Zellic

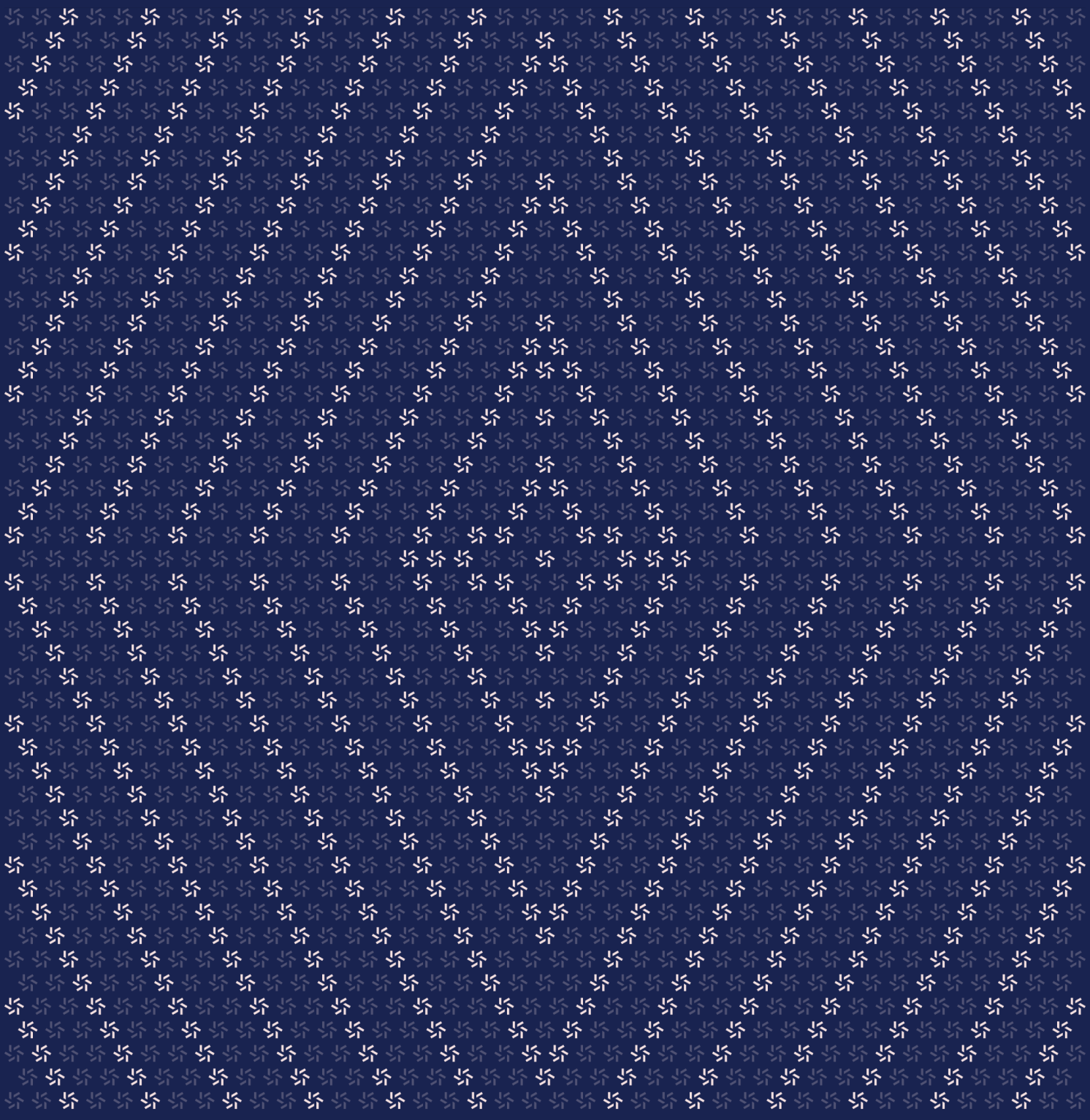
Prepared for
Chef Momo
PancakeSwap

Prepared by
Katerina Belotskaia
Daniel Lu
Zellic

October 1, 2024

PancakeSwap Infinity Core

Smart Contract Security Assessment



Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for PancakeSwap from July 9th to August 21st, 2024. During this engagement, Zellic reviewed PancakeSwap Infinity Core's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Is the vault accounting model implemented correctly?
- Could any malicious ERC20 tokens break the contract?
- Are the contracts resilient to common attack vectors, such as reentrancy, overflow/underflow, and front-running that could lead to a loss of funds?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.4. Results

During our assessment on the scoped PancakeSwap Infinity Core contracts, we discovered four findings. No critical issues were found. One finding was of high impact, one was of medium impact, and two were of low impact.

Additionally, Zellic recorded its notes and observations from the assessment for PancakeSwap's benefit in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	1
Medium	1
Low	2
Informational	0

2. Introduction

2.1. About PancakeSwap Infinity Core

PancakeSwap contributed the following description of PancakeSwap Infinity Core:

> PancakeSwap v4 is the latest iteration of the DEX, introducing several key features and improvements like Singleton contract, Flash Accounting, Hooks, and support for multiple AMM curves to enhance user experience and efficiency.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect

its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

PancakeSwap Infinity Core Contracts

Type	Solidity
Platform	EVM-compatible
Target	infinity-core
Repository	https://github.com/pancakeswap/infinity-core ↗
Version	`df64445b02596747090e9ba43972bbce2ae5cbde`
Programs	`src/*`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 7.05 person-weeks. The assessment was conducted by two consultants over the course of six calendar weeks.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald
Engagement Manager
chad@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Katerina Belotskaia
Engineer
kate@zellic.io ↗

Daniel Lu
Engineer
daniel@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

July 9, 2024	Start of primary review period
July 10, 2024	Kick-off call
August 21, 2024	End of primary review period

3. Detailed Findings

3.1. The mint/burn actions are more profitable than swaps

Target	BinPool		
Category	Business Logic	**Severity**	High
Likelihood	Medium	**Impact**	High

> **Note:** PancakeSwap independently discovered and promptly addressed this issue. Because the vulnerability exists in the audit version, we document it in this finding for completeness.

Description

The BinPoolManager contract provides users with several ways to interact with liquidity pools, including the `mint` and `burn` liquidity tokens' actions as well as performing token `swaps`. Specifically, for this type of pool, liquidity is allocated across bins, where each bin represents a single price point, and the gap between two consecutive bins is defined by the bin step. When a user performs the `mint` action, they add the new liquidity to the specified bin, receiving liquidity shares in return. If users add liquidity to the currently active bin, they also pay protocol commission and a liquidity-pool fee specified by the pool creator. The liquidity-pool fee is distributed among the current liquidity providers.

```
function _updateBin(State storage self, MintParams memory params, uint24 id,
    bytes32 maxAmountsInToBin)
    internal
    returns (
        [...]
    )
{
        [...]

    if (id == activeId) {
        bytes32 fees;
        (fees, feeForProtocol) =
            binReserves.getCompositionFees(slot0Cache.protocolFee,
    slot0Cache.lpFee, amountsIn, supply, shares);
        compositionFee = fees;
        if (fees != 0) {
            {
                uint256 userLiquidity
    = amountsIn.sub(fees).getLiquidity(price);
                uint256 binLiquidity = binReserves.getLiquidity(price);
                shares = userLiquidity.mulDivRoundDown(supply, binLiquidity);
```

```
            }

            if (feeForProtocol != 0) {
                amountsInToBin = amountsInToBin.sub(feeForProtocol);
            }
        }
    }
}
```

The `burn` function allows users to burn part or all of their liquidity shares and withdraw the provided liquidity, including a part of liquidity-pool fee. However, there is a problem here; the `mint` and subsequent `burn` actions, which can be considered equivalent to a swap, result in a slightly more profitable outcome compared to executing a swap action. This happens because the liquidity-pool fee provided during the `mint` action is not taken into account during the share amount calculation. As a result, when users execute the `burn` function, they also receive a part of the paid liquidity-pool fee back, which does not happen when performing an exchange.

Impact

This issue can create an imbalance in the protocol if users start exploiting `mint`/`burn` actions for marginal gains.

Recommendations

We recommend implementing the following change.

```
function _updateBin(State storage self, MintParams memory params, uint24 id,
    bytes32 maxAmountsInToBin)
    internal
    returns (
    [...]
    )
{
    [...]
    if (id == activeId) {
       [...]
        if (fees != 0) {
            {
                uint256 userLiquidity
    = amountsIn.sub(fees).getLiquidity(price);

    uint256 binLiquidity = binReserves.add(fees.sub(feeForProtocol)
                ).getLiquidity(price);
            uint256 binLiquidity = binReserves.getLiquidity(price);
```

```
                shares = userLiquidity.mulDivRoundDown(supply, binLiquidity);
            }
        }
    }
    [...]
}
```

Remediation

PancakeSwap independently discovered this bug during the assessment period and fixed it in commit e87395bd ↗.

3.2. Double spending

Target	Vault		
Category	Business Logic	**Severity**	High
Likelihood	Low	**Impact**	Medium

> **Note:** PancakeSwap independently discovered and promptly addressed this issue. Because the vulnerability exists in the audit version, we document it in this finding for completeness.

Description

The `settle` function is intended to be called at the end of a transaction after all necessary actions by a user to pay off all their debts. This function handles native and ERC20 tokens separately. If the user wants to settle with native tokens, the `msg.value` will be used as the payment amount. Otherwise, the difference between the balance since the last `sync` function call and the current balance will be used as the payment amount.

```
function settle(Currency currency)
    external payable override isLocked returns (uint256 paid) {
    if (!currency.isNative()) {
        if (msg.value > 0) revert SettleNonNativeCurrencyWithValue();
        uint256 reservesBefore = currency.getVaultReserves();
        uint256 reservesNow = sync(currency);
        paid = reservesNow - reservesBefore;
    } else {
        paid = msg.value;
    }
    SettlementGuard.accountDelta(msg.sender, currency, paid.toInt128());
}
```

The problem occurs on some networks that support the ERC-20 interface for native tokens, such as the Celo network. In this case, the same native token can be processed by the `settle` function both as a native and as an ERC20 token.

Impact

The user can execute the `settle` function twice — the first time, for the `currency` as a native token, providing the `msg.value` as payment. The second time, the user can provide the `currency` as an ERC20 token, using the difference between the balance since the last sync and the new vault

balance, increased after the first `settle` call. This allows the user to pay the debt twice without providing additional funds.

Recommendations

We recommend enforcing a stricter flow for the settlement process.

Remediation

This issue has been acknowledged by PancakeSwap, and a fix was implemented in commit `37d3cd35` ↗.

3.3. Missing slippage check for donations

Target	Vault		
Category	Business Logic	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

Both application styles that are currently implemented grant users the ability to donate tokens to pools. These donations are given to active liquidity providers as fees. Specifically, the bin pools apply donations to the currently active bin, while the concentrated liquidity pools apply donations to the currently active tick.

Impact

An attacker could front-run a donation by adding liquidity, stealing a portion of it for themselves. Further, in lower-liquidity pools, they could induce a price movement that causes the donation to enter an unexpected bin or tick. This may let them keep a larger portion of the donation for themselves, if there is less liquidity in the new bin or tick.

This style of transaction is discussed further in Discussion point 4.3. ↗.

Recommendations

We recommend allowing the user to specify a price range for their donation and enforcing that the funds are given to the intended liquidity providers.

Remediation

This finding has been acknowledged by PancakeSwap. Their official response is paraphrased below:

> Slippage checks are handled by the periphery contracts. The same principle applies to donations, where the responsibility for slippage checks remains outside the core contract.

3.4. Shares have no minimum precision

Target	BinPool		
Category	Business Logic	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The bin-style pools track liquidity provider positions with shares. Initially, each share corresponds exactly a unit of liquidity, which is computed as the sum of the two token *values* (taking the price of the first token). But as the pool receives fees and donations, the value of a share will increase with respect to that sum.

Because the pool allows donations, it's possible for the value of a share to become excessively inflated.

Impact

This may increase the impact of precision loss due to rounding. It is worth noting that the issue is quite unlikely, because the donations would cause an attacker to lose funds unless they are the only liquidity provider.

Recommendations

We recommend increasing the initial precision of the shares. For example, the application can start the share value at the liquidity multiplied by a scale factor.

Remediation

This issue has been acknowledged by PancakeSwap, and a fix was implemented in commit `194f7b4e` ↗.

The team has also commented on the issue:

> To mitigate the issue of share inflation, we have introduced a requirement for a minimum number of shares in the pool before donations are allowed for the BinPool.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Important execution flow for Vault contract

This point highlights important contract behavior for users to consider.

The Vault contract requires executing functions in a specific order. The first function to be called should be the `lock` function. Only after this, other functions with the `isLocked` modifier can be triggered, such as `take`, `burn`, or `mint` functions. These functions modify the account delta, which should be reset to zero at the end of the transaction for all modified account deltas to successfully complete the transaction. Users should call the `settle` function to adjust the delta if it becomes negative after taking funds from the vault. However, before doing so, they should transfer the required funds to the contract (in the case of an ERC20 token), and only the difference between the previous balance state and the current state will be taken into account as payment.

To set up this previous balance state, the `sync` function should be triggered. Otherwise, two behaviors are possible: the transaction may be reverted if `sync` was not triggered at all, or the contract balance may be initialized at the wrong time. It is important to note that if the `sync` function is triggered after providing funds to the vault without calling the `settle` function, the balance state will be updated without updating the account delta. So the `settle` function will not be able to take into account these funds, because the difference between the previous balance state and the current state will be zero.

To summarize, the `sync` function should always be triggered before transferring the payment to the vault. After the `sync` function is called, users can transfer tokens, and only after that they can trigger the `settle` function.

A more detailed description of the Vault contract can be found in section 5.1. ↗.

4.2. Hook impact on delta consistency

This point highlights important contract behavior for users to consider.

The `swap` and `modifyLiquidity` functions of the CLPoolManager contract, as well as the `swap`, `mint`, and `burn` functions of the BinPoolManager contract, return the `delta` value that contains `amount0` and `amount1`, the specified amounts of tokens a user should provide and receive. The sign of these amounts determines the direction of the token transfer. If an amount is negative, a user provides that token; if the sign is positive, a user receives that token. Additionally, for example, when a user initiates a `swap` action, they specify the `zeroForOne` parameter. If `zeroForOne` is true, this means a user is swapping `token0` for `token1`; in other words, the user is selling `token0` and receiving `token1`. If `zeroForOne` is false, this means the user is swapping `token1` for `token0`. Accordingly, if the `ze-`

`roForOne` is true, `amount0` is negative and `amount1` is positive. Conversely, if `zeroForOne` is false, `amount1` is positive and `amount1` is negative.

However, it is important to note that this behavior is not guaranteed by the code of contracts CLPoolManager and BinPoolManager. The functionality of these contracts allows for the use of optional hook calls. These external hook calls can contain arbitrary logic controlled by the pool initializer. Specifically, after these external hook calls, the delta value may change; therefore, the sign of the amounts may also change.

The example code below illustrates that the pool contract initially returns the delta with the expected signs, but after the `afterSwap` hooks call, the delta is overwritten.

```
function swap(PoolKey memory key, ICLPoolManager.SwapParams memory params,
    bytes calldata hookData)
    external
    override
    whenNotPaused
    returns (BalanceDelta delta)
{
    [...]
    (delta, state) = pool.swap(
        CLPool.SwapParams({
            tickSpacing: key.parameters.getTickSpacing(),
            zeroForOne: params.zeroForOne,
            amountSpecified: amountToSwap,
            sqrtPriceLimitX96: params.sqrtPriceLimitX96,
            lpFeeOverride: lpFeeOverride
        })
    );

    [...]

    BalanceDelta hookDelta;
    (delta, hookDelta) = CLHooks.afterSwap(key, params, delta, hookData,
    beforeSwapDelta);
    [...]
}
```

This behavior should be taken into account when processing the delta. Always make sure that the sign corresponds to what is expected.

4.3. Users may induce misleading donations

A key feature of all pool types is the ability for users to create hooks that augment functionality of the base applications. Hooks are supported for donations as well, where they are notified with the donation amount. These donation amounts may not truly represent contributions to the pool. For instance, if a user owns a sufficiently large portion of the active liquidity, they could donate to the pool while keeping a significant amount of the funds for themselves.

Further, users could perform a flash swap to shift the price towards a range with less liquidity before performing the donation, and then they could move the price back.

Currently, none of the hooks implemented take actions on donations.

4.4. Protocol offers free flash loans

The protocol allows users to trade with more liquidity than they have — effectively letting them borrow tokens during an exchange — as long as all debts are settled at the end of the transaction. The Vault contract also includes a `take` function, which allows these borrowed tokens to be extracted and used in external exchanges. There is no fee charged for this behavior, so users can effectively take flash loans for free. There is a possibility that this functionality discourages liquidity providers from participating in the protocol.

4.5. Share accounting

Currently, the project has two pool styles implemented (Trader Joe–style bin pools and Uniswap-style concentrated liquidity pools). Naturally, each pool style has to compute balance owed to and from liquidity providers as their positions change. The accounting logic is substantially different in implementation.

Bin pools

Within a single bin, the application maintains

- a global *total reserve* of tokens,
- a global amount of *total shares* minted, and
- a per-position *share*.

Then, the proportion of the total reserve "owned" by a liquidity provider is the proportion of their shares to the total shares minted. Swap fees and donations are added to the total reserve without adjusting shares at all. When liquidity is added, the application computes

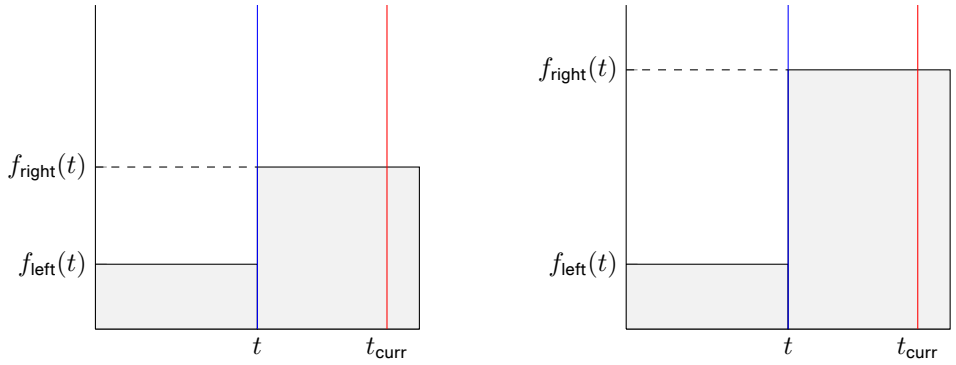
$$\text{liquidity added} \times \left(\frac{\text{total shares}}{\text{current liquidity}}\right) \leftarrow \text{shares per unit liquidity}$$

and adds that quantity to the user's share and the global total shares minted. And when liquidity is removed, the application computes the user's proportion of total shares minted, giving the user that proportion of the total reserve and removing their shares from the total shares.

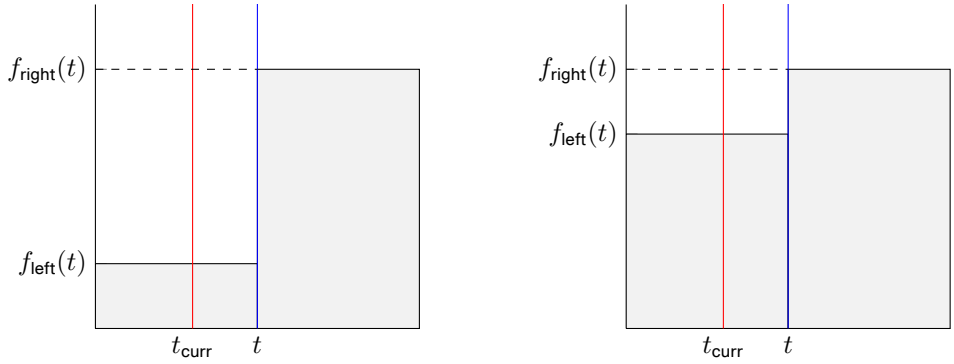
Concentrated liquidity pools

In contrast, the state maintained for concentrated liquidity ticks is different — largely because positions can span ranges of ticks. At any point in time, given a range of ticks, the application needs the ability to compute the share of fees and donations ever accrued when the tick was in the range.

To illustrate how this bookkeeping works, we will follow an example series of donations and consider the *absolute amount* of fees accrued on each side of a single tick. First, we will consider the quantities $f_{\text{left}}(t)$ and $f_{\text{right}}(t)$ for a tick t, as a fee is accrued with the current tick t_{curr} on the right.

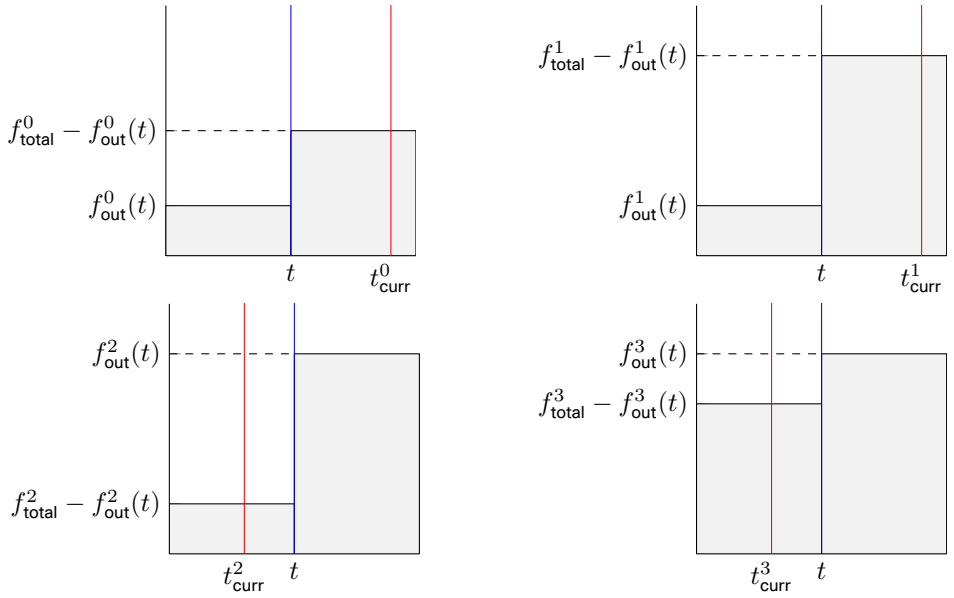


Now let us suppose that the current tick crosses to the left, and some more fees are accrued. That might look like this.



Is it feasible to track which side of a tick that fees are accrued on, for every tick simultaneously? In fact, it is. For any tick t, fees entering the pool only change one of $f_{\text{left}}(t)$ or $f_{\text{right}}(t)$, depending on which side the active tick is on. If we define $f_{\text{out}}(t)$ as the amount of fees on the opposite side (that is, $f_{\text{left}}(t)$ if $t < t_{\text{curr}}$ and $f_{\text{right}}(t)$ otherwise), then that quantity is constant with respect to fees entering the pool. And further, the data of $f_{\text{out}}(t)$ along with the total amount of fees f_{total} (a single global quantity) is sufficient to describe both sides of the tick, as the other side is simply the difference $f_{\text{total}} - f_{\text{out}}(t)$. Note that when the active tick crosses t, the application must update $f_{\text{out}}(t)$ to that difference.

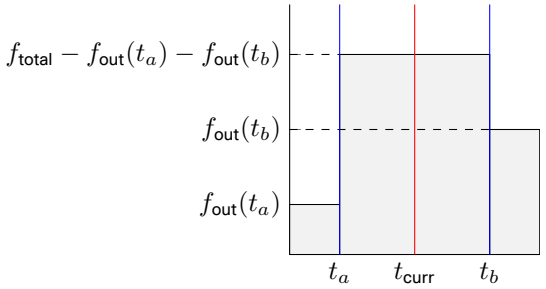
Using this scheme, the following is a sketch of how the application might account for a fee accrual, a price movement, and a second fee accrual through three time steps.



Since $f^0_{\text{out}}(t) = f^1_{\text{out}}(t)$, $f^1_{\text{total}} = f^2_{\text{total}}$, and $f^2_{\text{out}}(t) = f^3_{\text{out}}(t)$, the application only needs to perform a single update at each step to maintain this state across every tick. Namely, these are

1. $f^1_{\text{total}} = f^0_{\text{total}} +$ first fee,
2. $f^2_{\text{out}}(t) = f^1_{\text{total}} - f^1_{\text{out}}(t)$, and
3. $f^3_{\text{total}} = f^2_{\text{total}} +$ second fee.

Of course, if multiple ticks are crossed during the price movement, then (2) would have to happen for each one. Once the application maintains this information, it is easy to compute what fees were accrued with the current tick in any range. If a position provides liquidity for t_a to t_b, with the current tick between, then we have the following.



The actual accounting for this pool style is slightly more complex, and it

1. accumulates a *multiplier* instead of the *absolute* fees,
2. accounts for two different currencies in each pool, and
3. handles more cases, such as when the current tick is outside the range.

For each currency type, the pool maintains a global multiplier that represents fees accrued since the initialization of the pool. This data is analogous to f_{total} in the simplified example above. Each tick of the pool has a corresponding multiplier that denotes fees accrued outside the tick on one side, namely the side that the current tick is not on. This data is analogous to $f_{\text{out}}(t)$ in the simplified example above.

Then, for a given position, this data allows the application to determine the fee multiplier that corresponds to the lifetime earnings of a position's tick range. Each position also maintains its own saved multiplier, which represents the fees accrued until the time the position was last updated. This data allows the application to determine the amount of fees accrued in any position's tick range, since the last update, simply by subtraction.

4.6. Similarities with Uniswap v4 and Trader Joe

It is worth noting that the core protocol is extremely similar in structure and implementation to Uniswap v4. The majority of the vault and concentrated liquidity application is based on the Uniswap

v4 codebase. For example, a close inspection of the `swap` function in CLPool (whose analogous application is called Pool in Uniswap) shows that the logic is nearly identical, save for some differences in implementation style — variable names, organization, and so forth. An example of a difference in protocol logic can be found in the `modifyLiquidity` function. Uniswap conditionally checks if the maximum liquidity in a tick is reached, based on the direction of liquidity deposits.

```
if (liquidityDelta >= 0) {
    uint128 maxLiquidityPerTick
    = tickSpacingToMaxLiquidityPerTick(params.tickSpacing);
    if (state.liquidityGrossAfterLower > maxLiquidityPerTick) {
        TickLiquidityOverflow.selector.revertWith(tickLower);
    }
    if (state.liquidityGrossAfterUpper > maxLiquidityPerTick) {
        TickLiquidityOverflow.selector.revertWith(tickUpper);
    }
}
```

On the other hand, the CLPool contract checks this fact unconditionally, after moving it to the `update` function in Tick (whose counterpart is called `updateTick` in Pool).

Similarly, the bin-pool application that Uniswap does not have is closely modeled after the Trader Joe protocol. These feature more differences, because the Trader Joe protocol is not written as an application in Uniswap v4's framework. But regardless, much of the logic is virtually identical.

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

The project contains distinct components with clear security boundaries. Core to the protocol is the vault, which manages reserves and provides shared flash-swap functionality. Then, approved applications (currently concentrated liquidity and bin pools) can integrate with the vault to provide different AMM strategies. For any application, users can create pools with hooks that augment the DEX's behavior. On top of that, the protocol provides some periphery contracts to simplify interactions with the core protocol.

5.1. Vault

User interactions with the protocol begin with the vault, specifically with calls to the `lock` contract. Once this contract is called, the user receives a callback, during which they can

1. withdraw and deposit tokens from the vault,
2. interact with approved applications,
3. mint and burn vault tokens, and
4. interact with *external* projects.

In the first three types of actions, the vault keeps track of the user's debt to the protocol. For instance, if the user interacts with a concentrated liquidity pool to exchange token A for token B, it records that the user owes the vault token A, while the vault owes the user token B. After the callback completes, the vault verifies that the user's debt is settled. In this particular case, the user could accomplish this by depositing token A to the protocol and withdrawing token B.

An important feature of the vault is that separate applications have logically segmented reserves. Concretely, during actions like swaps, an application will tell the vault how much positive and negative debt to give the user. The vault will confirm that the application cannot owe the user more than the application's own reserves. This means that if the vault works as intended, vulnerabilities in one application should not affect the safety of another application's funds.

User-facing interface

The vault API exposed to the user (during the callback) consists of the following functionality.

1. The user can withdraw an underlying asset through `take`, which increments

the user's debt to the protocol.

2. The user can sync the protocol's record of its own reserves, which lets them do the following step.
3. The user can then call `settle`, which checks if the reserves increased and treats the difference as a deposit.
4. The user can also mint and burn vault tokens, which increments and decrements their debt, respectively.

So a user who wants to pay back the protocol would first call `sync`, then send the protocol some tokens, and finally call `settle` to let that difference be treated as a deposit.

Correctness

For the vault to function correctly, it should

- correctly verify that this interface is only accessible during the callback, where the contract can check the debt afterwards;
- properly track user withdrawals and deposits, with special attention that deposits cannot be double counted; and
- correctly track which tokens the user has left to settle, and properly check that their balances are resolved.

Currently, there are test cases checking that some interfaces are inaccessible outside the callback. There are also test cases verifying that the vault properly prevents a transaction from succeeding when there are unsettled balances.

Limitations

There is some slack in the `sync` and `settle` deposit pattern. For instance, if the user calls `sync` after paying the protocol, this payment is effectively forgotten. Similarly, if `sync` is not called after the protocol loses some balance (for example, from `take`), then the user will have to pay more than expected.

Application-facing interface

The primary vault interface exposed to applications is `accountAppBalanceDelta`, which updates the user's debt to the protocol while tracking the application's reserves. This function also has an implementation that handles two assets simultaneously. It also includes a `collectFee` function, which decrements reserves to be transferred out.

Correctness

To properly isolate application reserves, the protocol should never allow debt to users from `accountAppBalanceDelta` to exceed debt of users from `accountAppBalanceDelta` (for a given application and token).

We recommend adding more negative test cases for the application-facing interface. It is tested that `accountAppBalanceDelta` can only be called from authorized apps while the vault is locked, but scenarios where an application withdraws more than it deposits are not covered.

Owner-facing interface

The vault gives an owner the ability to register applications. Once registered, applications cannot be removed.

5.2. Applications

The AMM logic is largely handled by applications approved on the core vault. All actions that affect token balances — such as updates to liquidity provider positions, swaps, donations, and fee collection — do so by updating the user's debt to or from the protocol. This means they must be performed while the vault is locked, so that debts can be confirmed as settled at the end of transactions. Structuring the protocol this way allows applications to focus purely on accounting logic, while the vault handles interactions with actual tokens.

While applications can be structured differently, the two currently implemented both maintain a collection of pools in their contract state. Modulo some mild validation, any user can create a pool with an arbitrary

- currency pair,
- liquidity provider fee,
- curve parameters (such as bin size), and
- hooks.

User-facing interface

The vault requires applications to be approved, so they cannot feature arbitrary logic. But users do have control over hooks in the pool, which can augment its behavior. To take the concentrated liquidity pools as an example, an external contract can

- perform logic before and after its pool is initialized,
- modify the amount a liquidity provider is charged or paid during an update,
- modify the amount a user is charged and paid during a swap, and
- perform logic before and after donations.

Note that while the vault performs isolation on application reserves, it is not aware of application

pools. Clearly, it is important for pool reserves to be isolated — otherwise, misbehaving pools with user-controlled hooks might steal funds. This isolation is intended to be handled by the application itself.

The logic of how hooks can influence liquidity updates and swaps is written to prevent these issues. Hooks should not affect the net change in application balances during these actions; for example, after a swap, a hook can decide to give the user an additional amount of the output token — but this amount would have to come out of the hook's own funds.

Beyond ensuring the isolation of pools, the correctness of the application requires that

- only pools with valid parameters can be created;
- all actions correctly update user debt with the vault;
- fees are properly collected during swaps and cannot be bypassed through liquidity updates; and
- more generally, the accounting is done with sound economics.

Owner-facing interface

The key privileged functionality in the two applications is fee collection. The owner can delegate the ability to manage and collect protocol fees. The application awards these fees by calling `collectFee` on the vault, which directly sends funds to the recipient from the application's reserves.

For correctness, the application must prevent unauthorized users from modifying or collecting fees. Additionally, towards decentralization and safety for liquidity providers, it must prevent authorized users from withdrawing more fees than the protocol is owed.

5.3. Module: BinPoolManager.sol

Function: `burn(PoolKey key, IBinPoolManager.BurnParams params, bytes hookData)`

This removes the liquidity from the specified bins. This function allows the caller to provide several IDs of the bins from which the liquidity should be burned, along with the corresponding amounts of shares to be burned. Additionally, the caller can provide a `salt` if it was used during minting to specify the position when multiple positions are available.

Inputs

- `key`
  - **Control**: Full control.
  - **Constraints**: The pool for the provided `PoolKey` data should be initialized.
  - **Impact**: The `PoolKey` data that specified the initialized pool.
- `params`
  - **Control**: Full control.

    - **Constraints**: If any of the `amountsToBurn` is zero, the transaction will be reverted. Additionally, if the bin for the provided ID does not have sufficient reserves to burn the calculated amount of liquidity related to the specified shares, the transaction will also be reverted.
    - **Impact**: The params data contains the `ids` array, `amountsToBurn` array, and salt to specify the unique position to burn.
- `hookData`
    - **Control**: Full control.
    - **Constraints**: N/A.
    - **Impact**: Contains the data necessary for a hook call.

Branches and code coverage

Intended branches

- `feeDelta` matches the expected fee.
  ☑ Test coverage
- The delta matches with the expected delta, and `hookDelta` is zero when the `hooks` address is zero.
  ☐ Test coverage
- The delta matches with the expected delta when the `hooks` address is not zero.
  ☐ Test coverage
- `hookDelta` matches with the expected delta when the `hooks` address is not zero.
  ☐ Test coverage
- The bin reserve is updated as expected.
  ☑ Test coverage
- The caller's share balance is decreased by the provided amount of shares.
  ☐ Test coverage
- The bin ID is removed from the tree after all liquidity was burned from it.
  ☐ Test coverage
- The burn with salt
  ☑ Test coverage
- The burn can be executed during pause
  ☑ Test coverage

Negative behavior

- Locker is not set.
  ☐ Negative test
- `amountsToBurn` is zero.
  ☐ Negative test
- Pool is not initialized.
  ☑ Negative test
- Incorrect salt.
  ☐ Negative test

- The vault balance is not updated properly by delta and `hookDelta` at the end of the call.
  - ☐ Negative test

Function call analysis

- `BinPool.checkPoolInitialized(pool)`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** The function reverts if `activeId` of the pool is zero.
- `BinHooks.beforeBurn(key, params, hookData)`
  - **What is controllable?** `key`, `params`, and `hookData`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** If `HOOKS_BEFORE_BURN_OFFSET` for the pool is set, the `beforeBurn` function of the `hooks` contract will be called. The `hooks` contract is fully controlled by the initializer of the pool.
- `BinPool.burn(pool, BurnParams(from: msg.sender, ids: params.ids, amountsToBurn: params.amountsToBurn, salt: params.salt))`
  - **What is controllable?** `params.ids`, `params.amountsToBurn`, and `params.salt`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns the amount of delta that determines the amount of `token0` and `token1` that the user should receive as a result of burning the provided amount of shares.
  - **What happens if it reverts, reenters or does other unusual control flow?** The function reverts if any of the provided amount of shares is zero — also if the bin for the provided ID does not have sufficient reserves to burn the calculated amount of liquidity related to the specified shares. And additionally, the transaction will be reverted if the position of the caller does not have sufficient amount of shares to be burned.
- `BinHooks.afterBurn(key, params, delta, hookData)`
  - **What is controllable?** `key`, `params`, and `hookData`.
  - **If the return value is controllable, how is it used and how can it go wrong?** The function returns the modified delta for the caller and new delta for the hooks contract. The caller's delta can change the sign as a result of modification.
  - **What happens if it reverts, reenters or does other unusual control flow?** If `HOOKS_AFTER_BURN_OFFSET` is set for the pool, the `afterBurn` function of the `hooks` contract will be called. This function returns the new delta, but this delta will be taken into account only if the `HOOKS_AFTER_BURN_RETURNS_DELTA_OFFSET` is set for the pool.
- `this.vault.accountAppBalanceDelta(key, hookDelta, address(key.hooks))`

- **What is controllable?** `key` and `hookDelta` are controlled by the hooks contract. The `key.hooks` is controlled by the initializer of the pool.
- **If the return value is controllable, how is it used and how can it go wrong?** N/A.
- **What happens if it reverts, reenters or does other unusual control flow?** It calls the Vault contract to account for changes in the pool balance, updates the balance for the app, updates delta for the `hooks` address, and updates the `UNSETTLED_DELTAS_COUNT` if the current delta was changed from zero or the next delta was set to zero. The `UNSETTLED_DELTAS_COUNT` should be zero at the end of the transactions.

- `this.vault.accountAppBalanceDelta(key, delta, msg.sender)`
  - **What is controllable?** `key`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** It calls the Vault contract to account for changes in the pool balance, updates the balance for the app, updates delta for the `msg.sender` address, and updates the `UNSETTLED_DELTAS_COUNT` if the current delta was changed from zero or the next delta was set to zero. The `UNSETTLED_DELTAS_COUNT` should be zero at the end of the transactions.

Function: `donate(PoolKey key, uint128 amount0, uint128 amount1, bytes hookData)`

The function allows to donate the provided amounts of tokens to the pool. The `reserveOfBin[activeId]` is updated directly by adding the `amount0` and `amount1`.

Inputs

- `key`
  - **Control**: Full control.
  - **Constraints**: The pool for the provided `PoolKey data` should be initialized.
  - **Impact**: The `PoolKey` data that specified the initialized pool.
- `amount0`
  - **Control**: Full control.
  - **Constraints**: The caller should send to the vault an appropriate amount of `token0`.
  - **Impact**: The amount of the donation of `token0`.
- `amount1`
  - **Control**: Full control.
  - **Constraints**: The caller should send to the vault an appropriate amount of `token1`.
  - **Impact**: The amount of the donation of `token1`.

- `hookData`
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: The data provided to the `beforeDonate` and `afterDonate` functions' call of the `hooks` contract.

Branches and code coverage

Intended branches

- The pool is updated properly after donation.
  - ☐ Test coverage
- The `reserveOfBin` for appropriate `activeId` after update matches with the expected values.
  - ☐ Test coverage

Negative behavior

- Call without the locking of the Vault contract.
  - ☐ Negative test
- Pool is not initialized.
  - ☑ Negative test
- `binReserves` is zero.
  - ☑ Negative test
- `donate` when paused
  - ☑ Negative test

Function call analysis

- `CLHooks.beforeDonate(key, amount0, amount1, hookData)`
  - **What is controllable?** `key`, `amount0`, `amount1`, and `hookData`
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** If the `HOOKS_BEFORE_DONATE_OFFSET` flag is set in the pool settings, the `beforeDonate` function of the arbitrary hooks contract will be called with the following parameters: `msg.sender`, `key`, `amount0`, `amount1`, and `hookData`.
- `CLPool.donate(pool, amount0, amount1)`
  - **What is controllable?** `amount0` and `amount1`.
  - **If the return value is controllable, how is it used and how can it go wrong?** The `delta` contains the negatives `amount0` and `amount1`; the negative sign determines that caller should sent the appropriate amounts of `token0` and `token1` to the vault.
  - **What happens if it reverts, reenters or does other unusual control flow?** The

function can revert if the `reserveOfBin` for the appropriate `activeId` is zero.

- `this.vault.accountAppBalanceDelta(key, delta, msg.sender)`
  - **What is controllable?** `key` and `delta`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** It calls the Vault contract to account for changes in the pool balance, updates the balance for the app, updates delta for the `msg.sender` address, and updates the `UNSETTLED_DELTAS_COUNT` if the current delta was changed from zero or the next delta was set to zero. The `UNSETTLED_DELTAS_COUNT` should be zero at the end of the transactions.
- `CLHooks.afterDonate(key, amount0, amount1, hookData)`
  - **What is controllable?** `key`, `amount0`, `amount1`, and `hookData`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** If the `HOOKS_AFTER_DONATE_OFFSET` flag is set in the pool settings, the `afterDonate` function of the arbitrary hooks contract will be called with the following parameters: `msg.sender`, `key`, `amount0`, `amount1`, and `hookData`.

Function: `initialize(PoolKey key, uint24 activeId, bytes hookData)`

The function allows any caller to initialize a new pool.

Inputs

- `key`
  - **Control**: Full control.
  - **Constraints**: The `poolManager` should be equal to this contract address; `lpFee` should not exceed the 10%. The `binStep` should be in a range from one to 100.
  - **Impact**: The key object contains the address of `currency0`, the address of `currency1`, the `hooks` address, the `poolManager` address, the fee amount and parameters. The parameters contain the `binStep`.
- `activeId`
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: ID of the active bin of the new pool.
- `hookData`
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: Contains the arbitrary data necessary for a hook call.

Branches and code coverage

Intended branches

- The pool is initialized successfully when `hookData` is empty.
  - ☑ Test coverage

Negative behavior

- The `poolManager` does not match this contract address.
  - ☐ Negative test
- `binStep` is less than one.
  - ☑ Negative test
- `binStep` is more than 100.
  - ☑ Negative test
- `initialize` the same pool twice
  - ☑ Test coverage
- Invalid fee
  - ☑ Test coverage

Function call analysis

- `BinPoolParametersHelper.getBinStep(key.parameters)`
  - **What is controllable?** the parameters of the `key`.
  - **If the return value is controllable, how is it used and how can it go wrong?** This is a pure function of the bin pool parameters. The return value should be validated to be a sensible bin step.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `Hooks.validateHookConfig(key)`
  - **What is controllable?** `key`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** The function validates that if a pool does not support hooks (i.e., `key.hook = address(0)`), the LP fee is not dynamic. Otherwise, the bitmap from `key.parameters` should match the return value from the call to `key.hooks.getHooksRegistrationBitmap()`.
- `BinHooks.validatePermissionsConflict(key)`
  - **What is controllable?** `key`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** The function validates that hook parameters are set properly (e.g., if the `HOOKS_BEFORE_SWAP_RETURNS_DELTA_OFFSET` is set, the `HOOKS_BEFORE_SWAP_OFFSET` should also be set.

- `BinHooks.beforeInitialize(key, activeId, hookData)`
  - **What is controllable?** `key`, `activeId`, and `hookData`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** If `HOOKS_BEFORE_INITIALIZE_OFFSET` is enabled and the hook address is not equal to this contract address, the `beforeInitialize` function of the hooks contract will be called with these arguments: `msg.sender`,`key`, `activeId`, and `hookData`. The `hook.beforeInitialize` function does not return anything. The `hooks` contract is fully controlled by the caller.
- `this._fetchProtocolFee(key)`
  - **What is controllable?** `key`.
  - **If the return value is controllable, how is it used and how can it go wrong?** The function returns success status and the `protocolFee` amount. The `initialize` function ignores the success status. The `protocolFee` is validated that it does not exceed 0.1% (1,000).
  - **What happens if it reverts, reenters or does other unusual control flow?** If the `protocolFeeController` address was set by the owner of contract, the `protocolFeeForPool` function of `protocolFeeController` will be called with the `key` parameter. If the `protocolFeeController` address is zero, this function returns `false, 0`).
- `BinPool.initialize(this.pools[id], activeId, protocolFee, lpFee)`
  - **What is controllable?** `activeId` and `lpFee`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** If `activeId` for this pool is not zero, this pool has been already initialized. As a result, the function will be reverted.
- `BinHooks.afterInitialize(key, activeId, hookData)`
  - **What is controllable?** `key`, `activeId`, and `hookData`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** If `HOOKS_AFTER_INITIALIZE_OFFSET` is enabled and the hook address is not equal to this contract address, the `afterInitialize` function of the hooks contract will be called with these arguments: `msg.sender`, `key`, `activeId`, and `hookData`. The `hook.afterInitialize` function does not return anything. The `hooks` contract is fully controlled by the caller.

Function: `mint(PoolKey key, IBinPoolManager.MintParams params, bytes hookData)`

The function adds liquidity to the pool. The pool ID is determined by the provided `PoolKey` data. Additionally, the caller provides `liquidityConfigs`, the amount of liquidity to be added, and salt to

specify the different mint position for the same owner. The `liquidityConfigs` contains the arrays of IDs of the bins that should be updated by the added liquidity and amount of liquidity to be added.

Inputs

- `key`
  - **Control**: Full control.
  - **Constraints**: The pool for the provided `PoolKey` data should be initialized.
  - **Impact**: The `PoolKey` data that specified the initialized pool.
- `params`
  - **Control**: Full control.
  - **Constraints**: The desired amount of liquidity to be added to the pool should not overflow `liquidity`.
  - **Impact**: The params data contains `liquidityConfigs` and `amountIn` arrays and salt to specify the unique position to mint.
- `hookData`
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: Contains the data necessary for a hook call.

Branches and code coverage

Intended branches

- The delta matches with the expected delta, and `hookDelta` is zero when the `hooks` address is zero.
  - ☑ Test coverage
- The delta matches with the expected delta when the `hooks` address is not zero.
  - ☐ Test coverage
- `hookDelta` matches with the expected delta when the `hooks` address is not zero.
  - ☐ Test coverage
- The bin reserve is updated as expected.
  - ☑ Test coverage
- The caller's share balance is increased as expected.
  - ☐ Test coverage

Negative behavior

- Locker is not set.
  - ☐ Negative test
- `amountIn` is zero
  - ☐ Negative test
- Pool is not initialized.
  - ☑ Negative test

- The vault balance is not updated properly by delta and `hookdelta` at the end of the call.
  - ☑ Negative test
- `mint` when paused
  - ☑ Negative test
- Empty config
  - ☑ Negative test
- Zero shares
  - ☑ Negative test

Function call analysis

- `BinPool.checkPoolInitialized(pool)`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** The function reverts if `activeId` of the pool is zero.
- `BinHooks.beforeMint(key, params, hookData)`
  - **What is controllable?** `key`, `params`, and `hookData`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** If `HOOKS_BEFORE_MINT_OFFSET` for the pool is set, the `beforeMint` function of the `hooks` contract will be called. The `hooks` contract is fully controlled by the initializer of the pool.
- `BinPool.mint(pool, MintParams(to: msg.sender, liquidityConfigs: params.liquidityConfigs, amountIn: params.amountIn, binStep: BinPoolParametersHelper.getBinStep(key.parameters), salt: params.salt))`
  - **What is controllable?** `liquidityConfigs`, `amountIn`, and `salt`.
  - **If the return value is controllable, how is it used and how can it go wrong?** The function returns delta, `feeForProtocol`, `mintArray`, and `compositionFee`. The `delta` determines the amount of `token0` and `token1` that a user should provide to the vault to add the desired amount of liquidity to the pool, so if the delta is calculated incorrectly, the user will provide an excessive or insufficient number of tokens.
  - **What happens if it reverts, reenters or does other unusual control flow?** If `liquidityConfigs`'s length is zero, the transaction will be reverted. If the amount of liquidity desired to be added to the pool is too big, the `getLiquidity` function will revert as a result of `liquidity` overflow.
- `BinHooks.afterMint(key, params, delta, hookData)`
  - **What is controllable?** `key`, `params`, and `hookData`.
  - **If the return value is controllable, how is it used and how can it go wrong?** The function returns the modified delta for the caller and new delta for the hooks contract. The caller's delta can change the sign as a result of modification.

- **What happens if it reverts, reenters or does other unusual control flow?** If `HOOKS_AFTER_MINT_OFFSET` is set for the pool, the `afterMint` function of the `hooks` contract will be called. This function returns the new delta, but this delta will be taken into account only if the `HOOKS_AFTER_MINT_RETURNS_DELTA_OFFSET` is set for the pool.

- `this.vault.accountAppBalanceDelta(key, hookDelta, address(key.hooks))`
  - **What is controllable?** `key` and `hookDelta` are controlled by the hooks contract. The `key.hooks` is controlled by the initializer of the pool.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** It calls the Vault contract to account for changes in the pool balance, updates the balance for the app, updates the delta for the `hooks` address, and updates the `UNSETTLED_DELTAS_COUNT` if the current delta was changed from zero or the next delta was set to zero. The `UNSETTLED_DELTAS_COUNT` should be zero at the end of the transactions.
- `this.vault.accountAppBalanceDelta(key, delta, msg.sender)`
  - **What is controllable?** `key`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** It calls the Vault contract to account for changes in the pool balance, updates the balance for the app, updates delta for the `msg.sender` address, and updates the `UNSETTLED_DELTAS_COUNT` if the current delta was changed from zero or the next delta was set to zero. The `UNSETTLED_DELTAS_COUNT` should be zero at the end of the transactions.

Function: `updateDynamicLPFee(PoolKey key, uint24 newDynamicLPFee)`

The function allows the hooks contract to update the LP fee of the pool, if it supports the `DynamicLPFee`.

Inputs

- `key`
  - **Control**: Full control.
  - **Constraints**: The pool for the provided `PoolKey` data should be initialized. The pool should support `DynamicLPFee`.
  - **Impact**: The `PoolKey` data that specified the initialized pool.
- `newDynamicLPFee`
  - **Control**: Full control.
  - **Constraints**: Cannot exceed 10%.
  - **Impact**: The LPF fee determines the fee that will be charged during the swap

and mint.

Branches and code coverage

Intended branches

- The LP fee is updated properly for the appropriate pool.
  - ☑ Test coverage

Negative behavior

- The new LP fee is more than 10%.
  - ☑ Negative test
- The caller is not a hooks contract.
  - ☐ Negative test
- The pool doesn't support dynamic fee
  - ☑ Negative test

Function call analysis

- `BinPool.setLPFee(this.pools[id], newDynamicLPFee)`
  - **What is controllable?** `newDynamicLPFee`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** The function will revert if the pool is not initialized.

5.4. Module: CLPoolManager.sol

Function: `donate(PoolKey key, uint256 amount0, uint256 amount1, bytes hookData)`

The function allows to donate the provided amount of tokens to the pool.

Inputs

- `key`
  - **Control**: Full control.
  - **Constraints**: The pool for the provided `PoolKey` data should be initialized.
  - **Impact**: The `PoolKey` data that specified the initialized pool.
- `amount0`
  - **Control**: Full control.
  - **Constraints**: The caller should send to the vault an appropriate amount of `to-`

ken0.
    - **Impact**: The amount of the donation of `token0`.
- `amount1`
    - **Control**: Full control.
    - **Constraints**: The caller should send to the vault an appropriate amount of `token1`.
    - **Impact**: The amount of the donation of `token1`.
- `hookData`
    - **Control**: Full control.
    - **Constraints**: N/A.
    - **Impact**: The data provided to the `beforeDonate` and `afterDonate` functions' call of the `hooks` contract.

Branches and code coverage

Intended branches

- The pool is updated properly after donation.
    - ☐ Test coverage
- The `feeGrowthGlobal0X128` and `feeGrowthGlobal1X128` after update matches with the expected values.
    - ☑ Test coverage

Negative behavior

- Call without locking of the Vault contract.
    - ☐ Negative test
- The pool is not initialized.
    - ☑ Negative test
- Liquidity of the pool is zero.
    - ☑ Negative test
- `donate` when paused
    - ☑ Negative test

Function call analysis

- `CLHooks.beforeDonate(key, amount0, amount1, hookData)`
    - **What is controllable?** `key`, `amount0`, `amount1`, and `hookData`.
    - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
    - **What happens if it reverts, reenters or does other unusual control flow?** If the `HOOKS_BEFORE_DONATE_OFFSET` flag is set in the pool settings, the `beforeDonate` function of the arbitrary hooks contract will be called with the following parameters: `msg.sender`, `key`, `amount0`, `amount1`, and `hookData`.

- `CLPool.donate(pool, amount0, amount1)`
  - **What is controllable?** `amount0` and `amount1`.
  - **If the return value is controllable, how is it used and how can it go wrong?** The `delta` contains the negatives `amount0` and `amount1`; the negative sign determines that the caller should send the appropriate amounts of `token0` and `token1` to the vault. The `tick` is the current pool tick; the function does not control this tick and returns it from the storage.
  - **What happens if it reverts, reenters or does other unusual control flow?** The function can revert if the current pool liquidity is zero.
- `this.vault.accountAppBalanceDelta(key, delta, msg.sender)`
  - **What is controllable?** `key` and `delta`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** It calls the Vault contract to account for changes in the pool balance, updates the balance for the app, updates delta for the `msg.sender` address, and updates the `UNSETTLED_DELTAS_COUNT` if the current delta was changed from zero or the next delta was set to zero. The `UNSETTLED_DELTAS_COUNT` should be zero at the end of the transactions.
- `CLHooks.afterDonate(key, amount0, amount1, hookData)`
  - **What is controllable?** `key`, `amount0`, `amount1`, and `hookData`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** If the `HOOKS_AFTER_DONATE_OFFSET` flag is set in the pool settings, the `afterDonate` function of the arbitrary hooks contract will be called with the following parameters: `msg.sender`, `key`, `amount0`, `amount1`, and `hookData`.

Function: `initialize(PoolKey key, uint160 sqrtPriceX96, bytes hookData)`

The function allows any caller to initialize a new pool.

Inputs

- `key`
  - **Control**: Full control.
  - **Constraints**: The `poolManager` should be equal to this contract address; `lpFee` should not exceed 100%.
  - **Impact**: The key object contains the address of `currency0`, the address of `currency1`, the `hooks` address, the `poolManager` address, the fee amount, and parameters.
- `sqrtPriceX96`
  - **Control**: Full control.

  - **Constraints**: The `sqrtPriceX96` should not be zero and should be in the range from `4295128739` to `1461446703485210103287273052203988822378723970342`.
  - **Impact**: The current price.
- `hookData`
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: Contains the data necessary for a hook call.

Branches and code coverage

Intended branches

- The pool is initialized successfully when `hookData` is empty.
  - ☑ Test coverage
- The tick for the provided `sqrtPriceX96` matches with the expected tick.
  - ☑ Test coverage

Negative behavior

- `sqrtPriceX96` is zero.
  - ☐ Negative test
- `sqrtPriceX96 < 4295128739`.
  - ☐ Negative test
- `sqrtPriceX96` >= than `1461446703485210103287273052203988822378723970342`.
  - ☐ Negative test
- The `poolManager` does not match this contract address.
  - ☐ Negative test
- pool already exists
  - ☑ Negative test
- tickSpacing > MAX_TICK_SPACING
  - ☑ Negative test
- tickSpacing < MIN_TICK_SPACING
  - ☑ Negative test
- currency0 >= currency1
  - ☑ Negative test

Function call analysis

- `CLPoolParametersHelper.getTickSpacing(key.parameters)`
  - **What is controllable?** `key.parameters`.
  - **If the return value is controllable, how is it used and how can it go wrong?** The value is controllable, but there is a validation that `TickSpacing` is in the range from one to `type(int16).max`.

- **What happens if it reverts, reenters or does other unusual control flow?** The function decodes the input `bytes32` user's `parameters`, gets the `uint24` `TickSpacing`, casts it to `int24`, and returns the `int24` `TickSpacing`.

- `ParametersHelper.checkUnusedBitsAllZero(key.parameters, CLPoolParametersHelper.OFFSET_MOST_SIGNIFICANT_UNUSED_BITS)`
  - **What is controllable?** `key.parameters`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** The function validates that unused and unnecessary bytes are actually zero.
- `Hooks.validateHookConfig(key)`
  - **What is controllable?** `key`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** The function validates that if a pool does not support hooks (i.e., `key.hook = address(0)`), the LP fee is not dynamic. Otherwise, the bitmap from `key.parameters` should match the return value from the call to `key.hooks.getHooksRegistrationBitmap()`.
- `CLHooks.validatePermissionsConflict(key)`
  - **What is controllable?** `key`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** The function validates that hook parameters are set properly (e.g., if the `HOOKS_BEFORE_SWAP_RETURNS_DELTA_OFFSET` is set, the `HOOKS_BEFORE_SWAP_OFFSET` also should be set).
- `LPFeeLibrary.getInitialLPFee(key.fee)`
  - **What is controllable?** `key.fee`.
  - **If the return value is controllable, how is it used and how can it go wrong?** There is a validation that the LP fee does not exceed 100% (i.e., 1,000,000).
  - **What happens if it reverts, reenters or does other unusual control flow?** If the `DYNAMIC_FEE_FLAG` flag is set in params, it means that the LP fee is a dynamic fee, so the function will return zero fee.
- `LPFeeLibrary.validate(lpFee, LPFeeLibrary.ONE_HUNDRED_PERCENT_FEE)`
  - **What is controllable?** `lpFee`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** The function validates that the provided LP fee does not exceed 100%.
- `CLHooks.beforeInitialize(key, sqrtPriceX96, hookData)`
  - **What is controllable?** `key`, `sqrtPriceX96`, and `hookData`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.

- **What happens if it reverts, reenters or does other unusual control flow?** If `HOOKS_BEFORE_INITIALIZE_OFFSET` is enabled and the `hook` address is not equal to this contract address, the `beforeInitialize` function of the hooks contract will be called with these arguments: `msg.sender`, `key`, `sqrtPriceX96`, and `hookData`. The `hook.beforeInitialize` function does not return anything. The `hooks` contract is fully controlled by the caller.

- `this._fetchProtocolFee(key)`
  - **What is controllable?** `key`.
  - **If the return value is controllable, how is it used and how can it go wrong?** The function returns success status and the `protocolFee` amount. The `initialize` function ignores the success status. The `protocolFee` is validated that it does not exceed 0.1% (1,000).
  - **What happens if it reverts, reenters or does other unusual control flow?** If the `protocolFeeController` address was set by the owner of the contract, the `protocolFeeForPool` function of `protocolFeeController` will be called with the `key` parameter. If the `protocolFeeController` address is zero, this function returns `false, 0`).
- `CLPool.initialize(this.pools[id], sqrtPriceX96, protocolFee, lpFee)`
  - **What is controllable?** `sqrtPriceX96` and `lpFee`.
  - **If the return value is controllable, how is it used and how can it go wrong?** The function returns the `tick` that is calculated using the `getTickAtSqrtRatio` function for `sqrtPriceX96` provided by the caller.
  - **What happens if it reverts, reenters or does other unusual control flow?** If `sqrtPriceX96` is zero or exceeds the range, the function will revert. Also, the function creates the `Slot0` object with `sqrtPriceX96`, `tick`, `protocolFee`, and `lpFee`.
- `CLHooks.afterInitialize(key, sqrtPriceX96, tick, hookData)`
  - **What is controllable?** `key`, `sqrtPriceX96`, and `hookData`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** If `HOOKS_AFTER_INITIALIZE_OFFSET` is enabled and the `hook` address is not equal to this contract address, the `afterInitialize` function of the hooks contract will be called with these arguments: `msg.sender`, `key`, `sqrtPriceX96`, `tick`, and `hookData`. The `hook.afterInitialize` function does not return anything. The `hooks` contract is fully controlled by the caller.

Function: `modifyLiquidity(PoolKey key, ICLPoolManager.ModifyLiquidityParams params, bytes hookData)`

The function allows any user to add or remove liquidity from the position of the pool. The contract should not be paused to add liquidity.

Inputs

- `key`
  - **Control**: Full control.
  - **Constraints**: The pool for the provided `PoolKey` data should be initialized.
  - **Impact**: The `PoolKey` data that specified the initialized pool.
- `params`
  - **Control**: Full control.
  - **Constraints**: If `params.liquidityDelta` is more than `int128` during cast, the `toInt128` function will revert in the case of overflow or underflow — `tickLower` should be less than `tickUpper`, `tickLower` should not be less than -887,272, and `tickUpper` should not be more than 887,272.
  - **Impact**: The params data contains `tickLower`, `tickUpper`, `liquidityDelta`, and salt to specify the unique liquidity position at the same range.
- `hookData`
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: Contains the data necessary for a hook call.

Branches and code coverage

Intended branches

- `feeDelta` matches with the expected fee.
  - ☐ Test coverage
- The delta matches with the expected delta, and `hookDelta` is zero when the `hooks` address is zero.
  - ☐ Test coverage
- The delta matches with the expected delta when the `hooks` address is not zero.
  - ☐ Test coverage
- `hookDelta` matches with the expected delta when the `hooks` address is not zero.
  - ☐ Test coverage
- The liquidity position was updated as expected.
  - ☐ Test coverage

Negative behavior

- Call `modifyLiquidity` when locker is not set.
  - ☐ Negative test
- The vault balance is not updated properly by delta and `hookdelta` at the end of the call.
  - ☐ Negative test
- `tickLower` is less than -887,272.
  - ☑ Negative test
- `tickUpper` is more than 887,272.
  - ☑ Negative test

- The new liquidity updated by `liquidityDelta` is more than `maxLiquidity`.
  - ☑ Negative test

Function call analysis

- `CLPool.checkPoolInitialized(pool)`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** The function reverts if `sqrtPriceX96` of the provided pool is zero.
- `CLHooks.beforeModifyLiquidity(key, params, hookData)`
  - **What is controllable?** `key`, `params`, and `hookData`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** If `liquidityDelta` > 0, the `beforeAddLiquidity` function of the hooks contract will be called with `msg.sender`, `key`, `params`, and `hookData` params, otherwise the `beforeRemoveLiquidity` with the same params. The `hooks` contract is fully controlled by the pool creator. The `hook.beforeAddLiquidity` and `hook.beforeRemoveLiquidity` functions do not return anything.
- `CLPool.modifyLiquidity(pool, ModifyLiquidityParams(owner: msg.sender, tickLower: params.tickLower, tickUpper: params.tickUpper, liquidityDelta: SafeCast.toInt128(params.liquidityDelta), tickSpacing: CLPoolParametersHelper.getTickSpacing(key.parameters), salt: params.salt))`
  - **What is controllable?** `params.tickLower`, `params.tickUpper`, `params.liquidityDelta`, and `tickSpacing`.
  - **If the return value is controllable, how is it used and how can it go wrong?** The function returns the delta amount, which determines how many tokens the user should provide/receive as a result of this operation, so if the function returns the wrong delta, the user will be able to provide/receive more or less tokens than they should.
  - **What happens if it reverts, reenters or does other unusual control flow?** The function can revert if `tickLower` or `tickUpper` incorrect — also this function will revert if the new liquidity updated by the `liquidityDelta` exceeds the `maxLiquidityPerTick`, which is calculated using the provided `tickSpacing`.
- `SafeCast.toInt128(params.liquidityDelta)`
  - **What is controllable?** `params.liquidityDelta`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** The function will revert if `liquidityDelta` exceeds the `int128`.
- `CLHooks.afterModifyLiquidity(key, params, delta + feeDelta, hookData)`
  - **What is controllable?** `key`, `params`, and `hookData`.

- **If the return value is controllable, how is it used and how can it go wrong?** The function returns the modified delta. The delta can change the sign as a result of modification.
- **What happens if it reverts, reenters or does other unusual control flow?** If `liquidityDelta` > 0, the `afterAddLiquidity` function of the hooks contract will be called with `msg.sender`, `key`, `params`, delta, and `hookData` params, otherwise the `afterRemoveLiquidity` with the same params. The `hooks` contract is fully controlled by the pool creator. The `hook.beforeAddLiquidity` and `hook.beforeRemoveLiquidity` functions return the `hookDelta`. The initial `callerDelta` will be decreased by `hookDelta`. Because the `callerDelta` has int type (i.e., it is sign value), the sign of the initial delta can change as a result of decreasing by `hookDelta`. If both `hookDelta` and `callerDelta` are negative, the result of subtracting can be positive if the absolute value of the `hookDelta` is greater than the absolute value of the `callerDelta`. Conversely, if both `callerDelta` and `hookDelta` are positive, the result of subtracting can be negative, if the absolute value of the `hookDelta` is greater than the absolute value of the `callerDelta`.

- `this.vault.accountAppBalanceDelta(key, hookDelta, address(key.hooks))`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** It calls the Vault contract to account for changes in the pool balance, updates the balance for the app, updates delta for the `hooks` address, and updates the `UNSETTLED_DELTAS_COUNT` if the current delta was changed from zero or the next delta was set to zero. The `UNSETTLED_DELTAS_COUNT` should be zero at the end of the transactions.
- `this.vault.accountAppBalanceDelta(key, delta, msg.sender)`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** It calls the Vault contract to account for changes in the pool balance, updates the balance for the app, updates delta for the `msg.sender` address, and updates the `UNSETTLED_DELTAS_COUNT` if the current delta was changed from zero or the next delta was set to zero. The `UNSETTLED_DELTAS_COUNT` should be zero at the end of the transactions.

Function: `swap(PoolKey key, ICLPoolManager.SwapParams params, bytes hookData)`

The function allows to perform the swap of two currencies.

The user provides the following parameters: the key of the pool that the user wants to use for the swap, the parameters of the swap, and arbitrary `hookData` that is used for the hook call. The pool

corresponding to the key should be initialized before the call and must contain sufficient liquidity. Additionally, the address of the hooks contract is not controlled by the user who initiates the swap, but it is controlled by the pool initializer. Therefore, this `hooks` contract contains arbitrary third-party code. Thus, the user should trust the pool that they intend to use.

User-controlled `swap` parameters include `zeroForOne`, `amountSpecified`, and `sqrtPriceLimitX96`. The `zeroForOne` is a boolean flag that determines the direction of the swap. If `zeroForOne` is true, the user is swapping `token0` (input token) for `token1` (output token); otherwise, the user is swapping `token1` for `token0`. The `amountSpecified` is a signed `int256` that indicates the amount of the token for swapping. If this amount is negative, it represents the input amount for the swap; if it is positive, it represents the output amount. In the first case, the caller does not spend more than `amountSpecified`, and in the second case, they receive this amount as a result of the swap. The `sqrtPriceLimitX96` specifies the price limit. If `zeroForOne` is true, the current pool price cannot be less than the `sqrtPriceLimitX96`, which must also be greater than the minimum price, as determined by the `MIN_SQRT_RATIO` constant. If `zeroForOne` is false, the current pool price cannot exceed `sqrtPriceLimitX96`, which must also be less than the maximum price, as determined by the `MAX_SQRT_RATIO` constant.

The `protocolFee` specifed by the `protocolFeeController` encodes two protocol fee amounts.

If `zeroForOne` is true, the `protocolFee` for the current swap is represented by the lower 12 bits of `slot0Start.protocolFee`.

```
0b111111111111111111111111
  000000000000............
```

If `zeroForOne` is false, the `protocolFee` for the current swap is represented by the higher 12 bits of `slot0Start.protocolFee`.

```
0b111111111111111111111111
  ............000000000000
```

The main swap logic is executed in the while loop. The ending condition of the loop is `amountSpecifiedRemaining` is zero or `sqrtPriceX96` is equal to the `sqrtPriceLimitX96`. Also, there is a local `step` object that keeps the state of every loop step:

```
    struct StepComputations {
    // the price at the beginning of the step
    uint160 sqrtPriceStartX96;
    // the next tick to swap to from the current tick in the swap
    int24 tickNext;
    // whether tickNext is initialized or not
    bool initialized;
    // sqrt(price) for the next tick (1/0)
    uint160 sqrtPriceNextX96;
    // how much is being swapped in in this step
```

```
    uint256 amountIn;
    // how much is being swapped out
    uint256 amountOut;
    // how much fee is being paid in
    uint256 feeAmount;
}
```

Every loop step, the `nextInitializedTickWithinOneWord` function returns the `tickNext` initialized or uninitialized tick up to 256 ticks away from the current tick, as well as the status of this tick, using the current tick `state.tick`, `tickSpacing`, and `zeroForOne`. The `tickSpacing` determines the interval between allowable tick values. Smaller `tickSpacing` values allow for more tick intervals but require more computations and storage. Larger `tickSpacing` values can reduce the number of allowable ticks, which may decrease gas costs when interacting with the pool. If `zeroForOne` is true (`lte` is true), search for the next initialized tick to the left (less than or equal to the starting tick).

```
bitmask = 0b00000000000000000000000000001100
                    ^
                current tick
                        ------------------>
                The search is going this way
```

Otherwise, search on the right side.

The `tickNext` is verified that it does not exceed the min/max tick, and it is used for the `sqrtPriceNextX96` next tick price calculations. Further, the `computeSwapStep` function computes the result of swapping for the given arguments:

```
(state.sqrtPriceX96, step.amountIn, step.amountOut, step.feeAmount)
    = SwapMath.computeSwapStep(
    state.sqrtPriceX96,
    SwapMath.getSqrtPriceTarget(zeroForOne, step.sqrtPriceNextX96,
    sqrtPriceLimitX96),
    state.liquidity,
    state.amountSpecifiedRemaining,
    state.swapFee
);
```

After that, in the case of `exactInput`, the `amountSpecifiedRemaining` is increased by `amountIn + feeAmount` and `amountCalculated` is increased by `amountOut`. In this case, the `amountSpecifiedRemaining` is negative, so it will be increasing until it reaches zero (or `pricelimit` will be reached and the loop will complete). Otherwise, `tate.amountSpecifiedRemaining` is decreased by `amountOut`, and `amountCalculated` is decreased by `amountIn + feeAmount`. The `amountSpecifiedRemaining` is positive in this case, so it will be decreasing until it reaches zero (or the price limit).

When `amountSpecifiedRemaining` becomes zero or `sqrtPriceLimitX96` is reached, the pool state

will be updated; `self.slot0.sqrtPriceX96`, `self.slot0.tick`, and `self.liquidity` are updated by new values `state.sqrtPriceX96`, `state.tick`, and `state.liquidity`. Also, the global fee is updated, namely `self.feeGrowthGlobal0X128` or `self.feeGrowthGlobal1X128` depending on which side received the fee.

The resulting delta is calculated as follows.

- If `zeroForOne` is true and `exactInput` is also true, the `amount0` determines the input token and calculates the difference between the `amountSpecified` and `amountSpecifiedRemaining`, and the resulting amount will be negative. The `amount1` determines the output token and is equal to the `amountCalculated`, and this amount is positive.
- If `zeroForOne` is false and `exactInput` also is false, the `amount0` determines the output token and calculates the difference between the `amountSpecified` and `amountSpecifiedRemaining`, and the resulting amount will be positive. The `amount1` determines the input token and is equal to the `amountCalculated`, and this amount is negative.
- Otherwise, if `zeroForOne` is true and `exactInput` is false, the `amount0` determines the input token and is equal to the `amountCalculated`, and this amount is negative. The `amount1` determines the output token and calculates the difference between the `amountSpecified` and `amountSpecifiedRemaining`, and the resulting amount will be positive.
- If `zeroForOne` is false and `exactInput` is true, the `amount0` determines the output token and is equal to the `amountCalculated`, and this amount is positive. The `amount1` determines the input token and calculates the difference between the `amountSpecified` and `amountSpecifiedRemaining`, and the resulting amount will be negative.

```
unchecked {
    (int128 amount0, int128 amount1) = zeroForOne == exactInput
        ? ((params.amountSpecified
    - state.amountSpecifiedRemaining).toInt128(),
    state.amountCalculated.toInt128())
        : (
            (state.amountCalculated.toInt128()),
            (params.amountSpecified
    - state.amountSpecifiedRemaining).toInt128()
        );

    balanceDelta = toBalanceDelta(amount0, amount1);
}
}
```

Function: `updateDynamicLPFee(PoolKey key, uint24 newDynamicLPFee)`

The function allows to update the LP fee of the pool. The function can be called only by the `hook` address related to the pool. The fee can be updated only for the pool that supports the `DynamicLPFee`. The LP fee determines the fee that will be charged during the swap or modification-liquidity process.

The maximum LP fee is 100%.

Inputs

- `key`
  - **Control**: Full control.
  - **Constraints**: The pool for the provided `PoolKey` data should be initialized.
  - **Impact**: The `PoolKey` data that specified the initialized pool.
- `newDynamicLPFee`
  - **Control**: Full control.
  - **Constraints**: Max 100%.
  - **Impact**: The LP fee determines the fee that will be charged during the swap or modification-liquidity process.

Branches and code coverage

Intended branches

- The LP fee is updated properly for the appropriate pool.
  - ☑ Test coverage

Negative behavior

- The new LP fee is more than 100%.
  - ☑ Negative test
- The caller is not a hooks contract.
  - ☑ Negative test

Function call analysis

- `CLPool.setLPFee(this.pools[id], newDynamicLPFee)`
  - **What is controllable?** `newDynamicLPFee`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** The function will revert if the pool is not initialized.

6. Assessment Results

At the time of our assessment, the reviewed code was not deployed to the Ethereum Mainnet.

During our assessment on the scoped PancakeSwap Infinity Core contracts, we discovered four findings. No critical issues were found. One finding was of high impact, one was of medium impact, and two were of low impact.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.