Security Audit Report

# dYdX Q3 2025: Proposer Selection Updates

Last Revised
2025/09/19

Authors:
Martin Hutle, Vukašin Dokmanović

# Contents

# Audit overview

## The project

In September 2025, dYdX engaged Informal Systems to perform a security audit of a new feature that modifies the block production mechanism to provide finer-grained control over which validators are eligible to propose blocks. Instead of all active validators participating equally in the proposer rotation, the feature introduces a validator-level configuration flag (`ProposeDisabled`) that determines eligibility.

A new governance-controlled message type has been introduced to manage this mechanism. When executed via a governance proposal, the message stores an explicit list of validator addresses permitted to participate in block production. Whenever the proposer set needs to be refreshed, each validator's `ProposeDisabled` flag is recalculated based on this list and then propagated to CometBFT, ensuring proposer rotation reflects the governance-approved configuration.

## Scope of this report

The audit focused on evaluating the correctness and security properties of the changes to the features in dYdX's forks of Cosmos SDK and CometBFT.

## Audit plan

The audit was conducted between September 10th to September 19th, 2025, by the following personnel:

- Martin Hutle
- Vukašin Dokmanović

## Conclusions

The audit team has thoroughly reviewed the changes to the proposer set management logic. We commend the development team for the high quality of their implementation. Overall, no critical security or liveness issues were identified during this audit.

That said, we highlight two notable areas where additional safeguards could further strengthen the robustness of the system:

1. **Assumption on Proposer Availability**
   - Without further checks, redelegations, slashing, or jailing could reduce the active proposer set to a single validator. In such a scenario, if that validator fails during proposal rounds, consensus may stall indefinitely. While this risk is mitigated by the assumption that there is always at least one correct active proposer, this assumption is stricter than the standard CometBFT safety model. We recommend formalizing this assumption and implementing safeguards to detect violations (e.g., by enforcing a minimum proposer stake threshold or triggering fallback mechanisms such as automatically re-enabling all validators as proposers).
2. **Lack of Uniqueness Enforcement in Proposer Sets**
   - The current `checkProposerSetInvariants()` implementation does not enforce uniqueness, meaning duplicate proposers can be included in the proposer set and still pass validation. While this does not affect

current CometBFT behavior, it weakens governance semantics and could increase computational costs in certain loops. We recommend adding explicit checks to enforce uniqueness when setting proposer sets.

In summary, the audit confirms that the system's proposer selection and consensus integration are secure, deterministic, and free from critical vulnerabilities. The identified findings are non-critical and primarily concern long-term safety guarantees and robustness against unlikely edge cases. Addressing them will further strengthen the system's resilience against governance misconfigurations and unexpected validator dynamics.

# Audit Dashboard

## Target Summary

- **Type:** Protocol and implementation
- **Platform:** Go
- **Artifacts:** The following commits
  - Commits on CometBFT fork:
    - → commit 1↗
    - → commit 2↗
  - Commits on Cosmos SDK fork:
    - → commit 1↗
    - → commit 2↗
  - Commit on v4-chain:
    - → commit 1↗

## Engagement Summary

- **Dates**: September 10th, 2025 → September 19th,
- **Method**: Manual code review

## Severity Summary

| Finding Severity | Number |
|---|---|
| Critical | 0 |
| High | 1 |
| Medium | 0 |
| Low | 0 |
| Informational | 2 |
| **Total** | 3 |

# System Overview

The Proposer Selection Updates modify the block production mechanism in dYdX's consensus layer to allow more control over which validators are eligible to propose blocks. Instead of all active validators participating equally in the proposer rotation, the system introduces a configuration flag (a boolean field) at the validator level.

Originally introduced as a `CanPropose` flag, the mechanism was later inverted into a `ProposeDisabled` flag to ensure backward compatibility and simplify integration with existing code paths. With the original `CanPropose` design, validators would need to explicitly set the flag to `true` in order to propose blocks. This defaulted all existing validators to a "disabled" state, requiring a tightly coordinated network-wide upgrade to restore block production. In contrast, the `ProposeDisabled` approach defaults to `false` for validators that do not explicitly set the field, meaning they remain eligible to propose blocks, as intended. This approach ensures that the default behavior remains unchanged for validators, while still allowing dYdX to explicitly restrict certain validators from proposing blocks.

To achieve this, coordinated changes were applied across both the CometBFT and Cosmos SDK forks to align proposer selection logic with the new flag. The change also highlights a separation of concerns between the Cosmos SDK and CometBFT layers. While the SDK is responsible for validator set management and governance-driven updates (through the `MsgSetProposers` flow), CometBFT applies the proposer selection logic during consensus.

From a protocol perspective, these changes are based on several assumptions about proposer availability and system safety. The design assumes that the selected proposer set is generally more reliable and engaged than the wider validator set, and therefore more likely to ensure liveness. However, if all validators in the proposer set become unavailable (either by downtime or by exiting the active validator set through redelegation), the network may halt. This risk is considered acceptable under the reasoning that "being outside the active set" is equivalent to being down.

A minimum threshold of five proposers was chosen as a safeguard: if the proposer set falls below this number, all active validators automatically regain proposer eligibility. This mechanism prioritizes liveness over performance, ensuring the network can continue producing blocks even if the curated proposer set becomes too small.

With respect to fairness, the standard CometBFT proposer priority algorithm was chosen, where priorities are adjusted for all validators, but only enabled validators are eligible to propose. While this can lead to oscillations and raise questions about fairness across the proposer set, internal tests suggest that excluded validators do not gain outsized influence if later re-enabled.

## Architecture Overview

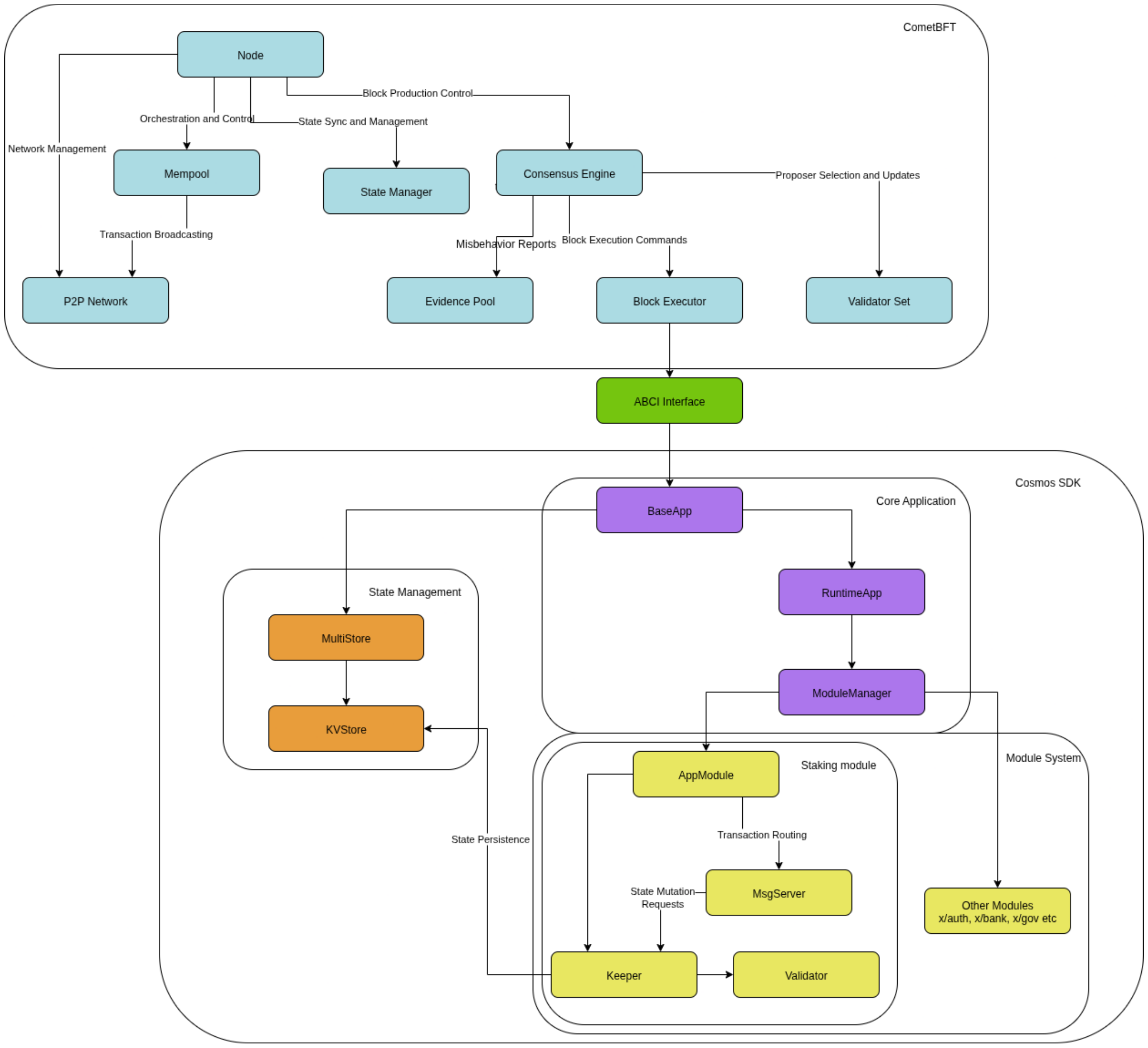

Figure 1: Architecture Overview Diagram

## Protocol Overview

**`FinalizeBlock` → `x/staking` `EndBlocker` Flow:**

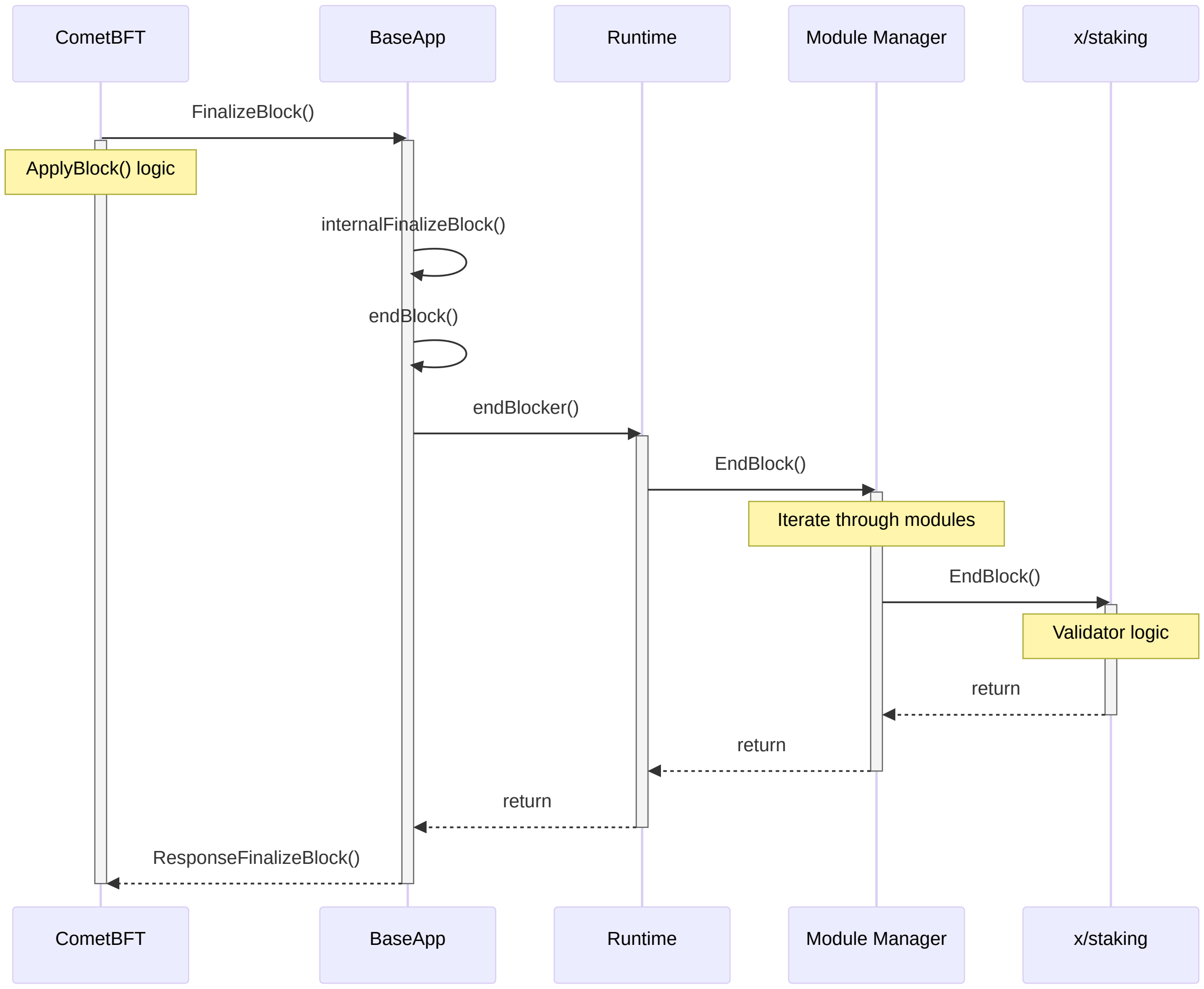

Figure 2: `FinalizeBlock` → `x/staking` `EndBlocker` Flow

The `ApplyBlock()` logic implemented in the CometBFT's `BlockExecutor` is described in one of the following diagrams.

The `FinalizeBlock()` function serves as the ABCI interface orchestrator that manages optimistic execution by checking if pre-computed results from a previous optimistic run can be reused, or if the block needs to be re-executed from scratch. It handles concurrency control, streaming service hooks, and delegates the actual block execution to `internalFinalizeBlock()` while managing the overall execution flow and response formatting.

The `internalFinalizeBlock()` function performs the core block execution by setting up the execution context and sequentially running `PreBlock()`, `BeginBlock()`, transaction processing, and `EndBlock()` phases. It handles the actual state transitions, transaction validation, event collection, and gas metering while supporting cancellation for optimistic execution scenarios.

On the level of Runtime, the `EndBlocker()` method delegates to `ModuleManager.EndBlock(ctx)`.

Module Manager calls `EndBlock()` on all modules that implement `HasABCIEndBlock` interface and executes modules in the configured order (from app config).

The `x/staking` module implements the `module.HasABCIEndBlock` interface. The `EndBlock()` method calls the `keeper.EndBlocker(ctx)` method. The rest of the validator logic implemented in the `x/staking` module is described in the next diagram.

**Validator logic in `x/staking`:**

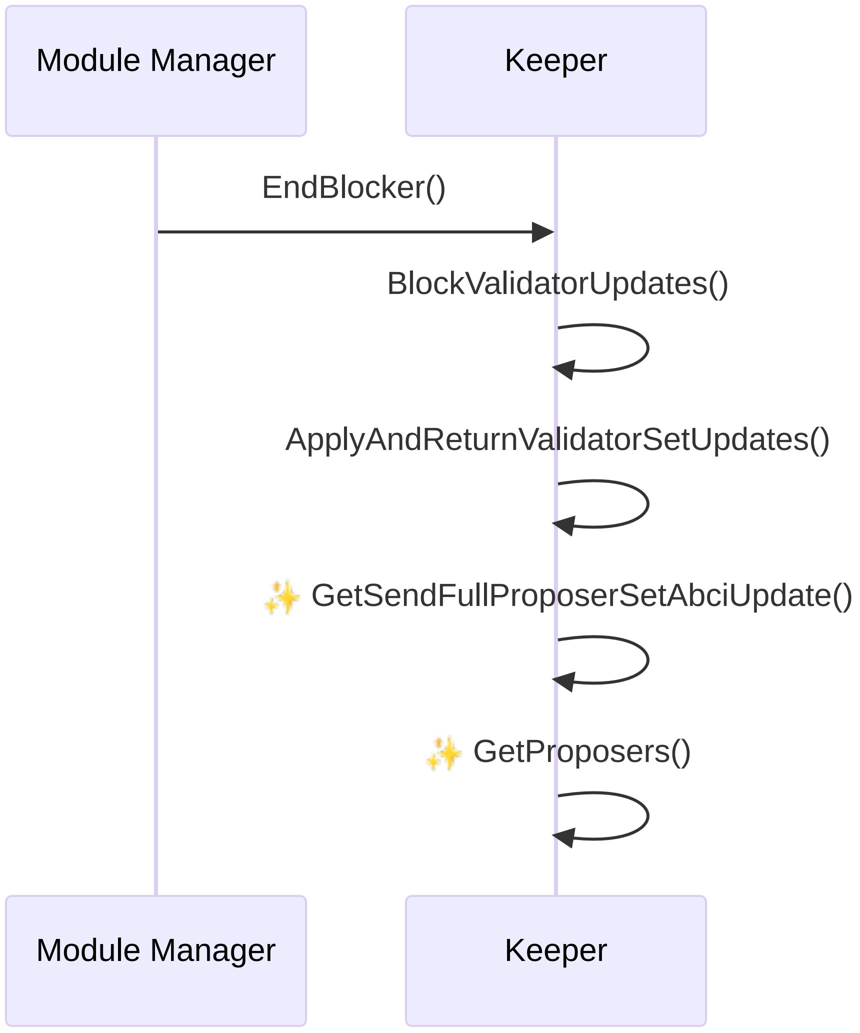

Figure 3: Entry Point and Proposer Retrieval

The `EndBlocker()` function is called by the Module Manager at the end of each block to update the validator set. It simply delegates to `BlockValidatorUpdates()` and handles telemetry measurement for performance monitoring.

The main orchestrator that calculates and applies all validator set changes for the current block is the `BlockValidatorUpdate()` function. It coordinates validator power updates, handles unbonding/redelegation completions, and returns the final validator updates to CometBFT. This ensures the consensus engine stays synchronized with staking state changes.

The `ApplyAndReturnValidatorSetUpdates()` function processes the active validator set by iterating through validators by power (highest first) and determining state transitions. It handles bonding/unbonding validators, calculates power changes, and prepares ABCI validator updates for CometBFT.

`sendFullProposerSetUpdate` is obtained by calling the `GetSendFullProposerSetAbciUpdate()` function. This function checks the `SendFullProposerSetAbciUpdateKey` form the `x/staking` module's state store, which is updated by sending the adequate `MsgSetProposer` message (described in the later diagram).

`GetProposers()` fetches the current list of validators eligible to propose blocks. This list is used to set the `ProposeDisabled` field in validator updates, controlling which validators can propose blocks.

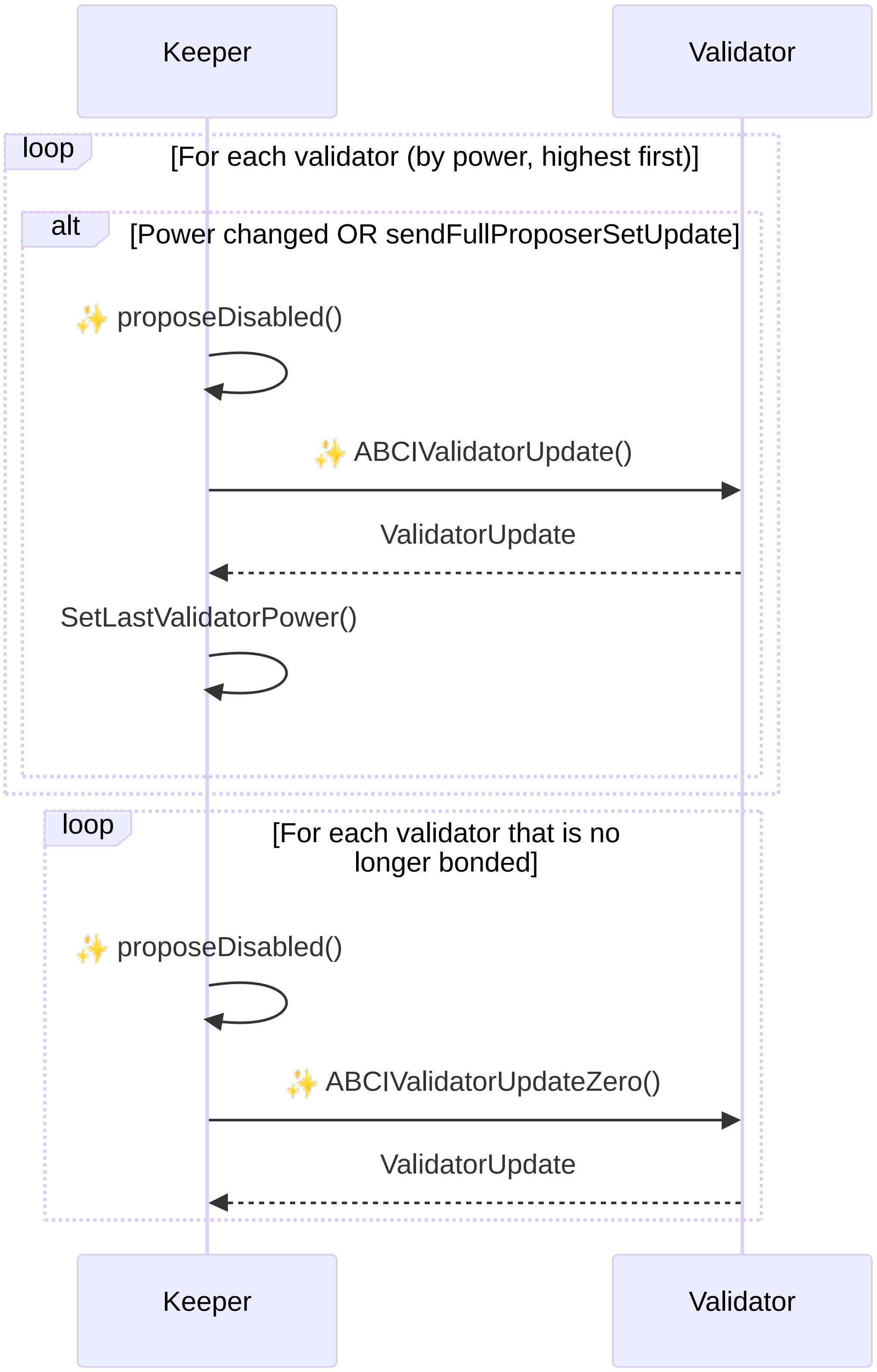
Keeper
Validator
loop
[For each validator (by power, highest first)]
alt
[Power changed OR sendFullProposerSetUpdate]
✨ proposeDisabled()
✨ ABCIValidatorUpdate()
ValidatorUpdate
SetLastValidatorPower()
loop
[For each validator that is no longer bonded]
✨ proposeDisabled()
✨ ABCIValidatorUpdateZero()
ValidatorUpdate
Keeper
Validator

Figure 4: Processing Active and Leaving Validators

`proposeDisabled()` A closure function that determines if a validator should have block proposal rights disabled. Returns true if a proposer set exists and the validator is not in it, used for governance-controlled block proposal permissions.

`ABCIValidatorUpdate` function returns an `abci.ValidatorUpdate` from a staking validator type with the full validator power, while the `ABCIValidatorUpdateZero` returns an `abci.ValidatorUpdate` from a staking validator type with zero power used for validator updates. The only change here is the addition of the `proposeDisabled` flag.

The `SetLastValidatorPower()` persists the validator's current voting power to the store for use in the next block's validator set calculations. This maintains the historical power information needed to detect power changes between blocks.

`ABCIValidatorUpdateZero()` creates an ABCI validator update with zero voting power to remove a validator from CometBFT's active set. This effectively tells the consensus engine to stop considering this validator for block validation and proposal. The only change here is the addition of the `proposeDisabled` flag.

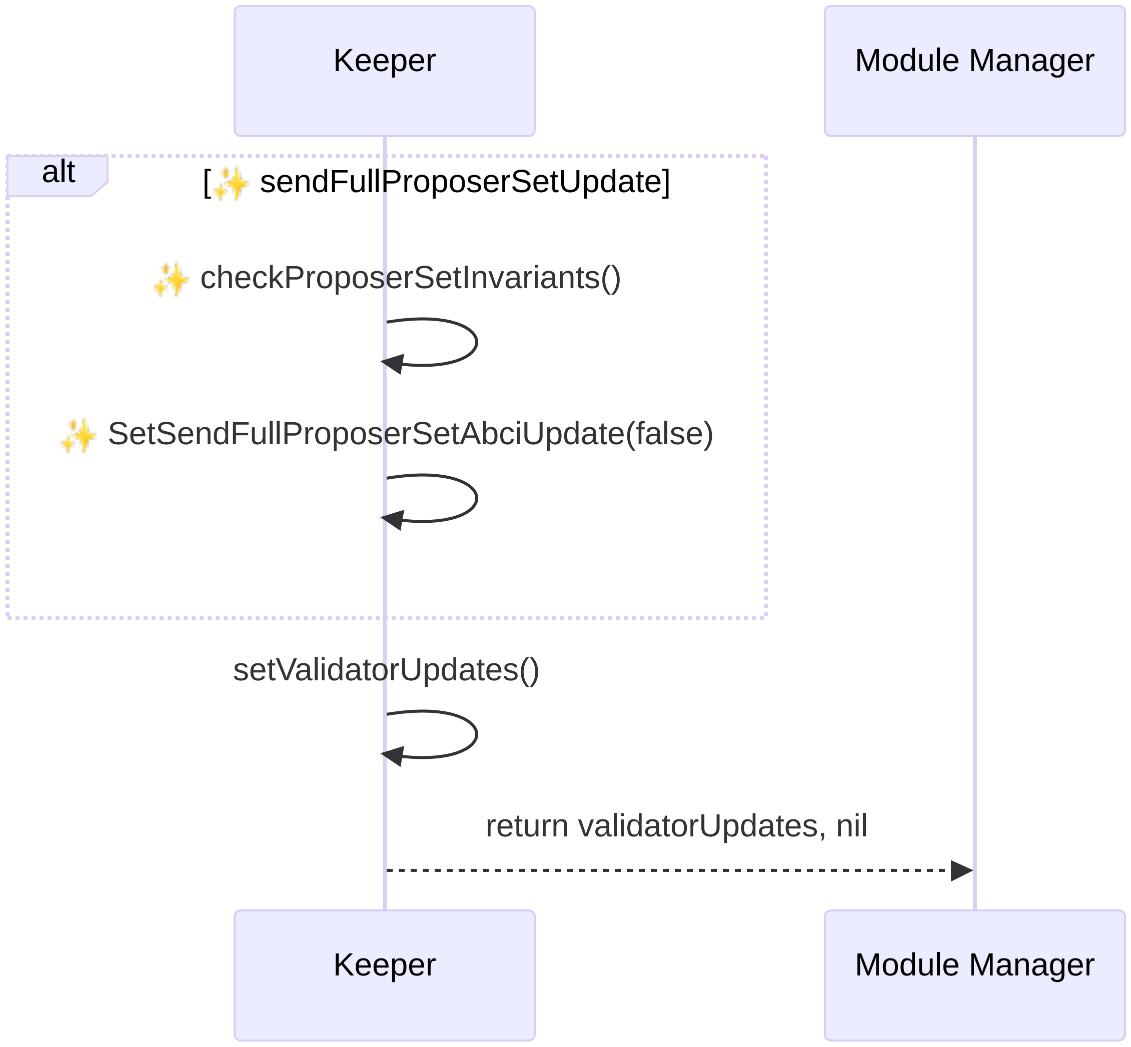
Keeper
Module Manager
alt
[✨ sendFullProposerSetUpdate]
✨ checkProposerSetInvariants()
✨ SetSendFullProposerSetAbciUpdate(false)
setValidatorUpdates()
return validatorUpdates, nil
Keeper
Module Manager

Figure 5: Finalization and Return

`checkProposerSetInvariants()` validates that the updated proposer set maintains the required invariants (e.g., no duplicate validators, valid addresses). If validation fails, it logs errors and defaults to allowing all validators to propose blocks as a safety mechanism.

The `SetSendFullProposerSetAbciUpdate(false)` function resets the full proposer set update flag back to false after processing. This ensures subsequent blocks return to incremental update behavior rather than sending complete validator sets every block.

**`MsgSetProposers` Execution Flow:**

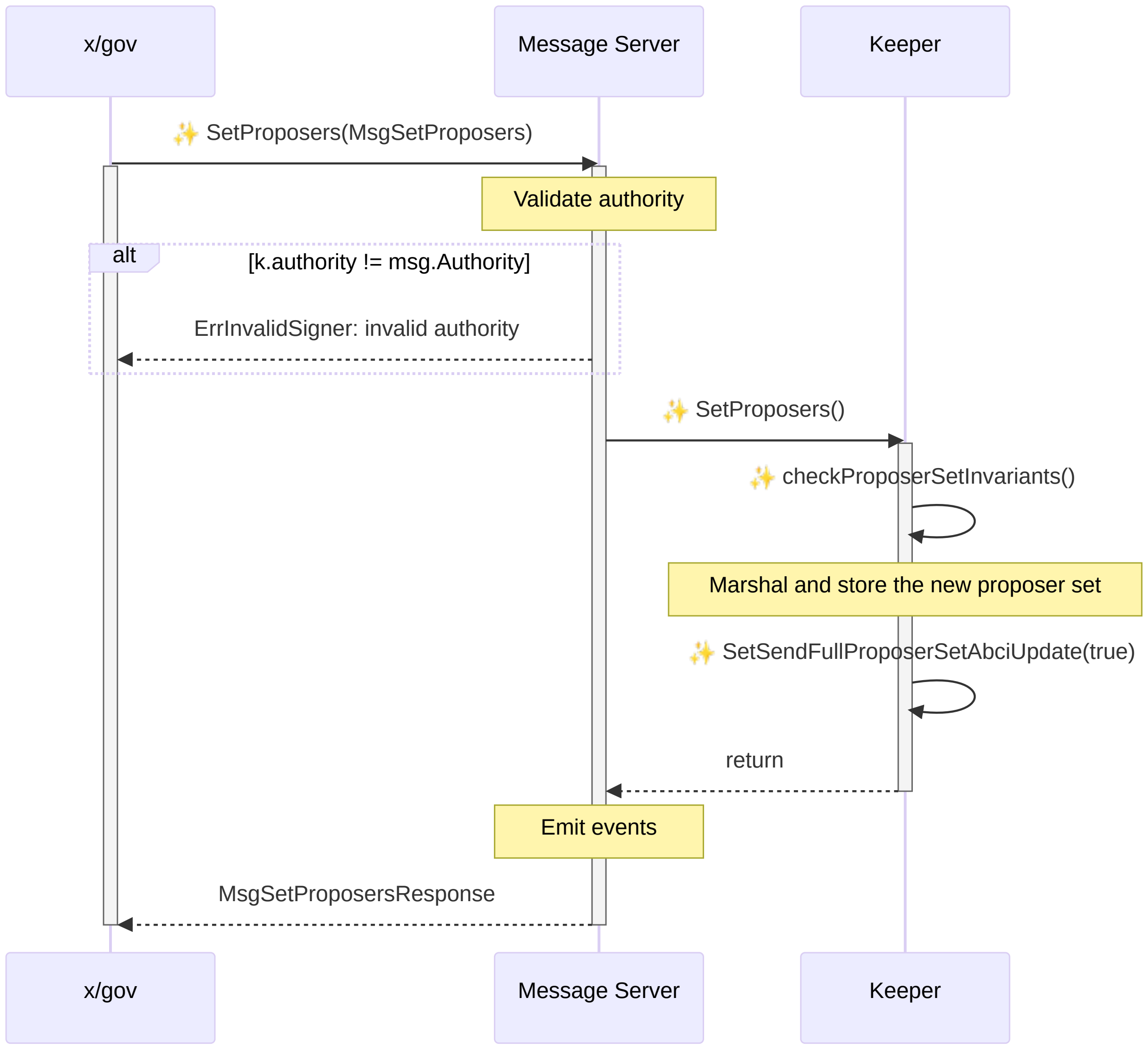
x/gov
Message Server
Keeper
✨ SetProposers(MsgSetProposers)
Validate authority
alt
[k.authority != msg.Authority]
ErrInvalidSigner: invalid authority
✨ SetProposers()
✨ checkProposerSetInvariants()
Marshal and store the new proposer set
✨ SetSendFullProposerSetAbciUpdate(true)
return
Emit events
MsgSetProposersResponse
x/gov
Message Server
Keeper

Figure 6: `MsgSetProposers` Execution Flow

When the `SetProposers()` method from the Message Server is called, it first makes sure that the authority is valid. If that is the case, it delegates the call to the `SetProposers()` method implemented in the Keeper.

Pay attention to the `true` flag in the `SetSendFullProposerSetAbciUpdate()` function. The flag is being stored in the staking module's state, waiting for the next `EndBlock`.

When the very next block is processed, the ABCI `EndBlock` sequence ends up in the `ApplyAndReturnValidatorSe-tUpdates()` function that checks if a full proposer set update is needed by calling the `GetSendFullProposerSetAb-ciUpdate()` function.

If the update is needed, each validator has its `ProposeDisabled` flag updated. It is set to `false` if the validator is in the new proposer set, and `true` if it is not.

In the end, the `SetSendFullProposerSetAbciUpdate()` function is called to reset the flag to `false`, so this update only happens once.

**CometBFT `ApplyBlock()` Execution Flow:**

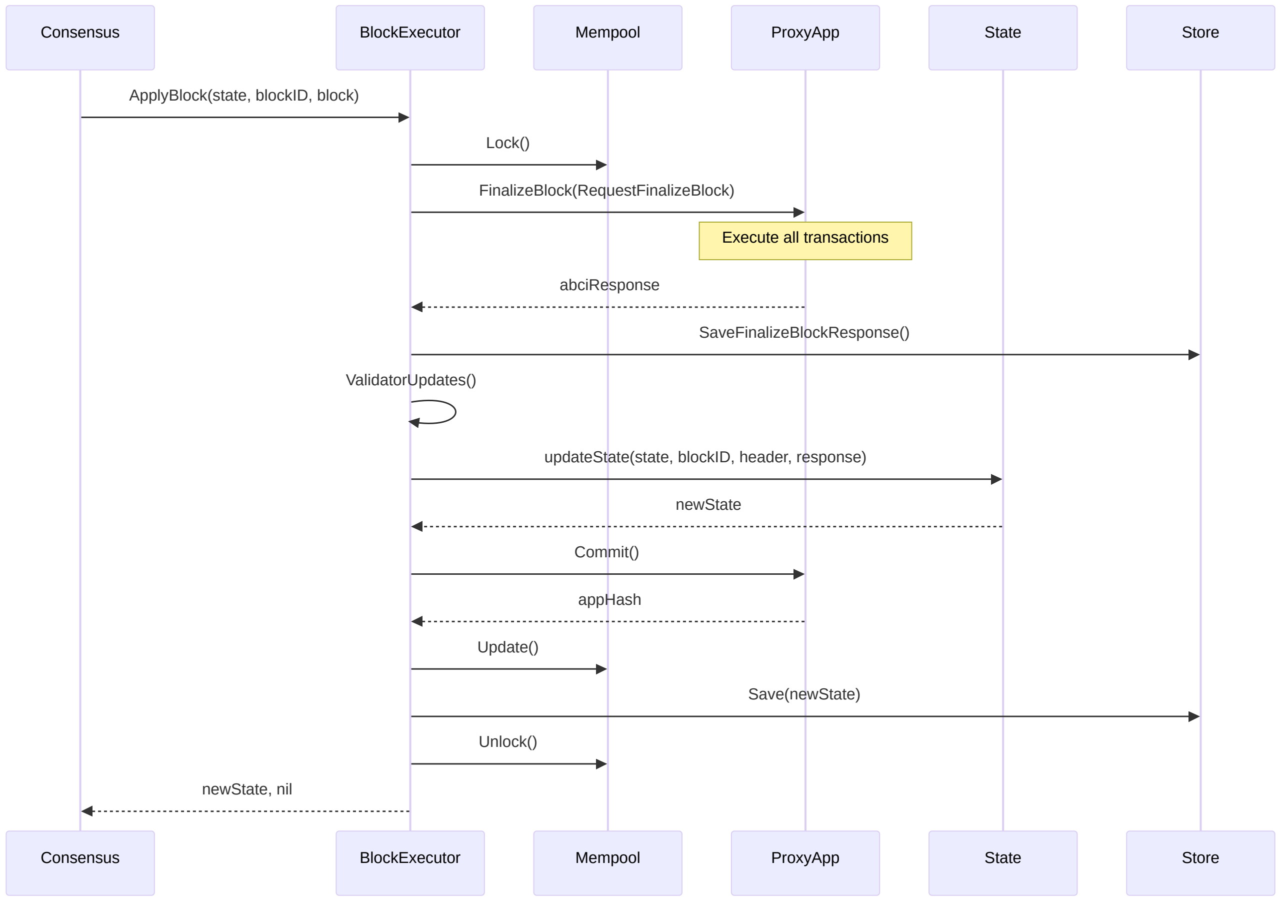

Figure 7: CometBFT `ApplyBlock()` Execution Flow Diagram

The `FinalizeBlock()` function sends block to the application layer for processing via ABCI.

The `SaveFinalizeBlockResponse()` function saves the ABCI response before committing; by doing so, it ensures crash recovery capability.

The `ValidatorUpdates()` function converts the ABCI `ValidatorUpdate` messages to internal CometBFT `Validator` types.

The `updateState()` function creates a new blockchain state by applying changes from a processed block, including validator set updates and consensus parameter changes. It handles the rotation of validator sets (*current* → *last*, *next* → *current*, *updated* → *next*) and implements a delayed activation system where validator changes at height *H* take effect at height *H+2*, while consensus parameter changes at height *H* take effect at height *H+1*.

The `Commit()` function commits application state changes, updates the mempool after commit, and returns height for potential block pruning.

**CometBFT Proposer Selection Flow:**

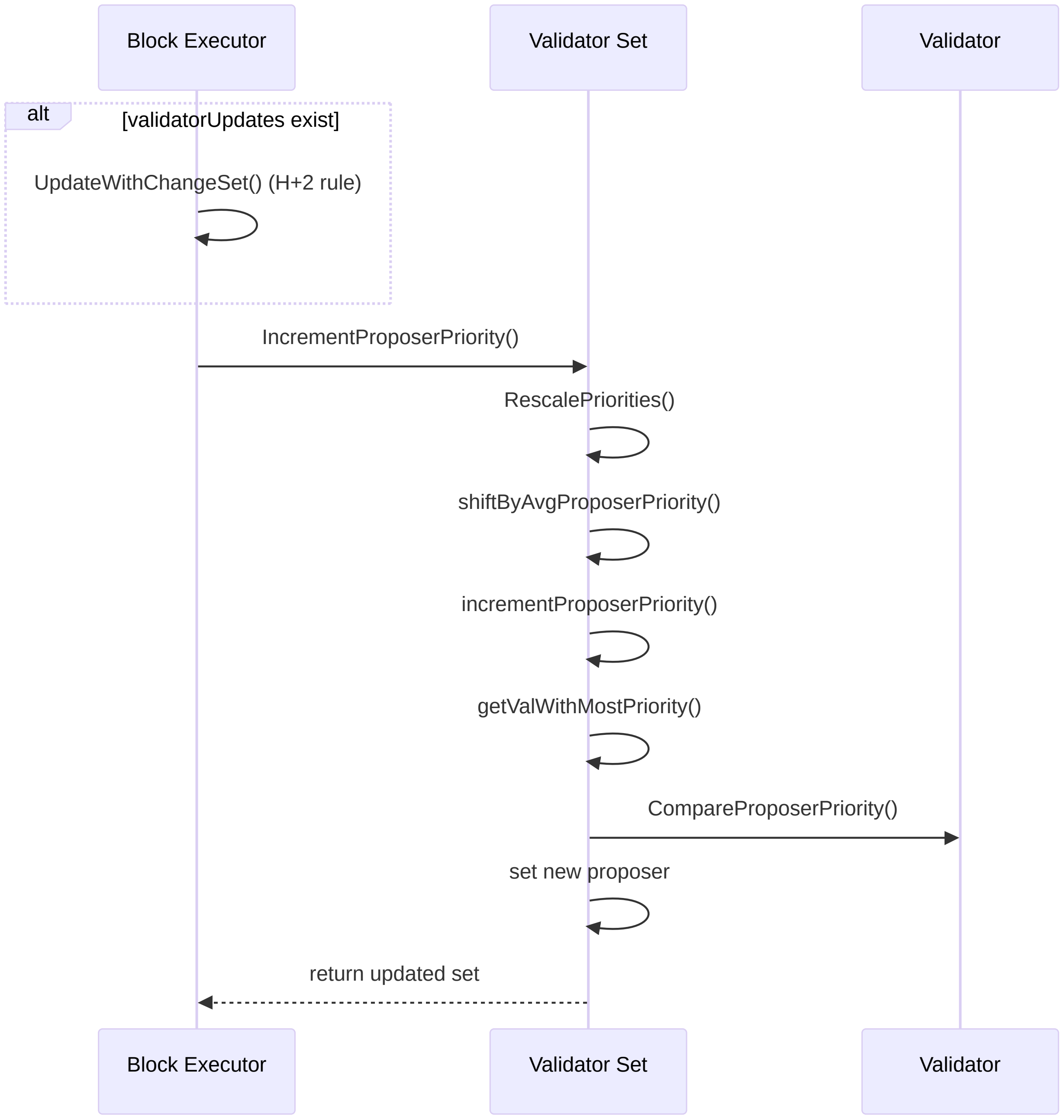

Figure 8: CometBFT Proposer Selection Flow Diagram

The `UpdateWithChangeSet()` function applies validator updates (additions, removals, voting power changes) to the validator set while enforcing the *H+2* rule, where changes at height *H* take effect at height *H+2*. It validates updates, computes new priorities for added validators, and maintains the integrity of the validator set.

The `IncrementProposerPriority()` function is the main entry point for proposer rotation that rescales priorities to prevent overflow, normalizes them around zero, and then executes the core proposer selection algorithm. It calls the internal `incrementProposerPriority()` method and updates the validator set's proposer field.

The `RescalePriorities()` function prevents integer overflow by rescaling all validator priorities when the difference between the max and min priorities becomes too large. It divides all priorities by a calculated ratio to keep them within safe bounds while maintaining their relative ordering.

The `shiftByAvgProposerPriority()` function normalizes validator priorities by subtracting the average priority from each validator's priority, centering the priority distribution around zero. This prevents priorities from drifting too far from zero over time.

The `incrementProposerPriority()` function implements the core proposer selection algorithm by adding each validator's voting power to their priority, finding the validator with the highest priority, then subtracting the total voting power from the winner's priority. This creates a weighted round-robin rotation.

The `getValWithMostPriority()` function finds and returns the validator with the highest proposer priority among all validators that are eligible to propose (not disabled). It uses `CompareProposerPriority()` to compare validators and panics if no validator can propose.

The only change is that the function only considers validators that can propose.

The `CompareProposerPriority()` function compares two validators' proposer priorities and returns the one with higher priority, using the validator address as a tie-breaker for deterministic selection. It returns the validator with higher priority, or the one with the lexically smaller address if priorities are equal.

**Other notable changes in the CometBFT fork:**

The `findPreviousProposer()` function finds the validator with the lowest proposer priority (opposite of `findProposer()`). It uses the reverse of `CompareProposerPriority()` to find the validator that would have been the previous proposer.

The only change is that the function only considers validators that can propose.

The `findProposer()` function finds the validator with the highest priority among non-disabled validators. It iterates through all validators, using `CompareProposerPriority()` to find the one with the highest priority. Panics if no validators can propose (all disabled), which indicates an invalid network state.

# Threat Model

## Integration Model & Threats

### Safety

**Property 1:** There are always at least one correct proposer in the set of validators.

- Threat: The number of proposers may fall below f+1 because of redelegations, slashing or jailing.

  This threat does **not hold only by assumption**. According to the discussions with the development team during the audit, it is assumed that there is at least one active proposer in the validator set at any time.

  We believe this assumption maybe hard to maintain and summarize our concerns and recommendations in the finding "Redelegations, slashing and jailing can lead to scenarios where there is no correct proposer".

- Threat: The number of proposers may fall below f+1 because of a `MsgSetProposers`.

  Similar as discussed above, there is no correlation with the number 5 that is used as minimum number of proposers and the stake they represent. Again it is in general possible (without further assumption) that all proposers fail if they do not represent more than one third of the stake.

- Threat: The violation of the liveness assumptions on proposers is not detected.

  In the current implementation, changes that can lead to a scenario where no progress is possible remain unnoticed.

  Even if the design decision is to accept a stronger assumption on the correctness of validators, they should be properly formalized and safeguards should be installed in order to detect violations of this assumption and either prevent these situations or fail in a coordinated way. Depending on the precise semantics of such an assumption, the algorithm could detect/prevent scenarios where the set of proposers fall or their associated stake below some threshold due to redelegations/slashing/jailing. The implementation could then eg. automatically enable all proposers again or prevent the redelegation or halt the chain by a well-defined panic.

The property therefore holds only by additional assumptions that need to be properly defined. Violations of these assumptions should be detected and either prevented or a transferred in a clean error state.

**Property 2:** A change in the set of proposers at height *H* by a `MsgSetProposers` leads takes effect exactly at height *H+2*.

- Threat: Changes in the validator set at *H+1* have an effect on the proposer set for *H+2* (should be only for *H+3* then).

  This threat **doesn't hold**. When validator updates are received at height *H*, they are immediately applied to `state.NextValidators` and the code correctly sets `lastHeightValsChanged = header.Height + 1 + 1` (which equals *H+2*). The three-tier validator set rotation system (`LastValidators ← Validators ← NextValidators`) ensures that changes flow through the system with proper timing: at height *H+1*, the updated validators move from `NextValidators` to `Validators`; then, at height *H+2*, they move from `Validators` to `LastValidators` where they become active for consensus verification (validating commits and signatures). The proposer selection process occurs through a priority-based round-robin algorithm that operates on the currently active validator set. The *H+2* rule ensures that by the time updated validators are responsible for consensus verification, all nodes have had sufficient time to learn about the changes and synchronize their validator sets. The implementation correctly ensures that validator changes have a 2-block safety buffer before they bear full consensus responsibility, which prevents race conditions and ensures network-wide consistency during validator set transitions.

Cosmos SDK ensures that the proposer set changes from `MsgSetProposers` are only included in validator updates at `EndBlock(H)`, and therefore only applied in CometBFT starting at *H+2*.

The `MsgSetProposers` is executed in a transaction. In the `SetProposers()` method, there is a state update: `k.SetProposers(ctx, msg.Proposers)`, which stores the new proposer list.
After that, `k.SetSendFullProposerSetAbciUpdate(ctx, true)` sets the flag to be executed in the next `EndBlock()`, meaning that there are no validator updates sent to CometBFT before the next `EndBlock()`.

In the next `EndBlock()`, the validator updates with new `ProposeDisabled` are generated and sent to CometBFT.

- Threat: The changes conflict with a (unlikely but theoretically possible) governance proposal at *H+1*.

  This threat does **not hold.** Both validator set updates and governance-driven proposer set updates follow the same deterministic update pipeline and are serialized through `EndBlock`. Validator updates scheduled at height *H* take effect at *H+2*. A governance proposal (`MsgSetProposers`) included at *H* also takes effect at *H+2*, while one included at *H+1* takes effect at *H+3*. This ensures that proposer set changes never overtake validator set changes, and CometBFT always applies a consistent, correctly delayed update batch. As a result, there is no race condition or inconsistency between validator set updates and governance-driven proposer updates.

- Threat: The implementation is incorrect, some proposers are disabled or some non-proposers are enabled.

  The implementation has no issues, meaning that the threat **does not hold**.

  In CometBFT, the state transition happens in the `updateState()` method. This method does not independently validate the correctness of the `ProposeDisabled` assignments; it trusts the updates provided, meaning it only takes `validatorUpdates` and performs its logic. The correctness of the proposer set depends entirely on the accuracy of the `validatorUpdates` input to `updateState()`. If those updates are wrong, the proposer set will be wrong.

  The new updated proposer set is sent from the governance to the `x/staking` module via the `MsgSetProposers` message. This message in itself holds the authority address and the list of proposers (the list of operator addresses of validators who are eligible to propose blocks). The `msgServer.SetProposers()` method is called, which first validates the authority, and then calls the `keeper.SetProposers()` method, which is the one implementing the proposer set update logic.

  The `keeper.SetProposers()` implements a sanity check (`checkProposerSetInvariants()`) that performs a sanity check to ensure the proposer set is valid. These checks include making sure that all proposers have valid operator addresses, all proposers correspond to existing validators, and at least `MinBondedInProposerSet` (5) proposers are bonded.

  After the sanity checks, the proposer set is marshaled and stored using the `ProposerSetKey`. If any of these steps fail at some point, they return an error after which the transaction is reverted, meaning there are no possibilities for partial updates. These proposers can later be retrieved using `GetProposers()`.

  Which is done in the `ApplyAndReturnValidatorSetUpdates()` method. The set is used to check within the active validator set who should be in the updated proposer set, and those updates are then finally sent to CometBFT.

This property **holds**. Both validator set updates and governance-driven proposer set changes follow the same deterministic update pipeline, with updates included in `EndBlock(H)` and becoming effective in CometBFT at *H+2*. Even when a governance proposal is included at *H+1*, its effects only activate at *H+3*, preventing overtaking or conflicts with earlier validator updates. The Cosmos SDK enforces correctness through `checkProposerSetInvariants()` and atomic state updates, ensuring no proposer is incorrectly enabled or disabled. As a result, proposer set transitions are consistent, delayed appropriately, and resistant to race conditions or partial failures.

**Property 3:** After migration no validator is disabled by default (until there is a `MsgSetProposers` to restrict the set of proposers)

- Threat: `proposeDisabled` is not set to false during migration.

  This threat **does not hold**. In this case, no migration logic is needed because the upgrade only introduces new store keys and a new message type, without requiring data restructuring or key format changes. Since `proposeDisabled` is a boolean field, its default value is `false`, meaning validators are not disabled at initialization. The only mechanism that can disable a validator's proposer role is an explicit governance action (`MsgSetProposers`). The initialization path is handled by `InitGenesis()`. If the genesis state specifies a proposer set, it is written directly to the store; if not, the keeper defaults to enabling all validators as proposers by calling `SetProposers()`. As a result, migration cannot inadvertently disable validators, and proposer availability is preserved by default.

This property **holds**. After migration, validators are not disabled by default. Either the proposer set is explicitly set in the genesis state, or all validators are enabled as proposers by default. The `proposeDisabled` flag is initialized safely, and only explicit governance actions can modify the proposer set. Therefore, migrations do not threaten validator proposer availability.

## Protocol Model & Threats

### Safety

**Property 4:** After there was a governance proposal, only those processes from the proposal are selected as proposers.

- Threat: There are some implementation errors in the handling of the set of proposers.

  This threat **does not hold.** The proposer set management logic in the Cosmos SDK and its integration with CometBFT were reviewed in detail, with a focus on proposer set updates, `proposeDisabled` flag handling, and state transitions during validator jailing, unjailing, bonding, and governance-driven updates (`MsgSetProposers`). The code paths ensure that proposer set changes are always derived deterministically from the on-chain state and applied in `EndBlock`, guaranteeing consistency across all validators.

  Potential edge cases, such as duplicate proposers, incorrect flag resets, or misaligned validator updates, were examined and either mitigated by existing safeguards or confirmed to not affect fairness or consensus safety. Additional targeted test cases were executed to fact-check these scenarios, further confirming that no exploitable errors exist in proposer set handling.

  In conclusion, the current implementation correctly and deterministically manages the proposer set without exposing consensus or fairness vulnerabilities.

- Threat: Some validators stay proposers during some later re-start/migration of the chain.

  This threat **does not hold**. The proposer set can be initialized via the `InitGenesis()` method, but this only occurs in specific scenarios:

  - When starting a new chain from height 0.
  - When synchronizing a node from the genesis state (though this is uncommon, as most nodes use state sync).
  - During upgrades that export the state to a new genesis file and restart the chain.

  If the proposer set is explicitly defined in the genesis state, it is restored accordingly; otherwise, all validators are defaulted to proposers. Since the proposer state is deterministically reconstructed from the genesis data and `proposeDisabled` defaults to `false`, there is no risk of unintended validators persisting as proposers across migrations or restarts. Any restriction of the proposer set must occur explicitly through a governance action (`MsgSetProposers`), ensuring consistency and preventing unintended carryover of proposer status.

- Threat: A validator that is new in the set of (active) validators is enabled by default although there was a gov proposal in the past.

This threat **does not hold**. When a `MsgCreateValidator` transaction is submitted, the message server performs validation (including uniqueness checks) and creates a new validator object, which is stored in state via `SetValidator`.

At the end of the block, `ApplyAndReturnValidatorSetUpdates()` enforces proposer eligibility based on the current proposer set:

- If a proposer set is already defined, the method retrieves this set and iterates over all validators. For each validator, if its operator address is present in the proposer set, its `proposeDisabled` flag is set to `false`; otherwise, the flag is set to `true`. This ensures that newly created validators default to *disabled* unless they are explicitly included in the governance-defined proposer set.
- If no proposer set is defined, the retrieved proposer set is empty, which is interpreted as "all validators are eligible." In this case, `ApplyAndReturnValidatorSetUpdates()` iterates over all validators and sets their `proposeDisabled` flag to `false`. A newly created validator is therefore enabled by default, but this is consistent with the semantics of an undefined proposer set.

As a result, the system maintains consistency with governance decisions:

- Validators are restricted to the governance-defined proposer set if one exists.
- Otherwise, all validators (including new ones) are eligible proposers by design.

Thus, there is no risk that a new validator bypasses proposer restrictions established by a prior governance proposal.

- Threat: The `proposeDisabled` flag is not set properly after unjailing or after re-joining the active set.

  This threat **does not hold**. When a validator is jailed, its `proposeDisabled` flag is not explicitly updated; however, this does not introduce inconsistencies. The key mechanism is the `DeleteValidatorByPowerIndex` call within `jailValidator()` (`x/staking/keeper/val_state_change.go`↗), which removes the validator from the power index. As a result, in `ApplyAndReturnValidatorSetUpdates()`, the jailed validator is excluded from the iteration over active validators (`ValidatorsPowerStoreIterator`).

  For validators that are no longer bonded (including jailed ones), the Cosmos SDK performs an additional pass: if such a validator exists in the proposer set, an `ABCIValidatorUpdateZero` is emitted, ensuring its voting power is set to zero in CometBFT. Since CometBFT removes any validator with zero power from both the validator set and the proposer set, the unchanged `proposeDisabled` flag becomes irrelevant because the validator is excluded from the consensus.

  When a validator is unjailed, it re-enters the `ValidatorsPowerStoreIterator`. At the next `EndBlocker()`, `ApplyAndReturnValidatorSetUpdates()` checks whether the validator's operator address is present in the proposer set. If so, its `proposeDisabled` flag is updated accordingly. Its voting power is also restored, and a normal `ABCIValidatorUpdate` is emitted, reintegrating it into the consensus process.

  Thus, although the `proposeDisabled` flag itself is not actively reset at jail/un-jail events; the surrounding mechanisms (power index updates, proposer set checks, and ABCI updates) guarantee correct behavior. There is no risk of a jailed validator proposing or of an unjailed validator being improperly excluded.

In conclusion, this property **holds**. Proposer set updates in the Cosmos SDK are handled deterministically through governance actions (`MsgSetProposers`), validator lifecycle events, and migration logic, ensuring only the intended validators are eligible as proposers. Edge cases such as restarts, migrations, validator creation, jailing, and unjailing were reviewed in detail, and in each case, proposer eligibility is reconstructed consistently from state. The `proposeDisabled` flag behaves as expected, with proposer status only changing via explicit governance updates or validator lifecycle changes. Overall, no scenarios were found where validators outside of the governance-defined set can persist or gain proposer rights incorrectly.

### Liveness

**Property 5:** The proposer selection algorithm is fair.

- Threat: If some processes are never selected they accumulate highest priority. Due to integer division and rounding there is no fair rotation between the proposers.

  After extensive testing, this threat **does not hold**. Multiple test cases with different validator configurations (including extreme imbalances such as *A = 9999, B = 1*) confirmed that validators with nonzero voting power are always eventually selected. Integer division and rounding did not introduce starvation or prevent fair proposer rotation.

  In the extreme imbalance scenario, validator *B* (1 voting power) was still selected, albeit very infrequently (~0.01% of the time). While long gaps between *B*'s proposals are possible, this is mathematically consistent with the weighted round-robin design and not a consequence of rounding errors.

  The proposer selection algorithm maintains its guarantee: *all validators with nonzero voting power are eventually selected*. There is **no permanent exclusion** due to integer division or rounding. The only observable effect is that validators with minimal voting power may propose very rarely, which is expected and correct under the algorithm's design.

- Threat: A validator that becomes a proposer after a long period leads to unfair proposer selection.

  Based on the available test cases and additional experiments, this threat **does not hold** in practice. The CometBFT proposer selection mechanism ensures that even validators re-enabled after prolonged inactivity do not destabilize fairness guarantees. The shifting and rescaling mechanisms effectively prevent disabled validators from indefinitely accumulating unbounded priority.

  In a controlled scenario with four validators of different weights (*A*: 50%, *B*: 20%, *C*: 10%, *D*: 20%), validator *A* was initially disabled and later re-enabled. Upon re-enablement, *A* was selected in **3 consecutive rounds** and **4 out of the first 6 rounds**. While this appears skewed in the short term, the system quickly stabilized, and thereafter *A* was chosen proportionally to its voting power (~50% of the time), consistent with the guarantees defined in the CometBFT specification↗.

  This behavior is not considered *unfair proposer selection*, but rather an expected outcome of weighted round-robin scheduling. Validators with higher voting power may temporarily dominate proposer slots immediately after re-enablement, but long-term fairness is preserved.

In conclusion, property 5 **holds**. The proposer selection algorithm in CometBFT ensures fairness through a weighted round-robin mechanism, guaranteeing that every validator with nonzero voting power is eventually selected. Integer division and rounding do not cause starvation or permanent exclusion, only expected proportional differences in frequency. Validators rejoining after downtime may temporarily propose more often, but the system quickly normalizes, preserving long-term fairness. Overall, proposer selection remains fair and proportional to voting power.

## Implementation Threats

- Threat: Non-determinism or randomization in the application (Cosmos SDK module)

  This threat **does not hold**. The Cosmos SDK is explicitly designed to guarantee determinism across all validators, as consensus safety requires every node to produce the same state root for the same block. State transitions are implemented using deterministic integer arithmetic, and the framework avoids floats, time-dependent logic, or random number generation inside consensus-critical code.

  In the audited module, no sources of non-determinism, such as randomness, system time, or unordered map iteration, were identified. Proposer set updates and flag handling (`proposeDisabled`) are consistently derived from the on-chain state, ensuring all validators compute identical results.

- Threat: Overflows or rounding problems

  This threat **does not hold.** Both the Cosmos SDK and CometBFT include multiple safeguards against arithmetic overflows and uncontrolled rounding errors, and no scenarios were identified where these could be exploited to break proposer fairness, validator set updates, or consensus safety.

  The Cosmos SDK uses arbitrary-precision types (`math.Int`, `math.LegacyDec`) for token amounts and stake accounting, which eliminates integer overflow risks in core staking and token logic.

  Consensus power is derived deterministically as `tokens / powerReduction`. This operation uses arbitrary-precision integers and only introduces safe truncation toward zero (deterministic across all nodes).

  The proposer set logic in the Cosmos SDK does not manipulate token balances directly; it only evaluates validator status and membership, so no arithmetic overflow risk exists there.

  On the CometBFT side, the proposer selection relies on `ProposerPriority` and `VotingPower`, both stored as `int64`. To prevent overflow:

  - `MaxTotalVotingPower = MaxInt64 / 8` enforces a strict upper bound.
  - `updateTotalVotingPower()` panics if this bound is exceeded, halting the chain rather than allowing undefined behavior.

  Priority rescaling (`RescalePriorities`) uses integer division. Truncation may cause differences of *±1* in priorities, but this is deterministic and does not bias proposer selection.

  Arithmetic updates use clipped add/subtract helpers (`safeAddClip`, `safeSubClip`), which prevent overflows and underflows.

  Any invalid divisor (zero or negative) in rescaling is explicitly guarded against, with early returns to avoid panics.

  Rounding occurs only in integer division (e.g., when rescaling proposer priorities or converting tokens to consensus power). This truncation is deterministic and uniformly applied across all nodes, so it cannot introduce consensus divergence or unfair proposer rotation.

  In conclusion, precision loss is limited to small, deterministic truncations in integer division, which are expected and harmless. Overflow is systematically prevented through arbitrary-precision math in the Cosmos SDK and explicit safeguards in CometBFT's proposer priority mechanism. As a result, the system ensures deterministic, overflow-safe arithmetic across all consensus-critical paths.

- Threat: Panics in `BeginBlock` or `EndBlock` (unless the chain is supposed to halt)

  Panics within `BeginBlock` and `EndBlock` are restricted to invariant checks that serve as safeguards against state corruption. Under normal and correctly functioning state transitions, these panics are unreachable. Therefore, no unintended panics occur during block execution, and the only circumstances under which a panic would arise are those where the chain is expected to halt due to a critical, unrecoverable state inconsistency.

  Some notable explicit panics:

  - In the `bondedToUnbonding()` method, inside the `x/staking/keeper/val_state_change.go`↗ file, the panic could occur if the method is called on a validator that is not bonded. In the current implementation, this scenario cannot occur. The `noLongerBonded` list is built from the `last` map, which contains validators that were in the previous block's validator set (i.e., they were bonded in the last block) but are not in the current set after the main loop. During the main loop, all currently bonded validators are removed from `last` as they are processed. Only those who were bonded in the previous block and are not bonded now remain in `last` and are processed here.
  - Similarly, the `unbondingToBonded()` method would panic if it is called on a validator that is not unbonding. However, in the current implementation, this method is only called within the validator update loop when a validator's status is explicitly checked to be Unbonding before the transition. This ensures that only validators in the correct state undergo this transition, making the panic unreachable under normal, correct operation.

  - Likewise, the `unbondedToBonded()` method would panic if invoked on a validator that is not Unbonded. This method is only called after confirming the validator's status is Unbonded, so the panic serves as a safeguard against unexpected state corruption or logic errors, but should never be triggered during normal execution.
  - The `BeginUnbondingValidator()` method would panic if called on a validator that is not Bonded. In the current implementation, this method is only reached after a status check ensures the validator is Bonded, so the panic acts as a defensive invariant check. If triggered, it would indicate a serious bug or data inconsistency, but such a scenario should not occur with correct state transitions.
  - In the main iterator loop inside the `ApplyAndReturnValidatorSetUpdates()` method, if the validator that is retrieved from the power store is jailed, it would result in a panic. This is the desired behavior, but it is still not a scenario that could occur since all the validators that are jailed are being automatically removed from the power store within the same transaction.

- Threat: Some inputs can lead to unbounded computation

  This threat **partially holds.** While most critical computation paths in the Cosmos SDK (such as validator updates, proposer selection, and consensus power calculations) are strictly bounded by protocol parameters (`MaxValidators`, `MaxTotalVotingPower`), the handling of proposer sets reveals a potential inefficiency.

  Specifically, the `checkProposerSetInvariants()` method and the marshalling/unmarshalling of proposer sets (`GetProposers()`, `SetProposers()`) iterate through the proposer list without filtering duplicates. This means that if a proposer set containing a very large number of duplicate entries is submitted via `MsgSetProposers`, the application will perform redundant checks and memory allocations proportional to the size of the duplicated array.

  Since the proposer set size is not directly capped by `MaxValidators`, but only by transaction size (and ultimately block size), this could allow an attacker to construct excessively large proposer sets that consume significant CPU and memory resources during processing. This presents a potential denial-of-service (DoS) vector.

  In practice, additional test cases have confirmed that duplicates are accepted, stored, and iterated over without deduplication. While the current impact does not compromise consensus safety or correctness, it does create unnecessary overhead and opens the door for DoS attacks under adversarial conditions.

  In conclusion, most computation paths remain safely bounded, but proposer set handling lacks a uniqueness check. This exposes the system to possible DoS scenarios if extremely large duplicate-heavy proposer sets are submitted. We recommend enforcing proposer uniqueness in `SetProposers()` to fully mitigate this issue.

- Threat: Duplicate validators can be added to the proposer set via `MsgSetProposers`:

  This threat **does hold.** The current implementation allows duplicate validator operator addresses to be included in the proposer set submitted via `MsgSetProposers`. These duplicates are persisted in the `KVStore` as part of the JSON-encoded proposer array, since there is no mechanism in place to enforce uniqueness at the time of storage.

  Importantly, while this does not create immediate consensus safety or fairness issues, it introduces unnecessary state bloat and weakens the robustness of the module against future changes. In particular:

  - **Consensus safety**: Duplicates do not increase a validator's voting power, as voting power is determined solely by bonded stake.
  - **Fairness**: Proposer selection remains unaffected, since the proposer updates are derived from the validator power store (`ValidatorsPowerStoreIterator`), and each validator is updated once regardless of how many times it appears in the proposer set.
  - **CometBFT updates**: In `EndBlocker()`, validator updates are consolidated before being sent to CometBFT, which prevents duplicate proposer entries from producing duplicate ABCI updates.

  The real risk lies in validation logic and future extensibility:

  - The `checkProposerSetInvariants()` method currently validates that all proposers have valid operator ad-

dresses, exist as validators, and meet the minimum number of bonded proposers (`MinBondedInProposerSet` = 5). However, it does not enforce uniqueness. This means a proposer set made entirely of duplicates could incorrectly satisfy the minimum bonded threshold (e.g., five copies of the same validator).
- Also, this could potentially lead to some unbounded computation, which was already explained within the previous threat inspection.
- While harmless today, this gap could become problematic if future functionality builds on proposer set semantics.

Additional tests were added to explicitly verify this behavior. These tests confirmed that proposer sets containing duplicate entries are currently accepted by the module and stored in state, further validating the existence of this issue.

We have reported a recommendation to explicitly check uniqueness in finding "Recommendation to enforce uniqueness in proposer set validation".

# Findings

| Finding | Type | Severity | Status |
|---|---|---|---|
| Redelegations, slashing and jailing can lead to scenarios where there is no correct proposer | Design | High | Acknowledged |
| Recommendation to enforce uniqueness in proposer set validation | Implementation | Informational | Acknowledged |
| Miscellaneous code improvements | Implementation | Informational | Acknowledged |

## Redelegations, slashing and jailing can lead to scenarios where there is no correct proposer

**Severity** High **Impact** 3 - High **Exploitability** 2 - Medium

**Type** Design **Status** Acknowledged

### Description

Without further assumptions, because of redelegations, slashing or jailing, the number of proposers may fall below a value where according to their stake and the normal fault assumptions of CometBFT all of them may fail.

According to the discussions with the development team during the audit, it is assumed that there is at least one active proposer in the validator set at any time, therefore acknowledging that there is some additional assumptions on the correctness of validators.

However, this assumption is in general strictly stronger than the standard safety assumption for CometBFT consensus, where a set of validators that comprises up to one third of the voting power might be faulty. To ensure that there is always a non-faulty proposer in the active set of validators, a sufficient condition would be that the set of proposers has always a total voting power of more than a third of the total voting power of all validators.

It also needs to be mentioned that the acceptance of a `MsgSetProposer` by a governance proposal is a rare and long-term event, while redelegations and slashing might happen faster and are not in the hands of the developers and operators of the chain.

### Problem scenarios

An example scenario would be that 5 validators are selected by `MsgSetProposers`. Due to redelegations and/or slashing and jailing only 1 validator `p` remains in the active set of validators and thus the only proposer. During the execution of the consensus protocol for some height *H*, `p` fails during sending the proposal in round 1 and the consensus algorithm needs to start a second round with a new proposer. However, `p` is the only proposer in the set of potential proposers. In this case, the consensus algorithm does not terminate, and, since no new blocks are produced no new set of proposers can be elected, and no redelegations to bring in new proposers are possible. Note that no node halts with a panic in this case.

### Recommendation

Even if the design decision is to accept a stronger assumption on the correctness of validators, they should be properly formalized and safeguards should be installed in order to detect violations of this assumption and either prevent these situations or fail in a coordinated way. Depending on the precise semantics of such an assumption, the algorithm could detect/prevent scenarios where the set of proposers fall or their associated stake below some threshold due to redelegations/slashing/jailing. The implementation could then, for example, automatically enable all proposers again or prevent the redelegation or halt the chain by a well-defined panic.

## Recommendation to enforce uniqueness in proposer set validation

**Severity** Informational **Impact** 1 - Low Exploitability:

**Type** Implementation **Status** Acknowledged

### Involved artifacts

- `x/staking/keeper/msg_server.go`↗ → `SetProposers()`
- `x/staking/keeper/proposer_set.go`↗ → `checkProposerSetInvariants()`, `SetProposers()`
- `x/staking/types/keys.go`↗ → `ProposerSetKey` (where the proposer set is persisted)

### Description

The current implementation of proposer set validation (`checkProposerSetInvariants()`) ensures that all proposers correspond to valid operator addresses, exist as validators, and that the set meets the minimum bonded proposer threshold (`MinBondedInProposerSet = 5`). However, it does not enforce uniqueness of proposer entries. As a result, a proposer set containing duplicate validators can pass invariant checks and be stored in the state.

### Problem scenarios

A proposer set could consist entirely of duplicate validators (e.g., five copies of the same validator). This incorrectly satisfies the `MinBondedInProposerSet` requirement, potentially weakening governance assumptions.

Loops iterating over proposer sets (e.g., in `SetProposers()` and `checkProposerSetInvariants()`) could become computationally expensive if duplicates are present in large numbers, raising potential denial-of-service (DoS) concerns.

While duplicates do not currently affect proposer selection or validator power in CometBFT, future protocol changes that rely more heavily on proposer set semantics may be impacted.

Some example test cases showcasing this scenario can be found in the Appendix B.

### Recommendation

Introduce explicit checks to ensure the uniqueness of proposers at the time of setting the proposer set (e.g., inside `SetProposers()` or `checkProposerSetInvariants()`).

For example, maintain a temporary map keyed by proposer operator addresses while iterating over the input proposer set. Each time a proposer is encountered, check if it already exists in the map. If not, add it to the map; if yes, skip it or return an error, depending on the desired approach.

This safeguard ensures proposer sets are semantically correct, mitigates potential performance risks, and strengthens resilience against future protocol extensions.

## Miscellaneous code improvements

**Severity** Informational Impact: Exploitability:

**Type** Implementation **Status** Acknowledged

- In function `ValidateGenesis()`↗ of `x/staking` module. Validation could be added for the `Proposers` field of `GenesisState`.↗

# Appendix A: Additional Tests for Proposer Set Validation

To strengthen confidence in proposer set handling and explicitly test against duplicate entries, we recommend introducing the following test cases to the `x/staking/keeper/proposer_set_test.go`↗ file.

The `TestRejectDuplicateProposers` ensures that a proposer set containing only duplicates of the same validator does not pass validation. Its purpose is to confirm that duplicates are explicitly handled and do not silently persist in the state.

```
func (s *KeeperTestSuite) TestRejectDuplicateProposers() {
    ctx, keeper := s.ctx, s.stakingKeeper
    require := s.Require()

    // Create one bonded validator
    valPubKey := PKs[0]
    valAddr := sdk.ValAddress(valPubKey.Address().Bytes())
    validator := testutil.NewValidator(s.T(), valAddr, valPubKey)
    validator.Status = types.Bonded
    err := keeper.SetValidator(ctx, validator)
    require.NoError(err)

    // Create a proposer set with duplicates of the single validator
    dupProposers := []string{validator.OperatorAddress,
↪   validator.OperatorAddress, validator.OperatorAddress,
↪   validator.OperatorAddress, validator.OperatorAddress}

    // Try setting with duplicates
    err = keeper.SetProposers(ctx, dupProposers)

    // The system should either reject duplicates or deduplicate them
    proposers, getErr := keeper.GetProposers(ctx)
    require.NoError(getErr)

    unique := make(map[string]struct{})
    for _, p := range proposers {
        unique[p] = struct{}{}
    }

    if err != nil {
        // Should return an error for duplicates
        require.Error(err, "expected error when setting duplicate proposers")
    } else {
        // If no error, the stored proposer set should not contain duplicates
        require.Equal(len(unique), len(proposers), "proposer set should not
↪   contain duplicates")
        require.Equal(1, len(unique), "all proposers should be unique, only one
↪   unique proposer expected")
```

```
    }
}
```

The `TestRejectMixedDuplicateProposers` validates that proposer sets with a mix of unique and duplicate validators are also correctly handled. Its purpose is to ensure proposer sets remain semantically correct even when partial duplication is introduced.

```
func (s *KeeperTestSuite) TestRejectMixedDuplicateProposers() {
    ctx, keeper := s.ctx, s.stakingKeeper
    require := s.Require()

    // Create 3 unique bonded validators
    validators := make([]string, 3)
    for i := 0; i < 3; i++ {
        valPubKey := PKs[i]
        valAddr := sdk.ValAddress(valPubKey.Address().Bytes())
        validator := testutil.NewValidator(s.T(), valAddr, valPubKey)
        validator.Status = types.Bonded
        validators[i] = validator.OperatorAddress
        err := keeper.SetValidator(ctx, validator)
        require.NoError(err)
    }

    // Proposer set: 3 unique, 2 duplicates (validators[0] and validators[1]
     ↪  duplicated)
    dupProposers := []string{validators[0], validators[1], validators[2],
↪   validators[0], validators[1]}

    err := keeper.SetProposers(ctx, dupProposers)

    proposers, getErr := keeper.GetProposers(ctx)
    require.NoError(getErr)

    unique := make(map[string]struct{})
    for _, p := range proposers {
        unique[p] = struct{}{}
    }

    if err != nil {
        // Should return an error for duplicates
        require.Error(err, "expected error when setting duplicate proposers")
    } else {
        // If no error, the stored proposer set should not contain duplicates
        require.Equal(len(unique), len(proposers), "proposer set should not
↪   contain duplicates")
        // Should only have 3 unique proposers
        require.Equal(3, len(unique), "all proposers should be unique, only three
↪   unique proposers expected")
    }
}
```

# Appendix B: Vulnerability classification

For classifying vulnerabilities identified in the findings of this report, we employ the simplified version of Common Vulnerability Scoring System (CVSS) v3.1↗, which is an industry standard vulnerability metric. For each identified vulnerability we assess the scores from the *Base Metric Group*, the Impact score↗, and the Exploitability score↗. The *Exploitability score* reflects the ease and technical means by which the vulnerability can be exploited. That is, it represents characteristics of the *thing that is vulnerable*, which we refer to formally as the *vulnerable component*. The *Impact score* reflects the direct consequence of a successful exploit, and represents the consequence to the *thing that suffers the impact*, which we refer to formally as the *impacted component*. In order to ease score understanding, we employ CVSS Qualitative Severity Rating Scale↗, and abstract numerical scores into the textual representation; we construct the final *Severity score* based on the combination of the Impact and Exploitability sub-scores.

As blockchains are a fast evolving field, we evaluate the scores not only for the present state of the system, but also for the state that deems achievable within 1 year of projected system evolution. E.g., if at present the system interacts with 1-2 other blockchains, but plans to expand interaction to 10-20 within the next year, we evaluate the impact, exploitability, and severity scores wrt. the latter state, in order to give the system designers better understanding of the vulnerabilities that need to be addressed in the near future.

## Impact Score

The Impact score captures the effects of a successfully exploited vulnerability on the component that suffers the worst outcome that is most directly and predictably associated with the attack.

| ImpactScore | Examples |
|---|---|
| High | Halting of the chain; loss, locking, or unauthorized withdrawal of funds of many users; arbitrary transaction execution; forging of user messages / circumvention of authorization logic |
| Medium | Temporary denial of service / substantial unexpected delays in processing user requests (e.g. many hours/days); loss, locking, or unauthorized withdrawal of funds of a single user / few users; failures during transaction execution (e.g. out of gas errors); substantial increase in node computational requirements (e.g. 10x) |
| Low | Transient unexpected delays in processing user requests (e.g. minutes/a few hours); Medium increase in node computational requirements (e.g. 2x); any kind of problem that affects end users, but can be repaired by manual intervention (e.g. a special transaction) |

| ImpactScore | Examples |
|---|---|
| None | Small increase in node computational requirements (e.g. 20%); code inefficiencies; bad code practices; lack/incompleteness of tests; lack/incompleteness of documentation |

## Exploitability Score

The Exploitability score reflects the ease and technical means by which the vulnerability can be exploited; it represents the characteristics of the vulnerable component. In the below table we list, for each category, examples of actions by actors that are enough to trigger the exploit. In the examples below:

- *Actors* can be any entity that interacts with the system: other blockchains, system users, validators, relayers, but also uncontrollable phenomena (e.g. network delays or partitions).
- *Actions* can be
  - *legitimate*, e.g. submission of a transaction that follows protocol rules by a user; delegation/redelegation/bonding/unbonding; validator downtime; validator voting on a single, but alternative block; delays in relaying certain messages, or speeding up relaying other messages;
  - *illegitimate*, e.g. submission of a specially crafted transaction (not following the protocol, or e.g. with large/incorrect values); voting on two different alternative blocks; alteration of relayed messages.
- We employ also a *qualitative measure* representing the amount of certain class of power (e.g. possessed tokens, validator power, relayed messages): *small* for < 3%; *medium* for 3-10%; *large* for 10-33%, *all* for >33%. We further quantify this qualitative measure as relative to the largest of the system components. (e.g. when two blockchains are interacting, one with a large capitalization, and another with a small capitalization, we employ *small* wrt. the number of tokens held, if it is small wrt. the large blockchain, even if it is large wrt. the small blockchain)

| ExploitabilityScore | Examples |
|---|---|
| High | illegitimate actions taken by a small group of actors; possibly coordinated with legitimate actions taken by a medium group of actors |
| Medium | illegitimate actions taken by a medium group of actors; possibly coordinated with legitimate actions taken by a large group of actors |
| Low | illegitimate actions taken by a large group of actors; possibly coordinated with legitimate actions taken by all actors |
| None | illegitimate actions taken in a coordinated fashion by all actors |

## Severity Score

The severity score combines the above two sub-scores into a single value, and roughly represents the probability of the system suffering a severe impact with time; thus it also represents the measure of the urgency or order in which vulnerabilities need to be addressed. We assess the severity according to the combination scheme represented graphically below.

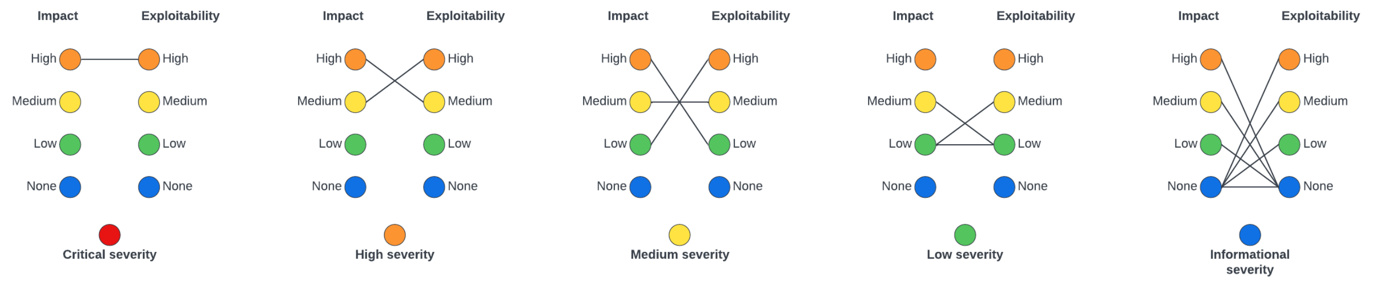

Figure 9: Severity classification

As can be seen from the image above, only a combination of high impact with high exploitability results in a Critical severity score; such vulnerabilities need to be addressed ASAP. Accordingly, High severity score receive vulnerabilities with the combination of high impact and medium exploitability, or medium impact, but high exploitability.

| SeverityScore | Examples |
|---|---|
| Critical | Halting of chain via a submission of a specially crafted transaction |
| High | Permanent loss of user funds via a combination of submitting a specially crafted transaction with delaying of certain messages by a large portion of relayers |
| Medium | Substantial unexpected delays in processing user requests via a combination of delaying of certain messages by a large group of relayers with coordinated withdrawal of funds by a large group of users |
| Low | 2x increase in node computational requirements via coordinated withdrawal of all user tokens |
| Informational | Code inefficiencies; bad code practices; lack/incompleteness of tests; lack/incompleteness of documentation; any exploit for which a coordinated illegitimate action of all actors is necessary |

# Disclaimer