This assumes a setup as follows:

- There is a buffer of size BUFFER_SIZE
- Every slot, TAKE_SIZE

proposers are taken from the front of the buffer, and TAKE_SIZE

new proposers are added.

- Every slot, SHUFFLE_SIZE

randomly selected indices in the buffer are reshuffled

The script models the buffer as probability distributions, and computes basic properties of these distributions. For simplicity it focuses on the average probability distribution of the first

proposer in each slot, so a probability of 0.038 means that you can identity a validator that has a 3.8% chance of being the first proposer.

One result:

5000 rounds completed Tested shuffling 512 proposers in each slot and taking 65, with a buffer size of 2048 Average probability of most likely proposer: 0.038 (1 in 26.327) Average entropy: 4.344 nats (equiv to uniform distribution of 77.046 proposers)

The script:

import random, math

BUFFER_SIZE = 2048 SHUFFLE_SIZE = 512 TAKE_SIZE = 65 ROUNDS = 5000

Object representing probability distributions

class ProbabilityDistribution(): def **init**(self, probs): assert 0.9999999 < sum(probs.values()) < 1.0000001 self.probs = {k:v for k,v in probs.items()}

```
@classmethod
def average(cls, dists):
    out = {}
    L = len(dists)
    for p in dists:
        for k, v in p.probs.items():
            out[k] = out.get(k, 0) + v/L
    return cls(out)

def __str__(self):
    return str({k: int(v*10000)/10000 for k,v in self.probs.items()})
```

Randomly select K of N indices

def select_indices(buffer_size, selections): # If K > N/2 more efficient to select the complement if selections > buffer_size // 2: inverse_selections = select_indices(buffer_size, buffer_size - selections) return set(i for i in range(buffer_size) if i not in inverse_selections) o = set() while len(o) < selections: o.add(random.randrange(buffer_size)) return o

def simulate_proposer_selection(): # Initialize the buffer with known proposers proposer_buffer = [ProbabilityDistribution({i:1}) for i in range(BUFFER_SIZE)] # These variables help us later compute the average max and entropy max_accumulator = 0 entropy_accumulator = 0 # For each round..... for r in range(ROUNDS): # Pick indices to shuffle shuffle_indices = sorted(select_indices(BUFFER_SIZE, SHUFFLE_SIZE)) # Average over all possible shuffles avg = ProbabilityDistribution.average([proposer_buffer[index] for index in shuffle_indices]) for index in shuffle_indices: proposer_buffer[index] = avg # Probability of most likely proposer max_prob = max(list(proposer_buffer[0].probs.values())) # Entropy of the probability distribution entropy = sum([-x * math.log(x) for x in proposer_buffer[0].probs.values()]) max_accumulator += max_prob entropy_accumulator += entropy # Remove the desired number of proposers and replace them with new ones for _ in range(TAKE_SIZE): proposer_buffer.pop(0) proposer_buffer.append(ProbabilityDistribution({BUFFER_SIZE+r:1})) if r % 100 == 99: print("{} rounds completed".format(r+1)) # Print parameters and results print("Tested shuffling {} proposers in each slot and taking {}, with a buffer size of {}".format(SHUFFLE_SIZE, TAKE_SIZE, BUFFER_SIZE)) avg_max = max_accumulator / ROUNDS print("Average probability of most likely proposer: {:.3f} (1 in {:.3f})".format(avg_max, 1/avg_max)) avg_entropy = entropy_accumulator / ROUNDS print("Average entropy: {:.3f} nats (equiv to uniform distribution of {:.3f} proposers)".format(avg_entropy, math.exp(avg_entropy))) if **name** == '**main**': simulate_proposer_selection()