TL;DR:

A polynomial commitment scheme that stores key-value pairs in a hashmap, rather than using multiple polynomials. This allows a simpler construction that does not rely on complex map-merging arguments, and does not need any caching.

Background

Ethereum 2.0 is going to implement a “stateless consensus” architecture, which means that all blocks can be fully validated without access to state. The same is currently attempted for the legacy Ethereum 1.0 chain, so that it can be included in Eth2 as a shard. The motivation is that this will allow validators to not keep any shard state, lowering validator hardware requirements and making it more accessible.

One of the difficulties with this is that the witness sizes required for this can be very substantial. We can reduce this by using polynomial commitments, in which any number of data elements can be proven using just a single group element as a witness. One such scheme has been described by @vbuterin [1], relying on sorted key-value lists. However, it introduces significant complexity in the form of several layers of caching and needing permutation arguments to merge those separate commitments.

Kate commitments

The Kate commitment scheme [2] is a way to commit to a polynomial $f(x)=\sum_{i=0}^n f_i x^i$

and decommitting a value $y=f(r)$

(i.e. giving y

and a witness that proves that it is the evaluation of the previously committed polynomial at r

). The witness is a single group element.

Let $e: G_1 \times G_2 \rightarrow G_t$

be a pairing of two elliptic curves G_1

and G_2

with generators $G \in G_1$

and $H \in G_2$

(I use additive notation for group operations). We need a trusted setup that provides $s^i G$

and $s^i H$

for $i=0,\ldots, n-1$

where n

is the degree of the polynomial to be committed to and s

is a random value not known to anyone. Then a commitment to a polynomial $f(x)=\sum_{i=0}^{n-1} f_i x^i$

consists of

$$\displaystyle C = \sum_{i=0}^{n-1} f_i s^i G \text{ .}$$

To prove an evaluate this polynomial at $x=r$

, compute the polynomial

$$\displaystyle h(x) = \frac{f(x) - f(r)}{x-r} = \sum_{i=0}^{n-2} h_i x^i \text{ .}$$

The witness for the evaluation $f(r) = y$

then consists of

$$\displaystyle \Pi = \sum_{i=0}^n h_i s^i G$$

which can be checked using the pairing equation

\displaystyle e(\Pi, (s - r) H) = e(C - y G, H) \text{ .}

Kate multi-reveal

As a matter of notation, we will denote by f//g

the whole part of the polynomial division of f

by g

and by f \% g

the remainder. So

\displaystyle f(x) = f // g g(x) + f \% g

Let’s say we want to prove evaluations of the polynomial f

at y_0 = f(x_0), \ldots , y_{m-1} = f(x_{m-1})

. Let S = {x_0, ..., x_{m-1}}

and Z_S(x) = \prod_{z\in S}(x -z)

. To prove that the polynomial was correctly evaluated, the prover provides the group element \pi = f // Z_S G

, which can be computed from the SRS. The verifier will check that the values are correct by computing the value f \% Z_S G

, which he can do by only knowing the values of f

on S

using Lagrange interpolation.

The verifier can checks using the pairing equation

\displaystyle e(\pi, Z_S(s) H) e(f \% Z_S G, H) = e(C, H) \text{ .}

Naïve hashmap

A simple key-value store can be implemented using a hashmap primitive. We use a polynomial f

to store the roots of tries; each value f(i)

can be either 0 for an empty trie or the root of a trie. The key k

will be stored in the trie at location \mathrm{hash}(k) \% n

.

Performance

Computing a commitment takes n

group operations by using a Lagrange basis for commitments. Updates can be done locally. Proofs need O(n \log n)

field operations using Fourier transforms and n

group operations.

Under normal operation, this would give a very efficient access to the all the key-value pairs, as most tries would be depth 1 and thus do not need any additional data to be decommitted (all of these will be part of a single multi-reveal). The total witness size can be shrunk to the size of the data plus one group element. Only for rare collisions some additional hashes will be needed.

However, since the number of slots in the polynomial n

is “small” (ca. 2^{24}-2^{30}

), this allows a DOS attack on specific keys. An attacker knowing that a key will be inserted at location k (for example, because it is created by a contract and has a predictable address, or because an address is already in the pool) can create a large number of keys that will also be inserted at k

.

Hashmap with spam protection

We can remedy the DOS vector by assigning each key to δ

different points on the polynomial $\mathrm{hash}(k,1) \% n, \ldots, \mathrm{hash}(k,\delta) \% n$

.

We then add the rule that any key will have to be inserted into the trie at the least full location. This will mean each non-empty trie will need to have the number of elements stored (this could be an extra key, or it could be the root of size and the trie root).

In addition to the group element as witness, we will need to add the information which of the different keys it was stored under. This will add $\log \delta$

overhead per accessed element.

- Proofs that access already existing keys do not have any additional overhead
- A proof of insertion or non-existence will have to provide data for all δ

tries that the key can be in. However, we can limit the impact of this in the average case by providing shorter witnesses for empty tries.

DOS attack vectors

There are two possible DOS vectors to make inserting/accessing a key more expensive in this scheme:

1. An attacker can attack the specific δ

tries that a key can be inserted in. This needs computational resources n^δ

. For example, if $n=2^{28}$

, then at $\delta=3$

this becomes computationally infeasible (complexity 2^{84}

) unless massive ASIC resources similar to Bitcoin mining rigs are employed.

1. An attacker can choose to fill up ϵ

tries instead of just δ

. For each insertion, he will have to spend $(\frac{\epsilon}{n})^\delta$

computational resources. He will additionally need to perform ϵ

times the insertions needed to fill up just one tree. Let’s say 2^{60}

is a repeatedly achievable computational complexity and $\delta=3$

; then $\epsilon=2^8$

is needed, so he would need to pay the cost of filling up 2^8

tries to achieve the desired level of filling. I assume that this is prohibitively expensive.

Appendix: Multi-reveal

Another nice way to compute the multi-reveal is due to an observation by@JustinDrake: We can create a different form of multi-reveal that is simple to compute and evaluate.

Let f

be the polynomial of which we want to prove evaluations $y_0 = f(x_0), \ldots , y_{m-1} = f(x_{m-1})$

, which are all at in subgroup H

(which will be the case for us). Let g

be the polynomial that is 1

on y_i

and 0

otherwise. Then the polynomial $h = f \cdot g // Z_H$

can be computed as

$h = \sum_{i=0}^{m-1} q_i$

where

$q_i = (L_{x_i} f) // Z_H = \frac{x_i}{n} f // (x-x_i) \text{ .}$

L_{x_i}

is the Lagrange interpolating polynomial on H

that is 1 on x_i

. This is because

$\frac{L_{x_i} (x) }{Z(x)} =\frac{1}{x^n-1} \prod_{j \in [n]\x_i \ne \omega^j} \frac{x-\omega^j}{x_i - \omega^j} = \frac{1}{x-x_i} \prod_{j \in [n]\x_i \ne \omega^j} \frac{1}{x_i - \omega^j} = \frac{x_i}{n (x-x_i)}$

So q_i

is a constant multiple of the opening at x_i

. Note that all openings can be computed efficiently using the trick shown here [3].

The multi-opening proof is $\pi = h(s) G$

. The verifier can check this using the pairing equation

$e(C, g(s) H) = e(\pi, Z(s)) e(\sum_i y_i L_i(s) G, 1) \text{.}$

[1] Using polynomial commitments to replace state roots

[2] https://www.iacr.org/archive/asiacrypt2010/6477178/6477178.pdf

[3] https://github.com/khovratovich/Kate/blob/66aae66cd4e99db3182025c27f02e147dfa0c034/Kate_amortized.pdf