Sortition Mining

@shotaronowhere

A sortition mechanism with off-chain mining.

Suppose R is a source of randomness. Fix this value during a drawing phase when jurors can no longer change their stakes.

For each dispute with

- disputeID d
- voteID v ∈ {0, 1, 2, … , minJurors}

Derive a unique random number $R_{d,v} = keccak256(abi.encode(R,\ d,\ v))$

For each juror, with juror address j, define a sortition "lottery" number, as a function of a user specified nonce, as

$$sortition\, number(j,\ nonce)\ =\ keccak256(abi.encode(j,\ R\ ,\ nonce))$$

The drawn juror, or "winner of the lottery", is

$$nonce_{drawn},\, j_{drawn}\ =\ argmin_{j \in stakedJurors,\, nonce\, \in Strategy}\ |R_{d,v} - sortition\, number\ (j,\ nonce)|$$

Where the nonce is subjected to a sortition strategy constraint. For, example:

Linear

$$Strategy\ =\ \{Z\,|\,nonce\ <\ stake\ (jurorAddress)\}\ =\ \{0,\ 1,\ 2\ldots\ stake(juror)\ -\ 1\}$$

SybilResistant

$$Strategy\ =\{Z\,|\,nonce\ <\ isPoh(jurorAddress)\ *\ stake\ (jurorAddress)\}$$

Quadratic

$$Strategy\ =\ \{Z\,|\,nonce\ <\ isPoh(jurorAddress)\ *\ sqrt(stake\ (jurorAddress))\}$$

Conviction

$$Strategy\ =\ \{Z\,|\,nonce\ <\ (timeStaked\ -\ block.timestamp)\ *\ (stake\ (jurorAddress))\}$$

In other words, the juror with the closest 'lottery ticket' to the target $R_{d,v}$ wins, and because the hash function is collision resistant, the 'winner' is unique. The strategies are also very flexible and simple to implement.

The resolution of the sortition probability can be adjusted. In the above strategies, the nonce is restricted to integers. That is, the minimum stake is 1 PNK for jurors to obtain 1 sortition 'lottery' number. This reduces the search space of the optimization problem to in the worst case 1 billion unique nonce (max supply pnk), jurorAddress pairs. This reduces the sortition to an **off-chain mining problem** to solve the constrained optimization problem. I suggest a minimum safe value for the min stake per sortition number to be **at least 1 μ pnk** (6 decimals), this makes the off-chain computation tractable in terms of calculating the number of hashes I would suggest from an economic perspective, **10 milli PNK** is sufficient resolution. Then, in the worst case,

Sortition Mining

@shotaronowhere

100 billion off-chain hashes are required to find the sortition numbers. With benchmarks evaluated with hashcat, a consumer grade laptop (Macbook Air M1), the keccak256 hashrate is 90 Mh/s. A top-end consumer grade gpu has a keccak256 hashrate of 2.2 Gh/s. So a macbook can solve this problem in ~10 seconds, a gpu in ~0.5 seconds.

Simplification

No derived random number is necessary. Instead, simply use the same source of randomness for each dispute d and each vote v.

Calculate the objective function below for each sortition number and sort the list O(nlogn). Take the first m jurors in the ordered list as the 'winners'. Allocate the winners to votes in disputes sequentially in the dispute id and vote id.

$$sortition\ number(j,\ nonce)\ |_{j \in stakedJurors, nonce\ \in Strategy} = keccak256(abi.encode(j,\ R\ ,\ nonce))$$

$$Strategy\ = \{Z\,|\,nonce\ <\ stake\,(jurorAddress)\}\ =\ \{0,\ 1,\ 2\ldots\ stake(juror)\ -\ 1\}$$

Worst Case Scenario

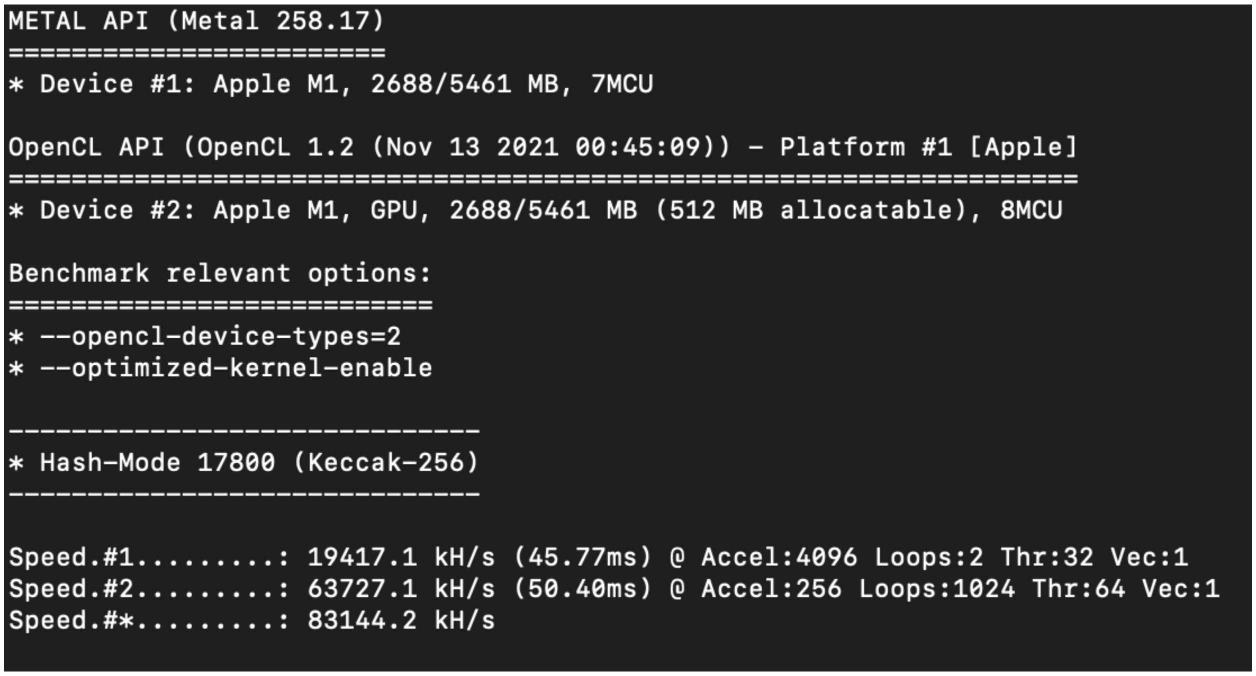
100% of PNK staked in the general court -> **1 Billion PNK**

Limit **0.01 PNK** per sortition number / 'lottery ticket', **100 Billion hashes to calculate all sortition numbers.**

With a single Apple M1 Macbook, the keccak hashrate is ~**83 MH/s -> ~20 min**

Sortition Mining

@shotaronowhere



```
METAL API (Metal 258.17)
========================
* Device #1: Apple M1, 2688/5461 MB, 7MCU

OpenCL API (OpenCL 1.2 (Nov 13 2021 00:45:09)) - Platform #1 [Apple]
====================================================================
* Device #2: Apple M1, GPU, 2688/5461 MB (512 MB allocatable), 8MCU

Benchmark relevant options:
===========================
* --opencl-device-types=2
* --optimized-kernel-enable

------------------------------
* Hash-Mode 17800 (Keccak-256)
------------------------------

Speed.#1.........: 19417.1 kH/s (45.77ms) @ Accel:4096 Loops:2 Thr:32 Vec:1
Speed.#2.........: 63727.1 kH/s (50.40ms) @ Accel:256 Loops:1024 Thr:64 Vec:1
Speed.#*.........: 83144.2 kH/s
```

In other words, only 1 honest actor (**proposer**) needs to perform the off-chain (O(n) where n is the number of sortition numbers) computation once, and bring the result on-chain by proposing the solution to the argmin constrained optimization problem. In a sample worst case scenario where 100 billion sortition numbers are calculated, this computation requires a single consumer grade laptop and a computation time of 20 minutes. The average individual juror with a consumer grade laptop can verify and compare their own sortition number with the proposed optimization solution. If a juror finds their sortition number results in a better argmin metric, then they can dispute the proposed set of jurors by passing their nonce on chain, and the computational dispute between **sortition proposer** and the **sortition verifier** is resolved in O(1) time on-chain.

Arbiturm Special Consideration

Currently on arbitrum, a malicious sequencer can censor transactions. However, users can force delayed transactions in a censorship resistant manner after a delay period. Currently on Arbitrum One, and the Nitro testnet, the delay period is 1 day. In the case of the sortition tree, no liveliness is required. If the bots fail to draw jurors due to censorship, simply no jurors are drawn. A malicious sequencer could censor proposals and challenges to the proposed sortition mining solution. However, as long as the evidence period + draw period > 1 day + off-chain computation time, jurors can always force their transactions through. the evidence period + draw period should be at least 1 day enough time for off-chain computation. The evendence period is already often in many subcourts set to a period of at least on the order of days which satisfies the criterion.