blltprf.xyz
konnov.phd

Igor Konnov, Thomas Pani

Formal Verification Report

Aztec Governance Contracts

September 15, 2025

prepared for

AZTEC

Table of contents

Version history

- [05.09.2025] Version 1: Findings and verification results
- [12.09.2025] Version 2: Issue triage
- [15.09.2025] Version 3: Final report

1 Project summary

1.1 Project scope

Project name	Repository (URL)	Commit hash	Platform
Aztec Governance	https://github.com/AztecProtocol/aztec-packages/	Start: ffc8af0c End: 8b10b2b2	EVM

1.2 Project overview

This report presents the specification and formal verification of the **Aztec Governance Protocol** using the following tools:

- Quint is an engineer-friendly specification language that builds on top of the Temporal Logic of Actions (TLA$^+$). The authors of this report were core developers of Quint, which is currently developed at Informal Systems.
- Apalache is a symbolic model checker for Quint and TLA$^+$. The authors of this report are core developers of Apalache. Started in 2016 by Igor Konnov, Apalache has been developed at TU Wien, INRIA, Informal Systems, and through independent open-source contributions.
- Microsoft Z3 is a satisfiability-modulo-theories solver (SMT solver) that solves symbolic constraints of industrial complexity. Apalache uses Z3 to discharge the verification conditions.

We manually inspected the code to identify code fragments that would benefit from formal verification. We have semi-manually translated the selected contracts into Quint, identified and formulated core invariants, and checked correctness of these invariants with Apalache. The main body of work was done from **August 4, 2025** to **August 29, 2025**.

The following contracts were included in the scope:

- `l1-contracts/src/governance/CoinIssuer.sol`
- `l1-contracts/src/governance/GSE.sol`

- l1-contracts/src/governance/Registry.sol
- l1-contracts/src/governance/Governance.sol
- l1-contracts/src/governance/GSEPayload.sol
- l1-contracts/src/governance/RewardDistributor.sol
- l1-contracts/src/governance/proposer/EmpireBase.sol
- l1-contracts/src/governance/proposer/GovernanceProposer.sol
- l1-contracts/src/governance/interfaces/ICoinIssuer.sol
- l1-contracts/src/governance/interfaces/IGovernance.sol
- l1-contracts/src/governance/interfaces/IPayload.sol
- l1-contracts/src/governance/interfaces/IRegistry.sol
- l1-contracts/src/governance/interfaces/IEmpire.sol
- l1-contracts/src/governance/interfaces/IGovernanceProposer.sol
- l1-contracts/src/governance/interfaces/IProposerPayload.sol
- l1-contracts/src/governance/interfaces/IRewardDistributor.sol
- l1-contracts/src/governance/libraries/AddressSnapshotLib.sol
- l1-contracts/src/governance/libraries/ConfigurationLib.sol
- l1-contracts/src/governance/libraries/Errors.sol
- l1-contracts/src/governance/libraries/CheckpointedUintLib.sol
- l1-contracts/src/governance/libraries/DepositDelegationLib.sol
- l1-contracts/src/governance/libraries/ProposalLib.sol
- l1-contracts/src/core/slashing/Slasher.sol
- l1-contracts/src/core/slashing/SlashingProposer.sol
- l1-contracts/src/shared/interfaces/IMintableERC20.sol
- l1-contracts/src/shared/libraries/BN254Lib.sol
- l1-contracts/src/shared/libraries/CompressedTimeMath.sol
- l1-contracts/src/shared/libraries/SignatureLib.sol
- l1-contracts/src/shared/libraries/TimeMath.sol

In the initial stages, we ran the Quint randomized simulator and randomized symbolic executor. However, we quickly found that the potential attack surface of the contracts is too large for fuzzing-like approaches. Hence, we have constructed inductive invariants, and formally verified them using Apalache. This approach allows us to symbolically reason about a multitude of scenarios, more than could have been covered by fuzzing of concrete values. The approach also scales better, as it allows us to decompose the verification problem and massively parallelize its verification. See more details in the Methodology section.

The Apalache model checker has identified multiple issues with our specification and the invariants. We have triaged these issues and taken the following actions:

- Fixed the specification, whenever we found issues with the specification. We have carefully reviewed the Solidity code to make sure that similar issues do not appear in the contracts.
- Fixed the invariants, whenever we found the invariants to be inadequate.

- Opened issues on potentially problematic behavior in the Solidity contracts (included in this report).

2 Protocol overview

Aztec's governance framework is engineered to manage a network of blockchains by directing inflationary rewards to a single, designated "canonical rollup." The system deliberately concentrates authority in its sequencers—staked block producers whose vested capital and operational duties align their interests with the network's stability and survival.

This power is exercised through a consensus model for initiating proposals, with voting rights derived directly from stake managed by a central Governance Staking Escrow (GSE). This critical contract not only translates stake into voting power but also enables the seamless migration of these assets to new canonical chains. This mechanism is specifically designed to solve the "cold-start" problem by guaranteeing immediate operational support for network upgrades. To balance this concentration of power, the model incorporates fallback mechanisms, such as an emergency proposal option for any large token holder, ensuring a vital check against absolute sequencer control.

Among all the contracts in scope, the following contracts constitute the most sophisticated mechanisms of the governance logic:

- `EmpireBase` and `GovernanceProposer` constitute the bottom layer of the voting system. They implement a round-based signalling mechanism to decide on which proposals are forwarded to `Governance`.
- `Governance` is the key contract that manages the governance proposal lifecycle, including proposal submission, voting and proposal execution.
- `GSE` acts as an escrow that manages delegations and voting power across multiple rollups, including a feature to "follow-along" with the latest rollup instance.
- `SlashingProposer` implements a round-based signalling mechanism to decide which slashing proposals are approved.

3 Methodology

In this section, we give the details on the specification and verification efforts.

3.1 Specifications

The contracts are translated to the formal specification language called Quint. This specification language is backed by the formal semantics (mathematical meaning) in the logic of TLA$^+$ (Lamport 2002). The Quint specifications are automatically translated into TLA$^+$. This gives our specifications a solid mathematical foundation, which is based on several decades of academic and industrial experience.

Our specifications are organized in four layers:

- **Functional specifications** that capture the behavior of individual contracts. These specifications are purely functional, as they do not maintain their own states. Each contract method definition in these specifications receives the global `EvmState` at their input and the definition parameters. Each definition returns `Result[EvmState]`, which indicates either an error (with a non-empty field `err`), or a successful result (with the error field `err` being empty). In this project, the functional specifications can be found in the modules:
  - `spec/empire_base.qnt`
  - `spec/erc20.qnt`
  - `spec/governance_proposer.qnt`
  - `spec/governance.qnt`
  - `spec/gse_payload.qnt`
  - `spec/gse.qnt`
  - `spec/registry.qnt`
  - `spec/reward_distributor.qnt`
  - `spec/slasher.qnt`
  - `spec/slashing_proposer.qnt`
- **Key properties** capture the state invariants that should not be violated by the smart contracts. These are typically state invariants. In this project, the key properties are to be found the following modules:
  - `spec/governance_proposer_props.qnt`

  - `spec/slashing_proposer_props.qnt`
  - `spec/governance_props.qnt`

- **Invariant specifications** capture the key logical relations between the states of the smart contracts. Since we capture the global state in `EvmState`, these invariants are defined over parts of this data structure. In contrast to the properties, the invariants capture the whole state of the system. Often, these invariants are called *inductive*. More on that below. In this project, the invariants are defined in the module:

  - `spec/invariant.qnt`

- **State-machine specifications** assemble the contract specifications into a single state machine. Such a machine stores the EVM in a single variable `evm`. The state-transitions of this machine (called "actions") non-deterministically select the input arguments and execute the functional contract methods. Importantly, the specification parameters in this state machine are *bounded*: a small number of contracts is instantiated, bounds on the sizes of the data structures are given, etc. Nevertheless, this state machine is designed for symbolic analysis. For instance, the values of the integer variables lie in the range $[0, 2^{256})$, unless the data structures require smaller ranges. In this project, the state machine is defined in the module:

  - `spec/invariant_model.qnt`

3.2 Analysis surface

Our formal analysis has to deal with the behavior of about a dozen of smart contracts. Each of these contracts allows execution of several methods under different conditions. The following table summarizes the potential external call methods (we exclude constructors and some of the methods):

Contract	Methods
GovernanceProposer	signal, submitRoundWinner
Governance	updateGovernanceProposer, updateConfiguration, deposit, initiateWithdraw, finaliseWithdraw, propose, proposeWithLock, vote, execute, dropProposal
GSE	setGovernance, addRollup, deposit, withdraw, finaliseWithdraw, proposeWithLock, delegate, vote, voteWithBonus
SlashingProposer	signal, submitRoundWinner
Slasher	vetoPayload, slash
Registry	addRollup, updateRewardDistributor

Contract	Methods
RewardDistributor	claim

This gives us 28 methods to call in every execution prefix. Many of the methods can be called independently for every prefix, with the right choice of the inputs. Exhaustive enumeration of all these possibilities up to 20 method calls would require analysis of 28^{20} symbolic paths! Randomized simulation with random choice of inputs becomes even less feasible, as the number of paths has to be multiplied by the potential set of inputs. Given that consensus-like contracts such as `GovernanceProposer` and `SlashingProposer` require voting steps from multiple voters, limiting the number of method calls to small numbers is not feasible.

Given complexity of this task, we apply the technique of inductive invariants, which we briefly introduce below.

3.3 Formal verification

3.3.1 Overview of the approach

In this work, we apply the techniques of symbolic and bounded model checking. For a comprehensive overview of these techniques, see (Clarke et al. 2018). The model checker Apalache translates formal specifications into satisfiability-modulo theory constraints (SMT) and discharges them via Microsoft Z3. The formal foundations of the applied tooling are peer-reviewed and published at the top venues on computer-aided verification (Konnov, Kukovec, and Tran 2019) and (Otoni et al. 2023).

From the high-level point of view, the state machine is checked with two queries:

1. Consider an arbitrary symbolic state that satisfies the inductive invariant `all_inv` over the global state `evm: EvmState`. Symbolically analyze all possible (symbolic) steps of the state machine from the arbitrary state. This is proven by Z3, as translated by Apalache. At the technical level, this is done by executing the command:

   ```
   $ quint verify --max-steps=1 \
       --invariant=all_inv --init=init --step=step
   ```

2. Prove with Z3 that an arbitrary state satisfying `all_inv` also satisfies the safety properties. For example, this is how we check solvency of the Governance contract:

   ```
   $ quint verify --max-steps=0 \
       --invariant=governance_solvency_inv --init=init --step=step
   ```

3.3.2 Witnesses of liveness

It is well-known that overly restrictive inductive invariants may prohibit protocol transitions from being executed. To make sure that our inductive invariant does not overconstrain the protocol, we introduce so-called *falsy invariants*. The purpose of this invariants is to receive a counterexample from the model checker, to make sure that certain good states are reachable.

Below is an example of such a falsy invariant. In our specification, all such invariants have the prefix `_ex`, which indicates example generation.

```
// Check this invariant to find an example of having at least one
// executable proposal:
// quint verify --max-steps=0 --invariant=gov_proposals_executable_ex \
//   spec/invariant_model.qnt
val gov_proposals_executable_ex = {
  not(evm.governances.keys().forall(ga => {
    val g = evm.governances.get(ga)
    g.proposals.indices().exists(proposalId => {
      val proposal = g.proposals[proposalId]
      val result = Governance::getProposalState(evm, ga, proposalId)
      isOk(result) and (result.v == ProposalState_Executable)
    })
 }))
}
```

3.3.3 Compositional verification

Our combined state machine in `spec/invariant_model.qnt` produces 31 symbolic transitions and 32 verification conditions. This gives us 992 verification tasks in total. Moreover, our initialization predicate `init` consists of constraints for all the contracts. Direct verification of these invariants can easily make this task infeasible.

We build the inductive invariant in such a way that invariant verification can be decomposed without betraying correctness. More precisely, the inductive invariant is constructed as conjunction of the invariants for each contract:

```
  // all invariants combined
  pure def _all_invariants(_evm: EvmState): bool = and {
    _general_inv(_evm),
    // ERC20 invariants
    _all_asset_inv(_evm),
    // Registry invariants
    _all_registry_inv(_evm),
```

```
    // RewardDistributor invariants
    _all_reward_distributor_inv(_evm),
    // Governance invariants
    _all_governance_inv(_evm),
    // GovernanceProposer invariants
    _all_governance_proposer_inv(_evm),
    // GSE invariants
    _all_gse_inv(_evm),
    // Slasher invariants
    _all_slasher_inv(_evm),
    // SlashingProposer invariants
    _all_slashing_proposer_inv(_evm),
  }
```

The most general initializer looks as follows:

```
  action init = {
    init_with_post(I::_all_invariants)
  }
```

This gives us a natural way of decomposing the initialization predicate into smaller pieces. For example, to verify the invariants of `GovernanceProposer`, we need only small part of `_all_invariants`. The other parts may remain unconstrained. This is how we produce the initialization predicate for `GovernanceProposer`:

```
  action init_gp = {
    init_with_post(_evm => and {
      I::_all_registry_inv(_evm),
      I::_all_governance_proposer_inv(_evm)
    })
  }
```

This approach significantly reduces the verification times.

3.3.4 Parallel verification

As we have mentioned above, checking the invariants of our specification produces 992 verification tasks in total. Some of these tasks take a few minutes to check, and some of them take a few hours. This is caused by the nature of the constraints. It is well-known that the SMT solvers, including Z3, are challenged by non-linear integer arithmetic.

Since Apalache decomposes invariant checking into smaller tasks, we employ GNU parallel to massively parallelize the verification (Tange 2011). We use two servers to run the experiments:

1. AMD Ryzen 9 5950X 16-Core Processor, 128 GB memory
2. 2× Intel Xeon Platinum 8280 processor (28C/56T each), pinned at 3.1 GHz, 240 GB memory

3.3.5 Limitations and recommended further steps

Our approach has several limitations:

- **Bounded data structures.** Since Apalache uses a quantifier-free encoding, it needs upper bounds on the initial size of the data structures, though it does not require bounds on integer variables. We impose relatively small bounds on the sizes of lists, maps, and sets. This is in line with the *small model hypothesis*, namely, that many bugs are found on small models. Nevertheless, if one wants to prove the results about large data structures or unbounded data structures, one has to write **proofs in a theorem prover** such as Lean or Rocq.
- **No code-level verification.** Apalache verifies protocol specifications in Quint, which are semi-manually translated from Solidity in this project. Even though we have peer-reviewed the translation multiple times, the actual contracts may expose unexpected behavior that is related to Solidity or EVM. To gain more confidence in our protocol specification, we recommend the following further steps:
  - Producing a **diff-testing suite** that would test the Quint specifications against the Solidity code.
  - In addition to that, our verification conditions should be valuable for the **code-level formal verification tools** such as Certora, Kontrol, Halmos, or HEVM.
- **No cryptographic primitives.** Logic-based verification tools are not equipped with built-in primitives to reason about cryptography. As is usual, we are abstracting the cryptography away.

4 Findings summary

The following table summarizes our findings, including their type and severity.

Severity	Found	Confirmed	Fixed
CRITICAL	0	-	-
HIGH	0	-	-
MEDIUM	5	5	1
LOW	3	3	1
INFO	6	6	1

For quick reference, the following table lists all the findings. The details are to be found in Chapter 5.

ID	Link	Title	Severity	Status
M01	Section 5.1	Executing an unexecuted slashing proposal from deep past	MEDIUM	ACK
M02	Section 5.2	SlashingProposer allows for signaling in the past	MEDIUM	ACK
M03	Section 5.3	Governance is allowed to deposit to itself, leading to insolvency	MEDIUM	FIXED
M04	Section 5.4	GSE withdrawal does not enforce deposit()'s `_withdrawer`	MEDIUM	ACK
M05	Section 5.5	Initial delegatee after deposit is the calling rollup	MEDIUM	ACK
L06	Section 5.6	Missing consistency check in Registry.updateRewardDistributor	LOW	ACK
L07	Section 5.7	Desync between GSE + Registry rollups	LOW	ACK

ID	Link	Title	Severity	Status
I08	Section 5.8	EmpireBase allows quorum of size 1	INFO	ACK
I09	Section 5.9	Anyone can vote with zero voting power in Governance	INFO	ACK
I10	Section 5.10	Anyone can initiate a withdrawal of zero amount in Governance	INFO	ACK
I11	Section 5.11	Non-existant withdrawals can be finalized in Governance	INFO	FIXED
I12	Section 5.12	Spamming of non-existant / zero-balance positions via GSE::delegate()	INFO	ACK
I13	Section 5.13	EmpireBase reports errors with the prefix GovernanceProposer	INFO	FIXED
I14	Section 5.14	Delegations to 0x00 are possible, temporarily burn voting power	INFO	ACK

5 Detailed findings

5.1 M01: Executing an unexecuted slashing proposal from deep past

Severity	Impact	Likelihood	Status
MEDIUM	MEDIUM	LOW	ACKNOWLEDGED

The method `submitRoundWinner` of `SlashingProposer.` has a protection mechanism against executing proposals that are older than `LIFETIME_IN_ROUNDS`.

Source

```
function submitRoundWinner(uint256 _roundNumber) external
      override(IEmpire) returns (bool) {
    // Need to ensure that the round is not active.
    address instance = getInstance();
    require(instance.code.length > 0,
      Errors.GovernanceProposer__InstanceHaveNoCode(instance));

    IEmperor selection = IEmperor(instance);
    Slot currentSlot = selection.getCurrentSlot();

    uint256 currentRound = computeRound(currentSlot);...
    require(
      currentRound > _roundNumber + EXECUTION_DELAY_IN_ROUNDS,
      Errors.GovernanceProposer__RoundTooNew(_roundNumber, currentRound)
    );

    require(
      currentRound <= _roundNumber + LIFETIME_IN_ROUNDS,
      Errors.GovernanceProposer__RoundTooOld(_roundNumber, currentRound)
    );
```

Correctness of this code relies on the following assumption: **The rollup instance is always returning slots in non-decreasing order.**

If the rollup instance is faulty (or malicious), it may return older slot numbers. When this happens, old signaling proposals may go through `submitRoundWinner` for the rounds older than `LIFETIME_IN_ROUNDS`. Consider a slashing proposal has collected a quorum in round `r1` but was not executed due to its expiration after `r1 + LIFETIME_IN_ROUNDS`. If the rollup returns a slot that fits into round `r1`, then this proposal may go through.

5.1.1 Impact

Consider the case of the same slashing proposal `p` having collected quorums in rounds `r1` and `r2`. Only the proposal in round `r2` was executed, whereas the proposal in round `r1` has expired. If the proposal `p` is executed in the past round `r1`, validators may be slashed multiple times.

5.1.2 POC

We have written the following invariant in Quint:

```
def SlashingProposer::past_proposals_not_executable(
    __evm_state: EvmState, __self: ISlashingProposer): bool = {
  val sp = __evm_state.slashingProposers.get(__self)
  val currentSlot = __evm_state.rollupSlot.get(sp.INSTANCE)
  pure def _no_actions(_evm: EvmState): Result[EvmState] = { ok(_evm) }
  // make sure that the rounds very much in the past are not executable
  sp.empireBase.rounds.keys().forall(rollup => {
    val rounds = sp.empireBase.rounds.get(rollup)
    // compute the maximum round that has been ever
    // seen by SlashingProposer
    val maxRound = rounds.keys().fold(-1,
      (maxRound, round) => if (round > maxRound) round else maxRound)
    rounds.keys().forall(round => {
      val accounting = rounds.get(round)
      // EmpireBase.sol#L195: require(currentRound
      //   <= _roundNumber + LIFETIME_IN_ROUNDS, ...)
      not(accounting.executed) and
          (maxRound > round + sp.empireBase.LIFETIME_IN_ROUNDS)
        implies {
        // it should be impossible to submit a round winner
        // for this round
        val result = SlashingProposer::submitRoundWinner(__evm_state,
          __self, _no_actions, round)
        isErr(result)
      }
```

```
    })
  })
}
```

This invariant is checked as follows:

```
$ quint verify --max-steps=0 --init=init_sp --step=slashing_proposer_step \
  --invariant=slashing_proposer_safety spec/invariant_model.qnt
```

The counterexample shows that this invariant can be violated, when the rollup returns outdated slots:

```
      ...
      rollupSlot:
        Map("r2" -> 53, ...)

      slashingProposers:
        Map(
          "sp1" ->
            {
              INSTANCE: "r2",
              SLASHER: "slasher1",
              _ghostExecuted: Set((10000, "payload1")),
              empireBase:
                {
                  EXECUTION_DELAY_IN_ROUNDS: 0,
                  LIFETIME_IN_ROUNDS: 1,
                  QUORUM_SIZE: 16,
                  ROUND_SIZE: 18,
                  nonces: Map("r2" -> 0),
                  rounds:
                    Map(
                      "r2" ->
                        Map(
                          3 -> {
                              executed: false,
                              lastSignalSlot: 69,
                              payloadWithMostSignals: "payload3",
                              signalCount: Map("payload3" -> 16)
                            },
                          1 -> {
                              executed: false,
                              lastSignalSlot: 35,
                              payloadWithMostSignals: "payload2",
                              signalCount: Map("payload2" -> 18)
```

```
      },
    4 -> {
      executed: false,
      lastSignalSlot: 87,
      payloadWithMostSignals: "payload3",
      signalCount:
        Map("payload3" -> 2, "payload1" -> 2)
    }
...
```

In the above example, the proposal in round 1 is obviously outdated, as there is a signaling proposal in round 4, whereas `LIFETIME_IN_ROUNDS == 1`. However, since `rollupSlot.get("r2") == 53`, `SlashingProposer` believes that `roundNumber == 2`. Hence, `payload2` may be submitted.

One can argue that there is a window for manually vetoing the proposal. This is true, though vetoing this proposal may veto the current proposal with the same payload.

5.1.3 Recommendation

Record the `lastSignalSlot` per rollup instance, not per round: source

There should be no reason for receiving outdated slot numbers. Require `currentSlot > lastSignalSlot` in `submitRoundWinner`.

5.1.4 Customer's Response

If the assumption is broken due to a malicious rollup, the proposer could also be faked and it could just have a fresh proposal that way.

We see that it may cover a case where the slot derivation is faulty, but the rollup should otherwise be functional. But we have a hard time seeing that it would not require the rollup to be completely broken.

We are leaning to not addressing these, as it would only happen if the rollup is faulty in a very specific manner, e.g., during signaling increasing time, but then jumping into the past for execution. And the fix would increase the gas costs as we would have a separate storage slot to deal with, not a large amount, but we do not benefit from the packing anymore.

5.2 M02: SlashingProposer allows for signaling in the past

Severity	Impact	Likelihood	Status
MEDIUM	MEDIUM	LOW	ACKNOWLEDGED

The method signal in `SlashingProposer` has a protection mechanism against rollup's slots from the past:

Source

```
// Ensure that time have progressed since the last slot. If not,
// the current proposer might send multiple signals
require(
  currentSlot > round.lastSignalSlot.decompress(),
    Errors.GovernanceProposer__SignalAlreadyCastForSlot(currentSlot)
);
```

However, this protection mechanism works only in the same round. Similar to Section 5.1, correctness of the above code relies on the following assumption: **The rollup instance is always returning slots in non-decreasing order.**

If the rollup instance is faulty (or malicious), it may return older slot numbers. This may be especially interesting for the case when a proposal needs just a few signals for a quorum.

5.2.1 Impact

If a malicious rollup returns an outdated slot, e.g., for the previous round `r`, then `SlashingProposer` may collect a quorum for a recent proposal in the previous round. If this round `r` has not expired in terms of `LIFETIME_IN_ROUNDS` yet, then the proposal may become executable.

5.2.2 POC

We have written the following invariant in Quint:

```
def SlashingProposer::past_signals(__evm_state: EvmState,
    __self: ISlashingProposer): bool = {
  val sp = __evm_state.slashingProposers.get(__self)
  val currentSlot = __evm_state.rollupSlot.get(sp.INSTANCE)
  pure def _no_actions(_evm: EvmState): Result[EvmState] = { ok(_evm) }
```

```
    // make sure that the rounds very much in the past are not executable
    sp.empireBase.rounds.keys().forall(rollup => {
        val rounds = sp.empireBase.rounds.get(rollup)
        // compute the maximum round that has been ever seen by
        // SlashingProposer
        val maxRound = rounds.keys().fold(-1,
          (maxRound, round) => if (round > maxRound) round else maxRound)
        rounds.keys().forall(round => {
          val accounting = rounds.get(round)
          val payload = accounting.payloadWithMostSignals
          and {
            payload != ZERO_ADDRESS,
            accounting.signalCount.get(payload) ==
              sp.empireBase.QUORUM_SIZE - 1,
            round < maxRound,
            currentSlot >= round * sp.empireBase.ROUND_SIZE,
            currentSlot < (round + 1) * sp.empireBase.ROUND_SIZE,
          } implies {
            // it should be impossible to submit a signal for
            // this old currentSlot
            val result = SlashingProposer::signal(__evm_state,
              __self, sp.INSTANCE, payload)
            isErr(result)
          }
        })
    })
}
```

This invariant is checked as follows:

```
$ quint verify --max-steps=0 --init=init_sp \
  --invariant=past_signals spec/invariant_model.qnt
```

The counterexample shows that this invariant can be violated, when the rollup returns outdated slots:

```
      rollupSlot: Map("r1" -> 593, "r2" -> 0, "r3" -> 594, "r4" -> 595),
      ...
      slashingProposers:
        Map("sp1" -> {
              INSTANCE: "r1",
              SLASHER: "slasher1",
              empireBase: {
                  EXECUTION_DELAY_IN_ROUNDS: 0,
                  LIFETIME_IN_ROUNDS: 1,
```

```
QUORUM_SIZE: 34, ROUND_SIZE: 66,
nonces: Map("r1" -> 0),
rounds: Map("r1" -> Map(
        9 -> {
            executed: false,
            lastSignalSlot: 647,
            payloadWithMostSignals: "payload3",
            signalCount: Map("payload3" -> 54)
          },
        8 -> {
            executed: false,
            lastSignalSlot: 560,
            payloadWithMostSignals: "payload1",
            signalCount: Map("payload1" -> 33)
          }
```

In the above example, there is a round with a large slot value 647 in round 9. But since the rollup returned slot 593 for round 8, `SlashingProposer` accepts the signal.

5.2.3 Recommendation

Similar to M01, record the `lastSignalSlot` per rollup instance, not per round, see source.

5.2.4 Customer's Response

If the assumption is broken due to a malicious rollup, the proposer could also be faked and it could just have a fresh proposal that way.

We see that it may cover a case where the slot derivation is faulty, but the rollup should otherwise be functional. But we have a hard time seeing that it would not require the rollup to be completely broken.

We are leaning to not addressing these, as it would only happen if the rollup is faulty in a very specific manner, e.g., during signaling increasing time, but then jumping into the past for execution. And the fix would increase the gas costs as we would have a separate storage slot to deal with, not a large amount, but we do not benefit from the packing anymore.

5.3 M03: Governance is allowed to deposit to itself, leading to insolvency

Severity	Impact	Likelihood	Status
MEDIUM	MEDIUM	LOW	FIXED

`Governance.deposit` does not prevent `this` from being `msg.sender`.

Source

```
function deposit(address _beneficiary, uint256 _amount) external
      override(IGovernance) isDepositAllowed(_beneficiary) {
  ASSET.safeTransferFrom(msg.sender, address(this), _amount);
  users[_beneficiary].add(_amount);
  total.add(_amount);
  emit Deposit(msg.sender, _beneficiary, _amount);
}
```

If `msg.sender == this`, then it is possible to violate the solvency invariant:

1. `ASSET.safeTransferFrom(msg.sender, address(this), _amount)` goes through without updating the balances, as long as Governance has at least `amount` on its balance.
2. At the same time, the `users` entry for `_beneficiary` gets increased by `amount`, that is, `users[_beneficiary].add(_amount)`, and `total` is increased by `_amount`.
3. As a result, the beneficiary can initiate a withdraw of the `amount` tokens. Once these tokens are withdrawn, Governance does not have enough tokens on its balance, that is, it is insolvent.
4. Interestingly, it is possible for Governance to call its `deposit` by executing a proposal that contains a call to its `deposit`. Of course, this requires the proposal to be approved.

This only works if the underlying ERC20 does not require an approval when using `transferFrom` for the case `sender == spender`. OpenZepellin's implementation requires an explicit approval in this case.

5.3.1 Recommendation

Forbid the signature of the `deposit` method from being called on itself, similar to how it is done for the asset.

5.3.2 Customer's response

Fixed this in PR #16917.

Instead of dealing with as part of the executing a proposal, we think we can make it simple to read through at least by addressing it on the modifier `isDepositAllowed` for the deposit function directly, it seems like a good fit, as the deposit here is not allowed so making that fail makes sense to me.

5.4 M04: GSE withdrawal does not enforce deposit()'s `_withdrawer`

Severity	Impact	Likelihood	Status
MEDIUM	MEDIUM	MEDIUM	ACKNOWLEDGED

`GSE::deposit()` takes a `_withdrawer` argument. The NatSpec for `_withdrawer` says:

Source

```
* @param _withdrawer  - Address which can initiate a withdraw for the `_attester`
```

From the GSE's standpoint, this is not true. `GSE::withdraw()` enforces that only the rollup instance on which the deposit was made can initiate the withdrawal:

Source

```
  function withdraw(address _attester, uint256 _amount)
    external
    override(IGSECore)
    onlyRollup
    returns (uint256, bool, uint256)
  {
    // ...
    address withdrawingInstance = msg.sender;
    InstanceAttesterRegistry storage attesterRegistry =
                                                instances[msg.sender];
    bool foundAttester = attesterRegistry.attesters.contains(_attester);

    // ...
    require(foundAttester, Errors.GSE__NothingToExit(_attester));
```

In addition, the initiating rollup itself is the receiver of the withdrawal:

Source

```
uint256 withdrawalId = getGovernance()
                          .initiateWithdraw(msg.sender, amountWithdrawn);
```

5.4.1 Impact

An attester's `withdrawer` cannot withdraw from GSE directly. A faulty or malicious rollup can withdraw an attester's deposit and not return it to the depositor.

5.4.2 Recommendation

Allow the withdrawer to initiate withdrawal from GSE. If withdrawals are solely to be made through the rollup contracts, move withdrawer authorization into the rollup code.

5.4.3 Customer's recommendation

Will not be fixing this.

The withdrawals needs to go through the rollup because it will need to be able to also deal with slashes etc. The reason that we have the `withdrawer` specified in the GSE and not in the rollup directly, is because of the bonus, to allow "automatic" moving to the next rollup.

5.5 M05: Initial delegatee after deposit is the calling rollup

Severity	Impact	Likelihood	Status
MEDIUM	MEDIUM	MEDIUM	ACKNOWLEDGED

`GSE::deposit()` sets the initial delegatee to be the calling rollup:

Source

```
function deposit(
  address _attester,
  address _withdrawer,
  G1Point memory _publicKeyInG1,
  G2Point memory _publicKeyInG2,
  G1Point memory _proofOfPossession,
  bool _moveWithLatestRollup
) external override(IGSECore) onlyRollup {
  // ...
  delegation.delegate(recipientInstance, _attester, recipientInstance);
```

5.5.1 Impact

The calling rollup obtains voting power equivalent to the delegator's deposit.

A faulty or malicious rollup can by default vote against the delegator's intent, at least until the delegator re-delegates.

5.5.2 Recommendation

Allow delegator to set an initial delegatee. Alternatively, initial delegations could go to 0x00 (see also Finding Section 5.14).

5.5.3 Customer's response

It is on purpose that it is delegated to the rollup. We agree that it is a risk, but it was decided to improve speed - the tradeoffs have been weighed.

5.6 L06: Missing consistency check in Registry.updateRewardDistributor

Severity	Impact	Likelihood	Status
LOW	LOW	LOW	ACKNOWLEDGED

The method does not check that the new distributor does not refer to the registry:

Source

```
function updateRewardDistributor(address _rewardDistributor)
    external override(IRegistry) onlyOwner {
  $.rewardDistributor = IRewardDistributor(_rewardDistributor);
  emit RewardDistributorUpdated(_rewardDistributor);
}
```

The reward distributor can be only updated by the `Registry` owner, that is, `Governance`. If the new reward distributor points to another instance of `Registry`, then this update would result in a broken configuration.

5.6.1 Impact

An incorrect registry may allow a malicious address to claim the rewards:

Source

```
function claim(address _to, uint256 _amount)
    external override(IRewardDistributor) {
  require(msg.sender == canonicalRollup(),
    Errors.RewardDistributor__InvalidCaller(msg.sender,
                                            canonicalRollup()));
  ASSET.safeTransfer(_to, _amount);
}
```

5.6.2 Recommendation

Since `RewardDistributor` has immutable `REGISTRY`, we recommend adding a consistency test in `updateRewardDistributor`:

```
require(_rewardDistributor.REGISTRY == address(this), ...);
```

5.6.3 Customer's response

We see that this might help against a misconfiguration, but it at the same time is one of those where it would not help much else.

In the example this is mentioned to potentially help against malicious claiming, but that seem to already rely on a "wrong" implementation behaving correct for the extra check.

5.7 L07: Desync between GSE + Registry rollups

Severity	Impact	Likelihood	Status
LOW	MEDIUM	LOW	ACKNOWLEDGED

It is possible to add rollups only to GSE, or only to Registry. Thus, the "latest" GSE rollup may not be the "canonical" Registry rollup.

Developers are aware of this, but it is not enforced by the protocol:

Source

> NB: The "latest" rollup in this contract does not technically need to be the "canonical" rollup according to the Registry, but in practice, it will be unless the new rollup does not use the GSE. Proposals which add rollups that DO want to use the GSE MUST call addRollup to both the Registry and the GSE. See RegisterNewRollupVersionPayload.sol for an example.

5.7.1 Impact

Depends on governance users' economic reaction to this misconfiguration: For example, if depositors are not shifting their deposits to the GSE "latest" rollup, the normal governance route can get blocked (by GSEPayload's `amIValid` test), even though a proposal receives signal from the Registry's "canonical" rollup in GovernanceProposer.

Practically, this is an issue whenever the interests of the entitites returned by Registry "canonical"'s `getCurrentProposer()` and GSE "latest"'s attesters are not aligned, or when their interests shift between signalling and submission.

5.7.2 Recommendation

Provide a uniform interface that adds rollups to both GSE and Registry.

5.7.3 Customer's response

We are not interested in adding to both, as that could cause a "move" of stake into a version that will not allow it to exit.

For example, say that a new key-pair is needed for the validators. To get that working, a new GSE with updated structure would be deployed and when a rollup is to be added, it should add be added to the registry and the *new* GSE (not added to the old). If we forced them to add to both it would be more painful here.

5.8 I08: EmpireBase allows quorum of size 1

Severity	Impact	Likelihood	Status
INFO	INFO	LOW	ACKNOWLEDGED

The model checker has produced an example of having the quorum size of 1:

```
NewGovernanceProposerStep({
  governanceProposer: "gp1", gse: "gse1",
  quorumSize: 1, registry: "reg1", roundSize: 1
})
```

Source

```
require(QUORUM_SIZE > ROUND_SIZE / 2,
   Errors.GovernanceProposer__InvalidQuorumAndRoundSize(QUORUM_SIZE,
     ROUND_SIZE));
```

5.8.1 Impact

The protocol is allowed to have a completely centralized voter in the governance proposer.

5.8.2 Recommendation

Either require a non-singleton quorum, or make this possibility clear in the documentation.

5.8.3 Customer's response

This is expected.

If there is only one vote, it makes sense that that one vote decides completely.

Also, slightly strange, but we don't think this would actually be completely centralized, because every round would have a new "god".

5.9 I09: Anyone can vote with zero voting power in Governance

Severity	Impact	Likelihood	Status
INFO	INFO	LOW	ACKNOWLEDGED

We have a counterexample from the model checker. An arbitrary sender can vote with zero voting power. Even though they do not affect the voting outcome, they can spam the event log.

Source

```
if (_support) {
  userBallot.yea += _amount;
  summedBallot.yea += _amount;
} else {
  userBallot.nay += _amount;
  summedBallot.nay += _amount;
}
```

5.9.1 Customer's response

The behavior is similar to ERC20 spec where 0-transfers are also possible, it is just costly spam.

5.10 I10: Anyone can initiate a withdrawal of zero amount in Governance

Severity	Impact	Likelihood	Status
INFO	INFO	LOW	ACKNOWLEDGED

Similar to Section 5.9, we see withdrawals of zero amount. It does not seem to affect the protocol, except producing spam events.

Source code

```
function _initiateWithdraw(address _from, address _to,
    uint256 _amount, Timestamp _delay) internal returns (uint256) {
  users[_from].sub(_amount);
  total.sub(_amount);
  uint256 withdrawalId = withdrawalCount++;
  withdrawals[withdrawalId] = Withdrawal({amount: _amount,
    unlocksAt: Timestamp.wrap(block.timestamp) + _delay,
    recipient: _to, claimed: false});
  emit WithdrawInitiated(withdrawalId, _to, _amount);
  return withdrawalId;
}
```

5.10.1 Customer's response

The behavior is similar to ERC20 spec where 0-transfers are also possible, it is just costly spam.

5.11 I11: Non-existant withdrawals can be finalized in Governance

Severity	Impact	Likelihood	Status
INFO	LOW	MEDIUM	FIXED

`Governance::finaliseWithdraw()` can be called with a non-existing future withdrawal ID, resulting in a zero-amount transfer to address 0x00.

Source

```
function finaliseWithdraw(uint256 _withdrawalId)
    external override(IGovernance) {
  Withdrawal storage withdrawal = withdrawals[_withdrawalId];
  require(!withdrawal.claimed,
          Errors.Governance__WithdrawalAlreadyClaimed());
  require(
    Timestamp.wrap(block.timestamp) >= withdrawal.unlocksAt,
    Errors.Governance__WithdrawalNotUnlockedYet(
      Timestamp.wrap(block.timestamp),
      withdrawal.unlocksAt)
  );
  withdrawal.claimed = true;

  emit WithdrawFinalised(_withdrawalId);

  ASSET.safeTransfer(withdrawal.recipient, withdrawal.amount);
}
```

5.11.1 Impact

The contract will emit out-of-order `WithdrawFinalised` events. For example, this can trigger unexpected behavior in off-chain components that rely on consecutive order of withdrawal IDs (e.g., when the legitimate withdrawal with this withdrawalId is executed in the future).

5.11.2 Recommendation

Revert when someone attempts to finalize a non-existing withdrawal ID.

5.11.3 Customer's response

This has been fixed, the PR is here: AztecProtocol/aztec-packages#16476

5.12 I12: Spamming of non-existant / zero-balance positions via GSE::delegate

Severity	Impact	Likelihood	Status
LOW	LOW	LOW	ACKNOWLEDGED

`GSE::delegate()` can be called on a non-existing position.

Source

```
function delegate(address _instance, address _attester,
                  address _delegatee) external override(IGSECore) {
  require(isRollupRegistered(_instance),
      Errors.GSE__InstanceDoesNotExist(_instance));
  address withdrawer = configOf[_attester].withdrawer;
  require(msg.sender == withdrawer,
      Errors.GSE__NotWithdrawer(withdrawer, msg.sender));
  delegation.delegate(_instance, _attester, _delegatee);
}
```

5.12.1 Impact

1. Produces a zero-balance position for the new delegatee.
2. Does *not* create a `delegation.votingAccount` for the delegatee, because `moveVotingPower()` short-circuits on zero-value amounts.

5.12.2 Recommendation

Revert when someone attempts to re-delegate a non-existing position.

5.12.3 Customer's response

We won't address this.

Only possible from people that have previously deposited into the GSE and it won't have an effect on the system and only spam the event.

5.13 I13: EmpireBase reports errors with the prefix GovernanceProposer

Severity	Impact	Likelihood	Status
INFO	INFO	HIGH	FIXED

The source of `EmpireBase` contains multiple mentions of `GovernanceProposer` in the errors. For example:

```
require(
  currentRound > _roundNumber + EXECUTION_DELAY_IN_ROUNDS,
  Errors.GovernanceProposer__RoundTooNew(_roundNumber, currentRound)
);
```

However, `SlashingProposer` is also inheriting from `EmpireBase`. This may confuse the operator.

5.13.1 Recommendation

Remove the "Governance" prefix.

5.13.2 Customer's response

Addressed in PR #16950.

5.14 I14: Delegations to 0x00 are possible, temporarily burn voting power

Severity	Impact	Likelihood	Status
INFO	LOW	LOW	ACKNOWLEDGED

GSE withdrawers can delegate to address 0x00 – effectively suspending their delegation. Internally, `moveVotingPower()` "burns" the corresponding votingPower. However, the voting power can be recovered by "minting" by re-delegating the balance from 0x00 to another address.

Source

```
function delegate(
  DepositAndDelegationAccounting storage _self,
  address _instance,
  address _attester,
  address _delegatee
) internal {
  address oldDelegate = getDelegatee(_self, _instance, _attester);
  if (oldDelegate == _delegatee) {
    return;
  }
  _self.ledgers[_instance].positions[_attester].delegatee = _delegatee;
  emit DelegateChanged(_attester, oldDelegate, _delegatee);

  moveVotingPower(_self, oldDelegate, _delegatee,
                  getBalanceOf(_self, _instance, _attester));
}
```

Source

```
function moveVotingPower(
    DepositAndDelegationAccounting storage _self,
    address _from, address _to, uint256 _amount)
  private
{
  if (_from == _to || _amount == 0) {
    return;
  }

  if (_from != address(0)) {
```

```
    (uint256 oldValue, uint256 newValue) =
        _self.votingAccounts[_from].votingPower.sub(_amount);
    emit DelegateVotesChanged(_from, oldValue, newValue);
  }

  if (_to != address(0)) {
    (uint256 oldValue, uint256 newValue) =
        _self.votingAccounts[_to].votingPower.add(_amount);
    emit DelegateVotesChanged(_to, oldValue, newValue);
  }
}
```

5.14.1 Impact

Unnecessarily weakens the accounting invariant: the voting power is no more than the delegated balances

$$\forall_{delegatee}. \left(\text{votingAccounts}[delegatee].\text{votingPower} \leq \sum_{\substack{instance \in \text{instances} \\ p \in \text{ledgers}[instance].\text{positions} \\ p.\text{delegatee} = delegatee}} p.\text{balance} \right),$$

but the following does not hold:

$$\forall_{delegatee}. \left(\text{votingAccounts}[delegatee].\text{votingPower} = \sum_{\substack{instance \in \text{instances} \\ p \in \text{ledgers}[\text{instance}].\text{positions} \\ p.\text{delegatee} = delegatee}} p.\text{balance} \right),$$

No exploitable impact was found in the current implementation – nevertheless, this subtlety may lead to exploits if it is overlooked in future changes.

5.14.2 Recommendation

Ideally, disallow delegations to 0x00.
If (temporary) suspension of a delegation is desired, double-check uses of `moveVotingPower` on future changes.

5.14.3 Customer's response

Acknowledged, however we don't see an issue with removing the delegation of ones power while still being deposited.

6 Formal verification summary

We have outlined our methodology in Chapter 3.

6.1 Formalization assumptions

Since we specify the protocol logic, we do not have to use mocks. Our main goal is to specify contract behavior as precise as possible. However, we have to make the following protocol-level abstractions:

- Whenever possible, we use rich data structures offered by Quint, instead of low-level data structures of Solidity. For example, if a Solidity data structure implements a set, we simply use a set in Quint.
- Since the rollup behavior is outside the scope of this project, we model the interactions via the `IEmperor` instance as follows:
  - `IEmperor(...).getCurrentSlot()` returns the slot value `rollupSlot.get(rollup)`, which is non-deterministically set chosen each step, but it does not change within a single step.
  - `IEmperor(...).getCurrentProposer()` returns the address `rollupProposer.get(rollup)`, which is non-deterministically chosen for each step, but it does not change within a single step.
- We did not specify deposit control and flood gates in `Governance`. Instead, we consider the more general case of allowing any address to deposit. This is strictly more general behavior, which includes both the cases of the flood gates open and closed.
- We specify payload execution as a functional transformation from `EvmState` to `Result[EvmState]`. This technically allows us to capture side effects that include arbitrary modifications of `EvmState`. In case of `GSEPayload`, we introduce an additional execution phase `ExecuteGSEPayload`. In this phase, all steps by the initiating governance are allowed. In the end of this phase, the invariant `amIValid` of `gse_payload` is checked.
- We omit signature verification in the protocol specification. This does not overconstrain the protocol behavior. Rather, it introduces additional behaviors.

- ERC20 transfers are coupled with approvals. Hence, every `transferFrom` is considered to be approved by the sender.
- Binary search optimization in OpenZeppelin's `Trace224` upperLookup/lowerLookup is modeled as a linear filter operation. The upperLookupRecent optimization for recent checkpoint lookups is not modeled separately. This is a modeling simplification of optimized code, but does not affect correctness.

6.2 Notation

When presenting the verification results, we use the following notation:

- **Formally verified.** The invariant has been verified by the model checker. Technically, this means that Z3 terminated and returned "UNSAT" when assuming invariant violation.
- **Not falsified.** The model checker could not find a counterexample within gracious time budget. While this may seem like a negative result, we believe that this verdict is similar to fuzzing results, which are incomplete by definition. Technically, this means that Z3 either did not terminate with "SAT" when assuming invariant violation, or Z3 reported "UNKNOWN". This typically happens when the invariant involves non-linear integer arithmetic.
- **Violated.** The model checker produces a counterexample that has meaningful interpretation in the Solidity code. Technically, this means that Z3 has terminated with "SAT".

6.3 Key properties

In addition to a large number of lemmas that we use in the inductive invariant, we have identified a small number of key state invariants that the users may expect from the protocol. Not all of them hold true. In the following, we enumerate these properties and give their status. The exact invariants can be found in the files `governance_props.qnt`, `governance_proposer_props.qnt`, `gse_props.qnt`, `slashing_proposer_props.qnt`.

Contract	Property	Description	Status
Governance	emergency NotDroppable	An emergency proposal issued with `proposeWithLock` not droppable	Verified
Governance	executed NotEarlier	A proposal cannot be executed before queuedThrough + 1	Verified

Contract	Property	Description	Status
Governance	steadyState	Cached state does not change	Verified
Governance	executedWasAccepted	If a proposal was executed, it must have been accepted	Not falsified
Governance Proposer	safety	In every round, at most one payload has a quorum	Not falsified
Slashing Proposer	safety	In every round, at most one payload has a quorum	Not falsified
Slashing Proposer	past_proposals_not_executable	If a proposal was not executed LIFETIME_IN_ROUNDS in the past, then it cannot be executed at all	Violated by faulty rollup
Slashing Proposer	past_signals	A signal for an older slot cannot be executed	Violated by faulty rollup
GSE	twoThirds	$\frac{2}{3}$ of the stake are following the canonical rollup	Violated by deposits and withdrawals

6.4 Inductive invariant

As discussed in Chapter 3, we capture an arbitrary state of the protocol via an inductive invariant. Concretely, this is the definition `_all_invariants` that is defined in `invariant_model.qnt`:

```
// all invariants combined
pure def _all_invariants(_evm: EvmState): bool = and {
  _general_inv(_evm),
  // ERC20 invariants
  _all_asset_inv(_evm),
  // Registry invariants
  _all_registry_inv(_evm),
  // RewardDistributor invariants
  _all_reward_distributor_inv(_evm),
  // Governance invariants
  _all_governance_inv(_evm),
  // GovernanceProposer invariants
  _all_governance_proposer_inv(_evm),
  // GSE invariants
  _all_gse_inv(_evm),
  // Slasher invariants
  _all_slasher_inv(_evm),
```

```
    // SlashingProposer invariants
    _all_slashing_proposer_inv(_evm),
}
```

The invariants for individual components are further given in the respective definitions. We give the details on the smaller parts of these large invariants in Chapter 7.

7 Detailed invariants for formal verification

In the following, we list the conditions that the inductive invariant consists of. For the Quint version of these conditions we refer the reader to `invariant.qnt`. By looking at the condition label, you can find the corresponding expression.

The conditions below are **verified** with Apalache and Z3, unless explicitly noted. A few of them are **violated** or **not falsified**.

7.1 ERC20 governance token

ASSET-01: Token balances are non-negative

ASSET-02: Total supply is U256 and sums up to balances

7.2 Registry

REGISTRY-01: Registry has a valid owner

REGISTRY-02: Registry's reward distributor is a valid address

REGISTRY-03: The versions in `versions[]` must be exactly the keys of `versionToRollup`

REGISTRY-04: `versionToRollup` matches `ROLLUP_VERSIONS_MAP`

REGISTRY-05: Versions in `versions[]` must be unique

REGISTRY-06: `Registry.rewardDistributor`'s `REGISTRY` field points back to the registry

Violated (see Section 5.6)

7.3 RewardDistributor

RD-01: The asset must be a valid address

RD-02: The asset must exist

RD-03: The registry must be a valid address

RD-04: The registry must exist

7.4 Governance

GOV-01: Governance must have a valid proposer

GOV-02: Governance proposer cannot be the governance contract itself

GOV-03: Governance configuration must be valid

GOV-04: Every proposal must have a valid config

GOV-05: Every proposal's cached state is one of: Pending, Executed, Dropped

GOV-06: Every proposal must have a valid payload address

GOV-07: Every proposal must have a valid proposer

It is either the governance itself, or one of the governance proposers.

GOV-08: Every proposal must be created in the past until now

GOV-09: Proposal creation times are in non-decreasing order

GOV-10: Proposal's summed ballot counts are U256

GOV-11: If a proposal has some yea or nay, it must have ballots

GOV-12: A governance proposal cannot be executed early

Specifically, it cannot be executed earlier than `queuedThrough() + 1`

GOV-13: Each governance ballot points to a valid proposal ID

GOV-14: All users casting a governance ballot are beneficiaries

This is only true when the vote is non-zero (see Section 5.9).

GOV-15: Ballots' yeas and neas are U256

GOV-16: If the proposal has not been active yet, then no votes have been cast

GOV-17: A users's votes on a ballot are at most their voting power

This voting power is queried at the time `proposal.pendingThrough`.

GOV-18: Summed ballot's votes do not exceed the total voting power

This voting power is queried at the time point `proposal.pendingThrough`.

GOV-19: `summedBallot` is the sum of the per-user ballots

GOV-20: The timestamps in the `users` traces are ordered

GOV-21: For each entry in `users` voting power, there is an entry in `total`

GOV-22: User voting power trace values are U224

GOV-23: The timestamps in the total trace are in the past until now

GOV-24: Total voting power trace values are U224

GOV-25: The timestamps in the total trace are ordered

GOV-26: For each timestamp `t`, `total[t]` equals to the sum of the users' voting power at `t`

GOV-27: Withdrawal amounts are non-negative

Zero amounts are allowed in withdrawals (see Section 5.10).

GOV-28: Solvency: Governance holds enough balance to cover all future withdrawals

This is violated when Governance deposits to itself (see Section 5.3).

GOV-29: Key invariant connecting Governance to GovernanceProposer

Each "regular" (non-`proposeWithLock`) proposal in Governance must have a corresponding, submitted proposal in the proposal's GovernanceProposer

- GOV-29-01: The proposal's `proposer` has a `proposalProposer` entry at `proposalId`.
- GOV-29-02: The proposal's `proposer` has round accounting of a matching, executed proposal.

7.5 GovernanceProposer

GP-01: Basic GovernanceProposer state validity assertions

- GP-01-01: `REGISTRY` is a valid address
- GP-01-02: `GSE` is a valid address
- GP-01-03: EmpireBase `rounds` map from rollup addresses
- GP-01-04: EmpireBase `nonces` map from proposer addresses
- GP-01-05: `QUORUM_SIZE` is U256
- GP-01-06: `ROUND_SIZE` is positive, U256

- GP-01-07: `2 * QUORUM_SIZE > ROUND_SIZE`
- GP-01-08: `QUORUM_SIZE <= ROUND_SIZE`
- GP-01-09: `EXECUTION_DELAY_IN_ROUNDS` is U256
- GP-01-10: `LIFETIME_IN_ROUNDS > EXECUTION_DELAY_IN_ROUNDS`
- GP-01-11: `LIFETIME_IN_ROUNDS` is U256

GP-02: Key invariant connecting GovernanceProposer to Governance

Each proposal submitted from GovernanceProposer must be recorded in Governance

- GP-02-01: For each submitted proposal in `proposalProposer`, there is round accounting for a corresponding executed proposal (i.e., submitted to Governance).
- GP-02-02: Round accounting for that proposal is correct:
    1. There is a proposal of matching ID and payload in Governance.
    2. The Governance proposal's `.proposer` points back to the GovernanceProposer.

GP-03: Each instance in `rounds` must be in the rollup history of the GovernanceProposer's `REGISTRY`

GP-04: `rounds` keys are U256

GP-05: `payloadWithMostSignals` must be a valid payload address

GP-06: if a proposal has been executed, then `payloadWithMostSignals` is defined

GP-07: `payloadWithMostSignals` appears in `signalCount`

GP-08: A proposal cannot be executed without a quorum

The other direction is not true, as a proposal is not executed automatically upon reaching a quorum. Instead, it must be submitted via submitRoundWinner.

GP-09: Payloads in round accounting are valid addresses

GP-10: For each payload, signal count is in the valid range

This range is $(0, mostSignalCount]$.

GP-11: Every payload in the round accounting satisfies one of three conditions

1. The payload with mostSignalCount has not reached `QUORUM_SIZE`
2. Payload's signal count is strictly below mostSignalCount
3. The payload is the one with the most signals

GP-12: lastSignalSlot is in the valid range

This range is `[round * ROUND_SIZE, (round + 1) * ROUND_SIZE)`.

GP-13: The number of signals is in the right range

It does not exceed `lastSignalSlot % ROUND_SIZE + 1`. This property is very hard for Z3. It is not falsified.

7.6 GSE

GSE-01: Basic GSE state validity assertions

- GSE-01-01: `OWNER` is a governance address
- GSE-01-02: `ASSET` is an asset address
- GSE-01-03: `ACTIVATION_THRESHOLD` and `EJECTION_THRESHOLD` are non-negative
- GSE-01-04: `configOf` keys and values are valid addresses
- GSE-01-05: `instances` keys are rollup addresses (including the bonus instance)
- GSE-01-06: `governance` is a governance address

GSE-02: all attesters in `configOf` are registered on a rollup

Violated: attesters can be ejected by withdraw()

GSE-03: `delegation.ledgers` keys are rollup addresses or the bonus instance

GSE-04: `delegation.ledgers` keys are registered in `instances`

GSE-05: instance supply is an ordered checkpointed trace with ascending timestamps

GSE-06: instance supply checkpoint timestamps are mirrored in total supply

GSE-07: ledger positions keys are valid addresses

GSE-08: delegatees are valid addresses

GSE-09: attester balances are non-negative

Zero-balance positions can be created by calling delegate() on a non-existing position (see Section 5.12)

GSE-10: attester delegatees appear in `votingAccounts`

Restrictions:

1. This is only true for positions that have not been created as described above - moveVotingPower() short-circuits on zero amounts and does not create an entry in `.votingAccounts` (see Section 5.12).

2. Delegations to 0x00 are possible, and temporarily burn voting power (see Section 5.14)

GSE-11: sum of instance balances equals instance supply

GSE-12: voting account keys are valid addresses

GSE-13: the zero address does not appear in `votingAccounts`

this is important: delegating away from 0x00 results in minting in `moveVotingPower`

GSE-14: delegatee's voting power is an ordered checkpointed trace with ascending timestamps

GSE-15: delegatee's voting power is non-negative

GSE-16: `powerUsed` is positive

Voting with zero amounts is possible (see Section 5.9)

GSE-17: for each proposal that the `delegatee` has `powerUsed` on, Governance contains that proposal

GSE-18: the proposal cannot still be pending

GSE-19: `powerUsed` cannot exceed the attester's voting power at the time of the proposal's `pendingThrough`

GSE-20: `votingPower` is the sum of delegations on all rollups

GSE-21: `delegation.supply` is an ordered checkpointed trace with ascending timestamps

GSE-22: total supply history timestamps coincide with at least one of the instance supply histories

GSE-23: `delegation.supply` at each checkpoint is the sum of all `delegation.ledgers[instance].supply` at that time

GSE-24: `instances` keys are either the bonus instance or a rollup address

GSE-25: `instances` rollups other than the bonus instance are in the history of `rollups`

Violated: `rollups` only contains the "latest-at-timestamp" rollups – if `addRollup` is called multiple times in the same block, only the latest one remains in `rollups`.

GSE-26: `instances[instance]` is an ordered checkpointed trace with ascending timestamps

GSE-27: attesters registered in `instances[instance]` have a config in `configOf`

GSE-28: `rollups` is an ordered checkpointed trace with ascending timestamps

GSE-29: `rollups` values are rollup addresses

GSE-30: the bonus instance does not appear in the `rollups` history

GSE-31: `rollups` values are registered in `instances`

7.7 Slasher

SLASHER-01: `PROPOSER` is a valid slashing proposer

SLASHER-02: `VETOER` is an address

SLASHER-03: `vetoedPayloads` is a subset of payload addresses

7.8 SlashingProposer

SP-01: Basic SlashingProposer state validity assertions

- SP-01-01: `INSTANCE` is a rollup address
- SP-01-02: `SLASHER` is a valid Slasher address
- SP-01-03: round keys are exactly the `INSTANCE`
- SP-01-04: nonce keys are exactly the `INSTANCE`
- SP-01-05: `QUORUM_SIZE` is U256
- SP-01-06: `ROUND_SIZE` is positive, U256
- SP-01-07: `2 * QUORUM_SIZE > ROUND_SIZE`
- SP-01-08: `QUORUM_SIZE <= ROUND_SIZE`
- SP-01-09: `EXECUTION_DELAY` is U256
- SP-01-10: `LIFETIME_IN_ROUNDS > EXECUTION_DELAY_IN_ROUNDS`
- SP-01-11: `LIFETIME_IN_ROUNDS` is U256

SP-02: round keys are U256

SP-03: `payloadWithMostSignals` must be a valid payload address

SP-04: if a proposal has been executed, then `payloadWithMostSignals` is defined

SP-05: `payloadWithMostSignals` has non-zero signal count

SP-06: A proposal cannot be executed without a quorum

The other direction is not true, as a proposal is not executed automatically upon reaching a quorum. Instead, it must be submitted via submitRoundWinner.

SP-07: Payloads in round accounting are valid addresses

SP-08: For each payload, signal count is in the valid range

This range is $(0, mostSignalCount]$.

SP-09: Every payload in the round accounting satisfies one of three conditions

1. The payload with mostSignalCount has not reached `QUORUM_SIZE`
2. Payload's signal count is strictly below mostSignalCount
3. The payload is the one with the most signals

This property is very hard for Z3. It is not falsified.

SP-10: lastSignalSlot is in the valid range

This range is `[round * ROUND_SIZE, (round + 1) * ROUND_SIZE)`.

SP-11: The number of signals is correct.

It does not exceed `lastSignalSlot % ROUND_SIZE + 1`. This is particulary hard property for Z3. It is verified.

8 Disclaimer

The authors of this report have made their best efforts to specify the protocol behavior and to verify it using the tools described. Due to the inherent limitations of manual code review and the dependence of symbolic tools on expert-guided inputs, we provide no warranty of any kind, express or implied. This report does not constitute a guarantee that the contracts in scope are secure in all respects, nor should it be relied upon as the sole basis for making decisions. To the maximum extent permitted by applicable law, the authors disclaim liability for any damages arising from or in connection with the use of this report, whether in contract, tort, or otherwise, including but not limited to indirect, incidental, or consequential damages. This report is intended for professional use only.

References

Clarke, Edmund M., Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, eds. 2018. *Handbook of Model Checking*. Springer. https://doi.org/10.1007/978-3-319-10575-8.

Konnov, Igor, Jure Kukovec, and Thanh-Hai Tran. 2019. "TLA+ Model Checking Made Symbolic." *Proc. ACM Program. Lang.* 3 (OOPSLA): 123:1–30. https://doi.org/10.1145/3360549.

Lamport, Leslie. 2002. *Specifying Systems, the TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley. http://research.microsoft.com/users/lamport/tla/book.html.

Otoni, Rodrigo, Igor Konnov, Jure Kukovec, Patrick Eugster, and Natasha Sharygina. 2023. "Symbolic Model Checking for TLA+ Made Faster." In *TACAS*, 13993:126–44. LNCS. Springer.

Tange, Ole. 2011. "GNU Parallel: The Command-Line Power Tool." *Login Usenix Mag.* 36 (1). https://www.usenix.org/publications/login/february-2011-volume-36-number-1/gnu-parallel-command-line-power-tool.