The current [sharding phase 1 doc](#) specifies running the proposer eligibility function getEligibleProposer

onchain. We suggest an alternative approach based on a fork choice rule, complemented with optional “partial validation” and slashing.

The benefit of the fork choice rule is that getEligibleProposer

is run offchain. This saves gas when calling addHeader

and unlocks the possibility for fancier proposer eligibility functions. At the end we detail two proposal mechanisms, one for variable-size deposits and one for private sampling.

Fork choice rule

Collation validity is currently done as a fork choice rule, and collation header validity is done onchain with addHeader

. We suggest extending the fork choice rule to collation headers as follows:

- addHeader

always returns True, and always records a corresponding CollationAdded

log

- getEligibleProposer

is run offchain and filtering of invalid collation headers is done as a fork choice rule

For the logic to fetch candidate heads (c.f. [fetching in reverse sorted order](#)) to work and the fork choice rule to be enforceable, the CollationAdded

logs need to be filterable for validity post facto. This relies on historical data availability of validator sets (and other auxiliary data for sampling, such as entropy).

We are already assuming that the historical CollationAdded

logs are available so it suffices to extend this assumption to validator sets. A clean solution is to have ValidatorEvent

logs for additions and removals, and an equivalent getNextLog

method for such logs.

Partial validation and slashing condition

To simplify the fork choice rule and lower the dependence on historical availability of validator sets, there are two hybrid approaches that work well:

1. Partial validation

: have addHeader

return True only if the signature sig

corresponds some

collaterised validator

1. Slashing condition

(building upon partial validation): if the validator that called addHeader

does not match getEligibleProposer

(run offchain) then a whitleblower can run getEligibleProposer

onchain to burn half the validator’s deposit and keep the other half

Variable-size deposits

Let v_1, ..., v_n

be the validators with deposits d_1, ..., d_n

. Fairly sampling validators when the d_i can have arbitrary size can be tricky because the amount of work to run getEligibleProposer

is likely bounded below by $\log(n)$

, which is not ideal. With the fork choice rule we can take any (reasonable) fair sampling function and run it offchain.

For concreteness let's build getEligibleProposer

as follows. Let E

be 32 bytes of public entropy (e.g. a blockhash, as in the phase 1 sharding doc). Let S_j

be the partial sums $d_1 + ... + d_j$

and let $\tilde{E} \in [0, S_n-1]$

where $\tilde{E} \equiv E \mod S_n$

. Then getEligibleProposer

selects the validator v_i

such that $S_{i-1} \le \tilde{E} \lt S_i$

.

Private sampling

We now look at the problem of private sampling. That is, can we find a proposal mechanism which selects a single

validator per period and provides "private lookahead", i.e. it does not reveal to others which validators will be selected next?

There are various possible private sampling strategies (based on MPCs, SNARKs/STARKs, cryptoeconomic signalling, or fancy crypto) but finding a workable scheme is hard. Below we present our best attempt based on one-time ring signatures. The scheme has several nice properties:

1. Perfect privacy

: private lookahead and

private lookbehind (i.e. the scheme never matches eligible proposers with specific validators)

1. Full lookahead

: the lookahead extends to the end of the epoch (epochs are defined below, and have roughly the same size as the validator set)

1. Perfect fairness

: within an epoch validators are selected proportionally according to deposit size, with zero variance

The setup assumes validators have deposits in fixed-size increments (e.g. multiples of 1000 ETH). Without loss of generality we have one validator per fixed-size deposit. The proposal mechanism is organised in variable-size epochs. From the beginning of each epoch to its end, every validator has the right to push a log to be elected in the next epoch. The log contains two things:

1. A once-per-epoch ring signature proving membership in the current validator set
2. An ephemeral identity

The logs are ordered chronologically in an array, and the size of the array at the end of one epoch corresponds to the size of the next epoch (measured in periods). To remove any time-based correlation across logs, we publicly shuffle the array using public entropy. This shuffled array then constitutes a sampling, each entry corresponding to one period. To call addHeader

, the validator selected for a given period must sign the header with the corresponding ephemeral identity.

With regards to publishing logs, log shards work well for several reasons:

1. They provide cheap logging facilities (data availability, ordering, witnesses)
2. Gas is paid out-of-bound so this limits opportunities to leak privacy through onchain gas payments