Formal Verification Engagement Summary: Cork Protocol

Introduction

In October 2024, Runtime Verification engaged with Cork Protocol to perform formal verification on their DeFi protocol. At Runtime Verification, we specialize in building rigorous, mathematically grounded proofs to ensure that smart contracts behave as intended under all possible inputs and scenarios. Our tool, **Kontrol**, is designed to integrate seamlessly with Solidity-based projects, enabling developers to write property-based tests in Solidity and leverage symbolic execution to verify them. The following document summarizes the key aspects of our formal verification engagement with Cork Protocol, focusing on the tests, invariants, and findings from this collaboration.

Reproducing the Proofs

To reproduce the results of this verification locally, follow the steps below:

Install Kontrol:

```
bash <(curl https://kframework.org/install)
kup install kontrol
```

Clone the Test Repository:

```
git clone git@github.com:runtimeverification/Depeg-swap.git
cd Depeg-swap
git checkout kontrol-tmp-changes
git submodule update --init --recursive
```

Run the Tests:

```
export FOUNDRY_PROFILE=kontrol-properties
kontrol build
kontrol prove
```

View the Results:
`kontrol list` (optionally, it is possible to pass the flag `--xml-test-report` to the `kontrol prove` command and obtain a report in XML format)

Additionally, you can view the results in the KaaS dashboard: https://kaas.runtimeverification.com/job/9df2b13d-bbbf-4b41-910e-448e0484d680/xml-report.

Scope

The goal of this engagement was to verify the correctness of several components of the Cork Protocol, specifically, that selected functions in Peg Stability Module, Liquidity Vault, and Flash Swap Router, as well as library functions they rely on, preserve desired properties.

The files under the scope of the engagement were:

- `contracts/core/ModuleCore.sol`
- `contracts/core/Psm.sol`
- `contracts/core/Vault.sol`
- `contracts/core/CorkConfig.sol`
- `contracts/core/ModuleState.sol`
- `contracts/core/flash-swaps/FlashSwapRouter.sol`
- `contracts/core/assets/Asset.sol`

We used the branch `tmp-changes` in https://github.com/Cork-Technology/Depeg-swap to sync the changes and fixes implemented by the Cork Protocol team and `kontrol-tmp-changes` branch in https://github.com/runtimeverification/Depeg-swap to push our tests and improvements to the code. The latest commit that we have used to run the proofs where they were passing was e899676bffe939ff7c633bc9bf13d1e14c0bd270. On later commits, such as 7897c382e7793ecb658e8c950ed38c1c643a941a — the most recent one at the time of writing — the proof for Liquidity Vault's redemption functionality (`LiquidityVaultTest.test_redeemLV_correctness`) is failing because the function of the logic has changed and does not adhere to the previously specified invariant.

Invariants Proven

Below are the definitions of the contracts and the invariants we sought to formally verify in this engagement:

PsmCore

The `PsmCore` contract, inherited by `ModuleCore`, implements the base functionality of the Cork Protocol, allowing the user to deposit Redemption Assets (RA) and receive Cover Tokens (CT) and Depeg Swaps (DS).

PSMCore Invariants:

- **Invariant 1:** State Reinitialization Not Allowed
  - Status: Verified
  - Summary: Reinitializing `PsmCore`, or `ModuleCore`, state for an existing Pair `id` is not allowed.

- **Invariant 2:** Each CT & DS are Backed by PA & RA
  - Status: Verified
  - Summary: Each CT and DS minted by the protocol should be backed by PA, RA, or a combination of both, to ensure that the holders of DS will be able to redeem the RA through the appropriate `redeem` function.
  - Observation: We have verified that two functions of `PsmCore`, or `ModuleCore` importing it, preserve this invariant; specifically, `depositPsm` and `redeemWithDs`.

- **Invariant 3:** CT Balance And Accounting Consistency
  - Status: Verified
  - Summary: The value of the `ModuleCore`'s Vault's CT balance accounting variable (`states[id].vault.balances`) should be equivalent to the CT token balance of `ModuleCore`, ensuring the consistency between internal accounting in the protocol and token balances.

    - Observation: We have verified that two functions of `PsmCore`, or `ModuleCore` importing it, preserve this invariant; specifically, `depositPsm` and `redeemWithDs`.

- **Invariant 4:** Liquidity Vault's DS Balance and Reserves Accounting Consistency
    - Status: Verified
    - Summary: The value of the `ModuleCore`'s Vault's DS reserves — transferred to the `FlashSwapRouter` contract — accounting variable (`reserves[id].ds[dsId].lvReserve`) should be equivalent to the DS token balance of `FlashSwapRouter`, ensuring the consistency between internal accounting in the protocol and token balances.
    - Observation: We have verified that two functions of `PsmCore`, or `ModuleCore` importing it, preserve this invariant; specifically, `depositPsm` and `redeemWithDs`.

- **Invariant 6:** Redemption with DS Invariant Preservation
    - Status: Verified
    - Summary: The `redeemWithDs` function, when called with valid input parameters, should preserve the above mentioned invariants.
    - Observation: Invariant 2 only holds if we can assume that the amounts of RA and CT that should be sent to the AMM — which depend on the values returned by the functions `__getAmmCtPriceRatio` and `calculateProvideLiquidityAmountBasedOnCtPrice` — are different than 0. Additionally, for `redeemWithDs` to maintain the invariant, we assume that the exchange rate between DS/CT and RA is 1.

VaultCore

The `VaultCore` contract, inherited by `ModuleCore`, implements a Liquidity Vault that enables yield generation on Redemption Assets (RA) by providing it as liquidity to the AMM pool trading RA and CT, minting and selling DS tokens through a FlashSwapRouter, and collecting fees. Upon a deposit, the users are provided with LV tokens, which can then be used to redeem the deposit with some accrued yield.

VaultCore Invariants:

- **Invariant 1:** Deposit Not Allowed After DS Expiry
  - Status: Verified
  - Summary: This invariant ensures that the deposit into the LV reverts in the DS corresponding to the supplied `Id` has expired.

- **Invariant 2:** Each CT & DS are Backed by PA & RA
  - Status: Verified
  - Summary: Each CT and DS minted by the protocol should be backed by PA, RA, or a combination of both, to ensure that the holders of DS will be able to redeem the RA through the appropriate `redeem` function.
  - Observation: We have verified that two functions of `VaultCoreCore`, or `ModuleCore` importing it, preserve this invariant; specifically, `depositLv` and `redeemEarlyLv`.

    This invariant only holds if we can assume that the amounts of RA and CT that should be sent to the AMM — which depend on the values returned by the functions `__getAmmCtPriceRatio` and `calculateProvideLiquidityAmountBasedOnCtPrice` — are different than 0.

- **Invariant 3:** CT Balance And Accounting Consistency
  - Status: Verified
  - Summary: The value of the `ModuleCore`'s Vault's CT balance accounting variable (`states[id].vault.balances`) should be equivalent to the CT token balance of `ModuleCore`, ensuring the consistency between internal accounting in the protocol and token balances.
  - Observation: We have verified that two functions of `VaultCore`, or `ModuleCore` importing it, preserve this invariant; specifically, `depositLv` and `redeemEarlyLv`.

- **Invariant 4:** Liquidity Vault's DS Balance and Reserves Accounting Consistency
  - Status: Verified
  - Summary: The value of the `ModuleCore`'s Vault's DS reserves — transferred to the `FlashSwapRouter` contract — accounting variable (`reserves[id].ds[dsId].lvReserve`) should be equivalent to the DS token balance of `FlashSwapRouter`, ensuring the consistency between internal accounting in the protocol and token balances.
  - Observation: We have verified that two functions of `VaultCore`, or `ModuleCore` importing it, preserve this invariant; specifically, `depositLv` and `redeemEarlyLv`.

FlashSwapRouter

The `FlashSwapRouter` contract implements a flash swap mechanism that allows buying Depeg Swaps (DS) for RA and selling RA for DS through the AMM. `FlashSwapRouter` additionally enables selling of the DS that accumulates in the Liquidity Vault's reserves and facilitates the selling of DS during the rollover, if applicable. In `FlashSwapRouter` proofs, we focused on proving correctness of some of the state updates performed by `swapRaForDs` and `swapDsForRa`, as described in the Proof Overview section.

The `swapRaForDs` function uses several different means to source the DS tokens for the swap, specifically, part of the swap can be covered by the rollover sale, by selling DS tokens for RA from reserves, and through the flashswap. Within the scope of this engagement, we focused on one possible execution flow, when the swap is fulfilled entirely during the rollover sale. Correctness of the execution involving a flash swap when selling RA for DS, or involving selling from reserves was not verified, however, the latter involves the execution of the function swapping DS for RA which has been analyzed separately.

Due to the time constraints and scope of the engagement, in the `FlashSwapRouter` as well as `ModuleCore` proofs we have abstracted the functionality of the AMM, or `CorkHook`, contract, assuming its functions return symbolic outputs adhering to the specific assumptions. To facilitate reasoning, we have made additional simplifying assumptions about the functionality related to interactions with the AMM, as also described in Proof Overview.

Additionally, we assume that the RA, PA, CT, and DS tokens have 18 decimals.

Proof Overview

The invariant proofs of the `ModuleCore` can be found in the `LiquidityVaultTest` and `PSMTest` contracts, whereas the proofs for the `FlashSwapRouter` can be found in the `RouterTest` contract. Additionally, proofs of the `MathHelper` library functions can be found in the `MathHelperTest` contract.

All four of these contracts inherit from `KontrolTest` contract, which inherits the Foundry `Tests` contract and also the `KontrolCheats` contract. `KontrolCheats` contains the `kevm` address as well as the library of Kontrol cheatcodes that are necessary to run the tests.

`LiquidityVaultTest, PSMTest, and RouterTest` share the same `setUp` method inherited from the `ModuleCoreTest` contract that deploys all necessary contracts and makes their respective storage symbolic through the use of the Kontrol cheatcode `kevm.symbolicStorage(address(this))`. The use of this cheatcode in `setUp` makes the storage of respective contracts symbolic, or arbitrary, which allows proving that invariants hold for any state of the contract and its storage, under the given assumptions.

We designed the following tests to prove the specified invariants:

ModuleCore

☑ Test 1: Property-Based Test for State Reinitialization

- **Objective:** This test verifies that once a `Pair` with a specific `RA, PA, and expiryInterval` is created and initialized, it cannot be reinitialized.
- **Current Status:** Verified and passed.

- **Details:** Before calling the `initializeModuleCore` function in `moduleCore`, we assume that a `Pair` has already been initialized with the provided parameters. After the call, we check that it reverts with the `ICommon.AlreadyInitialized` error.

- **Assumptions before function call:**
  - A `Pair` with the provided parameters is initialized in `moduleCore`
- **Assertions:**
  - **After function call:**
    - `initializeModuleCore` reverts `with AlreadyInitialized`

☑ Test 2: Property-Based Test for PSM Deposit

- **Objective:** This test verifies that `psmDeposit` accepts the RA provided by the user, splits it into CT & DS and transfers these tokens back to the user, preserving the invariants between CT & DS and RA & PA, as well as Vault's CT and DS accounting.

- **Current Status:** Verified and passed.

- **Details:** We assume that the desired invariants are holding and the user has a sufficient balance of RA. Then, invoke the `depositPsm` function with the symbolic positive amount input. Finally, we assert that the invariants are still holding and that the state of moduleCore is updated as expected.

- **Assumptions before function call:**
  - Amount of RA to be deposited is positive and is sufficiently big to prove the correctness properties of interest
  - Deposits are not paused
  - DS has not expired
  - The depositor has a sufficient (symbolic) balance of RA to transfer `amount`
  - The depositor has granted a sufficient allowance to moduleCore to transfer `amount`
  - No overflow occurs during the tokens transfers
  - `MathHelperTest.testCalculatePercentageFee` returns a symbolic value `ra` that is assumed to be less than or equal to the RA locked in `ModuleCore` (`moduleCore.getPsmBalances(id).ra.locked`); we separately prove that the resulting `ra` is equal to (`amount * exchangeRates / UNIT`), as intended
  - `MathHelper.calculatePercentageFee` returns a symbolic `fee` that is assumed to be not greater than (`ra * 5 ether) / (100 * UNIT)`), since 5 * 10^18, or 5 ether, is the upper bound of `psmBaseRedemptionFeePercentage`; it is separately prove that the resulting `fee` is not greater than the input value `amount` (in this case, `ra`) that the function is calculating the percentage of
  - `MathHelper.calculateWithTolerance` returns a concrete value of 0 since the `CorkHook.addLiquidity` function in these proofs relies on a mock implementation and does not utilize the return value of `calculateWithTolerance`
  - `RouterState.getCurrentPriceRatio` returns a symbolic value `ctRatio` that is greater than 0
  - `MathHelper.calculateProvideLiquidityAmountBasedOnCtPrice` returns symbolic values (`raAmount, ctAmount`), where `ctAmount` is assumed to be greater than 0 and less than `fee` returned from `MathHelper.calculatePercentageFee`, and `raAmount` is calculated as `fee - ctAmount`
  - `CorkHook.addLiquidity` returns symbolic values `amountRa`, `amountCt`, and `mintedLp`. The `amountRa` and `amountCt` are correspondingly the amounts of RA and CT that are going to be transferred to the AMM, so we assumed that they are less that their corresponding `raAmount` and `ctAmount` passed as parameters to the function. This function then transfers `amountCt` CT tokens and `amountRa` RA tokens from

`ModuleCore` to the AMM, and mints `mintedLp` `LP` tokens to `ModuleCore`

- **Assertions:**
  - **After function call:**
    - Invariants 2, 3 and 4 are holding
    - `moduleCore` RA balance increased by `amount`
    - Depositor's RA balance decreased by `amount`
    - Depositor's DS balance increased by `amount`
    - Depositor's CT balance increased by `amount`
    - DS `totalSupply` increased by amount
    - `CT totalSupply` increased by amount
    - `moduleCore` RA locked (`moduleCore.getPsmBalances(id).ra.locked`) increased by amount.

☑ **Test 3: Property-Based Test for Redeeming RA with DS and PA**

- **Objective:** This test verifies that `redeemRaWithDs` accepts DS and PA, and transfers the RA to the user, preserving the invariant between CT & DS and RA & PA, as well as Vault's CT and DS accounting.

- **Current Status:** Verified and passed.

- **Details:** We assume that desired invariants are holding and the user has a sufficient balance of DS and PA. Then, invoke the `redeemRaWithDs` function. Finally, we assert that the invariants are still holding and that the state of `ModuleCore` is updated as expected. We are abstracting the return values of the following `MathHelper` library functions: `calculateEqualSwapAmount`, `calculatePercentageFee`, `calculateWithTolerance`, `calculateProvideLiquidityAmountBasedOnCtPrice`, and of the `RouterState.getCurrentPriceRatio` function, and we make the assumptions about the return values that are outlined below.

- **Assumptions before function call:**
  - `amount` of DS and PA to be transferred to the Vault is positive and sufficiently big to prove correctness
  - Withdrawal in `ModuleCore` are not paused
  - DS has not expired
  - Exchange rate between DS/CT and RA is 1
  - The call to `redeemRaWithDs` is not reentrant
  - `Redeemer` has a sufficient (symbolic) balance of DS and PA to transfer `amount`

        - `Redeemer` has granted a sufficient allowance to `ModuleCore` to transfer `amount`
        - No overflow occurs during the tokens transfers
        - We assume that the fee returned by the mocked function `calculatePercentageFee` is less than (amount * 5 ether) / (100 * UNIT)), since this was proved in the corresponding test for `calculatePercentageFee`. We also assume that the values returned by the mocked function `calculateRedeemLv` are consistent with the internal accounting and balances of the `moduleCore`
    - **Assertions:**
        - **After function call:**
            - Invariants 2, 3 and 4 are holding.

VaultCore

☑ **Test 1: Property-Based Test for Deposit Reverting After DS Expiry**

- **Objective:** This test verifies that the `depositLv` function reverts if the DS token has reached an expiry.
- **Current Status:** Verified and passed.
- **Details:** We assume that deposits have not been paused and that the DS token corresponding to the `Id` has expired. Then, we check that a call to the `depositLv` with a positive `amount` reverts with the expected error.
- **Assumptions:**
    - `amount` deposited by the user is positive
    - Deposits in the Vault have not been paused
- **Assertions:**
    - **After function call:**
        - The call to `depositLv` reverts with the `Guard.Expired` custom error.

☑ **Test 2: Property-Based Test for Vault Deposit**

- **Objective:** This test verifies that `depositLv` accepts the RA provided by the user and updates the state, including token balances, as expected.

- **Current Status:** Verified and passed.

- **Details:** Before invoking the `depositLv` function, we assume that the invariants are holding. Then, we call `depositLv` with a positive `amount`. Finally, we assert that the invariants are still holding and that the state of `ModuleCore` is updated as expected. To facilitate reasoning, we abstract the return values of the following functions of the `MathHelper` library: `calculatePercentageFee`, `calculateProvideLiquidityAmountBasedOnCtPrice`, `calculateDepositLv`, as well as the return value of the `RouterState.getCurrentPriceRatio` function. The assumptions we are making about the values returned by these functions are outlined below.

- Assumptions before the call:
  - `amount` of RA deposited to the Vault by the user is positive and sufficiently big to prove correctness
  - Vault deposits are not paused in `ModuleCore`
  - `DS` token did not expire
  - No overflows occur during token transfers
  - We assume that the fee returned by the mocked function `calculatePercentageFee` is less than (amount * 5 ether) / (100 * UNIT)), because 5 ether is the maximum fee and this was proved in the corresponding test for `calculatePercentageFee`. We also assume that the values returned by the mocked function `calculateRedeemLv` are consistent with the internal accounting and balances of the `moduleCore`

- **Assertions:**
  - **After function call:**
    - `Depositor`'s LV token balance increased by the LV tokens minted
    - Total supply of the LV token increased by the LV tokens minted
    - `ModuleCore`'s LP token balance increased by the amount of LP tokens minted
    - Total supply of the LP token balance increased by the amount of LP tokens minted
    - `Depositor`'s RA token balance decreased by the `amount` transferred to `ModuleCore`, minus some RA dust, if refunded
    - `AMM`'s RA token balance increased by the RA amount provided to it as liquidity, according to the strategy as calculated via `calculatePercentageFee` and `calculateProvideLiquidityAmountBasedOnCtPrice`
    - `AMM`'s CT token balance decreased by the `amount` provided to it as liquidity, according to the strategy

- `Depositor`'s RA token balance increased by the CT token amount corresponding to the dust incurred during the AMM liquidity provision, if any
- `ModuleCore`'s RA token balance and `ModuleCore` RA locked (`moduleCore.getPsmBalances(id).ra.locked`) should increase by the `splitted` amount and `ctAmount`, with `splitted` being equal to the `vault.ctHeldPercetage` of the `amount` and `ctAmount` being a part of the RA amount left, i.e, `amount - splitted`, that is again going to be used to issue `DS` and `CT`.
- `ModuleCore`'s CT token balance increased by the amount of CT tokens minted and added to the reserve, according to the strategy
- Total supply of the CT token increased by the total amount of DS tokens minted
- Total supply of the DS token should be increased by the total amount of DS tokens minted
- `FlashSwapRouter`'s balance of DS tokens should be increased by the amount of DS tokens transferred there, according to the strategy

☑ **Test 3: Property-Based Test for Vault Early Redemption**

- **Objective:** This test verifies that `redeemEarlyLv` accepts the LV provided by the user and updates the state, including token balances, as expected.

- **Current Status:** Verified and passed on commit e899676bffe939ff7c633bc9bf13d1e14c0bd270. Invariant 2 is failing on commit 7897c382e7793ecb658e8c950ed38c1c643a941a due to the change in the function implementation logic that now returns RA *and PA* to the user, breaking the invariant as it was specified initially.

- **Details:** Before invoking the `redeemEarlyLv` function, we assume that the invariants are holding. Then, we call `redeemEarlyLv` with a positive `amount`. Finally, we assert that the invariants are still holding and that the state of `ModuleCore` is updated as expected. To facilitate reasoning, we abstract the return values of the following functions of the `MathHelper` library: `calculatePercentageFee`, `calculateRedeemLv`, as well as the return value of the `Withdrawal.add` function. The assumptions we are making about the values returned by these functions are outlined below.

- **Assumptions before function call:**
  - `amount` of LV provided to the Vault by the redeemer is positive and sufficiently big to prove correctness

- The call to `redeemEarlyLv` is not reentrant
- Vault withdrawals are not paused in `ModuleCore`
- `DS` token did not expire
- No overflows occur during token transfers
- `MathHelper.calculateRedeemLv` returns symbolic values (`ctReceived, dsReceived, lpLiquidated, paReceived, idleRaReceived`), which satisfy the following constraints:
  - `ctReceived` is less than or equal to Vault's CT balance
  - `dsReceived` is less than or equal to Vault's DS reserve in `FlashSwapRouter`
  - `lpLiquidated` is less than or equal to `ModuleCore`'s LP balance
  - `paReceived` is less than or equal to `ModuleCore`'s PA balance
  - `idleRaReceived` is less than or equal to `ModuleCore`'s RA balance
- `Withdrawal.add` returns 0

- **Assertions:**
  - **After function call:**
    - Invariants 2, 3, and 4 are holding
    - Total supply of LV decreased by `amount` burned
    - Redeemer's balance of LV decreased by `amount` burned
    - `ModuleCore`'s balance of LP decreased by the amount of LP tokens liquidated
    - Total supply of LP decreased by the amount of LP tokens liquidated
    - `ModuleCore`'s RA balance did not change
    - Redeemer's RA balance increased by the amount of tokens redeemed from the AMM
    - AMM's RA balance decreased by the amount of RA tokens redeemed from the AMM and transferred to the user
    - RA locked in `ModuleCore` didn't change
    - Total supply of DS didn't change
    - Total supply of CT didn't change
    - `FlashSwapRouter`'s DS balance decreased by the amount of DS tokens withdrawn from the Vault's reserves in the Router
    - `ModuleCore`'s CT balance decreased by the amount of CT tokens withdrawn from the Vault
    - AMM's CT balance decreased by the amount of CT tokens redeemed from the AMM and transferred to the redeemer
    - Redeemer's CT balance increased by the sum of CT tokens transferred from AMM and Vault

FlashSwapRouter

☑ Test 1: Property-Based Test for Swapping RA for DS From Rollover

- **Objective:** This test verifies that the `swapRaForDs` function updates the state including token balances as expected if the whole order is fulfilled during a rollover sale from PSM and LV reserves.

- **Current Status:** Verified and passed.

- **Details:** Before invoking the `swapRaForDs` function, we make the assumptions about input values and protocol state as described below. Then, we call `swapRaForDs` with symbolic positive `amount` and `amountOutMin` as inputs.

  We are abstracting the return values and execution of functions `SwapperMathLibrary.calculateRolloverSale`, `ModuleCore.lvAcceptRolloverProfit`, `SwapperMathLibrary.calcHIYAaccumulated`, and `SwapperMathLibrary.calcVHIYAaccumulated`, and make assumptions about their return values as outlined below.

- **Assumptions:**
  - `amount` provided by the user is positive
  - `hiya` is greater than zero and less than 1e18
  - `dsId` is not 1, has a concrete value of 2
  - `block.number` is within the rollover sale timeframe
  - `Seller` has a sufficient balance of RA to transfer `amount` to the Router
  - `Seller` granted a sufficient allowance of RA to transfer `amount` to the `FlashSwapRouter`
  - No overflows occur on token transfers
  - `SwapperMathLibrary.calculateRolloverSale` returns the following symbolic values that are sufficiently big to prove correctness: (`lvProfit`, `psmProfit`, `raLeft`, `dsReceived`, `lvReserveUsed`, `psmReserveUsed`), representing the result of the rollover sale; to the assertions defined in `SwapperMathLibrary.calculateRolloverSale` hold
  - After the transfer of `amount` RA from the user, the RA balance of `FlashSwapRouter` is sufficient to transfer `lvProfit + psmProfit` to `ModuleCore`
  - `raLeft` is zero, indicating that the whole swap is covered by the rollover sale
  - `dsReceived` is greater than or equal to `amountOutMin` provided by the `Seller`

        - `FlashSwapRouter`'s DS balance is sufficient to transfer `dsReceived` to the `Seller`
        - LV and PSM reserves are greater than or equal to `lvReserveUsed` and `psmReserveUsed`, respectively
        - `ModuleCore.lvAcceptRolloverProfit` correctly accepts LV profit; the logic of the `depositLv` function in the Vault is covered by a separate proof

    - **Assertions:**
        - **After function call:**
            - RA balance of `Seller` decreased by `amount` transferred to `FlashSwapRouter`
            - DS balance of `Seller` increased by the amount of DS tokens attributed to them for the amount of RA provided
            - RA balance of `FlashSwapRouter` increased by the `amount` of RA supplied by the user and decreased by the RA corresponding to the profits made by PSM and Vault
            - `ModuleCore`'s RA balance increased by the RA corresponding to the profits made by PSM and Vault
            - RA corresponding to the profit made by PSM during the rollover sale is accounted for correctly in the respective `states[id].psm.poolArchive[dsId].rolloverProfit` variable
            - `FlashSwapRouter`'s DS token balance decreased by the amount of DS tokens transferred to `Seller` for the supplied RA
            - Accounting variables storing the values of PSM and LV reserves (`assetPair.psmReserves` and `assetPair.lvReserves`) are updated accordingly, i.e., decreased by `psmReserveUsed` and `lvReserveUsed`, respectively

☑ **Test 2: Property-Based Test for Swapping DS for RA Through Flash Swap**

- **Objective:** This test verifies that the `swapDsForRa` function updates the state including token balances as expected.

- **Current Status:** Verified and passed.

- **Details:** Before invoking the `swapDsForRa` function, we make the assumptions about input values and protocol state as described below. Then, we call `swapDsForRa` with symbolic positive `amount` and `amountOutMin` as inputs.

- This test verifies that the `swapDsForRa` function accepts the DS tokens provided by the user in exchange for the DS tokens that are attributed to the user for the RA provided. The DS tokens are obtained through a redemption from Vault via the flash swap mechanism involving a flash loan of the required CT from the AMM.

- **Current Status:** Verified and passed.

- **Details:** Before invoking the `swapDsForRa` function, we are making the assumptions about the protocol state, input parameters, and token balances as described below. Then, we call `swapDsForRa` and check if the resulting state updates are as expected.

  Due to time constraints and the scope of this engagement, we are abstracting the return values and execution of functions related to the `CorkHook` integration, such as `CorkHook.swap` and `CorkHook.getAmountIn`, and we use a minimized implementation of the `CorkCall` callback functionality. Additionally, we abstract the return values of `MathHelper.calculatePercentageFee`, `SwapperMathLibrary.calcHIYAaccumulated`, and `SwapperMathLibrary.calcVHIYAaccumulated`, and make the assumptions about their returned values as outlined below.

- **Assumptions:**
  - `amount` provided by the `Seller` is positive
  - `hiya` is greater than zero and less than 1e18
  - `dsId` is not 1, has a concrete value of 2
  - `Seller` has a sufficient balance of DS to transfer `amount` to the Router
  - `Seller` granted a sufficient allowance of DS to transfer `amount` to the `FlashSwapRouter`
  - No overflows occur on token transfers
  - `CorkHook.getAmountIn` function returns a symbolic value `amountIn` representing the tokens that should be provided to the AMM and that is sufficiently big to prove correctness properties
  - `getAmountIn` is greater than or equal to `amount` provided by the `Seller`
  - The expected `amountOut` that the `Seller` should receive, calculated as `amountIn - amount`, is not less than `amountOutMin`; after the call to `swapDsForRa`, we assert that the expected and actual `amountOut` are equal
  - `CorkHook.swap` assumes that the AMM has enough CT tokens to transfer the requested CT (`amountCtOut`) to the `FlashSwapRouter` contract, executes the transfer, and calls a mocked implementation of the

`CorkCall` callback function (`TestFlashSwapRouter.CorkCallMockDsToRa`)
        - This function performs the following actions: approves the spend of CT and DS to `ModuleCore`, and assumes the `ModuleCore` redeems a symbolic amount of RA is return which is transferred to the `FlashSwapRouter`, then transfers the attributed RA to the `Seller` and repays the flash loan in the AMM. Correctness of the actual `redeemPsm` function is analyzed as a separate proof
        - `ModuleCore` has enough RA to perform the redemption
        - After the redemption, the `FlashSwapRouter`'s RA balance is sufficient to repay the loan and complete the swap by transferring RA to the `Seller`
    - **Assertions:**
        - **After function call:**
            - `Seller` RA balance increased by the RA amount attributed for the DS provided, which is returned from the `swapDsForRa` function
            - `Seller` DS balance decreased by the DS `amount` provided to the `swapDsForRa` function
            - `FlashSwapRouter` RA balance does not change

MathHelper Library Functions Properties

In addition to proving the above mentioned properties, we verified the correctness of several `MathHelper` library functions involved in the calculations, ensuring that many of the assumptions we are making regarding their return values hold. Specifically, these functions are: `calculatePercentageFee`, `calculateProvideLiquidityAmountBasedOnCtPrice`, `calculateRedeemLv`, `calculateEqualSwapAmount`.

The following properties were verified:

calculatePercentageFee

- **Purpose:** The function is designed to calculate the fee that the user should pay during an early redemption from a Vault. This function takes two parameters (`fee1e18` – the fee percentage in 1e18, and the `amount` — the amount of LV the user has provided during the withdrawal). Throughout the codebase, the function is used to calculate the value representing a specific percentage of a variable. It returns a single value:
    - `percentage`: the value representing the `fee1e18` percentage from `amount`

- **Assumptions:** we make the following assumptions about input parameters:

    - `amount` is positive
    - `fee1e18` is between `0.001 * 1e18` and `100 * 1e18`, representing 0.001% and 100%
    - No overflow occurs during multiplication of `amount` and `fee1e18`

- **Properties verified**: we verify that the following properties hold:
    - The return value `percentage` is less than or equal to `amount`
    - `percentage` is equal to `(amount * fee1e18) / (100 * 1e18)`

calculateProvideLiquidityAmountBasedOnCtPrice

- **Purpose:** The function is used to calculate the amount of liquidity in the form of RA and CT tokens that need to be provided to the AMM given the current price ratio and the amount of RA the user has deposited into the Vault. This function takes two parameters (`amountra` – the amount of RA tokens the user provided, and `priceRatio` — the price ratio between the tokens in the pair that is retrieved from the AMM). The function returns two return values:
    - `ra:` the number of RA tokens to be provided as liquidity to AMM
    - `ct`: the number of CT tokens to be provided as liquidity to AMM

- **Assumptions:** we make the following assumptions about input parameters:
    - `priceRatio` is positive
    - No overflow occurs during multiplication of `amountRa` and `1e18`
    - No overflow occurs during multiplication of `priceRatio` and `1e18`

- **Properties verified**: we verify that the following properties hold:
    - The return value `ct` is less than or equal to `amountRa`
    - The return value `ra` is less than or equal to `amountRa`
    - `ct` is equal to `(amountRa * 1e18) / (priceRatio + 1e18)`
    - `ra` is equal to `amountRa - ct`

calculateRedeemLv

- **Purpose:** The function is designed to calculate the result of the LV redemption in the Vault. The function takes an instance of `RedeemParams param` as a parameter summarizing the amount claimed by the user and the state of the Vault:

```
struct RedeemParams {
    uint256 amountLvClaimed;
    uint256 totalLvIssued;
    uint256 totalVaultLp;
```

```
        uint256 totalVaultCt;
        uint256 totalVaultDs;
        uint256 totalVaultPA;
        uint256 totalVaultIdleRa;
    }
```

and returns the `RedeemResult result` structure as a result:

```
    struct RedeemResult {
        uint256 ctReceived;
        uint256 dsReceived;
        uint256 lpLiquidated;
        uint256 paReceived;
        uint256 idleRaReceived;
    }
```

- **Assumptions:** we make the following assumptions about input parameters:
  - `params.amountLvClaimed` is positive
  - `params.amountLvClaimed` is less than `params.totalLvIssued`
  - No overflow occurs during multiplication of `params.amountLvClaimed` and `1e18`
  - `params.totalLvIssued` is less than 2^95
  - `params.totalVaultLp` is less than 2^95
  - `params.totalVaultCt` is less than 2^95
  - `params.totalVaultDs` is less than 2^95

- **Properties verified**: we verify that the following properties hold:
  - Resulting `result.lpLiquidated` is less than or equal to `params.totalVaultLp`
  - `result.ctReceived` is less than or equal to `params.totalVaultCt`
  - result.`dsReceived` is less than or equal to `params.totalVaultDs`

    Additionally, we calculate `proportionalClaim` as `(params.amountLvClaimed * UNIT) / params.totalLvIssued`, and assert the following:

  - `result.ctReceived equals`
    `(proportionalClaim * params.totalVaultCt) / 1e18`
  - `result.dsReceived equals`
    `(proportionalClaim * params.totalVaultDs) / 1e18`
  - `result.lpLiquidated equals`
  - `(proportionalClaim * params.totalVaultLp) / 1e18`

calculateEqualSwapAmount

- **Purpose:** The function is designed to calculate how much RA the user will receive and how much DS they should provide with respect to a given amount of PA based on the current exchange rate between RA and DS + CT. This function takes two parameters (`pa` – the amount of PA provided by the user, and `exchangeRate` — the current exchange rate between RA and (CT + DS). It returns a single value:
  - `amount`: the amount of RA tokens that the user will receive, and the amount of DS tokens that should be provided

- **Assumptions:** we make the following assumptions about input parameters:
  - `exchangeRate` is positive
  - No overflow occurs during multiplication of `pa` and `exchangeRate`

- **Properties verified**: we verify that the following properties hold:
  - The return value `amount` is equal to `(amount * exchangeRate) / 1e18`

Findings During the Engagement

During the formal verification campaign, we identified the following issues:

- **Issue 1: Buying DS Prematurely Reverts During Rollover Sale**

  Impact: In `FlashSwapRouter.swapRaForDs`, the function that attempts to swap RA to DS via rollover contains reverts if the amount received from the rollover sale is less than the `amountOut` parameter provided as input to `swapRaForDs`, instead of proceeding with selling tokens from the reserves and through the flash swap mechanism. This blocks selling DS tokens through the Router in cases if the rollover sale is feasible but only partially covers the swap:

  ```
  if (dsReceived < amountOutMin) {
        revert InsufficientOutputAmount();
  }
  ```

  Resolution: The check should be removed; it should instead be enforced in `swapRaForDs` on the resulting `amountOut` value.

  Status: Fixed in c4ba0581ceb249a1c2c16f7a72f918a3c915a73b.

- **Issue 2: Incorrect Sender Identification in `rollOverCt`**

  Impact: The `PsmCore.rollOverCt` function that does not include the permit-related functionality still accepts `address owner` as an input parameter and uses it as a sender identification. This may allow burning and rolling over another user's tokens, provided that they granted the sufficient allowance previously, and the inconsistency between the sender of a transaction and the account used in calculations and state updates.

  Resolution: rollOverCt should use _msgSender() to reference the account that should be affected by the function execution.

  Status: Fixed in 6bd3099269ddc9f7da8421d8b901322d29d37ffe.

- **Issue 3: Incorrect Pausability Check In VaultCore's Deposit**

  Impact: Both `LVDepositNotPaused` and `LVWithdrawalNotPaused` modifiers check the value of the `states[id].vault.config.isWithdrawalPaused` variable that reflects whether withdrawals, but not deposits, were paused. This disables the pausability mechanism for Liquidity Vault's reposits.

  Resolution: The LVDepositNotPaused modifier should instead check for the value of the `states[id].vault.config.`**`isDepositPaused`** variable.

  Status: Fixed in 59ce98b0a5ac7ca14d29e56827e0cb170acd23c4.

- **Issue 4: Accounting Inconsistency in emptyReservePartialLv**

  Impact: In `DsFlashSwapLibrary.emptyReservePartialLv`, here is a potential inconsistency between the accounting variable storing the value of the LV reserves for a specific DS and the actual DS balance, since the reserves variable is smaller than the amount being transferred by 1:

```
self.ds[dsId].lvReserve -= amount - 1;

self.ds[dsId].ds.transfer(to, amount);
```

  Resolution: The `lvReserve` variable should be decreased by the same value `amount` that is being transferred:

```
self.ds[dsId].lvReserve -= amount;
```

  Status: Fixed in b08590ccbeaa32b01ff503cc955c4680566c6114.

- **Issue 5: Missing Sanity Checks in PSM Initialization**

  Impact: During the initialization of PSM, no sanity checks are enforced on the value of redemption fee (`psmBaseRedemptionFee`), even though it can not take values of more than 5e18 (5%) when updated through `updatePSMBaseRedemptionFeePercentage`. This might lead to the fee taking on invalid values.

  Resolution: Add a sanity check to the initialize function similar to the check present in updatePSMBaseRedemptionFeePercentage:

```
if (newFees > 5 ether) {

    revert ICommon.InvalidFees();

}
```

  Status: Acknowledged by the client.

- **Issue 5: CT and DS Can Have 0 (Infinite) Expiry, Missing Sanity Checks on expiryInterval**

  Impact: LV can have an infinite (0) expiry, but CT and DS assets should not. However, it is possible to initialize a token with such expiration time.

  Resolution: Add a sanity check preventing CT and DS tokens from being initialized with a `0` expiry, e.g., in `ModuleCore.initializeModuleCore` or `AssetFactory.deploySwapAssets`.

  Status: Acknowledged by the client. In the more recent version of the codebase, the `expiry` parameter has been changed to `expiryInterval` that is used to calculate the next expiry time for the DS token as (`block.timestamp + expiryInterval`). There are, however, no sanity checks on the value of `expiryInterval` in `ModuleCore.initializeModuleCore` or `AssetFactory.deploySwapAssets`.

- **Issue 6: RA and PA In a Pair Can Be The Same**

  Impact: RA and PA tokens are assumed to be different by design, however, a sanity check that would prevent them from being set to the same address is missing.

  Resolution: Add a sanity check preventing a Pair from having RA and PA set as the same address, e.g., in `ModuleCore.initializeModuleCore` or

`PairLibrary.initialize`. Alternatively, it is important to ensure that the RA and PA addresses supplied during initialization are correct.

Status: Acknowledged by the client.

- **Issue 7: Accounting Might Become Inconsistent If Tokens Take Fees on Transfers, Transferred Directly**

  Impact: There is a potential inconsistency between the internal accounting and contract token balances in case tokens are transferred to the protocol contracts directly or if they take fees on transfer.

  Resolution: It is essential to guarantee that tokens that take fees on transfer are not onboarded into the protocol or that internal accounting is not used to perform calculations.

  Status: Acknowledged by the client.

- **Issue 8: `previewRepurchase` and `repurchase` will revert if `exchangeRate` is 0**

  Impact: Some of the functions, such as `previewRepurchase` and `repurchase`, calling `MathHelper.calculateDepositAmountWithExchangeRate` will always return if `exchangeRate` is 0, because it will lead to a division by 0.

  Resolution: It is important to guarantee that `exchangeRate` is set to a value that is greater than 0 (for example, 1, as expected), for example, by adding a sanity check on its value.

  Status: Acknowledged by the client.

- **Issue 9: Redundant Expressions, Function Calls Can Be Removed**

  Impact: There are multiple expressions and function calls that can be simplified or removed to reduce the gas consumption and improve readability. For example, the following check that always evaluates to false considering that all variables have the uint256 type:

```
if (x < 0 || y < 0 || e < 0) {
    revert IMathError.InvalidParam();
}
```

or a call to `getReservesSorted` before a revert:

```
if (!success) {

    (uint256 raReserve, uint256 ctReserve) = assetPair.getReservesSorted(hook);

    revert IMathError.InsufficientLiquidity();

}
```

Resolution: These redundant expressions can be safely removed.

Status: Fixed in PR #231.

- **Issue 10: Expressions Can Be Simplified**

  Impact: There are several statements that can be simplified to reduce the gas consumption and improve readability. For example,

  the following expression in `SwapperMathLibrary.getAmountOutBuyDs`

  ```
  s = (r3 * 1e18) / 2e18;
  ```

  is equivalent to the following one, since the division by 2e18 cancels out the 18 decimals precision scaling:

  ```
  s = r3 / 2;
  ```

  Similarly, the following statement in calculatePercentage

  ```
  result = (((amount * 1e18) * percentage) / (100 * 1e18)) / 1e18;
  ```

  can be simplified to

  ```
  result = (amount * percentage) / 1e20;
  ```

  Resolution: These, and similar, expressions can be safely simplified. The equivalence between the two versions of the expressions can be used using the z3 SMT solver with a simple Python script as follows:

  ```
  pip3 install z3-solver

  from z3 import *

  X = Int('X')
  ```

```
lhs = X * 1000000000000000000 / (2 * 1000000000000000000)

rhs = X / 2

s = Solver()

s.add(Not(lhs == rhs))

print(s.check())
```

If the results of these two expressions are not equivalent, this script will print a counterexample, or the value that `X` should take for that to happen.

Status: Fixed in [6500f20cf41d6a507d379676eb0ee7506ebd8f06](6500f20cf41d6a507d379676eb0ee7506ebd8f06).

Conclusion

This engagement is a good example of how formal verification can be crucial in ensuring the correctness of protocols. The specification of invariants and properties helped not only identify issues but also be confident that the desired properties hold under certain pre-conditions.

All the properties proven in the tests hold **under the assumptions** mentioned in each test. It is very important to ensure that when these functions are called the corresponding assumptions hold, otherwise preservation of the invariants is not guaranteed.

An interesting follow-up engagement would be to prove the invariants for the `CorkHook` functions that are being invoked by different components verified within the scope of this engagement, as well as the functions of other contracts, such as `CorkHook`, `Liquidator` and `HedgeUnit`. In addition, the proofs for the `FlashSwapRouter` contract could be extended further to cover all possible scenarios of its execution. By following this approach, we would ensure the correctness of the entire system.