---

breaks: false

Formal Verification Of AAVE V2 AStETH

Summary

This document describes the specification and verification of LIDO's AAVE V2 AStETH using the Certora Prover. The work was undertaken from June 5th to June 26th 2022 after the contract was deployed and listed on the AAVE V2 platform. The latest commit reviewed and ran through the Certora Prover was f3b54f82.

The scope of our verification was the `AStETH.sol` contract.

The verification of the contract was a joined effort by Certora and the AAVE community - writing formal specifications and manually auditing the code. All rules written by the community were reviewed by Certora, and relevant rules were adapted and added to the final specification side by side with the rules provided by Certora.

During this verification process, no issues were discovered. All the rules and specification files are publicly available and can be found in LIDO Finance's fork to the AAVE V2 protocol.

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.

Summary of Formal Verification

Overview of AStETH

AStETH is AAVE's interest bearing token for stETH, which was created by Lido finance. The main goal of integration is to provide the ability to deposit stETH into AAVE and allow it to be used as collateral. The motivation behind this design is to encourage using stETH as collateral rather than borrowing it. stETH is pegged steadily to ETH, so using it as collateral involves low liquidation risks.

The stETH is implemented as a rebasing token. In normal conditions, balances of users update once per day with Oracle report. Under the hood, stETH stores balances of users as holder's shares in the total amount of Ether controlled by the Lido protocol. When stETH rebases, AStETH rebases along with it at a 1:1 ratio. To make rebase profit accountable, `AStETH` introduces an additional index - stETH rebasing index. The stETH rebasing index tracks the income from rebases of stETH token in time.

Similarly to regular ATokens, AStETH's value is pegged to the value of the corresponding deposited asset at a 1:1 ratio and can be safely stored, transferred, or traded. All interest collected by the AStETH reserve (from rebasing and AAVE income) is distributed to ATokens holders directly by continuously increasing their wallet balance (in case of negative rebases of stETH it might decrease).

Simplifying Assumptions

- We unroll loops. Violations that require a loop to execute more than once will not be detected.

Due to computational difficulties, we applied the following underlying simplifications:

- We fixed the liquidity index to 1 by summarizing the rayMath arithmetics to the identity operator.
- We fixed stEthRebasingIndex to 2 RAY in order to consolidate the rebasing ratio to 2:1.

By applying these simplifications we limit the rebasing index to a single value, which reduces the generality of our specification (fixing 1 parameter). As a result, violations that may occur due to negative rebasing, index values with greater orders of magnitude, and dynamic ratio changes may be overlooked.

Notations

✅ indicates the rule is formally verified on the latest reviewed commit. We write ✅* when the rule was verified on the simplified assumptions described above in "Assumptions and Simplifications Made During Verification".

❌ indicates the rule was violated under one of the tested versions of the code.

🔁 indicates the rule is timing out.

We use Hoare triples of the form {p} C {q}, which means that if the execution of program C starts in any state satisfying p, it will end in a state satisfying q. This logical structure is reflected in the included formulae for many of the properties below. Note that p and q here can be thought of as analogous to require and assert in Solidity.

The syntax {p} (C1 ~ C2) {q} is a generalization of Hoare rules, called relational properties. {p} is a requirement on the states before C1 and C2, and {q} describes the states after their executions. Notice that C1 and C2 result in different states. As a special case, C1 ~op C2, where op is a getter, indicating that C1 and C2 result in states with the same value for op.

Properties

Community Rules

The following rules were contributed by the AAVE community as part of the Certora-AAVE grants program. All rules were processed and reviewd by Certora. Some of the rules were adapted and unified in order to increase rule coverage and remove duplications.

Contributors:

- parth-15
- 0xDatapunk
- fyang1024.

---

1. **burnAdditive** ✅

burn operation is additive.

```
    burn(user, receiverOfUnderlying, amount1, index)
    burn(user, receiverOfUnderlying, amount2, index)
{
    balanceOfUnderlyingTokenAfterSeparate := UNDERLYING_ASSET.balanceOf(receiverOfUn
    balanceSeparate := balanceOf(user),
    totalSupplySeparate := totalSupply()
}
    ~
    burn(user, receiverOfUnderlying, amount1 + amount2, index)
{
    balanceOfUnderlyingTokenAfterSeparate == UNDERLYING_ASSET.balanceOf(receiverOfUn
    balanceSeparate == balanceOf(user),
    totalSupplySeparate == totalSupply()
}
```

> This rule was contributed by parth-15 and 0xDatapunk

2. **transferUnderlyingToAdditivity** ✓

   transferUnderlyingTo operation is additive

```
    transferUnderlyingTo(target, amount1)
    transferUnderlyingTo(target, amount2)


{
    _balance := UNDERLYING_ASSET.balanceOf(target)
}
    ~
    transferUnderlyingTo(target, amount1 + amount2)
{
    _balance == UNDERLYING_ASSET.balanceOf(target)
}
```

> This rule was contributed by 0xDatapunk

3. **mintAdditive** ✓

   mint operation is additive

```
    mint(user, amount1, index)
    mint(user, amount2, index)
{
    balanceOfUnderlyingTokenAfterSeparate := UNDERLYING_ASSET.balanceOf(receiverOfUn
    balanceSeparate := balanceOf(user),
    totalSupplySeparate := totalSupply()
}
    ~
    mint(user, amount1 + amount2, index)
{
    balanceOfUnderlyingTokenAfterSeparate == UNDERLYING_ASSET.balanceOf(receiverOfUn
    balanceSeparate == balanceOf(user),
    totalSupplySeparate == totalSupply()
}
```

> This rule was contributed by parth-15 and 0xDatapunk

4. **mintPreservesUnderlyingAsset** ✓

   mint operation do not affect the totalSupply or UserBalanceOfUnderlyingAsset

```
{
    totalSupplyOfUnderlyingAssetBefore := UNDERLYING_ASSET.totalSupply(),
    balanceOfUnderlyingAssetBefore := UNDERLYING_ASSET.balanceOf(user)
}
   mint(user, amount, index)
{
    totalSupplyOfUnderlyingAssetBefore == UNDERLYING_ASSET.totalSupply(),
    balanceOfUnderlyingAssetBefore == UNDERLYING_ASSET.balanceOf(user)
}
```

> This rule was contributed by parth-15

5. **proportionalBalancesAndTotalSupplies** ✓

   scaledBalance / internalBalance should be always equal to scaledTotalSupply / internalTotalSupply, that is, internalBalance * scaledTotalSupply should be always equal to scaledBalance * internalTotalSupply. Similarly, scaledBalance * totalSupply should be always equal to balance * scaledTotalSupply, and internalBalance * totalSupply should be always equal to balance * internalTotalSuppl.

```
{
    _ATokenInternalBalance * _ATokenScaledTotalSupply == _ATokenScaledBalance * _ATo
    _ATokenScaledBalance * _ATokenTotalSupply == _ATokenBalance * _ATokenScaledTotal
    _ATokenInternalBalance * _ATokenTotalSupply == _ATokenBalance * _ATokenInternalT
}
    <invoke any method f>
```

```
{
    ATokenInternalBalance_ * ATokenScaledTotalSupply_ == ATokenScaledBalance_ * ATok
    ATokenScaledBalance_ * ATokenTotalSupply_ == ATokenBalance_ * ATokenScaledTotalS
    ATokenInternalBalance_ * ATokenTotalSupply_ == ATokenBalance_ * ATokenInternalTo
}
```

> This rule was contributed by fyang1024

6. **equivalenceOfMint** ✓

   Minting to RESERVE_TREASURY_ADDRESS and invoking mintToTreasury method should be equivalent.

```
    mintToTreasury(amount, index)
{
    _ATokenInternalBalance := internalBalanceOf(RESERVE_TREASURY_ADDRESS()),
    _ATokenScaledBalance := scaledBalanceOf(RESERVE_TREASURY_ADDRESS()),
    _ATokenBalance := balanceOf(RESERVE_TREASURY_ADDRESS()),
    _ATokenInternalTotalSupply := internalTotalSupply(),
    _ATokenScaledTotalSupply := scaledTotalSupply(),
    _ATokenTotalSupply := totalSupply()
}
    ~
    mint(RESERVE_TREASURY_ADDRESS(), amount, index)
{
    _ATokenInternalBalance == internalBalanceOf(RESERVE_TREASURY_ADDRESS()),
    _ATokenScaledBalance == scaledBalanceOf(RESERVE_TREASURY_ADDRESS()),
    _ATokenBalance == balanceOf(RESERVE_TREASURY_ADDRESS()),
    _ATokenInternalTotalSupply == internalTotalSupply(),
    _ATokenScaledTotalSupply == scaledTotalSupply(),
    _ATokenTotalSupply == totalSupply()
}
```

> This rule was contributed by fyang1024

7. **burnNoInterferenece** ✓

   burn operation does not change other user's balance.

```
{
    _balance := balanceOf(user2),
    _underlyingBalance := UNDERLYING_ASSET.balanceOf(receiverOfUnderlying2)

}
```

```
    burn(user1, receiverOfUnderlying, amount, index)
{
    _balance == balanceOf(user2),
    _underlyingBalance == UNDERLYING_ASSET.balanceOf(receiverOfUnderlying2)
}
```

> This rule was contributed by 0xDatapunk

8. **integrityOfTransferOnLiquidation** ✓

   When invoking `transferOnLiquidation()` The balance of a receiver should increase and the balance of a sender should decrease

```
{
    totalSupplyBefore := totalSupply(),
    balanceOfSenderBefore := balanceOf(sender),
    balanceOfReceiverBefore := balanceOf(receiver),


}
    transferOnLiquidation(sender, receiver, amount)
{
    assert e.msg.sender == LENDING_POOL,
    assert amount != 0 => balanceOfSenderAfter < balanceOfSenderBefore,
    assert amount != 0 => balanceOfReceiverAfter > balanceOfReceiverBefore,
    assert totalSupplyAfter == totalSupplyBefore
}
```

> This rule was contributed by fyang1024

Certora Rules

9. **integrityOfTransferUnderlyingTo** ✓

   The balance of a reciver in TransferUnderlyingTo() should increase by amount and the balance of a sender in TransferUnderlyingTo() should decrease by amount.

```
{
    user != currentContract,
    userBalanceBefore := UNDERLYING_ASSET.balanceOf(user),
    totalSupplyBefore :=  UNDERLYING_ASSET.balanceOf(currentContract)
}
    amountTransfered := transferUnderlyingTo(user, amount)
{
    UNDERLYING_ASSET.balanceOf(user) = userBalanceBefore + amountTransfered,
    UNDERLYING_ASSET.balanceOf(currentContract) = totalSupplyBefore - amountTransfer
}
```

10. **monotonicityOfMint** ✓

    Minting AStETH must increase the AStETH totalSupply and user's balance

```
{
    ATokenInternalBalanceBefore := internalBalanceOf(user),
    ATokenScaledBalanceBefore := scaledBalanceOf(user),
    ATokenBalanceBefore := balanceOf(user),
    ATokenInternalTotalSupplyBefore := internalTotalSupply(),
    ATokenScaledTotalSupplyBefore := scaledTotalSupply(),
    ATokenTotalSupplyBefore := totalSupply()
}
    mint(user, amount, index)
{
    ATokenInternalBalanceBefore < internalBalanceOf(user),
    ATokenScaledBalanceBefore < scaledBalanceOf(user),
    ATokenBalanceBefore < balanceOf(user),
    ATokenInternalTotalSupplyBefore < internalTotalSupply(),
    ATokenScaledTotalSupplyBefore < scaledTotalSupply(),
    ATokenTotalSupplyBefore < totalSupply()
}
```

11. **monotonicityOfBurn** ✓

    Burning AStETH must decrease the AStETH totalSupply and user's balance. It should also not decrease reciver's underlying asset.

```
{
    ATokenInternalBalanceBefore := internalBalanceOf(user),
    ATokenScaledBalanceBefore := scaledBalanceOf(user),
    ATokenBalanceBefore := balanceOf(user),
    ATokenInternalTotalSupplyBefore := internalTotalSupply(),
    ATokenScaledTotalSupplyBefore := scaledTotalSupply(),
    ATokenTotalSupplyBefore := totalSupply(),
    underlyingReciverBalance := UNDERLYING_ASSET.balanceOf(reciver)
}
    burn(user, reciver, amount, index)
{
    ATokenInternalBalanceBefore > internalBalanceOf(user),
    ATokenScaledBalanceBefore > scaledBalanceOf(user),
    ATokenBalanceBefore > balanceOf(user),
    ATokenInternalTotalSupplyBefore > internalTotalSupply(),
    ATokenScaledTotalSupplyBefore > scaledTotalSupply(),
    ATokenTotalSupplyBefore > totalSupply(),
    underlyingReciverBalanceBefore <= UNDERLYING_ASSET.balanceOf(reciver)
}
```

12. **mintBurnInvertability** ✓

    Minting and then burning has no effect within the AStETH context.

```
{
    ATokenInternalBalanceBefore := internalBalanceOf(user),
    ATokenScaledBalanceBefore := scaledBalanceOf(user),
    ATokenBalanceBefore := balanceOf(user),
    ATokenInternalTotalSupplyBefore := internalTotalSupply(),
    ATokenScaledTotalSupplyBefore := scaledTotalSupply(),
    ATokenTotalSupplyBefore := totalSupply()

}
    mint(user, amount, index)
    burn(user, reciver, amount, index)
{
    ATokenInternalBalanceBefore = internalBalanceOf(user),
    ATokenScaledBalanceBefore = scaledBalanceOf(user),
    ATokenBalanceBefore = balanceOf(user),
    ATokenInternalTotalSupplyBefore = internalTotalSupply(),
    ATokenScaledTotalSupplyBefore = scaledTotalSupply(),
    ATokenTotalSupplyBefore = totalSupply()
}
```

13. **aTokenCantAffectUnderlying** ✓

    AStETH cannot change the totalSupply of the underlying asset

```
{
    underlyingTotalSupply := UNDERLYING_ASSET.totalSupply()
}
    <invoke any method f>
{
    UNDERLYING_ASSET.totalSupply() = underlyingTotalSupply
}
```

14. **operationAffectMaxTwo** ✓

    Check that each possible operation changes the balance of at most two users

```
{
    ATokenInternalBalance1 := internalBalanceOf(user1),
    ATokenInternalBalance2 := internalBalanceOf(user2),
    ATokenInternalBalance3 := internalBalanceOf(user3),
    ATokenScaledBalance1 := scaledBalanceOf(user1),
    ATokenScaledBalance2 := scaledBalanceOf(user2),
    ATokenScaledBalance3 := scaledBalanceOf(user3),
    ATokenBalance1 := balanceOf(user1),
    ATokenBalance2 := balanceOf(user2),
    ATokenBalance3 := balanceOf(user3)
}
    <invoke any method f>
{
    ATokenInternalBalance1 = internalBalanceOf(user1) ∨ ATokenInternalBalance2 = int
    ATokenScaledBalance1 = scaledBalanceOf(user1) ∨ ATokenScaledBalance2 = scaledBal
    ATokenBalance1 = balanceOf(user1) ∨ ATokenBalance2 = balanceOf(user2) ∨ ATokenBa
}
```

15. **integrityBalanceOfTotalSupply** ✓

Check that the changes to total supply are coherent with the changes to balance

```
{
    ATokenInternalBalance1 := internalBalanceOf(user1),
    ATokenInternalBalance2 := internalBalanceOf(user2),
    ATokenScaledBalance1 := scaledBalanceOf(user1),
    ATokenScaledBalance2 := scaledBalanceOf(user2),
    ATokenBalance1 := balanceOf(user1),
    ATokenBalance2 := balanceOf(user2),
    ATokenInternalTotalSupply = internalTotalSupply(),
    ATokenScaledTotalSupply = scaledTotalSupply(),
    ATokenTotalSupply = totalSupply()
}
    <invoke any method f>
{
    ATokenInternalBalance1 ≠ internalBalanceOf(user1) ∧ ATokenInternalBalance2 ≠ int
    ATokenScaledBalance1 ≠ scaledBalanceOf(user1) ∧ ATokenScaledBalance2 ≠ scaledBal
    ATokenBalance1 ≠ balanceOf(user1) ∧ ATokenBalance2 ≠ balanceOf(user2) ⇒ (AToken

}
```

16. **atokenPeggedToUnderlying** ✓

Checks that the totalSupply of AStETH must be backed by underlying asset in the contract when deposit or withdraw is called

```
{
    _underlyingBalance >= _aTokenTotalSupply
}
    LENDING_POOL.deposit(UNDERLYING_ASSET, amount, user, referralCode);
                                OR
    LENDING_POOL.withdraw(e, UNDERLYING_ASSET, amount, user);
{
    msg.sender ≠ currentContract => underlyingBalance_ >= aTokenTotalSupply_
}
```

17. **totalSupplyIsSumOfBalances** ✓

    The AStETH totalSupply, tracked by the contract, is the sum of AStETH balances across all users.

```
totalsGhost() == internalTotalSupply()
```

18. **totalSupplyGESingleUserBalance** ✓

    The AStETH balance of a single user is less or equal to the total supply

```
totalsGhost() >= internalBalanceOf(user)
```

Note: for every function that transfers funds between 2 users within the system, we required that the initial sum of balances of the users is less than or equal to totalSupply.