This post assumes familiarity with data availability sampling as inhttps://arxiv.org/abs/1809.09044.

The holy grail of data availability sampling is it we could remove the need for fraud proofs to check correctness of an encoded Merkle root. This can be done with polynomial commitments (including FRI), but at a high cost: to prove the value of any specific coordinate D[i] \in D

, you would need a proof that \frac{D - x}{X - i}

where x

is the claimed value of D[i]

, is a polynomial, and this proof takes linear time to generate and is either much larger (as in FRI) or relies on much heavier assumptions (as in any other scheme) than the status quo, which is a single Merkle branch.

So what if we just bite the bullet and directly use a STARK (or other ZKP) to prove that a Merkle root of encoded data D[0]...D[2n-1]

with degree < n

is “correct”? That is, we prove that Merkle root actually is the Merkle root of data encoded in such a way, with zero mistakes. Then, data availability sampling would become much easier, a more trivial matter of randomly checking Merkle branches, knowing that if more than 50% of them are available then the rest of D

can be reconstructed and it would be valid. There would be no fraud proofs and hence no network latency assumptions.

Here is a concrete construction for the arithmetic representation of the STARK that would be needed to prove this. Let us use the MIMC hash function with a sponge construction for our hash:

p = [modulus] r = [quadratic nonresidue mod p] k = [127 constants]

def mimc_hash(x, y): L, R = x, 0 for i in range(127): L, R = (L**3 + 3*q*L*R**2 + k[i]) % p, (3*L*2b + qR3) % p L = y for i in range(127): L, R = (L3 + 3*qLR**2 + k[i]) % p, (3*L**2*b + q*R**3) % p return L

The repeated loops are cubing in a quadratic field. If desired, we can prevent analysis over Fp^2 by negating the x coordinate (ie. L) after every round; this is a permutation that is a non-arithmetic operation over Fp^2 so the function could only be analyzed as a function over Fp of two values.

Now here is how we will set up the constraints. We position the values in D

at 128 * k

for 2n \le k < 4n

(so x[128 * (2n + i)] = D[i]

). We want to set up the constraint: x(128 * i) = H(x(128 * 2i) + x(128 * (2i+1))

, at least for i < 2n

; this alone would prove that x(128)

is the Merkle root of D

(we’ll use a separate argument to prove that D

is low-degree). However, because H

is a very-high-degree function, we cannot just set up that constraint directly. So here is what we do instead:

- We have two program state values, x

and y

, to fit the two-item MIMC hash function.

- For all i

where i\ \%\ 128 \ne 0

, we want x(i) = x(i+1)^3 + 3q*x(i+1)y(i+1)^2 + k(i)

and y(i) = 3*x(i+1)^2y(i+1) + q*y(i+1)^3

- For all i

where i\ \%\ 128 = 0

, we want: * If i < 128 * 2n

, x(i) = x(2i-127)

- If i\ \%\ 256 = 128

, y(i) = y(i+1)

- If i\ \%\ 256 = 0

, y(i) = 0

- If i < 128 * 2n

, x(i) = x(2i-127)

- If i\ \%\ 256 = 128

, y(i) = y(i+1)

- If i\ \%\ 256 = 0

, y(i) = 0

[Edit 2019.10.09: some have argued that for technical reasons the x(i) = x(2i-127)

constraint cannot be soundly enforced. If this is the case, we can replace it with a PLONK-style coordinate-accumulator-based copy-constraint argument]

[

extended_data(3)

1062×224 5.97 KB

](https://ethresear.ch/uploads/default/original/2X/b/bdd6020dc0095bde0595faa2aa80d26b49dbfe82.png)

We can satisfy all of these constraints by defining some piecewise functions: C_i(x(i), y(i), x(i+1), y(i+1), x(2i), k(i))

, where C_0

and C_1

might represent the first constraint, C_2

the second, C_3

the third and C_4

the last. We add indicator polynomials I_0

… I_4

for when each constraint applies (these are public parameters that can be verified once in linear time), and then make the constraints I_i(x) * C_i(x(i), y(i), x(i+1), y(i+1), x(2i)) = 0

. We can also add a constraint x(2n + 128 * i) = z(i)

and verify with another polynomial commitment that z

has degree < n

.

The only difference between this and a traditional STARK is that it has high-distance constraints (the x(i) = x(2i-127)

check). This does double the degree of that term of the extension polynomial, though the term itself is degree-1 so its degree should not even dominate.

Performance analysis

The RAM requirements for proving N bytes of original data can be computed as follows:

- Size of extended data: 2N bytes
- Total hashes in trace per chunk: 256
- Total size in trace: 512N bytes
- Total size in extended trace: 4096N bytes

Hence proving 2 MB of data would require 8 GB of RAM, and proving 512 MB would require 2 TB. The total number of hashes required for N bytes of data is (N/32) * 2 (extension) * 2 (Merkle tree overhead) = N/8, so proving 2 MB would require proving 2^{18}

hashes, and proving 512 MB would require proving 2^{26}

hashes.

An alternative way to think about it is, a STARK would be ~512 times more expensive than raw FRI. This does show that if we expect a Merkle root to be accessed more than 512 times (which seems overwhelmingly likely), then STARKing the Merkle root and then using Merkle proofs for witnesses is more efficient than using FRI for witnesses.

Current STARK provers seem to be able to prove ~600 hashes per second on a laptop, which implies a 7 minute runtime for 2 MB and a day-long runtime for 512 MB. However, this can easily be sped up with a GPU and can be parallelized further.