Formal Verification Report of Aave's Stake Token Rev 3

Summary

This document describes the specification and verification of Aave's Staked Token (Aave Safety Module) v3 using the Certora Prover. The work was undertaken from 5th July 2023 to 27th July 2023. The latest commit reviewed and ran through the Certora Prover was 748858a.

The scope of this verification is Aave's Stake token V3 code which includes the following contracts:

- AaveDistributionManager.sol
- StakedAaveV3.sol
- StakedTokenV2.sol
- StakedTokenV3.sol

The Certora Prover proved the implementation is correct with respect to the formal rules written by the Certora team. During the verification process, the Certora Prover discovered bugs in the code listed in the two tables below. The first table contains two issues, that were also present in the live version of the contract. The second table is a list of all issues found in the new version of the contract. All issues were promptly corrected, and the fixes were verified to satisfy the specifications up to the limitations of the Certora Prover. The next section formally defines the high-level specifications of Aave's Stake token V3. All the rules are publicly available in the original 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.

Overview of the Aave Staked Token

The staked token is deployed on Ethereum, with the main utility of participating in the safety module. There are currently two proxy contracts which utilise the staking token - stkAAVE and stkABPT. The new version, the StakedTokenV3, adds enhanced mechanics for slashing in the case of shortfall events. Therefore a new exchange rate is introduced, which will reflect the ratio between stkToken and Token. Initially, this exchange rate will be initialised as 1e18, which reflects a 1:1 peg. In the case of StakedAaveV3, the upgrade adds a hook for managing GHO discounts.

Assumptions and Simplifications Made During Verification

We made the following assumptions during our verification:

- Our verification assumes that the staked and reward tokens are the same. We used this assumption since this is the system's current state (both are AAVE tokens).
- We assume the reward vault is a contract different from the staked token or any other safety module contract.
- We unroll loops. Violations that require executing a loop more than three times will not be detected.
- We excluded the permit function from our verification by assuming it operates in a valid non-deterministic function.

Notations

✔️ By adding this checkmark to a property in the list below we indicate, that the latest commit we reviewed, satisfies this property.

✔️ * indicates that the property was verified on with simplified assumptions described above in "Assumptions and Simplifications Made During Verification".

❌ indicates the property was violated under one of the tested versions of the code.

🔁 indicates that verification of the property is timing out.

Our tool uses 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. This logical structure is reflected in the included formulae for many of the properties below. Note that p and q here can be thought of as analogous to require and assert in Solidity.

The syntax {p} (C1 ~ C2) {q} is a generalisation 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.

Verification Of Aave's Stake Token Rev 3

Properties

1. integrityOfStaking ✓

A successful `stake()` function must move an amount of the staking token from the sender to the contract and must increase the sender's share balance accordingly.

```
{
    balanceStakeTokenDepositorBefore := stake_token.balanceOf(msg.sender),
    balanceStakeTokenVaultBefore := stake_token.balanceOf(currentContract),
    balanceBefore := balanceOf(onBehalfOf)
}
    stake(onBehalfOf, amount)
{
    balanceOf(onBehalfOf) = balanceBefore + amount * currentExchangeRate / EXC
    stake_token.balanceOf(msg.sender) = balanceStakeTokenDepositorBefore - amo
    stake_token.balanceOf(currentContract) = balanceStakeTokenVaultBefore + am
}
```

3. previewStakeEquivalentStake ✓

Preview stake must return the same shares amount as actual staking.

```
{
    amountOfShares := previewStake(amount),
    receiverBalanceBefore := balanceOf(receiver)
}
    stake(receiver, amount)
{
    amountOfShares = previewStake(amount) - receiverBalanceBefore
}
```

4. noEntryUntilSlashingSettled ❌ ✓

Users must not be able to stake until slashing is settled (after the post-slashing period).

```
    stake@withrevert(msg.sender, amount)
{
    inPostSlashingPeriod ⇒ stake function reverted
}
```

5. integrityOfSlashing ✓

A successful slash function call must increase the recipient balance by the slashed amount, decrease the vault's balance by the same amount, and turn on the post-slashing period flag.

```
{
    recipientStakeTokenBalanceBefore := stake_token.balanceOf(recipient),
    vaultStakeTokenBalanceBefore := stake_token.balanceOf(currentContract)
}
    slash(recipient, amountToSlash)
{
    stake_token.balanceOf(recipient) = recipientStakeTokenBalanceBefore + amou
    stake_token.balanceOf(currentContract) = vaultStakeTokenBalanceBefore − am
    inPostSlashingPeriod = True
}
```

6. noSlashingMoreThanMax ✓

Slashing must not exceed the available slashing amount.

```
{
    vaultBalanceBefore := stake_token.balanceOf(currentContract),
    maxSlashable := vaultBalanceBefore * MaxSlashablePercentage / PERCENTAGE_F
}
    slash(recipient, amount)
{
    vaultBalanceBefore − stake_token.balanceOf(currentContract) = maxSlashable
}
```

7. slashingIncreaseExchangeRate ❌ ✓

Slashing must increase the exchange rate.

```
{
    ExchangeRateBefore := getExchangeRate()
}
    slash(args)
```

```
{
    getExchangeRate() ≥ ExchangeRateBefore
}
```

8. slashAndReturnFundsOfZeroDoesntChangeExchangeRate ❌ ✓

Slashing 0 and returning funds of 0 must not affect the exchange rate.

```
{
    ExchangeRateBefore := getExchangeRate()
}
    slash(recipient, 0) || returnFunds(0)
{
    getExchangeRate() = ExchangeRateBefore
}
```

9. integrityOfReturnFunds ✓

A successful `returnFunds()` function call must decrease the sender balance by the returned amount and increase the vault's balance by the same amount.

```
{
    senderStakeTokenBalanceBefore := stake_token.balanceOf(msg.sender),
    vaultStakeTokenBalanceBefore := stake_token.balanceOf(currentContract)
}
    returnFunds(amount)
{
    stake_token.balanceOf(msg.sender) = recipientStakeTokenBalanceBefore − amo
    stake_token.balanceOf(currentContract) = vaultStakeTokenBalanceBefore + am
}
```

10. returnFundsDecreaseExchangeRate ❌ ✓

Returning funds must monotonically decrease the exchange rate.

```
{
    ExchangeRateBefore := getExchangeRate()
}
    returnFunds(args)
{
    getExchangeRate() ≤ ExchangeRateBefore
}
```

11. integrityOfRedeem ✓

A successful redeem function must increase the staked token balance of the recipient by the redeemed amount and must decrease the staked token balance of the contract by the same amount. In addition, the sender's shares balance must decrease by the amount it wanted to redeem.

```
{
    balanceStakeTokenToBefore := stake_token.balanceOf(to),
    balanceStakeTokenVaultBefore := stake_token.balanceOf(currentContract),
    balanceBefore := balanceOf(msg.sender)
}
    redeem(to, amount)
{
    if (amount > balanceBefore) {
        amountToRedeem := balanceBefore * EXCHANGE_RATE_FACTOR / getExchangeRa
    } else {
        amountToRedeem := amount * EXCHANGE_RATE_FACTOR / getExchangeRate();
    }

    stake_token.balanceOf(to) = balanceStakeTokenToBefore + amountToRedeem,
    stake_token.balanceOf(currentContract) = balanceStakeTokenVaultBefore - am
    amount > balanceBefore ⇒ balanceOf(msg.sender) = 0,
    amount ≤ balanceBefore ⇒ balanceOf(msg.sender) = balanceBefore - amount
}
```

12. noRedeemOutOfUnstakeWindow ✓

A successful redeem function call must mean that the user's timestamp is within the unstake window or the safety module is in post-slashing period.

```
{
    cooldown := stakersCooldowns(msg.sender)
}
    redeem(to, amount)
{
    (inPostSlashingPeriod = true) ||
    (block.timestamp > cooldown + getCooldownSeconds() &&
    UNSTAKE_WINDOW ≥ block.timestamp - (cooldown + getCooldownSeconds()))
}
```

13. previewRedeemEquivalentRedeem ✓

`previewRedeem()` must return the same underlying amount as calling redeem.

```
{
    totalUnderlying := previewRedeem(amount),
    receiverBalanceBefore := stake_token.balanceOf(receiver)
}
    redeem(receiver, amount)
```

```
{
    totalUnderlying = stake_token.balanceOf(receiver) - receiverBalanceBefore
}
```

14. exchangeRateNeverZero ❌ -- deprecated and deleted

ExchangeRate must never be zero.

```
{
    ExchangeRateBefore := getExchangeRate()
}
    <invoke any method f>
{
    getExchangeRate() ≠ 0
}
```

15. airdropNotMutualized ✔

Transferring tokens to the contract must not change the exchange rate.

```
{
    exchangeRateBefore := getExchangeRate()
}
    stake_token.transfer(currentContract, amount)
{
    getExchangeRate() = exchangeRateBefore
}
```

16. noStakingPostSlashingPeriod ✔

No user should be able to stake while in the post-slashing period.

```
    stake(onBehalfOf, amount)
{
    inPostSlashingPeriod = true ⇒ function reverts
}
```

17. cooldownCorrectness ❌ -- Rule replaced by cooldownDataCorrectness and cooldownAmountNotGreaterThanBalance.

stakersCooldowns must function correctly.

```
{
    windowBefore := stakersCooldowns(msg.sender) + getCooldownSeconds() + UNST
}
    <invoke any method f>
```

```
{
    windowAfter := stakersCooldowns(msg.sender) + getCooldownSeconds + UNSTAKE

    (stakersCooldowns(msg.sender) + getCooldownSeconds()) ≤ block.timestamp ⇒
    (stakersCooldowns(msg.sender) + getCooldownSeconds()) > block.timestamp ⇒
}
```

18. rewardsGetterEquivalentClaim ✓

Rewards getter must return an amount equal to the max rewards the user deserves (if the user were to withdraw max).

```
{
    deservedRewards := getTotalRewardsBalance(from),
    receiverBalanceBefore := reward_token.balanceOf(receiver)
}
    claimedAmount := claimRewardsOnBehalf(from, receiver, max_uint256)
{
    deservedRewards = claimedAmount,
    reward_token.balanceOf(receiver) = receiverBalanceBefore + claimedAmount
}
```

19. rewardsMonotonicallyIncrease ❌ ✓

Rewards must increase monotonically (except for claim methods).

```
{
    deservedRewardsBefore := getTotalRewardsBalance(user)
}
    <invoke any method f>
{
    deservedRewardsBefore < getTotalRewardsBalance(user) ⇒
        f = claimRewards(address, uint256) ||
        f = claimRewardsOnBehalf(address, address, uint256) ||
        f = claimRewardsAndStake(address, uint256) ||
        f = claimRewardsAndStakeOnBehalf(address, address, uint256) ||
        f = claimRewardsAndRedeem(address, uint256, uint256) ||
        f = claimRewardsAndRedeemOnBehalf(address, address, uint256, uint256)
}
```

20. collectedRewardsMonotonicallyIncrease ❌ removed

21. indexesMonotonicallyIncrease ✓

Global index and personal indexes must increase monotonically.

```
{
    globalIndexBefore := getAssetGlobalIndex(asset),
    personalIndexBefore := getUserPersonalIndex(asset, user)
}
    <invoke any method f>
{
    getAssetGlobalIndex(asset) ≥ globalIndexBefore,
    getUserPersonalIndex(asset, user) ≥ personalIndexBefore
}
```

22. PersonalIndexLessOrEqualGlobalIndex ✓

The personal index of a user on a specific asset must at most be equal to the global index of the same asset. The user's index is derived from the global index and, therefore, must not exceed it.

```
for all asset, user : getUserPersonalIndex(asset, user) ≤ getAssetGlobalIndex(
```

23. totalSupplyGreaterThanUserBalance ✓

The total supply amount of shares must be greater than or equal to any user's share balance.

```
for all user : totalSupply() ≥  balanceOf(user)
```

24. allSharesAreBacked[1] ✗ ✓

Solvency rule - all shares must be backed. previewRedeem() of all shares must be less than or equal to the balance of the staked-token.

```
previewRedeem(totalSupply()) ≤ stake_token.balanceOf(currentContract)
```

25. totalSupplyDoesNotDropToZero ✓

For non-redeem functions (excluding `redeem`, `redeemOnBehalf`, `claimRewardsAndRedeem`, `claimRewardsAndRedeemOnBehalf`), once the `totalSupply` is positive, it can never become zero again.

```
{
    supplyBefore := totalSupply()
}
    <invoke any non-redeem method>
{
    supplyAfter := totalSupply()
    supplyBefore > 0 ⇒ supplyAfter > 0
}
```

26. rewardsIncreaseForNonClaimFunctions ✓

Rewards monotonically increasing for non claim functions (excluding `claimRewards`, `claimRewardsOnBehalf`, `claimRewardsAndStake`, `claimRewardsAndStakeOnBehalf`, `claimRewardsAndRedeem`, `claimRewardsAndRedeemOnBehalf`).

```
{
    deservedRewardsBefore := getTotalRewardsBalance(user)
}
    <invoke any non-claim method>
{
    deservedRewardsAfter := getTotalRewardsBalance(user)
    deservedRewardsAfter ≥ deservedRewardsBefore
}
```

27. balanceOfZero ✓

The balance of address zero is 0.

```
balanceOf(0) = 0
```

28. cooldownDataCorrectness ✓

When is cooldown amount of user non-zero, the cooldown had to be triggered, thus cooldown timestamp is non-zero.

```
cooldownAmount(user) > 0 => cooldownTimestamp(user) > 0
```

29. cooldownAmountNotGreaterThanBalance ✓

User cannot have greater cooldown amount than is their balance.

```
balanceOf(user) >= cooldownAmount(user)
```

Additional Properties for voting and delegation

30. invariant delegateCorrectness ✓

User's delegation flag is switched on iff user is delegating to an address other than his own own or 0.

31. invariant sumOfVBalancesCorrectness ✓

Sum of delegated voting balances and undelegated balances is equal to total supply.

32. invariant sumOfPBalancesCorrectness ✓

Sum of delegated proposition balances and undelegated balances is equal to total supply.

33. transferDoesntChangeDelegationMode ✓

Transfers don't change voting delegation state.

34. noFeeOnTransferFrom ✓

Verify that there is no fee on `transferFrom()`.

35. noFeeOnTransfer ✓

Verify that there is no fee on `transfer()`.

36. userIndex_LEQ_index ✓

The users index at last update for a given asset cannot be greater than the system's index for the same asset.

37. transferCorrect ✓

Token transfer works correctly. Balances are updated if not reverted.

38. transferFromCorrect ✓

Test that transferFrom works correctly: - Balances are updated if not reverted. - If reverted, it means the transfer amount was too high, or the recipient is 0.

39. ZeroAddressNoBalance ✓

Balance of address 0 is always 0.

40. NoChangeTotalSupply ✓

Contract calls don't change token total supply.

41. ChangingAllowance ✓

Allowance changes correctly as a result of calls to `approve`, `transfer`, `increaseAllowance`, `decreaseAllowance`.

42. TransferSumOfFromAndToBalancesStaySame ✓

Transfer from a to b doesn't change the sum of their balances.

43. TransferFromSumOfFromAndToBalancesStaySame ✓

Transfer using `transferFrom()` from a to b doesn't change the sum of their balances.

44. TransferDoesntChangeOtherBalance ✓

Transfer from `msg.sender` to alice doesn't change the balance of other addresses.

45. TransferFromDoesntChangeOtherBalance ✓

Transfer from alice to bob using transferFrom doesn't change the balance of other addresses.

46. OtherBalanceOnlyGoesUp ✓

Balance of an address, who is not a sender or a recipient in transfer functions, doesn't decrease as a result of contract calls.

47. powerWhenNotDelegating ✓

If an account is not receiving delegation of power (one type) from anybody, and that account is not delegating that power to anybody, the power of that account must be equal to its token balance.

48. vpTransferWhenBothNotDelegating ✓

Verify correct voting power on token transfers, when both accounts are not delegating.

49. ppTransferWhenBothNotDelegating ✓

Verify correct proposition power on token transfers, when both accounts are not delegating.

50. vpDelegateWhenBothNotDelegating ✓

Verify correct voting power after Alice delegates to Bob, when both accounts were not delegating.

51. ppDelegateWhenBothNotDelegating ✓

Verify correct proposition power after Alice delegates to Bob, when both accounts were not delegating.

52. vpTransferWhenOnlyOneIsDelegating ✓

Verify correct voting power after a token transfer from Alice to Bob, when Alice was delegating and Bob wasn't.

53. ppTransferWhenOnlyOneIsDelegating ✓

Verify correct proposition power after a token transfer from Alice to Bob, when Alice was delegating and Bob wasn't.

54. vpStopDelegatingWhenOnlyOneIsDelegating ✓

Verify correct voting power after Alice stops delegating, when Alice was delegating and Bob wasn't.

55. ppStopDelegatingWhenOnlyOneIsDelegating ✓

Verify correct proposition power after Alice stops delegating, when Alice was delegating and Bob wasn't.

56. vpChangeDelegateWhenOnlyOneIsDelegating ✓

Verify correct voting power after Alice delegates.

57. ppChangeDelegateWhenOnlyOneIsDelegating ✓

Verify correct proposition power after Alice delegates.

58. vpOnlyAccount2IsDelegating ✓

Verify correct voting power after Alice transfers to Bob, when only Bob was delegating.

59. ppOnlyAccount2IsDelegating ✓

Verify correct proposition power after Alice transfers to Bob, when only Bob was delegating.

60. vpTransferWhenBothAreDelegating ✓

Verify correct voting power after Alice transfers to Bob, when both Alice and Bob were delegating.

61. ppTransferWhenBothAreDelegating ✓

Verify correct proposition power after Alice transfers to Bob, when both Alice and Bob were delegating.

62. votingDelegateChanges ✓

Verify that an account's delegate changes only as a result of a call to the delegation functions.

63. votingPowerChanges ✓

Verify that an account's voting and proposition power changes only as a result of a call to the `delegation`, `transfer`, `slash`, `returnFunds`, and `redeem` / `stake` methods.

64. delegationTypeIndependence ✓

Verify that only `delegate()` and `metaDelegate()` may change both voting and proposition delegates of an account at once.

65. cantDelegateTwice ✓

Verifies that delegating twice to the same delegatee changes the delegatee's voting power only once.

66. transferAndTransferFromPowerEquivalence ✓

transfer and `transferFrom` change voting/proposition power identically.

67. permitIntegrity ✓

Successful permit function increases the nonce of owner by 1 and also changes the allowance of owner to spender.

68. addressZeroNoPower ✓

Address 0 has no voting or proposition power.

69. metaDelegateByTypeOnlyCallableWithProperlySignedArguments ✓

Verify that `metaDelegateByType` can only be called with a signed request.

70. metaDelegateNonRepeatable ✓

Verify that it's impossible to use the same arguments to call `metaDalegate` twice.

71. delegatingToAnotherUserRemovesPowerFromOldDelegatee ✓

Power of the previous delegate is removed when the delegatee delegates to another delegate.

72. powerChanges ✓

Voting and proposition power change only as a result of specific functions.

73. delegateIndependence ✓

Changing a delegate of one type doesn't influence the delegate of the other type.

74. allowanceStateChange ✓

Allowance only changes as a result of specific subset of functions.

---

1. Due to computational difficulties, we assumed that `_getRewards(uint256, uint256, uint256)` and `_getAssetIndex(uint256, uint256, uint128, uint256)` return non-deterministic values. This should not effect correctness of this invariant since the invariant is agnostic to the rewards accounting in the system. ↩