I used AirAssembly to create a zk-STARK which can be used to prove that verification of a Schnorr signature was executed correctly. Specifically:

Given a message m

, a public key P

, and a signature (R, s)

a prover can generate a proof that $s \cdot G = R + hash(P, R, m) \cdot P$

.

In the current implementation, the verifier needs to know m

, P

, and R

(but not s

) to check the proof. It should be possible (with some effort) to update the scheme so that the verifier can verify the proof with only m

and $hash(P)$

, and I believe this would result in a quantum-resistant signature scheme.

benchmarks

The STARK can be used to prove verification of one or more signatures. Proof sizes look roughly as follows:

- 1 signature: 110 KB
- 8 signatures: 140 KB
- 64 signatures: 180 KB

Extrapolating this further, it seems like a proof for 10K signatures should be somewhere around 300 KB in size. Moreover, with some optimizations, it may be possible to reduce proof sizes by 10% - 20%.

Proving time is currently rather slow: on my machine, it takes just under 2 seconds to prove a single signature verification, and a bit over 2 mins to prove 64 signature verifications. But, all is not lost:

- The field I’m using has not been optimized with WebAssembly - so, all math happens in JavaScript which is terribly slow. Moving math operations to WebAssembly should speed things up by a factor of 6x - 8x (and native code would be even faster than that).
- AirAssembly compiler hasn’t been optimized yet - so, the code it outputs is pretty inefficient. Optimizing it could speed things up by a factor of 2x - 3x.
- I’m running everything in a single thread. With multithreading, things will get significantly faster.

STARK structure

AirAssembly source code for the STARK is here and the runable example is here. Despite looking intimidating, the structure is rather simple:

- Execution trace has 14 registers:
- The first 7 are used to compute $s \cdot G$

,

- The other 7 are used to compute $R + h \cdot P$

, where h

is an input equal to $hash(P, R, m)$

.

- The first 7 are used to compute $s \cdot G$

,

- The other 7 are used to compute R + h \cdot P

, where h

is an input equal to hash(P, R, m)

.

- I use a simple double-and-add algorithm for elliptic curve multiplication:
- At each step the base point is doubled, and when needed, added to the accumulated result (x, y coordinates for base points and accumulated results account for 4 out of 7 registers used in each multiplication).
- I also pre-compute slopes for addition/doubling one step before the actual addition/doubling to keep constraint degree low.
- At each step the base point is doubled, and when needed, added to the accumulated result (x, y coordinates for base points and accumulated results account for 4 out of 7 registers used in each multiplication).
- I also pre-compute slopes for addition/doubling one step before the actual addition/doubling to keep constraint degree low.
- The total number of transition constraints is 18, and the highest constraint degree is 6.

The elliptic curve I'm currently using is P-224 because it is one of the standard curves defined over a “STARK-friendly” field. But swapping it out for some other curve is trivial.

If anyone sees any issues, or has any feedback - let me know!