Performing pairings within SNARK circuits opens up a variety of promising applications, includingtrustless bridges, light clients, and BLS signature aggregation for ERC4337, among others. In this post, I introduce a scheme known as SIPP (Statistically sound Inner Pairing Product), which efficiently computes the product of multiple pairings. I also present benchmark results from its implementation in plonky2.

## SIPP

SIPP is a method for aggregating pairings introduced inBMMTV19.

The goal of SIPP is for the Verifier to efficiently compute the pairing Z: = \vec{A}*\vec{B} = \prod_{i=0}^{n-1} e(A_i, B_i)

of \vec{A} \in \mathbb{G}_1^n

and \vec{B} \in \mathbb{G}_2^n

.

### Procedure

1. The Prover computes Z = A*B

and passes it to the Verifier.

The Prover and Verifier then follow these steps:

1. The Prover computes Z_L = \vec{A}*{[n/2:]} * \vec{B}*[:n/2]}

, Z_R = \vec{A}*{[:n/2]} * \vec{B}*[n/2:]}

and sends them to the Verifier. Here, \vec{A}_{[:n/2]}

represents the vector composed of the first n/2

elements of \vec{A}

, and \vec{A}_{[n/2:]}

represents the vector composed of the last n/2

elements of \vec{A}

.

1. The Verifier samples a random x \in \mathbb{F}_r

and provides it to the Prover.

1. Both the Verifier and Prover compute \vec{A}' = \vec{A}*{[:n/2]} + x \vec{A}*[n/2:]}

, \vec{B}' = \vec{B}*{[:n/2]} + x^{-1} \vec{B}*[n/2:]}

1. The Verifier computes Z' = Z_L^x Z Z_R^{x^{-1}}

2. The following updates are made: A\leftarrow A'

, B\leftarrow B'

, Z \leftarrow Z'

, n \leftarrow n/2

Once n = 1

, the Verifier checks if e(A, B) \overset{?}{=} Z

, and if so, accepts.

# Implementation with Plonky2

I implemented the SIPP Verifier in plonky2. The exponentiation operations in \mathbb{G}_1

, \mathbb{G}_2

, and \mathbb{F}_{q^{12}}

are costly in snark, so I divided them into separate circuits and aggregated them at the end. Particularly, the exponentiations in \mathbb{G}_2

and \mathbb{F}_{q^{12}}

are very costly even on their own. Hence, I decided to break down the exponent into 6 bits and 5 bits respectively and aggregate them at the end using recursive proof.

On an EC2 c6a.32xlarge

instance, the proof generation for exponentiation over BN254 required the following execution times:

G1 exponentiation: 12s

G2 exponentiation: total proof generation time for 6bits each circuits 13s

- aggregation 10s

= 23s

Fq12 exponentiation: total proof generation time for 6bits each circuits 118s

- aggregation 20s

= 138s

Applying SIPP to n

pairings requires G1 exponentiation and G2 exponentiation n-1

times each, and Fq12 exponentiation 2 \log_2 n

times. Therefore, the total time is 35 * (n-1) + 276 * log n

seconds, plus the time required for aggregating these proofs.

It's important to note that all proofs can be executed entirely in parallel, which means the execution time decreases inversely proportional to the number of threads available on the machine.

Here are the benchmarks when changing the value of n

. I performed this test while varying the value of LOG_N

.

- When n = 4

, proofs 507s

- aggregation 2s

= total 509s

- When n = 8

, proofs 776s

- aggregation 5s

= total 781s

- When n = 16

, proofs 1121s

- aggregation 9s

= total 1130s

For uses such as BLS signature aggregation in ERC4337, the cost is still quite high. However, this implementation involves

highly parallelizable processes that repeatedly prove the same circuit. Therefore, it is expected that increasing the number of machines would result in realistic processing times.

Recursive proof schemes for uniform circuits like Nova could potentially serve as alternatives to plonky2.