CERTORA

Formal Verification of MasterChef V2

Summary

This document describes the specification and verification of **MasterChefV2** from SushiSwap using Certora Prover. The work was undertaken from **March 15 - 30 March, 2021**. The latest commit that was reviewed and run through the Certora Prover was **414f55587c25b4a761f402d29995aded08806da2**.

The scope of our verification was the MasterChefV2 contract, which rewards users with Sushi tokens for depositing their liquidity pool tokens.

The Certora Prover proved that the implementation of the MasterChefV2 is correct with respect to the formal rules written by the SushiSwap and the Certora teams. The next section formally defines high level specifications.

All the rules are publically available in a public github:
https://github.com/sushiswap/sushiswap/tree/master/spec

Certora Prover verification results:
MasterChefV2 rules
Additional rule (depositThenWithdraw) on simplified version

Disclaimer

Notations

1. ✔️ indicates the rule is formally verified on the latest commit.
2. ✔️* indicates that the rule is verified on a simplified version.
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.

CERTORA

Verification of MasterChefV2

MasterChefV2 allows users to deposit liquidity pool tokens and receive rewards in Sushi. A total of 100 Sushi per block is distributed to all active pools according to allocPoint. **allocPoint** is set for each liquidity pool representing the amount of reward for the pool out of the sum of all allocation points. In addition, for each pool, **accSushiPerShare** represents the accumulated sushi per share for a lpToken since the pool was added and up to the latest updated block (**lastRewardBlock**).

Each user receives a reward that is proportional to the amount of tokens deposited and the duration it was deposited. At any point users can withdraw their assets: liquidity tokens or Sushi tokens. MasterChefV2 is managed by an owner.

Assumptions

The following behavior of the owner is assumed:

1. Owner is responsible for not adding duplicate lpTokens, either by adding or migrating.
2. There is always at least one pool with allocPoint greater than zero.
3. Whenever adding a pool or setting a new allocPoint to an existing pool, **accSushiPerShare** and **lastRewardBlock** are updated for all currently active pools**.**
4. SUSHI token is not added as an lpToken.

Data Structures

1. **lpToken**
   An array that stores the LP tokens corresponding to a pool.

2. **rewarder**
   An array that contains the rewarder corresponding to a pool.

3. **poolInfo**
   An array of structures that contain information about pools. The PoolInfo structure has three main components:
   a. **accSushiPerShare:** Represents the accumulated sushi per share for a lpToken since the pool was added and up to the latest updated block (lastRewardBlock).
   b. **lastRewardBlock:** The last block number when the reward was given out.
   c. **allocPoint:** Representing the amount of reward for the pool out of the sum of all allocation points.

4. **userInfo**
   Given a pool ID and an user's address, userInfo returns a structure that stores the information about the corresponding user. The UserInfo structure has two main components:

   a. **amount:** LP token amount the user has provided.
   b. **rewardDebt:** The amount of SUSHI reward not entitled to the user, either because of joining after the start of the poll, already harvested reward, or withdrawal of LP tokens.

Functions

1. **deposit(uint256 pid, uint256 amount, address to) : N/A**
   Allows a user to deposit LP tokens to a specific pool for SUSHI allocation.

2. **withdraw(uint256 pid, uint256 amount, address to) : N/A**
   Allows a user to withdraw LP tokens from a specific pool.

3. **harvest(uint256 pid, address to) : bool**
   Harvests the SUSHI profit that has been collected for msg.sender and transfers it to the "to" user.

4. **pendingSushi(uint256 pid, address user) : uint256**
   Calculates the SUSHI reward at the current state for a specific user.

Properties

1. Valid states

A set of invariants that define the valid state of the contract at any given point.

a. **Integrity of internal data structures' length** ✔ (rule: IntegrityOfLength)

    poolInfo.length = lpToken.length = rewarder.length

b. **Validity of LP tokens** ✔ (rule: validityOfLpToken)

    userInfo(pid)(u).amount > 0 ⇒ lpToken(pid) ≠ 0

c. **Integrity of total allocation points** ✔ (rule: integrityOfTotalAllocPoint)

$$\text{totalAllocPoint} = \textstyle\sum_{pid} \text{poolInfo(pid).allocPoint}$$

d. **Monotonicity of accSushiPerShare** ✔ (rule: monotonicityOfAccSushiPerShare)

```
{ b = poolInfo(pid).accSushiPerShare }

      op

{ poolInfo(pid).accSushiPerShare ≥ b }
```

e. **Monotonicity of lastRewardBlock** ✔ (rule: monotonicityOfLastRewardBlock)

```
{ b = poolInfo(pid).lastRewardBlock }

      op

{ poolInfo(pid).lastRewardBlock ≥ b }
```

2. No change to other's assets

a. **No change to other's amount** ✔ (rule: noChangeToOtherUsersAmount)

User *u* does not change the balance of user *a*. User *u* may increase the balance of user *a* if they are depositing to user *a*.

```
{ a ≠ u ∧ balance = userInfo(pid)(a).amount }

      op_u

{ userInfo(pid)(a).amount = balance ∨ (userInfo(pid)(a).amount ≥
balance ∧ op_u = deposit(pid, x, a)) }

where op_u is any operation performed by u
```

b. **No change to other's reward debt ✔** (rule: noChangeToOtherUsersRewardDebt)

   User *u* does not change the reward debt of user *a*. User *u* may increase the reward debt of user *a* if they are depositing to user *a*.

```
{ a ≠ u ∧ reward = userInfo(pid)(a).rewardDebt }

      op_u

{ userInfo(pid)(a).rewardDebt = reward ∨

  (userInfo(pid)(a).rewardDebt ≥ reward ∧

  op_u = deposit(pid, x, a)) }

where op_u is any operation performed by u
```

3. **Operation on one pool has no effect on another pool ✔** (rule: noChangeToOtherPool)

```
{ pid ≠ otherPid ∧

  a = poolInfo(otherPid).accSushiPerShare ∧

  b = poolInfo(otherPid).lastRewardBlock ∧

  c = poolInfo(otherPid).allocPoint }

      op_pid

{ poolInfo(otherPid).accSushiPerShare = a ∧

  poolInfo(otherPid).lastRewardBlock = b ∧

  poolInfo(otherPid).allocPoint = c }

where op_pid is any operation on pid except massUpdatePools
```

4. **Preserve total assets of user ✔** (rule: preserveTotalAssetOfUser)
   Assuming SUSHI is not an lpToken and user *u* only deposits, withdraws, or emergency withdraws from and to themselves. If so, user *u's* total balance is preserved. User *u* is also not the MasterChefV2 contract itself.

```
{ totalBalance = lpToken(pid).balanceOf(u) + userInfo(pid)(u).amount }

      op

{ lpToken(pid).balanceOf(u) + userInfo(pid)(u).amount = totalBalance }

where op is any operation
```

5. **Solvency of the system**

   The total asset of the system is the sum of the users' amounts.

   ```
   lpToken(pid).balanceOf(MasterChefV2) = ∑u in user lpToken(pid).balanceOf(u)
   ```

   Proven by showing that:

   a. **Each operation changes at most one user's amount** ✔ (rule: changeToAtmostOneUserAmount)
   b. **System balance change is coherent with a user's amount changes** ✔ (rule: solvency)

6. **Correct computation**

   a. **pendingSushi returns the amount that will be given in harvest** ✔ (rule: sushiGivenInHarvestEqualsPendingSushi)

   ```
   { x = sushi.balanceOf(u) ∧ pending = pendingSushi(pid,user) }
          harvest(pid,user)
   { sush.balanceOf(u) = x + pending } (on the same block)
   ```

7. **Inverse operation: deposit + withdraw** ✔* (rule: depositThenWithdraw)

   Assuming that the user *u* only deposits and withdraws from and to themselves.

   ```
   { b = userInfo(pid)(u).amount ∧ r = userInfo(pid)(u).rewardDebt }

          deposit(pid, amount, to)

          withdraw(pid, amount, to)

   { userInfo(pid)(u).amount = b ∧ userInfo(pid)(u).rewardDebt = r }
   ```

8. **Additivity**

   a. **Deposit is additive for user's amount** ✔ (rule: additivityOfDepositOnAmount)

   Deposit of amount is additive, i.e., it doesn't matter if some amount is deposited in one larger transaction or in two smaller ones.

   ```
   deposit(pid, x, to); deposit(pid, y, to)

   ~ deposit(pid, x + y, to)
   ```

b. **Withdraw is additive for user's amount** ✔ (rule: additivityOfWithdrawOnAmount)

Withdrawal of a user's amount is additive, i.e., it doesn't matter if some amount is withdrawn in one larger transaction or in two smaller ones.

```
withdraw(pid, x, to); withdraw(pid, y, to)

~ withdraw(pid, x + y, to)
```