Custom Property Fuzzing Results for Gearbox Smart Contracts

Consensys Diligence

12/13/2021

Executive Summary

This report presents the results of a property writing and fuzzing engagement with *Gearbox* on the core `PoolService` and `CreditManager` contracts. During this engagement, 24 properties of the contracts under scope were written in the Scribble specification language. The properties were then compiled into executable assertions alongside the original code, and this instrumented code was subjected to several fuzzing campaigns.

We ran about 370 CPU hours of exploratory fuzzing campaigns. During those the fuzzer discovered several edge cases that required fixes to our original properties and limiting the fuzzing state space to avoid unrealistic scenarios (e.g., billions of years between transactions).

Finally, we ran a 96 CPU hours fuzzing campaign with the fixed properties. In the final campaign the fuzzer reached 21 of the 24 properties and we found **0 violations**. Manual inspection found that one of the 24 properties may still be violated by an edge case discovered earlier by the fuzzer. However that violation scenario seems benign from a security point of view (equivalent to an attacker donating funds to the pool).

Overview

In this section we provide a brief overview of our methodology, related concepts and our fuzzing setup.

Property Fuzzing vs. Audits

This document summarizes the results of fuzz testing the gearbox system and not a manual review.

During this engagement, we formalized a range of desirable properties for the Gearbox system, that were consecutively tested using the Diligence Fuzzing platform. The scope of the engagement and this document are therefore limited to the tested properties.

The absence of fuzzing violations for a given property does not necessarily prove that the property holds for all inputs. The space of possible inputs to the contracts under scope is unbounded, and the fuzzer can only spend a finite amount of time exploring this space. The results of fuzzing are meant to increase confidence in the stated properties, *alongside* manual inspection.

Methodology

During this engagement, with the assistance of Gearbox, we defined 24 properties specifying aspects of the behavior of the contracts in scope. These properties appear as comments in the original code that do not affect the behavior of the contracts. We used an automated tool (`scribble`) to convert the property comments into actual code inserted alongside the original program (this process is called 'instrumentation').

The instrumented code was then compiled down to EVM and passed to our in-house grey-box fuzzer Harvey.

We refined the initial properties by running several iterations of short fuzzing campaigns, analyzing any failures found, and either fixing the initial properties or limiting the state space of the fuzzer.

One important limitation we imposed was restricting the duration between consecutive operations, updating the borrowing rate on the pool contract to less than 24h. In early campaigns, we found that the fuzzer could violate 4 of the written properties. However, the counterexamples it found required billions of years between consecutive transactions. We considered this an unrealistic scenario, so we limited that time duration.

Scribble properties

Properties added to the contracts were in the Scribble language, and appeared in the docstring before certain functions. Below is an example of one such property:

```
/**
 * #if_succeeds
 *     {:msg "Can only be called by account holder"}
 *         old(creditAccounts[msg.sender]) != address(0x0);
 */
function repayCreditAccount(address to)
```

The above example property states that if `repayCreditAccount` succeeds, then it must have been called by an address that already had a `CreditAccount`. After instrumentation the above comment is translated into an actual check inserted into the code as shown below:

```
function repayCreditAccount(address to) override external {
    old_7 = creditAccounts[msg.sender];
    ...
    _original_CreditManager_repayCreditAccount(to);
    ...
    if (!(old_7 != address(0x0))) {
        emit AssertionFailed("12: Can only be called by account holder");
        assert(false);
    }
```

Fuzzing

Fuzzing is an automatic testing and property checking technique. Our architecture is based on the most recent developments in academia, including the contributions of ConsenSys Diligence:

- Harvey: A Greybox Fuzzer for Smart Contracts (ESEC/FSE 2020)
- Targeted Greybox Fuzzing with Static Lookahead Analysis (ICSE 2020)

Briefly, our in-house fuzzer (Harvey) generates a large set of initial random inputs and observes the execution of the contracts under scope for those inputs. Next, the fuzzer utilizes interpolation and domain-specific optimizations to look for new inputs that explore more branches in the target contract. Since the original properties were translated into branches in the instrumented code, the fuzzer will aim to find inputs that violate those properties (i.e., flip their corresponding branches).

Scope

Our fuzzing campaign targeted the following core contracts at commit hash `fb73c17105a2d0a8de2c9b3b2da55628700167eb`.

- PoolService
- CreditManager

24 annotations were added to the `PoolService` and `CreditManager` contracts. The code for the fuzzed smart contracts is located here.

Fuzzing State Setup

We derived the initial configuration from the standard network deployment script of GearBox with the following modifications:

1. We deployed 5 mock tokens.
2. We deployed a mock Uniswap V2 router contract.
3. We deployed mock pricing oracles for each pair of mock tokens.
4. We distributed initial tokens and ETH to several attacker accounts
5. We provided initial liquidity to the single pool in the configuration
6. One of the attacker accounts opened a credit account and executed a sample swap.

The initial configuration contains a single `PoolService` as well as a single connected `CreditManager`, with a single opened `CreditAccount` belonging to one of the attacker accounts.

Starting from this initial configuration, our fuzzer explored the system under test with the following two additional limitations and hints:

1. No more than 24h were allowed between the block timestamps of 2 consecutive updates to the borrow rate in the PoolService (specifically updates to the `_borrowAPY_RAY` state variable). This limitation was added as `Harvey` quickly found several violations to the properties under test that required billions of years of time between transactions. After discussions with developers, we agreed that 24 hours is a realistic upper limit on the time between updates to the internal state.
2. We put a limitation on `Harvey` not to update the interest rate model of the pool under test, since that is a privileged operation and abusing it can trivially break the system.

Coverage

The final fuzzing campaign ran for 96 CPU hours (12-hour in real-time with 8 cores).

In this time, we were able to achieve significant coverage. We outline basic coverage metrics in the table below.

Coverage Metrics

Metric	Value
CPU Hours	96
Covered Instructions	90612
Covered Paths	10992
Covered BB Transitions	9893
Generated Tests	85508456

Metric	Value
Residual Risk	0.0000001169%
Discovered Violations	0

In the above table *Covered Instructions* refers to the total number of EVM instructions covered in all contracts (not just the two target contracts). *Covered Paths* refers to the total number of unique code paths explored by the fuzzer. *Covered BB Transitions* refers to the total number of unique control-flow edges (i.e. jumps) exercised by the fuzzer. *Generated Tests* refers to the total number of unique test cases the fuzzer generated. Finally *Residual Risk* is an estimate of the probability that the fuzzer would discover new branches, given the trends in its rate of discovery.

Coverage over time

In addition, we would like to highlight the progression of three of the primary coverage metrics over time. In the below graphs, we show the change in the number of covered instructions (Fig. 1), the number of discovered branch transitions (Fig. 2), and the number of discovered paths (Fig. 3) as the campaign progressed. Note that the x-axis of the instruction coverage graph in Fig. 1 goes up to 6h instead of 12h, as the last newly discovered instruction during the campaign was found around the 6h mark.

The sharp jump at the beginning of all 3 graphs is due to our re-use of the corpus of discovered tests from earlier exploratory fuzzing campaigns during this engagement.

Both instructions covered over time and branch transitions covered over time level off towards the end of the 12-hour window. It is interesting that the number of discovered paths continues to rise steadily, despite the number of new branches discovered leveling out. This suggests that the fuzzer is still able to find new paths, that are permutations of already covered branches.

Additionally, the fuzzer reports statistics on the set of paths which discovered a new instruction.

As shown in Fig 4, the majority of paths that increased coverage consisted of 2 transactions. The longest transaction sequences that increased coverage consisted of 8 transactions.

Results

During the fuzzing campaign, we tested the correctness of 24 properties. Of those 24 properties, 21 were reached at least once, and 0 violations were found among those. Please refer to the Annotations appendix below for a complete list of added annotations.

In earlier exploratory campaigns, the fuzzer found several edge cases that forced us to re-think our properties and are useful to outline here:

1. Setting the payment recipient argument of several functions (e.g. `CreditManager.closeCreditAccount`, `CreditManager.repayCreditAccount`, `CreditManager.liquidateCreditAccount`, etc.) to a known contract in the system (e.g., the treasury, an existing `CreditAccount`, or the pool itself). These edge cases can break simple invariants that check for exact changes in the balance of the pool or the treasury across repayment, closing, and liquidation operations. To account for this we modified Property 7 and Property 11 by weakening them and/or adding additional guards. These

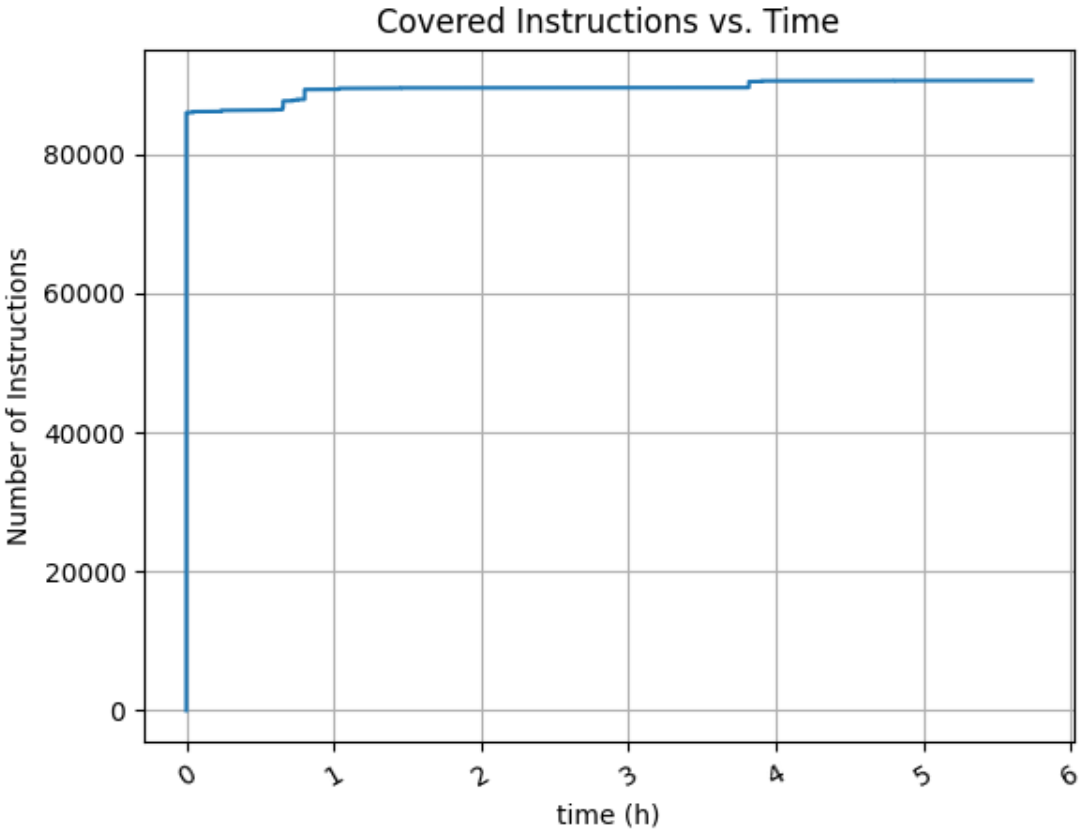

Figure 1: Instruction Covered over time

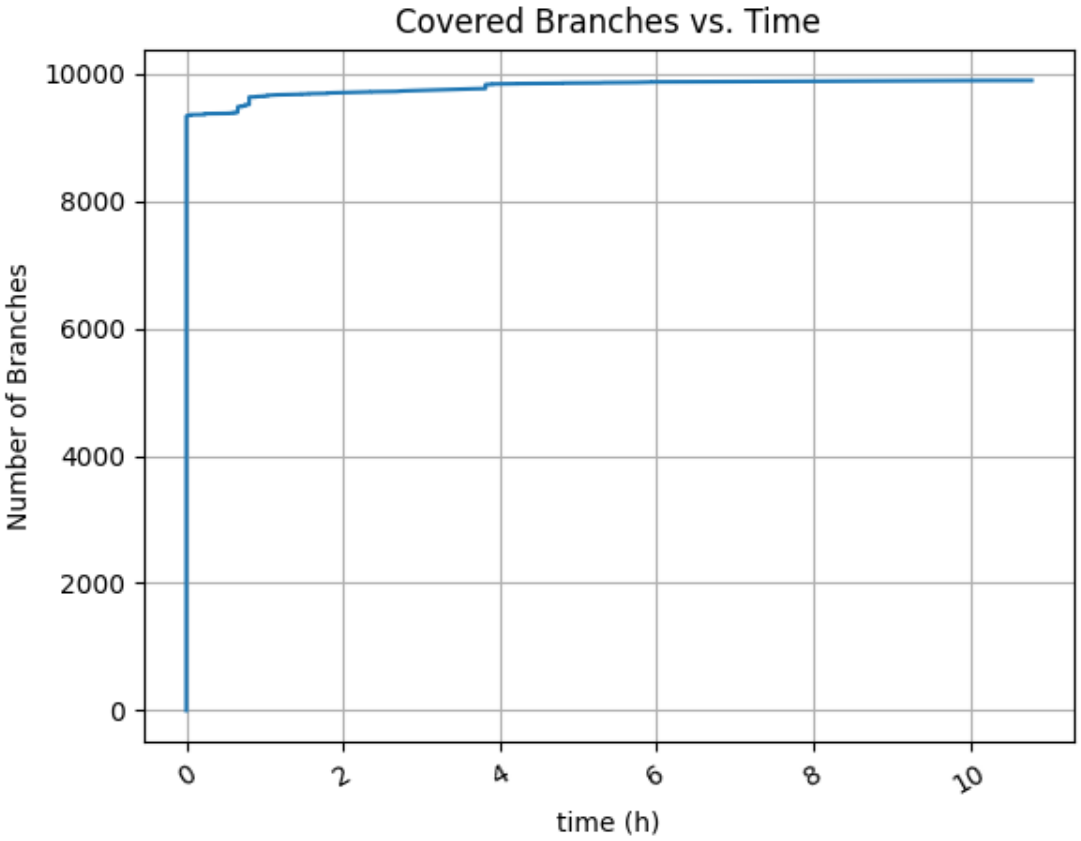

Figure 2: Branch Transitions over time

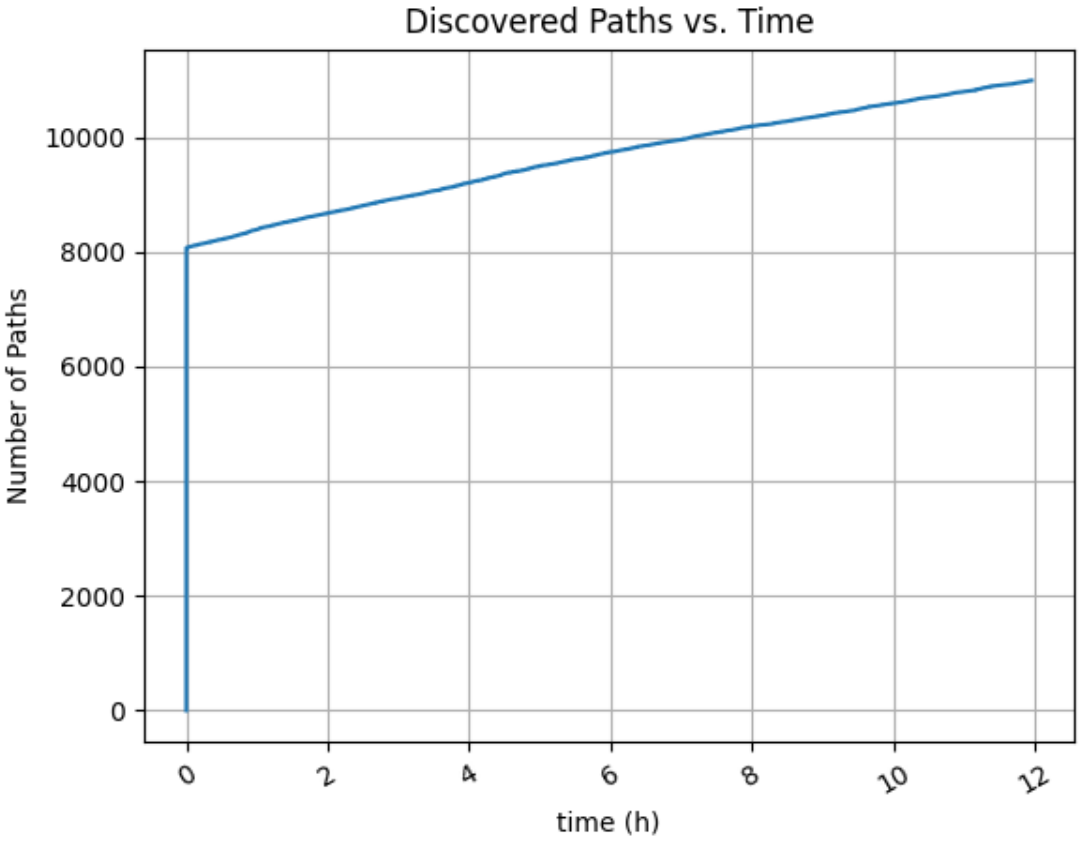

Figure 3: Paths Covered over time

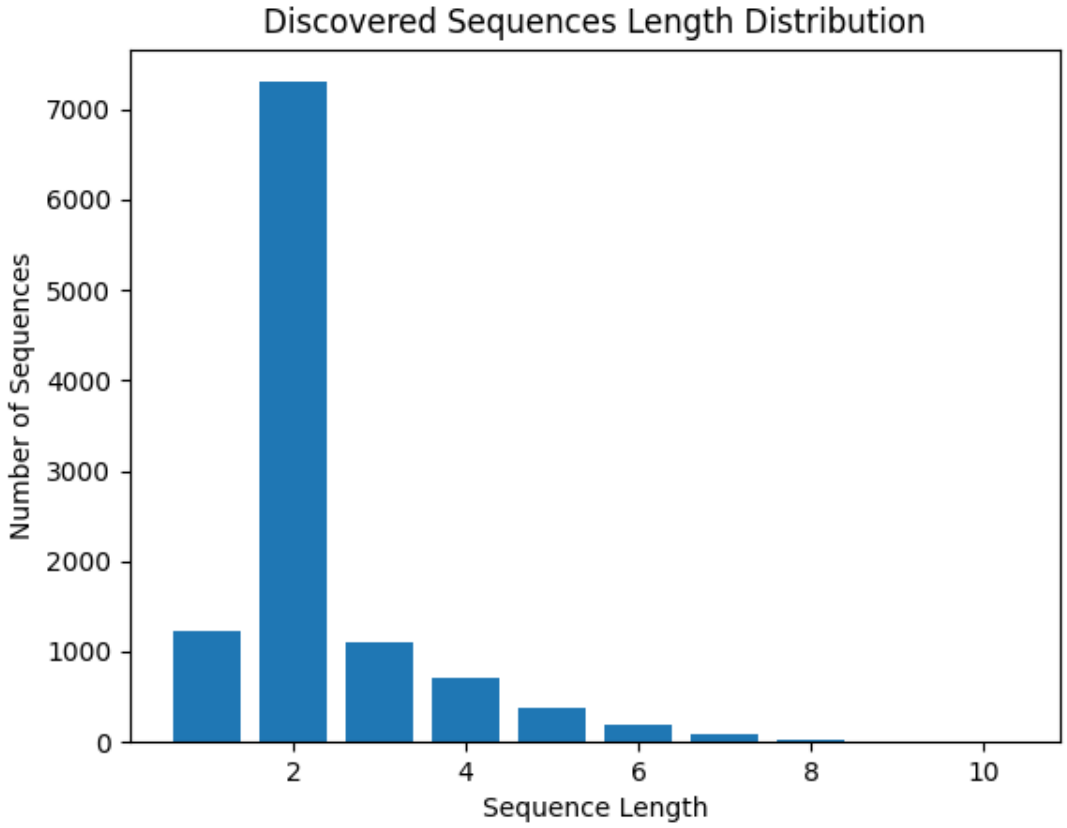

Figure 4: Distribution of transaction sequence lengths

edge cases so far seem benign, as their result is that the caller willingly forfeits their earnings in favor of the pool itself.

2. As an optimization, `CreditAccounts` can sometimes be left with a balance of `1` for tokens. Thus certain equality operations in invariants needed to be weakened (e.g. Property 11)
3. In earlier campaigns, the fuzzer found violations for several of the properties when unrestricted control over the block timestamp was allowed (namely properties 8, 11, 13, and 26 in the Annotations appendix). All failures involved the fuzzer putting billions of years of time difference between block timestamps, which we felt was unrealistic. To silence those failures, we added a restriction that updates to internal state (borrowing rate) happen no more than 24h apart.
4. Finally even though the fuzzer did not violate Property 15 (see Annotations appendix), given more time it would have likely violated it by sending unexpected funds to an unused `CreditAccount` before its re-use (similar to how it violated Property 7 earlier). Again this violation is likely benign, as it amounts to an attacker donating their own funds to future borrowers of the pool.

Recommendations

Since the deployed GearBox system is relatively complex, we recommend continued fuzzing to increase coverage and explore other system components. Here are several examples of future fuzzing campaigns to further explore the system:

1. A campaign with either no limit on timestamps or longer limits on timestamps would be helpful to explore the behavior of the system when long disruption periods are possible.
2. While this campaign reached the code of `CreditFilter` and `UniswapV2Adapter`, we believe these components are important enough to warrant additional focused campaigns. Specifically, these campaigns would entail formulating Scribble properties for those contracts and tweaking the fuzzing set up to target those contracts directly.
3. Our fuzzing campaigns did not exercise 3 of the written properties. Specifically properties related to `CreditManager.liquidateCreditAccount` and `CreditManager.repayCreditAccountETH` were not exercised, as the fuzzer did not find suitable inputs for those functions. Further work is required to assist the fuzzer in exercising those functions.
4. During our campaign, the fuzzer was free to change the exchange rate between any two tokens as reported by the mock oracles whenever it chooses. However, since the fuzzer is guided by increasing coverage, if it has already covered the functions for changing exchange rates, then it has no incentive to revisit them. Thus we believe it would be helpful to perform another campaign, where we force the fuzzer (potentially by using harness contracts) to change the price of some tokens at given key points. This would simulate more aggressive flash-loan scenarios.
5. A core property that we were not able to express in this engagement was that

"As long as for each token T_i, its price P_i does not decrease by more than D_i in a time interval I_1, and as long as for each credit account A_j, a liquidation is run within time I_2 from the moment that it becomes unhealthy, then the pool will not lose money."

Expressing this property requires temporal logic or ghost state, which Scribble currently does not support.

We plan on adding such support in the coming few months. Until then, we recommend manual inspection and additional focused campaigns with manually added ghost code expressing the above property.

Appendix: Annotations

Properties

Property 1: After addLiquidity() the pool gets the correct amount of underlyingToken(s)

Status: **PASSED**

Location: `PoolService.addLiquidity` in `contracts/pool/PoolService.ts`

```
    * #if_succeeds
    *   {:msg "After addLiquidity() the pool gets the correct amount of underlyingToken(s)"}
    *      IERC20(underlyingToken).balanceOf(address(this)) ==
    *      old(IERC20(underlyingToken).balanceOf(address(this))) + amount;
...
function addLiquidity(
    uint256 amount,
    address onBehalfOf,
    uint256 referralCode
)
...
```

Property 2: After addLiquidity() onBehalfOf gets the right amount of dieselTokens

Status: **PASSED**

Location: `PoolService.addLiquidity` in `contracts/pool/PoolService.ts`

```
     * #if_succeeds {:msg "After addLiquidity() onBehalfOf gets the right amount of dieselTokens"}
     *      IERC20(dieselToken).balanceOf(onBehalfOf) ==
     *      old(IERC20(dieselToken).balanceOf(onBehalfOf)) + old(toDiesel(amount));
...
    function addLiquidity(
        uint256 amount,
        address onBehalfOf,
        uint256 referralCode
    )
...
```

Property 3: After addLiquidity() borrow rate decreases

Status: **PASSED**

Location: `PoolService.addLiquidity` in `contracts/pool/PoolService.ts`

```
     * #if_succeeds {:msg "After addLiquidity() borrow rate decreases"}
     *      amount > 0 ==> borrowAPY_RAY <= old(currentBorrowRate());
...
    function addLiquidity(
        uint256 amount,
        address onBehalfOf,
        uint256 referralCode
    )
...
```

Property 4: For removeLiquidity() sender must have sufficient diesel

Status: **PASSED**

Location: `PoolService.removeLiquidity` in `contracts/pool/PoolService.ts`

```
     * #if_succeeds {:msg "For removeLiquidity() sender must have sufficient diesel"}
     *      old(DieselToken(dieselToken).balanceOf(msg.sender)) >= amount;
...
    function removeLiquidity(uint256 amount, address to)
...
```

Property 5: After removeLiquidity() to gets the liquidity in underlyingToken(s)

Status: **PASSED**

Location: `PoolService.removeLiquidity` in `contracts/pool/PoolService.ts`

```
     * #if_succeeds {:msg "After removeLiquidity() `to` gets the liquidity in underlyingToken(s)"}
     *      (to != address(this) && to != treasuryAddress) ==>
     *          IERC20(underlyingToken).balanceOf(to) ==
     *          old(IERC20(underlyingToken).balanceOf(to) +
     *              (let t:= fromDiesel(amount) in t.sub(t.percentMul(withdrawFee))));
...
    function removeLiquidity(uint256 amount, address to)
...
```

Property 6: After removeLiquidity() treasury gets the withdraw fee in underlyingToken(s)

Status: **PASSED**

Location: `PoolService.removeLiquidity` in `contracts/pool/PoolService.ts`

```
     * #if_succeeds
     *   {:msg "After removeLiquidity() treasury gets the withdraw fee in underlyingToken(s)"}
     *      (to != address(this) && to != treasuryAddress) ==>
```

```
     *           IERC20(underlyingToken).balanceOf(treasuryAddress) ==
     *           old(IERC20(underlyingToken).balanceOf(treasuryAddress) +
     *               fromDiesel(amount).percentMul(withdrawFee));
...
    function removeLiquidity(uint256 amount, address to)
```

Property 7: After removeLiquidity() borrow rate increases

Status: **PASSED**

Location: `PoolService.removeLiquidity` in `contracts/pool/PoolService.ts`

```
     * #if_succeeds {:msg "After removeLiquidity() borrow rate increases"}
     *       (to != address(this) && amount > 0) ==> borrowAPY_RAY >= old(currentBorrowRate());
...
    function removeLiquidity(uint256 amount, address to)
```

Note: In its original form, the fuzzer violated this property by setting the destination of the liquidity back to the pool itself. The property was strengthened with the precedent that `to != address(this)` to avoid this edge case.

Property 8: After lendCreditAccount() borrow rate increases

Status: **PASSED**

Location: `PoolService.lendCreditAccount` in `contracts/pool/PoolService.ts`

```
    * #if_succeeds {:msg "After lendCreditAccount() borrow rate increases"}
    *       borrowedAmount > 0 ==> borrowAPY_RAY >= old(currentBorrowRate());
...
    function lendCreditAccount(uint256 borrowedAmount, address creditAccount)
```

Property 9: Can't have both profit and loss

Status: **PASSED**

Location: `PoolService.repayCreditAccount` in `contracts/pool/PoolService.ts`

```
    * #if_succeeds {:msg "Cant have both profit and loss"} !(profit > 0 && loss > 0);
...
    function repayCreditAccount(
        uint256 borrowedAmount,
        uint256 profit,
        uint256 loss
    )
```

Property 10: After repayCreditAccount() if we are profitable, or treasury can cover the losses, diesel rate doesn't decrease

Status: **PASSED**

Location: `PoolService.repayCreditAccount` in `contracts/pool/PoolService.ts`

```
    * #if_succeeds
    *     {:msg "After repayCreditAccount() if we are profitable, or treasury can cover the losses,
    *             diesel rate doesn't decrease"}
    *      (profit > 0 || toDiesel(loss) >= DieselToken(dieselToken).balanceOf(treasuryAddress))
    *          ==> getDieselRate_RAY() >= old(getDieselRate_RAY());
...
    function repayCreditAccount(
        uint256 borrowedAmount,
        uint256 profit,
        uint256 loss
    )
```

Property 11: A credit account with the correct balance is opened.

Status: **PASSED**

Location: `CreditManager.openCreditAccount` in `contracts/credit/CreditManager.ts`

```
     * #if_succeeds {:msg "A credit account with the correct balance is opened."}
     *      let newAccount := creditAccounts[onBehalfOf] in
     *      newAccount != address(0) &&
     *          IERC20(underlyingToken).balanceOf(newAccount) >=
     *          amount.add(amount.mul(leverageFactor).div(Constants.LEVERAGE_DECIMALS));
...
    function openCreditAccount(
        uint256 amount,
        address onBehalfOf,
        uint256 leverageFactor,
        uint256 referralCode
    )
```

Note that the annotation as written checks that the new `CreditAccount` has **at least** the expected amount. Initially, we tried checking that the new account has exactly the collateral plus leverage. However, the fuzzer found two edge cases due to which that property didn't hold as written:

1. As a gas optimization, the system may leave `CreditAccount`'s with a balance of 1 after closing.
2. `CreditAccounts` can receive funds directly even when not used (for example, when specified as recipients for liquidation/repayment).

We found these violations to be benign after discussion with the Gearbox development team.

Property 12: Sender loses amount tokens.

Status: **PASSED**

Location: `CreditManager.openCreditAccount` in `contracts/credit/CreditManager.ts`

```
     * #if_succeeds {:msg "Sender looses amount tokens." }
     *      IERC20(underlyingToken).balanceOf(msg.sender) ==
     *      old(IERC20(underlyingToken).balanceOf(msg.sender)) - amount;
...
    function openCreditAccount(
        uint256 amount,
        address onBehalfOf,
        uint256 leverageFactor,
        uint256 referralCode
    )
```

Property 13: Pool provides correct leverage (amount x leverageFactor).

Status: **PASSED**

Location: `CreditManager.openCreditAccount` in `contracts/credit/CreditManager.ts`

```
     * #if_succeeds {:msg "Pool provides correct leverage (amount x leverageFactor)." }
     *      IERC20(underlyingToken).balanceOf(poolService) ==
     *      old(IERC20(underlyingToken).balanceOf(poolService)) -
     *          amount.mul(leverageFactor).div(Constants.LEVERAGE_DECIMALS);
...
    function openCreditAccount(
        uint256 amount,
        address onBehalfOf,
        uint256 leverageFactor,
        uint256 referralCode
    )
```

Property 14: The new account is healthy.

Status: **PASSED**

Location: `CreditManager.openCreditAccount` in `contracts/credit/CreditManager.ts`

```
     * #if_succeeds {:msg "The new account is healthy."}
     *      creditFilter.calcCreditAccountHealthFactor(creditAccounts[onBehalfOf]) >=
     *      PercentageMath.PERCENTAGE_FACTOR;
...
    function openCreditAccount(
        uint256 amount,
        address onBehalfOf,
        uint256 leverageFactor,
```

```
        uint256 referralCode
    )
```

Property 15: The new account has balance <= 1 for all tokens other than the underlying token.

Status: **PASSED**

Location: `CreditManager.openCreditAccount` in `contracts/credit/CreditManager.ts`

```
     * #if_succeeds
     *  {:msg "The new account has balance <= 1 for all tokens other than the underlying token."}
     *     let newAccount := creditAccounts[onBehalfOf] in
     *         forall (uint i in 1...creditFilter.allowedTokensCount())
     *             IERC20(creditFilter.allowedTokens(i)).balanceOf(newAccount) <= 1;
...
    function openCreditAccount(
        uint256 amount,
        address onBehalfOf,
        uint256 leverageFactor,
        uint256 referralCode
    )
```

Note that while the fuzzer did not violate this property, one can trivially violate it by sending tokens to `CreditAccounts` between uses. This violation is benign, as it just adds funds to the system at the expense of a potential attacker.

Property 16: Can only be called by account holder

Status: **PASSED**

Location: `CreditManager.closeCreditAccount` in `contracts/credit/CreditManager.ts`

```
     * #if_succeeds {:msg "Can only be called by account holder"}
     *  old(creditAccounts[msg.sender]) != address(0x0);
...
    function closeCreditAccount(address to, DataTypes.Exchange[] calldata paths)
```

Property 17: Can only close healthy accounts

Status: **PASSED**

Location: `CreditManager.closeCreditAccount` in `contracts/credit/CreditManager.ts`

```
     * #if_succeeds {:msg "Can only close healthy accounts" }
     *     old(creditFilter.calcCreditAccountHealthFactor(creditAccounts[msg.sender])) >
     *     PercentageMath.PERCENTAGE_FACTOR;
```

```
...
    function closeCreditAccount(address to, DataTypes.Exchange[] calldata paths)
```

Property 18: If this succeeded, the pool gets paid at least borrowed + interest

Status: **PASSED**

Location: `CreditManager.closeCreditAccount` in `contracts/credit/CreditManager.ts`

```
     * #if_succeeds {:msg "If this succeeded the pool gets paid at least borrowed + interest"}
     *     let minAmountOwedToPool := old(borrowedPlusInterest(creditAccounts[msg.sender])) in
     *         IERC20(underlyingToken).balanceOf(poolService) >=
     *           old(IERC20(underlyingToken).balanceOf(poolService)).add(minAmountOwedToPool);
...
    function closeCreditAccount(address to, DataTypes.Exchange[] calldata paths)
```

Property 19: Can only be called by account holder

Status: **NOT REACHED**

Location: `CreditManager.liquidateCreditAccount` in `contracts/credit/CreditManager.ts`

```
     * #if_succeeds {:msg "Can only be called by account holder"}
     *     old(creditAccounts[msg.sender]) != address(0x0);
...
    function liquidateCreditAccount(
        address borrower,
        address to,
        bool force
    )
```

Property 20: Can only liquidate an un-healthy account

Status: **NOT REACHED**

Location: `CreditManager.liquidateCreditAccount` in `contracts/credit/CreditManager.ts`

```
     * #if_succeeds {:msg "Can only liquidate an un-healthy accounts" }
     *      old(creditFilter.calcCreditAccountHealthFactor(creditAccounts[msg.sender])) <
     *        PercentageMath.PERCENTAGE_FACTOR;
...
    function liquidateCreditAccount(
        address borrower,
        address to,
        bool force
    )
```

Property 21: Can only be called by account holder

Status: **PASSED**

Location: `CreditManager.repayCreditAccount` in `contracts/credit/CreditManager.ts`

```
    * #if_succeeds {:msg "Can only be called by account holder"}
    *     old(creditAccounts[msg.sender]) != address(0x0);
...
    function repayCreditAccount(address to)
```

Property 22: If this succeeded the pool gets paid at least borrowed + interest

Status: **PASSED**

Location: `CreditManager.repayCreditAccount` in `contracts/credit/CreditManager.ts`

```
    * #if_succeeds {:msg "If this succeeded the pool gets paid at least borrowed + interest"}
    *     let minAmountOwedToPool := old(borrowedPlusInterest(creditAccounts[msg.sender])) in
    *         IERC20(underlyingToken).balanceOf(poolService) >=
    *         old(IERC20(underlyingToken).balanceOf(poolService)).add(minAmountOwedToPool);
...
    function repayCreditAccount(address to)
```

Property 23: If this succeeded the pool gets paid at least borrowed + interest

Status: **NOT REACHED**

Location: `CreditManager.repayCreditAccountETH` in `contracts/credit/CreditManager.ts`

```
    * #if_succeeds {:msg "If this succeeded the pool gets paid at least borrowed + interest"}
    *     let minAmountOwedToPool := old(borrowedPlusInterest(creditAccounts[borrower])) in
    *         IERC20(underlyingToken).balanceOf(poolService) >=
    *         old(IERC20(underlyingToken).balanceOf(poolService)).add(minAmountOwedToPool);
...
    function repayCreditAccountETH(address borrower, address to)
```

Property 24: Credit account balances should be <= 1 for all allowed tokens after closing

Status: **PASSED**

Location: `CreditManager._closeCreditAccountImpl` in `contracts/credit/CreditManager.ts`

```
    * #if_succeeds
    *     {:msg "Credit account balances should be <= 1 for all allowed tokens after closing"}
    *     forall (uint i in 0...creditFilter.allowedTokensCount())
    *         IERC20(creditFilter.allowedTokens(i)).balanceOf(creditAccount) <= 1;
...
    function _closeCreditAccountImpl(
```

Helpers

To assist in the annotation 2 Scribble helper functions were added as shown below:

```
* #define borrowedPlusInterest(address creditAccount) uint =
*     let borrowedAmount, cumIndexAtOpen := getCreditAccountParameters(creditAccount) in
*     let curCumulativeIndex := IPoolService(poolService).calcLinearCumulative_RAY() in
*         borrowedAmount.mul(curCumulativeIndex).div(cumIndexAtOpen);
...
contract CreditManager is ICreditManager, ACLTrait, ReentrancyGuard {
```

```
* #define currentBorrowRate() uint =
*     let expLiq := expectedLiquidity() in
*     let availLiq := availableLiquidity() in
*         interestRateModel.calcBorrowRate(expLiq, availLiq);
...
contract PoolService is IPoolService, ACLTrait, ReentrancyGuard {
```

Limitations

We introduced the following 'limiting' annotations to prevent the fuzzer from wasting energy on uninteresting states.

1. Several annotations can be violated by opening a credit account, waiting a very long time (in the fuzzer examples billions of years), and then attempting to close or repay the accounts. This scenario was deemed unrealistic as there is an implicit expectation that some action is taken within a timely manner. To limit the fuzzer, we added limitations of the form shown below to several functions of `PoolService`:

   ```
   * #limit {:msg "Not more than 1 day since last borrow rate update"}
   *     block.timestamp <= _timestampLU + 3600 * 24;
   ```

The effect of the limitation is that actions that impact the borrow rate (including opening, closing, liquidating and repaying credit account, adding and removing liquidity from pools) cannot be performed more than 24 hours apart.

2. The administrative function `PoolService.updateInterestRateModel` was placed off-limits to the fuzzer with the following annotation:

   ```
   * #limit {:msg "Disallow updating the interest rate model after the constructor"}
   *     address(interestRateModel) == address(0x0);
   ```

The intent is to prevent the fuzzer from setting the `interestRateModel` to an invalid contract and thus trivially breaking the system