Formal Verification Report Of Aave Governance V3

Summary

This document describes the specification and verification of Aave's Governance V3 using the Certora Prover. The work was undertaken from April 16th to September 14th.

The scope of our verification includes the following contracts:

Mainnet:

- GovernanceCore.sol
- Governance.sol
- GovernancePowerStrategy.sol

Voting Chain:

- DataWarehouse.sol
- VotingMachine.sol
- VotingMachineWithProofs.sol
- VotingStrategy.sol

Execution Chain:

- PayloadsControllerCore.sol
- PayloadsController.sol
- Executor.sol

Multi-Chain:

- VotingPortal.sol
- BaseVotingStrategy.sol

The Certora Prover proved the implementation correct with respect to the formal rules written by the Certora team. During verification, the Certora Prover discovered bugs in the code which are listed in the tables below. All issues were promptly addressed. The fixes were verified to satisfy the specifications up to the limitations of the Certora Prover. The following section formally defines the high-level specifications of Aave Governance V3.

List of Main Issues Discovered

Severity: Medium

Issue:	**Preventing any proposal from being queued in the future.**
Description:	If the expiration delay is ever to be set to a value lower than the minimal flow duration (creation, voting, cooling down, bridging), any future proposal will not be able to be queued. Since the expiration delay can be updated only by an owner, and since the owner of the system is the executor which initiates actions only following a full proposal process, fixing this bad state will be impossible as a proposal to update the expiration delay will not be able to pass.
Mitigation/Fix:	Fixed in PR#170.

Severity: Low

Issue:	**An owner may resurrect an expired proposal.**
Rules Broken:	Property #4.
Description:	An owner may resurrect an expired proposal by changing the global voting duration of a certain access level when calling `Governance.setVotingConfigs()`. It will effectively set the expiration date into the future on previously expired payload.
Mitigation/Fix:	Fixed in PR#285.

Severity: Low

Issue:	**An owner may resurrect an expired payload.**
Rules Broken:	Property #7.
Description:	An owner may resurrect an expired payload by calling `PayloadsController.updateExecutors()` with a long enough

Issue:	**An owner may resurrect an expired payload.**
	`gracePeriod` or `delay` that will effectively set the expiration date to the future on previously expired payload.
Mitigation/Fix:	Fixed in PR#301.

Severity: Recommendation

Recommendation:	**Allow a mechanism for the invoker of `executePayload()` to supply the necessary ETH value for execution upon invocation.**
Description:	Currently, payment of payload actions must be done in 2 steps: (1) Send ETH to the `payloadController` (collected through `receive()`). (2) Call `executePayload()` to execute the actions with the stored necessary value.
Mitigation/Fix:	We analyzed the flow from scratch, and made some adjustments: - Executor does accept ETH. - `executePayload` method changed payable. The reasoning behind it is that we have 2 cases: (1) We want to put ETH in advance, so there is no point to put it on `PayloadsController`, we can put it straight to the executor. (2) If you want to pay during the execution, you set `action.value` in advance and send ETH to `executePayload`. (3) If you want to pay in advance, you send ETH to `PayloadsController` and set the action.value in advance The changes are incorporated in PR#321.

Severity: Recommendation

Issue:	**`VotingMachineWithProofs.submitVoteBySignature()` is incompliant with EIP-712 in the digest's calculation.**
Description:	`votingAssetsWithSlot` in `VOTE_SUBMITTED_TYPEHASH` is an array of `VotingAssetWithSlot` structs. The calculation of the digest in `VotingMachineWithProofs.submitVoteBySignature()` is directly using the `underlyingAssetsWithSlot` variable, which is an array of `VotingAssetWithSlot` structs corresponding to `votingAssetsWithSlot`. Directly using the actual variable instead of encoding the array values goes against the EIP-712 specification.
Recommendation:	See Appendix

Issue:	**`VotingMachineWithProofs.submitVoteBySignature()` is incompliant with EIP-712 in the digest's calculation.**
Mitigation/Fix:	Fixed in PR#314.

Severity: Informational

Issue:	**An owner can gain proposal cancellation power by abusing other power he possess.**
Description:	`cancelProposal()` is callable by the proposal creator if they have enough power, or by the guardian as long as the voting portal is approved. Since `cancelProposal()` uses `_checkGuardian()` and not `_checkOwnerOrGuardian`, it is clear that an owner should not have the power to cancel a proposal. An owner, however, can in fact bypass `_checkGuardian()` to successfully cancel a proposal in the following way: (1) call `removeVotingPortals()` to remove the approved voting portal from the `_votingPortals` list. (2) Call `cancelProposal()` which will pass without calling `_checkGuardian()` due to `isVotingPortalApproved(proposal.votingPortal) == false`. Such an ability gives more power than intended in the hand of a single entity which leads to greater centralization.
Mitigation/Fix:	The owner of the governance will be the executor, so an owner to abusing its power meaning a proposal and vote will have to pass. This flow should be allowed since it would be the result of voter intention.

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.

Assumptions and Simplifications Made During Verification

We made the following assumptions during our verification:

- We unroll loops. Violations that require executing a loop more than:
  - Once on `VotingPortal.sol` and some properties on `VotingMachine.sol`.
  - Twice on `VotingStrategy.sol`.
  - Three times on `Governance.sol`, `payloadController`, some properties on `VotingMachine.sol` and `GovernancePowerStrategy.sol`. will not be detected.
- We do not verify the cryptographic correctness of functions that involves calls to the `keccak256()` function.
- We check a single chain ID at at time.
- The proposal power of a user is being determined once and does not change over time.

Notations

✔️ indicates the rule is formally verified on the latest reviewed commit.

✔️* indicates that the rule is verified on the simplified assumptions described above in "Assumptions and Simplifications Made During Verification".

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

⏱ indicates the rule is currently timing out and therefore was not proved and no violations were found.

Properties of Aave Governance V3

Governance

1. ✔️ Proposal IDs are consecutive and incremental. Proposal ID increments by 1 iff `createProposal` was called
2. ✔️ Every proposal should contain at least one payload with all its required data. For initialized proposal, the payload list is not empty.
3. ✔️ Address zero cannot be a creator of a proposal

4. ✔️ If a voting portal gets invalidated during the proposal life cycle, the proposal should not transition to any non-terminal state, i.e. `Cancelled`, `Expired`, and `Failed`.
5. ✔️ If the proposer's proposition power goes below the minimum required threshold, an action on proposal should not go to any state apart from `Failed` or `Canceled`.
6. ✔️ Once a voting portal was configured for a proposal, it cannot be changed.
7. ✔️ State transitions are impossible if `proposal.state > 3`. I.e. `Executed`, `Failed`, `Cancelled`, `Expired` are terminal states.
8. ✔️ `proposal.state` can't transition backwards, it can only transition forward.
9. ✔️ It is impossible to do more than 1 state transition per proposal per block. Three exception were found and approved by the spec: 9.1 Cancellation due proposition power change. 9.2 Cancellation after proposal creation by creator. 9.3 Proposal execution after proposal queuing if `COOLDOWN_PERIOD = 0`. 9.4 Disapproval of voting portals by the `owner`, calling `removeVotingPortals()` or `setVotingConfigs()`.
10. ✔️ Only owner can set voting power strategy and voting config.
11. ✔️ A proposal cannot be queued if its voting portal is unapproved.
12. ✔️ Guardian can only cancel proposals with `proposal.state < 4`. I.e. while the proposal is still in process - `Null`, `Created`, `Active`, `Queued`.
13. ✔️ A guardian and an owner can cancel any proposal. A creator of a proposal can cancel his own proposal.
14. ✔️ The proposal parameters `creator`, `accessLevel`, `votingPortal`, `votingDuration`, `creationTime`, `ipfsHash`, `payloads` cannot be updated post proposal creation. 14.1. ✔️ The payload parameters `payloadChain`, `payloadAccessLevel`, `payloadsController`, `PayloadId` cannot be updated post proposal creation.
15. ✔️ The proposal parameters `votingActivationTime`, `snapshotBlockHash`, `snapshotBlockNumber` cannot be updated post voting activation.
16. ✔️ Voting via Portal can occur only when proposal `Active` and only once per voter. Property became **obsolete** after an architectural change.
17. ✔️ Only a valid voting portal can queue proposals.
18. ✔️ A voting portal can queue a proposal only if it is in Active state.

19. ✔ A proposal can be executed only after the `cooldownPeriod` has elapsed since it was queued.
20. ✔ When a proposal is in a terminal state, no function can change its state.
21. ✔ Only `createProposal()`, `activateVoting()`, `queueProposal()`, `executeProposal()`, `cancelProposal()` can progress the state of a proposal.

Cancellation Fee

22. ✔ For any proposal that wasn't created, the cancellation fee must be 0.
23. ✔ The ETH balance held in the contract can cover the total cancellation fee of all live proposals.
24. ✔ In any case that `proposal.CancellationFee` doesn't change, ETH balance cannot decrease.

Representative Logic

25. ✔ `updateRepresentatives()` modifies the representative set correctly.

GovernancePowerStrategy

26. ✔ An invalid token or slot is refused.
27. ✔ If a voter's power in each token is 0, then the voter has no voting power at all.
28. ✔ Transferring does not raise the power of the sender nor lowers the power of the receiver.
29. ✔ Delegating does not increase delegator's power nor decrease delegatee's power.

VotingMachine

29. ✔ A voter cannot vote twice. 29.1. ✔ An existing proposal must have a non-zero proposal blockhash. 29.2 ✔ Stored voting power is immutable (once positive).
30. ✔ Vote tally can change only for active and properly configured proposals.
31. ✔ Vote tally may change only if voter had zero stored voting power before. It can only change to a positive value.
32. ✔ When starting a proposal vote it already has a valid config.
33. ✔ Once a proposal vote is started the required roots exist.

34. ✔ Existing proposal config has non-zero duration.
35. ✔ New proposal must have a unique, unused ID.
36. ✔ A proposal's configuration is immutable once set.
37. ✔ A proposal's vote start time must be set to an earlier time than its end time.
38. ✔ A started proposal's end time is derived from the start time and voting duration.
39. ✔ A proposal can only be within one of its valid states: 39.1. ✔ A proposal not created iff the configured `endTime` is 0. 39.2. ✔ A proposal is active iff the proposal `endTime` is set (>0) and current timestamp is before `endTime`. 39.3. ✔ If current timestamp is after `endTime` then the proposal cannot be active. 39.4. ✔ A proposal is Finished iff the end time is set (>0), current timestamp is past `endTime` the proposal wasn't yet sent to mainnet. 39.5. ✔ A proposal is sent to governance iff `endTime` is set (>0) and current timestamp is after `endTime`.
40. ✔ A proposal can only transition between its valid states according to the defined state machine: 40.1 ✔ `NotCreated` can only transition to `Active` by a call to `startProposalVote()`. 40.2 ✔ `Active` can only transition to `Finished` by a passage of time. Not function can cause a state transition. 40.3 ✔ `Finished` can only transition to `SentToGovernance` by a call to `closeAndSendVote` 40.4 ✔ `SentToGovernance` is a terminal state.
41. ✔ The proposal parameters `id`, `startTime`, `endTime`, `creationBlockNumber` are immutable once created.
42. ✔ The voting machine can only send results for finished votes.
43. ✔ The stored voting power equals `getUserProposalVote`.
44. ✔ The sum of votes in favor and against equals the sum of stored voting powers.
45. ✔ If a proposal's votes tally changed then a vote was cast on the proposal. I.e. if the proposal's votes tally changed then there exists a voter `v` whose stored voting power on the proposal changed from zero to positive.
46. ✔ Casting a vote changes the proposal's votes tally. If a vote was cast, then the proposal's tally changed by an amount equal to the power of the vote cast.
47. ✔ Vote tally can be changed only by one of the voting methods - `submitVote`, `submitVoteBySignature` and `settleVoteFromPortal`.
48. ✔ Voting tally cannot decrease.

49. ✔ Only a single proposal's tally and votes may change by a single method call.
50. ✔ Only a single voter's stored voting power may change by a single method call on a given proposal.

PayloadsController.sol

52. ✔ A payload is uninitialized if it's beyond `_payloadsCount`. 52.1. ✔ The payload cannot have any set actions. 52.2. ✔ The payload state must be None. 52.3. ✔ The payload maximal access level must be null. 52.4. ✔ The payload creator is the address 0. 52.5. ✔ The payload expiration time is 0.
53. ✔ A payload maximal access level is greater than or equal to the access level of its actions.
54. ✔ A valid payload must have valid maximal access level. ❌ - found issues in previous commits.
55. ✔ Payloads IDs are consecutive.
56. ✔ A payload must include at least one action.
57. ✔ The payload parameters `creator`, `maximumAccessLevelRequired`, `createdAt`, `expirationTime`, `delay` and `gracePeriod` cannot be updated post payload creation.
58. ✔ The payload actions fields `target`, `withDelegateCall`, `accessLevel`, `value`, `signature` and `callData` cannot be updated post payload creation.
59. ✔ Payload executor cannot be address 0.
60. ✔ Post payload creation, all action access level must not be null.
61. ✔ Post payload creation, payload maximal access level must not be null.
62. ✔ A Payload can only be executed once in queued state, time lock has finished and before the grace period has passed.
63. ✔ The Guardian can cancel a Payload if it has not been executed yet.
64. ✔ A payload cannot execute after a guardian canceled it.
65. ✔ It's impossible to cancel a payload before creation.
66. ✔ It's impossible to cancel a payload after is was executed.

67. ✔ A state of a payload cannot can't transition backwards, it can only transition forward.
68. ✔ `Cancelled` and `Executed` are terminal states.
69. ✔ A Payload's grace period is global and determined by the controller.
70. ✔ Payload's delay is within the range `[MIN_EXECUTION_DELAY, MAX_EXECUTION_DELAY]`.
71. ✔ Executor delay of payload's max access level is within the range `[MIN_EXECUTION_DELAY, MAX_EXECUTION_DELAY]`.
72. ✔ Executor delay is within the range `[MIN_EXECUTION_DELAY, MAX_EXECUTION_DELAY]`.
73. ✔ Executor delay is within the range `[MIN_EXECUTION_DELAY, MAX_EXECUTION_DELAY]`.
74. ✔ A Payload can never be executed if it has not been queued before the `EXPIRATION_DELAY` defined.
75. ✔ Queuing time cannot occur after creation time and must occur before `creationTime + EXPIRATION_DELAY`.

VotingStrategy

76. ✔ `DelegationModeHarness.Mode` equals `DelegationMode`.
77. ✔ Zero power implies zero voting power.
78. ✔ For undelegated balance, the voting power is equal to the balance.
79. ✔ Token slot that aren't approved yield zero voting power.
80. ✔ Assets that aren't the white listed voting tokens aren't possessing any voting power.

Appendix

The correct way to encode the digest would be:

```
    // TODO: Declare this as immutable.
    // Defines the VotingAssetWithSlot struct's typehash
    bytes32 VOTING_ASSET_WITH_SLOT_TYPEHASH = keccak256(
      'VotingAssetWithSlot(address underlyingAsset,uint128 slot)'
    );
```

```
// Designate new memory underlyingAssetsWithSlot hashes.
bytes32[] memory underlyingAssetsWithSlotHashes = new bytes32[](
  underlyingAssetsWithSlot.length
);

// Iterate over each underlyingAsset
for (uint256 i = 0; i < votingBalanceProofs.length; ++i) {
  // Hash the VotingAssetWithSlot and place the result into memory.
  underlyingAssetsWithSlotHashes[i] = keccak256(
    abi.encode(
      VOTING_ASSET_WITH_SLOT_TYPEHASH,
      votingBalanceProofs[i].underlyingAsset,
      votingBalanceProofs[i].slot
    )
  );
}

// Derive and return the order hash as specified by EIP-712.
bytes32 digest = _hashTypedDataV4(
  keccak256(
    abi.encode(
      VOTE_SUBMITTED_TYPEHASH,
      proposalId,
      voter,
      support,
      keccak256(abi.encodePacked(underlyingAssetsWithSlotHashes))
    )
  )
);
```