Formal Verification Of Benqi's Liquid Staking Contracts

Summary

This document describes the specification and verification of BENQI's Liquid Staking system using the Certora Prover. The work was undertaken from March 14, 2022 to April 14, 2022. The Certora team verified the code that was delivered to us on February 23, 2022.

The scope of our verification is the Liquid Staking system, defined in the `StakedAvax` contract. The Certora Prover proved the implementation of Benqi's StakedAvax contract is correct with respect to the formal rules written by the Certora team.

List of Main Issues Discovered

We did not find any major issues in the `StakedAvax` contract. However, it is important to note that addresses with the `ADMIN` role are completely trusted. Admins are able to deposit and withdraw AVAX from the system without limitation, and are able to violate several of the properties verified below. For each property, we have indicated which admin methods can violate the property.

Our manual review of the code did suggest several potential gas optimizations; these are listed in the gas optimization section below.

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.

Notations

✓ indicates the rule is formally verified on the latest reviewed commit. We write ✓* when the rule was verified on a simplified version of the code (or under some assumptions).

✗ indicates the rule was violated under one of the tested versions of the code.

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.

The syntax {p} (C1 ~ C2) {q} is a generalization of Hoare rules, called relational properties. {p} is a requirement on the states before C1 and C2, and {q} describes the states after their executions. Notice that C1 and C2 result in different states. As a special case, C1~op C2, where op is a getter, indicating that C1 and C2 result in states with the same value for op.

Summary of the verification

Overview of StakedAvax protocol (taken from Benqi's README)

Users can stake ("submit") their AVAX by locking it for a minimum duration determined by the `cooldownPeriod` variable. In doing so, they receive a variable amount of sAVAX tokens which are representative of their share of the whole pool. As delegation rewards are accrued to the contract, the amount of pooled AVAX is increased, and thus the exchange rate of sAVAX to AVAX changes to make burning sAVAX more valuable.

To unstake ("redeem") their initial stake, the user must start the cooldown period for the release of their AVAX tokens. This triggers an action by the staking bot to incrementally decrease the amount of delegated AVAX to cover the requested redemption amount after the cooldown period has elapsed. After the cooldown period, the user has a time window, determined by the `redeemPeriod` variable, within which they have to redeem their AVAX by burning the initially set amount of sAVAX tokens. If they fail to do so, the window is closed and the process has to be started again.

Contract Functions

- Standard methods: pause / resume, ERC20 methods
- Unlock requests:
  - requestUnlock: Start unlocking cooldown period for `shareAmount` AVAX.
  - cancelUnlockRequest: Cancel an unexpired unlock request.
  - cancelPendingUnlockRequests: Cancel all unlock requests that are pending the cooldown period to elapse.
  - cancelRedeemableUnlockRequests: Cancel all unlock requests that are redeemable.
  - getters: getUnlockRequestCount, getPaginatedUnlockRequests
- Submitting and redeeming:
  - submit/receive: Process user deposit, mints liquid tokens and increase the pool buffer.
  - redeem: Redeem all redeemable AVAX from all unlocks that are in redemption period.
  - redeemOverdueShares: Redeem all sAVAX held in custody for overdue unlock requests.
  - getSharesByPooledAvax / getPooledAvaxByShares: conversion between stored AVAX and shares.
- Admin functions:
  - accrueRewards: Accrue staking rewards to the pool.
  - withdraw: Withdraw AVAX from the contract for delegation.
  - deposit: Deposit AVAX into the contract without minting sAVAX.
  - pauseMinting / resumeMinting: Pause / resume minting
  - setCooldownPeriod: Update the cooldown period.
  - setRedeemPeriod: Update the redeem period.
  - setTotalPooledAvaxCap: Set a upper limit for the total pooled AVAX amount.

Harness Overview

We've written the following contracts in order to add more getters to `StakedAvax` and make reasoning about AVAX transfering easier for us:

IAvaxReceiver

Since low level calls are assumed to change the contract's state, we defined an interface for receiving AVAX through a function which can be called explicitly. Throughout the specification file, whenever we reason about AVAX balance of a given address it is assumed to implement this interface.

AvaxReceiverA, AvaxReceiverB, AvaxReceiverC

Three different contracts implementing the `IAvaxReceiver` interface.

StakedAvaxHarness

Inherits from `StakedAvax` and implements the `IAvaxReceiver` interface. In this contract we added some getters, replaced the low level calls with calls to `ReceiveAvax` and the `hasRole` calls with a check that msg.sender is `AvaxReceiverC`.

Simplifying Assumptions

- We unroll loops. Violations that require a loop to execute more than once will not be detected.
- One user has all the roles discribed in StakedAvaxStorage.

Properties

1. totalSupplyEqualsSumAllShares ✓

Sum of all shares in the system equals to totalSupply

```
∑ balanceOf(user) = totalSupply
```

2. stakerCountEqualsSumAllShareHolders ✓

stakerCount equals to the number of share holders

```
stakerCount = #{address u | u has positive amount of shares}
```

3. sharesInUnlockRequestAtMostUserCustody ✓

Amount of shares in user's unlock request is at most the number of user's shares in custody

```
∀ address u, ∀ index i: userUnlockRequests[u][i].shareAmount ≤
userSharesInCustody[u]
```

4. sumAllCustodyAtMostPoolBalance ✓*

Amount of all shares in custody is at most contract's shares balance. When calling `transfer`, msg.sender is assumed to be different from the contract itself and when calling `transferFrom`, sender is assumed to be different from the contract itself.

```
∑ userSharesInCustody(user) ≤ balanceOf(StakedAvax contract)
```

5. unlockRequestValidState ✓

For each unlock request must be in only one of the following states: cooldown period, redemption period or expired.

```
∀ unlockRequest req:
    (isWithinCooldownPeriod(req) ∨ isWithinRedemptionPeriod(req) ∨
isExpired(req)) ∧
    ¬(isWithinCooldownPeriod(req) ∧ isWithinRedemptionPeriod(req)) ∧
    ¬(isWithinCooldownPeriod(req) ∧ isExpired(req)) ∧
    ¬(isWithinRedemptionPeriod(req) ∧ isExpired(req))
```

6. sharesAndAvaxWeakCorrelation ✓*

If user's shares change then user's avax changes (weakly) to the opposite direction. If user's avax changes then user's shares change (weakly) to the opposite direction. Not checked deposit/withdraw.

```
{
    user = msg.sender,
    avaxBefore := address(user).balance,
    sharesBefore := balanceOf(user),
    inCustodyBefore := userSharesInCustody(user)
}

<invoke any method f ≠ deposit, withdraw>

{
    sharesBefore < balanceOf(user) ⇒ avaxBefore ≥ address(user).balance,
    sharesBefore > balanceOf(user) ⇒ avaxBefore ≤ address(user).balance,
    avaxBefore > address(user).balance  ⇒ sharesBefore < balanceOf(user),
    avaxBefore < address(user).balance ⇒ inCustodyBefore >
userSharesInCustody(user)
}
```

7. noChangeToOtherShares ✓

Each action affects at most two user's shares.

```
{
    user1 ≠ user2,
    shares1Before = balanceOf(user1),
    shares2Before = balanceOf(user2),
    sharesOtherBefore = balanceOf(other)
}

<invoke any method f>

{
    (shares1Before ≠ balanceOf(user1) ∧
     shares2Before ≠ balanceOf(user2) ∧
     sharesOtherBefore ≠ balanceOf(other)) ⇒ (
        other == user1 v other == user2
    )
}
```

8. unlockRequestStateTransition ✓

state transitions of UnlockRequest with respect to block.timestamp

```
{ t1 < t2 }

{
    isWithinCooldownPeriod(req) at time t1 ⇒ (
        isWithinCooldownPeriod(req) at time t2 v
isWithinRedemptionPeriod(req) at time t2 v isExpired(req) at time t2
    ),

    isWithinRedemptionPeriod(e1, req) at time t1 ⇒ (
        ¬isWithinCooldownPeriod(req) at time t2 ∧
(isWithinRedemptionPeriod(req) at time t2 v isExpired(req) at time t2)
    ),

    isExpired(req) at time t1 ⇒ (
        ¬(isWithinCooldownPeriod(req) at time t2 v
isWithinRedemptionPeriod(req) at time t2) ∧ isExpired(req) at time t2
    )
}
```

9. actionEffectsAtMostOneUser ✓

Each action except transfer/transferFrom can affect at most one user’s shares

```
{
    user1 ≠ user2,

    avax1Before := address(user1).balance,
    avax2Before := address(user2).balance,

    shares1Before := balanceOf(user1),
    shares2Before := balanceOf(user2),

    custody1Before := userSharesInCustody(user1),
    custody2Before := userSharesInCustody(user2)
}

<invoke any method f ≠ transfer, transferFrom>

{
    ((
        avax1Before ≠ address(user1).balance v
        shares1Before ≠ balanceOf(user1) v
        custody1Before ≠ userSharesInCustody(user1)
        ) ∧ (
        avax2Before ≠ address(user2).balance v
        shares2Before ≠ balanceOf(user2) v
        custody2Before ≠ userSharesInCustody(user2)
    )) ⇒ (
        (user1 = msg.sender ∧ user2 = currentContract) v
        (user2 = msg.sender ∧ user1 = currentContract))
}
```

10. unlockRequestIsConstant (first case) ✓

data in UnlockRequest does not change (checked on methods different from cancelUnlockRequest, redeem, redeemOverdueShares)

```
{
    shareAmountBefore := userUnlockRequests[user][requestIdx].shareAmount,
    startedAtBefore := userUnlockRequests[user][requestIdx].startedAt,
    getUnlockRequestCount(user) < max_uint128
}
    <invoke any method f ≠ cancelUnlockRequest, redeem,
redeemOverdueShares>
{
    shareAmountBefore = userUnlockRequests[user][requestIdx].shareAmount ∧
startedAtBefore = userUnlockRequests[user][requestIdx].startedAt
}
```

11. unlockRequestIsConstant (second case) ✓

data in UnlockRequest does not change (checked on cancelUnlockRequest, redeem, redeemOverdueShares). The unlock request that is checked is any request that is not removed by the operation.

```
{
    shareAmountBefore := userUnlockRequests[user][requestIdx].shareAmount,
    startedAtBefore := userUnlockRequests[user][requestIdx].startedAt,
    getUnlockRequestCount(user) < max_uint128
    idx,
    requestIdx ≠ idx,
    isLast := (requestIdx = getUnlockRequestCount(user) − 1 ∧ user =
msg.sender);
}
    <invoke one of cancelUnlockRequest(idx), redeem(idx),
redeemOverdueShares(idx)>
{
    isLast ⇒ shareAmountBefore = userUnlockRequests[user][idx].shareAmount
∧ startedAtBefore = userUnlockRequests[user][idx].startedAt
    ¬isLast ⇒ shareAmountBefore = userUnlockRequests[user]
[requestIdx].shareAmount ∧ startedAtBefore = userUnlockRequests[user]
[requestIdx].startedAt
}
```

12. integrityOfPoolPaused ✔

pool is paused -> The states of the users in the pool should no change except for users with withdraw or deposit role and the contract itself.

```
{
    paused(),
    stakerCountBefore := stakerCount(),
    unlockRequestCountBefore := getUnlockRequestCount(user1),
    totalPooledAvaxBefore := totalPooledAvax(),
    userAvaxBalanceBefore := address(user2).balance,
    userShareBalanceBefore := balanceOf(user3)
}
    <invoke any method f> (allow reverts)
{
    stakerCountBefore = stakerCount(),
    unlockRequestCountBefore = getUnlockRequestCount(user1),
    (totalPooledAvaxBefore ≠ totalPooledAvax()) ⇒ f = accrueRewards,
    userAvaxBalanceBefore ≠ address(user2).balance ⇒ (
        (user2 = currentContract ∨ user2 is admin) ∧
        (f = deposit ∨ f = withdraw)
    ),
    assert userShareBalanceBefore = balanceOf(user3)

}
```

13. cantRequestZeroOrMoreThanShares ✓

calling requestUnlock with amount too high or zero should revert.

```
{
    amountToRequest > balanceOf(msg.sender) v amountToRequest = 0
}
requestUnlock(amountToRequest) (allow reverts)
{
    call reverted
}
```

14. submitIntegrity ✓

User's balance should increase properly after successful submit.

```
{
    totalPooledBefore := totalPooledAvax(),
    avaxAmount := msg.value,
    totalPooledBefore > 0,
    supplyBefore := totalSupply(),
    balanceBefore := balanceOf(msg.sender)
}
submit()
{
    balanceOf(msg.sender) := balanceBefore + (avaxAmount * supplyBefore /
totalPooledBefore)
}
```

15. integrityOfRedeemRedeemableReq ✓*

redeeming user UnlockRequest in redemption period -> the user's shares should not change, user's shares in custody amount should decrease, user's AVAX amount should increase and user's request count should decrease by one. Verified only for requests in the first place of the array.

```
{
    isWithinRedemptionPeriod(userUnlockRequests[msg.sender],[0]),
    userReqCountBefore = getUnlockRequestCount(msg.sender),
    avaxAmountBefore = msg.sender.balance,
    sharesInCustodyBefore = userSharesInCustody(msg.sender),
    sharesBefore = balanceOf(msg.sender)
}
redeem()
{
    avaxAmountBefore < msg.sender.balance,
    userReqCountBefore > getUnlockRequestCount(msg.sender),
    sharesInCustodyBefore > userSharesInCustody(msg.sender),
    balanceOf(msg.sender) = sharesBefore
}
```

16. cantRedeemLockedRequest ✓*

Unlock requests in cooldown period can not be redeemed (checked only on the first request in the user's requests array).

```
{
    msg.sender = user,
    userReqCountBefore := getUnlockRequestCount(user),
    avaxAmountBefore := address(user).balance,
    sharesInCustodyBefore := userSharesInCustody(user),
    sharesBefore := balanceOf(user)
}
redeem()
{
    avaxAmountBefore = address(user).balance,
    userReqCountAfter = getUnlockRequestCount(user),
    sharesInCustodyBefore = userSharesInCustody(user),
    sharesBefore = balanceOf(user)
}
```

17. integrityOfRedeemOverdueShares ✓*

redeeming overdue unlock request -> the user's shares should increase, user's shares in custody amount should decrease, user's AVAX amount should not change and user's request count should decrease by one. (checked only on the first request in the user's requests array)

```
    {
        msg.sender = user,
        isExpired(userUnlockRequests[user][requestIdx]),

        reqShareAmount := userUnlockRequests[user][requestIdx].shareAmount
        userReqCountBefore := getUnlockRequestCount(user),
        avaxAmountBefore := address(user).balance,
        sharesInCustodyBefore := userSharesInCustody(user),
        sharesBefore := balanceOf(user)
    }

    redeemOverdueShares(index);

    {
        avaxAmountBefore = address(user).balance,
        userReqCountBefore > getUnlockRequestCount(user),
        (sharesInCustodyBefore > userSharesInCustody(user)) v
        (sharesInCustodyBefore = userSharesInCustody(user) ∧ reqShareAmount
= 0),
        balanceOf(user) > sharesBefore ∨ (balanceOf(user) = sharesBefore ∧
reqShareAmount = 0)

    }
```

18. integrityOfCancelUnlockRequests ✓

Canceling an unlock request -> the user's shares should increase, user's shares in custody amount should decrease, user's AVAX amount should not change and user's request count should decrease by one.

```
    {
        msg.sender = user,
        userReqCountBefore := getUnlockRequestCount(user),
        userReqCountBefore > index,
        sharesInCustodyBefore := userSharesInCustody(user),
        avaxAmountBefore := address(user).balance,
        sharesBefore := balanceOf(user)
    }

    cancelUnlockRequest(index);

    {

        userReqCountBefore = getUnlockRequestCount(user) + 1,
        sharesInCustodyBefore > userSharesInCustody(user),
        avaxAmountBefore = address(user).balance,
        balanceOf(user) > sharesBefore
    }
```

19. integrityOfAllowance ✓

Integrity of allowance mechanism

```
    {
        msg.sender ≠ user1,

        allowanceBefore := allowance(user1, msg.sender),
        shares1Before := balanceOf(user1),
        shares2Before := balanceOf(user2)
    }

    transferFrom(user1, user2, amount)

    {
        allowanceBefore - amount = allowance(user1, msg.sender),
        shares1Before - amount = balanceOf(user1),
        shares2Before + amount = balanceOf(user2)
    }
```

20. noNewSharesWhenMintingPaused ✔

Minitng new shares is not possible when mintingPaused is true.

```
    {
        mintingPaused() = true,

        totalSupplyBefore := totalSupply(),
    }

    <invoke any method f> (allow reverts)

    {

        totalSupplyBefore >= totalSupply();
    }
```

21. noNewSharesWhenCapReached ✔

Minitng new shares is not possible when the amount of pooled avax reached it's capacity.

```
    {
        totalSupplyBefore := totalSupply(),
        totalPooledAvax() > totalPooledAvaxCap()
    }

    <invoke any method f> (allow reverts)

    {

        totalSupplyBefore >= totalSupply();
    }
```

22. redeemMonotonicityByShareAmount ✔

Redeeming unlock request is monotonic with respect to unlockRequest.shareAmount.

```
    {
        address(user1).balance = address(user2).balance,
        request1 := userUnlockRequests[user1][0],
        request2 := userUnlockRequests[user2][0],
        isWithinRedemptionPeriod(request1),
        isWithinRedemptionPeriod(request2),
        request1.startedAt = request2.startedAt,
        request1.shareAmount < request2.shareAmount
    }

    redeem(0) with msg.sender = user1
    ~
    redeem(0) with msg.sender = user2

    {
        address(user1).balance < address(user2).balance
    }
```

An important assumption underlying our verification is that the contract admins are trusted. Using `withdraw`, admins can remove all of the AVAX from the pool, without restriction.

Gas optimizations

During our review of the contract, we noticed several potential gas optimizations. We have not verified the correctness of these optimizations.

- In `getPaginatedUnlockRequests`, `to.sub(from)` is being calculated multiple times, including inside a loop. It is better to save it in a local variable instead of recomputing it.
- In `getPaginatedUnlockRequests`, `userUnlockRequests[user].length` is being accessed 3 times. Gas could be reduced by storing the length in a local variable.
- In `_cancelUnlockRequest`, `userSharesInCustody[msg.sender] = userSharesInCustody[msg.sender].sub(shareAmount)`, consider caclulating the subtraction operation using regular `-` operator instead of `SafeMathUpgradeable.sub()` because `userSharesInCustody[msg.sender] ≥ shareAmount` so it will never underflow. Same goes for `redeemOverdueShares`, `_redeem`, `redeemOverdueShares(uint unlockIndex)`.
- In `_getExchangeRateByUnlockTimestamp`, `mid.add(1)` is being calculated 3 times inside the loop. Gas could be reduced by storing the result in a local variable.
- In `cancelRedeemableUnlockRequests`, consider saving `userUnlockRequests[msg.sender].length` in a local variable that is decremented when calling `_cancelUnlockRequest(unlockIndex)`. This variable could be used for comparisons with `unlockIndex`, instead of repeatedly accessing storage.

- In `_transferShares`, consider using the regualar subtraction operator `-` instead of `safeMathUpgradable.sub` at `stakerCount = stakerCount.sub(1)`. If the sender had any shares before the transfer we can say that `stakerCount >= 1` (and it is indeed the case because we verify that `amount > 0` and that `shares[sender] = currentSenderShares.sub(shareAmount)` does not revert). The same optimization is possible for for `_burnShares`.