Formal Verification Report For Static AToken V3

Summary

This document describes the specification and verification of Aave's static aToken V3 using the Certora Prover. The work was undertaken from January 18th to March 31st, 2023. The latest commit reviewed and run through the Certora Prover was 51d46b1.

The scope of our verification includes one main contract:

- StaticATokenLM.sol

With one library:

- RayMathExplicitRounding.sol

The Certora Prover proved the implementation is correct with respect to the formal specification written by the Certora team. During the verification process, the Certora Prover discovered bugs in the code listed in the table below. All issues were promptly corrected, and the fixes were verified to satisfy the specifications up to the limitations of the Certora Prover. The following section formally defines high-level specifications of the contract at hand. All the rules are publically available in a public github.

List of Main Issues Discovered

Severity: **Medium**

Issue:	**A `staticAToken` user can be owed a large sum of reward by the `staticAToken` if it's not yet registered**
Rules Broken:	Property #44 - `getClaimableRewards_stable`. see violation
Description:	Consider a reward token (`REW`) distributed through the `INCENTIVES_CONTROLLER` to a reward-bearing token that the

Issue:	A `staticAToken` user can be owed a large sum of reward by the `staticAToken` if it's not yet registered
	`StaticAToken` holds. At any time, a `staticAToken` user can invoke any claim method with an array that contains this `REW` token. If this token is not registered in the `StaticATokenLM`'s reward list, the user's `rewardsIndexOnLastInteraction` will be 0 by default. When, later, `getClaimableRewards()` is called, the `rewardsIndexOnLastInteraction` sent to compute the pending rewards will depend on the value of the user's `rewardsIndexOnLastInteraction`. That value is the one which will be sent if it is non-zero; however, if the value is zero, `_startIndex[REW]` will be sent instead. This variable represents a snapshot of the index at the time of registration. However, since the reward isn't registered, the user's index has never been updated, and therefore it is zero. In addition, since the reward isn't registered the `startIndex[REW]` is 0. that means that the pending rewards will be calculated as: `(balance * currentRewardsIndex)/assetUint` This will make the `staticAToken` owe a large amount of rewards to any user who makes a claim before the registration of the asset. In fact, the `staticAToken` may owe one or more of its users a larger amount than the incentive owes to the static Token.
Concrete Example:	1. Take an initialized `staticAToken` that just started to work with a list of 3 reward tokens that deserve to be collected thanks to the distribution of the incentive controller. 2. On initialization, the contract will register those three tokens in the `staticToken` internal list. 3. User1 is depositing for the first time at a time `t`, which updates his user state, i.e. unclaimed, as well as a snapshot of the index at the time of operation. 4. At time `t' > t`, a new token, `REW`, is added to the list of rewards that the `staticAToken` is eligible to claim. 5. Nobody refreshes the reward list in the `StaticAToken`. 6. User1 take any action he desires, including depositing a lot of money. 7. At time `t'' > t'`, after some `REW`s are starting to accumulate in the incentive controller for the `staticToken`'s right, user1 realises that he's eligible for his share of `REW` and tries to claim it through one of the `claim` functions of the `staticAToken`. 8. Since the rewards aren't registered, the user's `rewardsIndexOnLastInteraction` is never updated and stays at the default value of 0. 9. The pending rewards that `staticToken` is now owed to the user is `[user_balance * (currentIndex - 0)]/assetUnit`. This value can be greater than the value actually owed by the incentive controller to the `staticAToken`. If user1 is the only user in the system `staticAToken.balanceOf(user) =`

Issue:	A `staticAToken` user can be owed a large sum of reward by the `staticAToken` if it's not yet registered
	`incentiveController.deservedRewards(asset, reward, staticAToken)` however, the current index of the `staticAToken` on the incentive controller must be greater than 0 because the reward-bearing token (AToken in this case) makes sure to update the code upon transfer (in `handleAction`).
Mitigation/Fix:	Fixed in PR #29

Severity: **Low**

Issue:	Losing one share worth of assets upon deposit
Rules Broken:	Property #14 - `withdrawCheck`. see violation
Description:	Suppose a user calls the `deposit()` function with an asset amount anywhere in the range `[x/2, x+1)` AToken. In that case, the `deposit()` function will round up the amount of Atokens that needs to be transferred from the sender but will round down the number of `staticAToken` shares the receiver gets in return. For example, the user sends x+1 Atokens, but receives only x `staticAtokens`.
Concrete Example:	1. A user calls `deposit()` with an asset amount of 9 underlying tokens. The `fromUnderlying` flag is `false`, so the `AToken.transferFrom()` function is called. 2. `AToken.transferFrom()` function eventually calls the `AToken._transfer()` function. This function converts the user-specified underlying asset amount to the number of ATokens by calling the `rayDiv` function with the current rate and asset amount. 3. Due to round-up, `rayDiv` returns 1 AToken, which is equivalent to 18 underlying tokens. 4. 1 AToken is transferred from the user to the `staticAToken` contract. 5. The `Deposit()` function proceeds to calculate the shares to be minted to the user. `rayDivRoundDown` is called with the rate and the asset amount (9). Due to round-down, `rayDivRoundDown` returns 0. 6. The user gets 0 shares in return for the 1 AToken deposited in the vault.
Mitigation/Fix:	Fixed in PR #25

Severity: **Low**

Issue:	Non-compliance of `deposit()` with the EIP4626 standard
Rules Broken:	Properties #11 & #12 - `depositCheck`. see violation

Issue:	Non-compliance of `deposit()` with the EIP4626 standard
Description:	As per EIP4626, `deposit()` must revert if all the assets cannot be deposited. In the contract, if the user calls `deposit()` with an amount of underlying assets that are less than the equivalent of half an AToken, the function will end up depositing no assets but will inappropriately fail to revert.
Concrete Example:	1. A user calls the deposit function with 547 of underlying asset tokens. The `fromUnderlying` flag is `false`, so the `AToken.transferFrom()` function is called. 2. `AToken.transferFrom()` function eventually calls `AToken._transfer()`. This function converts the user-specified underlying asset amount to the number of `ATokens` by calling `rayDiv` with the current rate and asset amount. 3. `rayDiv` function returns `0` as the asset amount is less than `rate/(2RAY)`. As a result, `0` `ATokens` get transferred from the user to the `staticAToken` contract. 4. The function proceeds with the rest of the execution without reverting.
Mitigation/Fix:	Fixed in PR #25

Severity: **informational**

Issue:	Inconsistency of `getUnclaimedRewards()` return value units
Rules Broken:	Property #38 - `rewardsConsistencyWhenInsufficientRewards`. see violation
Description:	All the reward-related methods - `_getPendingRewards()`, `collectAndUpdateRewards`, `_getClaimableRewards`, etc., compute and return values in the reward token units. However, `getUnclaimedRewards()` incorrectly assumes that the user's `unclaimedReward` amount is stored in `RAY` units and converts it to `WAD`. This can cause external protocols relying on the getter's result to interpret and function in a false state.
Mitigation/Fix:	Fixed in PR #24

Severity: **informational**

Issue:	Non-compliance of `totalAssets()` and `maxWithdraw` with the EIP4626 standard
Rules Broken:	Property #21 & #26 - `totalAssetsMustntRevert`, `maxWithdrawMustntRevert` see violation

Issue:	**Non-compliance of `totalAssets()` and `maxWithdraw` with the EIP4626 standard**
Description:	As per EIP4626, both `maxWithdraw()` and `totalAssets()` mustn't revert by any means. The implementation, however may revert due to overflow when calling `rayMulRoundDown` in the first function and `rayMul` in the second.
Mitigation/Fix:	The only way these methods can revert is when `a*b > type(uint256).max`, where `a` is the `amount` and `b` is the `normalizedIncome`. With the following assumptions in mind: (1) `type(uint256).max ~=10^77`, (2) `normalizedIncome` will always be around `10^27`, even with some margin on the index, the revert case will occur only when `amount > 10^45`. This is an unreasonably large amount of tokens assuming `token decimals <= 18`. All this makes the compliance violation purely theoretical with the currently used tokens.

Severity: **Recommendation**

Issue:	**`_claimRewardsOnBehalf()` stops handling rewards after it encounters address 0**
Recommendation:	The function `_claimRewardsOnBehalf()` iterates over an array of rewards passed to it by the user via one of the external claim functions. Currently, the function short-circuits at the first occurrence of address 0 in the array - it exits the function by executing `return`. Although it is a viable way to handle claims, replacing the `return` with `continue` and being more forgiving to protocols that make mistakes in constructing the array is possible.
Mitigation/Fix:	Fixed in PR #27

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.

Overview of Static AToken V3

Aave's aTokens are "rebasing" assets. Their balance grows over time, representing the amount of underlying assets that can be extracted from the protocol upon withdrawal. Rebasing assets are nicely integrated with specific models of external protocols, e.g. Curve's liquidity pools, but create an implementation overhead for others, e.g. Balancer's bb-pools.

The `staticAToken` is a wrapper to Aave's aToken, which omits the rebasing factor of the aToken, and allows it to integrate with models that better support a static balance rather than a changing balance.

The static aToken will also account for any potential liquidity mining rewards that a parallel aToken holder may have been eligible for.

Assumptions and Simplifications Made During Verification

We made the following assumptions during our verifications:

- We assume that none of the reward tokens distributed for a particular reward-bearing aToken are the aToken itself.
- We assume that a registered reward exists in the Incentives Controller. The reward exists in the mapping `RewardsController._assets.[_atoken].availableRewards` and `RewardsController._assets.[_atoken].availableRewardsCount > 0`. In other words, we assume `RewardsController.configureAssets(RewardsDataTypes.RewardsConfigInput[])` was called.
- We assume that the sum of all balances is equal to `totalSupply()`. In particular, for any user we assume that `balanceOf(user) <= totalSupply()`.
- We unroll loops.
    - In all of the properties that aren't addressing multi-rewards, violations that require a loop to execute more than once will not be detected.
    - In all of the properties that address multi-rewards, violations that require a loop to execute more than twice will not be detected.

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.

✍️ indicates the rule is not yet formally specified.

🔁 indicates the rule is postponed (due to other issues, low priority) .

⏱️ indicates the rule is currently timing out and therefore isn't proved nor finding violations.

Verification of StaticATokenLM

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. previewWithdrawIndependentOfMaxWithdraw

- EIP: `previewWithdraw()` MUST NOT account for withdrawal limits like those returned from maxWithdraw.
- Finding: the value returned by `previewWithdraw()` is independent of any level of `maxWithdraw()` for any user.

✓ 7. 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

✓ 8. 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()`.

✓ 9. 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.

✓ 10. 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

✓ * 11. depositCheckIndexERayAssert [1]

❌ - Detected issue #3.

- 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.

✓ * 12. depositCheckIndexGRayAssert [1]

❌ - Detected issue #3.

- 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

✓ * 13. mintCheckIndex [1]

- 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

✓ 14. withdrawCheck

❌ - Detected issue #2.

- 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

✓ 15. redeemCheck

- 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

✓ 16. convertToAssetsCheck, amountConversionRoundedDown, toAssetsCallerIndependent

- 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 identity.
  - calculation are always rounding down.

✓ 17. toAssetsDoesNotRevert [2]

- EIP: `convertToAssets()` MUST NOT revert unless due to integer overflow caused by an unreasonably large input.

- Finding: `convertToAssets()` does not revert.

> 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

✓ 18. convertToSharesCheck, sharesConversionRoundedDown, toSharesCallerIndependent

- 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.

✓ 19. toSharesDoesNotRevert [3]

- 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

✓ 20. maxWithdrawRedeemCompliance

- 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`.

✗ 21. maxWithdrawMustntRevert

- EIP: `maxWithdraw()` MUST NOT revert.
- Finding: `maxWithdraw()` returns `_convertToAssets(balanceOf(user))` which returns the `rayMulRoundDown` of the scaled balance and the index. This result can

revert when multiplying balance and rate smaller than `RAY`

maxRedeem

✓ 22. maxWithdrawRedeemCompliance

- 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`.

✓ 23. maxRedeemMustntRevert

`maxRedeem()` MUST NOT revert.

maxDeposit

✓ 24. maxDepositMustntRevert

`maxDeposit()` MUST NOT revert.

maxMint

✓ 25. maxMintMustntRevert

`maxMint()` MUST NOT revert.

totalAssets

❌ 26. totalAssetsMustntRevert

- EIP: `totalAssets()` MUST NOT revert.
- Finding: `totalAssets()` returns `aToken. balanceOf(staticAToken)` which returns the `rayMul` of the scaled balance and the index. This result can revert according to the `rayMul` limitations.

Extending EIP4626 Properties

Rounding Range Properties

✓ 27. 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`.

✓ 28. 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.

✓ 29. amountConversionPreserved

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

✓ 30. sharesConversionPreserved

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

✓ 31. 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.

✓ 32. 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.

⏱ 33. withdrawSum

Loss of shares due to withdrawal of assets in multiple steps MUST NOT exceed a total of 1 share compared to withdrawing the same amount of assets in one step. This occurs due to rounding errors.

✓ 34. redeemSum

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

✓ 35. maxDepositConstant

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

aToken Properties

✓ 36. aTokenBalanceIsFixed [4] [5] [6]

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 StaticATokenLM, nor the reward token, nor aToken
- `claimRewardsToSelf`
- `claimRewards` -- under additional assumption that `receiver` is not the StaticATokenLM, nor the reward token, nor aToken
- `refreshRewardTokens`

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

Rewards Related Properties

Single Rewards Properties

✓ 37. rewardsConsistencyWhenSufficientRewardsExist [4]

If the `staticAToken` 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.

✓ 38. rewardsConsistencyWhenInsufficientRewards [4] [7]

❌ - Detected issue #4.

If the `staticAToken` 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.

✓ 39. rewardsTotalDeclinesOnlyByClaim [4]

`getTotalClaimableRewards` MUST ONLY decrease due to a call to one of the claiming functions. To avoid timeouts this was split into three parametric rules, each containing some of the functions.

> Note: With the exception of `initialize()`. Note: `metaDeposit` and `metaWithdraw` were not addressed in this rule.

✓ 40. 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)`

✓ 41. totalAssets_stable

Claiming rewards MUST NOT change `totalAssets()`.

✓ 42. reward_balance_stable_after_collectAndUpdateRewards

At a given block, increasing the contract's AToken balance MUST NOT affect the rewards fetched by `collectAndUpdateRewards()`.

✓ 43. totalClaimableRewards_stable

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

✓ 44. getClaimableRewards_stable

❌ - Detected issue #1.

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

✓ 45. 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

✓ 46. prevent_duplicate_reward_claiming_single_reward_sufficient [4]

Claiming the same reward twice (assuming the contract has sufficient rewards to give) MUST NOT yield more than claiming it once.

✓ 47. prevent_duplicate_reward_claiming_single_reward_insufficient [4]

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

Solvency

✓ 48. solvency_total_asset_geq_total_supply

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

✓ 49. solvency_positive_total_supply_only_if_positive_asset

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

---

1. We assume the index (rate) is greater or equal to 1 `RAY` ↩️ ↩️[2] ↩️[3]
2. We assume that `rate() < 10^32` and that `shares < 10^45` ↩️
3. We assume that `rate() > 0` and that `assets < 10^50` ↩️
4. We assume `msg.sender` is not the `StaticATokenLM` ↩️ ↩️[2] ↩️[3] ↩️[4] ↩️[5] ↩️[6]
5. We assume `msg.sender` is not the aToken (`sender != asset()`) ↩️
6. We assume `msg.sender` is not a reward token ↩️
7. We assume that `msg.sender` is not the `TransferStrategy` ↩️