CERTORA

Formal Verification Report For Aave's GHO Token

Summary

This document describes the specification and verification of Aave's GHO Token using the Certora Prover. The work was undertaken from the 30th of January 2023 to the 28th of February 2023. The latest commit that was reviewed and ran through the Certora Prover was e6fabb9.

The scope of our verification includes the following contracts:

- GhoToken.sol
- GhoDiscountRateStrategy.sol
- GhoInterestRateStrategy.sol
- GhoAToken.sol
- GhoVariableDebtToken.sol
- GhoFlashMinter.sol
- GhoOracle.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 next section formally defines high level specifications of Aave's GHO Token. All the rules are publically available in a public github.

List of Main Issues Discovered

Severity: Low

Issue:	Bypassing rebalance timelock
Rules Broken:	Property #3 onlyRebalanceCanUpdateUserDiscountRate (CVT output)
Description:	In `GhoVariableDebtToken.sol`, the `rebalanceUserDiscountPercent` function checks that a user's rebalance lock period (`rebalanceTimestamp`) is over before it allows rebalancing of the discount percentage. However, `_burnScaled` and `_mintScaled`, `stkAAVE.transfer` (triggered by `burn`, `mint` and `updateDiscountDistribution` respectively) are doing the same functionality of updating the discount percentage without checking whether the lock period is over.
Mitigation/Fix:	The `rebalance` timelock was deprecated (81bb51b)

Severity: Low

Issue:	Non-compliance of **`maxFlashLoan`** in the FlashMinter facilitator to the EIP3156 standard
Description:	EIP3156 states that the function `maxFlashLoan` must return the maximum loan possible for the token, and return 0 instead of reverting if the token is not currently supported. The `GhoFlashMinter` implmentation, however, may revert if `bucketLevel > bucketCapacity`. This can happen if the bucket's capacity gets reduced below the bucket's level.
Mitigation/Fix:	Fixed on commit 038442d.

Severity: Low

Issue:	Accumulated interest can be manipulated by the user
Description:	Calling `rebalanceUserDiscountPerecent` calculates the accumulated interest since the last operation made by the user. As part of this operation, the scaled discount given to the user is burned. This call decreases the total interest accumulated for this debt compared to the case of the exact same position taken without calling rebalance.
Concrete Example:	Consider a user with `scaledBalance = 100`, 50% discount rate and 0 accumulated interest at time `t0`. The global index is 1, 2, and 4 at times `t0`, `t1`, and `t2` respectively. In the first scenario the user calls `rebalanceUserDiscountPercent` at `t1`

Issue:	Accumulated interest can be manipulated by the user
	which updates the scaled balance to 75 and the accumulated interest to 50. At `t2` the user does the same call which updates the accumulated interest to 125. If instead, the user does only a single call to `rebalanceUserDiscountPerecent` at `t2`, the accumulated interest balance would reach a total of 150.
Mitigation/Fix:	The use of `rebalanceUserDiscountPercent` or any other function that accumulates the user's interest, results in insignificant benefits for the end user given that the values of the expected configuration of interest and discount rates are low.

Severity: Informational

Issue:	Using of `WadRayMath.sol` not according to guidelines
Description:	`WadRayMath.sol` states that `wadMul/wadDiv` and `rayMul/rayDiv` should be called with both operands should have the same format of `WAD` / `RAY` respectively. In `ghoVariableDebtToken.sol` there are multiple occasions where `rayMul` and `rayDiv` are called with the first operand being some token balance (both scaled and not-scaled) which is formatted as `WAD`, and the second operand being the index which is formatted as `RAY`.
Mitigation/Fix:	The `GhoVariableDebtToken` contains code that belongs to the standard Aave `VariableDebtToken` implementation. Although it is not natural to use `WadRayMath` functions with operands that aren't in the same format, these calculations provides a result with correct format.

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 GHO Token

GHO is a decentralized, multi-collateral token that is fully backed and native to the Aave Protocol. GHO introduces the concept of facilitators. A facilitator (e.g., a protocol, an entity, etc.) has the ability to trustlessly generate (and burn) GHO tokens. The first facilitator is the Aave Protocol - specifically the Aave Pool on Ethereum. The Aave – GHO integration employs the same mechanisms as any other asset listed on the Aave Protocol – a specific GHO aToken and GHO Debt Token will be deployed. As a decentralized token on the Ethereum Mainnet, GHO will be minted by Aave upon every new debt position and the tokens will be burned upon repayment of the debt. All the interest payments accrued by minters of GHO are directly transferred to the GHO DAO treasury.

The implementation of GHO includes a Discount Strategy mechanism. The discount strategy allows for Safety Module participants (stkAAVE holders) to access a discount on the GHO borrow rate.

Assumptions and Simplifications Made During Verification

We made the following assumptions during our verifications:

- Our verification included verifying all of the GHO contracts that are required for the utilization of the GHO Token as an asset under the Aave's V3 pool. For the purpose of verifying the GHO Token we assume that the V3 Pool contracts are working as intended.
- In cases where a contract is referencing another contract and the reference is set by a dedicated setter function and not via the constructor/initializer, we assume that the reference was already set when verifying the contract (e.g. the reference to GhoAToken in GhoVariableDebtToken).
- We unroll loops. Violations that require a loop to execute more than two times will not be detected.
- We assume that all token balances are formatted with 18 decimals (WAD), the global index is formatted with 27 decimals (RAY) and discount rate is formatted with 2 decimals (BPS).

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.

Verification of GhoVariableDebtToken

Implements a variable debt token to track the borrowing positions of users at variable rate modes for GHO. The token is an interest-accruing token that is minted and burned on borrowing and repayment of the GHO token, representing the debt owed by the token holder.

High Level Properties

✔️ 1. discountCantExceed100Percent

No account may have more than a 100% (10000 bps) discount at any point in time.

✔️ 2. discountCantExceedDiscountRate

No account may have more than `DISCOUNT_RATE()` discount at any point in time.

✔️ 3. disallowedFunctionalities

Calls to `transfer`, `allowance`, `approve`, `transferFrom`, `increaseAllowance`, or `decreaseAllowance` must always revert.

❌ 4. onlyRebalanceCanUpdateUserDiscountRate

Only `rebalanceUserDiscountPercent` can recalaulate and update user's discount rate.

> Since commit 81bb51b, which removed the timelock protection on the user's discount rate, this property is no longer true and both `mint`, `burn` and `updateDiscountDistribution` can update the user's discount rate as well.

✔️ 5. nonMintFunctionCantIncreaseBalance

A user's debt token balance (as reported by `balanceOf`) must not increase by calling any external function other than the mint function.

> We assume here that the global index, as reported by `POOL::getReserveNormalizedVariableDebt`, is the same as the index stored in the user state in the contract storage (`_userState[user].additionalData`).

✓ 6. nonMintFunctionCantIncreaseScaledBalance

A user's debt token scaled balance (as reported by `scaledBalanceOf`) must not increase by calling any external function other than the mint function.

✓ 7. debtTokenIsNotTransferable

Actual debt tokens (represented by the scaled balance) must not be transferable.

✓ 8. onlyCertainFunctionsCanModifyScaledBalance

Functions other than `burn`, `mint`, `rebalanceUserDiscountPercent` and `updateDiscountDistribution` must not be able to modify the user's scaled balance.

✓ 9. userAccumulatedDebtInterestWontDecrease

Only a call to `decreaseBalanceFromInterest` can decrease the user's accumulated interest as stored in the user's state (`_ghoUserState[user].accumulatedDebtInterest`).

✓ 10. userCantNullifyItsDebt

A user must not be able to nullify (have a zero balance when calling `balanceOf`) their debt without calling `burn`. This rule assumes the the index is fixed before, during, and after the call to `burn`.

Integrity of `mint`

✓ 11. integrityOfMint_updateDiscountRate

After calling `mint`, the user's discount rate (as stored in `_ghoUserState[user].discountPercent`) must be equal to the output of `GhoDiscountRateStrategy::calculateDiscountRate` when called with the current debt balance and the current balance of stakedAave.

✓ 12. integrityOfMint_updateIndex

After calling `mint`, the user's state (`_userState[user].additionalData`) must be updated with the current value of the global index.

✓ 13. integrityOfMint_userIsolation

Calling `mint` must not effect another user's scaled balance.

✓ 14. onlyMintForUserCanIncreaseUsersBalance

When calling `mint`, the user's balance will increase if the call is made on behalf of the user.

⏱️ 15. integrityOfMint_cantDecreaseInterestWithMint

A user cannot decrease her accumulated interest by increasing her debt position (i.e. by calling `mint`).

✓ 16. mint_cant_increase_bal_by_more_than_amountScaled

The balance of a user post minting cannot exceed his balance before minting by more than the scaledAmount corresponding to the amount specified.

Integrity of `burn`

✓ 17. integrityOfBurn_updateDiscountRate

After calling `burn`, the user's discount rate (as stored in `_ghoUserState[user].discountPercent`) must be equal to the output of `GhoDiscountRateStrategy::calculateDiscountRate` when called with the current debt balance and the current balance of stakedAave.

✓ 18. integrityOfBurn_updateIndex

After calling `burn`, the user's state (`_userState[user].additionalData`) must be updated with the current value of the global index.

✓ 19. integrityOfBurn_fixedIndex

When the index stored in the user's balance (`_userState[user].additionalData`) is the same as the current global index, calling `burn(user, amount)` must decrease the user's scaled balance by the scaled amount.

✓ 20. burnZeroDoesntChangeBalance

Calling burn with 0 amount must not change the user's balance.

✓ 21. integrityOfBurn_fullRepay_concrete

After calling `burn` with `amount` equal to the user's current balance (as returned by `balanceOf`) the user's scaled balance must be zero.

> Note: This rule assumes that the index in the user's state (`_userState[user].additionalData`) equals 1ray() and the global index during the call to `burn` equals 2ray().

✓ 22. integrityOfBurn_userIsolation

Calling `burn` must not effect another user's scaled balance.

✓ 23. burnAllDebtReturnsZeroDebt

When calling `burn` for an amount of current debt, the debt position must be nullified.

Integrity of `updateDiscountDistribution`

⏱️ 24. integrityOfUpdateDiscountDistribution_discountRate

After calling `updateDiscountDistribution`, the user's discount rate (as stored in `_ghoUserState[user].discountPercent`) must be equal to the output of `GhoDiscountRateStrategy::calculateDiscountRate` when called with the current debt balance and the current balance of stakedAave.

✓ 25. integrityOfUpdateDiscountDistribution_updateIndex

After calling `updateDiscountDistribution`, the user's state (`_userState[user].additionalData`) must be updated with the current value of the global index, unless the sender is the recipient, then the indexes must remain unchanged.

✓ 26. integrityOfUpdateDiscountDistribution_userIsolation

Calling `updateDiscountDistribution` must not effect any user's scaled balance other than the sender and the recipient.

✓ 27. sendersDiscountPercentCannotIncrease

when calling `updateDiscountDistribution`, the sender's discount rate cannot increase.

✓ 28. discount_takes_place_in_updateDiscountDistribution__sender, discount_takes_place_in_updateDiscountDistribution__recipient

When calling `updateDiscountDistribution`, if `discountScaled > 0` then the balance of the sender and recipient must decrease, otherwise it must stay the same.

✓ 29. in_updateDiscountDistribution_amount_affects_sender_discount, in_updateDiscountDistribution_amount_affects_recpient_discount

When calling `updateDiscountDistribution`, the argument `amount` has influence on the `discountPercent` of the sender and recipient.

Integrity of `rebalanceUserDiscountPercent`

✓ 30. integrityOfRebalanceUserDiscountPercent_updateDiscountRate

After calling `rebalanceUserDiscountPercent`, the user's discount rate (as stored in `_ghoUserState[user].discountPercent`) must be equal to the output of `GhoDiscountRateStrategy::calculateDiscountRate` when called with the current debt balance and the current blance of stakedAave.

✓ 31. integrityOfRebalanceUserDiscountPercent_updateIndex

After calling `rebalanceUserDiscountPercent`, the user's state (`_userState[user].additionalData`) must be updated with the current value of the global index.

✓ 32. integrityOfRebalanceUserDiscountPercent_userIsolation

Calling `rebalanceUserDiscountPercent` must not effect any user's scaled balance other than the sender and the recipient.

Integrity of `balanceOf`

✓ 33. integrityOfBalanceOf_fullDiscount

If a user has 100% discount, the output of `balanceOf` must be fixed over time since this account doesn't accumulate any interest.

✓ 34. integrityOfBalanceOf_noDiscount

If a user has 0% discount, the output of `balanceOf` must be equal to `rayMul(scaledBalanceOf, current value of the global index)`.

✓ 35. integrityOfBalanceOf_zeroScaledBalance

`balanceOf` must return zero if the user has zero scaled balance.

Integrity of `accrueDebtOnAction`

✓ 36. accrueAlwaysCalleldBeforeRefresh

Accrue of a user is always called before refresh of a user.

> Note: We proved that accrue is called exactly once before a refresh although accruing more than that might be fine to be on the safe side.

✓ 37. positive_balanceIncrease [1]

`accumulatedDebtInterest` is monotonically increasing on accrue and the `additionalData` of a user is updated to the current index.

updates of special addresses in the contract

✓ 38. nonzeroNewDiscountToken

`updateDiscountToken` cannot set the discount token to address 0.

✓ 39. integrityOfUpdateDiscountRateStrategy

`updateDiscountRateStrategy` is setting the corresponding storage slot to the passed value

✓ 40. _ghoAToken_cant_be_zero

`setAToken` cannot set the ghoAToken to address 0.

Verification of GhoDiscountRateStrategy

Implements the calculation of the discount rate depending on the current strategy.

Properties

✓ 41. maxDiscountForHighDiscountTokenBalance

`calculateDiscountRate` must return the maximal discount rate (`DISCOUNT_RATE`) if the user's debt balance and discount token balance are above the defined threasholds (`MIN_DEBT_TOKEN_BALANCE` and `MIN_DISCOUNT_TOKEN_BALANCE` respectively) and the user entitled balance for discount is above his debt balance.

✓ 42. zeroDiscountForSmallDiscountTokenBalance

`calculateDiscountRate` must return 0 if either the debt balance or the discount token balance is below the defined threashold (`MIN_DEBT_TOKEN_BALANCE` and `MIN_DISCOUNT_TOKEN_BALANCE` respectively).

✓ 43. limitOnDiscountRate

`calculateDiscountRate` output must be limited by `DISCOUNT_RATE` .

Verification of GhoToken

The main contract of the GHO Token, implementing the ERC20 interface. This contract also manages the different facilitators and is responsible for the bookkeeping of the GHO token utilization by the facilitators.

ERC20 properties

✓ 44. inv_balanceOf_leq_totalSupply

A user balance must never exceed the total supply.

✓ 45. sumAllBalance_eq_totalSupply

The sum of all user balances must be equal to the total supply.

Intergrity of Facilitators maintenance

✓ 46. length_leq_max_uint160

At any point in time there must not be more than 2^160 facilitators listed in the state of the contract.

✓ 47. total_supply_eq_sumAllLevel

The sum of all facilitator levels (i.e. their current utilization) must be equal to the total supply.

✓ 48. sumAllLevel_eq_sumAllBalance

The sum of all user balances must be equal to the sum of all facilitator levels.

✓ 49. inv_valid_capacity

All facilitators with non-zero capacity must have a non-empty label.

✓ 50. inv_valid_level

All facilitators with non-zero level must have a non-empty label.

✓ 51. address_in_set_values_iff_in_set_indexes

`_facilitatorsList` and `_facilitators` must be in sync.

✓ 52. addr_in_set_iff_in_map

All facilitators in `_facilitators` must have a non-empty label.

✓ 53. level_leq_capacity

A facilitator's level must be lower than its capacity unless `setFacilitatorBucketCapacity` was called.

✓ 54. facilitator_in_list_after_setFacilitatorBucketCapacity

After calling `setFacilitatorBucketCapacity`, the facilitator must still be registered correctly under `_facilitators` and `_facilitatorsList`.

✓ 55. getFacilitatorBucketCapacity_after_setFacilitatorBucketCapacity

Calling `getFacilitatorBucketCapacity` after calling `setFacilitatorBucketCapacity` must return the correct facilitator capacity.

✓ 56. facilitator_in_list_after_addFacilitator

Calling `addFacilitator` must add the facilitator to `_facilitators` and `_facilitatorsList`.

✓ 57. address_not_in_list_after_removeFacilitator

Calling `removeFacilitator` must remove the facilitator from `_facilitatorsList`.

HL properties

✓ 58. mint_after_burn

If a facilitator's level is below its capacity the first call to `mint` after calling `burn` must succeed unless the call to `burn` followed a call to either `setFacilitatorBucketCapacity` or `removeFacilitator`.

✓ 59. burn_after_mint

Calling `burn` after calling `mint` with the same amount must succeed.

✓ 60. level_unchanged_after_mint_followed_by_burn

A facilitator's level must remain the same after calling `mint` and `burn` with the same amount.

✓ 61. level_after_mint

Calling `mint(amount)` must increase that facilitator's level by that amount.

✓ 62. level_after_burn

Calling `burn(amount)` must decrease that facilitator's level by that amount.

✓ 63. facilitator_in_list_after_mint_and_burn

If a call to either `mint` or `burn` succeeded, the facilitator must be registered correctly under `_facilitators` and `_facilitatorsList`.

✓ 64. balance_after_mint

Calling `mint(user, amount)` must increase the user balance and the total supply by that amount.

✓ 65. balance_after_burn

Calling `burn(amount)` must increase the `msg.sender` balance and the total supply by that amount.

✓ 66. mintLimitedByFacilitatorRemainingCapacity

It must not be posible to mint more than the facilitator's remaining capacity.

✓ 67. burnLimitedByFacilitatorLevel

It must not be posible to burn more than the facilitator's current level.

✓ 68. OnlyFacilitatorManagerAlterFacilitatorExistence

Only facilitator manager can add or remove a facilitator

✓ 69. OnlyBucketManagerCanChangeCapacity

Only bucket manager can change the capacity of a facilitator

Verification of GhoAToken

The implementation of the interest bearing token for the Aave Protocol. GHO is supplied to the Aave Protocol via facilitators. The GhoAToken is a facilitator containing the logic for GHO to work as a reserve with the Aave Protocol.

Properties

✓ 70. noMint

`mint` must always revert.

✓ 71. noBurn

`burn` must always revert.

✓ 72. noTransfer

`transfer` must always revert.

✓ 73. transferUnderlyingToCantExceedCapacity

Calling `transferUnderlyingTo` must revert if the amount exceeds the remaining capacity.

✓ 74. totalSupplyAlwaysZero

The total supply of GhoAToken must always be zero.

✓ 75. userBalanceAlwaysZero

All user balances must always be zero.

Verification of GhoFlashMinter

The implementation a flashloan facilitator.

Properties

✓ 76. balanceOfFlashMinterGrows

The GHO balance of the flash minter should grow when calling any function, excluding `distributeFees`.

✓ 77. integrityOfTreasurySet

Checks the integrity of `updateGhoTreasury` - after update the given address is set.

✓ 78. integrityOfFeeSet

Checks the integrity of `updateFee` - after update the given value is set.

✓ 79. availableLiquidityDoesntChange

Checks that the available liquidity, retreived by `maxFlashLoan`, stays the same after any action.

✓ 80. integrityOfDistributeFeesToTreasury

Checks the integrity of `distributeFees`:

1. As long as the treasury contract itself isn't acting as a flashloan minter, the flashloan facilitator's GHO balance should be empty after distribution.
2. The change in balances of the receiver (treasury) and the sender (flash minter) is the same. i.e. no money is being generated out of thin air.

✓ 81. feeSimulationEqualsActualFee

Checks that the fee amount reported by `flashFee` is the the same as the actual fee that is taken by flashloaning.

> Note: the simulation was done through a mock contract, and excluded calls to `distributeFees` and `flashLoan` while in the flashloan.

1. Also proved by `user_index_up_to_date` ↩