
test_afterInitialize_invalidReturn() (gas: -11 (-0.013%))
test_fuzz_getLiquidity((int24,int24,int256,bytes32)) (gas: 33 (0.013%))
test_initialize_fetchFeeWhenController(uint24) (gas: -11 (-0.013%))
test_ffi_fuzz_addLiquidity_defaultPool_ReturnsCorrectLiquidityDelta((int24,int24,int256,bytes32)) (gas:
↪  40 (0.015%))
test_updateDynamicLPFee_afterInitialize_initializesFee() (gas: -11 (-0.015%))
test_initialize_succeedsWithOverflowFeeController(uint160) (gas: -11 (-0.016%))
test_initialize_succeedsWithOutOfBoundsFeeController(uint160) (gas: -11 (-0.016%))
test_initialize_initializesFeeTo0() (gas: -11 (-0.016%))
test_updateDynamicLPFee_revertsIfPoolHasStaticFee() (gas: -11 (-0.016%))
test_updateDynamicLPFee_afterInitialize_failsWithTooLargeFee() (gas: -11 (-0.016%))
test_initialize_succeedsWithMaxTickSpacing(uint160) (gas: -11 (-0.017%))
```

```
test_dynamicReturnSwapFee_initializeZeroSwapFee() (gas: -11 (-0.019%))
test_initialize_gas() (gas: -11 (-0.019%))
test_fetchProtocolFee_succeeds() (gas: -11 (-0.022%))
test_fuzz_getPositionInfo((int24,int24,int256,bytes32),uint256,bool) (gas: 151 (0.025%))
test_initialize_revertsWhenPoolAlreadyInitialized(uint160) (gas: -25 (-0.041%))
test_fuzz_getPositionLiquidity((int24,int24,int256,bytes32),(int24,int24,int256,bytes32)) (gas: 285
↪  (0.065%))
test_fuzz_ProtocolAndLPFee(uint24,uint16,uint16,int256) (gas: 141 (0.070%))
test_fuzz_getFeeGrowthInside((int24,int24,int256,bytes32),bool) (gas: 476 (0.079%))
test_fuzz_nextInitializedTickWithinOneWord(int24,bool) (gas: -75 (-0.108%))
test_fuzz_swap(uint160,uint24,uint16,uint16,(int24,bool,int256,uint160,uint24)) (gas: 26 (0.159%))
test_shouldSwapEqualMultipleLP(uint24,int24,(int24,int24,int256)[],int256,int128,bool) (gas: -19223
↪  (-0.224%))
test_fuzz_extsload(uint256,uint256,bytes) (gas: 7173 (0.563%))
test_swap_accruesProtocolFees(uint16,uint16,int256) (gas: -11497 (-1.617%))
test_fuzz_getTickLiquidity_two_positions((int24,int24,int256,bytes32),(int24,int24,int256,bytes32))
↪  (gas: 16970 (4.059%))
test_fuzz_collectProtocolFees(address,uint256,uint256) (gas: -11601 (-13.457%))
Overall gas change: -27267 (-0.007%)
```

Uniswap: Different optimisation applied in PR 825.

Spearbit: The new approach also looks cheaper, one still needs to measure by how much.

5.3.8 Gas optimization in `clear()` function

Severity: Gas Optimization

Context: PoolManager.sol#L303

Description: Because the `amount` argument to `clear()` is non-negative, the `amountDelta` value obtained by safe-casting `amount` to `int128` is also non-negative, and thus the negation of `amountDelta` cannot overflow. Therefore, an `unchecked` block could be used here to reduce gas usage and bytecode size, consistent with what is done in other functions like `take` and `mint`.

Recommendation: Put the line containing the negation within an `unchecked` block:

```
+ unchecked {
        _accountDelta(currency, -(amountDelta), msg.sender);
+ }
```

Uniswap: Fixed in PR 826.

Spearbit: Fix verified.

5.3.9 Non-assembly version of `state.tick` setter possibly more gas efficient

Severity: Gas Optimization

Context: Pool.sol#L415-L422

Description: The non-assembly version of the setting of `state.tick` seems to be more efficient than the current implementation.

```
unchecked {
  int24 _zeroForOne = zeroForOne ? int24(1) : int24(0);
  state.tick = step.tickNext - _zeroForOne;
}
```

Recommendation: In addition to adopting the above recommendation, revisit assembly blocks and re-test to see if their non-assembly counterparts could be more efficient. This could possibly be due to the number of optimizer runs with the IR optimizer.

Uniswap: Fixed in PR 827.

Spearbit: Fixed.

5.3.10 `mulDiv()` is redundant for fee growth calculation

Severity: Gas Optimization

Context: Pool.sol#L391-L393, Pool.sol#L463-L468

Description: In `donate()`, `amount0` and `amount1` are safely casted to `int128`. As such, `FullMath.mulDiv()` isn't required because `amount * Q128 <= type(int128).max) * Q128 = 0x7fffffffffffffffffffffffffffffff00000000000000000000000000000000 < type(uint256).max`, ie. the intermediate value will not overflow `uint256`.

Therefore the calculation could use native operands, or a simplified version of `mulDiv`:

```
function simpleMulDiv(uint256 a, uint256 b, uint256 denominator) internal pure returns (uint256 result)
↪  {
  assembly ("memory-safe") {
    result := div(mul(a, b), denominator)
  }
}
```

Under the assumption that supported tokens can have a maximum supply of `type(uint128).max`, the same can be applied in `swap()` when incrementing fee growth global.

Recommendation: Replace `FullMath.mulDiv()` with a simplified and more gas efficient version for the referenced lines.

Uniswap: Fixed in PR 844.

Spearbit: Fixed.

5.3.11 More efficient mask derivation in `TickBitmap`

Severity: Gas Optimization

Context: TickBitmap.sol#L96-L97

Description: The `mask` derivation has been modified from UniswapV3 to be slightly more efficient:

```
  // UniV3
- uint256 mask = (1 << bitPos) - 1 + (1 << bitPos);
  // = 2 * (1 << bitPos) - 1
  // = (1 << bitPos + 1) - 1
  // = UniV4
+ uint256 mask = (1 << (uint256(bitPos) + 1)) - 1;
```

This can be further optimised to `uint256 mask = type(uint256).max >> (uint256(type(uint8).max) - bitPos);`, which is 1 operand less. Essentially, it's doing SHR of the full mask by `255 - bitPos` bits.

Recommendation:

```
- uint256 mask = (1 << (uint256(bitPos) + 1)) - 1;
+ uint256 mask = type(uint256).max >> (uint256(type(uint8).max) - bitPos);
```

Uniswap: Fixed in PR 828.

Spearbit: Fixed.

5.3.12 BitMath

Severity: Gas Optimization

Context: BitMath.sol#L16, BitMath.sol#L23

Description:

1. `mostSignificantBit` can be slightly optimised since we require that `x > 0` and so `shl(8, iszero(x))` would just be `0`.
2. The constant `0x0706060506020504060203020504030106050205030304010505030400000000` which is used as a lookup bitmap is slightly different from how one would construct it. The assumes that the following can include values `7` and `14` which is not true:

```
A = 0x8421084210842108cc6318c6db6d54be
B = and(0x1f, shr(shr(r, x), A))
```

In the above snippet `shr(r, x)` would have at most `8` bits and thus shifting `0x8421084210842108cc6318c6db6d54be` to the right by the `shr(r, x)` amount and then masking by `0x1f` which picks the least 5 bits of the shifted value gives us the following table:

most significant bit of `shr(r, x)`	possible values of `B` in binary	binary portion of `A` which is relev
`0b 111`	`00000`	`00..00`
`0b 110`	`00001, 00010, 00100, 01000, 10000`	`10000100001000010000100001000`
`0b 101`	`00110, 00011, 10001, 11000, 01100, 10011, 11001`	`11001100011000110001100011000`
`0b 100`	`01101, 10110, 11011`	`1101101101101101`
`0b 011`	`10100, 01010, 10101, 11010`	`01010100`
`0b 010`	`01011, 00101, 10010, 01001`	`1011`
`0b 001`	`01111, 10111`	`11`
`0b 000`	`11111`	`1`
`0b .`	`11110`	`0`

> **Note:** In the above table the symbol `0b .` represents the case/state that the code never ends up at but it is included for the sake of completeness. This is when `shr(r, x) == 0` aka when `x == 0` but we never end up at this case since we have the `require(x > 0)` statement.

And so the set of possible values of `B` does not include `7` (`00111`) or `14` (`01110`). And so the `7`th or `14`th byte of `0x0706060506020504060203020504030106050205030304010505030400000000` is never queried:

```
C = 0x0706060506020504060203020504030106050205030304010505030400000000
byte(B, C)
```

And that is why the `7`th and `14`th bytes of `C` can be any value and it would be best to just set them as `00`.

```
"""
suggested value
0x07060605060205_00_060203020504_00_0106050205030304010505030400000000
00000111 00000110 00000110 00000101 00000110 00000010  00000101 [00000000]
00000110 00000010 00000011 00000010 00000101 00000100 [00000000] 00000001
00000110 00000101 00000010 00000101 00000011 00000011  00000100  00000001
00000101 00000101 00000011 00000100 00000000 00000000  00000000  00000000

current value
0x07060605060205_04_060203020504_03_0106050205030304010505030400000000
00000111 00000110 00000110 00000101 00000110 00000010  00000101 [00000100]
00000110 00000010 00000011 00000010 00000101 00000100 [00000011] 00000001
00000110 00000101 00000010 00000101 00000011 00000011  00000100  00000001
00000101 00000101 00000011 00000100 00000000 00000000  00000000  00000000
"""
```

Proof of concept: See the following Python code to verify and construct different constants:

```
import re

MAX_RANGE = 1 << 256

### LSB
print("\n--- LSB ---\n")

m = 0xb6db6db6ddddddddd34d34d349249249210842108c6318c639ce739cfffffff

"""
    1000 0000 0100 0000 0100 0000 0101 0101
    0100 0011 0000 0000 0101 0010 0110 0110
    0100 0100 0011 0010 0000 0000 0000 0000
    0101 0000 0010 0000 0110 0001 0000 0110
    0111 0100 0000 0101 0011 0000 0010 0110
    0000 0010 0000 0000 0000 0000 0001 0000
    0111 0101 0000 0110 0010 0000 0000 0001
    0111 0110 0001 0001 0111 0000 0111 0111
"""
L = 0x804040554300526644320000502061067405302602000010750620017611707 7


patterns = [set() for _ in range(8)]

for i in range(256):
    block = i // 32
    pattern = ((m << i) % MAX_RANGE) >> 250
    patterns[block].add(pattern)

for i in range(8):
    s = f'block {i:03b} : ' + ', '.join([f'{p:06b}' for p in patterns[i]])

    print(s)

for i in range(8):
    for j in range(8):
        if i == j:
            continue
        intersection = patterns[i].intersection(patterns[j])
        if len(intersection) != 0:
            print(f'collision ({i}, {j}): {intersection}')

for i in range(8):
```

```
    for p in patterns[i]:
        b = ((L << (p << 2)) % MAX_RANGE) >> 252
        if b != i:
            print(f"error on block {i} and pattern {p:06b}")

# make sure L is computed correctly.
h = 1 << 255
for i in range(8):
    for p in patterns[i]:
        if p == 0:
            print('0 pattern detected')
        h |= i << ((256 - 4) - (p << 2))

print(f"h == L: {h == L}")

# 11010111011001000101001111100000
m2 = 0xd76453e0
pattern2 = []
L2 = 0x001f0d1e100c1d070f090b19131c1706010e11080a1a141802121b1503160405
h2 = 0

# make sure L2 is computed correctly.
for i in range(0, 32):
    p = (m2 >> i) & 31
    pattern2.append(p)
    h2 |= i << ((256 - 8) - (p << 3))

print(pattern2)
print(f"h2 == L2: {h2 == L2}")

### MSB

print("\n--- MSB ---\n")

"""
7 - 00..00 - 00000
6 - 1000010000100001000010000100001000010000100001000010000100001000 - 00001, 00010, 00100, 01000, 10000
5 - 1100110001100011000110001100011000110 - 00110, 00011, 10001, 11000, 01100, 10011, 11001
4 - 1101101101101101 - 01101, 10110, 11011,
3 - 01010100 - 10100, 01010, 10101, 11010
2 - 1011 - 01011, 00101, 10010, 01001
1 - 11 - 01111, 10111
0 - 1 - 11111
. - 0 - 11110
"""
m3 = 0x8421084210842108cc6318c6db6d54be

"""
7 -  0 /
6 -  1,  2,  4,  8, 16 /
5 -  3,  6, 12, 17, 19, 24, 25 /
4 -  7, 13, 22, 27                  ?? ( 7)
3 - 10, 14, 20, 21, 26              ?? (14)
2 -  5,  9, 11, 18 /
1 - 15, 23 /
0 - 28, 29, 30, 31
. - ??
"""
L3 = 0x0706060506020504060203020504030106050205030304010505030400000000

ranges3 = [[0]]
ranges3.extend([[i + (1 << j) for i in range(1 << j)] for j in range(8)])
```

```
patterns3 = [set() for _ in range(len(ranges3))]

for i in range(len(ranges3)):
    for j in ranges3[i]:
        patterns3[i].add((0x8421084210842108cc6318c6db6d54be >> j) & 31)

print(patterns3)

for i in range(8):
    for pattern in patterns3[i+1]:
        j = 0b11111 & (L3 >> ((256 - 8) - (pattern << 3)))
        if i != j:
            print(f'(i, j, pattern): {i}, {j}, {pattern:05b}')

h3 = 0
for i in range(8):
    for pattern in patterns3[i+1]:
        h3 |= i << ((256 - 8) - (pattern << 3))

for i in range(8):
    for pattern in patterns3[i+1]:
        j = 0b11111 & (h3 >> ((256 - 8) - (pattern << 3)))
        if i != j:
            print(f'(i, j, pattern): {i}, {j}, {pattern:05b}')

print("h3: " +  ' '.join(re.findall('.{8}', f'{h3:0256b}')))
print("L3: " +  ' '.join(re.findall('.{8}', f'{L3:0256b}')))
print("h3: " +  hex(h3))

print(f"h3 == L3: {h3 == L3}")

"""
suggested value
0x7060605060205000602030205040001060502050303040105050304000000000
00000111 00000110 00000110 00000101 00000110 00000010  00000101 [00000000]
00000110 00000010 00000011 00000010 00000101 00000100 [00000000] 00000001
00000110 00000101 00000010 00000101 00000011 00000011  00000100  00000001
00000101 00000101 00000011 00000100 00000000 00000000  00000000  00000000

current value
0x0706060506020504060203020504030106050205030304010505030400000000
00000111 00000110 00000110 00000101 00000110 00000010  00000101 [00000100]
00000110 00000010 00000011 00000010 00000101 00000100 [00000011] 00000001
00000110 00000101 00000010 00000101 00000011 00000011  00000100  00000001
00000101 00000101 00000011 00000100 00000000 00000000  00000000  00000000
"""
```


Recommendation: Apply the following changes:

```
diff --git a/src/libraries/BitMath.sol b/src/libraries/BitMath.sol
index 500d6f7e..6e4e8c7a 100644
--- a/src/libraries/BitMath.sol
+++ b/src/libraries/BitMath.sol
@@ -13,14 +13,14 @@ library BitMath {
         require(x > 0);

         assembly ("memory-safe") {
-            r := or(shl(8, iszero(x)), shl(7, lt(0xffffffffffffffffffffffffffffffff, x)))
+            r := shl(7, lt(0xffffffffffffffffffffffffffffffff, x))
             r := or(r, shl(6, lt(0xffffffffffffffff, shr(r, x))))
             r := or(r, shl(5, lt(0xffffffff, shr(r, x))))
             r := or(r, shl(4, lt(0xffff, shr(r, x))))
             r := or(r, shl(3, lt(0xff, shr(r, x))))
             // forgefmt: disable-next-item
             r := or(r, byte(and(0x1f, shr(shr(r, x), 0x8421084210842108cc6318c6db6d54be)),
-                0x0706060506020504060203020504030106050205030304010505030400000000))
+                0x0706060506020500060203020504000106050205030304010505030400000000))
         }
     }
```

Uniswap: Fixed in PR 822.

Spearbit: Verified.

5.4 Informational

5.4.1 Some contracts don't follow Uniswap's version convention

Severity: Informational

Context: CurrencyReserves.sol#L2, IProtocolFees.sol#L2

Description: The Solidity pragma statements in various contracts within the v4-periphery repository do not adhere to Uniswap's stated rules for version specification:

Uniswap's stated rules:

1. Contracts to be deployed should have a fixed compiler version for safety (`0.8.26`).
2. Open-source libraries without transient storage should use `^0.8.0`.
3. Open-source libraries with transient storage should use `^0.8.24`.

Current pragma statements that don't follow this:

- `^0.8.20`: `CurrencyReserves`.
- `^0.8.19`: `IProtocolFees`.

Recommendation: Standardize the version in order to align the codebase with Uniswap's stated best practices, locking the pragma version where posible or setting the correct range where needed.

Uniswap: Fixed in PR 858.

5.4.2 `computeSwapStep` can be simplified for `exactIn` swaps when `amountIn` is greater than `amountRemainingLessFee`

Severity: Informational

Context: SwapMath.sol#L74-L81

Description: In the above context we are in the case of `exactIn` swaps when `amountIn` is greater than `amountRemainingLessFee`:

```
sqrtPriceNextX96 = SqrtPriceMath.getNextSqrtPriceFromInput(
    sqrtPriceCurrentX96, liquidity, amountRemainingLessFee, zeroForOne
);
amountIn = zeroForOne
    ? SqrtPriceMath.getAmount0Delta(sqrtPriceNextX96, sqrtPriceCurrentX96, liquidity, true)
    : SqrtPriceMath.getAmount1Delta(sqrtPriceCurrentX96, sqrtPriceNextX96, liquidity, true);
// we didn't reach the target, so take the remainder of the maximum input as fee
feeAmount = uint256(-amountRemaining) - amountIn;
```

Notations:

parameter	description
$\sqrt{p_c}$	`sqrtPriceCurrentX96`
$\sqrt{p_t}$	`sqrtPriceTargetX96`
$\sqrt{p_n}$	`sqrtPriceNextX96`
a_i	`amountIn`
a_o	`amountOut`
a_w	`amountRemainingLessFee`
a_r	`amountRemaining`
a_f	`feeAmount`
L	`liquidity`
f	`feePips`

1. Case $0 \to 1$ swaps

$$\sqrt{p_n} = \frac{L}{\frac{L}{\sqrt{p_c}} + \frac{a_w}{2^{96}}}$$

$$a_i = 2^{96} \frac{L}{\sqrt{p_n}} - \frac{L}{\sqrt{p_c}}$$

$= a_w$

2. Case $1 \to 0$ swaps

$$\sqrt{p_n} = \sqrt{p_c} + \frac{2^{96} a_w}{L}$$

$$a_i = \frac{\left(\sqrt{p_n} - \sqrt{p_c}\right) L}{2^{96}} = a_w$$

so in both directions in this inner `else` block one could have just set `amountIn` as `amountRemainingLessFee` and the `feeAmount` ends up being:

$$a_f = \frac{f \cdot a_r}{10^6}$$

or in other words `amountIn` gets capped by `amountRemainingLessFee`. Doing so, make the code more unified when compared to the implementation in the outer `else` block below where `exactIn == false`.

Recommendation: The following modification can be applied:

```
diff --git a/src/libraries/SwapMath.sol b/src/libraries/SwapMath.sol
index e0f4b264..59232535 100644
--- a/src/libraries/SwapMath.sol
+++ b/src/libraries/SwapMath.sol
@@ -71,12 +71,10 @@ library SwapMath {
                         ? amountIn
                         : FullMath.mulDivRoundingUp(amountIn, _feePips, MAX_FEE_PIPS - _feePips);
                 } else {
+                    amountIn = amountRemainingLessFee;
                     sqrtPriceNextX96 = SqrtPriceMath.getNextSqrtPriceFromInput(
                         sqrtPriceCurrentX96, liquidity, amountRemainingLessFee, zeroForOne
                     );
-                    amountIn = zeroForOne
-                        ? SqrtPriceMath.getAmount0Delta(sqrtPriceNextX96, sqrtPriceCurrentX96,
↪  liquidity, true)
-                        : SqrtPriceMath.getAmount1Delta(sqrtPriceCurrentX96, sqrtPriceNextX96,
↪  liquidity, true);
                     // we didn't reach the target, so take the remainder of the maximum input as fee
                     feeAmount = uint256(-amountRemaining) - amountIn;
                 }
```

Uniswap: Fixed in PR 718.

Spearbit: Verified.

5.4.3 Add comments regarding the derivation of `SQRT_PRICE_A_B` constant

Severity: Informational

Context: Constants.sol#L5-L10

Description: The constants `SQRT_PRICE_A_B` in this context are calculated as:

$$\left\lfloor \sqrt{\frac{A}{B}} \cdot 2^{96} \right\rfloor$$

Where A and B are reserve amounts in the pair of currencies involved in the pool.

Recommendation: Add comments regarding the derivation of `SQRT_PRICE_A_B` constant. And make sure the named constant are imported from this utility/library instead of declaring them within each test file such as `TickMathTestTest`.

Uniswap: Fixed in PR 859.

Spearbit: Verified.

5.4.4 `amountIn` is always `0` in an inner branch of `computeSwapStep`

Severity: Informational

Context: SwapMath.sol#L71

Description: In the above context we have:

```
if (exactIn) {
    uint256 amountRemainingLessFee =
        FullMath.mulDiv(uint256(-amountRemaining), MAX_FEE_PIPS - _feePips, MAX_FEE_PIPS);
    amountIn = zeroForOne
        ? SqrtPriceMath.getAmount0Delta(sqrtPriceTargetX96, sqrtPriceCurrentX96, liquidity, true)
        : SqrtPriceMath.getAmount1Delta(sqrtPriceCurrentX96, sqrtPriceTargetX96, liquidity, true);
    if (amountRemainingLessFee >= amountIn) {
        // ...
        feeAmount = _feePips == MAX_FEE_PIPS
            ? amountIn // <<<
            // ...
```

If `_feePips == MAX_FEE_PIPS` then `amountRemainingLessFee == 0` which in the above second `if` branch forces `amountIn` to be `0`:

```
0 = amountRemainingLessFee >= amountIn
```

Recommendation: If we rewrite this as:

```
feeAmount = _feePips == MAX_FEE_PIPS
     ? 0
     : FullMath.mulDivRoundingUp(amountIn, _feePips, MAX_FEE_PIPS - _feePips);
```

according to the forge test case it would cost more gas:

```
forge snapshot --diff
```

```
test_shouldSwapEqual(uint24,int24,int24,int24,int256,int256,int128,bool) (gas: -20 (-0.000%))
test_swap_100PercentFee_AmountIn_WithProtocol() (gas: -1 (-0.001%))
test_swap_100PercentLPFee_AmountIn_NoProtocol() (gas: -1 (-0.001%))
test_ffi_fuzz_addLiquidity_defaultPool_ReturnsCorrectLiquidityDelta((int24,int24,int256,bytes32)) (gas:
↪  -2 (-0.001%))
test_fuzz_getTickLiquidity((int24,int24,int256,bytes32)) (gas: -2 (-0.001%))
test_fuzz_getTickBitmap((int24,int24,int256,bytes32)) (gas: -2 (-0.001%))
test_fuzz_getPositionLiquidity((int24,int24,int256,bytes32),(int24,int24,int256,bytes32)) (gas: -16
↪  (-0.004%))
test_fuzz_getPositionInfo((int24,int24,int256,bytes32),uint256,bool) (gas: 27 (0.005%))
test_fuzz_getFeeGrowthInside((int24,int24,int256,bytes32),bool) (gas: 40 (0.007%))
test_fuzz_getTickLiquidity_two_positions((int24,int24,int256,bytes32),(int24,int24,int256,bytes32))
↪  (gas: -29 (-0.007%))
test_fuzz_nextInitializedTickWithinOneWord(int24,bool) (gas: -75 (-0.108%))
test_fuzz_swap(uint160,uint24,uint16,uint16,(int24,bool,int256,uint160,uint24)) (gas: 26 (0.159%))
test_fuzz_extsload(uint256,uint256,bytes) (gas: 7173 (0.563%))
Overall gas change: 7118 (0.002%)
```

It might still be useful to leave a comment that at this specific edge case `amountIn` and thus `feeAmount` would be `0`. And if there are no test cases present for this edge case to also add some tests for it.

Uniswap: Comments added in PR 857.

Spearbit: Verified.

5.4.5 Unused code should be removed

Severity: Informational

Context: StateLibrary.sol#L15-L16

Description: Unused code should be removed, this would help decreasing cognitive load and make easier the read, additionally reducing a little the contract codesize. Some instances:

- `FEE_GROWTH_GLOBAL1_OFFSET` is not used neither at v4-core, v4-periphery, or universal router codebases. However, it provides useful information about the storage layout.

Recommendation: Consider commenting the storage layout and removing unused variables

Uniswap: The line corresponding to the above constant has been commented out in the library in PR 857.

Spearbit: Verified.

5.4.6 Unnecessary `unchecked` blocks

Severity: Informational

Context: UnsafeMath.sol#L13-L19

Description: `divRoundingUp` in UnsafeMath is wrapped in an `unchecked` block. However, this block is unnecessary because the function uses inline assembly for its calculations. The `unchecked` keyword in Solidity is used to disable overflow and underflow checks for arithmetic operations, but it has no effect on assembly code, which is inherently unchecked.

Recommendation: Remove the `unchecked` block as it serves no purpose in this context. The function can be simplified to:

```
function divRoundingUp(uint256 x, uint256 y) internal pure returns (uint256 z) {
    assembly ("memory-safe") {
        z := add(div(x, y), gt(mod(x, y), 0))
    }
}
```

Uniswap: Fixed in PR 857.

Spearbit: Verified.

5.4.7 Confusing error message in `ERC6909.transferFrom()`

Severity: Informational

Context: ERC6909.sol#L38

Description: When the allowance is lower than the transferred amount, ERC6909.transferFrom() returns a low level "arithmetic underflow or overflow" error:

```
uint256 allowed = allowance[sender][msg.sender][id];
if (allowed != type(uint256).max) allowance[sender][msg.sender][id] = allowed - amount;
```

The error can be confusing for users because it doesn't explicitly says that the allowance is too low.

Recommendation: Consider returning a meaningful error. For example, see this ERC6909 implementation or the OpenZeppelin's ERC20 implementation.

Uniswap: Added a custom revert for `InsufficientAllowance` and `InsufficientBalance` in PR 833. Currently, it's causing us to exceed contract bytecode size limits. We may elect to not do custom reverts.

5.4.8 `getSqrtPriceAtTick` assumes that the allowed tick range is centered at `0`

Severity: Informational

Context: TickMath.sol#L67

Description: `getSqrtPriceAtTick` assumes that the allowed tick range is centered at `0`, ie `MAX_TICK == -MIN_TICK` due to the following bound check:

```
if (absTick > uint256(int256(MAX_TICK))) InvalidTick.selector.revertWith(tick);
```

Recommendation: Perhaps this needs to be documented/highlighted in case the codebase is changed in the future where the invariant `MAX_TICK == -MIN_TICK` is not satisfied anymore.

Uniswap: Comments have been added in PR 851.

Spearbit: Verified.

5.4.9 The current or next tick is not always on the tick spacing grid or within the allowed range

Severity: Informational

Context: Pool.sol#L343-L349, Pool.sol#L421, Pool.sol#L425

Description/Recommendation:

- ☐ Pool.sol#L343-L349: clipping `step.tickNext` to the `TickMath.MIN_TICK` and `TickMath.MAX_TICK` range breaks the assumptions that `step.tickNext` is always on the `tickSpacing` grid.

  The following is not always true when clipped:

  $$\Delta i \mid i_{next}$$

  For these out of bound `step.tickNext`, `step.initialized` should be (is) `false`.

- ☐ Pool.sol#L421: Doing the following can push the `state.tick` out of the minimum bound `TickMath.MIN_TICK` when `_zeroForOne` is `1`:

  ```
  state.tick = step.tickNext - _zeroForOne
  ```

- ☐ Pool.sol#L421, Pool.sol#L425: in this context when the tick is decremented or recalculated from the price:

  ```
  state.tick = step.tickNext - _zeroForOne;
  ```

  ```
  state.tick = TickMath.getTickAtSqrtPrice(state.sqrtPriceX96);
  ```

  and later when one updates the storage the `self.slot0.tick()` will not necessarily be on the `tickSpacing` grid or just off by `1` from it.

Uniswap: Addressed in PR 852.

5.4.10 `unchecked` blocks

Severity: Informational

Context: Pool.sol#L369-L372, Pool.sol#L381-L382, Pool.sol#L384, Pool.sol#L408

Description/Recommendation:

☐ Pool.sol#L369-L372: It is true that it is safe. But had to double check for this branch in `SwapMath.computeSwapStep` due to different rounding direction for the inequalities:

```
if (exactIn) {
    uint256 amountRemainingLessFee =
        FullMath.mulDiv(uint256(-amountRemaining), MAX_FEE_PIPS - _feePips, MAX_FEE_PIPS);
    amountIn = zeroForOne
        ? SqrtPriceMath.getAmount0Delta(sqrtPriceTargetX96, sqrtPriceCurrentX96, liquidity, true)
        : SqrtPriceMath.getAmount1Delta(sqrtPriceCurrentX96, sqrtPriceTargetX96, liquidity,
↪  true);
    if (amountRemainingLessFee >= amountIn) {
        // `amountIn` is capped by the target price
        sqrtPriceNextX96 = sqrtPriceTargetX96;
        feeAmount = _feePips == MAX_FEE_PIPS
            ? amountIn
            : FullMath.mulDivRoundingUp(amountIn, _feePips, MAX_FEE_PIPS - _feePips);
```

for the notations see this discussion and $f_{swap} = f(\texttt{feePips})$:

in the second `if` branch we know `amountRemainingLessFee >= amountIn`:

$$a_w = \left\lfloor \frac{(-a_r)\left(10^6 - f_{swap}\right)}{10^6} \right\rfloor \geq a_i$$

where a_f is:

$$a_f = \left\lceil \frac{a_i \cdot f_{swap}}{10^6 - f_{swap}} \right\rceil$$

and so we need to make sure the following inequality is guaranteed:

$$-a_r \geq a_i + a_f = \left\lceil \frac{a_i \cdot 10^6}{10^6 - f_{swap}} \right\rceil$$

But in general we have for $a, b \in \mathbb{Z}$ and $k \in \mathbb{R}^+$:

$$\lfloor k \cdot a \rfloor \geq b \Rightarrow a \geq \left\lceil \frac{b}{k} \right\rceil$$

The comment can be more accurate though since the `state.amountSpecifiedRemaining` is negated in the inequality.

☐ Pool.sol#L381-L382:

- **Case.** `exactInput == true`

  We have in Pool.sol#L370-L372:

```
unchecked {
    state.amountSpecifiedRemaining += (step.amountIn + step.feeAmount).toInt256();
}
```

and based on this discussion, we know that `state.amountSpecifiedRemaining` always stays non-positive. So at the very end of this function where we have

```
(params.amountSpecified - state.amountSpecifiedRemaining).toInt128()
```

for the first or second component of `result`. Thus, we can deduce that:

$$a_{\text{spec}} - a_{\text{remain}} = -\sum_j (a_{\text{in}}^j + a_f^j) \geq -2^{127}$$

and thus for each iteration of the loop we would have:

$$a_{\text{in}}^j + a_f^j \leq 2^{127}$$

so even if multiplied by 10^3 it would still not overflow in the `uint256` range.

- **Case.** `exactInput == false`

  We have in Pool.sol#L367:

  ```
  state.amountCalculated -= (step.amountIn + step.feeAmount).toInt256();
  ```

  Note that this is a checked block and the type of `state.amountCalculated` is `int256`. So the negative summation of these value for all the iterations cannot underflow. We also have at the very end:

  ```
  state.amountCalculated.toInt128()
  ```

  for either the first or second component of `result`. And thus like the previous case we would have:

$$-\sum_j (a_{\text{in}}^j + a_f^j) \geq -2^{127}$$

☐ Pool.sol#L384: To prove that this context doesn't underflow, we need to show:

$$a_f \geq \left\lfloor \frac{(a_i + a_f) \cdot f_{\text{proto}}}{10^6} \right\rfloor$$

Let $f_s = f_{\text{swap}}$, $f_p = f_{\text{proto}} \in [0, 10^3]$ and $f_L = f_{\text{LP}} \in [0, 10^6]$. Then

$$f_s = f_p + f_L - \left\lceil \frac{f_p \cdot f_L}{10^6} \right\rceil \geq f_p$$

The above is true since for all $x \in \mathbb{N} \cup \{0\}$ and $k \in [0, 1]$ we have:

$$\lceil kx \rceil \leq x$$

- **Case 1.** $f_s \neq 10^6$

  To show the original inequality in the first comment we need to prove:

$$\left\lfloor \frac{\left\lceil \frac{a_i}{10^6 - f_s} \cdot 10^6 \right\rceil f_p}{10^6} \right\rfloor \leq \left\lceil \frac{a_i}{10^6 - f_s} \cdot f_s \right\rceil$$

or even a stronger inequality since we know $f_p \leq f_s$:

$$\left\lfloor \frac{\left\lceil \frac{a_i}{10^6 - f_s} \cdot 10^6 \right\rceil f_p}{10^6} \right\rfloor \leq \left\lceil \frac{a_i}{10^6 - f_s} \cdot f_p \right\rceil$$

Let $x = \frac{a_i}{10^6 - f_s} \in \mathbb{Q}^{\geq 0}$ and $y = f_p$, then we need to show:

$$\left\lfloor \frac{\lceil x \cdot 10^6 \rceil \, y}{10^6} \right\rfloor \leq \lceil x \cdot y \rceil$$

Let $x = a + \frac{b+\epsilon}{10^6}$ where $a \in \{0, 1, 2, \cdots\}$, $b \in \{0, 1, \cdots, 10^6 - 1\}$ and $\epsilon \in [0, 1)$. Then we need to show that:

$$\left\lfloor \left\lceil \left(10^6 a + b + \epsilon\right) \right\rceil \cdot \frac{y}{10^6} \right\rfloor \leq \lceil x \cdot y \rceil$$

or

$$\left\lfloor \left\lceil \left(10^6 a + b + \epsilon\right) \right\rceil \cdot \frac{y}{10^6} \right\rfloor \leq \left\lceil (10^6 a + b + \epsilon) \cdot \frac{y}{10^6} \right\rceil$$

Note that we can subtract ay from both sides to get:

$$\left\lfloor \lceil (b+\epsilon) \rceil \cdot \frac{y}{10^6} \right\rfloor \leq \left\lceil (b+\epsilon) \cdot \frac{y}{10^6} \right\rceil$$

and we can even try to prove stronger inequality:

$$\left\lfloor (b+1) \cdot \frac{y}{10^6} \right\rfloor \leq \left\lceil b \cdot \frac{y}{10^6} \right\rceil$$

let $k = \frac{y}{10^6} \in [0, 10^{-3}]$, then we need to show:

$$\lfloor (b+1)k \rfloor \leq \lceil bk \rceil$$

or

$$\lfloor bk \rfloor + \lfloor \{bk\} + k \rfloor \leq \lfloor bk \rfloor + \lceil \{bk\} \rceil$$

or

$$\lfloor \{bk\} + k \rfloor \leq \lceil \{bk\} \rceil$$

But from the range of k we know that $\{bk\} + k \in \left[0, 1 + 10^{-3}\right)$ and so both sides of the inequality above can either be 0 or 1 and the right hand side can only be 1 if and only if $\{bk\} + k \geq 1$ which implies that whenever it is 1 then $\{bk\}$ needs to be non-zero and thus $\lceil \{bk\} \rceil = 1$ which proves the inequality.

– **Case 2** $f_s = 10^6$

Then we know that we should have $a_{spec} \leq 0$ or only the exact input branches are reached. Also we know in this case $f_L = 10^6$. Then there are 2 cases.

* **Case 2.1** (see SwapMath.sol#L70-L71)

  In this case both $a_f = a_i = 0$ which then the inequality is obvious.

* **Case 2.2** (see SwapMath.sol#L77-L81)

```
    sqrtPriceNextX96 = SqrtPriceMath.getNextSqrtPriceFromInput(
        sqrtPriceCurrentX96, liquidity, amountRemainingLessFee, zeroForOne
    );
    amountIn = zeroForOne
        ? SqrtPriceMath.getAmount0Delta(sqrtPriceNextX96, sqrtPriceCurrentX96,
↪  liquidity, true)
        : SqrtPriceMath.getAmount1Delta(sqrtPriceCurrentX96, sqrtPriceNextX96,
↪  liquidity, true);
    // we didn't reach the target, so take the remainder of the maximum input as fee
    feeAmount = uint256(-amountRemaining) - amountIn;
```

We know that `amountRemainingLessFee == 0` and thus `sqrtPriceNextX96 == sqrtPriceCurrentX96` which implies that `amountIn == 0` and `feeAmount == uint256(-amountRemaining)`. So in this case we have:

$$a_i = 0, a_f = -a_r$$

and the inequality becomes:

$$\left\lfloor \frac{f_p}{10^6} \cdot a_f \right\rfloor \leq a_f$$

☐ Pool.sol#L408: We have that:

$$L_{i,g} = L_{i,l} + L_{i,u}$$

$$L_{i,n} = L_{i,l} - L_{i,u}$$

and we know that max gross liquidity of a tick cannot be greater than the `tickSpacingToMaxLiquidityPerTick(tickSpacing)`:

$$L_{i,g} \leq \frac{2^{128} - 1}{\left\lfloor \frac{i_{max}}{\Delta i} \right\rfloor + \left\lfloor \frac{|\, i_{min} \,|}{\Delta i} \right\rfloor + 1} < \frac{2^{128}}{3} < 2^{127}$$

and since $L_{i,l}, L_{i,u}$ are non-negative values, one can deduce that:

$$-2^{127} < L_{i,n}$$

parameter	description
$L_{i,g}$	`liquidityGross` at the tick i
$L_{i,n}$	`liquidityNet` at the tick i

parameter	description
$L_{i,l}$	Sum of all the liquidity of all positions with their lower tick equal to i
$L_{i,u}$	Sum of all the liquidity of all positions with their upper tick equal to i
Δi	`tickSpacing`

5.4.11 Dirty bit cleaning

Severity: Informational

Context: SwapMath.sol#L31, Pool.sol#L419

Description:

SwapMath.sol#L31: In the context of the codebase this and some other upper bit cleanings are not necessary `and(zeroForOne, 0xff)` since on external calls the `solc` compiler performs cleaning. But in the context of a `library` and `internal` functions why the `zeroForOne` value is not completely cleaned by doing `and(zeroForOne, 1)` ?

☐ Pool.sol#L419: Why not just `and` with `1`?

Recommendation: Apply the following bit cleaning instead:

```
and(zeroForOne, 1)
```

Uniswap:

- Fixed in PR 838.
- Pool.sol#L419: is transformed into and thus avoiding the bit cleanup necessary. See PR 827.

```
unchecked {
    result.tick = zeroForOne ? step.tickNext - 1 : step.tickNext;
}
```

Spearbit: Verified.

5.4.12 Named return are unused in `settle()` and `settleFor()`

Severity: Informational

Context: PoolManager.sol#L288-L291

Description: `settle()` and `settleFor()` functions declare a named return variable `paid`, but do not explicitly use it in the function body. Instead, they directly return the result of the `_settle()` function call.

Recommendation: Either use the named return variable explicitly or remove it.

```
  function settle() external payable onlyWhenUnlocked returns (uint256 paid) {
- return _settle(msg.sender);
+ paid = _settle(msg.sender);
```

Uniswap: Fixed in PR 829.

Spearbit: Fixed. The `paid` parameter was removed.

5.4.13 `collectProtocolFees` lacks an own event to track fee collections

Severity: Informational

Context: ProtocolFees.sol#L57

Description: `collectProtocolFees` transfers collected protocol fees to a recipient but only emits a generic currency Transfer event. This lacks specificity and makes it difficult to track protocol fee collection activities separately from other transfers. A dedicated event for protocol fee collection would improve transparency and make it easier to monitor and analyze these specific transactions.

Recommendation: Implement a specific event for protocol fee collection. For example:

```
event ProtocolFeeCollected(address indexed recipient, Currency indexed currency, uint256 amount,
↪  address caller);
```

Uniswap: Acknowledged. We are not going to be adding an event to `collectProtocolFees`.

Spearbit: Acknowledged.

5.4.14 Best practices for handling action flows

Severity: Informational

Context: PoolManager.sol#L271-L285

Description: Developers have to be aware of potential issues that may cause swaps or flash loans to revert. First, `sync()` can be called outside of unlocks, and at most 1 currency can be synced each time before settlement. In other words, `sync()` cannot be called in succession, which enables a Denial of Service (DoS) attack vector.

Second, native token transfers via the `take()` action which executes `Currency(native).transfer()` would hand over the control flow to the recipient, allowing it to revert the entire transaction.

Recommendation: Recommend best practices for integrators and developers, and highlight present limitations that they should be aware of. Specifically:

- Consider checking for an existing sync and calling `settle()` before invoking `sync()`.
- Be cautious with native token transfers to untrusted recipients.

Uniswap: 2 PRs that change the current behaviour:

1. Lock added to `sync` in PR 856.
2. `sync` no longer reverts (just overrides), and allows native to be synced to remove DoS attack vectors in PR 866.

Spearbit: Acknowledged on the new behaviour. There is a footgun introduced that developers should be aware of: if one syncs one currency → transfers tokens → syncs another without settlement, the token transfer will not be accounted for.

5.4.15 Pools with maximum `lpFee` do not support exact output swaps

Severity: Informational

Context: Pool.sol#L312-L314

Description: While it is possible to set `lpFee` to 100%, it will cause exact output swaps to revert. In other words, such pools will only work with `exactIn` swaps.

Recommendation: Developers and pool creators should be aware of this side-effect should they choose to set maximum `lpFee`.

Uniswap: Fixed in PR 842.

Spearbit: Fixed.

5.4.16 `Currency.isZero()` is equivalent to `Currency.isNative()`

Severity: Informational

Context: Currency.sol#L104-L110

Description: `isZero()` is equivalent to `isNative()`. Either function could be removed for simplicity and its instances replaced with the other.

Recommendation: Consider removing either `isZero()` or `isNative()` and replace all its instances with the other function.

Uniswap: Fixed in PR 834.

Spearbit: Fixed. `isZero()` is renamed to `isAddressZero()` and `isNative()` has been removed.

5.4.17 Comment Improvements

Severity: Informational

Context: IPoolManager.sol#L104, IPoolManager.sol#L190, IHooks.sol#L9-L11, IHooks.sol#L72, IHooks.sol#86, ProtocolFees.sol#L61, LPFeeLibrary.sol#L21, LPFeeLibrary.sol#L34, TransientStateLibrary.sol#L27, Pool.sol#L559, ProtocolFees.sol#L67, IPoolManager.sol#L140-L142, IPoolManager.sol#L140-L142

Description: The following are comment clarifications for correctness and clarity, and typos.

Recommendation:

```
- /// @dev The only functions callable without an unlocking are `initialize` and `updateDynamicLPFee`
+ /// @dev The only functions callable without an unlocking are `initialize`, `sync` and
↪  `updateDynamicLPFee`

- retreivable
+ retrievable

- /// @notice The PoolManager contract decides whether to invoke specific hooks by inspecting the
↪  leading bits
- /// of the hooks contract address. For example, a 1 bit in the first bit of the address will
- /// cause the 'before swap' hook to be invoked. See the Hooks library for the full spec.
+ /// @notice V4 decides whether to invoke specific hooks by inspecting the lowest significant bits of
↪  the address that
+ /// the hooks contract is deployed to.
+ /// For example, a hooks contract deployed to address: 0x0000000000000000000000000000000000002400
+ /// has the lowest bits '10 0100 0000 0000' which would cause the 'before initialize' and 'after add
↪  liquidity' hooks to be used.
+ /// See the Hooks library for the full spec.

- liquidty
+ liquidity

- overriden
+ overridden

- beforeSwaphook
+ beforeSwap hook

- maxmimum
+ maximum

- zerod
+ zeroed

- /// @dev Executed within the pool constructor
+ /// @dev Executed when adding liquidity
```

```
- /// @dev the success of this function must be checked when called in setProtocolFee
  (the function under the comment above is not called in setProtocolFee)

+ /// Whether to swap token zero for token one or vice versa
  bool zeroForOne;
+ /// The desired input amount if negative ("exact in"), or the desired output amount if positive
↪  ("exact out")
  int256 amountSpecified;
+ /// The most extreme square-root-price the pool may reach by the end of the swap
  uint160 sqrtPriceLimitX96;
```

The `IPoolManager` have stale comments vs. `PoolManager`:

```
- /// @return feeDelta The balance delta of the fees generated in the liquidity range. Returned for
↪  informational purposes.
+ /// @return feesAccrued The balance delta of the fees generated in the liquidity range. Returned for
↪  informational purposes.
    function modifyLiquidity(PoolKey memory key, ModifyLiquidityParams memory params, bytes calldata
↪  hookData)
        external
-        returns (BalanceDelta callerDelta, BalanceDelta feeDelta);
+        returns (BalanceDelta callerDelta, BalanceDelta feesAccrued);
```

Uniswap: Fixed in PR 846.

Spearbit: Fixed.

5.4.18 `memory-safe` annotation

Severity: Informational

Context: CurrencyDelta.sol#L20-L22, CurrencyReserves.sol#L25, CurrencyReserves.sol#L31, CurrencyReserves.sol#L37, CurrencyReserves.sol#L44, CustomRevert.sol#L69-L74

Description/Recommendation:

- CurrencyDelta.sol#L20-L22, CurrencyReserves.sol#L25, CurrencyReserves.sol#L31, CurrencyReserves.sol#L37, CurrencyReserves.sol#L44: missing `memory-safe` annotation.
- CustomRevert.sol#L69-L74: this `assembly` block does not follow the `memory-safe` annotation requirement since it writes to memory space right passed the scratch memory slots. To be safe one should use the free memory pointer and write to memory right at and after that location.

Uniswap: Fixed in PR 830.

Spearbit: Verified.