certora

Security Assessment & Formal Verification Final Report

Parallel

Parallel Protocol

April 2025

Prepared for Mimo Capi

Table of content

Project Summary

Project Scope

Project Name	Repository (link)	Latest Commit Hash	Platform
Parallel Protocol	Parallel-Parallelizer Parallel-Tokens	c56bbe6 and 4473277 respectively	EVM

Project Overview

This document describes the specification and verification of **Parallel Protocol** using the Certora Prover. The work was undertaken from **Apr 3, 2025** to **May 1, 2025**.

The following contract list is included in our scope:

From Parallelizer
https://github.com/parallel-protocol/parallel-parallelizer/commit/c56bbe62d8ce8a421a6851d0e63bae0b58f32a44:
contracts/parallelizer/facets/Swapper.sol
contracts/parallelizer/facets/Redeemer.sol
contracts/parallelizer/facets/RewardHandler.sol
contracts/parallelizer/libraries/LibOracle.sol

From Parallel Tokens
https://github.com/parallel-protocol/parrallel-tokens/tree/4473277ff40e8cb21aeda727c3126fa648f1f1e4:
contracts/tokens/TokenP/TokenP.sol
contracts/tokens/BridgeableTokenP/BridgeableTokenP.sol
contracts/flashloan/FlashParallelToken.sol

Other files within both repositories on which the above files depended were also reviewed as needed.

The Certora Prover demonstrated that the implementation of the **Solidity** contracts above is correct with respect to the formal rules written by the Certora team. During the verification process, the Certora team discovered bugs in the Solidity contracts code, as listed on the following pages.

Protocol Overview

The Parallel Protocol is intended to generate an asset-backed stablecoin using multiple collateral assets. Stablecoins can be minted from or burned for any individual collateral. They can also be redeemed for a proportional share of all collateral types. The protocol relies on external oracles to determine the conversion ratio between collateral and stablecoins for mints and burns, and manages the risk of exposure to the different collateral types via economic incentives.

Mint and Burn Fees

The protocol implements a complex fee model in which the cost to mint or burn with a given collateral is dependent on the fraction of the total stablecoin supply backed by that asset. This allows the protocol to encourage a particular backing asset composition with economic incentives–e.g. when a particular asset is below some target range of fractional backing, the fee to mint with that asset will be very low and the fee to burn stablecoins for that asset will be very high (and vice versa when the asset is backing a fraction of stablecoins that is larger than the upper bound of its target range). Fees are linearly interpolated between different backing fractions so that they change smoothly. An important property of the fee algorithm is that breaking a particular operation into several smaller operations (within the same block) results in the same economic outcome. For example, burning 100 stablecoins for collateral token XYZ in one transaction will result in the same final balances (up to rounding error) as burning 20 stablecoins for XYZ in one transaction and then burning another 80 stablecoins in the following transaction within the same block.

Redemption Fees

Fees are only charged during redemptions if the protocol is undercollateralized (as calculated according to oracle prices for all collaterals). These fees help re-collateralize the protocol and discourage “bank run” scenarios against collaterals that are declining in value.

Reward Selling

In case collaterals deposited in the protocol accrue reward tokens that are not themselves collaterals for any reason, the protocol includes a mechanism to sell such rewards through the 1inch aggregator. One or more trusted actors is authorized to do such selling; this is a sensitive role since the protocol does relatively little validation of the results of the arbitrary call made to the 1inch router (it only checks that no collateral balances decreased).

Oracle Logic

The protocol relies heavily on external oracles and includes logic for reading from a wide variety of common oracles. When pricing collaterals for burning, the oracle price with the greatest deviation from target is used to value the collateral, discouraging users from redeeming strong collaterals and leaving the protocol saddled with weak ones.

Token Logic

The protocol includes a simple token contract for the stablecoin as well as a Layer Zero-compatible version for use on L2s and other non-mainnet chains that includes bridging functionality.

Findings Summary

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

Severity	Discovered	Confirmed	Fixed
Critical	-	-	-
High	-	-	-
Medium	1	1	1
Low	4	4	2
Informational	6	6	2
Total	11	11	5

Severity Matrix

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

Detailed Findings

ID	Title	Severity	Status
M-01	accrueInterestToFeeRecipient() may revert if called with multiple tokens	Medium	Fixed
L-01	A Small Quantity of Stables May Be Unable To Be Redeemed	Low	Acknowledged, won't fix
L-02	The protocol can become very slightly insolvent due to rounding error	Low	Acknowledged, won't fix
L-03	The hard cap check can be violated for collaterals due to rounding error	Low	Fixed
L-04	Unsafe casts to int256 in BridgeableTokenP.sol	Low	Partially fixed
I-01	TokenP.burnFrom() could use the _spendAllowance() function for gas efficiency and simplicity	Info	Fixed
I-02	Tokens can be drained by calling updateNormalizer()	Info	Acknowledged
I-03	Authorized reward sellers can call any 1inch function	Info	Acknowledged
I-04	Unchecked cast in LibHelper	Info	Acknowledged

I-05	Two different addresses for the 1inch router	Info	Fixed
I-06	Use transient to save gas	Info	Acknowledged

Medium Severity Issues

M-01 accrueInterestToFeeRecipient() may revert if called with multiple tokens		
Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: FlashParallelToken.sol	Status: Fixed	Violated Property: P-15. Integrity of accrueInterestToFeeRecipient accrueInterestDoesNotRevert

Description: The `accrueInterestToFeeRecipient()` function may revert if called with a multiple tokens array as a parameter. In each iteration of the loop, the result of `token.balanceOf(address(this))` of the current token is being added to the `balance` variable, which should be transferred to the fee recipient using `safeTransferFrom()`. However, if there are multiple tokens with non-zero balance, the second transfer will revert since the value of the `balance` variable will be greater than the contract's balance for that token.

```JavaScript
    //-------------------------------------------
    // Treasury Only Function
    //-------------------------------------------

    /// @notice Accrues interest to the fee recipient for a given list of tokens
    /// @param tokens List of addresses of tokens to accrue interest for
    /// @return balance Amount of interest accrued
    function accrueInterestToFeeRecipient(address[] calldata tokens) external returns
(uint256 balance) {
        for (uint256 i = 0; i < tokens.length; i++) {
            IERC20 token = IERC20(tokens[i]);
            balance += token.balanceOf(address(this));
            token.safeTransfer(flashLoanFeeRecipient, balance);
        }
    }
```

Recommendations: Transfer just `token.balanceOf(address(this))` in each iteration, not the sum of the balances of all the tokens.

Customer's response: Fixed in commit `c5c902e.`

Fix Review: The rule is verified when rerunning on commit `c5c902e` (required minor adjustments to the rule). See P-15. (`accrueInterestDoesNotRevert`).

Low Severity Issues

L-01 A Small Quantity of Stables May Be Unable To Be Redeemed		
Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: [Redeemer.sol]()	Status: Acknowledged	

Description: In `Redeemer._updateNormalizer` the `newNormalizerValue` and `newCollateralNormalizedStable` values should be rounded up, so that the contract overestimates the number of minted coins. This ensures that no one ends up with unredeemable stablecoins after the protocol has already redeemed all collateral.

Recommendations: Suggested diff:

```javascript
-    newNormalizerValue = _normalizer + (amount * BASE_27) / _normalizedStables;
+    newNormalizerValue = _normalizer + (amount * BASE_27 + _normalizedStables - 1) /
_normalizedStables;
     } else {
     newNormalizerValue = _normalizer - (amount * BASE_27) / _normalizedStables; }
...
     uint128 newCollateralNormalizedStable = (
-    (uint256(ts.collaterals[collateralListMem[i]].normalizedStables) * newNormalizerValue) /
BASE_27
+    (uint256(ts.collaterals[collateralListMem[i]].normalizedStables) * newNormalizerValue +
BASE_27 - 1) / BASE_27
     ).toUint128();
```

Customer's response: Acknowledged, won't fix.

L-02 The protocol can become very slightly insolvent due to rounding error		
Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: [LibGetters.sol]()	Status: Acknowledged	

Description: In `LibGetters.getCollateralRatio()` the last `mulDiv` uses `Math.Rounding.Ceil` and this overestimation of the collateral ratio can allow the protocol to technically be slightly insolvent. Rounding down instead of up would prevent this from occurring.

Recommendations: Replace `Math.Rounding.Ceil` by `Math.Rounding.Floor`:

```javascript
  collatRatio = (totalCollateralization.mulDiv(BASE_9, stablecoinsIssued,
Math.Rounding.Floor)).toUint64();
```

Customer's response: Acknowledged, won't fix.

L-03 The hard cap check can be violated for collaterals due to rounding error

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: Swapper.sol	Status: Fixed	Violated Property: P-11. Hard Caps hold

Description: The per-collateral hard cap check is computed as follows:

```javascript
 function _checkHardCaps(Collateral storage collatInfo, uint256 amount, uint256 normalizer)
internal view {
    if (amount + (collatInfo.normalizedStables * normalizer) / BASE_27 >
collatInfo.stablecoinCap) {
      revert InvalidSwap();
    }
  }
```

However, the actual update to the normalized stables is done with rounding-up after the above check is performed:

```javascript
       uint128 changeAmount = (amountOut.mulDiv(BASE_27, ts.normalizer,
Math.Rounding.Ceil)).toUint128();
        // The amount of stablecoins issued from a collateral are not stored as absolute
variables, but
        // as variables normalized by a `normalizer`
        collatInfo.normalizedStables = collatInfo.normalizedStables + uint216(changeAmount);
```

This can lead to the hard cap being violated after a swap by roughly `normalizer / 10^27`.

Recommendations: We recommend moving the check to after the token's `normalizedStables` is updated.

```
JavaScript
@@ -209,11 +209,11 @@ contract Swapper is ISwapper, AccessManagedModifiers {
       if (amountIn > 0 && amountOut > 0) {
       ParallelizerStorage storage ts = s.transmuterStorage();
       if (mint) {
-      _checkHardCaps(collatInfo, amountOut, ts.normalizer);
       uint128 changeAmount = (amountOut.mulDiv(BASE_27, ts.normalizer,
Math.Rounding.Ceil)).toUint128();
       // The amount of stablecoins issued from a collateral are not stored as absolute
variables, but
       // as variables normalized by a `normalizer`
       collatInfo.normalizedStables = collatInfo.normalizedStables + uint216(changeAmount);
+      _checkHardCaps(collatInfo, 0, ts.normalizer);
       ts.normalizedStables = ts.normalizedStables + changeAmount;
       if (permitData.length > 0) {
```

The `_checkHardCaps()` function can also be implemented more simply in this case (and without truncating division):

```
JavaScript
 function _checkHardCaps(Collateral storage collatInfo, uint256 normalizer) internal view {
    if (collatInfo.normalizedStables * normalizer > collatInfo.stablecoinCap * BASE_27) {
      revert InvalidSwap();
    }
  }
```

Customer's response: Fixed.

Fix Review: The rule is verified when rerunning on commit `4f4e449`.

L-04 Unsafe casts to int256 in BridgeableTokenP.sol

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: BridgeableTokenP.sol	Status: Partially fixed, not fixed in `globalCreditLimit.` Acknowledged.	Violated Properties: • P-05. Integrity of receive methods • P-07. Global and daily limits (credit and debit)

Description:

Unsafe cast in Lines 528–530 (in the function `_calculatePrincipalTokenAmountToCredit()`) causes wrong amounts to be used, surpassing the global credit limit. When `_amount` is large enough, the value of `int256(_amount)` becomes negative, causing `principalTokenAmountToCredit` in the lines below to become `_amount`. See this Report for an example, see also P-05 and P-07 globalCreditLimit. A similar issue exists in Line 440 (in function `_debit`), causing the violation in P-07 globalDebitLimit.

```JavaScript
528 principalTokenAmountToCredit = int256(_amount) + creditDebitBalance >
int256(globalCreditLimit)
529      ? uint256(int256(globalCreditLimit) - creditDebitBalance)
530      : _amount;
```

Recommendations: Use `SafeCast.toInt256` when casting from `uint256` to `int256`.

Customer's response: Fixed the unsafe cast in Line 528. Acknowledged the unsafe cast in Line 440.

Fix Review: Rerunning the failed rules on commit `c5c902e`:

- P-05. Integrity of receive methods is verified,
- In P-07. Global and daily limits (credit and debit) `globalDebitLimit` is verified while `globalCreditLimit` still fails.

Informational Issues

I-01. `TokenP.burnFrom()` could use the `_spendAllowance()` function for gas efficiency and simplicity

Description: In TokenP.sol, the `burnFrom()` function is implemented like this:

```javascript
function burnFrom(uint256 amount, address burner, address sender) external restricted {
     if (burner != sender) {
         uint256 currentAllowance = allowance(burner, sender);
         if (currentAllowance < amount) revert ErrorsLib.BurnAmountExceedsAllowance();
         _approve(burner, sender, currentAllowance - amount);
     }
     _burn(burner, amount);
 }
```

The manual allowance check does not account for the common semantic convention (adhered to the OZ implementation otherwise) that an allowance of `type(uint256).max` is not decremented to save gas.

Recommendation: This function could instead use the already-existing `_spendAllowance()` function for simplicity and gas efficiency (and also make the custom error `BurnAmountExceedsAllowance` unnecessary):

```javascript
function burnFrom(uint256 amount, address burner, address sender) external restricted {
     if (burner != sender) {
         _spendAllowance(burner, sender, amount);
     }
     _burn(burner, amount);
 }
```

Customer's response: Fixed.

I-02. Tokens can be drained by calling updateNormalizer()

Description: This is a permissioned function protected by either `canCall` or `isTrusted[]`. It must be strictly monitored which addresses have these permissions as it can corrupt the accounting of how many stable coins were minted and thus can be used to drain all funds using the `redeem()` function.

For example, someone with the `isTrusted` role can mint a large amount of stable coins (e.g. 20%), then call `updateNormalizer()` with -80% of the tokens to decrease the normalizer to 20% of its value. If this is followed by `redeem()` they will receive all tokens in the protocol.

Recommendation: Ensure a robust process for granting the ability to call updateNormalizer() is in place, such as not allowing any single individual or entity to unilaterally grant this permission, or using a timelock to allow time to react to malicious permission grants.

Customer's response: Acknowledged.

I-03. Authorized reward sellers can call any 1inch function

Description: The RewardHandler allows reward sellers to call arbitrary 1inch functions. This allows such actions as selling to themselves in a private liquidity pool that they control, or routing through public pools where the trading pair includes a token they can mint at will, essentially enabling them to steal all rewards.

Recommendation: Use caution when deciding how seller permission is granted (e.g. avoid allowing single individuals to control this capability). Alternatively, more specific integrations with dexes or aggregators could limit the risk more robustly (e.g. enforcing rewards are always sold for a particular desired token through a selection of trusted pools).

Customer's response: Acknowledged, we intend to switch to Odos.

I-04. Unchecked cast in LibHelper

Description: In `LibHelpers.piecewiseLinear` there is an unchecked cast from `uint64` to `int64`, which may cause it to return the wrong value.

With fee magnitudes generally limited by validation logic, there cannot be an overflow here, but there may be other places where this function is used.

Recommendation: Cast to `int256` and do all computations with full precision. Then do a single checked `int64` cast at the end (or let the function return an `int256`).

Customer's response: Acknowledged, fees will be managed by DAO.

I-05. Two different addresses for the 1inch router

Description: The file Constants.sol defines `ONE_INCH_ROUTER` which is the v5 router and `ONEINCH_ROUTER`, which is the v6 router.

Recommendation: Remove one of the constants, or include the version number in the constant names if both are needed.

Customer's response: Fixed and replaced by Odos.

I-06. Use transient to save gas

Description: The reentrancy lock and the `consumingSchedule` entry could be transient to save gas.

Recommendation: Consider using transient storage for the noted fields.

Customer's response: Acknowledged.

Formal Verification

Verification Methodology

We performed verification of the **Parallel** protocol using the Certora verification tool which is based on Satisfiability Modulo Theories (SMT). In short, the Certora verification tool works by compiling formal specifications written in the [Certora Verification Language (CVL)]() and **Parallel**'s implementation source code written in Solidity.
More information about Certora's tooling can be found in the [Certora Technology Whitepaper.]()

If a property is verified with this methodology it means the specification in CVL holds for all possible inputs. However specifications must introduce assumptions to rule out situations which are impossible in realistic scenarios (e.g. to specify the valid range for an input parameter). Additionally, SMT-based verification is notoriously computationally difficult. As a result, we introduce overapproximations (replacing real computations with broader ranges of values) and underapproximations (replacing real computations with fewer values) to make verification feasible.

Rules: A rule is a verification task possibly containing assumptions, calls to the relevant functionality that is symbolically executed and assertions that are verified on any resulting states from the computation.

Inductive Invariants: Inductive invariants are proved by induction on the structure of a smart contract. We use constructors as a base case, and consider all other (relevant) externally callable functions that can change the storage as step cases.
Specifically, to prove the base case, we show that a property holds in any resulting state after a symbolic call to the respective constructor. For proving step cases, we generally assume a state where the invariant holds (induction hypothesis), symbolically execute the functionality under investigation, and prove that after this computation any resulting state satisfies the invariant.

Verification Notations

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

General Assumptions and Simplifications

For the verification of the `parallel-parallelizer` repository, we use a **CVL implementation of the ERC-20 token standard** rather than the actual deployed contracts. This abstraction allows us to focus on the interactions relevant to the protocol under verification while avoiding external complexity or variability in third-party token implementations. Our CVL model captures the standard ERC-20 interface and expected behaviors, ensuring that properties related to token transfers, balances, and approvals hold under the ERC-20 specification.

Formal Verification Properties

RewardHandler

RewardHandler General Assumptions

- The function `AccessManager.canCall` will only allow calls to `sellRewards` from the governor.
- The access related function `isConsumingScheduledOp` will not have side-effects.

RewardHandler Properties

P-01. sellRewards can only be called by governor or trusted seller	
Status: Verified	

Rule Name	Status	Description	Link to rule report
sellRewardsNeedsPermissions	Verified	*This rule verifies that the caller was either the governor contract or a trusted seller before the call.*	Report

P-02. sellRewards does not decrease balance of collateral tokens	
Status: Verified	

Rule Name	Status	Description	Link to rule report
sellRewardsDoesNotDecreaseBalance	Verified	*This rule verifies that the balance of all collateral tokens does not decrease when calling sellRewards.*	Report

P-03. sellRewards increases balance of at least one collateral token

Status: Verified	

Rule Name	Status	Description	Link to rule report
sellRewardsIncreasesOneBalance	Verified	*This rule verifies that the balance of at least one of the collateral tokens increases.*	[Report]()

BridgeableTokenP

BridgeableTokenP General Assumptions

- The following functions are assumed to have no side-effects:
  - `IAccessManager.consumeScheduledOp`
  - `IAccessManaged.setAuthority`
- The fee returned by `ISendLib.send` and `ISendLib.quote` depends only on `packet.srcEid`, `packet.sender`, `packet.dstEid`, `packet.receiver`, `payInLzToken`, and `block.timestamp`.

Module Properties

P-04. Integrity of send	
Status: Verified	

Rule Name	Status	Description	Link to rule report
sendIntegrityNativeBalances	Verified	*The `send` method transfers native balances only between three specific addresses: `msg.sender`, `sendLib` and `refundAddress`.*	[Report](#)
sendIntegrityTokenBalances	Verified	*The `send` method correctly transfers tokens:* *1. `amountSentLD` is burnt from `msg.sender` in either `BridgeableTokenP` or `TokenP`* *2. `fee.lzTokenFee` is sent from `msg.sender` to `sendLib`* *3. Fee is transferred to `sendLib` and excess balance to `refundAddress`*	
sendThirdPartyProtectionNativeBalance	Verified	*Only sender, `sendLib` and refund addresses' native balances can be affected by `send`.*	[Report](#)
sendThirdPartyProtectionTokenBalances		*Only sender, `sendLib` and refund addresses' token balances can be affected by `send`.*	

P-05. Integrity of receive methods (`lzReceive` and `lzReceiveSimulate`)

Status: Verified after fix	

Rule Name	Status	Description	Link to rule report
receiveIntegrity	Verified after fix	*Integrity of `lzReceive` and `lzReceiveSimulate`:* • *The only balances affected are those of `to` and `feesRecipient`.* • *Tokens are minted.* • *Correct amounts are transferred.* *See L-04.*	Report

P-06. End point balance is zero.

Status: Verified	

Rule Name	Status	Description	Link to rule report
endpointLzTokenBalanceZero	Verified	*Excluding donations, the `lzToken` balance of `endpoint` is always zero.*	Report

P-07. Global and daily limits (credit and debit)

Status: Partially violated

Rule Name	Status	Description	Link to rule report
maxGlobalCreditLimit	Verified	*`globalCreditLimit` is at most `MAX_GLOBAL_LIMIT`*	[Report]
maxGlobalDebitLimit	Verified	*`globalDebitLimit` is between `-MAX_GLOBAL_LIMIT` and 0*	
dailyDebitAmountLimits	Verified	*Daily debit limit holds*	
dailyCreditAmountLimits	Verified	*Daily credit limit holds*	
globalDebitLimit	Verified after fix	*Global debit limit holds. See L-04.*	
globalCreditLimit	Violated	*Global credit limit holds. See L-04.* *Customer response: Acknowledged, won't fix.*	

Swapper

Module General Assumptions

- Calls to `LibManager.invest` and `IKeyringGuard.isAuthorized` are considered to have no side-effects.
- In P-08. Swap integrity and P-09. Total normalized stables follows swaps the `quoteFees` function's return value was considered to be arbitrary, since its actual value was not relevant to these properties (an over-approximation of the possible states).
- Mathematical functions `mulDiv`, `sqrt`, `convertDecimalTo` and also `LibHelpers.findLowerBound` were summarized to equivalent functions in CVL for better tractability.
- We assume that the values read from the oracle are constant.
- We assume that `tokenIn` is different from `tokenOut`.
- In P-12 we assume that all `yFeeMint` and `yFeeBurn` satisfy conditions set in `LibSetters.checkFees`.

Module Properties

P-08. Swap integrity	
Status: Verified	

Rule Name	Status	Description	Link to rule report
swapExactInputIntegrity	Verified	*Integrity of* *1. `swapExactInput`* *2. `swapExactInputWithPermit`*	*Report*
swapExactOutputIntegrity	Verified	*Integrity of* *3. `swapExactOutput`* *4. `swapExactOutputWithPermit`*	
thirdPartyProtection	Verified	*Third party balances are not affected by swaps*	

P-09. Total normalized stables follows swaps

Status: Verified	

Rule Name	Status	Description	Link to rule report
swappingChangesTotalNormalized	Verified	*Swapping changes the value of `normalizedStables`, provided the swapped amount is large enough*	[Report]()
normedStablesProportionalToTotalSupply	Verified	*The value of `normalizedStables`* ***weakly*** *increases or decreases with the total supply of `TokenP`.*	
normedStablesIsUpperBoundForTotalSupply	Verified	*The denormalizing the normalized stable amount gives an upper bound for the total supply*	

P-10. Quote functions preserve zero (zero input implies zero output)

Status: Verified	

Rule Name	Status	Description	Link to rule report
zeroInZeroOutQuoteIn	Verified	*Zero input yields zero output in `quoteIn`*	[Report]()
zeroInZeroOutQuoteOut	Verified	*Zero input yields zero output in `quoteOut`*	[Report]()

P-11. Hard Caps hold	
Status: Verified after fix	

Rule Name	Status	Description	Link to rule report
hardCapsHold	Verified after fix	*The caps, as set in `_checkHardCaps`, always hold. See [L-03](L-03).*	Report

P-12. Positive fees make swapping less lucrative compared to feeless conversion, the opposite for negative fees, *assuming all oracle values are 1*	
Status:Verified	

Rule Name	Status	Description	Link to rule report
feeCostBurnsOutNegativeTest	Verified	*In `quoteOut`, when burning, negative fees imply swapping yields more than conversion when oracle values are one.*	Report
feeCostBurnsOutPositiveTest	Verified	*In `quoteOut`, when burning, positive fees imply swapping yields less than conversion when oracle values are one.*	
feeCostMintsOutNegativeTest	Verified	*In `quoteOut`, when minting, negative fees imply swapping yields more than conversion when oracle values are one.*	Report
feeCostMintsOutPositiveTest	Verified	*In `quoteOut`, when minting, positive fees imply swapping yields less than conversion when oracle values are one.*	

feeCostMintsIn NegativeTest	Verified	*In `quoteIn`, when minting negative fees imply swapping yields more than conversion when oracle values are one.*	[Report]
feeCostMintsIn PositiveTest	Verified	*In `quoteIn`, when minting, positive fees imply swapping yields less than conversion when oracle values are one.*	
feeCostBurnsIn PositiveTest	Verified	*In `quoteIn`, when burning, positive fees imply swapping yields less than conversion when oracle values are one.*	[Report]
feeCostBurnsIn NegativeTest	Verified	*In `quoteIn`, when burning, negative fees imply swapping yields more than conversion when oracle values are one.*	[Report]

Redeemer

Module General Assumptions

- We use the `MockManager` from the test contracts for managed tokens..

Module Properties

P-13. MinAmount and Deadline is honored	
Status: Verified	

Rule Name	Status	Description	Link to rule report
redeemMinAmountAndDeadline	Verified	*This rule verifies that the amount returned by redeem is at least minAmount and that the deadline is after block timestamp.*	[Report]

P-14. Tokens are sent by redeem to receiver	
Status: Verified	

Rule Name	Status	Description	Link to rule report
redeemSendsTokens	Verified	*This rule verifies that the token balance of receiver for token[i] is increased by amount[i], where i is an arbitrary index into the returned arrays.* *The rule requires that the receiver is neither the redeemer contract nor the MockManager. It also requires that the token occurs only once in the returned array.*	[Report]

FlashParallelToken

Module General Assumptions

- Using two `TokenP` contracts as the tokens.

Module Properties

P-15. Integrity of accrueInterestToFeeRecipient	
Status: Verified	

Rule Name	Status	Description	Link to rule report
accrueInterestIntegrity	Verified	*\`accrueInterestToFeeRecipient\` integrity.*	*Report*
accrueInterestDoesNotRevert	Verified after fix	*\`accrueInterestToFeeRecipient\` does not revert except for overflows. See M-01.*	*Report*

Disclaimer

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

About Certora

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

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