CERTORA

Formal Verification of KashiPair

Summary

This document describes the specification and verification of **KashiPair** from SushiSwap using Certora Prover. The work started at an early development stage, and was undertaken from **Feb 8 - Mar 3, 2021**. The latest commit that was reviewed and run through the Certora Prover was **8e8ce65d5d2f4e416ac2089dd0a02733752e9708**. In addition, the flat source code has been verified with the same compiler settings as the deployed code.

The scope of our verification was the KashiPair contract, which allows users to deposit assets as collateral and to borrow other assets against them with flexible oracle and interest rates based on the utilization of the system. Each instance of KashiPair is a pair of collateral and asset tokens. KashiPair uses BentoBox for all token operations (deposit, withdraw, borrow, repay, liquidation).

The Certora Prover proved the implementation of the KashiPair is correct with respect to the formal rules written by the SushiSwap and the Certora teams. During the verification process, the Certora Prover discovered issues in the code listed in the table below. All issues were promptly corrected, and the fixes were verified so as to satisfy the specifications up to the limitations of the Certora Prover. The Certora development team is currently handling these limitations. The next section formally defines high-level specifications.

All the rules are publically available in a public GitHub repository:
https://github.com/sushiswap/kashi-lending/tree/master/spec

Certora Prover verification results:

1. KashiPair source code
2. KashiPair flat file
3. KashiPair simplified (additional properties verified on a simplified version)

Outstanding Issues:

- None

List of Main Issues Prevented By Certora Prover

Severity: Critical **Status: Fixed**

Issue:	Withdrawal of all KashiPair assets
Rules Broken:	No change to other's borrowed asset
Description:	A user can borrow all available asset tokens on behalf of a third party that is not checked for solvency.
Fix:	Users can only borrow for themselves and only when they are in a solvent state.

Severity: Critical **Status: Fixed**

Issue:	Loss of system's assets during liquidation
Rules Broken:	Balance change in liquidation
Description:	The cook function, a function for batch processing, allows a user to invoke KashiPair to perform a liquidation, in which case, the collateral is transferred to the user but no assets are transferred to the system.
Fix:	Disabled calls to KashiPair in the cook function

Severity: High **Status: Fixed**

Issue:	Denial of service in deposit
Rules Broken:	Integrity of add collateral
Description:	Due to the miscalculation of total collateral, when a user skims (adds collateral to the KashiPair and then claims the excess balance) the transaction reverts.
Fix:	Total collateral calculation corrected

Severity: Medium **Status:** Fixed

Issue:	Malicious KashiPair may trick users to lose assets
Rules Broken:	N/A
Description:	A kashiPair initialized with only asset token and no collateral token can be reinitialized with a different asset token, thus causing loss to asset providers.
Fix:	Cannot initialize a KashiPair with zero collateral

Severity: Medium **Status:** Fixed

Issue:	Utilization computation
Rules Broken:	Integrity of interest accrued
Description:	During accrue, the utilization is miscalculated, which might make the utilization more than 100%.
Fix:	Utilization calculation corrected in accrue

Severity: Low **Status:** Fixed

Issue:	Loss of assets and higher joining requirement for asset providers
Rules Broken:	Add then remove asset
Description:	Due to the rounding error, depositing asset tokens can result in zero fractions. Repeating this process results in a state where asset providers wouldn’t want to join KashiPair.
Fix:	Require some minimum assets units in KashiPair

CERTORA

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 any cases not covered by the specification are not checked by the Certora Prover.

We hope that this information is useful, although it provides no warranty of any kind, express 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.

Notations

1. ✅ indicates the rule is formally verified on the latest commit.
2. ✅* indicates that the rule is verified on a simplified version of rebase.
3. We use Hoare triples of the form {p} C {q}, which means that if the execution of program C starts in any state satisfying p, it will end in a state satisfying q. In Solidity, p is similar to require, and q is similar to assert.
4. The syntax {p} ($C_1 \sim C_2$) {q} is a generalization of Hoare rules, called relational properties. {p} is a requirement on the states before C_1 and C_2, and {q} describes the states after their executions. Notice that C_1 and C_2 result in different states. As a special case, $C_1 \sim_{op} C_2$, where op is a getter, indicating that C_1 and C_2 result in states with the same value for op.

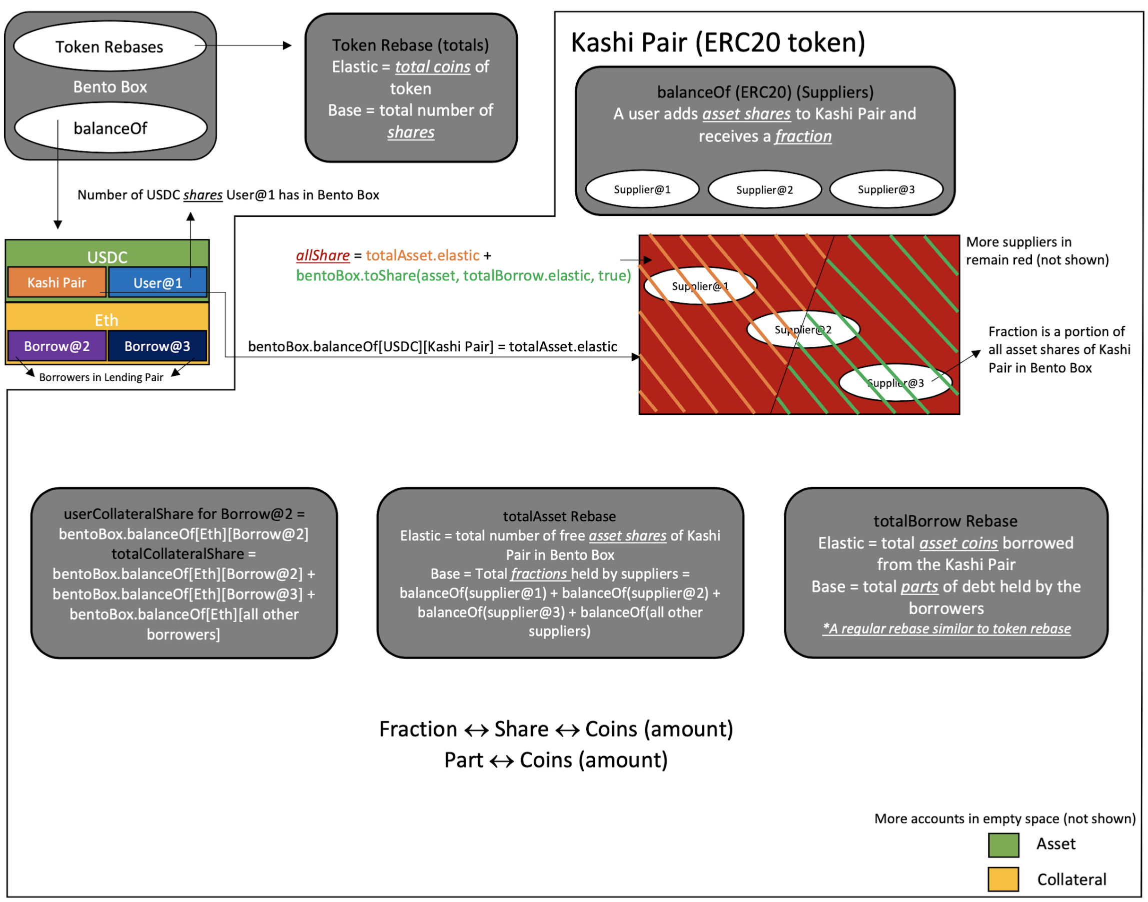
Verification of KashiPair

A KashiPair contract providing loans of **asset token** backed up by a **collateral token** is an ERC20 token with two additional data structures:

1. **Collateral Data Structure:** contains information about the collateral added by users before borrowing
2. **Borrow Data Structure:** contains information about the asset borrowed by users

The standard ERC20 data structure (balanceOf, totalSupply) represents the asset providers' holdings.

System/Data Structures



Functions

1. **bentobox.balanceOf(token t, address user) : uint**
   The amount of token **t** shares a **user** has in the BentoBox.

2. **isSolvent(address user, bool open, uint exchangeRate) : bool**
   Returns true when a **user's** account is in a solvent state according to some **exchangeRate** and the ratio required for **open** solvency or **close** solvency.

3. **feeTo() : address**
   Returns the address that receives the fees

4. **feesEarnedFraction() : uint**
   The current amount of fees to be transferred to feeTo()

5. **accrueInterest() : (uint fullAssetAmount, uint feeAmount, uint utilization)**
   Updates the interest rate and the total borrowed amount, and it returns:
   - fullAssetAmount: equal to the allShare
   - feeAmount: fee amount added
   - utilization: the updated utilization of the system after updation

Properties

Properties safely assume that the msg.sender is not the KashiPair contract itself.

Properties 2 - 7 safely assume all the valid states defined in Property 1 initially.

1. Valid states

A set of invariant properties defining the valid state that the contract can reach

a. Total collateral ✅ (rule: totalCollateralEqUserCollateralSum)

$$\texttt{totalCollateralShare} = \sum_{\texttt{address a}} \texttt{userCollateralShare(a)}$$

b. Total collateral less than or equal to bentoBox balanceOf ✅ (rule: totalCollateralLeBentoBoxBalanceOf)

$$\texttt{totalCollateralShare} \le \texttt{bentobox.balanceOf(collateral, KashiPair)}$$

c. Total asset fraction ✅ (rule: totalSupplyEqUserBalanceOfSum)

$$\texttt{totalAsset.base} = \sum_{\text{address } a} \texttt{balanceOf(a)} + \texttt{feesEarnedFraction()}$$

d. Total borrow part ✅(rule: totalBorrowEqUserBorrowSum)

$$\texttt{totalBorrow.base} = \sum_{\text{address } a} \texttt{userBorrowPart(a)}$$

e. Total asset less than or equal to bentoBox balanceOf ✅ (rule: totalAssetElasticLeBentoBoxBalanceOf)

$$\texttt{totalAsset.elastic} \leq \texttt{bentobox.balanceOf(asset, KashiPair)}$$

f. Validity of total Supply ✅(rule: validityOfTotalSupply)

$$((\texttt{totalBorrow.base} > 0) \Rightarrow (\texttt{totalAsset.base} > 0)) \land$$

$$((\texttt{totalAsset.base} = 0) \Rightarrow (\texttt{totalAsset.elastic} = 0))$$

g. Integrity of zero borrow assets ✅* (rule: integrityOfZeroBorrowAssets)

$$(\texttt{totalBorrow.elastic} >= \texttt{totalBorrow.base}) \land$$

$$((\texttt{totalBorrow.elastic} = 0) \Leftrightarrow (\texttt{totalBorrow.base} = 0))$$

2. **Integrity of accrue function** ✅ (rule: integrityOfAccrueInterest)
The fullAssetAmount must be greater than zero and feeAmount to prevent reverts, and the utilization must be in the valid range (greater than or equal to zero and less than or equal to full utilization).

```
{ totalBorrow.base ≠ 0 }

    (fullAssetAmount, feeAmount, utilization) =

    accrueInterest()

{ fullAssetAmount ≠ 0 ∧ fullAssetAmount > feeAmount ∧

  0 ≤ utilization ≤ FULL_UTILIZATION() }

where FULL_UTILIZATION() is a constant representing 100%.
```

3. No change to other user's holding

a. No change to other's borrowed part ✅
(rule: noChangeToOthersBorrowPart)

```
{ a ≠ u ∧ part = userBorrowPart(a) }

      op_u

{ part ≥ userBorrowPart(a) }

where op_u is any operation performed by u
```

b. No change to other's asset fraction ✅
(rule: noChangeToOthersAssetFraction)

User u, not allowed on behalf of user a, does not change balance of a. User u may increase the balance of the fee address or in case of transferring or depositing to user a

```
{ a ≠ u ∧ v = balanceOf(a) }

      op_u

{ v = balanceOf(a) ∨

  (v ≤ balanceOf(a) ∧ (a = feeTo() ∨ op_u = op_add)) }

Where op_u is any operation performed by u, op_add is one of
addAsset, transferFrom, or transfer
```

c. No change to other's collateral share ✅
(rule: noChangeToOthersCollateralShare)

```
{ a ≠ u ∧ s = userCollateralShare(a) ∧ op_u ≠ liquidation() }

      op_u

{ s = userCollateralShare(a) ∨

  (s ≤ userCollateralShare(a) ∧ op_u = op_add) }
```

4. Inverse Operations

a. RemoveCollateral is inverse of addCollateral ✅
(rule: noChangeToOthersAssetFraction)

```
{ skim = false ∧ c = totalCollateralShare ∧
  u = userCollateralShare(to) }

      addCollateral(to, skim, share);

      removeCollateral(to, share)

{ totalCollateralShare = c ∧ userCollateralShare(to) = u }

where all operations are performed by to
```

b. Repay is inverse of borrow ✅*
(rule: borrowThenRepay)

```
{ skim = false ∧ e = totalBorrow.elastic ∧
  b = totalBorrow.base ∧ p = userBorrowPart(to) }

      borrow(to, amount);

      repay(to, skim, part)

{ totalBorrow.elastic ≤ e ∧ totalBorrow.base = b ∧

  userBorrowPart(to) = p }
```

c. addAsset + removeAsset ✅*

```
{ skim = false ∧ e = totalAsset.elastic ∧ s = totalAsset.base ∧
  b = balanceOf(to) }

      fraction = addAsset(to, skim, share)

      removeAsset(to, fraction)

{ totalAsset.elastic ≥ e ∧ totalAsset.base = s ∧

  balanceOf(to) = b }
```

5. **Solvency**
   a. Open solvency is always clode solvency ✅

      `isSolvent(user, true, rate) ⇒ isSolvent(user, false, rate)`

   b. One cannot change from being in a solvent state to a non-solvent state on the same exchange rate ✅*

      ```
      { isSolvent(user, open, exchangeRate) }

            op_user

      { ¬isSolvent(user, open, exchangeRate) }
      ```

6. **Integrity of liquidation**
   Changes to KashiPair's balance in bentoBox is as expected: collateral balance can only decrease, asset balance can only increase, and collateral balance decreases only if asset balance increases.

   For the rule, we refer bentobox.balanceOf(t, KashiPair) as bentoBalance(token t)

   ```
   { a = bentoBalance(asset) ∧ c = bentoBalance(collateral) }

         liquidation(users, borrowParts, to, swapper, open)

   { bentoBalance(asset) ≥ a ∧ bentoBalance(collateral) ≤ c ∧
     ((bentoBalance(asset) > a) ⇔ (bentoBalance(collateral) < c)) }
   ```

7. **Cook**
   One can not bypass solvency check by using cook

   `cook(op) ∼ op_user`

   equivalent with respect to isSolvent(user, true)

Operations

1. **addCollateral(address to, bool skim, uint share) : N/A**
   Adds **shares** of collateral for user **to**. When **skim** is true, the shares are already in KashiPair's bentoBox account. When **skim** is false, the shares are transferred to KashiPair's bentoBox account.

2. **removeCollateral(address to, uint share) : N/A**
   Removes **share** collateral for user **to**, updates the total collateral, and transfers the shares from KashiPair to the user in the BentoBox.

3. **addAsset(address to, bool skim, uint share) : uint**
   Calculates the fraction for the given **share**, updates the **to's** asset fraction, and transfers the shares from the user to the KashiPair in the BentoBox.

4. **removeAsset(address to, uint fraction) : uint**
   Calculates the share for the given **fraction,** updates the **caller's** asset fraction, and transfers the shares from the KashiPair to the **specified to** in the BentoBox.

5. **borrow(address to, uint amount) : (uint, uint)**
   Borrows an **amount** of assets, calculates the part, and updates the user **to's** borrow part.

6. **repay(address to, bool skim, uint part) : uint**
   Repays **part** of **to's** borrow, possibly by skimming assets.

7. **cook(uint8[] actions, uint[] values, bytes[] datas) : (uint, uint)**
   Performs the set of given actions (inside and possibly outside of the kashiPair)

8. **liquidate(address[] users, uint[] borrowParts, address to, address swapper, bool open) : N/A**
   Liquidates **users** that are in an insolvent state, possible partial liquidation, given a swapper for swapping collateral to assets.