Formal Verification of Aave V3 upgrade to V3.0.1

Summary

This document describes the specification and verification of an upgrade to Aave V3 into version V3.0.1 using the Certora Prover. The work was undertaken from November 17, 2022, to December 15, 2022, on commit `428e258`. The new version introduces fixes that were discovered after a year of testing V3 across different networks - context.

This verification project aims to enhance the existing specifications of Aave V3 core and add new properties. The scope of our specifications includes particularly the following contracts:

- `StableDebtToken.sol` (`Verification Results`)
- `VariableDebtToken.sol` (`Verification Results`)
- `Atoken.sol` (`Verification Results`)
- `ReserveConfiguration.sol` (`Verification Results`)
- `UserConfiguration.sol` (`Verification Results`)

And partial verification of:

- `Pool.sol` (`Verification Results`)

The Certora Prover proved the protocol implementation is correct with respect to formal specifications written by the Certora team. The team also performed a manual review of the contracts.

The resulting specification files are available on Aave's public git repository.

List of Main Issues Discovered

Severity: Low

Issue:	Inconsistency in `getNormalizedDebt()`
Description:	The getter `getNormalizedDebt()` simulates the variable debt accumulated since the last update of the index. Since the debt is ever growing, it is expected from this value to be non-decreasing between 2 Separate calls; however, due to an optimisation done in `_updateIndexes` (which resides in `ReserveLogic`), when the total debt is 0, the borrowing index isn't being updated. This means that if a user calls the getter in block `n` when the total debt is 0, then executes an action that updates the indexes, and calls the getter again, even in the same block, the result of the second simulation will be lower then the first one.
Property violated:	The debt index should be monotonically increasing
Mitigation/Fix:	`getNormalizedDebt()` usage (directly or other intermediate functions) is safe for the protocol, given that it is always used as a factor of multiplications with another component, which in the problematic cases (non-monotonic increase) will be 0. Leaving this inconsistency on edge cases is preferable to the architectural changes required to be fully consistent. The decision has been to make really explicit on the header of the externally exposed `getReserveNormalizedVariableDebt()` that external usage should be done with caution, understanding the possible divergence with the variable borrow index.

Disclaimer

The Certora Prover takes a contract, and a specification as input 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 this information is useful, but 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.

Overview of the verification

Description of the system

Aave is a decentralised non-custodial liquidity market protocol where users can participate as suppliers or borrowers. Suppliers provide liquidity to the market to earn a passive income, while borrowers can borrow in an over-collateralised (perpetually) or under-collateralized (one-block liquidity) fashion.

Assumptions and Simplifications

We made the following assumptions during the verification process:

- `getReserveNormalizedIncome` and `getReserveNormalizedVariableDebt` return a constant value of a whole RAY.
- Due to computational complexities, all properties of `AToken` are proven assuming `finalizeTransfer` and `handleAction` are never called.
- We assume that hash operations return an arbitrary deterministic value
- We unroll loops. Violations that require executing a loop more than once will not be detected.
- When verifying contracts that make external calls, we assume that those calls can have arbitrary side effects outside of the contracts but do not affect the state of the contract being verified. This means that some reentrancy bugs may not be caught.

Verification Conditions

Notation

✓ indicates the rule is formally verified on the latest reviewed commit. Footnotes describe any simplifications or assumptions used while verifying the rules (beyond the general assumptions listed above).

This document shows verification conditions as logical formulas or Hoare triples of the form {p} C {q}. A verification condition given by a logical formula denotes an invariant that holds if every reachable state satisfies the condition.

Hoare triples of the form {p} C {q} hold if any non-reverting execution of program C that starts in a state satisfying the precondition p ends in a state satisfying the postcondition q. The notation {p} C@withrevert {q} is similar but applies to reverting and non-reverting executions. Preconditions and postconditions are similar to the Solidity require and assert statements.

Formulas relate the results of method calls. In most cases, these methods are getters defined in the contracts, but in some cases, they are getters we have added to our harness or definitions provided in the rules file. Undefined variables in the formulas are treated as arbitrary: the rule is checked for every possible value of the variables.

Formal Properties

Due to rounding errors, equalities in `VariableDebtToken` and `AToken`'s specifications are correct within an error bar defined by `epsilon` such that:
$actual\ result = mathematical\ result \pm \epsilon$

where `epsilon` is defined as:
$[\frac{index}{RAY} + 1] \cdot \frac{1}{2}$

All rules that are affected by this will be marked with an asterisk (*)

StableDebtToken

Invariants

1. principalLessThanBalance ✓

The principal balance of a given user is always smaller than or equal to the current debt balance of the user.

```
principalBalanceOf(user) <= balanceOf(user)
```

Integrity Rules

2. integrityBurn ✓

Burning user `u` amount of `x` tokens decreases their debt balance by `x`.

```
{
    balanceBefore = balanceOf(user)
}
    burn(user,x)
{
    balanceOf(user) = balanceBefore - x
}
```

3. integrityMint ✓

Minting to user `u` amount of `x` tokens increases their debt balance by `x`.

```
{
    balanceBefore = balanceOf(user)
}
    mint(delegatedUser,user,x,index)
{
    balanceOf(user) = balanceBefore + x
}
```

4. integrityTimeStamp ✓

lastUpdated timestamp mustn't be set to the future.

```
getUserLastUpdated(user) <= currentBlock.timestamp
```

5. integrityDelegationWithSig ✓

DelegationwithSig advances the nonce by one and changes the borrow allowance to the given value.

```
{
    oldNonce = nonces(delegator)
}
    delegationWithSig(delegator, delegatee, value, deadline, v, r, s);
{
    nonces(delegator) = oldNonce + 1 ∧ borrowAllowance(delegator, delegatee) =
}
```

Equivalency of Multiple operations

6. additiveBurn ✓

Burn is additive; burning all at once has the same consequences as burning gradually.

```
burn(user, x, index)
burn(user, y, index)
~
burn(user, x+y, index)
```

7. additiveMint ✓

Mint is additive; burning all at once has the same consequences as burning gradually.

```
mint(delegatedUser, a, x, index)
mint(delegatedUser, a, y, index)
```

```
~
mint(delegatedUser, a, x+y, index)
```

8. inverseMintBurn ✓

Minting and then burning Variable Debt Tokens should not affect the user's balance

```
{
    balancebefore = balanceOf(User)
}
        mint(delegatedUser, User, amount, rate)
        burn(User, amount)
{
        balancebefore == balanceOf(User)
}
```

Permissions

9. whoChangeTotalSupply ✓

Only the pool can change the total supply with burn or mint operation.

```
{
    oldTotalSupply = totalSupply(e);
}
    < call to any contract method f >
{
    oldTotalSupply != totalSupply() ⇒
        msg.sender == POOL(e) ∧
        (f = burn(args) v f = mint(args))
}
```

10. nonceChangePermits ✓

only `delegationWithSig` operation can change the nonce

```
{
    oldNonce = nonces(user)
}
    < call to any contract method f >
{
    oldNonce != nonces(user) ⇒
        f = delegationWithSig(args)
}
```

Misc

11. balanceOfChange ✓

Each operation of a Stable Debt Token can change at most one user's balance.

```
{
    balanceABefore = balanceOf(UserA) ∧
    balanceBBefore = balanceOf(UserB)
}
    < call to any contract method f >
{
    (balanceABefore == balanceAAfter v
    balanceBBefore == balanceBAfter)
}
```

12. burnZeroDoesntChangeBalance ✓

Buring zero amount of tokens should have no effect.

```
{
    balanceBefore = balanceOf(User)
}
    burn(User, 0)
{
    balanceBefore = balanceOf(User)
}
```

13. disallowedFunctionalities ✓

This property makes sure that each disallowed function reverts every time. Although trivial, it is a handy alarm mechanism for further development through CI.

```
{ }
    < call to any contract method f >
{
    lastReverted
}
```

VariableDebtToken

Invariants

1. totalSupplyEqualsSumAllBalance ✓

The total debt accumulated with variable interest equals the sum of the variable interest debt of all users.

```
totalSupply() == sumAllBalance()
```

Integrity Rules

2. integrityOfBurn ✓ *

See property No. 2 in StableDebtToken

3. integrityMint ✓ *

See property No. 3 in StableDebtToken

4. integrityDelegationWithSig ✓

See property No. 5 in StableDebtToken

Equivalency of Multiple operations

5. additiveBurn ✓ *

See property No. 6 in StableDebtToken

6. additiveMint ✓ *

See property No. 7 in StableDebtToken

7. inverseMintBurn ✓

See property No. 8 in StableDebtToken

Permissions

8. whoChangeTotalSupply ✓

See property No. 9 in StableDebtToken

9. nonceChangePermits ✓

See property No. 10 in StableDebtToken

Misc

10. balanceOfChange ✓

See property No. 11 in StableDebtToken

11. burnZeroDoesntChangeBalance ✓

See property No. 12 in StableDebtToken

12. disallowedFunctionalities ✓

See property No. 13 in StableDebtToken

AToken

Invariants

1. totalSupplyEqualsSumAllBalance ✓

See property No. 1 in VariableDebtToken

Integrity Rules

2. integrityBurn ✓*

Burn from user `u` amount of `x` tokens decreases total supply and the user's underlying balance by `x`, within the bounded error range.

```
{
    user ≠ currentContract ∧
    balanceBeforeUser = balanceOf(user) ∧
    balanceBeforeTo = balanceOf(to) ∧
    underlyingBeforeTo =  _underlyingAsset.balanceOf(to) ∧
    underlyingBeforeUser =  _underlyingAsset.balanceOf(user) ∧
    underlyingBeforeSystem =  _underlyingAsset.balanceOf(currentContract) ∧
    uint256 totalSupplyBefore = totalSupply()
}
        burn(user, to, amount, indexRay)
{
    user ≠ to ⇒
        (balanceOf(to) == balanceBeforeTo ∧
        bounded_error_eq(underlyingBeforeUser, _underlyingAsset.balanceOf(user
    ∧
    to ≠ currentContract ⇒
        bounded_error_eq(_underlyingAsset.balanceOf(currentContract), underlyi
        bounded_error_eq(_underlyingAsset.balanceOf(to),  underlyingBeforeTo +
    ∧
    to = currentContract ⇒
        _underlyingAsset.balanceOf(currentContract) = underlyingBeforeSystem
    ∧
    bounded_error_eq(totalSupply(), totalSupplyBefore - amount, 1)
    ∧
    bounded_error_eq(balanceOf(user), balanceBeforeUser - amount, 1)
}
```

3. integrityMint ✓*

Mint to user `u` amount of `x` tokens increases AToken total supply and the user's underlying balance by `x`, within the bounded error range.

```
{
    underlyingBalanceBefore = balanceOf(u) ∧
    atokenBlanceBefore = scaledBalanceOf(u) ∧
    totalATokenSupplyBefore = scaledTotalSupply()
}
    mint(b, u, x, indexRay);
{
    (scaledBalanceOf(u) - atokenBlanceBefore = scaledTotalSupply() - totalATok
    (totalATokenSupplyAfter > totalATokenSupplyBefore) ∧
    bounded_error_eq(balanceOf(u), underlyingBalanceBefore+x, 1);
}
```

4. integrityTransfer ✔ *

Transferring Atokens between users mustn't change the underlying balance of any entity; The balances of the sender and receiver must change by the amount within a bounded error range.

```
{
    balanceBeforeFrom = balanceOf(from)
    balanceBeforeTo = balanceOf(to)
    underlyingBeforeOther =  _underlyingAsset.balanceOf(other)
}
    transfer(to, amount);
{
    _underlyingAsset.balanceOf(other) = underlyingBeforeOther ∧
    from != to ⇒ (bounded_error_eq(balanceOf(from), balanceBeforeFrom - amount
    bounded_error_eq(balanceOf(to), balanceBeforeTo + amount, 1)) ∧
    from != to ⇒ balanceAfterFrom == balanceAfterTo , "unexpected balance of f
}
```

5. permitIntegrity ✔

Permit advances the nonce by one and sets the allowance correctly

```
{
    nonceBefore = nonces(owner)
}
    permit(owner, spender, value, deadline, v, r, s)
{
    allowance(owner, spender) = value ∧
    nonces(owner) == nonceBefore + 1
}
```

Equivalency of Multiple operations

6. additiveBurn ✓*

See property No. 7 in StableDebtToken

7. additiveMint ✓*

See property No. 8 in StableDebtToken

8. additiveTransfer ✓*

Transfer is additive. Transferring all at once has the same consequences as transferring gradually.

```
transfer(e1, to1, x);
transfer(e1, to1, y);
~
transfer(e2, to2, x+y);

bounded_error_eq(balanceFromScenario1, balanceFromScenario2, 3)  ∧
bounded_error_eq(balanceToScenario1, balanceToScenario2, 3)
```

Misc

9. balanceOfChange ✓

Each possible operation changes the balance of at most two users.

```
{
        balanceABefore = balanceOf(UserA)
        balanceBBefore = balanceOf(UserB)
        balanceCBefore = balanceOf(UserC)
}
        < call to any contract method f >
{
    balanceABefore = balanceOf(UserA) v balanceBBefore == balanceOf(UserB) v b
}
```

ReserveConfiguration

1. Integrity Setters/Getters ✓

The setters and getters of parameter `p` are coherent.

```
{ }
    set_P(value)
```

```
{
    get_P() = value
}
```

2. Independence of Setters and Getters ✔

Setting parameter `p` doesn't affect any other parameters' (`p'`) value.

```
{
    valueBefore_p' = get_P'()
}
    set_P(value)
{
    get_P'() = valueBefore_p'
}
```

UserConfiguration

Integrity Rules

1. setBorrowing/setUsingAsCollateral ✔

The integrity of setBorrowing/setUsingAsCollateral (`set_P`) functions and correct retrieval of the corresponding getter (`get_P`)

```
{ }
    set_P(reserveIndex, value)
{
    get_P(reserveIndex) = value
}
```

2. setBorrowingNoChangeToOther/setCollateralNoChangeToOther ✔

Changes made to a specific borrowing/Collateral asset don't affect the other assets

```
{
    otherReserveBorrowingBefore = isBorrowing(reserveIndexOther) ∧
    otherReserveCollateralBefore = isUsingAsCollateral(reserveIndexOther)
}
    set_P(reserveIndex, value);
{
    (reserveIndex ≠ reserveIndexOther ⇒
        (isBorrowing(reserveIndexOther) = otherReserveBorrowingBefore ∧
        isUsingAsCollateral(reserveIndexOther) = otherReserveCollateralBefore)
}
```

3. isUsingAsCollateralOrBorrowing ✓

Verifying the correlation between `isUsingAsCollateralOrBorrowing`, `isUsingAsCollateral` and `isBorrowing`

```
(isUsingAsCollateral(reserveIndex) v
isBorrowing(reserveIndex)) ⇔
isUsingAsCollateralOrBorrowing(reserveIndex)
```

4. integrityOfisUsingAsCollateralOne ✓

If one asset is used as collateral and `isUsingAsCollateralOne` is true, then any other asset is not used as collateral.

```
isUsingAsCollateral(reserveIndex) ∧
    isUsingAsCollateralOne()) ⇒
(!isUsingAsCollateral(reserveIndexOther) v
    reserveIndexOther = reserveIndex)
```

5. integrityOfisUsingAsCollateralAny ✓

If at least one asset is used as collateral, `isUsingAsCollateralAny` is true.

```
isUsingAsCollateral(reserveIndex) ⇒
    isUsingAsCollateralAny()
```

6. integrityOfisBorrowingOne ✓

If one asset is used for borrowing and isBorrowingOne, then any other asset is not used for borrowing.

```
(isBorrowing(reserveIndex) ∧
    isBorrowingOne()) ⇒
!isBorrowing(reserveIndexOther) v
    reserveIndexOther = reserveIndex
```

7. integrityOfisBorrowingAny ✓

If at least one asset is borrowed, isBorrowing is true.

```
isBorrowing(reserveIndex) ⇒
    isBorrowingAny()
```

8. integrityOfEmpty ✓

if user data is empty, then for any index, neither borrowing nor collateral is set

```
isEmpty() ⇒
    !isBorrowingAny() ∧
    !isUsingAsCollateralOrBorrowing(reserveIndex)
```

9. integrityOfIsolationModeState ✓

If `IsolationModeState` is active, only one asset must be registered as collateral.

```
!isUsingAsCollateralOne() ⇒ !isIsolated()
```

10. integrityOfSiloedBorrowingState ✓

If `IsolationModeState` is active, only one asset must be registered as a borrowing asset.

```
!isBorrowingOne() ⇒ !isSiloed()
```

Pool

1. getReserveNormalizedVariableDebtCheck ✓

The borrowing index should monotonically increase when there's total debt.

```
{
    asset ≠ _aToken ∧
        oldIndex = getReserveNormalizedVariableDebt(args) ∧
        totalDebtBefore = getCurrScaledVariableDebt(asset)
}
    supply(asset, amount, onBehalfOf, referralCode)
{
    totalDebtBefore ≠ 0 ⇒
        getReserveNormalizedVariableDebt(args) ≥ oldIndex
}
```