Formal Verification Report Of Aave Proof of Reserve

Summary

This document describes the specification and verification of Aave's Proof of Reserve project using the Certora Prover. The work was undertaken from 06/11/2022 to 22/11/2022. The latest commit reviewed and run through the Certora Prover was 12296ce.

The scope of our verification was the integration and utilisation of Chainlink's Proof of Reserve in the Aave protocol, which includes the following contracts:

- `AvaxBridgeWrapper.sol`
- `ProofOfReserveAggregator.sol`
- `ProofOfReserveExecutorBase.sol`
- `ProofOfReserveExecutorV2.sol`
- `ProofOfReserveExecutorV3.sol`
- `ProofOfReserveKeeper.sol`

Certora also performed a manual audit of these contracts.

The Certora Prover proved the implementation of the contracts listed above is correct with respect to the formal rules written by Certora.

The specifications are publicly available and can be found in Aave Proof of Reserve repository.

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.

Summary of Formal Verification

Overview Of AAVE Proof Of Reserve Protocol

The following description is taken from Aave's repository:

Proof-of-Reserve is a system by Chainlink that allows for reliable monitoring of reserve assets and usage of that data feed directly on-chain. If an anomaly is detected for a single asset, the system will try to apply the highest possible protections on the pool.

The repository implements a risk component that utilises Chainlink's Proof-of-Reserve assessment to execute emergency actions.

Below is the general flow of the proof of reserve check:

1. Anyone can call the publicly opened method executeEmergencyAction() of the Executor for the desired pool.
2. The Executor asks the Aggregator if any of the reserves is currently unhealthy.
3. Aggregator compares total supply against Chainlink's Proof of Reserve feed for every token enabled in prior.
4. If at least one reserve is compromised, then
    - for Aave V2 Executor disables borrowing for every asset on the pool and freezes only the exploited assets.
    - for V3, the broken asset is freezed and its LTV is set to 0.

Assumptions and Simplifications Made During Verification

We made the following assumptions during our verification:

- In our verification, we assume that the actions on `PoolConfigurator` and `LendingPoolConfigurator` are working as intended. We only checked that the relevant functions were invoked.
- In our verification of `ProofOfReserveExecutorV2.sol` and `ProofOfReserveExecutorV3.sol` we summarised the method `areAllReservesBacked` in `ProofOfReserveAggregator` to return arbitrary values.

- We unroll loops. Violations that require a loop to execute more than three times 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.

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 generalisation 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.

Verification of Example contract

Properties

Common for Executor

1. integrityOfDisableAssets ✔

The integrity of disabling an asset. After calling `disableAsset(asset)`, the asset's state should be disabled (False). If the asset already exists, then one asset should be removed; otherwise, no asset should be removed.

```
{
    assetStateBefore := getAssetState(asset),
    assetsLengthBefore := getAssetsLength()
}
```

```
disableAsset(asset)
{
    ¬assetStateAfter,
    ¬assetStateAfter ⇒ assetsLengthBefore = assetsLengthAfter,
    assetStateAfter ⇒ assetsLengthBefore = assetsLengthAfter + 1
}
```

2. integrityOfEnableAssets ✔

The integrity of enabling an asset. After calling `enableAsset(asset),` the asset's state should be enabled. If the asset already exists and is enabled, then no assets should be added; otherwise, one asset should be added.

```
{
    assetStateBefore := getAssetState(asset),
    assetsLengthBefore := getAssetsLength()
}
enableAsset(asset)
{
    assetStateAfter,
    assetStateBefore ⇒ assetsLengthBefore = assetsLengthAfter,
    ¬assetStateBefore ⇒ assetsLengthBefore = assetsLengthAfter − 1
}
```

3. enableDuplicationsWithStorage ✔

Enable the same asset twice is equal to enabling the asset once.

```
{
    assetStateBefore := getAssetState(asset),
    assetsLengthBefore := getAssetsLength()
}
enableAsset(asset)
enableAsset(asset)
{
    assetStateAfter2Calls := getAssetState(asset),
    assetsLengthAfter2Calls := getAssetsLength()
}
enableAsset(asset)
{
    assetStateAfter1Call := getAssetState(asset),
    assetsLengthAfter1Call := getAssetsLength()
}
{
    assetStateAfter2Calls = assetStateAfter1Call,
    assetsLengthAfter2Calls = assetsLengthAfter1Call,
    assetStateBefore ⇒ assetsLengthBefore = assetsLengthAfter1Call ∧ assetStat
    ¬assetStateBefore ⇒ assetsLengthAfter1Call = assetsLengthBefore + 1 ∧ asse
}
```

4. disableDuplicationsWithStorage ✓

Disabling the same asset twice is equal to disabling the asset once.

```
{
    assetStateBefore := getAssetState(asset),
    assetsLengthBefore := getAssetsLength()
}
disableAsset(asset)
disableAsset(asset)
{
    assetStateAfter2Calls := getAssetState(asset),
    assetsLengthAfter2Calls := getAssetsLength()
}
disableAsset(asset)
{
    assetStateAfter1Call := getAssetState(asset),
    assetsLengthAfter1Call := getAssetsLength()
}
{
    assetStateAfter2Calls = assetStateAfter1Call,
    assetsLengthAfter2Calls = assetsLengthAfter1Call,
    assetStateBefore ⇒ assetsLengthBefore = assetsLengthAfter1Call + 1 ∧ ¬asse
    ¬assetStateBefore ⇒ assetsLengthAfter1Call = assetsLengthBefore ∧ ¬assetSt
}
```

5. Asset State Is Valid** ✓

Invariant to verify that an address in the assets array is marked as with state at assetsState map.

6. Asset Is Not Zero** ✓

Invariant to verify that none of the assets is addressed zero.

7. Assets Uniqueness** ✓

Invariant to verify that the assets array is unique (without duplicated assets).

Unique for Executor V2

8. integrityOfExecuteEmergencyAction for Executor V2 ✓

Call `executeEmergencyAction()`; if `areAllReservesBacked` is false, then `_disableborrowing()` was called. otherwise, `_disableborrowing()` was not called

```
{
    allReservesBacked := areAllReservesBacked()
}
executeEmergencyAction()
{
    ¬allReservesBacked ⇒ disableBorrowingCalled
    allReservesBacked ⇒ ¬disableBorrowingCalled
}
```

Unique for Executor V3

9. integrityOfExecuteEmergencyAction for Executor V3 ✓

Call `executeEmergencyAction()`; if `areAllReservesBacked` is false, then `freezeReserve()` was called. otherwise, `freezeReserve()` was not called

```
{
    allReservesBacked := areAllReservesBacked()
}
executeEmergencyAction()
{
    ¬allReservesBacked ⇒ freezeReserveWasCalled
    allReservesBacked ⇒ ¬freezeReserveWasCalled
}
```

Aggregator

10. PoRFeedChange** ✓

Modification to the PoR feed of assets: if enableProofOfReserveFeed is called -> price feed must be a non-zero address if enableProofOfReserveFeedWithBridgeWrapper is called -> price feed and bridgeWrapper must be a non-zero address if disableProofOfReserveFeed called -> price feed and bridgeWrapper assigned to the asset must be nullified if any other function called -> price and bridgeWrapper must change

11. notAllReservesBacked_UnbackedArry_Correlation** ✓

If `areAllReservesBacked` is false, then at least one slot in `tokenBacked` array is false. otherwise, all slots in `tokenBacked` array are true.