Formal Verification of Saddle Finance

February 2023

Summary

This document describes the specification and verification of Saddle's contract Swap using the Certora Prover. The latest commit that was reviewed and run through the Certora Prover was 49734a33.

The scope of this verification is Saddles's contract Swap.sol.

The Certora Prover proved the implementation of the contract above is correct with respect to formal specifications written by the Saddle and Certora teams.

The formal specifications focus on validating correct behavior for Swap.sol as described by the Saddle team and the contract documentation. The rules verify valid states for the system, proper transitions between states, method integrity, and high-level properties (which often describe more than one element of the system and can even be cross-system).

The formal specifications have been submitted into a seperate branch in Saddle's public git repository.

Disclaimer

The Certora Prover takes as input a contract and a specification and formally proves that the contract satisfies the specification in all scenarios. Importantly, the guarantees of the Certora Prover are scoped to the provided specification, and the Certora Prover does not check any cases not covered by the specification.

We hope that this information is useful, but 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.

Summary of formal verification

Notations

✅ passing indicates the rule is formally verified on the latest reviewed commit.
❌ failing indicates the rule was violated under one of the tested versions of the code.
✍️ todo indicates the rule is not yet formally specified.
⏱️ timeout indicates that some functions cannot be verified because the rules timed out.

Assumptions and simplifications for verification

- Due to the complexity of `removeLiquidityImbalance` it was considered out of scope for this verification.
- We unroll loops. Violations that require a loop to execute more than twice will not be detected. This means we assume at most 2 tokens per pool.
- The scope of this verification ignores many of the complex math in the contract. We overapproximate `getD`, `getY` and `getYD` meaning we assume they can return any value for any given inputs and proved properties under these conditions. In some cases we summarized getD to return a value between the constant sum and constant product invariants.
- Many proven properties or corollaries of properties which were proven are often assumed in other rules. These properties are defined in the following CVL function called `basicAssumptions`.

```
function basicAssumptions(env e) {
      requireInitialized();
      assumeNormalDecimals(1);
      requireInvariant oneUnderlyingZeroMeansAllUnderlyingsZero(0);
      requireInvariant oneUnderlyingZeroMeansAllUnderlyingsZero(1);
      requireInvariant LPsolvency();
      require lpToken.balanceOf(e, e.msg.sender) <= getTotalSupply();
      requireInvariant underlyingsSolvency();
      requireInvariant underlyingTokensAndLPDifferent();
      requireInvariant underlyingTokensDifferent(0,1);
      requireInvariant lengthsAlwaysMatch();
      requireInvariant adminFeeNeverGreaterThanMAX();
      requireInvariant swapFeeNeverGreaterThanMAX();
      require e.msg.sender != currentContract;
}
```

Verification of Swap.sol

Invariants

(✓) *oneUnderlyingZeroMeansAllUnderlyingsZero*

If balance of one underlying token is zero, the balance of all other underlying tokens must also be zero.

```
basicAssumptions() =>
(swapStorage.balances[i] == 0 => sum_all_underlying_balances == 0)
```

(✓) *ifSumUnderlyingsZeroLPTotalSupplyZero*

If the sum of all underlying tokens is 0 then the total supply of `lpToken` must be 0

```
basicAssumptions() =>
(sum_all_underlying_balances == 0 => swapStorage.lpToken.totalSupply() ==
```

(✓) *ifLPTotalSupplyZeroThenIndividualUnderlyingsZero*

If total supply of `lpToken` is zero then every underlying token balance must also be zero.

```
basicAssumptions() =>
      swapStorage.lpToken.totalSupply() == 0 => getTokenBalance(i) == 0
```

(✓) *underlyingTokensAndLPDifferent*

Underlying tokens must be different from the LP token.

```
swapStorage.lpToken != swapStorage.pooledTokens[0]
&& swapStorage.lpToken != swapStorage.pooledTokens[1]
```

(✓) *underlyingTokensDifferent*

Underlying tokens must be different from eachother.

```
i != j => swapStorage.pooledTokens[i] ! swapStorage.pooledTokens[j]
```

(✓) *swapFeeNeverGreaterThanMAX*

`swapFee` can never be greater `MAX_SWAP_FEE`.

```
swapStorage.swapFee <= MAX_SWAP_FEE
```

(✓) *adminFeeNeverGreaterThanMAX*

`adminFee can never be greater MAX_ADMIN_FEE.`

```
swapStorage.adminFee <= MAX_ADMIN_FEE
```

(✓) *LPSolvency*

Sum of all users' LP balance must be equal to LP's `totalSupply`.

```
getTotalSupply() == sum_all_users_LP
```

(✓) *underlyingsSolvency*

Sum of all underlying balances must equal the contract's sum.

```
getSumOfUnderlyings() == sum_all_underlying_balances
```

(✓) *LPTotalSupplyZeroWhenUninitialized*

LPToken totalSupply must be zero if `addLiquidity` has not been called.

```
f != addLiqudity(...) => getTotalSupply() == 0
```

(✓) *lengthsAlwaysMatch*

The length of `pooledTokens` must match the length of `balances`.

```
swapStorage.pooledTokens.length == swapStorage.balances.length;
```

Rules

(✓) *cantReinit*

Contract can't be initialized again if it has already been initialized.

```
{
    initialized == true && initializing == false
}
    initialize@withrevert(...)
{
    lastReverted
}
```

(✓) *onlyAdminCanSetSwapFees*

Only admin can set swap fees

```
{
    initialized == true && initializing == false
    swapFeeBefore = getSwapFee()
}
    < call to any function f >
    swapFeeAfter = getSwapFee()
{
    swapFeeAfter != swapFeeBefore => f == setSwapFee && msg.sender == owner()
}
```

(✓) *onlyAdminCanSetAdminFees*

Only admin can set admin fees

```
{
    initialized == true && initializing == false
    adminFeeBefore = getAdminFee()
}
    < call to any function f >
    adminFeeAfter = getSwapFee()
{
    adminFeeAfter != adminFeeBefore => f == setAdminFee && msg.sender == owner()
}
```

(✓) *pausedMeansLPMonotonicallyDecreases*

When paused, total LP's total supply must not increase.

```
{
    totalSupplyBefore = swapStorage.lpToken.totalSupply()
}
    < call to any function f >
    totalSupplyAfter = swapStorage.lpToken.totalSupply()
{
    paused() => totalSupplyAfter == totalSupplyBefore
}
```

(✓) *swapAlwaysBeforeDeadline*

A swap must never be executed after the given deadline.

```
{
    initialized == true && initializing == false
}
    swap(tokenIndexFrom, tokenIndexTo, dx, minDy, deadline)
{
    block.timestamp <= deadline
}
```

(✓) *addLiquidityCheckMinToMint*

Providing liquidity must always output at least minToMint amount of LP tokens.

```
{
    initialized == true && initializing == false
    requireInvariant underlyingTokensAndLPDifferent()
    balanceBefore = balanceOfLPOfUser(sender)
}
    addLiquidity(e, amounts, minToMint, deadline)
    balance = balanceOfLPOfUser(sender)
{
    balanceAfter >= balanceBefore + minToMint
}
```

(✓) *addLiquidityAlwaysBeforeDeadline*

Adding liquidity must not happen after deadline.

```
{
    initialized == true && initializing == false
}
    addLiquidity(amounts, minToMint, deadline)
{
    block.timestamp <= deadline
}
```

(✓) *removeLiquidityAlwaysBeforeDeadline*

Removing liquidity must not happen after deadline.

```
{
    initialized == true && initializing == false
}
    removeLiquidity(amounts, minToMint, deadline)
{
    block.timestamp <= deadline
}
```

(✓) *swappingCheckMinAmount*

Swapping A for B will always output at least minAmount of tokens B

```
{
    basicAssumptions()
    i != j
    sender = msg.sender
    balanceBefore = balanceOfUnderlyingOfUser(sender, tokenIndexTo)

}
    swap(i, j, dx, minDy, deadline)
    balanceAfter = balanceOfUnderlyingOfUser(sender, tokenIndexTo)

{
    balanceAfter >= balanceBefore + minDy
}
```

(✓) *swappingIndependence*

Swapping token A for token B doesn't change underlying balance of token C

```
{
    initialized == true && initializing == false
    i != j
    j != k
    i != k
    msg.sender != currentContract
    sender = msg.sender
    swapStorage.pooledTokens[i] != swapStorage.pooledTokens[j]
    swapStorage.pooledTokens[j] != swapStorage.pooledTokens[k]
    swapStorage.pooledTokens[i] != swapStorage.pooledTokens[k]
    balanceBefore = balanceOfUnderlyingOfUser(sender, k)
}
    swap(i, j, dx, minDy, deadline)
    balanceBefore = balanceOfUnderlyingOfUser(sender, k)
{
    balanceAfter == balanceBefore
}
```

(✓) *tokenRatioDoesntGoBelowOne*

Ratio between underlying tokens must stay above one.

```
{
    basicAssumptions()
    i != j
    tokenIBalanceBefore = swapStorage.balances[i]
    tokenJBalanceBefore = swapStorage.balances[j]
    tokenIBalanceBefore >= tokenJBalanceBefore
    tokenIBalanceBefore > 0
    tokenJBalanceBefore > 0
    ratioBefore = tokenIBalanceBefore / tokenJBalanceBefore
}
    < call to any function f >
    tokenIBalanceAfter = swapStorage.balances[i]
    tokenJBalanceAfter = swapStorage.balances[j]
    tokenIBalanceAfter >= tokenJBalanceAfter
    tokenIBalanceAfter > 0
    tokenJBalanceAfter > 0
    ratioBefore = tokenIBalanceBefore / tokenJBalanceBefore
{
    ratioBefore >= 1 <=> ratioAfter >= 1
```