# Formal Verification of Aave Protocol V3

## Summary

This document describes the specification and verification of Aave's V3 protocol using the Certora Prover. The work was undertaken from Nov. 12, 2021 to Jan. 24, 2022. The latest commit that was reviewed and ran through the Certora Prover was `a87d546`.

The scope of this verification is Aave's governance system, 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 verificaiton of: `Pool.sol`

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

The specification program was modularized to optimize coverage. First, the tokenization contracts were found to uphold to the same properties the Aave V2 tokenization did, as well as additional properties. On the main Pool contract, the focus of the verification was the protocol's storage of its reserves data, their classification to EModes - a new feature of the V3 protocol - and its compatibility with the user's action. This was done by modularly checking the userConfiguration and reservesConfiguration libraries first.

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

## List of Main Issues Discovered

**Severity:** Critical

| **Issue:** | **Loss of assets** |
|---|---|
| Description: | RepayWithATokens function lacks an HF check, can be exploited to withdraw liquidity from the system for free. |
| Mitigation/Fix: | Canceled repayment with ATokens on behalf of another user |
| Property violated: | Any Operation Should Preserve User's HF>1 |

**Severity:** High

| **Issue:** | **Risk Exposure** |
|---|---|
| Description: | User can come to hold both an isolated and a non-isolated assets as collaterals upon calling AToken.transfer(), liquidation call and mintUnbacked(). Can be exploited to surpass the debt ceiling |
| Mitigation/Fix: | A check for isolation mode was added to the functions |
| Property violated: | A User Can't Hold Both an Isolated and non-Isolated Assets as Collaterals |

**Severity:** High

| **Issue:** | **Loss of assets** |
|---|---|
| Description: | Confusion of Asset and EMode price feed for liquidations |
| Mitigation/Fix: | Price Sources are called according to EMode |
| Property violated: | Emode source price usage |

**Severity:** Medium

| Issue: | Loss of Users' Profitability |
|---|---|
| Description: | EMode liquidation may use wrong liquidation bonus |
| Mitigation/Fix: | Bonus rewarded correctly according to EMode |

**Severity:** Medium

| Issue: | Loss of revenue |
|---|---|
| Description: | When repaying with aToken, the interest rate is updated as if we provided the equivalent liquidity in underlying instead of in AToken. In fact there's no liquidity provided to the system. It can be used to manipulate the interest rates. |
| Mitigation/Fix: | Call to rates updating function was changed to use 0 as the added liquidity |
| Rule Coverage: | No Change in Underlying's Balance Implies No Change in rate |

**Severity:** Low

| Issue: | Integrity of ReserveList |
|---|---|
| Description: | *_addReserveToList* function will push a new reserve into all empty spots of the reserves list, instead of just the first one |
| Mitigation/Fix: | A *return* call was inserted to the loop |
| Rule Coverage: | The same asset should not appear twice on the reserves list |

**Severity:** Recommendation

| Issue: | Denial of Service |
|---|---|
| Description: | Users can be forced into isolation mode through supply(),AToken.transfer() functions, thus temporarily preventing them from borrowing other assets |
| Property violated: | Informative Rule: Check which functions can revert for one user due to another user's action |
| Mitagation/Fix: | User can withdraw asset of isolation mode |

## 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 the verification

## Description of the system

Aave is a decentralized non-custodial liquidity markets protocol where users can participate as suppliers or borrowers. Suppliers provide liquidity to the market to earn a passive income, while borrowers are able to borrow in an overcollateralized (perpetually) or undercollateralized (one-block liquidity) fashion.

## Description of the specification files

The specification contains six files, three for the tokenization part, one for the pool and one for each of the reserve and user configuration contracts. The tokens' contracts have similar specifications, using (up to slight modifications) properties based on Certora's aggregated experience with ERC20 verificartion. On the main Pool contract, the focus of the coverage was the protocol's storage of its reserves data, their classification to EModes - a new feature of the V3 protocol - and its compatibility with the user's action. This was done by modularly checking the userConfiguration and reservesConfiguration libraries first.

# Assumptions and Simplifications

We made the following assumptions during the verification process:

- Accumulation rates in the protocol are assumed to be 1, so that no interest rate is accumulated.
- We assume that hash operations return an arbitrary deterministic value
- We unroll loops. Violations that require a loop to execute 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 that they do not affect the state of the contract being verified. This means that some reentrancy bugs may not be caught.
- No usage of incentives controller

# 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).

In this document, verification conditions are either shown 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} holds if any non-reverting execution of program C that starts in a state satsifying the precondition p ends in a state satisfying the postcondition q. The notation {p} C@withrevert {q} is similar but applies to both 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.

# Stable-Debt Token

## Rules

### 1. Integrity of User's Time-Stamp ✔

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

### 2. Any Operation Changes At Most One User's balance ✔

```
{ A!= B && balanceA == balanceOf(A) && balanceB == balanceOf(B) }
      op ;
{ balanceOf(A) == balanceA || balanceOf(B) == balanceB }
```

where `op` is any operation

### 3. Change in Total Supply iff an Equal Change in Some User's Balance ✔

```
{ balanceA == balanceOf(A) && total == totalSupply() }
      op ;
{ balanceOf(A) != balanceA  => balanceOf(A) - balanceA  == totalSupply() - total };
```

### 4. Additivity of Burn - Burning at Once Equals Burning in Two Steps in the Same Transaction ✔

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

### 5. Integrity of Mint ✔

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

### 6. Integrity of Burn ✔

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

#### 7. Invertibility of Mint and Burn ✔

```
{ b == balanceOf(user) }
    mint(delegatedUser,user,x,index); burn(user, x)
{ balanceOf(user) == b }
```

## Variable-Debt Token

### Rules

#### 1. Any Operation Changes At Most One User's balance ✔

```
{ A!= B && balanceA == balanceOf(A) && balanceB == balanceOf(B) }
      op ;
{ balanceOf(A) == balanceA || balanceOf(B) == balanceB }
```

#### 2. Change in Total Supply iff an Equal Change in Some User's Balance ✔

```
{ balanceA == balanceOf(A) && total == totalSupply() }
    op ;
{ balanceOf(A) != balanceA  => balanceOf(A) - balanceA  == totalSupply() - total };
```

#### 3. Only pool can change total supply with burn and mint ✔

```
{ total == totalSupply() }
    op ;
{ totalSupply() != total => msg.sender == pool && (op == mint() || op == burn()) };
```

#### 4. Mint and burn are inverse operations ✔

```
{ b == balanceOf(user) }
    mint(delegatedUser,user,amount,index); burn(user, amount, index)
{ balanceOf(user) == b }
```

#### 5. Additivity of burn ✔

```
burn(u, u', x); burn(u, u', y) ~ burn(u, u', x+y) at the same timestamp
```

#### 6. Integrity of mint ✔

```
{ b = balanceOf(u) }
    mint(u,x)
{ balanceOf(u) == b + x }
```

#### 7. Burn doesn't change other's balance and data ✔

```
{ other != user && bb = balanceOf(other) }
    burn(user, amount, index)
{ balanceOf(other) == bb }
```

#### 8. Mint doesn't change other's balance and data ✔

```
{ other != user && bbo = balanceOf(other) && bbu = balanceOf(user) }
    mint(user, onBehalfOf, amount, index)
{ balanceOf(other) == bbo && (user != onBehalfOf => balanceOf(user) == bbu) }
```

#### 9. Burn zero dosen't change balance ✔

```
{ b = balanceOf(user) }
    burn(user, 0, index)
{ balanceOf(user) == b }
```

## AToken

### Rules

#### 1. Any Operation Changes At Most Two User's balance ✔

```
{ A != B != C && balanceA == balanceOf(A) && balanceB == balanceOf(B) && balanceC == balanceOf(C) }
      op ;
{ balanceOf(A) == balanceA || balanceOf(B) == balanceB ||  balanceOf(C) == balanceC }
```

#### 2. Integrity of mint ✔

```
{ b == balanceOf(u) }
    mint(u,x)
{ b + x - ε ≤ balanceOf(u) ≤ b + x + ε }
```

#### 3. Additivity of mint ✔

```
mint(u,x); mint(u,y) ~ mint(u,x+y)
```

with resepct to balanceOf(u) up to some `ε`

#### 4. Mint of zero is not possible ✔

```
{  }
    mint(caller, user, amount, index)
{ amount == 0 => REVERT }
```

#### 4. Integrity of transfer ✔

```
I.
{ u ≠ u’ ∧ bu = balanceOf(u) ∧ bu’ = balanceOf(u’) }
    transfer(u, u’ x);
{ | balanceOf(u) - (bu - x) | ≤ ε ∧
  | balanceOf(u’) - (bu’ + x) | ≤ ε }
II.
{ b = underlyingAssetBalanceOf(u’’) }
    transfer(u, u’ x);
{ b = underlyingAssetBalanceOf(u’’) }
```

#### 5. Additivity of transfer ✔

```
{ f1 != f2, t1 != t2, f2 != t1, f1 != t2, f1 == t1 <=> f2 == t2,
  balanceOf(f1) == balanceOf(f2), balanceOf(t1) == balanceOf(t2) }
    transfer(f1, t1, x), transfer(f1, t1, y), transfer(f2, t2, x+y)
{ |balanceOf(f1) - balanceOf(f2)| ≤ 3ε, |balanceOf(t1) - balanceOf(t2)| ≤ 3ε }
```

#### 6. Integrity of burn ✔

```
{ bu = balanceOf(u) ∧ ba = underlyingAssetBalanceOf(u’) }
    burn(u, u’, x)
{ | balanceOf(u) - (bu - x) | ≤ ε ∧
  u’ ≠ AToken ⇒ | underlyingAssetBalanceOf(u’) - (ba + x) | ≤ ε }
```

#### 7. Additivity of burn ✔

```
burn(u, u’, x); burn(u, u’, y) ~ burn(u, u’, x+y) at the same timestamp
```

#### 8. burn doesn't change other's balance and data ✔

```
{ other != user, other != recieverOfUnderlying,
  db = additionalData(other), bb = balanceOf(other)}
    burn(user, recieverOfUnderlying, amount, index)
{ additionalData(other) == db && balanceOf(other) == bb }
```

#### 9. mint doesn't change other's balance and data ✔

```
{ other != user, db = additionalData(other), bb = balanceOf(other) }
    mint(caller, user, amount, index)
{ additionalData(other) == db && balanceOf(other) == bb }
```

## ReserveConfiguration

### Rules

#### 1. Integrity Setters/Getters ✔

```
{ r = ReserveConfiguration, x = some member of r }
    setX(r, y)
{ getX(r) == y }
```

#### 2. Each setter changes only one member ✔

```
{ r = ReserveConfiguration, x = some member of r, x' = another member of r }
    setX(r, y)
{ getX'(r) == x' }
```

## UserConfiguration

### Rules

#### 1. Consistency of isEmpty ✔

```
{isEmpty => !isBorrowingAny() && !isUsingAsCollateralOrBorrowing(reserveIndex) }
```

#### 2. Consistency of not isEmpty ✔

```
{ (isBorrowingAny() ||  isUsingAsCollateral(reserveIndex)) => !isEmpty() }
```

#### 3. Consistency of isBorrowingAny ✔

```
{ isBorrowing(reserveIndex) =>  isBorrowingAny() }
```

#### 4. Consistency of isUsingAsCollateral functions ✔

```
(isUsingAsCollateral(reserveIndex) || isBorrowing(reserveIndex))
    <=> isUsingAsCollateralOrBorrowing(reserveIndex)
```

#### 5. Integrity of User's Isolation mode configuration data ✔

```
!isUsingAsCollateralOne() => !isIsolated()
```

#### 6. Integrity of setBorrowing ✔

```
{ reserveIndex < 128 }
    setBorrowing(reserveIndex, borrowing)
{ isBorrowing(reserveIndex) == borrowing }
```

#### 7. Integrity of setUsingAsCollateral ✔

```
{ reserveIndex < 128 }
    setUsingAsCollateral(reserveIndex, usingAsCollateral)
{ isUsingAsCollateral(reserveIndex) == usingAsCollateral }
```

#### 8. setBorrowing for one reserve doesn't affect others ✔

```
{ reserveIndexOther != reserveIndex && reserveIndexOther < 128 &&
  reserveIndex < 128 &&
  otherReserveBorrowing == isBorrowing(reserveIndexOther) &&
  otherReserveCollateral == isUsingAsCollateral(reserveIndexOther) }
    setBorrowing(reserveIndex, borrowing)
{ otherReserveBorrowing == isBorrowing(reserveIndexOther) &&
  otherReserveCollateral == isUsingAsCollateral(reserveIndexOther) }
```

#### 9. setUsingAsCollateral for one reserve doesn't affect others ✔

```
{ reserveIndexOther != reserveIndex &&
  reserveIndexOther < 128 && reserveIndex < 128 &&
  otherReserveBorrowing = isBorrowing(reserveIndexOther) &&
  otherReserveCollateral = isUsingAsCollateral(reserveIndexOther) }
    setUsingAsCollateral(reserveIndex, isUsingAsCollateral)
{ otherReserveBorrowing == isBorrowing(reserveIndexOther) &&
  otherReserveCollateral == isUsingAsCollateral(reserveIndexOther) }
```

## Pool

### Rules

#### 1. Integrity of Supply Cap - aToken supply shall never exceed the cap ✔**

```
{ reserves[asset].aToken.totalSupply() <= reserves[asset].supplyCap() }
```

** rule times-out for some functions

#### 3. Integrity of setUserEMode ✔

```
{ }
    setUserEMode(category)
{ getUserEMode() == category }
```

#### 4. User can always set own eMode to 0 ✔

```
{ }
    setUserEMode(0);
{ NOTREVERT }
```

### 5. If borrow succeeds then reserve flags are correct ✔

```
{ r = _reserves[asset].configuration && a = r.getActive() &&
  f = r.getFrozen() && p = r.getPaused() && b = r.getBorrowingEnabled() &&
  s = r.getStableRateBorrowingEnabled() }
    borrow(asset, amount, interestRateMode, referralCode, onBehalfOf)
{ a && !f && !p && b  && (interestRateMode == 1 => s) }
```

### 6. `reservesCount()` is monotonic ✔

```
{ rb = reservesCount() }
    op
{ reservesCount() >= rb }
```

### 7. Each function changes at most one configuration ✔

```
{ rb1 = getReserveConfiguration(asset1)
  && rb2 = getReserveConfiguration(asset2)}
    op
{ asset1 != asset2 => (rb1 == getReserveConfiguration(asset1)
  || rb2 == getReserveConfiguration(asset2)) }
```

### 8. Reserve id always less than counter ✔

```
 {getReserveList(i) != 0 => i < reservesCount()}
```

### 10. Bijective property of reserve id ✔

```
I.
{(i != 0 and token != 0) => (getReserveList(i) == token
    <=> getReserveDataIndex(token) == i)}
II.
{(i == 0 and token != 0) => (getReserveList(i) == token
    => getReserveDataIndex(token) == i)}
```