certora

Security Assessment & Formal Verification Report

BORED GHOSTS DEVELOPING aave

StataTokenV2

September 2024

Prepared for

BGD Labs on Aave

Table of content

Project Summary

Project Scope

Project Name	Repository (link)	Latest Commit Hash	Platform
StataToken V2	https://github.com/aave-dao/aave-v3-origin	a59b175	EVM/Solidity 0.8

Project Overview

This document describes the specification and verification of StataTokenV2 using the Certora Prover and manual code review findings. The work was undertaken in September 2024.

The following contract list is included in our scope:

- **StataTokenV2.sol**
- **ERC4626StataTokenUpgradeable.sol**
- **ERC20AaveLMUpgradeable.sol**
- **StataTokenFactory.sol**

The Certora Prover demonstrated that the implementation of the Solidity contracts above is correct with respect to the formal rules written by the Certora team. In addition, the team performed a manual audit of all the Solidity contracts**.** During the verification process and the manual audit, the Certora team discovered bugs in the Solidity contracts code, as listed below.

Findings Summary

The table below summarizes the findings of the review, including type and severity details.

Severity	Discovered	Confirmed	Fixed
Critical	-	-	-
High	-	-	-
Medium	-	-	-
Low	1	1	1
Informational	-	-	-
Total	1	1	1

Severity Matrix

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
		Likelihood		

Detailed Findings

ID	Title	Severity	Status
L-01	`totalAssets()` returns the wrong result by querying the wrong token	Low	Fixed

Low Severity Issues

L-01 TotalAssets() returns the wrong result by querying the wrong token		
Severity: **Low**	Impact: **Low**	Likelihood: **High**
Files: ERC4626StataTokenUpgradeable.sol	Status: FIxed	

Description: According to EIP4626, `TotalAssets()` should return the total amount of the underlying asset that is “managed” by the Vault. In stataToken’s case, this is the corresponding aToken that’s being wrapped. The implementation, however, returns the total amount of the underlying asset of the aToken itself instead.

Let’s take the token derivatives of the AAVE token as an example—AAVE, aAAVE, and static aAAVE, where AAVE is the underlying asset if aAAVE and aAAVE is the asset managed by static aAAVE.
While the EIP specifies that `totalAssets()` should return the aAAVE amount, the implementation returns the AAVE balance of static aAAVE.

Exploit Scenario: This bug is not exploitable from within the contract or the Aave ecosystem since it’s not being used anywhere. However, reporting the wrong amounts of managed assets may hurt future internal or 3rd party integrations.

Customer’s response: The function was overridden to return the aToken balance to comply with the EIP.

Formal Verification

Assumptions and Simplifications

Project General Assumptions

- Loop unrolling: We assume any loop can have at most 1 iteration, except for specific multireward properties.
- View functions filtering: Rules checking state changes of all available functions do not check view functions.

Verification Notations

✅Indicates the rule is formally verified.
❌Indicates the rule is violated.

Formal Verification Properties

EIP4626 Properties

previewDeposit

✅ 1. `previewDepositAmountCheck`, `previewDepositSameAsDeposit`
- EIP: `previewDeposit()` MUST return as close to and no more than the exact amount of Vault shares that would be minted in a `deposit()` call in the same transaction.
- Finding: `previewDeposit()` returns the exact amount of shares getting minted by `deposit()`.

✅ 2. `previewDepositIndependentOfAllowanceApprove`
- EIP: `previewDeposit()` MUST NOT account for `maxDeposit()` limit or the allowance of asset tokens.
- Finding: the value returned by `previewDeposit()` is independent of allowance that the contract might have for transferring assets from any user.

previewMint

✅ 3. `previewMintAmountCheck`, `previewMintSameAsMint`
- EIP: `previewMint()` MUST return as close to and no fewer than the exact amount of assets that would be deposited in a mint call in the same transaction.
- Finding: `previewMint()` returns the exact amount of assets being transferred by `mint()`.

✅ 4. `previewMintIndependentOfAllowance`
- EIP: `previewMint()` MUST NOT account for mint limits like those returned from `maxMint()` and should always act as though the mint would be accepted, regardless of whether the user has approved the contract to transfer the specified amount of assets.
- Finding: the value returned by `previewMint()` is independent of allowance that the contract might have for transferring assets from any user.

previewWithdraw

✅ 5. `previewWithdrawAmountCheck`
- EIP: `previewWithdraw()` MUST return as close to, and no fewer, than the exact amount of Vault shares that would be burned in a `withdraw()` call in the same transaction.
- Finding: `previewWithdraw()` returns the exact amount of shares getting burnt by `withdraw()`.

✅ 6. `previewWithdrawIndependentOfBalance`
- EIP: `previewWithdraw()` should always act as though the withdrawal would be accepted, regardless if the user has enough shares, etc.
- Finding: the value returned by `previewWithdraw()` is independent of any level of share balance for any user.

previewRedeem

✅ 7. `previewRedeemAmountCheck`
- EIP: `previewRedeem()` MUST return as close to and no more than the exact amount of assets that would be withdrawn in a redeem call in the same transaction.
- Finding: `previewRedeem()` returns the exact amount of assets being transferred by `redeem()`.

✅ 8. previewRedeemIndependentOfMaxRedeem
- EIP: `previewRedeem()` MUST NOT account for redemption limits like those returned from maxRedeem.
- Finding: the value returned by `previewRedeem()` is independent of any level of `maxRedeem()` for any user.

✅ 9. `previewRedeemIndependentOfBalance`
- EIP: `previewRedeem()` should always act as though the redemption would be accepted, regardless if the user has enough shares, etc.
- Finding: the value returned by `previewRedeem()` is independent of any level of share balance for any user.

deposit

✅ 10. `depositCheckIndexERayAssert`, `depositCheckIndexERayAssert`, `depositWithPermitCheckIndexERayAssert`
- EIP: `deposit()` MUST revert if all of assets cannot be deposited (due to deposit limit being reached, slippage, the user not approving enough underlying tokens to the Vault contract, etc).
- Finding: `deposit()` could transfer from the user's wallet an amount worth up to 1 AToken more than the specified deposit amount.

✅ 11. `depositCheckIndexGRayAssert`, `depositCheckIndexGRayAssert`, `depositWithPermitCheckIndexGRayAssert`
- EIP: `deposit()` MUST revert if all of assets cannot be deposited (due to deposit limit being reached, slippage, the user not approving enough underlying tokens to the Vault contract, etc).
- Finding: `deposit()` could transfer from the user's wallet an amount worth up to 1 AToken more than the specified deposit amount.

mint

✅ 12. `mintCheckIndex`
- EIP: `mint()` MUST revert if all of shares cannot be minted.
- Finding: `mint()` doesn't always mint the exact number of shares specified in the function call due to rounding.
    * It mints upto 1 extra share over the amount specified by the caller.
    * It mints at least the amount of shares specified by the caller.

withdraw

✅ 13. `withdrawCheck`
- EIP:
    * `withdraw()` should check `msg.sender` can spend owner funds, assets needs to be converted to shares and shares should be checked for allowance.
    * `withdraw()` must revert if all of assets cannot be withdrawn (due to withdrawal limit being reached, slippage, the owner not having enough shares, etc).

- Finding:
   * `withdraw()` makes sure take to check `msg.sender`'s allowance on the owner's shares.
   * For any asset amount worth less than 1/2 AToken, the function will not withdraw anything and will not revert.

redeem

✅ 14. `redeemCheck`, `redeemATokenCheck`
- EIP:
   * `redeem()` should check `msg.sender` can spend owner funds using allowance.
   * `redeem()` MUST revert if all of shares cannot be redeemed (due to withdrawal limit being reached, slippage, the owner not having enough shares, etc).
- Finding:
   * `redeem()` makes sure take to check `msg.sender`'s allowance on the owner's shares.
   * the amount of shares burned by `redeem()` are exactly the specified share amount.

convertToAssets

✅ 15. convertToAssetsCheck, amountConversionRoundedDown
- EIP:
   * `covertToAssets()` MUST NOT show any variations depending on the caller.
   * `covertToAssets()` MUST round down towards 0.
- Finding:
   * `convertToAssets()` returns the same amount of assets for the given number of shares regardless of the caller's identity.
   * calculation are always rounding down.

✅ 16. `toAssetsDoesNotRevert`[1]
- EIP: `convertToAssets()` MUST NOT revert unless due to integer overflow caused by an unreasonably large input.
- Finding: `convertToAssets()` does not revert.

[1] We assume that `rate() < 10^32` and that `shares < 10^45`

> Note: We define a large input as `10^45`. Since `2^256~=10^77`, if we assume that `rate < 10^32 ~= 100,000 RAY` we get the requirement that `shares < 10^45`, which implies `shares * rate < 2^256`.

convertToShares

✅ 17. `convertToSharesCheck`, `sharesConversionRoundedDown`
- EIP:
  * `convertToShares()` MUST NOT show any variations depending on the caller.
  * `convertToShares()` MUST round down towards 0.
- Finding:
  * `convertToShares()` returns the same amount for shares for the given number of assets regardless of the caller identity.
  * calculation will always round down.

✅ 18 `toSharesDoesNotRevert`[2]
- EIP: `convertToShares()` MUST NOT revert unless due to integer overflow caused by an unreasonably large input.
- Finding: `convertToShares()` does not revert.

> Note: We define a large input as `10^50`. Since `2^256~=10^77` and `RAY=10^27` we get the requirement that `assets < 10^50`, which implies `RAY * assets < 2^256`.

maxWithdraw

✅ 19. `maxWithdrawConversionCompliance`
- EIP:
  * `maxWithdraw()` MUST return the maximum amount of assets that could be transferred from owner through withdraw and not cause a revert.
  * the returned value MUST NOT be higher than the actual maximum that would be accepted (it should underestimate if necessary).
- Finding: `maxWithdraw()` returns an amount that is greater or equal to the amount withdrawn by the same value of `assets`.

[2] We assume that `assets < 10^50`

✅ 20. `maxWithdrawMustntRevert`
- EIP: `maxWithdraw()` MUST NOT revert.
- Finding: `maxWithdraw()` returns `_convertToAssets(maxRedeem(user))` which can technically revert due to overflow when using `mulDiv`, however, taking into account the realistic range of arguments, the function will not revert.

maxRedeem

✅ 21. `maxRedeemCompliance`
- EIP:
  * `maxRedeem()` MUST return the maximum amount of shares that could be transferred from owner through redeem and not cause a revert.
  * the returned value MUST NOT be higher than the actual maximum that would be accepted (it should underestimate if necessary).
- Finding: `maxRedeem()` returns an amount that is greater or equal to the amount redeemed by the same value of `shares`.

✅ 22. `maxRedeemMustntRevert`
- EIP: `maxRedeem()` MUST NOT revert.
- Finding: `maxRedeem()` uses `convertToShares(virtualUnderlyingBalance)` to compute the available underlying tokens that can be withdrawn from the pool. While this call can technically revert due to overflow when using mulDiv, taking into account the realistic range of arguments, the function will not revert.

maxDeposit

✅ 23. `maxDepositMustntRevert`
- EIP: `maxDeposit()` MUST NOT revert.
- Finding: maxDeposit() can technically revert on multiple occasions due to overflow in computations, however, taking into account the realistic range of arguments, the function will not revert.

maxMint

✅ 24. `maxMintMustntRevert`
- EIP: `maxMint()` MUST NOT revert.
- Finding: `maxMint()` calls both `maxDeposit()` and `convertToShares()` which can technically revert, however, taking into account the realistic range of arguments, the function will not revert.

totalAssets

✅ 25. `totalAssetsMustntRevert`
- EIP: `totalAssets()` MUST NOT revert.
- Finding: `totalAssets()` returns `_convertToAssets(totalSupply())` which can technically revert due to overflow when using mulDiv, however, taking into account the realistic range of arguments, the function will not revert.

Extending EIP4626 Properties

Rounding Range Properties

✅ 26. `previewRedeemRoundingRange`
`previewRedeem` MUST tightly round down assets

The upper bound (i.e. `previewRedeem <= convertToAssets`) follows from ERC4626.
The lower bound (`previewRedeem + 1 + rate / RAY >= assets`) is based on the current implementation. This lower bound shows that `previewRedeem` is derived by `convertToAssets`.

✅ 27. `previewWithdrawRoundingRange`
`previewWithdraw` MUST tightly round up shares

The lower bound (i.e. `previewWithdraw >= convertToShares`) follows from ERC4626.
The upper bound is based on the current implementation.

✅ 28. `amountConversionPreserved`
Conversion of amount to shares and back MUST round the value down no more than `1 + rate() / RAY()`

✅ 29. `sharesConversionPreserved`
Conversion of shares to amount and back MUST round the value down no more than `1 + rate() / RAY()`

✅ 30. `accountsJoiningSplittingIsLimited`
Joining shares MUST provide limited advantage over splitting shares when converting to asset amounts. The account can gain up to 1 extra asset due to rounding error.

✅ 31. `convertSumOfAssetsPreserved`
Joining assets MUST provide a limited advantage over splitting assets when converting to shares amount. The joint account can gain up to 1 extra share due to rounding error.

✅ 32. `redeemSum`, `redeemATokensSum`
Loss of assets due to redeeming shares in multiple steps MUST NOT exceed a total of 1 asset compared to redeeming the same amount of shares in one step. This occurs due to rounding errors.

maxDeposit

✅ 33. maxDepositConstant
`maxDeposit()` MUST NOT change due to any action of a user.

aToken Properties

✅ 34. `aTokenBalanceIsFixed`[3][4][5]

[3] We assume `msg.sender` is not the aToken (`sender != asset()`)
[4] We assume `msg.sender` is not a reward token
[5] We assume `msg.sender` is not the `StataToken`

aToken balance of `msg.sender` MUST NOT change due to the invocation of the following functions:
- `collectAndUpdateRewards`
- `claimRewardsOnBehalf` -- under additional assumption that both `onBehalfOf` and `receiver` are not the StataToken, nor the reward token, nor aToken
- `claimRewardsToSelf`
- `claimRewards` -- under additional assumption that `receiver` is not the StataToken, nor the reward token, nor aToken
- `refreshRewardTokens`

Here aToken means the return value of the `asset()` method.

Rewards Related Properties

Single Rewards Properties

✅ 35. rewardsConsistencyWhenSufficientRewardsExist[5]
If the `StataToken` has enough rewards to pay the claiming user:
- The rewards balance of the receiver MUST NOT decrease.
- The claimable rewards a user is eligible for MUST be what they received.
- After claim, the unclaimed reward of the user MUST be nullified.
- The user's deserved rewards MUST NOT disappear from the contract's accounting system.

✅ 36. `rewardsConsistencyWhenInsufficientRewards`[5][6]

If the `stataToken` doesn't have enough rewards to pay the claiming user:
- The rewards balance of the receiver MUST NOT decrease.
- The user's deserved rewards MUST NOT disappear from the contract's accounting system.

✅ 37. rewardsTotalDeclinesOnlyByClaim[5]
`getTotalClaimableRewards` MUST ONLY decrease due to a call to one of the claiming functions.

[6] We assume that `msg.sender` is not the `TransferStrategy`

To avoid timeouts this was split into three parametric rules, each containing some of the functions.

***Note:** With the exception of `initialize()`.

✅ 38. `singleAssetAccruedRewards`
In a case where the token has a single reward, the total accrued value returned by `getUserAccruedRewards()` MUST be the reward accrued value.
`(RewardsDistributor._assetsList.length == 1 && RewardsDistributor._assetsList[0] == asset) => (RewardsDistributor.getUserAccruedRewards(reward, user) == RewardsDistributor._assets[asset].rewards[reward].usersData[user].accrued)`

✅ 39. `totalAssets_stable`
Claiming rewards MUST NOT change `totalAssets()`.

✅ 40. `totalClaimableRewards_stable`
At a given block, `getTotalClaimableRewards()` MUST NOT change unless rewards are claimed.

✅ 41. getClaimableRewards_stable

At a given block, `getClaimableRewards()` MUST NOT change unless rewards were claimed.

✅ 42. `getClaimableRewardsBefore_leq_claimed_claimRewardsOnBehalf`
The number of rewards that were received by a user after calling `claimRewards()` MUST NOT exceed the result returned by `getClaimableRewards()` right before the claim.

Multi Rewards Properties

✅ 43. `prevent_duplicate_reward_claiming_single_reward_sufficient`[5]
Claiming the same reward twice (assuming the contract has sufficient rewards to give) MUST NOT yield more than claiming it once.

✅ 44. prevent_duplicate_reward_claiming_single_reward_insufficient
Claiming the same reward twice (assuming the contract does ***not*** have sufficient rewards to give) MUST NOT yield more than claiming it once.

Solvency

✅ 45. solvency_total_asset_geq_total_supply
Total assets MUST be greater than or equal to the total supply.

✅ 46. solvency_positive_total_supply_only_if_positive_asset
Total supply MUST be non-zero only if total assets is non-zero.

Disclaimer

The Certora Prover takes a contract and a specification as input and formally proves that the contract satisfies the specification in all scenarios. Notably, 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.

Even though we hope this information is helpful, we 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.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.