Lesson 7

Week 2

Lesson 5 - Understanding and Analysing Layer 2
Lesson 6 - Agnostic Layer 2 Transaction Lifecycle
Lesson 7 - Optimistic Rollups v ZK Rollups
Lesson 8 -What's next in Layer 2 part 1 : Decentralised Sequencers

Optimisitic versus ZK Rollups

Also see article

Summary of Differences

The core variances between ZK-rollups and Optimistic rollups stem from their unique methods of transaction validation and the generation of proofs.

Security ZK-rollups anchor their security in the cryptography of zero-knowledge proofs, allowing the base layer to accept the rollup's legitimacy without verifying every single transaction. This maintains the security assurances of the primary blockchain, offering users a substantial degree of trust.

In contrast, Optimistic rollups provide security differently. They depend on a fraud-proof system, assuming that any deceitful transactions will be identified and contested by validators. This method requires the community's ongoing scrutiny to oversee and dispute rollups. Though secure, it's not as infallible as the ZK-rollups' method, requiring continual community involvement.

Validation of zk rollups is provided by zero knowledge proofs which affirm the correctness of a transaction set without necessarily exposing any specific data. Created in the L2 these proofs assure the legitimacy of the encapsulated transactions. The blockchain’s base layer simply confirms the proof, reducing computational and storage demands on the L1.

Conversely, Optimistic rollups use fraud-proofs, presuming all transactions in a rollup are valid unless proven otherwise. If a fraudulent transaction is hidden within, there's a designated challenge period for individuals to present a fraud-proof. If validated, the rollup is reversed, and the perpetrator is penalised. While there's a risk of network spamming with bogus fraud proofs, economic incentives are in place to deter such behaviour.

ZK-rollups, with their reliance on zero knowledge proofs, present both advantages and drawbacks. Although these zero knowledge proofs offer security and privacy, they tend to restrict the functionality of sophisticated smart contracts due to the high computational resources required for generating and verifying proofs. In future lessons we will look in much more detail at the issues of compatibility and the challenges provided by using zero knowledge proofs.

Optimistic rollups fare better in this regard. They function in an environment that is closer to the L1 allow easier compatibility for complex smart contracts. This makes them a more flexible option when migrating DApps from L1 to L2.

For ZK rollups, transaction finality is instantaneous once the base layer authenticates the zero-knowledge proof, leading to quicker confirmation times than optimistic rollups, which due to their dependence on fraud proof systems, incorporate a challenge period allowing for disputes if fraudulent activities are found.
This period, which can span several hours, prolongs confirmation times relative to ZK-rollups.

Limitations of zk-Rollups

1. Intensive Computational Requirements: The application of Zero-Knowledge Proofs (ZKPs) in affirming the validity and authenticity of transactions demands substantial computational effort and time to create the proof. Consequently, this is an overhead to transaction processing and elevated costs associated with computational operations.
2. Reliance on External Verification: zk rollups necessitate external verifiability (a smart contract on the L1) , implying the need for outside parties to ascertain the accuracy of ZK proofs.
   This dependence can give rise to concerns regarding trust and potential risks related to centralisation.

Limitations of Optimistic Rollups

1. The need for the challenge period increases the time to finality.
2. Potential for Reversion: The framework of Optimistic Rollups is predicated on the "optimistic" premise that transactions are inherently valid and devoid of conflicts. Nonetheless, in instances of disputes or invalid transactions, part of the state is reverted.
3. Reliant on validators / users to provide fraud proofs, this can be seen as a censorship risk.

Finality options

Different rollups handle finality differently, but this from Polygon is fairly representative.

- Once the transaction has been received by the sequencer it is a *trusted* state, at this stage, there is no data on L1.
- Once the transaction data is available on L1, the transaction is in *virtual* state, the order of the transactions within the rollup block is fixed.
- Once the proof has been verified by the L1 verifier contract, the transaction is in *Verified* state. It is now safe to for example withdraw assets.

If the user trusts the sequencer, finality could be said to have been achieved when the transaction is in *trusted* state.

The process for zksync is shown here

- Fill a batch with transactions (usually takes a few minutes).
- Commit the batch to Ethereum.
- Generate a proof for the whole batch (usually takes around an hour).
- Submit the proof for verification by the Ethereum smart contract.
- Finalise the batch on Ethereum (delayed by ~21 hours as a security measure during the alpha phase).

When this process is complete, around ~24 hours in total, the batch is as final as any Ethereum transaction included in the same Ethereum block.

SNARK overview

The important letter from the acronym is S for Succinct
A SNARK is a triple (S, P, V)
$S(C)$ provides public parameters pp and vp (a setup phase)
$P(pp, x, w)$ is a short proof π
$V(vp, x, \pi)$ - succinct verification

C here is a circuit, a representation of the program or statement we wish to prove.

We require
len(π) = O_λ (log (|C|))
time(V) = O_λ (|x|, log (|C|))

Since verification needs to be succinct, vp provides a summary of the circuit.

How SNARKS work

Lets start from a high level view and dig into more and more details

Types of SNARKS

- Non universal
  Circuit specific trusted setup
  Large CRS required
- Universal
  Requires one trusted setup
  Circuit 'preprocessing' is transparent
  Smaller SRS than non universal
- Transparent
  No trusted setup
  Small CRS
  Large proof size

General Process

A broad description is

- Represent your program written in a DSL as an arithmetic circuit
- Convert your circuit into a polynomial identity
- Evaluate the polynomial identity using a Polynomial Commitment Scheme
- Make the process non interactive

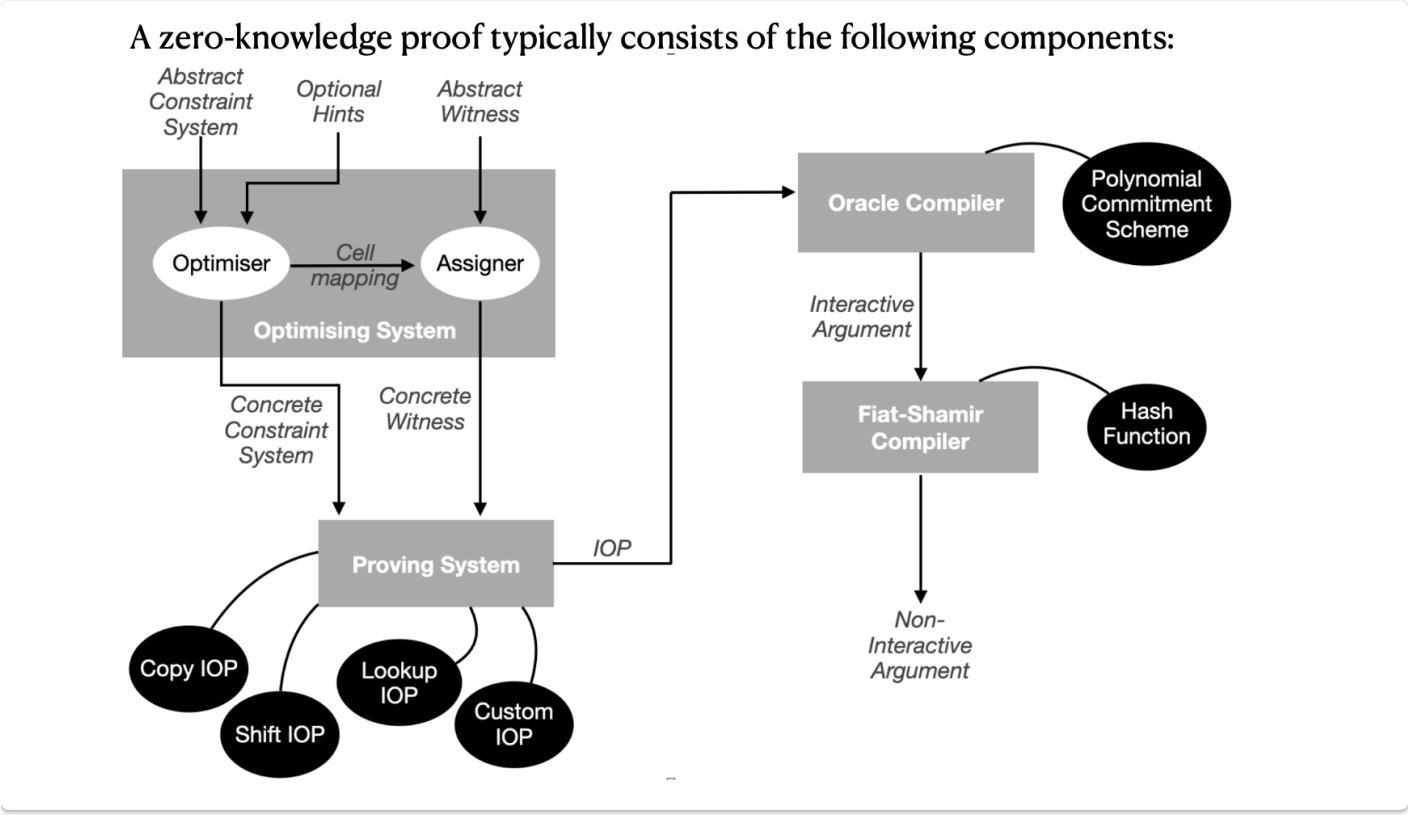
As a starting point we can imagine that our proof can be represented as a polynomial $P(x)$ and that this involves some constraints that need to be satisfied if the proof is correct.
There can be some inputs to the program, these can be public inputs, or private inputs (witness)

In a little more detail we

- Describe a circuit with addition and multiplication gates
- Represent n gates via a small number of degree n polynomials
- Use a PCS such as KZG

- Test the polynomial identity by evaluating the polynomials at random points. If the polynomial identity holds at a random point it is overwhelming likely to hold at all points

Diagram from Mary Maller presentation at ZKProofs conference



A general SNARK process may involve some of the following

- Arithmetisation
  - Flattening code
  - Arithmetic Circuit
  - R1CS
- Polynomials
- Polynomial Commitment Scheme
- Cryptographic proving system
- Make non interactive

You will also see talk of a front end and a back end.

Zero knowledge aspects can be built into to IOP layer as well as at cryptographic layer
Blinding at IOP layer is dependent on the PCS you use (the one you compile with)

Important things to keep in mind

- What do the prover and verifier know, the information asymmetry is difficult to get an intuition for.

- The different motivations of the prover and the verifier
- The fact that the verifier and creator may be the same
- The prover may have a restricted role in that they supply a witness but the rest of the process is automated and has been defined by the creator.

Starknet / STARK process

Computational Integrity

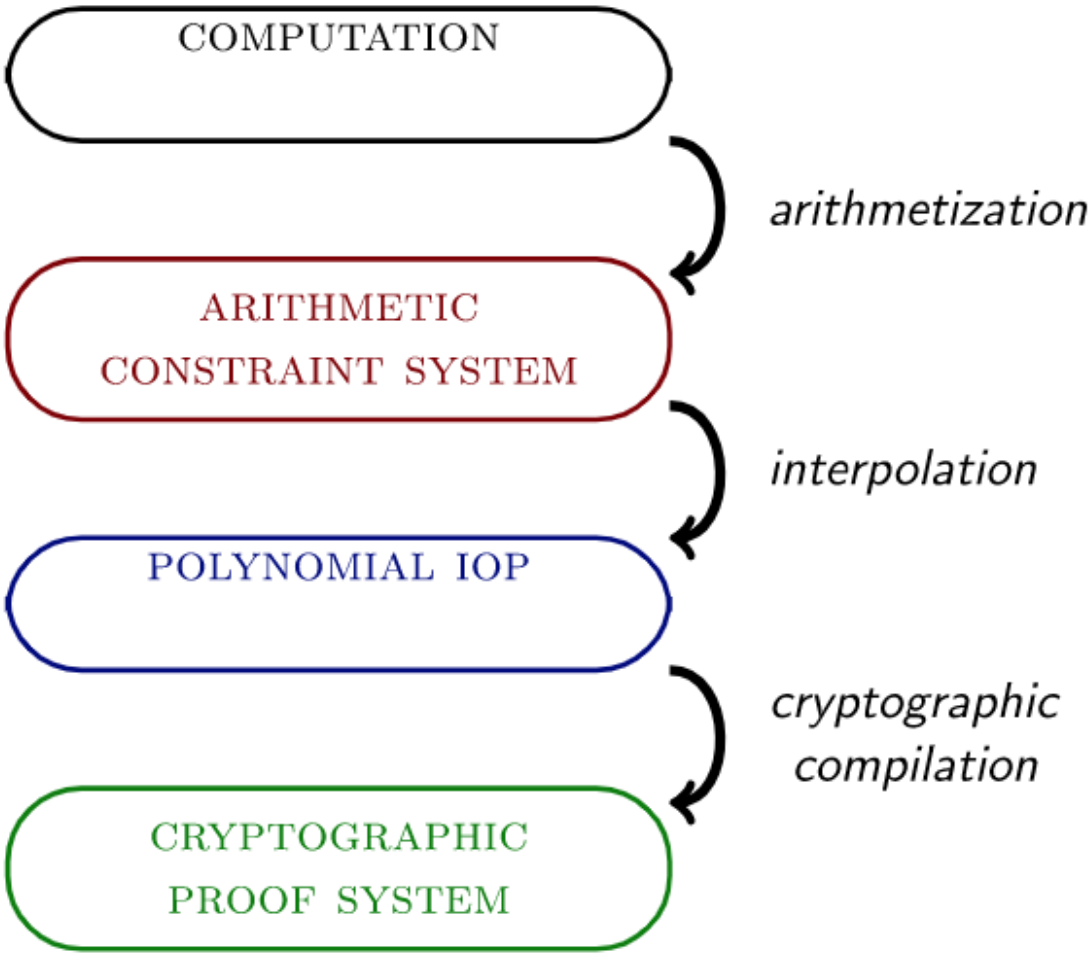
One of the (remarkable) features of zero knowledge proof systems is that they can be used to prove that some computation has been done correctly.
For example if we have a cairo program that is checking that a prover knows the square root of 25, they can run the program to test this, but the verifier needs to know that the computation was done correctly.

The issue of succinctness is important here, we want the time taken to verify the computation to be substantially less than the time taken to execute the computation, otherwise the verifier would just repeat the computation.

With the Starknet L2 we are primarily concerned that a batch of transactions has executed correctly giving a valid state change. Participants on the L1, wish to verify this, without the need to execute all the transactions themselves.

In the context of Starknet, computational integrity is more important than zero knowledge, all data on Starknet is public.

Overview of the Stark process



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

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.

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).

Arithmetisation - Simple example

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 succinctness 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.