The goal of this proposal is to cap the active validator set to some fixed value (eg. 2^{19}

validators) while at the same time ensuring three key properties:

1. An economic finality guarantee where close to 1/3 of the cap must be slashed to finalize two conflicting blocks
2. Low long-term variance in validator income
3. Maximum simplicity.

A validator size cap is desirable because it increases confidence that a given level of hardware will always

be sufficient to validate the beacon chain. The proposed 2^{19}

cap reduces the theoretical max number of validators from a near-term ~3.8M to a permanent ~524k, a factor of seven. This extra guarantee can be used to either guarantee a lower level of hardware requirements, increasing accessibility of staking, or reduce the deposit size, increasing the computational complexity of validation again but decreasing the possibly much larger factor of the minimum amount of ETH needed to participate.

The key properties are achieved by bringing back a modified version of the “dynasty” mechanism from theCasper FFG paper. In short, if there are more than MAX_VALIDATOR_COUNT

active validators, only MAX_VALIDATOR_COUNT

of them will be “awake” at any time, and every time the chain finalizes the portion of validators that are awake gets rotated so that it changes by ~1/64. This ensures that the level of finalization safety is almost the same (see the Casper FFG paper for reasoning), while at the same time ensures that rotation is fairly quick, ensuring low variance (meaning, there won’t be validators who get very unfairly low returns due to the bad luck of never being awake).

Constants

Constant

Value

Notes

MAX_VALIDATOR_COUNT

2**19 (= 524,288)

~16.7M ETH

ROTATION_RATE

2**6 (= 64)

Up to 1.56% per epoch

FINALIZED_EPOCH_VECTOR_LENGTH

2**16 (= 65,536)

32 eeks ~= 291 days

FINALITY_LOOKBACK

2**3 (= 8)

Measured in epochs

The state.is_epoch_finalized

bitarray

We add a new BeaconState

member value:

is_epoch_finalized: BitList[FINALIZED_EPOCH_VECTOR_LENGTH]

The first value stores which epochs have been finalized; the second stores a counter of how many epochs were finalized in the time that is too far back in history for the array to store.

In the weigh_justification_and_finalization

function, when state.finalized_checkpoint

is updated to new_checkpoint

, we update:

current_epoch_position = (get_current_epoch(state) % FINALIZED_EPOCH_VECTOR_LENGTH)
state.is_epoch_finalized[current_epoch_position] = False

In all cases where we do `state.finalized_checkpoint = new_checkpoint`

new_finalized_epoch_position = (new_checkpoint.epoch % FINALIZED_EPOCH_VECTOR_LENGTH)
state.is_epoch_finalized[new_finalized_epoch_position] = True

We can also refine the “was this epoch finalized?” helper:

def did_epoch_finalize(state: BeaconState, epoch: Epoch) -> bool: assert epoch < get_current_epoch(state) assert epoch + FINALIZED_EPOCH_VECTOR_LENGTH > get_current_epoch(state) return state.is_epoch_finalized[epoch % FINALIZED_EPOCH_VECTOR_LENGTH]

Definition of get_awake_validator_indices

and helpers

get_awake_validator_indices

returns a subset of get_active_validator_indices

. The function and helpers required for it are defined as follows.

This next function outputs a set of validators that get slept in a given epoch (the output is nonempty only if the epoch has been finalized). Note that we use the finality bit of epoch N and the active validator set of epoch N+8; this ensures that by the time the active validator set that will be taken offline is known there is no way to affect finality.

def get_slept_validators(state: BeaconState, epoch: Epoch) -> Set[ValidatorIndex]: assert get_current_epoch(state) >= epoch + MAX_SEED_LOOKAHEAD * 2 active_validators = get_active_validator_indices(state, epoch + FINALITY_LOOKBACK) if len(active_validators) >= MAX_VALIDATOR_COUNT: excess_validators = len(active_validators) - MAX_VALIDATOR_COUNT else: excess_validators = 0 if did_epoch_finalize(state, epoch): seed = get_seed(state, epoch, DOMAIN_BEACON_ATTESTER) validator_count = len(active_validators) return set(active_validators[compute_shuffled_index(i, validator_count, seed)] for i in range(len(excess_validators // ROTATION_RATE))) else: return set()

This next function outputs the currently awake validators. The idea is that a validator is awake if they have not been slept in one of the last ROTATION_RATE

finalized epochs.

def get_awake_validator_indices(state: BeaconState, epoch: Epoch) -> Set[ValidatorIndex]: o = set()
finalized_epochs_counted = 0 search_start = FINALITY_LOOKBACK search_end = min(epoch + 1, FINALIZED_EPOCH_VECTOR_LENGTH) for step in range(search_start, search_end): check_epoch = epoch - step if did_epoch_finalize(check_epoch): o = o.union(get_slept_validators(state, finalized_epoch)) finalized_epochs_counted += 1 if finalized_epochs_counted == ROTATION_RATE: break
return [v for v in get_active_validator_indices(state, epoch) if v not in o]

The intention is that get_awake_validator_indices

contains at most roughly MAX_VALIDATOR_COUNT

validators (possibly slightly more at certain times, but it equilibrates toward that limit), and it changes by at most 1/ROTATION_RATE

per finalized epoch. The restriction to finalized epochs ensures that two conflicting finalized blocks can only differ by at most an extra 1/ROTATION_RATE

as a result of this mechanism (it can also be viewed as an implementation of thedynasty mechanism from the original Casper FFG paper).

Protocol changes

All existing references to get_active_validator_indices

are replaced with get_awake_validator_indices

. Specifically, only awake indices are shuffled and put into any committee or proposer selection algorithm. Rewards and non-slashing penalties for non-awake active validators should equal 0. Non-aware active validators should still be vulnerable to slashing.

Economic effects

- Once the active validator set size exceeds MAX_VALIDATOR_COUNT

, validator returns should start decreasing proportionately to 1/total_deposits

and not 1/sqrt(total_deposits)

. But the functions to compute total_deposits -> validator_return_rate

and total_deposits -> max_total_issuance

remain continuous.

- Validators active in epoch N can affect the finalization status of epoch N. At that time, the active validator set in epoch N+8 is unknown. Hence, validators have no action that they can take to manipulate the randomness to keep themselves active.
- Validators can

delay finality to keep themselves active. But they cannot increase their profits by doing this, as this would put them into an inactivity leak.

Alternatives

- Upon activation, a validator receives a pseudorandomly generated timezone

in 0...511

. If VALIDATOR_COUNT > MAX_VALIDATOR_COUNT

, the validators awake are those whose (timezone + get_number_of_times_finalized(state, epoch)) % 512 < (512 * MAX_VALIDATOR_COUNT) // VALIDATOR_COUNT

. This could simplify the logic for determining who is awake

- Make awake and sleep periods longer, and allow validators who have been asleep recently to withdraw more quickly after exiting (this may only be worth exploring after when withdrawals are available)