CERTORA

Formal Verification of Aave's protocol-V2

Summary

This document describes the specification and verification using Certora Prover of Aave's protocol-V2. The work was undertaken from August 2nd through October 29th, 2020. The latest commit that was reviewed and run through the Certora Prover was 750920303e33b66bc29862ea3b85206dda9ce786.

The scope was the Aave Protocol, a decentralized, open-source, and non-custodial money market protocol. Depositors can deposit assets into the protocol and earn interest on their deposits without relying on middlemen. The main contracts considered were the LendingPool, AToken, StableDebtToken, and VariableDebtToken. In addition, the UserConfiguration library was verified independently of the calling contracts.

The Certora Prover proved the implementation of the Aave protocol correct with respect to the formal rules written by the Aave and the Certora teams. During the verification process, the Certora Prover and the team's manual review discovered a number of bugs in the code listed in the table below in Section 1.1. All the high-severity issues were promptly corrected, and the fixes were verified to satisfy the specifications up to limitations of the Certora Prover. These limitations are currently being handled by the Certora development team. Section 2 formally defines high level specifications of the protocol. All the rules are publically available in a public github: https://github.com/aave/protocol-v2/tree/master/specs

List of Main Issues Discovered

Issue	**Rule broken**	**Description**	**Severity**	**Mitigation**
Loss of user's assets	Total assets of user	The system does not return excess payment in repay and deposit transactions	High	The ETH specific code has been removed in favor of a pull payment strategy using ERC20

Lock of assets	Deposit is always possible	Due to chains of links in redirection of interest, user's transaction reverts without an option to remove the chain and their assets are locked	High	Postpone redirection feature to a new version
System loss of assets	Additive burn No gain due to redirection	Due to a missing update to a user's time-tracking property, a user burns his entire balance, but is left with some balance	High	Postpone redirection feature to a new version
Increase in user debt	Additive repay	Due to a missing update to a user's time-tracking property, a user pays his debt, but in fact his debt grows	High	Update user's property
Lock of assets	balanceOf should never revert	Due to division by zero, balanceOf reverts	High	Different calculation of accrued interest
Overestimating system's assets		When repaying, the system neglects the amount that is going to be reimbursed when computing the new interest rate	High	Fixed computation of interest rate
System loss of assets, user gain of assets	Additive burn (AToken)	Due to rounding in conversions to Atoken, if the conversion rate is high enough, one can withdraw a small amount that will result in the system transferring underlying tokens but burning zero ATokens of the user's account	Medium	Fixed to avoid transfer on burn of zero ATokens
System loss of assets, user gain of assets, user loss of assets	Additive mint (Stable debt token)	Due to rounding in conversions to stable debt token, if the conversion rate is high enough, one can deposit a small amount that will result in the system transferring underlying tokens but minting debt tokens to the user's account	Medium	Fixed to avoid transfer on mint/burn of zero stable debt tokens
Reduce health factor,	Valid operation	In case of a very low health factor, due to having the bonus as a fixed	Medium	Known issue from V1.

System loss of assets	with respect to health factor	percentage, the liquidation process may result in an even worse health factor. The system may lose assets		Unlikely in practice, as exploit requires drastic changes to token prices
Incorrect totalSupply reported	Integrity of totalSupply (DebtToken)	Due to missing interest accumulation, total supply reported is not the sum of all balances	Medium	Fixed to include interest accumulated
Assets locked as collateral can be withdrawn	Valid operation with respect to health factor	If the system changes the flag indicating if a token is used as collateral, one can withdraw regardless of his health factor	Low	System controlled functionality
Redundant line		Coding in LendingPool:redeemUnderlying	Low	Fixed

Notations

1. ✔ indicates the rule is formally verified on the latest commit. We write ✔* when the rule was verified on a simplified version of the code that ignores rounding errors.
2. ✍ indicates the rule is not yet formally specified.
3. 🔁 indicates the rule is only applicable to a future version of the code with redirections.
4. We define ε to be the allowed rounding error and enforce a global invariant that ε ≤ normalizedIncome()/ray.
5. We use **|.|** to denote the absolute value of a numeric value.
6. We use Hoare triples of the form {p} C {q}, which means that if program C executes starting in any state satisfying p, then it will end in a state satisfying q. In Solidity, p is similar to require and q is similar to assert. The syntax {Pre} $P_1 \sim P_2$ {Post} is a generalization of Hoare rules, called relational properties. Pre is a requirement on the states before P1 and P2 and {Post} describes the states after their executions. Notice that P1 and P2 operate in different states. As a special case, $P_1 \sim_g P_2$, where g is a getter function (e.g., getBalance()), indicating that P_1 and P_2 result in states in which the value of g() is the same.
   In a hoare triple {p} $P_1 \sim P_2$ {q}, we may refer to the value of a getter g() after P_1 or P_2 by the notation of $g_1()$ or $g_2()$, respectively.

CERTORA

Verification of AToken

Functions:

totalSupply() : uint
Returns the total supply.

balanceOf(address u) : uint
Returns the balance of address ***u*** with interest and redirected interest.

normalizedIncome() : uint
Returns the reserve's current normalized income index in high precision format (ray).

burn(address u, address t, uint x) : bool
Returns true if successfully burns ***x*** ATokens from the account of ***u*** and sends ***x*** underlying tokens to ***t***.

transfer(address u, address t, uint x)
Transfers ***x*** ATokens from the account of ***u*** to ***t***.

Properties:

1. **Integrity of totalSupply ✔***
   TotalSupply is the sum of all users' balances.

   `totalSupply() = ` $\Sigma_{\text{address } u}$ `balanceOf(u)`

2. **Integrity of mint ✔**
   Minting an amount of *x* tokens for user *u* increases their balance by *x*, up to rounding errors (ε).
   `{ b = balanceOf(u) } mint(u,x) {` $b + x - \varepsilon \le$ `balanceOf(u)` $\le b + x + \varepsilon$ `}`

3. **Additivity of mint ✔***
   Minting is additive, i.e., it can be performed either all at once or in steps.

   `mint(u,x); mint(u,y) ~ mint(u,x+y) { |` $\text{balanceOf}_1(u) - \text{balanceOf}_2(u)$ `|` $\le 3\varepsilon$ `}`

4. **Integrity of transfer ✔**
   Transferring ***x*** tokens from user ***u*** to user ***u'*** changes the balance of ***u*** and ***u'*** as expected and does not change the underlying asset balance of any user or of the system.

```
 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 ✔***
   Transfer is additive, i.e., it can be performed either all at once or in steps.
   `transfer(u, u’, x); transfer(u, u’, y) ~ transfer(u, u’, x+y)`

6. **Integrity of burn ✔**
   Transfer of ***x*** amount of tokens from a user ***u*** to a receiver user ***u'***

```
{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 ✔***
   Burning is additive, i.e., it can be performed either all at once or in steps.

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

8. **Revert characteristic of burn ✔***

Burning of ***x*** token of user ***u*** when transferring underlying token to receiver ***t*** succeeds when the following holds:

I. *msg.sender* is the LendingPoll contract and no value sent,
II. *totalSupply* in AToken is more than *amount* to burn,
III. The system has enough underlying tokens to transfer
IV. user ***u*** which burns his asset is valid and has enough balance
V. Receiver *t* is valid and transferring him underlying token will not overflow

```
{ msg.value = 0 ∧ msg.sender = LendingPool ∧
  totalSupply() ≥ x ∧
  underlyingAssetBalanceOf(AToken) > x  ∧
  u ≠ 0 ∧ balanceOf(u) < x
  t ≠ 0 ∧ underlyingAssetBalanceOf(t) + x< MAXINT }
       r = burn(u, to, x)
{ r }
```

Verification of StableDebtToken

9. **Integrity of getUserLastUpdated ✔***
   The time of the last update to a user cannot lie in the future.

   ```
   getUserLastUpdated(u) ≤  block.timestamp;
   ```

10. **Integrity of totalSupply ✔***
    TotalSupply is the sum of all users' balances.

    totalSupply(t) = $\Sigma_{\text{address } u}$ balanceOf(u,t)

    This is proven by showing properties 11 and 12 below:

11. **No two balances updated ✔***
    Each possible operation changes the balance of at most one user.

12. **Integrity balance and total supply ✔***
    Check that the changes to total supply are coherent with the changes to balance for each operation.

13. **Integrity of mint ✔***
    Minting an amount of ***x*** tokens for user ***u*** increases their balance by ***x***, up to rounding errors.

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

14. **Additivity of mint ✔***
    Minting is additive, i.e., it can be performed either all at once or in steps.

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

15. **Integrity of burn✔***
    Transfer of *x* amount of tokens from user ***u*** where receiver is user *u'*

    ```
    {bu = balanceOf(u) }
          burn(u, u', x)
    {balanceOf(u) = bu - x }
    ```

CERTORA

16. Additivity of burn ✔*

Burning is additive, i.e., it can be performed either all at once or in steps.

```
burn(u, u’, x); burn(u, u’, y) ~ burn(u, u’, x+y)
```

17. Invertibility of mint and burn ✔*

Minting and burning are inverse operations.

```
{bu = balanceOf(u) }
       mint(u,x); burn(u, u, x)
{balanceOf(u) = bu }
```

Verification of VariableDebtToken

18. **Integrity of totalSupply ✔***

    TotalSupply is the sum of all users' balances.

    `totalSupply(t) = Σ` $_{\text{address } u}$ `balanceOf(u,t)`

    This is proven by showing properties 19 and 20 below:

19. **No two balances updated ✔***

    Each possible operation changes the balance of at most one user.

20. **Integrity balance and total supply ✔***

    The changes to total supply are coherent with the changes to balance for each operation.

21. **Integrity of mint ✔***

    Minting an amount of ***x*** tokens for user ***u*** increases their balance by ***x***, up to rounding errors.

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

22. **Additivity of mint ✔***

    Minting is additive, i.e., it can be performed either all at once or in steps.

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

23. **Integrity of burn ✔***

    Transfer of ***x*** amount of tokens from user ***u*** where receiver is user ***u'***

    ```
    { bu = balanceOf(u) }
          burn(u, u', x)
    { balanceOf(u) = bu - x }
    ```

24. **Additivity of burn ✔***

    Minting is additive, i.e., it can be performed either all at once or in steps.

    ```
    burn(u, u', x); burn(u, u', y) ~ burn(u, u', x+y)
    ```

25. **Inverse of mint and burn ✔***

Minting and burning are inverse operations.

```
{ bu = balanceOf(u) }
      mint(u,x); burn(u, u, x)
{ balanceOf(u) = bu }
```

CERTORA

Verification of UserConfiguration Library

This library implements a bitmap logic to handle the user configuration, it stores whether a user has borrowed or/and deposited for collateral for each possible asset.

Functions:

setBorrowing(address u, uint256 reserveId, bool val)
Sets that user ***u*** borrowing flag for ***reserveId*** to ***val***

setUsingAsCollateral(address u, uint256 reserveId, bool val)
Sets that user ***u*** is using ***reserveId*** as collateral to ***val***

isUsingAsCollateralOrBorrowing(address u , uint256 reserveId) : bool
Returns true if the user ***u*** has been using a reserve ***reserveId*** for borrowing or as collateral

isBorrowing(address u , uint256 reserveId) : bool
Returns true if user ***u*** has been using a ***reserveId*** for borrowing

isUsingAsCollateral(address u , uint256 reserveId) : bool
Returns true if user ***u*** has been using a ***reserveId*** as collateral

isBorrowingAny(address u) : bool
Returns true if user ***u*** has been borrowing any reserve

isEmpty(address u) : bool
Returns true if user ***u*** has been borrowing or using as collateral any reserve

Properties:

26. Integrity of getters ✔

For each user ***u*** and reserve id ***r***:

```
 I.   isEmpty(u) ⇒
      (¬isBorrowingAny(u) ∧ ¬isUsingAsCollateralOrBorrowing(u,r))

 II.  (isBorrowingAny(u) ∨ isUsingAsCollateral(u,r)) ⇒ ¬isEmpty(user)

 III. isBorrowing(u,r) ⇒ isBorrowingAny(u)

 IV.  (isUsingAsCollateral(u,r) ∨ isBorrowing(u,r)) ⇔
      isUsingAsCollateralOrBorrowing(u,r)
```

27. Integrity of setBorrowing ✔

When setting the flag of user ***u*** borrowing from reserve ***r***, the value is updated as expected.

```
{ r < 128 }
      setBorrowing(u, r, b)
{ isBorrowing(u,r) = b }
```

28. No effect on other on setBorrowing ✔

When setting the flag of user ***u*** borrowing from reserve ***r***, the flags concerning reserve ***r`*** are not changed.

```
{ r` ≠ r ∧ r` < 128 ∧  r < 128 ∧ borrow = isBorrowing(u, r`) ∧
collateral = isUsingAsCollateral(u,r`) }
      setBorrowing(u, r, b)
{ borrow = isBorrowing(u, r`) ∧
collateral = isUsingAsCollateral(u,r`) }
```

29. Integrity of setUsingAsCollateral ✔

When setting the flag of user ***u*** using as collateral reserve ***r***, the value is updated as expected.

```
{ r < 128 }
      setUsingAsCollateral(u,r,b)
{ isBorrowing(u,r) = b }
```

30. **No effect on other on setUsingAsCollateral ✔**

When setting the flag of user ***u*** using as collateral reserve ***r***, the flags concerning reserve ***r`*** are not changed.

```
{ r` ≠ r ∧ r` < 128 ∧  r < 128 ∧ borrow = isBorrowing(u, r`) ∧
collateral = isUsingAsCollateral(u,r`) }
      setUsingAsCollateral(u, r, b)
{ borrow = isBorrowing(u, r`) ∧
collateral = isUsingAsCollateral(u,r`) }
```

CERTORA

Verification of LendingPool

Functions:

Properties of the system that have no side-effect (view functions)

getAToken(Token t) : AToken
Returns the AToken associated with a reserve of token ***t***.
i.e., reserves[**t**].aTokenAddress

getStableDebtToken(Token t) : DebtToken
i.e., reserves[**t**].stableDebtTokenAddress

getVariableDebtToken(Token t) : DebtToken
i.e., reserves[**t**].variableDebtTokenAddress

healthFactor(address a) : uint
Returns the health factor of user ***a***

totalDebt(address a) : uint
totalCollateralETH(address a) : uint

token.balanceOf(address a) : uint
Returns the balanceOf address ***a*** in ***token***

token.totalSupply() : uint
Returns the total supply of ***token***.

Operations:

All operations return true on successful execution of the operation (i.e., did not revert).
These operations have side effects on the state of the system.

deposit(Actor a, Token t, uint x, Actor b) : bool
Actor ***a*** deposits on behalf of actor ***b***, in the reserve of ***x*** amount of token ***t***.
Returns true on successful deposit (i.e., did not revert)

withdraw(Actor a, Token t, uint x) : bool
Actor ***a*** withdraws ***x*** amount of token ***t***

borrow(Actor a, Token t, uint x, Actor b) : bool
Actor ***a*** borrows on behalf of actor ***b*** amount of ***x*** tokens ***t***

repay(Actor a, Token t, uint x, Actor b) : bool
Actor ***a*** repays on behalf of actor ***b*** amount of ***x*** tokens ***t***

swap(Actor a, Token t) : bool
Swap the rate of debt of Actor ***a*** for token ***t***

rebalanceStableBorrowRate(Token t, Actor a) 🔁
FlashLoan 🔁
LiquidationCall 🔁

Properties:

31. Integrity of a reserve ✔*

The balance of a reserve for asset ***t*** is at least the sum of all deposits minus the borrowed.

```
 I.    t.balanceOf(getAToken(t))  ≥  getAToken(t).totalSupply() -
       (  getStableDebtToken(t).totalSupply()   +
          getVariableDebtToken(t).totalSupply() )

II.    reserve.liquidityIndex ≤ reserve.getNormalizedIncome() 🔁
```

32. Total assets of user is preserved ✔*

The total assets with regard to some (external) token ***t*** is preserved. Need to take into account that operation can change the balance of two actors

We define total assets of user ***u*** with respect to token ***t***
```
totalAssets(t, u) ≡ t.balanceOf(u) + getAToken(t).balanceOf(u) -
(getStableDebtToken(t).balanceOf(u) + getVariableDebtToken(t).balanceOf(u))

{ x = totalAssets(t, a)  }
      op;
{ totalAssets(t, a) = x}
```

Where op is an operation that affects the asset of at most one user

For case where *op* can change the balance of actor ***a*** performing on operation behalf of actor ***b*** the property is:

```
{ x = totalAssets(t, a)  + totalAssets(t, b) }
      borrow(a, t, x, b) | deposit(a, t, x, b) ;
{ assets(t, a)  + totalAssets(t, b)  = x }
```

33. One can always deposit 🔁

assuming the system is in active state, one can always deposit

```
{ isActive(t)  ∧ ¬isFreeze(t) ∧ x ≠ 0} ret = deposit(u, t, x) { ret }
```

34. Revert characteristic of deposit 🔁

Deposit fails if the reserve is in inactive or freeze state or the user attempts to deposits 0 tokens

```
{ ¬isActive(t)  ∨ isFreeze(t) ∨ x = 0} ret = deposit(u, t, x) { ¬ret }
```

35. Integrity of deposit ✔*

When actor ***u*** deposits ***x*** tokens of asset ***t*** on behalf of actor ***b*** (can be ***a****)*
The asset balance of ***u*** is decreased and the aToken of **b** is increased.

```
{ t_ = t.balanceOf(u)  ∧  a_ = getAToken(t).balanceOf(b) }
      deposit(u, t, x, b );
{ t.balanceOf(u) = t_ - x ∧ getAToken(t).balanceOf(b) = a_ + x   }
```

36. Integrity of borrow ✍

When actor ***u*** deposits ***x*** tokens of asset ***t*** on behalf of actor ***b*** (can be ***a****)*
The asset balance of ***u*** is increased and the debt of **b** is increased.

```
{ t_ = t.balanceOf(u)  ∧  a_ = totalDebt(b) }
      borrow(u, t, x, b );
{ t.balanceOf(u) = t_ + x ∧  totalDebt(b) = a_ + x   }
```

37. Revert characteristic of borrow 🔁

38. Total assets and debts of user is not influenced by others ✍

39. Operations at one interest rate does not change the other interest rate type debt ✍

40. Variable debt token is not influenced by stable rate operations ✍

41. **Each operation affects at most one reserve** ✍

42. **Valid Operations with respect to health factor** ✍

```
{ healthFactor(a) > 1 } Op_a {  healthFactor(a) >= 1 }

{ h = healthFactor(a) < 1 } Op {  healthFactor(a) < h }
```

43. **Inverse operations** ✍

    Deposit; Op ; Redeem ~ skip
    Borrow; Op ;Repay ~ skip
    Where op is any operation by other users or specific operation by the user.
    Here we mean that these operations return the same values that are important to us.

44. **Additivity properties at the same timestamp** ✍

    Deposit(x); Deposit(y) ~ Deposit(x+y) ✔*
    Redeem(x);Redeem(y) ~ Redeem(x+y)
    Borrow(x);Borrow(y) ~ Borrow(x+y)
    Repay(x);Repay(y) ~ Repay(x+y)

CERTORA

Properties regarding redirection of interest

Properties that have been excluded for the current release of the AToken contract.

Functions:

getInterestRedirectionAddress(address u) : address
Returns the address to which address ***u*** redirects the interest to.

getRedirectedBalance(address u) : uint
Returns the total amount of balance redirected to address ***u*** from all redirecting users.

Properties:

45. Safety of redirection 🔁

One can not redirect interest to himself.
`getInterestRedirectionAddress(u) ≠ u`

46. Integrity when no redirection 🔁

If account ***u*** does not redirect and has no redirection than the balance of equals the principle balance.

```
( getInterestRedirectionAddress(u) = 0 ∧  getRedirectedBalance(u) = 0  ) ⇒
balanceOf(u, t) = _calculateCumulatedBalance(u, principalBalanceOf(u), t)
```

47. Integrity of interest redirection 🔁

The balance of a user ***v*** that has redirection from user ***u*** but no principalBalance is the interest of user ***u***

```
( amount = principalBalanceOf(u) = getRedirectedBalance(v) ∧
principalBalanceOf(v) = 0 ) ⇒
_calculateCumulatedBalance(u, amount , t) - amount = balanceOf(v, t)
```

48. Integrity of interest redirection on transactions 🔁

The balance of a user ***v*** that has redirection from user ***u*** but no principalBalance is the interest of user ***u***.

```
{ amount = principalBalanceOf(u) = getRedirectedBalance(v) ∧
principalBalanceOf(v) = 0 ∧ t’ ≥ t }  Op_t {
_calculateCumulatedBalance(u, amount , t’) - principalBalanceOf(u) =
balanceOf(v, t’) }
```

Where Op_t is an operation performed at time t beside: redirectInterestStream,...

CERTORA

49. **No gain due to redirection of interest** 🔁

The total balance of two users, say ***u_1*** and ***u_2***, is not affected if one redirects the interest to the other.

```
Op ; x = balanceOf(u_1, s) + balanceOf(u_2, s) /*scenario 1*/
 ~
redirectInterestStream(u_1,u_2); Op ;
y = balanceOf(u_1, s) + balanceOf(u_2, s)  /*scenario 2*/
{ x == y }
```

CERTORA

Properties suggested by Aave

Global invariants

- P1: Whenever a user transfers aTokens from A -> B, the balanceOf(aToken) of the underlying asset transferred does not change (`Integrity of the system`)
- P2: Given an asset A with Bsa = total borrowed at stable of A, any action of borrowing/repaying/liquidating at variable rate does not change Bsa
- P3: Given an asset A with Bva = total borrowed at variable of A, any action of borrowing/repaying/liquidating at stable rate does not change Bva.
- P4: Given an user U and a set of borrowed assets A, with Bn = {Ba1, Ba2... Ban} the set of borrow balances for each asset a ∈ A, if U borrows another asset an+1, Ban+1 is equal to the amount of the asset an+1 being borrowed and all the balances B1...Bn are unchanged (explanation: borrowing a new assets doesn't affect other users positions)
- P5: Given an user U and a set of borrowed assets A, with Bn = {Ba1, Ba2... Ban} the set of borrow balances for each asset a ∈ A, if U repays an asset repays an amount M for any asset ax ∈ A, then Bax = Bax - M and all the other balances are unaffected (explanation: repaying a loan does not affect other loans)
- P6: On any action on a specific asset:
  - If the asset is not being borrowed at a variable rate, the variable borrow index doesn't change
  - If the asset is being borrowed at a variable rate, the variable borrow index increases, unless the following happens:
    - The index has been updated by another action in the same block
    - The interest accrued is so small that is below 10^-27

    In these conditions, the variable borrow index remains the same

- P7: On any action on a specific asset:
  - If the asset is not being borrowed, the liquidity index doesn't change
  - If the asset is being borrowed, the liquidity index increases, unless the following happens:
    - The index has been updated by another action in the same block
    - The interest accrued is so small that is below 10^-27

In these conditions, the variable borrow index remains the same

State transitions

Deposit()

- P6: It's not possible to deposit in an inactive reserve.
- P7: It's not possible to deposit in a frozen reserve.
- P8: It's not possible to deposit 0 amount.
- **P9: When a user deposits X amount of an asset, his collateral in the system is increased by X.** Integrity of deposit
- **P10: When a user deposits X amount of an asset, he receives exactly X amount of the corresponding aToken.**Integrity of deposit
- **P11: When a user deposits X amount of an asset, the protocol balance on that asset is increased by X.** Integrity of the system **+** Integrity of deposit
- P12: On any deposit, if the asset is being borrowed at variable(totalSupply(variableDebtToken) > 0 for the particular asset), the variable borrow rate goes down as long as the deposited amount isn't so small that the impact on the variable borrow rate is below the margin of error (10^-27)
- P13: On any deposit, if the asset is being borrowed at stable (totalSupply(stableDebtToken) > 0 for the particular asset), the stable borrow rate goes down as long as the deposited amount isn't so small that the impact on the variable borrow rate is below the margin of error (10^-27)
- P14: On any deposit, if the asset is being borrowed (totalSupply(stableDebtToken) + totalSupply(variableDebtToken) > 0 for the particular asset), the liquidity rate goes down as long as the deposited amount isn't so small that the impact on the liquidity rate is below the margin of error 10^-27.

Borrow()

- P15: It's not possible to borrow in an inactive reserve.
- P16: It's not possible to borrow in a frozen reserve.
- P17: It's not possible to borrow on a reserve disabled for borrowing.
- P18: It's not possible to borrow 0 amount.
- P19: It's not possible to borrow with an interest rate mode that is different than 1 (stable) or 2 (variable)
- P20: It's not possible to borrow more than the available liquidity in the reserve.

- P21: It's not possible to borrow at a stable rate in a reserve where the stable rate is not enabled.
- P22: It's not possible to borrow at a stable rate more than the percentage over the available liquidity defined by the parameters returned by getMaxStableRateBorrowSizePercent()
- **P23: A borrower whose health factor (HF) is below 1 can't borrow. Valid Operation with respect to health factor**
- **P24: A borrower can only borrow up to the amount which would set the HF to 1 (alternatively written, after a borrow action the HF of the borrower is always > 1) Valid Operation with respect to health factor**
- P25: Whenever a user borrows an asset at a variable rate, the balanceOf(user) on the VariableDebtToken increases by the amount borrowed
- P26: Whenever a user borrows an asset at a stable rate, the balanceOf(user) on the StableDebtToken increases by the amount borrowed
- **P27: When a user borrows X amount of an asset they will receive exactly X amount of that asset.Integrity of borrow**
- **P28: When a user borrows X amount of an asset, the protocol balance of that asset is decreased by X.Integrity of borrow**
- P29: On any borrow, the variable rate goes up as long as the borrowed amount isn’t so small that the impact on the variable borrow rate is below the margin of error (10^-27)
- P30: On any borrow, the stable rate goes up as long as the borrowed amount isn’t so small that the impact on the variable borrow rate is below the margin of error (10^-27)

withdraw()

- P31: It's not possible to withdraw from an inactive reserve.
- P32: It's not possible to withdraw a 0 amount.
- **P33: A user can't** withdraw **more than his current aToken balance.Total assets of user is preserved**
- **P34: A user can't** withdraw **more than the liquidity available in the reserve (the total deposited - the borrowed)**
- **P35: A user can't** withdraw **an amount that would set his Health Factor below 1e18 (alternatively written, after a redeem() the HF of the user can't be < 1e18)**
- P36: After withdraw, the withdrawn amount is subtracted from the user aToken balance

- P37: After withdraw, the user receives the exact withdrawn amount in the underlying token.
- **P38: Whenever a user withdraws, the balanceOf(aToken) for the underlying asset decreases of the amount withdrawn**
- P39: On any withdraw, if the asset is being borrowed at variable(totalSupply(variableDebtToken) > 0 for the particular asset), the variable borrow rate goes up as long as the withdrawn amount isn't so small that the impact on the variable borrow rate is below the margin of error (10^-27)
- P40: On any withdraw, if the asset is being borrowed at stable (totalSupply(stableDebtToken) > 0 for the particular asset), the stable borrow rate goes up as long as the withdrawn amount isn't so small that the impact on the variable borrow rate is below the margin of error (10^-27)
- P41: On any deposit, if the asset is being borrowed (totalSupply(stableDebtToken) + totalSupply(variableDebtToken) > 0 for the particular asset), the liquidity rate goes up as long as the withdrawn amount isn't so small that the impact on the liquidity rate is below the margin of error 10^-27.

Critical properties to ensure no funds can be trivially stolen

- **P9: When a user deposits X amount of an asset, his collateral in the system is increased by X.** Integrity of deposit
- **P10: When a user deposits X amount of an asset, he receives exactly X amount of the corresponding aToken.**Integrity of deposit
- **P11: When a user deposits X amount of an asset, the protocol balance on that asset is increased by X.** Integrity of the system **+** Integrity of deposit
- **P23: A borrower whose health factor is below 1 can't borrow. Valid Operation with respect to health factor**
- **P24: A borrower can only borrow up to the amount which would set the HF to 1 (alternatively written, after a borrow action the HF of the borrower is always > 1) Valid Operation with respect to health factor ?**
- P25: Whenever a user borrows an asset at a variable rate, the balanceOf(user) on the VariableDebtToken increases by the amount borrowed
- P26: Whenever a user borrows an asset at a stable rate, the balanceOf(user) on the StableDebtToken increases by the amount borrowed
- **P27: When a user borrows X amount of an asset they will receive exactly X amount of that asset.Integrity of borrow**

- **P28: When a user borrows X amount of an asset, the protocol balance of that asset is decreased by X.Integrity of borrow**

Disclaimer