certora

Security Assessment

aave

Aave v3.5

July-2025

Prepared for:
Aave DAO

Code developed by:
BORED GHOSTS DEVELOPING

Table of contents

Project Summary

Project Scope

Project Name	Repository (link)	Latest Commit Hash	Platform
Aave v3.5	Github Repository	f76f007	EVM

Project Overview

This document describes the verification of **Aave v3.5** code using manual code review. The work was undertaken from **June 12** to **July 3, 2025**.

The following contracts are considered in scope for this review:

- `Diff between commit 6c1b680 and f76f007`

The team performed a manual audit of all the solidity contracts. Issues discovered during the review are listed in the following pages.

Protocol Overview

Aave v3.5 is an upgrade to the core protocol, aiming to improve mathematical precision through deterministic rounding. It further simplifies the code, increases the predictability of the execution outcome, and ensures rounding consistently favours the protocol rather than producing arbitrary results.
Initially planned for inclusion in the v3.4 release, it is finally released as an independent version to facilitate reviews and other security procedures.

Findings Summary

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

Severity	Discovered	Confirmed	Fixed
Critical	-	-	-
High	-	-	-
Medium	2	2	2
Low	2	2	2
Informational	5	5	5
Total	9	9	8

Severity Matrix

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

Detailed Findings

Audit Goals

1. Rounding calculations are correctly implemented and consistently done in favor of the system.
    a. User operations:
        i. When depositing, a user should never receive more aTokens than the value of their actual underlying contribution.
        ii. When a user withdraws assets, the protocol must burn at least the corresponding amount of aTokens (rounded up) to ensure the user does not receive any excess due to rounding.
        iii. When a user borrows, the debt (vToken) is rounded up, ensuring the protocol never under-records user debt.
        iv. When a user repays their debt, the protocol must burn at most the corresponding amount of vTokens (rounded down) to ensure that the entire underlying debt is fully covered.
    b. When taking fees, the protocol rounds in favour of the system and not the treasury.
        i. When applying protocol fees, all rounding must favor the protocol, not the treasury, to prevent over-allocation.
        ii. When accruing fees to the treasury, the treasury must not receive more than its allocated share, regardless of rounding.
        iii. When minting aTokens to the treasury, the minted amount must be based on the actual accrued value, ensuring no excess due to scaling or rounding.
        iv. For flash loan fees, the treasury must only receive its defined portion, without gaining extra due to precision loss.
        v. For liquidation fees, the treasury's cut must be strictly limited to its allocated share — rounding must never inflate treasury gains.
2. When transferring aTokens, the target always gets at least the amount transferred.
    a. When transferring aTokens, the recipient must receive at least the amount specified — rounding is applied in favor of the receiver.

   b. The total supply of aTokens must remain unchanged, ensuring that no value is created or lost during user-to-user transfers.

3. Health Factor Accuracy
   a. Asset and debt conversions to the base currency must use pessimistic rounding to avoid overstating collateral or understating debt.
   b. Collateral values should be rounded down, and debt values rounded up, ensuring the protocol remains conservative in risk assessment and resistant to rounding-based exploits.
4. Scaled Accounting Consistency
   a. The protocol should minimize precision loss by avoiding unnecessary conversions between scaled and unscaled values.
   b. Functions like `mint` and `burn` should use precomputed `scaledAmount` where possible to avoid redundant recalculations and improve gas efficiency.
   c. Different paths should round consistently. Specifically, borrowing debt directly should result in the same amount of debt as flashloaning this amount and turning it into debt (instead of repaying immediately).
5. Allowance consumed must match the actual transferred amount, accounting for rounding, to prevent overspending and maintain consistency.

Coverage and Conclusions

1. Rounding calculations are correctly implemented and consistently done in favor of the system.
We extensively verified that all arithmetic operations across user and system-facing flows apply a consistent rounding strategy that favors the protocol and prevents users from gaining value through precision loss.

- **Deposits** convert the underlying contribution to aTokens using `rayDivFloor`, rounding down to ensure users never receive more tokens than they should.
- **Withdrawals** convert the requested amount to aToken units using `rayDivCeil`, intentionally burning slightly more aTokens to guarantee users always receive at least the amount they requested, without extracting more than their proportional share.
- **Borrows** apply `rayDivCeil` to scale up the user's debt, ensuring that the protocol never under-records the amount owed.
- **Repayments** use `rayMulCeil`, rounding up to ensure that the full debt is covered even if slightly more is paid, preventing stuck dust or under-repayment.

We confirmed these behaviors through manual inspection of arithmetic primitives and simulations across rounding edges, ensuring that in no case could users extract value from rounding artifacts. This approach preserves value integrity, maintains conservative accounting, and neutralizes edge-case exploits across the system.

2. When transferring aTokens, the target always gets at least the amount transferred.
We verified that aToken transfers preserve value integrity and apply rounding in favor of the recipient. When a user transfers aTokens, the recipient always receives **at least** the specified amount, ensuring that rounding never results in under-delivery. This is achieved by rounding up the scaled amount on the recipient side, avoiding any loss due to precision truncation. Meanwhile, the sender's balance is reduced by the **exact** transferred amount, and the total aToken supply remains unchanged throughout the operation. This ensures strict value conservation—no tokens are created or destroyed — preventing any systemic gain or loss during repeated user-to-user transfers.
We confirmed this behavior through targeted unit tests and simulations of small-value and rounding-edge scenarios, validating that both sender and recipient balances update as expected without introducing supply drift.

3. Health factor and liquidation calculations are conservatively rounded.

To maintain risk safety, we verified that the health factor logic uses pessimistic rounding – collateral is rounded down using `rayMulFloor`, while debt is rounded up using `rayMulCeil`. This ensures that users are never treated as overcollateralized, reducing the chance of bypassing liquidation thresholds.

We verified the correctness of this logic in `calculateUserAccountData()` and related liquidation checkpoints. Simulated user portfolios were stress-tested to confirm that the protocol always errs on the side of caution in collateral/debt ratio calculations.

4. Scaled Accounting Consistency
The protocol minimizes precision loss by consistently operating on scaled values and avoiding unnecessary conversions between scaled and unscaled units. Critical processes such as cap validations and treasury allocations are performed directly in scaled units, preserving accuracy and preventing rounding drift.
Functions like `mint()` and `burn()` leverage precomputed scaledAmount inputs where applicable, ensuring consistent arithmetic and reducing gas consumption. We validated these behaviors across key components to confirm alignment with the intended accounting model and system efficiency.
The protocol uses rounding consistently. Since the protocol always rounds in its own favor, the different paths (and in particular, the two ways of directly borrowing tokens) result in the same amount of aTokens and vTokens minted/burned, and the same amount of underlying token transferred.

5. Allowance Accounting Consistency
We confirmed that Aave v3.5 improves allowance handling by deducting `scaledUpFloor(scaledDownCeil(amount))`, aligning more closely with actual transfer behavior. To maintain backwards compatibility, the allowance deducted is the greater of the input amount or corrected amount, preventing overspending while preserving integration safety.

Medium Severity Issues

M-01. DoS Due To Inconsistent Rounding in Protocol Fee Transfer During Liquidation		
Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: libraries/logic/ LiquidationLogic.sol	Status: Fixed	

Description:

During the liquidation process in the protocol, the fee intended for the treasury (`liquidationProtocolFeeAmount`) is calculated and transferred based on the borrower's aToken balance.

A discrepancy arises between the way the fee is computed and how it is ultimately transferred, leading to unintended reverts under certain conditions.

1. **Pre-transfer check**:
   The `liquidationProtocolFeeAmount` is converted to a scaled amount using **floor division**:

```JavaScript
uint256 scaledDownLiquidationProtocolFee = vars.liquidationProtocolFeeAmount.rayDivFloor(
  vars.collateralReserveCache.nextLiquidityIndex
);
```

2. **Transfer execution**:
   The actual transfer, however, implicitly uses **ceiling division** to compute the scaled amount internally:

```JavaScript
IAToken(vars.collateralReserveCache.aTokenAddress).transferOnLiquidation(
  params.borrower,
  IAToken(vars.collateralReserveCache.aTokenAddress).RESERVE_TREASURY_ADDRESS(),
  vars.liquidationProtocolFeeAmount,
  vars.collateralReserveCache.nextLiquidityIndex
);
```

This mismatch causes the system to approve a liquidation path based on a *floor-rounded* balance check, but fail during execution due to a *ceil-rounded* token requirement.

Example:

- Liquidity Index: `1.39`
- LiquidationBonus: `110_00 (10%)`
- LiquidationFeePercentage: `20_00 (20%)`
- Borrower's Scaled Collateral Balance: `49*10^8`
- Total Collateral to be Paid: `floor(49*10^8 * 1.39) = floor(68.11*10^8) = 6,811,000,000`
    - Collateral from liquidation bonus: `6,811,000,000 - floor(6,811,000,000/1.1` (liquidation bonus)`) = 6,811,000,000 - 6,191,818,181 = 619,181,819`
    - Fee: `619,181,819 * 0.2` (protocol fee) `= 123,836,363.8` → `123,836,364`
    - Liquidator's Portion: `6,811,000,000 - 123,836,364` (protocol fee) = `6,687,163,636` → scaled: `ceil(6,687,163,636 / 1.39) = ceil(4,810,909,090.64) = 4,810,909,091`
    - Remaining Scaled Balance (after transferring the liquidator's part): `49*10^8 - 4,810,909,091 = 89,090,909`
- Treasury Fee: `123,836,364`
    - Scaled Down (floor): `floor(123,836,364 / 1.39) = floor(89,090,909.3525179856) = 89,090,909` → check passes
    - Actual Required (ceil): `ceil(123,836,364 / 1.39) = ceil(89,090,909.3525179856) = 89,090,910` → transfer fails

Even though the check passes under the floor assumption, the actual transfer reverts due to insufficient scale balance (`89,090,910` required, `89,090,909` available).

Likelihood:
The requirements for this revert are:

1. A non-zero liquidation bonus
2. A non-zero liquidation fee
3. The entire collateral a user has in the target token is liquidated.

This can occur either through bad debt or when a user has their collateral split between multiple tokens, and has a debt larger than one of them (the target token). This also requires the numbers to align such that the two rounding forms - `floor` and `ceil` diverge (i.e. the treasury fee does not divide evenly by the index). Although this requires a specific set-up, this state is not uncommon.

To determine what's the likelihood to revert depending on the parameters, we performed simulations running through scaled balance range for given liquidation parameters trio.

For example, simulating DAI's live configuration on Ethereum (July 2025):

- `Liquidity Index = 1.13`
- `Liquidation Bonus = 105%`
- `Liquidation Protocol Fee = 20%`

On a range of scaled balances between `1e18` and `1e18 + 1e8` `(100M values),` we found that ~55% of the values will cause revert.
Some other liquidation trios showed an even higher chance of reverting, for example :

- `Liquidity Index = 1.39`
- `Liquidation Bonus = 110%`
- `Liquidation Protocol Fee = 20%`

On the same range, we found the ~63.5% of the collateral balances values will cause revert.

Furthermore, due to dust protection in liquidation that forces either liquidation of all collateral, all debt or leave a min amount of at least $1000 worth of asset, this can DoS liquidations of positions within the criteria quite substantially and with good probability.

Recommendation:
To ensure consistency between the balance check and actual transfer logic, replace `rayDivFloor` with `rayDivCeil` during the scaled fee calculation:

```
JavaScript
uint256 scaledDownLiquidationProtocolFee = vars.liquidationProtocolFeeAmount.rayDivCeil(
  vars.collateralReserveCache.nextLiquidityIndex
);
```

This change ensures the scaled amount used in pre-transfer checks fully accounts for the value required during transfer, thereby eliminating these avoidable transaction reverts.

Customer's response: Issue fixed in commit b6567d4.

Fix Review: Fix confirmed.

M-02. Allowance decrease less than what it should		
Severity: **Medium**	Impact: **Low**	Likelihood: **High**
Files: tokenization/VariableDebtToken.sol	Status: Fixed	

Details:
When taking debt on another user's expense through approved allowance, we mint their debt rounded up to the closest scaled value, but decrease the allowance only by the amount requested. This means that giving someone an allowance to mint debt on your behalf allows them to mint more debt than the allowance cap.

Recommendation:

Decrease the allowance by at least the debt difference (getVTokenBalance() before and after).

Customer's response: Issue fixed in commit b7beed9 by rounding up the scaled debt (calling getVTokenBalance() with the amount minted).

Fix Review: Fix confirmed.

Low Severity Issues

L-01. Treasury almost always accrues 1 less than it should when calling executeMintToTreasury		
Severity: **Low**	Impact: **Low**	Likelihood: **Medium**
Files: Libraries/Logic/Pool.sol	Status: Fixed	

Details:
Each reserve accrues payment to the treasury in scaled aToken denomination.
When calling executeMintToTreasury(), this number is converted to the underlying token, rounding down (rayMulFloor). Then, it calls aToken.mintToTreasury(), which mints the corresponding scaled aTokens to the treasury, rounding down again (rayDivFloor).

When rounding the first time (scaled aToken -> underlying), the loss is denominated in the underlying token with a max of 1 wei underlying.
When rounding the second time, the deviation is denominated in the scaled aToken, with a max loss of 1 wei scaled.

Prior to the introduction of deterministic rounding, since the scaled token is by definition >= underlying, if both operations are rounded in the same direction, we ended up with **one scaled wei less/more** than we started with. However, if both are rounded in opposite directions, the amount rounded up/down during the first rounding will be less than 1 scaled wei, hence when rounding back we get the same value.

While this problem occurs consistently when calling executeMintToTreasury(), in practice the function is called infrequently. Technically, since the function is permissionless, an attacker can call this as often as they'd desire to incur some loss, but such behavior is unprofitable.

Mathematically:
More formally, we'll define:

```JavaScript
c := n * index - floor(n * index)
```

(*) So assuming we could send the accrued to treasury stored scaled amount directly to `mintScaled()`, we would've minted `n` scaled aTokens which are worth `n * index` underlying to the treasury.

However, with the current flow we scale up (rounding down) to get:
`n` scaled aTokens, converted to `n * index - c` underlying

Then, when scaling back down (rounding down) we get:
`n-1` scaled aToken, which worth `(n-1) * index`

Of course rounding is not an issue in the rare case where `c = 0`.

Recommendation:
By rounding up the first time (scale up), we can fix the rounding error:
We will define:

```JavaScript
c := ceil(n * index) - n * index
```

So compared to the "pure result" we defined in (*), scaling up, rounding up first will get us to:
`n` scaled aTokens, converted to `n * index + c`

Then, when scaling down again (rounding down) we get:
`n` scaled aTokens, which are worth `n * index` underlying, to the treasury.

Customer's response: Issue fixed in commit 070cd23 by passing the scaled accruedToTreasury directly to mintToTreasury() (and changing the function to work with the scaled value).

Fix Review: Fix confirmed. The solution solves the problem on a deeper level than suggested.

L-02. Redundant Multiplication and division may cause worse HF via gift/additional supply		
Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: libraries/logic/GenericLogic.sol	Status: Fixed	

Details:
Rounding revolving around the weighted average calculation of LT can, in specific cases, result in a slight drop in the total debt allowed before liquidation (HF reduction). This occurs due to a redundant division operation in line 156, which comes after a multiplication in line 131, although line 161 needs exactly the result calculated in line 131.

The code for the multiplication in line 131:

```JavaScript
vars.avgLiquidationThreshold +=
  vars.userBalanceInBaseCurrency *
  (vars.isInEModeCategory ? vars.eModeLiqThreshold : vars.liquidationThreshold);
```

The code for the division in line 156:

```JavaScript
vars.avgLiquidationThreshold = vars.totalCollateralInBaseCurrency != 0
  ? vars.avgLiquidationThreshold / vars.totalCollateralInBaseCurrency
  : 0;
```

The code for the final HF calculation in line 161:

```JavaScript
vars.healthFactor = (vars.totalDebtInBaseCurrency == 0)
  ? type(uint256).max
  : (vars.totalCollateralInBaseCurrency.percentMul(vars.avgLiquidationThreshold)).wadDiv(
    vars.totalDebtInBaseCurrency
  );
```

Example:

We will start with an initial state as follows:

- A user has 1 collateral of WETH on Ethereum.
- userBalanceInBaseCurrency(WETH) = 1e9
- LT(WETH) = 83_00

Now:

- accumulatedCollateral(WETH) = vars.avgLiquidationThreshold(WETH) (line 130) = userBalanceInBaseCurrency(WETH) * LT(WETH) = 8,300,000,000,000
- avgLiquidationThreshold(WETH) = accumulatedCollateral(WETH) / totalCollateralInBaseCurrency(WETH) = 83_00
- TotalDebtAllowed = totalCollateralInBaseCurrency.percentMul(avgLiquidationThreshold) = $830,000,000

Now, assuming the user supplies an amount from another collateral with a worse LT, or alternatively, a malicious user gifts this collateral (supply on behalf), say PT-sUSDE-31JUL2025:

- supply amount = 1 wei
- LT(PT) = 10
- LTV > 0 -> can be supplied

Now let's see the total variable debt the user can have after this gift:

- userBalanceInBaseCurrency(WETH + PT) = 1e9 + 1
- accumulatedCollateral(WETH + PT) =: vars.avgLiquidationThreshold (line 130) = userBalanceInBaseCurrency(WETH) * LT(WETH) userBalanceInBaseCurrency(PT) * LT(PT) = 8,300,000,000,000 + 1 * 10 = 8,300,000,000,010
- avgLiquidationThreshold(WETH + PT) = accumulatedCollateral(WETH + PT) / totalCollateralInBaseCurrency(WETH + PT) = 8,300,000,000,010 / (1e9 + 1) = 82_99
- TotalDebtAllowed = totalCollateralInBaseCurrency.percentMul(avgLiquidationThreshold) = (((1e9 + 1) * 82_99) + 5,000) / 10,000 = $829,900,001

From this example above, we can see that by adding collateral to a user's position - something that should always improve the LT - we are able to worsen the user's HF.

Recommendation:

Since the calculation of HF uses the numerator of the weighted average which is already precisely calculated in line 131, the HF calculation in line 161, should be done before the division in line 156.

The new HF calculation should read:

```
JavaScript
 vars.healthFactor = (vars.totalDebtInBaseCurrency == 0)
   ? type(uint256).max
   : vars.avgLiquidationThreshold.wadDiv(
     vars.totalDebtInBaseCurrency
   )/100_00;
```

Customer's response: Issue fixed in commit c3e88e1.

Fix Review: Fix confirmed.

Informational Issues

I-01. isCollateral flag not set to false when it should

Details:
There are multiple cases where the check for turning off the collateral flag depends on the scaled amount (rounded down), converted to the underlying token, being equal to the amount of tokens the user requested for an operation.

executeEliminateDeficit():

```javascript
if (isCollateral && balanceWriteOff == userBalance) {
  userConfig.setUsingAsCollateral(reserve.id, params.asset, params.user, false);
}
```

executeWithdraw():

```javascript
if (isCollateral && amountToWithdraw == userBalance) {
  userConfig.setUsingAsCollateral(reserve.id, params.asset, params.user, false);
}
```

executeFinalizeTransfer():

```javascript
if (params.balanceFromBefore == params.amount) {
  fromConfig.setUsingAsCollateral(reserveId, params.asset, params.from, false);
}
```

Since with rounding, in some cases, multiple distinct values of underlying token can be scaled to the same amount of shares (scaled tokens), these checks can fail if the user burns all the scaled tokens, although specifying a scaled-up amount that isn't their entire balance.

Example:

- Liquidity index: `1.54`

- User’s scaled balance: `12`
- User’s unscaled balance: `floor(12 * 1.54) = 18`
- Total collateral to be withdrawn: `17`
    - Total aTokens burned: `ceil(17 / 1.54) = 12`

So the user will have `12 - 12 = 0` scaled tokens left, but `18 != 17`, so the collateral flag will be left on.

Recommendation:
Either simulate the same rounding on the user input as on the user’s balance, or have burn() and transfer() return if there are any aTokens left.

Customer’s response: Issue fixed in commits 7bdd945 and 070cd23 by comparing the scaled values or by returning if anything is left from burn().

Fix Review: Fix confirmed.

I-02. Move new health check inside if block

Details:
When repaying with aTokens, the health factor is always checked.
This is unnecessary if the user does not have the corresponding reserve enabled as collateral; their health factor cannot worsen.

This prevents legitimate debt repayment by users that the protocol does not want to prevent.

Recommendation:
The health factor check can be placed inside the existing check:

```javascript
onBehalfOfConfig.isUsingAsCollateral(reserve.id)
```

To save gas in the general case.

Customer's response: Issue fixed in commit 3031d6e.

Fix Review: Fix confirmed.

I-03. Incorrect emit amount when calling _mintScaled()

Details:
When minting scaled tokens, the total amount reported as minted is calculated by adding the new mint amount to the user's balance increase due to index growth.

```javascript
amountToMint = amount + balanceIncrease;
```

However, this calculation can be inaccurate due to two key reasons:

- The user's input is provided in underlying units, but it is converted to scaled units (scaledAmount) using floor division.
- There's a difference in the result between:
  - Adding two (scaled) values, then rounding
  - Rounding two values and then adding them

The `amountToMint` calculation can overstate the actual balance increase because it adds the full underlying input (before rounding) to a balance increase that's already rounded down. Since both the minting and index-based growth involve floor rounding, this results in a reported amount that's higher than the real change in the user's balance, leading to misleading event emissions.

Example:

- User's old index (_userState[onBehalfOf].additionalData): 1.3
- New index: 1.5
- User's starting scaled balance: 3
- Amount minting: 10
- Scaled amount: Floor(10 / 1.5) = 6
- Scaled balance after: 3 + 6 = 9
- Balance increase: Floor(1.5 * 3) - Floor(1.3 * 3) = 4 - 3 = 1
- **Event emitted**: 10 (input) + 1 (balance increase) = 10 + 1 = 11
- balanceOf() before: Floor(1.3 * 3) = 3
- balanceOf() before: Floor(1.5 * 9) = 13
- **Actual increase**: 13 - 3 = 10

So, while the actual balance change (as returned by `balanceOf()`) is **10**, the emitted event incorrectly reports **11**, resulting in a mismatch between on-chain state and emitted logs.

Recommendation:

Use the difference between the **balanceOf()**s, whether directly or through an equivalent calculation (for example, adding the difference when raising the index and then when minting, but using the correct difference calculation).

Customer's response: Issue fixed in commit 070cd23 by calculating the diff in **balanceOf()** directly to determine amountToMint.

Fix Review: Fix confirmed.

I-04. Overprotective Overflow Guards in Ceil/Floor Math Variants

Description:
WadRayMath.rayMulFloor() uses:

```JavaScript
gt(a, div(sub(not(0), HALF_RAY), b))
```

While the function computes:

```JavaScript
c = (a * b) / RAY
```

with no rounding constant involved.

WadRayMath.rayDivCeil() uses:

```JavaScript
gt(a, div(sub(not(0), HALF_RAY), b))
```

While it rounds using mod(...) + 1, and does not add b / 2.

PercentageMath.percentMulCeil() uses:

```JavaScript
gt(a, div(sub(not(0), HALF_RAY), b))
```

While rounding is also done via mod(...) + 1.

All these overflow checks were inherited from earlier implementations like rayMul(), which used half-up rounding (+HALF_RAY, etc.). In the updated versions, the guards are unnecessarily conservative.

Impact:
Any multiplication result in the range:
(MAX_UINT256 - rounding_const, MAX_UINT256] (where rounding_const = HALF_RAY, b/2, HALF_PERCENTAGE_FACTOR)
will revert — even though the actual computation would safely fit within 256 bits.

This imposes unnecessary constraints and could trigger avoidable reverts in high-value or high-precision scenarios.

Of course, the likelihood of encountering such edge cases in current usage is very low, especially under typical reserve configurations. However, relaxing these guards would still improve robustness and gas efficiency.

Recommendation:
To reflect the true bounds of safe operations, the overflow checks should be simplified as follows:

```JavaScript
// rayMulFloor
gt(a, div(not(0), b))

// rayMulCeil
gt(a, div(not(0), b))

// rayDivFloor
gt(a, div(not(0), RAY))

// rayDivCeil
gt(a, div(not(0), RAY))

// percentMulCeil
gt(value, div(not(0), percentage))
```

Customer's response: Issue fixed in commit 8405f6b.

Fix Review: Fix confirmed.

I-05. paybackAmount Based on onBehalfOf Instead of user in useATokens Path

Description:
In executeRepay(), the following logic was introduced to calculate paybackAmount when a user repays using their aTokens:

```JavaScript
paybackAmount = IAToken(...).scaledBalanceOf(params.onBehalfOf).rayMulFloor(...);
```

However, the tokens are actually burned from params.user, not onBehalfOf. Therefore, this should ideally read:

```JavaScript
(...).scaledBalanceOf(params.user)
```

Related Logic Already Present in Previous Versions
A similar pattern exists in earlier versions and still present in the function body:

```JavaScript
if (params.useATokens) {
  IAToken(...).burn(params.user, ...);

  bool isCollateral = onBehalfOfConfig.isUsingAsCollateral(reserve.id);

  if (isCollateral && IAToken(...).scaledBalanceOf(params.user) == 0) {
    onBehalfOfConfig.setUsingAsCollateral(reserve.id, params.asset, params.user, false);
  }
}
```

What Happens Here:

- aTokens are correctly burned from params.user.
- The aToken balance check also references params.user, ensuring that collateral can be disabled if their balance is zero.
- However, the line:

```JavaScript
bool isCollateral = onBehalfOfConfig.isUsingAsCollateral(reserve.id);
```

uses onBehalfOfConfig, which refers to the configuration of params.onBehalfOf, and not params.user.

- Similarly, the collateral update:

```JavaScript
onBehalfOfConfig.setUsingAsCollateral(reserve.id, params.asset, params.user, false);
```

also uses onBehalfOfConfig, but updates the state for params.user.

While this is **not exploitable** since repayWithATokens() enforces params.user == params.onBehalfOf, this is conceptually inaccurate.

Why This Is A Problem:

- The mismatch will lead to issues if third-party aToken repayments are ever introduced as a feature in the protocol.

Recommendation:
To future-proof and clarify this path:

- Use params.user consistently in any aToken-related logic — when calculating balances, performing burns, or updating config flags.

This ensures a coherent mental model and protects against regressions if logic or call paths evolve in future releases.

Alternatively, consider splitting this logic into a dedicated executeRepayWithATokens() function to streamline flow and reduce contextual dependencies.

Customer's response: Acknowledged, as repayment with aTokens will always keep the params.user == params.onBehalfOf validation, and additionally, repayment with aTokens is a feature candidate for deprecation in the future.

Formal Verification

Verification Methodology

We performed verification of the **Aave 3.5** protocol using the Certora verification tool which is based on Satisfiability Modulo Theories (SMT). In short, the Certora verification tool works by compiling formal specifications written in the Certora Verification Language (CVL) and **Aave**'s implementation source code written in Solidity.
More information about Certora's tooling can be found in the Certora Technology Whitepaper.

If a property is verified with this methodology it means the specification in CVL holds for all possible inputs. However specifications must introduce assumptions to rule out situations which are impossible in realistic scenarios (e.g. to specify the valid range for an input parameter). Additionally, SMT-based verification is notoriously computationally difficult. As a result, we introduce overapproximations (replacing real computations with broader ranges of values) and underapproximations (replacing real computations with fewer values) to make verification feasible.

Rules: A rule is a verification task possibly containing assumptions, calls to the relevant functionality that is symbolically executed and assertions that are verified on any resulting states from the computation.

Inductive Invariants: Inductive invariants are proved by induction on the structure of a smart contract. We use constructors as a base case, and consider all other (relevant) externally callable functions that can change the storage as step cases.
Specifically, to prove the base case, we show that a property holds in any resulting state after a symbolic call to the respective constructor. For proving step cases, we generally assume a state where the invariant holds (induction hypothesis), symbolically execute the functionality under investigation, and prove that after this computation any resulting state satisfies the invariant.

Verification Notations

Formally Verified	The rule is verified for every state of the contract(s), under the assumptions of the scope/requirements in the rule.
Formally Verified After Fix	The rule was violated due to an issue in the code and was successfully verified after fixing the issue
Violated	A counter-example exists that violates one of the assertions of the rule.

Detailed Properties

P-01. The Solvency Property.	
Status: Verified	**High-level Description:** For each asset, the pool's total liability to its liquidity providers is **at most** the sum of the debtors' total liability to the pool and available liquidity.

Rule Name	Status	Description	Link to rule report
supply__solvency	Verified	*The solvency property is preserved when calling the supply() function.*	here
withdraw__solvency	Verified	*The solvency property is preserved when calling the withdraw() function.*	here
borrow__solvency	Verified	*The solvency property is preserved when calling the borrow() function.*	here
repay__solvency	Verified	*The solvency property is preserved when calling the repay() function.*	here

repay__revertsIF_totDbtEQ0	Verified	*If the scaled-total-supply of the variable-debt token is 0, then repay() reverts. (This is a lemma used in the rule repay__solvency.)*	here
repay__index_unchanged	Verified	*The return values of getNormalizedIncome() and getNormalizedDebt() are the same before, after, and during the call to repay(), but in the same timestamp. (This is a lemma used in the rule repay__solvency.)*	here
repayWithATokens__solvency	Verified	*The solvency property is preserved when calling the repayWithATokens() function.*	here
repayWithATokens__revertsIF_totDbtEQ0	Verified	*If the scaled-total-supply of the variable-debt token is 0, then repayWithATokens() reverts. (This is a lemma used in the rule repayWithATokens__solvency.)*	here
flashLoanSimple__solvency	Verified	*The solvency property is preserved when calling the flashLoanSimple() function.*	here
eliminateReserveDeficit__solvency	Verified	*The solvency property is preserved when calling the eliminateReserveDeficit() function.*	here
syncIndexesState__solvency	Verified	*The solvency property is preserved when calling the syncIndexesState() function.*	here
liquidationCall__solvency	Verified	*The solvency property is preserved when calling the liquidationCall() function. We divided it into three rules. The first for the debt asset, the second for the collateral asset (assuming these assets are different), and the third rule treats the case that these assets are identical.*	here, here, and here.
liquidationCall__index_unchanged	Verified	*The return values of getNormalizedIncome() and getNormalizedDebt() (of both debt-asset and collateral-asset) are the same before, after, and during the call to liquidationCall(), but in the same timestamp. (This is a lemma used in the rules liquidationCall__solvency.) As with the liquidationCall__solvency rule, we divided it into three rules.*	here, here, and here.
liquidationCall__revertsIF_totDbt_of_DBTasset_EQ0	Verified	*If scaledTotalSupplyCVL(_DBT_debt)==0, then liquidationCall() reverts. (This is a lemma used in the rules liquidationCall__solvency.)*	here

Details

We show that for a given asset, the pool's total liability to liquidity providers can never exceed the sum of the pool's assets, i.e., the sum of lent and available liquidity.
Formally, we define the solvency property **P** for a given asset **a** as:

total-supply ≤ total-debt + VB + deficit

where

- **total-supply** is the AToken supply of **a**, i.e. **`_reserves[a].aTokenAddress.totalSupply()`**
- **total-debt** is the VariableDebtToken supply of **a**, i.e. **`_reserves[a].variableDebtTokenAddress.totalSupply()`**
- **VB** is the virtual balance of **a**, i.e. **`_reserves[a].virtualUnderlyingBalance`**
- **deficit** is the deficit of **a**, i.e. **`_reserves[a].deficit.`**

We proved that the solvency property is preserved by any call to the following pool functions (also listed in the table above) with any arbitrary values as input:

- `supply`(),
- `withdraw`(),
- `borrow`(),
- `repay`(),
- `repayWithATokens`(),
- `flashLoanSimple`(),
- `liquidationCall`(),
- `eliminateReserveDeficit`(),
- `syncIndexesState`().

We stress that the solvency property, **P**, depends not only on the contract's storage, but also on the timestamp it is evaluated on. This is due to `totalSupply()` functions on both the aToken, and variableDebtToken being dependent on timestamp.
Formally, we assume an arbitrary fixed state **S** at timestamp **T** where **P** holds.
We then symbolically call a function **f** on state **S**, at timestamp **T**, and show that **P** still holds at the new formed state **S'** (still at timestamp **T**).
Note: Suppose we call **f** at time **T**, although we assume that **P** holds on the state **S** at time **T**, i.e. before the call to **f**, we **do not** assume that the last-updated-timestamp of the pool is also **T**. Namely, the last-updated-timestamp may be any **T'<T**.

Limitations

1. The property **does not** account for "time-passing", i.e.:

Assuming **P** holds at time **T**, if we allow time to propagate to **T'** such that **T<T'** with no function being invoked in the period **(T, T')**, then **P** also holds on time **T'**.

2. The proof with respect to `flashLoanSimple()` assumes no re-entrancy to the pool from the callback.

Assumptions

We assume both indexes – liquidity-index and variable-debt-index, are greater or equal to 10^27 (one RAY).

Disclaimer

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.