Lesson 12 - Plonk Circuits / Scroll

Planned Updates to Cairo

Cairo v 1.0 - available in 6 months
Focus - Developer happiness

Sierra

A new intermediate level representation
Transactions should always be provable
Asserts are converted to if statements, if it returns false we don't do any modifications to storage
Contracts will count gas
Still needs to be low level enough to be efficient
So the process would be
Cairo (new) => Sierra => Cairo bytecode
Sierra bytecode

- cannot fail
- counts gas
- compiles to Cairo with virtually no overhead

Plonkish protocols

(fflonk, turbo PLONK, ultra PLONK, plonkup, and recently plonky2.)

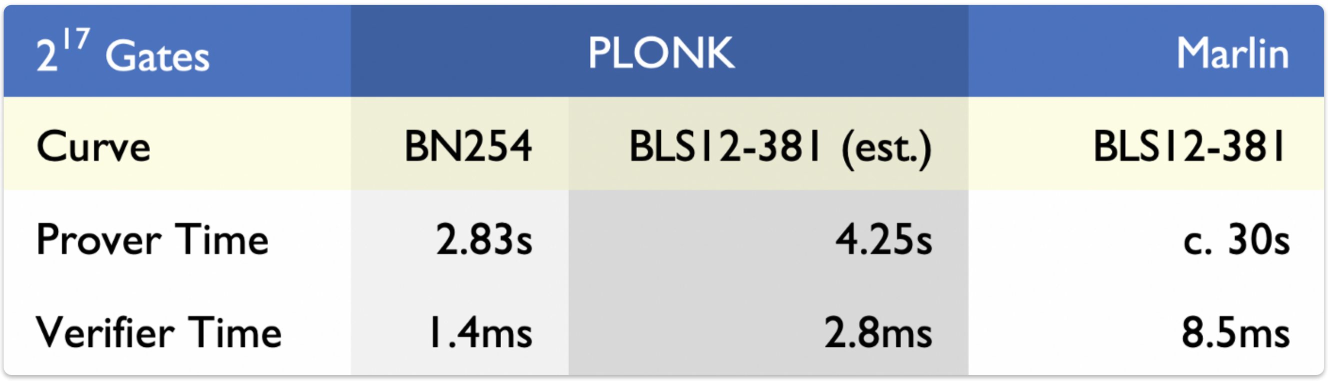
Before PLONK

Early SNARK implementations such as Groth16 depend on a common reference string, this is a large set of points on an elliptic curve.
Whilst these numbers are created out of randomness, internally the numbers in this list have strong algebraic relationships to one another. These relationships are used as short-cuts for the complex mathematics required to create proofs.
Knowledge of the randomness could give an attacker the ability to create false proofs.

A trusted-setup procedure generates a set of elliptic curve points $G, G \cdot s, G \cdot s^2 \ldots . G \cdot s^n$, as well as $G2 \cdot s$, where G and $G2$ are the generators of two elliptic curve groups and s is a secret that is forgotten once the procedure is finished (note that there is a multi-party version of this setup, which is secure as long as at least one of the participants forgets their share of the secret).
(The Aztec reference string goes upto the 10066396th power)

A problem remains that if you change your program and introduce a new circuit you require a fresh trusted setup.

In January 2019 Mary Maller and Sean Bowe et al released SONIC that has a universal setup, with just one setup, it could validate any conceivable circuit (up to a predefined level of complexity).
This was unfortunately not very efficient, PLONK managed to optimise the process to make the proof process feasible.



2^{17} Gates	PLONK		Marlin
Curve	BN254	BLS12-381 (est.)	BLS12-381
Prover Time	2.83s	4.25s	c. 30s
Verifier Time	1.4ms	2.8ms	8.5ms

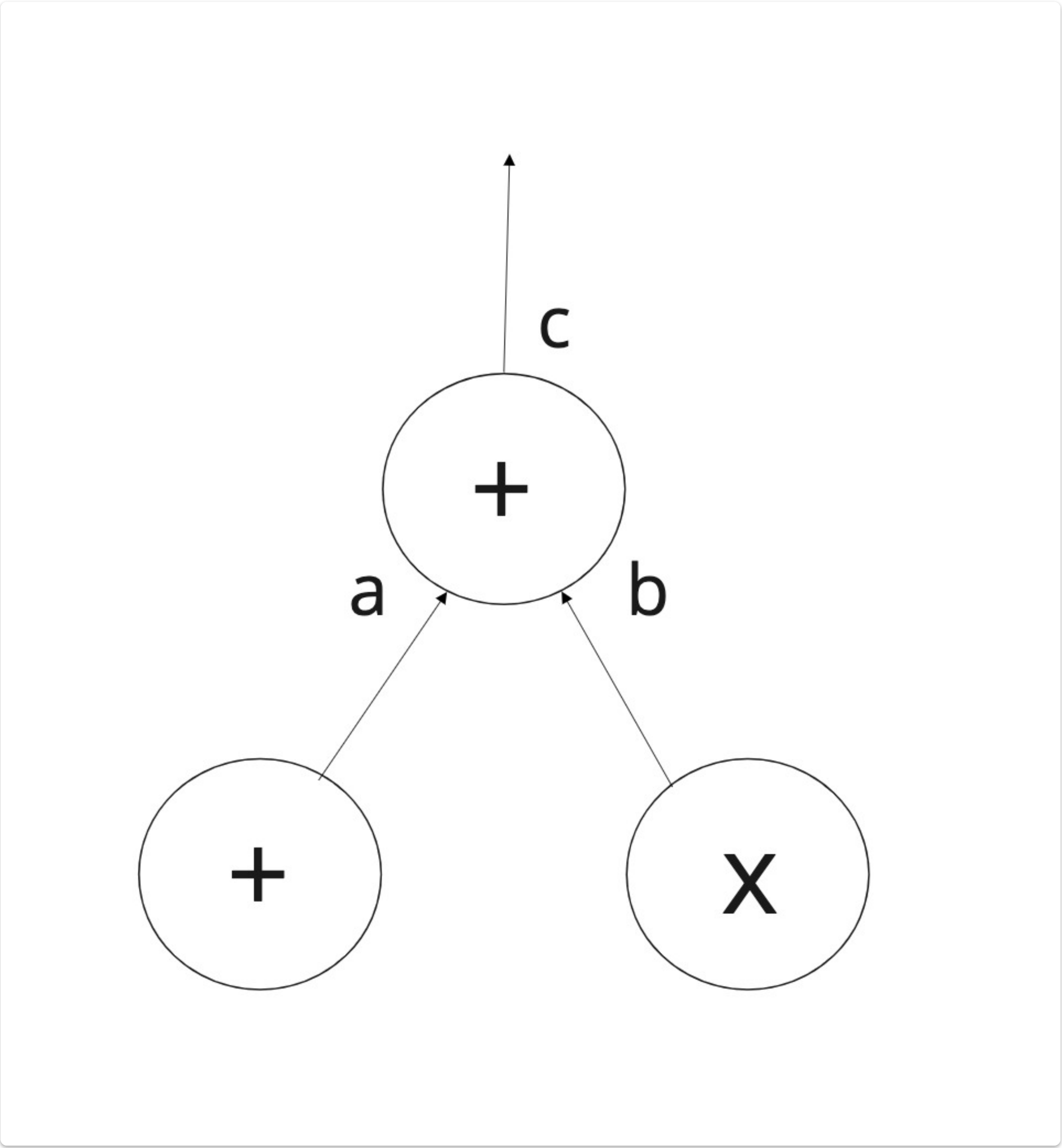
PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge
Also see Understanding PLONK

Trusted Setup

This is still needed, but it is a "universal and updateable" trusted setup.

- There is one single trusted setup for the whole scheme after which you can use the scheme with any program (up to some maximum size chosen when making the setup).
- There is a way for multiple parties to participate in the trusted setup such that it is secure as long as any one of them is honest, and this multi-party procedure is fully sequential:

Arithmetic Circuits



Developing a circuit

Once we have a (potentially large)circuit, we want to get it into a more usable form, so we can put the values into a table detailing the inputs and outputs for a gate.

So for gates 1 to i we can represent the a, b and c values as
a_i, b_i, c_i
And if the circuit is correct then for an addition gate

a_i+ b_i = c_i

or
a_i+ b_i - c_i = 0

and for a multiplication gate

a_i.b_i - c_i = 0

We would end up with a table like this

	a	b	c	S
1	a_1	a_1	a_1	1
2	a_2	b_2	c_2	1
3	a_3	b_3	c_3	0

But we would also want to know what type of gate it is, there is a useful trick where we introduce a selector S, which is 1 for an addition gate and 0 for a multiplication gate.

We can then generalise our equation as

$S_i(a_i + b_i) + (1 - S_i)a_i.b_i - c_i = 0$

These are called the *gate contraints* because they refer to the equalities for a particular gate.

We can also have *copy contraints* where we have a relationship between values that are no on the same gate, for example it may be the case that

a_7 = b_5 for a particular circuit, i fact this is how we link the gates together.

In PLONK we also have constant gates and more specialised gates.
For example representing a hash function as a series of generic gates (addition, multiplication and constant) would be inefficient.
From Zac Williamson

"PLONK’s strength is these things we call custom gates. It’s basically you can define your own custom bit arithmetic operations. I guess you can call them mini gadgets.
That are extremely efficient to evaluate inside a PLONK circuit.
So you can do things like do elliptical curve point addition in a gate.
You can do things like efficient Poseidon hashes, Pedersen hashes. You can do parts of a SHA-256 hash. You can do things like 8-bit logical XOR operations.
All these like explicit instructions which are needed for real world circuits, but they’re all quite custom."

Plookup

Some operations such as boolean operations do not need to be computed, but can be put into a lookup table. Similarly public data can be put in a lookup table.
For example we could use a lookup table for the XOR operation, or include common values as we saw in the range proofs in Aztec.

From circuit to R1CS to QAP

Once we have our circuit we can transform it into a set of polynomials which allows us to represent the large amount of data in a compact way.

Polynomial Commitments

A problem we face is that we can construct polynomials to represent our contraints, but these end up being being quite big.

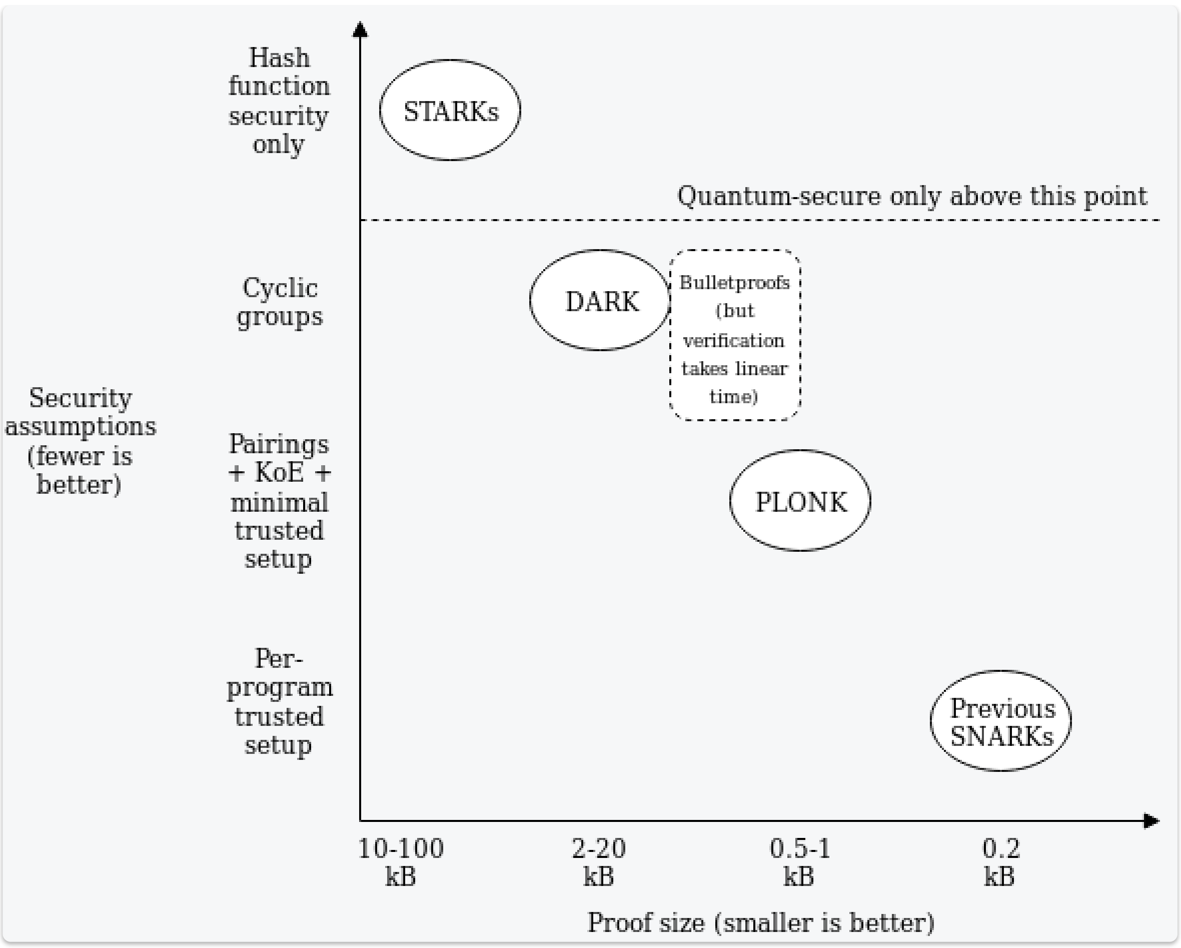
A polynomial commitment is a short object that "represents" a polynomial, and allows you to verify evaluations of that polynomial, without needing to actually contain all of the data in the polynomial.

That is, if someone gives you a commitment c representing $P(x)$, they can give you a proof that can convince you, for some specific z, what the value of $P(z)$ is.

There is a further mathematical result that says that, over a sufficiently big field, if certain kinds of equations (chosen before z is known) about polynomials evaluated at a random z are true, those same equations are true about the whole polynomial as well.

For example,
if $P(z) \cdot Q(z) + R(z) = S(z) + 5$, where z is a specific point
then we know that it's overwhelmingly likely that
$P(x) \cdot Q(x) + R(x) = S(x) + 5$ in general.

PLONK uses Kate commitments based on trusted setup and elliptic curve pairings, but these can be swapped out with other schemes, such as FRI (which would turn PLONK into a kind of STARK


Hash function security only
STARKs
Quantum-secure only above this point
Cyclic groups
DARK
Bulletproofs (but verification takes linear time)
Security assumptions (fewer is better)
Pairings + KoE + minimal trusted setup
PLONK
Per-program trusted setup
Previous SNARKs
10-100 kB
2-20 kB
0.5-1 kB
0.2 kB
Proof size (smaller is better)

This means the arithmetization - the process for converting a program into a set of polynomial equations can be the same in a number of schemes.
If this kind of scheme becomes widely adopted, we can thus expect rapid progress in improving shared arithmetization techniques.

For a detailed talk about some of the custom gates used in Plonk see this video

From Scroll documentation

Recent advances to improve efficieny of SNARKS

- **The usage of polynomial commitment.** In the past few years, most succinct zero-knowledge proof protocols stick to R1CS with a query encoded in an application-specific trusted setup.
- The circuit size usually blows up and you can't do many customized optimizations since the degree of each constraint needs to be 2 (bilinear pairing only allows one multiplication in the exponent).
- With polynomial commitment schemes, you can lift your constraints to any degree with a universal setup or even transparent setup. This allows great flexibility in the choice of backend.
- **The appearance of lookup table arguments and customized gadgets.** Another strong optimization comes from the usage of lookup tables. The optimization is firstly proposed in Arya and then gets optimized in Plookup. This can save A LOT for zk-unfriendly primitives (i.e., bitwise operations like AND, XOR, etc.) . Customized

gadgets allow you to do high degree constraints with efficiency. TurboPlonk and UltraPlonk defines elegant program syntax to make it easier to use lookup tables and define customized gadgets. This can be extremely helpful for reducing the overhead of EVM circuit.

- **Recursive proof is more and more feasible.** Recursive proof has a huge overhead in the past since it relies on special pairing-friendly cyclic elliptic curves (See). This introduces a large computation overhead. However, more techniques are making this possible without sacrificing the efficiency.
- For example, Halo can avoid the need of pairing-friendly curve and amortize the cost of recursion using special inner product argument. Aztec shows that you can do proof aggregation for existing protocols directly (lookup tables can reduce the overhead of non-native field operation thus can make the verification circuit smaller). It can highly improve the scalability of supported circuit size.
- **Hardware acceleration is making proving more efficient.** To the best of our knowledge, we have made the fastest GPU and ASIC/FPGA accelerator for the prover. Our paper describing ASIC prover has already been accepted by the largest computer conference (ISCA) this year. The GPU prover is around 5x-10x faster than Filecoin's implementation. This can greatly improve the prover's computation efficiency.

Scroll

This project is still at en early stage, there alpha testnet will be run on a private PoA fork of Ethereum (the testnet L1) operated by Scroll.
On top of this private chain, will run a testnet Scroll L2 supporting the following features:

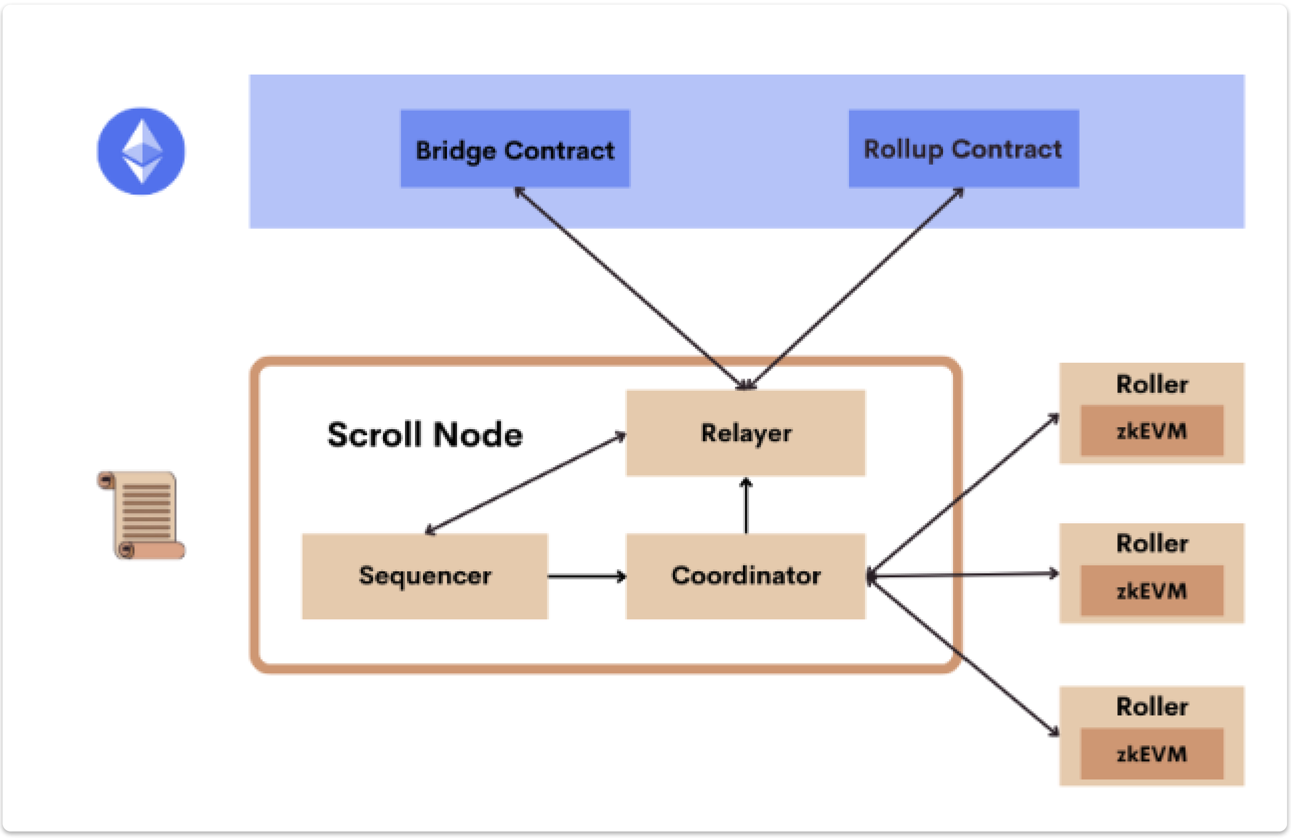
- Users will be able to play with a few key demo applications such as a Uniswap fork with familiar web interfaces such as Metamask.
- Users will be able to view the state of the Scroll testnet via block explorers.
- Scroll will run a node that supports unlimited read operations (e.g. getting the state of accounts) and user-initiated transactions involving interactions with the pre-deployed demo applications (e.g. transfers of ERC-20 tokens or swaps of tokens).
- Rollers will generate and aggregate validity proofs for part of the zkEVM circuits to ensure a stable release. In the next testnet phase, we will ramp up this set of zkEVM circuits.
- Bridging assets between these testnet L1 and L2s will be enabled through a smart contract bridge, though arbitrary message passing will not be supported in this release.

From article and article

Approaches to zkRollups on Ethereum

1. Building application-specific circuit (although this can be fairly generic as in Starknet)
2. Building a universal "EVM" circuit for smart contract execution

Scroll Architecture



Components

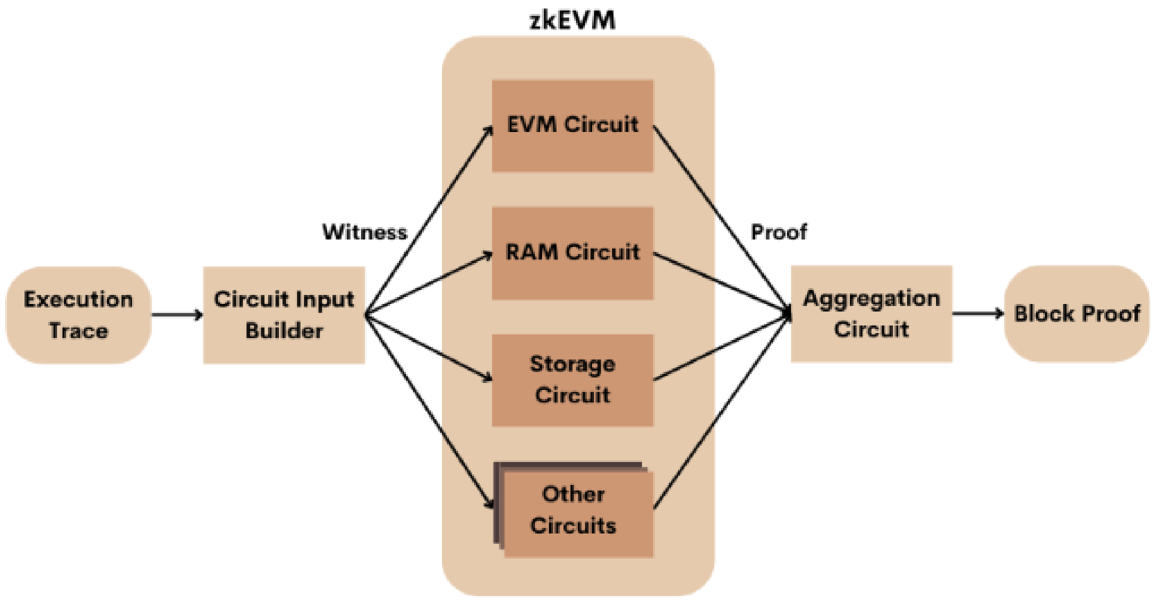
The Sequencer provides a JSON-RPC interface and accepts L2 transactions. Every few seconds, it retrieves a batch of transactions from the L2 mempool and executes them to generate a new L2 block and a new state root.

Once a new block is generated, the Coordinator is notified and receives the execution trace of this block from the Sequencer.
It then dispatches the execution trace to a randomly-selected Roller from the roller pool for proof generation.

The Relayer watches the bridge and rollup contracts deployed on both Ethereum and Scroll. It has two main responsibilities.

1. It monitors the rollup contract to keep track of the status of L2 blocks including their data availability and validity proof.
2. It watches the deposit and withdraw events from the bridge contracts deployed on both Ethereum and Scroll and relays the messages from one side to the other.

Rollers - creating proofs

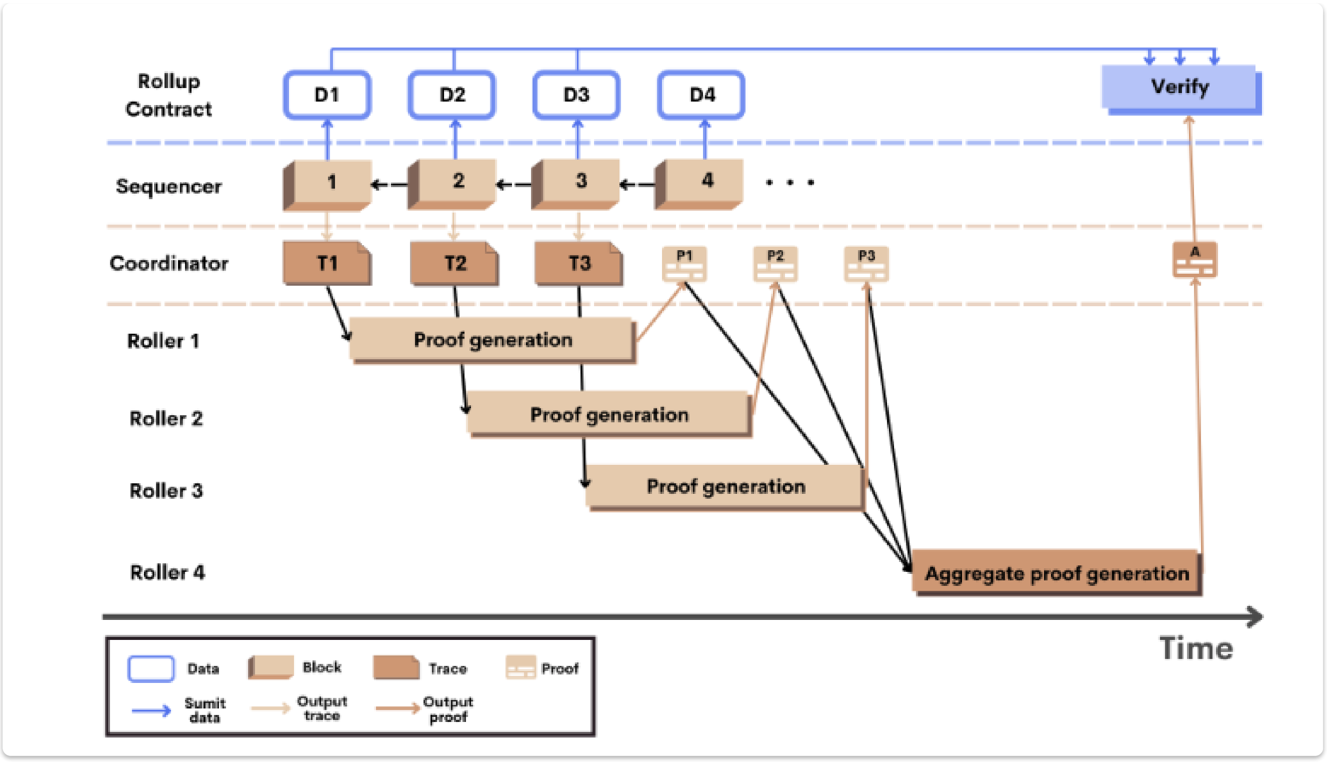


The Rollers serve as provers in the network that are responsible for generating validity proofs for the zkRollup

- A Roller first converts the execution trace received from the Coordinator to circuit witnesses.
- It generates proofs for each of the zkEVM circuits.
- Finally, it uses proof aggregation to combine proofs from multiple zkEVM circuits into a single block proof.

The Rollup contract on L1 receives L2 state roots and blocks from the Sequencer.
It stores state roots in the Ethereum state and L2 block data as Ethereum calldata.
This provides data availability for Scroll blocks and leverages the security of Ethereum to ensure that indexers including the Scroll Relayer can reconstruct L2 blocks.
Once a block proof establishing the validity of an L2 block has been verified by the Rollup contract, the corresponding block is considered finalized on Scroll.

A useful sequence diagram from the Scroll Documentation



L2 blocks in Scroll are generated, committed to base layer Ethereum, and finalized in the following sequence of steps:

1. The Sequencer generates a sequence of blocks. For the *i*-th block, the Sequencer generates an execution trace ***T*** and sends it to the Coordinator. Meanwhile, it also submits the transaction data ***D*** as calldata to the Rollup contract on Ethereum for data availability and the resulting state roots and commitments to the transaction data to the Rollup contract as state.
2. The Coordinator randomly selects a Roller to generate a validity proof for each block trace. To speed up the proof generation process, proofs for different blocks can be generated in parallel on different Rollers.
3. After generating the block proof ***P*** for the *i*-th block, the Roller sends it back to the Coordinator. Every *k* blocks, the Coordinator dispatches an aggregation task to another Roller to aggregate *k* block proofs into a single aggregate proof ***A***.
4. Finally, the Coordinator submits the aggregate proof ***A*** to the Rollup contract to finalize L2 blocks *i+1* to *i+k* by verifying the aggregate proof against the state roots and transaction data commitments previously submitted to the rollup contract.

EVM processing

In general the EVM will

1. Read elements from stack, memory or storage
2. Perform some computation on those elements
3. Write back results to stack, memory or storage

So our circuit has to model / prove this process, in particular

- The bytecode is correctly loaded from persistent storage
- The opcodes in the bytecode are executed in sequence
- Each opcode is executed correctly (following the above 3 steps)

Design challenges in designing a zkEVM

1. We are constrained by the cryptography (curves, hash functions) available on Ethereum.
2. The EVM is stack based rather than register based
3. The EVM has a 256 bit word (not a natural field element size)
4. EVM storage uses keccak and Merkle Patricia trees, which are not zkp friendly
5. We need to model the whole EVM to do a simple op code.

Scroll circuit design

1. We need an accumulator to provide the proofs of storage, merkle trees can provide this
2. The execution trace is needed to show the path that the execution took through the bytecode, as this would change because of jumps. This trace is then a witness provided to the circuit.
3. Two proofs are used to show the execution is correct for each opcode
    1. Proof of fetching the data required for the opcode
    2. Proof that the opcode executed correctly.

Scroll are working with Ethereum on this, see this repo for EVM circuit design, and this design document from Ethereum.