This week

Today - STARK theory and Polygon rollups
Tuesday - STARK theory 2 and zkEVM rollup approaches
Wednesday - Cryptographic alternatives
Thursday - Research / Review

STARK Theory

Polynomial Recap

A basic fact about polynomials and their roots is that if $p(x)$ is a polynomial, then $p(a) = 0$ for some specific value a,

if and only if there exists
a polynomial $q(x)$ such that
$(x - a)q(x) = p(x)$,

and therefore

$q(x) = \frac{p(x)}{(x-a)}$

and $deg(p) = deg(q) + 1$.

This is true for all roots

Process Overview

We are interested in Computational Integrity (CI), for example knowing that the Cairo program you wrote was computed correctly.

As with SNARKS we need to go through a number of transformations from the *trace* of our program, to the proof.
The first part of this is called arithmetisation, it involves taking our trace and turning it into a set of polynomials.

Our problem then becomes one where the prover that attempts to convince a verifier that the polynomial is of low degree (we will see why this is important later)

The verifier is convinced that the polynomial is of low degree if and only if the original computation is correct (except for an infinitesimally small probability).

Use of randomness

The prover uses randomness to achieve zero knowledge, the verifier uses randomness when generating queries to the prover, to detect cheating by the prover.

Succinctness and performance

Much of the work that is done in creating a proof is ensuring that it is succint and that it can be produced and verified in a reasonable time.

Arithmetisation

There are two steps

1. Generating an execution trace and polynomial constraints
2. Transforming these two objects into a single low-degree polynomial.

In terms of prover-verifier interaction, what really goes on is that the prover and the verifier agree on what the polynomial constraints are in advance.
The prover then generates an execution trace, and in the subsequent interaction, the prover tries to convince the verifier that the polynomial constraints are satisfied over this execution trace, unseen by the verifier.

The execution trace is a table that represents the steps of the underlying computation, where each row represents a single step

The type of execution trace that we're looking to generate must have the special trait of being succinctly testable — each row can be verified relying only on rows that are close to it in the trace, and the same verification procedure is applied to each pair of rows.

For example imagine our trace represents a running total, with each step as follows

Step	Amount	Total
0	0	0
1	5	5
2	2	7
3	2	9
4	3	12
5	6	18

If we represent the row as i , and the column as j , and the values as $A_{i,j}$
We could write some contraints about this as follows

$A_{0,2} = 0$

$\forall 1 >= i <= 5 : A_{i,2} - A_{i,1} - A_{i-1,2} = 0$

$A_{5,2} = 18$

These are linear polynomial constraints in $A_{i,j}$
Note that we are getting some succinctnesss here because we could represent a much larger number of rows with just these 3 constraints.

We want a verifier to ask a prover a very small number of questions, and decide whether to accept or reject the proof with a guaranteed high level of accuracy.
Ideally, the verifier would like to ask the prover to provide the values in a few (random) places in the execution trace, and check that the polynomial constraints hold for these places.
A correct execution trace will naturally pass this test.

However, it is not hard to construct a completely wrong execution trace (especially if we knew beforehand which points would be tested) , that violates the constraints only at a single place, and, doing so, reach a completely far and different outcome.
Identifying this fault via a small number of random queries is highly improbable.

But polynomials have some useful properties here
Two (different) polynomials of degree *d* evaluated on a domain that is considerably larger than *d* are different almost everywhere.
So if we have a dishonest prover, that creates a polynomial of low degree representing their trace (which is incorrect at some point) and evaluate it in a large domain, it will be easy to see that this is different to the *correct* polynomial.

Our plan is therefore to

1. Rephrase the execution trace as a polynomial
2. extend it to a large domain, and
3. transform that, using the polynomial constraints, into yet another polynomial that is guaranteed to be of low degree if and only if the execution trace is valid.

A more complex example

See article
Imagine our code calculates the first 512 Fibonacci sequence
1,1,2,3,5 ...

If we decide to operate on a finite field with max number 96769
And we have calculated that the 512th number is 62215.

Then our constraints are

$A_{0,2} - 1 = 0$
$A_{1,2} - 1 = 0$

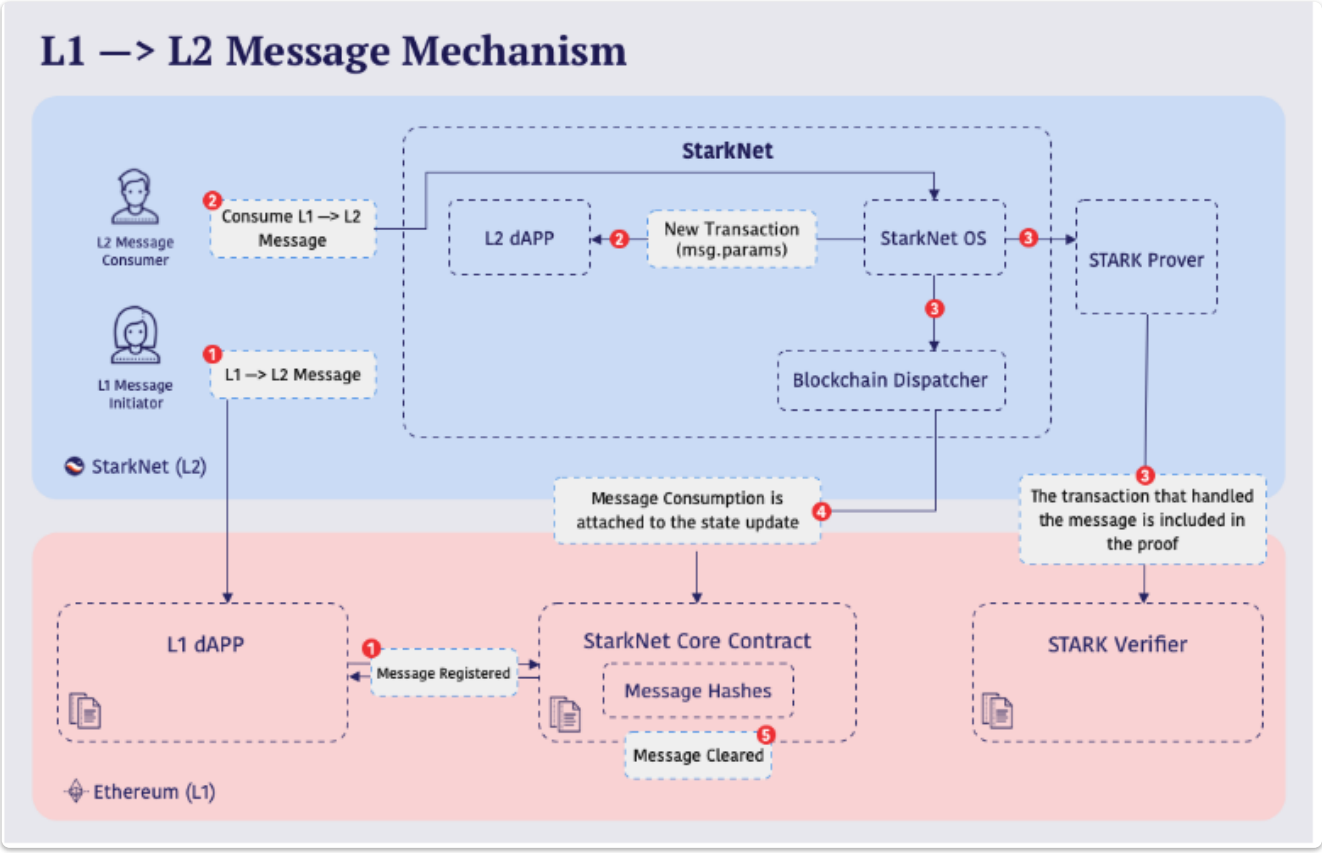
$\forall 0 >= i <= 510 : A_{i+2,2} = A_{i+1,2} + A_{i,2}$

$A_{511,2} - 62215 = 0$

Tomorrow we shall see how to develop this further.

Starknet Ecosystem

L1 to L2 Messaging

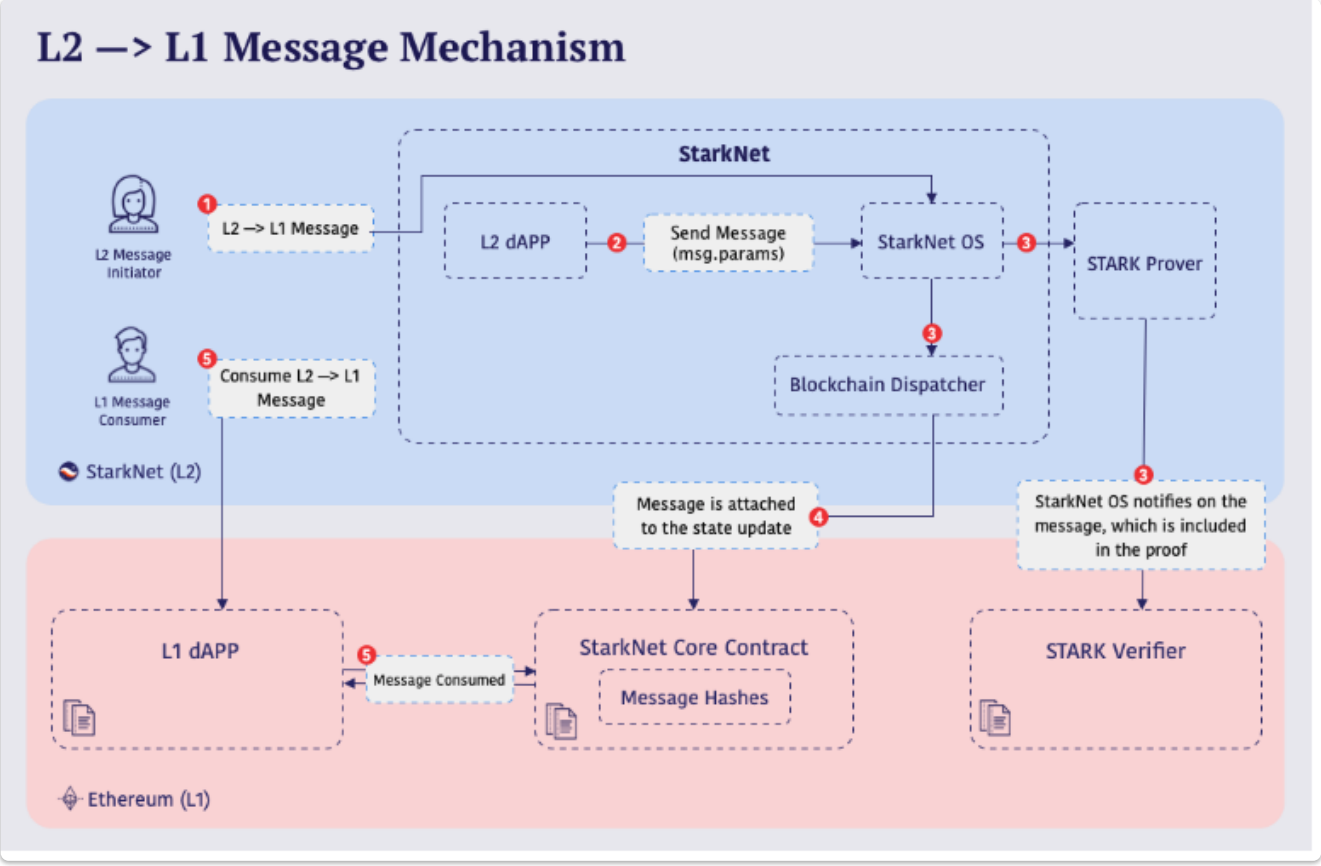


1. The L1 contract calls (on L1) the `send_message()` function of the StarkNet core contract, which stores the message. In this case the message includes an additional field - the "selector", which determines what function to call in the corresponding L2 contract.
2. The StarkNet Sequencer automatically consumes the message and invokes the requested L2 function of the contract designated by the "to" address.

This direction is useful, for example, for "deposit" transactions.

Note that while honest Sequencers automatically consume L1 -> L2 messages, it is not enforced by the protocol (so a Sequencer may choose to skip a message). This should be taken into account when designing the message protocol between the two contracts.

L2 to L1 Mesaging



Messages from L2 to L1 work as follows:

1. The StarkNet (L2) contract function calls the library function `send_message_to_l1()` in order to send the message. It specifies:

    1. The destination L1 contract ("to"),
    2. The data to send ("payload")

    The StarkNet OS adds the "from" address, which is the L2 address of the contract sending the message.

2. Once a state update containing the L2 transaction is accepted on-chain, the message is stored on the L1 StarkNet core contract, waiting to be consumed.

3. The L1 contract specified by the "to" address invokes the `consumeMessageFromL2()` of the StarkNet core contract.

Note: Since any L2 contract can send messages to any L1 contract it is recommended that the L1 contract check the "from" address before processing the transaction.

Starkgate Token Bridge

There are bridges for each token type available

Steps for Deposit

1. Call The Deposit Function on L1
    - Transfers the funds from the user account to the StarkNet bridge
    - Emits a deposit event
    - Sends a message to the relevant L2 bridge with details. The sequencer will now be aware of this.
2. Deposit Triggered on StarkNet
    - Mints the relevant token on starknet
3. A proof of the state change is created and accepted on L1.

Steps for Withdraw

1. Call The Withdraw Function on L2
    1. Burns the tokens on starknet
    2. Informs the L1 bridge with the details
2. A proof of the state change is created and accepted on L1.
3. Transfer The funds On L1 by calling the withdraw function on the L1 core contract.

Transfer Limits

Token	Max deposit	Max total value locked
ETH	0.25 Eth	320 Eth
DAI	50 Dai	100,000 Dai

Flash Loans on Starknet

https://github.com/tohrnii/flashloan-starknet/blob/main/contracts/FlashLoanLender.cairo

Polygon Products

See this guide

Polygon Zero

zkRollup solutions have a bottleneck in the time in takes to generate a proof.

Polygon Zero attempts to solve this with "recursive proofs", based on Plonky2.

Polygon Zero generates proofs simultaneously for every transaction in the batch. Theses are then aggregated into a single proof which is submitted on the Ethereum network.

This approach significantly reduces the effort it takes to generate reliable validity proofs. Polygon Zero's Plonky2 can generate a recursive proof in 0.17 seconds.

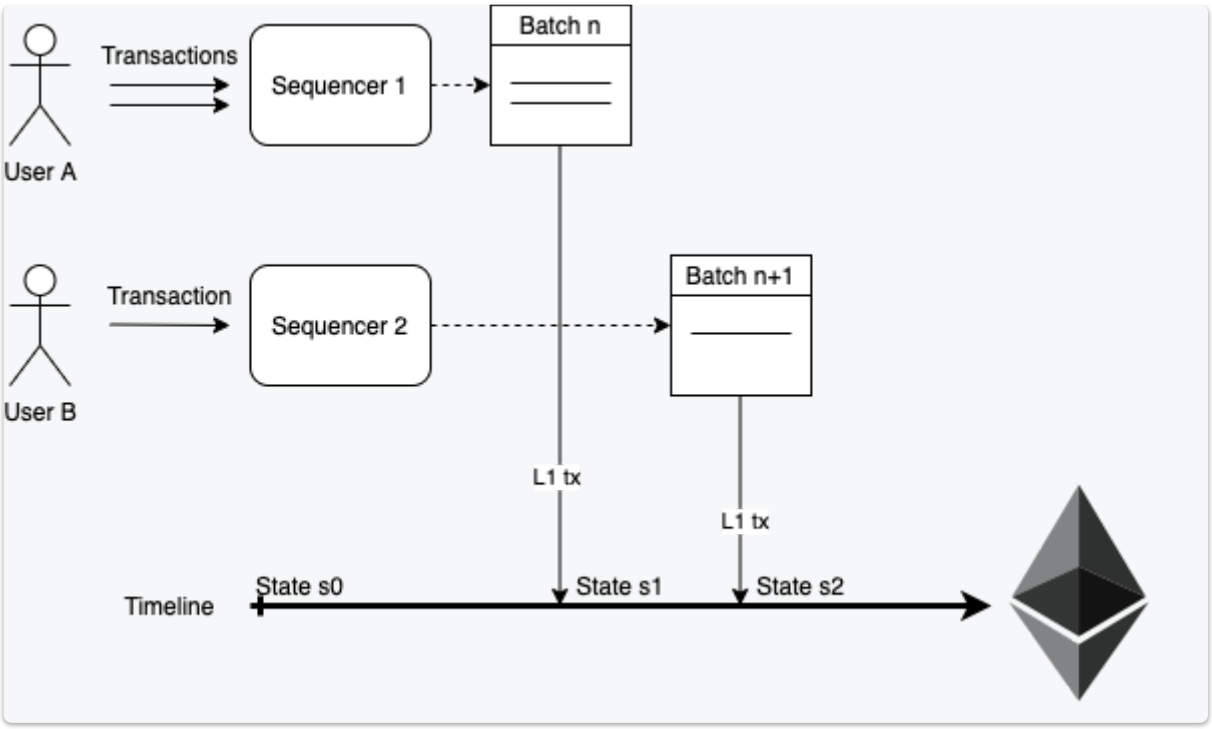
Polygon Hermez

Polygon (Hermez) team are working on a protocol Proof of Efficiency
This involves 2 permissionless roles :

- Sequencer
- Aggregator

From article

Sequencers collect transactions from users on the rollup, then select and pre-process new batches of this Layer 2 data. Finally, they send transactions to Layer 1 to be recorded. Sequencers also deposit a fee in $MATIC token as an incentive for Aggregators to include the batch in a zero knowledge proof.



Hermez 2.0

Hermez current functionality is limited to token transfers and atomic swaps, so there are plans to introduce Hermez 2.0 which will have EVM compatibility.

Polygon Midden

Polygon Miden is a general-purpose, STARK-based ZK rollup with EVM compatibility.
This will differ from Starknet in that it will be EVM compatible, so it should run Solidity contracts.
From their documentation
"Polygon Miden can process up to 5,000 transactions in a single block, with new blocks produced every five seconds. Although this ZK rollup exists as a prototype for now, it is expected to boost throughput to over 1,000 transactions per second (TPS) at launch."

Polygon Nightfall

From a collaboration with EY it is designed to allow private transactions.
It is a combination Optimistic rollups and zero knowledge, optimistic rollups for scalability and zk for privacy.
There is a beta version available on mainnet.