CERTORA

Formal Verification Report of AAVE L2 Bridge

Summary

This document describes the specification and verification of AAVE L2 Bridge using the Certora Prover. The work was undertaken from June 5th to June 26th, 2022, after the contracts were deployed to the L2 chain. The latest commit reviewed and run through the Certora Prover was 303bd3e1.

The scope of this verification is Optimism, Arbitrum and Polygon network bridge protocols which includes the following contracts:

- `OptimisimBridgeExecutor.sol`
- `ArbitrumBridgeExecutor.sol`
- `PolygonBridgeExecutor.sol`

And its parent contracts:

- `BridgeExecutorBase.sol`
- `L2BridgeExecutor.sol`

Certora also performed a manual audit of these contracts.

During this verification process, 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 governance crosschain bridges repository.

List of Main Issues Discovered

Severity: Low

Issue:	Unexpected revert of queued actions

Issue:	Unexpected revert of queued actions
Description:	When invoking `_queue` with a batch of transactions, an action hash is calculated using the `keccak256` algorithm on the transaction parameters as the arguments, together with the `executionTime`, defined as $$block.timestamp + _ delay $$ Given two similiar actions (same function and arguments) in two different blocks, it is possible that after the first one was queued in the first block, the second one will lead to revert since there is a check of queued actions duplication. The execution time could have the same value for both actions if the delay was shortened between these two blocks such that the sum of delay and time stamp is the same. The aforementioned situation leads to an unexpected revert for this function call.
Property Violated:	independentQueuedActions (Property #18)
AAVE Response:	The community should be aware that any update of the executor parameters could affect the regular behaviour of the queued action sets of the contract. Then, governance proposals that change the parameters of the executors should be carefully reviewed taking into the account the current state of the executor contract and its queued action sets.

|

Severity: Low

Issue:	Impossible to queue two similar actions
Description:	`_queue()` includes a check whether an action was previously queued but yet to be executed. It does that in order to prevent action duplication in the same block (same actions in different block are allowed). The check is done by mapping each action by its signature, arguments and execution time to a `bytes32` hash and assigning each action hash a boolean (mapping) variable `_queuedActions[bytes32]`. The revert due to the action duplicity prevents any proposal which includes two similar transactions, even if not subsequent.
AAVE Response:	In the case where the proposal wants to execute duplicated actions on purpose the use of payload contracts is encouraged. Payload contracts help to make all actions of a sophisticated / more complicated proposal clearer and easier to get tracked.

Severity: Low

Issue:	Unexpected action Occurrence upon low-level call
Description:	In the methods `executeDelegateCall` and `_executeTransaction` an explicit low-level call is being made on a given target address. The default behavior of such a call in case that the target address does not exist is returning `success == true` (see in solidity docs). In this case, if the proposal was to specify a target that does not exist the following consequences will apply: 1. The methods will return a success value while the no actual action is being executed. 2. An event will be emitted by `execute` with a new `actionsSetId`, the address of the `execute` caller, and the `returndata` which will be empty. While the Eth refunded to the executor can be restored by raising a proposal, the emitted event may be misleading - making observers think that the action indeed succeed.
Recommendation:	It might be worth to check whether the target contract exist by checking the address code size prior to executing.
AAVE Response:	There could be a case where an action wants to send some value to a pre-calculated address of a contract that will be deployed later on.

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.

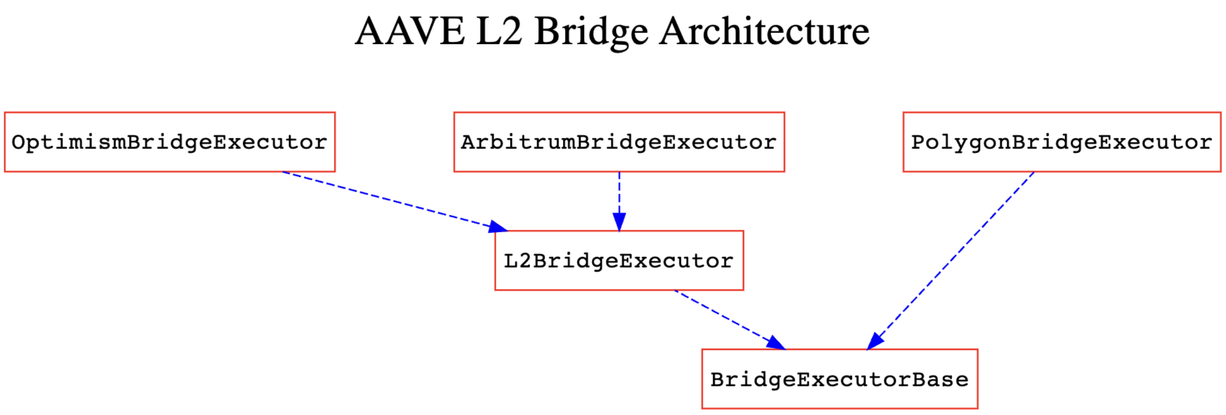
We hope that this information is useful, but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages 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 OptimismBridgeExecutor, ArbitrumBridgeExecutor, and PolygonBridgeExecutor Protocols

The basic contract in this complex is `BridgeExecutorBase.sol`. It is an abstract contact that is inherited by the main bridges contracts - Optimism, Arbitrum and Polygon.

The following graph shows the structure of the verified contracts:



In this verification we didn't distinguish between the functionality of the Optimism, Arbitrum and Polygon contracts, since they both inherit from the `BridgeExecutorBase` contract, which holds the implementations of the relevant functions.

The contract serves as an executor for a set of transactions/actions, proposed by a DAO community, to be executed on a target blockchain. The set of actions, including the target contracts, its functions, the arguments and message values, are proposed by the DAO and inserted to a queue, pending for a delay period before being allowed to execute. After a pre-defined time period has elapsed since the call to the queue, the queued actions-set can be executed by an external caller in a certain time period, also defined by the contract.

The transactions details are stored inside the contract's storage, while the set is given an exclusive set ID to distinguish it from others. There is a designated guardian address that is allowed to cancel transactions; the guardian address can be changed by the contract.

The actions-set could include transactions involving the contract itself, with or without delegate calls, and thus it is possible to change the state variables of the executor contract, allowing, among other update actions, to modify the very details of a queued action set. This introduces a potential abuse of the contract by any DAO community: the community is able to alter the proposals of other DAO's, already registered in this contract, at the time of the proposal execution (of first DAO).

Assumptions and Simplifications Made During Verification

We made the following assumptions during our verification:

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

Some rules are proven over the following set of simplified assumptions:

- The number of queued actions in the function `_queue()` was limited to no more than 2 actions in one set, in order to decrease computation complexity. In the rules depicted below, any call to queue() was replaced by queue2(), a simplified version of queue() for which there are only two transactions per action set, and the inputs are fixed length(2) arrays.
- Any delegate call (by requirement of the DAO proposal) should neither affect nor alter, the state variables of the executor contract, especially any details relating to the queued and/or executed action set. Thus no execute call should affect in any manner the details of any other queued set, but can only execute it (without altering the pre-defined actions by the DAO).

Therefore we ignore any violation found by our tool, due to a delegate-call, which changes any value of any state variable or details of another action set.

This assumption counts on the fact that every proposal is being manually reviewed by the DAO and other organizations like BGD Labs, etc.

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.

Formal Properties for BaseExecutor

Properties

1. **properDelay** ✓ The _delay variable should always have a proper value, between the minimum and maximum delay state variables.

```
getMinimumDelay() <= getDelay() ∧ getDelay() <= getMaximumDelay()
```

2. **actionNotCanceledAndExecuted** ✓ For any action set in the mapping `_actionSets[]`, its 'executed' and 'canceled' attributes must never be true at the same time.

```
    ¬(_actionSets[id].executed ∧ _actionSets[id].canceled)
```

3. **whoChangedStateVariables** ✓ Only the contract address can change the state variables of the contract.

```
    {
        delay1 = getDelay();
        period1 = getGracePeriod();
        minDelay1 = getMinimumDelay();
        maxDelay1 = getMaximumDelay();
        guardian1 = getGuardian();
    }
        < call to any contract method f >
    {
        delay2 = getDelay();
        period2 = getGracePeriod();
        minDelay2 = getMinimumDelay();
        maxDelay2 = getMaximumDelay();
        guardian2 = getGuardian();

        ¬( delay1 == delay2 ∧
        period1 == period2 ∧
        minDelay1 == minDelay2 ∧
        maxDelay1 == maxDelay2 ∧
        guardian1 == guardian2)
        =>
        msg.sender == currentContract
    }
```

4. **queueDoesntModifyStateVariables** ✓ The queue function must not change the state variables of contract (except `_actionsSetCounter`)

```
    {
        delay1 = getDelay();
        period1 = getGracePeriod();
        minDelay1 = getMinimumDelay();
        maxDelay1 = getMaximumDelay();
        guardian1 = getGuardian();
    }
        queue2(random arguments)
    {
        delay2 = getDelay();
        period2 = getGracePeriod();
        minDelay2 = getMinimumDelay();
        maxDelay2 = getMaximumDelay();
        guardian2 = getGuardian();

        delay1 == delay2 ∧
        period1 == period2 ∧
        minDelay1 == minDelay2 ∧
        maxDelay1 == maxDelay2 ∧
        guardian1 == guardian2
    }
```

5. **queueCannotCancel** ✓ A call to `_queue` cannot cancel any actions set.

```
    {
        environment e;
        getCurrentState(ID) at e ≠ canceled;
    }

        queue2(random arguments)

    {
        getCurrentState(ID) at e ≠ canceled;
    }
```

6. **executeCannotCancel** ✓ A call to execute cannot cancel any actions set.[1]

```
    {
        environment e
        getCurrentState(e, setCanceled) == 'queued'
        getGuardian() ≠ target contract address
    }
        execute(setCall) at e
    {
        getCurrentState(setCanceled) at e ≠ 'canceled'
    }
```

7. **noIncarnations** ✓ An action set ID is never queued twice, after being executed once.

```
{
    environment e
    environment e2
    environment e3
    actionsSetId = getActionsSetCount()
}
    queue(random arguments) at e
    execute(actionsSetId) at e2
    queue@withrevert(random arguments) at e3
{
    getCurrentState(actionsSetId) at e3 ≠ 'queued'
}
```

8. **executedForever** ✓ Once executed, an actions set ID remains executed forever.

```
    {
        environment e
        environment e2
        getCurrentState(actionsSetId) at e == 'executed'
    }
        < call to any contract method f>
    {
        getCurrentState(actionsSetId) at e2 == 'executed'
    }
```

9. **canceledForever** ✓ Once canceled, an actions set ID remains canceled forever.

```
    {
        environment e
        environment e2
        getCurrentState(actionsSetId) at e == 'canceled'
    }
        < call to any contract method f>
    {
        getCurrentState(actionsSetId) at e2 == 'canceled'
    }
```

10. **expiredForever** ✓ *[2] Once expired, an actions set ID remains expired forever.

```
    {
        environment e
        environment e2
        getCurrentState(actionsSetId) at e == 'expired'
    }
        < call to any contract method f>
    {
        getCurrentState(actionsSetId) at e2 == 'expired'
    }
```

11. **queuedStateConsistency** ✓ After calling to queue, the new action set state must be 'queued'.

```
{
    environment e
    ID = getActionsSetCount()
}
    queue2(random arguments)
{
    getCurrentState(ID) at e == 'queued'
}
```

12. **queuedChangedCounter** ✓ Queue must increase action counter set by 1.

```
{
    count1 = getActionsSetCount() < uint256.max
}
    queue2(random arguments)
{
    count2 = getActionsSetCount()
    count2 == count1+1
}
```

13. **onlyCancelCanCancel** ✓ Only `cancel(uint256)` function can cancel a queued set.[1]

```
{
    getCurrentState(actionsSetId) == 'queued'
    getGuardian() ≠ target contract address
}
    < call to any contract method f>
{
    getCurrentState(actionsSetId) == 'canceled'
    =>
    f.selector == cancel(uint256)
}
```

14. **cancelExclusive** ✓ `cancel(uint256)` only cancels one action set.

```
    {
        environment e;
        stateBefore = getCurrentState(actionsSetId2) at e
    }

        cancel(actionsSetId1) at e

    {
        stateAfter = getCurrentState(actionsSetId2) at e

        actionsSetId1 ≠ actionsSetId2 => stateBefore == stateAfter
    }
```

15. **holdYourHorses** ✓ When a delay is defined, a queued action set cannot be executed immediately.

```
{
    environment e
    actionsSetId = getActionsSetCount()
    delay = getDelay()
}
    queue2(random arguments) at e
    execute@withrevert(actionsSetId) at e
{
    delay > 0 => last reverted
}
```

16. **executedValidTransition1** ✓ An action set cannot transform from 'queued' to 'executed' by a function different than `execute(uint256)`.

```
{
    environment e
    state1 = getCurrentState(actionsSetId) at e
}
    < call to any contract method f ≠ execute(uint256) >
{
    state2 = getCurrentState(actionsSetId) at e

    ¬(state1 == 'queued' ∧ state2 == 'executed')
}
```

17. **executedValidTransition2** ✔ If an action set was executed, then this set (and only it) must change from 'queued' to 'executed'.

```
{
    environment e
    state1 = getCurrentState(actionsSetId) at e
}
    execute(actionsSetId2) at e
{
    state2 = getCurrentState(actionsSetId) at e

    actionsSetId2 == actionsSetId <=>
    state1 == 'queued' ∧ state2 == 'executed'
}
```

18. **independentQueuedActions** ❌ Two action sets in different blocks should be queued successfully even if the actions are identical.

```
{
    environment e1
    environment e2
    e1.msg.sender == e2.msg.sender
    e1.block.timestamp < e2.block.timestamp
}
    queue2(random arguments) at e1
    < call to a state changing f method > at e1
    queue2@withrevert(same random arguments) at e2
{
    ¬lastReverted
}
```

19. **executeRevertsBeforeDelay** ✔ Cannot execute before delay period passed.

```
{
    environment e
    environment e2
    actionsSetId = getActionsSetCount()
    delay = getDelay()
}
    queue2(random arguments) at e
    < call to some state changing function f> at e2
    execute@withrevert(actionsSetId) at e2
{
    e2.block.timestamp < e.block.timestamp + delay => lastReverted
}
```

20. **sameExecutionTimesReverts** ✓ [3] Two similar actions sets, cannot be queued together even in different blocks, when their execution time is equal.

```
{
    environment e1
    environment e2
    args = calldata random arguments
    t1 = e1.block.timestamp
    t2 = e2.block.timestamp
    t1 < t2
}
    queue2(args) at e1
    delay1 = getDelay()
    updateDelay(delay) at e1
    delay2 = getDelay()
    queue2@withrevert(args) at e2
{
    t1 + delay1 == t2 + delay2 => lastReverted
}
```

21. **actionDuplicate** ✓ `queue` cannot be called twice with the same arguments.

```
{
    environment e
    args = calldata random arguments
}
    queue2(args) at e
    queue2@withrevert(args) at e
{
    lastReverted
}
```

22. **executeFailsIfExpired**✓ Execute must fail if the actions set state is expired.

```
{
    environment e
    stateBefore = getCurrentState(actionsSetId) at e
}
    execute@withrevert(actionsSetId) at e
{
    stateBefore == 'expired' => lastReverted
}
```

23. **queuePriviliged**✓ `queue()` is a priviliged operation

```
{
    environment e1
    enviornment e2
}
    queue2(random arguments) at e1
    queue2@withrevert(other random arguments) at e2
{
    e1.msg.sender ≠ e2.msg.sender => lastReverted
}
```

24. **cancelPriviliged✔** `cancel(uint256)` is a priviliged operation

```
{
    environment e1
    enviornment e2
}
    cancel(random arguments) at e1
    cancel@withrevert(other random arguments) at e2
{
    e1.msg.sender ≠ e2.msg.sender => lastReverted
}
```

tags: AAVE-Continues

1. We assume the guardian is not a target contract. ↩ ↩2

2. Upon a call to updateGracePeriod(uint256), an actionSet becomes 'queued' again, but this is an expected behavior. For any other function call, an expired set remains so. ↩

3. This rule is meant to demonstrate that a change to the delay period between two blocks leads to revert of a second queue() if the transactions are identical and the execution time turns out to be the same. ↩