Formal Verification Report of Aave-Starknet Bridge

Summary

This document describes the specification and verification of Aave-StarkNet L1-L2 Bridge using the Certora Prover. The work was undertaken from August 15th to October 5th, 2022. The latest commit reviewed and run through the Certora Prover was 8302ab87 (https://github.com/aave-starknet-project/aave-starknet-bridge/tree/8302ab87abc0a3cd992e6606543a70bc76caa351).

The scope of this verification includes a single contract - the Aave-StarkNet L1 Bridge (`Bridge.sol`) which sits on the Ethereum side of the bridge. This contract interacts both with the Aave V2 app and the L2 bridge on Starknet, whose contracts are written in Cairo. Note that **the entire L2 side of the bridge is outside the scope of this review**.

This project has been a part of a joint Certora and Aave community program. Certora provided the initial setup for writing a formal specification and contributors from the community conducted independent formal verifications of the code.

11 out of the 15 community participants submitted spec files containing formal specifications resulting in 165 properties in total. Out of the 165 correctness rules, 104 rules passed our professional review, earning grants for their authors. Selected rules written by the community are included in this report in the *Community Properties* section.

The Certora team also verified and reviewed PR#106 (https://github.com/aave-starknet-project/aave-starknet-bridge/pull/106) which includes additions and modifications to the contract, based on issues found by Certora and Peckshield.

During these verifications, the Certora Prover discovered issues in the code which are listed in the tables below.

All the rules and specification files are publicly available and can be found in Aave StarkNet Bridge repository (https://github.com/aave-starknet-project/aave-starknet-bridge).

List of Main Issues Discovered

Discovered By The Community:

Severity: Low

Found by several contributors.
The specification in this report was written by:
https://github.com/jonatascm (https://github.com/jonatascm)

Issue:	**Token address duplicates not checked on calling `initialize`**
Description:	Upon calling `initialize`, no check is performed for duplicates in the L1 and L2 token arrays. A mistake in the function input can lead to two tokens on one side of the bridge corresponding to the same token on the other side.
Property Violated:	Community rule #2: `shouldRevertInitializeTokens`
AAVE Response:	It is assumed that no duplicates are introduced by the trusted party in charge (Aave governance). In addition, validation of approval protects against that case for L1 tokens. Can be seen here (https://github.com/aave-starknet-project/aave-starknet-bridge/blob/e756f25e161614e58b9a2df21c859d2125012aa0/contracts/l1/Bridge.sol#L283).

Discovered By Certora:

Severity: Low

Issue:	**Front running of withdrawal**
Description:	After a withdrawal request is initiated on the L2 side by a user, it is possible for anyone to call `withdraw` from the L1 side on their behalf. The withdrawal payload message (L2->L1) doesn’t include the boolean `toUnderlyingAsset` which determines the type of token to be paid to the redeemer - ATokens or underlying asset. Hence, a malicious user can front-run the original redeemer and decide the type of tokens they withdraw, with no special permissions necessary.
AAVE Response:	Fixed in commit 3d00e2a (https://github.com/aave-starknet-project/aave-starknet-bridge/commit/3d00e2aa60ba45142f4cec0968997053a1ede4bf).

Severity: Low

Issue:	**Transformation from dynamic to static amounts is not precisely reversible**
Description:	When transforming an amount of ATokens to static Atokens and back, usually by deposit and subsequent withdrawal or cancellation, the returned amount can be larger than the original one, meaning the user can gain more L1 tokens than they should recieve. This is a result of rounding errors in the ray math library. We were able to prove the inconsistency for an arbitrary asset with an arbitrary liquidity index (`index`). The difference between the values is bounded by `(index/RAY+1)/2` (see rule #3).
Property Violated:	Rule #2: `dynamicToStaticInversible2` Rule #8: `cancelAfterDepositGivesBackExactAmount`
AAVE Response:	To fix this issue, a deeper change on the Aave protocol will be required. We acknowledge this issue however no action will be taken on the Bridge contract.

Severity: Informational

Issue:	**Cancelling a deposit does not check for success of deposit on L2**
Description:	Cancelling a deposit by calling `startDepositCancellation()` checks only for non-zero message count of the deposit payload hash, i.e. that the payload indeed exists. It does not, however, check for the status of the deposit payload, i.e. whether it was handled by the other side. Therefore a great deal of trust is placed on the L2 (starkNet) confirmation proofs and update mechanism. A failure to send confirmation proof for successful deposit within the predetermined 5 day delay, will enable gaining tokens on both sides by canceling a successful deposit that has not yet received a confirmation.
Property Violated:	Rule #9: `cannotCancelDepositAndGainBothTokens`
AAVE Response:	We assume that Starknet can write proofs for successful transactions within 5 days. All projects creating bridges between Ethereum and Starknet make this assumption.

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 the Aave-Starknet Bridge

The following description is taken from the Aave-Starknet Bridge repository (https://github.com/aave-starknet-project/aave-starknet-bridge/blob/main/README.md):

The bridge allows users to deposit or withdraw their ATokens, and only ATokens, on Ethereum side, then mints or burns them wrapped ATokens named `static_a_tokens` on Starknet side. `static_a_tokens` are equivalent to ATokens except that the former grow in value when the latter grow in balance.

`Bridge.sol` is the main contract on the L1 (Ethereum) side, specifically:

It handles deposit of ATokens on L1, withdrawal of `staticatokens` from L2, and update of the L2 rewards index. L1 deposits and withdrawals can be done with aToken or with their underlying asset.

Core assumptions and modeling architecture

The code of all L2 contracts is written in Cairo, which isn't currently supported by the Certora Prover. Therefore we mock all L2 contracts assuming ideal functionality. By doing so we are able to isolate the `Bridge.sol` logic and examine whether it operates correctly given perfect surroundings. In some places we also make simplified assumptions and under-approximations in order to supply partial coverage where full coverage is not achievable due to computational complexity. See the *Assumptions and Simplifications* section below for descriptions.

The main modifications to the current system are in the L2 functionality and the interchain messaging mechanism.

An interchain bridge usually operates in a 2-step mechanism, where an action is initated on one side and sends a message to the other side. The receiving end then consumes the message and invokes the appropriate action on its side.

In our mocks we bypassed the entire messaging mechanism by rewriting the cairo contracts as solidity contracts, and overriding the implementation of the send/consume functions to call the appropriate function on the other side directly in one immediate step.

The main assumption behind this modification is that messaging works properly in terms of data transfer correctness, and that no external player can fool the system by sending fake messages.

We emphasize that we did not verify the Cairo contracts at all, and that our mock implementations are often simplistic to allow interaction with `Bridge.sol` while adding minimum complexity to the verification process. For example, we assume a single liquidity index value for all `staticATokens`.

List of contracts

`BridgeHarness.sol` - inherits from `Bridge.sol` and extends its functionality. Overrides the messaging methods to bypass a messaging mechanism which currently is not supported by the Prover. The messaging methods are either empty or directly call functions on the other end of the bridge.

`BridgeL2Harness.sol` - A solidity mock of the L2 Bridge (`bridge.cairo`). Contains an interface to “receive and send messages” from the L1 Bridge. The contract simply calls functions on `BridgeHarness` and gets called by it in return to bypass the messaging mechanism. It also interacts with `staticATokens` (see below). This harness does not implement the entire functionality of the original Cairo contract.

`SymbolicLendingPoolL1.sol` - A simplified implementation of Aave’s lending pool. It includes a dedicated liquidity index for each underlying asset.

`IncentivesControllerMock_L1.sol` - A mock controller used to obtain data for AToken and retrieve reward tokens.

`DummyERC20UnderlyingA_L1.sol` , `DummyERC20UnderlyingB_L1.sol` - Instances of standard ERC20 tokens. They are used by the Prover when simulations of underlying assets in the Aave lending pool are needed.

`ATokenWithPoolA_L1.sol` , `ATokenWithPoolB_L1.sol` - Instances of ATokens, holding the relevant interface for the bridge contract. These contracts are the original implementations by Aave, now modified to contain only the necessary functions for the bridge interaction.

`StaticATokenA_L2.sol` , `StaticATokenB_L2.sol` - Instances of standard ERC20 tokens with extra functionality. These tokens are minted\burned by the bridge L2 contract after a user deposits/withdraws ATokens into/from the bridge.

`DummyERC20RewardToken.sol` - A single instance of a standard ERC20 representing a rewards token. It acts as the reward token on both sides of the bridge to ensure easy correlation of rewards between them. Burnable and mintable by L2 bridge only.

Functions - BridgeHarness.sol

initiateWithdraw_L2

Calls the `initiateWithdraw` function on the L2 Bridge. Imitates the direct call from L2.

bridgeRewards_L2

Calls the `bridgeRewards` function on the L2 Bridge. Imitates the direct call from L2.

claimRewardsStatic_L2

Calls the `claimRewards` function on the L2 Bridge. Imitates the direct call from L2.

Functions - BridgeL2Harness.sol

l2RewardsIndexSetter

Sets the value of `l2RewardsIndex` to `value` . Used by an indirect call from L1 to mock messaging of the index through the bridge.

getStaticATokenAddress

Gets the address of a staticAToken by its matching AToken address.

getRewTokenAddress

Gets the address of the reward token.

address2uint256

Converts an address to `uint256` . Used to translate addresses on L1 to L2 addresses (`uint256`).

deposit

Mints `amount` of staticATokens to the depositer. Called by `deposit` on L1 instead of sending a message.

withdraw

Burns `amount` of staticATokens for the caller and then calls withdraw on L1. Called first by `initiateWithdraw_L2` on L1.

bridgeRewards

Transfers an `amount` of reward tokens from a caller's balance on L2 to its balance on L1 and then withdraws to a recipient's wallet on L1. Only called from L1, by the function `bridgeRewards_L2` .

claimRewards

Mints deserved reward tokens for a specific staticAToken for the caller. Each staticAToken contract has a mapping that stores the rewards for each user, which can only be claimed once. After claiming their rewards, a given user can never claim another amount again for that contract (the value will be permanently set to zero). See `unclaimedRewards` mapping variable in `DummyStaticATokenImpl.sol`).

Assumptions and Simplifications Made During Verification

- We assume that none of the contracts listed above make an external call to the main contract (`Bridge.sol`).

- Due to high computational complexity, we sometimes assume a constant value for the liquidity index. We denote uses this assumption with a special mark (see below - ✔ *).

- Since liquidity indexes are expected to be in orders of RAYs (= 1e27), we assume all indexes are >= 1 RAY.

- We preserve the relation between the underlying assets, their respective AToken, and their corresponding L2 static Atokens. We assume that a triplet of tokens are correlated in the expected manner.

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

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.

Our tool uses 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.

Our tool consists of a special struct type variable called environment, usually denoted by e. This complex type includes the various block data context accessible by solidity (e.g. block.timestamp, msg.sender, msg.value etc.)
These fields are accessible via the environment variable.
When necessary to indicate, a function call in a specific environment should be noted by `@ env [environment name]`.

To generally retrieve the balance of some token for any user, we use the notation: `tokenBalanceOf(token, user)` for the token address `token` and the user address `user`. This is equivalent to the Solidity call: `IERC20(token).balanceOf(user)`.

Formal Properties Written By The Community

The following properties were written and verified by contributors from the Aave community. The verification by the community was performed on commit version 8302ab87 (https://github.com/Certora/aave-starknet-bridge/tree/8302ab87abc0a3cd992e6606543a70bc76caa351).

1. Integrity Of Approved Tokens And Token Data ✓

Checks integrity of `_approvedL1Tokens` array length.
`totalApprovedTokens` counts the number of times a memory write was done to the mapping `_aTokenData`.

Contributed by Jonatascm (https://github.com/jonatascm).

```
totalApprovedTokens == _approvedL1Tokens.length
```

2. Initialize Tokens Revert Cases ❌

Verifies if `initialize` checks for invalid arrays of `l1Tokens` and `l2Tokens` .
If the L2 tokens array contains duplicates of addresses, `initialize` must revert.

Contributed by Jonatascm (https://github.com/jonatascm).

```
{
    _aTokenData[l1TokenA].l2TokenAddress() == 0
        ∧
    _aTokenData[l1TokenB].l2TokenAddress() == 0
}
    initialize@canrevert(l2Bridge, messagingContract, incentivesController,
        [l1TokenA, l1TokenB], [l2Token, l2Token])
{
    last call reverted
}
```

3. Integrity Of Deposit Expanded ✓ *

Verifies the correctness of balances after a successful deposit[1][2].

Contributed by Jessica Pointing (https://github.com/jessicapointing).

```
{
    setupTokens(underlyingAsset, AToken, staticAToken)

    recipientUnderlyingAssetBalanceBefore = tokenBalanceOf(underlyingAsset, rec
    recipientATokenBalanceBefore = tokenBalanceOf(AToken, recipient)
    recipientStaticATokenBalanceBefore = tokenBalanceOf(staticAToken, recipient
    recipientRewardTokenBalanceBefore = tokenBalanceOf(_rewardToken, recipient)

    senderUnderlyingAssetBalanceBefore = tokenBalanceOf(underlyingAsset, e.msg.
    senderATokenBalanceBefore = tokenBalanceOf(AToken, e.msg.sender)
    senderStaticATokenBalanceBefore = tokenBalanceOf(staticAToken, e.msg.sender
    senderRewardTokenBalanceBefore = tokenBalanceOf(_rewardToken, e.msg.sender)

}

     staticAmount = deposit(AToken, l2Recipient, amount, referralCode, fromUnde

{
    recipientUnderlyingAssetBalanceAfter = tokenBalanceOfunderlyingAsset, recip
    recipientATokenBalanceAfter = tokenBalanceOf(AToken, recipient);
    recipientStaticATokenBalanceAfter = tokenBalanceOf(staticAToken, recipient)
    recipientRewardTokenBalanceAfter = tokenBalanceOf(_rewardToken, recipient)

    senderUnderlyingAssetBalanceAfter = tokenBalanceOf(underlyingAsset, e.msg.s
    senderATokenBalanceAfter = tokenBalanceOf(AToken, e.msg.sender)
    senderStaticATokenBalanceAfter = tokenBalanceOf(staticAToken, e.msg.sender)
    senderRewardTokenBalanceAfter = tokenBalanceOf(_rewardToken, e.msg.sender)

    fromUnderlyingAsset =>
        senderUnderlyingAssetBalanceAfter == senderUnderlyingAssetBalanceBefore
        senderATokenBalanceAfter == senderATokenBalanceBefore ∧
        recipientStaticATokenBalanceAfter == recipientStaticATokenBalanceBefore

    !fromUnderlyingAsset =>

        senderUnderlyingAssetBalanceAfter == senderUnderlyingAssetBalanceBefore
        senderATokenBalanceBefore - senderATokenBalanceAfter - amount <= (index
        recipientStaticATokenBalanceAfter == recipientStaticATokenBalanceBefore

    e.msg.sender ≠ recipient =>

        senderStaticATokenBalanceAfter == senderStaticATokenBalanceBefore ∧
        recipientUnderlyingAssetBalanceAfter == recipientUnderlyingAssetBalance
        recipientATokenBalanceAfter == recipientATokenBalanceBefore

    senderRewardTokenBalanceAfter == senderRewardTokenBalanceBefore

    recipientRewardTokenBalanceAfter == recipientRewardTokenBalanceBefore
}
```

Additional Formal Properties for Aave-StarkNet Bridge

The following properties were written and verified by Certora.
The verification by the community was performed on commit 58cbfbf7 (https://github.com/aave-starknet-project/aave-starknet-bridge/tree/58cbfbf75e25e31273d310aa7b617796d9ac3bf8).

1. Dynamic To Static Amount Is Inversible ✓

For every asset, lending pool, and `amount` , transforming amount from dynamic to static and back must result in the initial amount.

```
_dynamicToStaticAmount(_staticToDynamicAmount(amount, asset, lendingPool), asse
```

2. Static To Dynamic Amount Is Inversible ❌

For every asset, lending pool, and `amount` , transforming amount from static to dynamic and back must result in the initial amount.

```
_staticToDynamicAmount(_dynamicToStaticAmount(amount, asset, lendingPool), asse
```

3. Dynamic To Static Amount Is Inversible Within An Error Bound ✓

For every asset, lending pool, and `amount` , transforming amount from dynamic to static and back must result an amount within a certain error bar from the initial amount.

```
stat = _dynamicToStaticAmount(amount, asset, lendingPool)
dynm = _staticToDynamicAmount(stat, asset, lendingPool)

dynm - amount <= (indexL1/RAY() + 1)/2
```

4. Integrity Of Withdraw ✓

After a successful call to `withdraw` , the token balances must change correctly.

```
{
    setupTokens(underlying, AToken, static)
    recipient ≠ AToken
    recipient ≠ Bridge

        underlyingBalanceBefore = tokenBalanceOf(underlying, recipient)
        ATokenBalanceBefore = tokenBalanceOf(AToken, recipient)
        rewardTokenBalanceBefore = tokenBalanceOf(_rewardToken, recipient)
}

    initiateWithdraw_L2(AToken, staticAmount, recipient, toUnderlyingAsset)

{
         underlyingBalanceAfter = tokenBalanceOf(underlying, recipient)
         ATokenBalanceAfter = tokenBalanceOf(AToken, recipient)
         rewardTokenBalanceAfter = tokenBalanceOf(_rewardToken, recipient)

    toUnderlyingAsset =>
    (underlyingBalanceAfter >= underlyingBalanceBefore) ∧
    (ATokenBalanceAfter == ATokenBalanceBefore)

    ¬toUnderlyingAsset =>
    (ATokenBalanceAfter >= ATokenBalanceBefore) ∧
    (underlyingBalanceAfter == underlyingBalanceBefore)

     rewardTokenBalanceAfter >= rewardTokenBalanceBefore;
}
```

5. Deposit Withdraw Reversed ✓ *

A call to deposit and a subsequent call to withdraw with the same amount of `staticATokens` received must yield the same original balance for the user[3][4][5].

```
  {
      l2Recipient = address2uint256(eF.msg.sender)
      setupTokens(asset, Atoken, static)
      setupUser(eB.msg.sender)

      balanceU1 = tokenBalanceOf(asset, eB.msg.sender)
      balanceA1 = tokenBalanceOf(Atoken, eB.msg.sender)
      balanceS1 = tokenBalanceOf(static, eB.msg.sender)
  }

      staticAmount = deposit(Atoken, l2Recipient, amount, referralCode, fromUA) @

      initiateWithdraw_L2(Atoken, staticAmount, eB.msg.sender, toUA) @ env eF
  {
       balanceU2 = tokenBalanceOf(asset, eB.msg.sender)
       balanceA2 = tokenBalanceOf(Atoken, eB.msg.sender)
       balanceS2 = tokenBalanceOf(static, eB.msg.sender)

       balanceS1 == balanceS2
           ∧
       fromUA == toUA => balanceU2 - balanceU1 <= (indexL1/RAY()+1)/2
           ∧
       fromUA == toUA => balanceA2 == balanceA1
  }
```

6. Initialize Integrity ✓

After a call to initialize, a pair of underlying asset and AToken addresses must be registered correctly.

```
  {
       AToken._underlyingAsset == 0
       lendingPool.underlyingAssetToAToken_L1[asset] == 0
  }

      initialize() <call with arbitrary arguments>

  {
       asset ≠ 0 ∧ AToken ≠ 0 =>

       AToken._underlyingAsset == asset
       <=>
       lendingPool.underlyingAssetToAToken_L1[asset] == AToken
  }
```

7. AToken Asset Pair ✓

Once initialize has been called, `asset` being registered as the underlying token of `AToken`, and `AToken` being connected to `asset` in the lending pool must always hold.

```
asset ≠ 0 ∧ AToken ≠ 0 =>

AToken._underlyingAsset == asset
     <=>
lendingPool.underlyingAssetToAToken_L1[asset] == AToken
```

8. Cancel After Deposit Gives Back Exact Amount ❌

Cancelling a deposit successfully by requesting the `staticAmount` minted after `deposit` , must return the user's balance to the original state.

```
{
    user == e1.msg.sender
    user == e2.msg.sender
    user == e3.msg.sender

    setupTokens(asset, AToken, static)

    e1.block.timestamp <= e2.block.timestamp
    e2.block.timestamp < e3.block.timestamp

    ATokenBalance1 = tokenBalanceOf(AToken, user);
    assetBalance1 = tokenBalanceOf(asset, user)
}
    staticAmount = deposit(AToken, recipient, amount, code, fromUA) @ env e1
    startDepositCancellation(AToken, staticAmount, recipient, rewardsIndex, blo
    cancelDeposit(AToken, staticAmount, recipient, rewardsIndex, blockNumber, n
{
    ATokenBalance2 = tokenBalanceOf(AToken, user)
    assetBalance2 = tokenBalanceOf(asset, user)

    fromUA =>
         assetBalance1 == assetBalance2 + amount
         ∧
         ATokenBalance2 == ATokenBalance1 + amount

    !fromUA =>
        assetBalance1 == assetBalance2
        ∧

        ATokenBalance1 == ATokenBalance2
}
```

9. Cannot Cancel Deposit And Gain Both Deposit And Refund ❌

Once a deposit has been cancelled, the user cannot keep the minted staticATokens.

```
{
    user == e1.msg.sender
    user == e2.msg.sender
    user == e3.msg.sender
    setupTokens(asset, AToken, static)

    ATokenBalance1 = tokenBalanceOf(AToken, user)
    staticBalance1 = tokenBalanceOf(static, user)
}
    staticAmount = deposit(AToken, recipient, amount, code, fromUA) @ env e1
{
    ATokenBalance2 = tokenBalanceOf(AToken, user)
    staticBalance2 = tokenBalanceOf(static, user)
}
    startDepositCancellation(AToken, staticAmount, recipient, rewardsIndex, blo
    cancelDeposit(AToken, staticAmount, recipient, rewardsIndex, blockNumber, n

{
    ATokenBalance3 = tokenBalanceOf(AToken, user)
    staticBalance3 = tokenBalanceOf(static, user)

    staticBalance2 > staticBalance1 => ATokenBalance3 == ATokenBalance2
}
```

1. We assume the recipient is not a contract. ↩

2. When `fromUnderlyingAsset=false` we verified the rule for specific values of the liquidity index, otherwise for an arbitrary value of the index. ↩

3. Since rule #2 is violated, the underlying asset balance is not conserved in the process, but the difference is bounded by the liquidity index. ↩

4. The third assertion statement was proven for specific values of the liquidity index. The other two for every value. ↩

5. `address2uint256` represents a classic one-to-one conversion between an address variable to a uint256 variable. ↩