Lesson 11

Week 3

Lesson 9 - What's next in L2 part 2 : L3s/Hyperchains
Lesson 10 - Privacy in Layer 2
Lesson 11 - What are ZK EVMs part 1 - overview
Lesson 12- What are ZK EVMs part 2 - universal circuits/circuit compiler

> ### zkVM introduction
>
> #### Background - Virtual Machines
>
> See article
> Virtual Machines (VMs) are designed to provide the functionality a computer, usually wholly in software (there is a subtle difference between virtualisation and emulation).
> The Instruction Set Architecture (ISA) is part of the abstract model of a computer that defines how the CPU is controlled by the software.
> There are 3 main types , register based, stack based and accumulator based, most VMs are of the first 2 types.
> The Ethereum Virtual Machine is stack based.
>
> In this course we will look at zkEVMs such as Polygon zkEVM / zkSync Era, and also zkVMs such as Risc zero.
>
> A zkVM is a zero-knowledge proof-based virtual machine that combines a zk proof and a Virtual Machine. The zkVM typically consists of two important components: a compiler that can compile high-level languages such as C++ and Rust into intermediate (IR) expressions for the ZK system to perform; the other is the ISA (Instruction Architecture) instruction set framework, which mainly executes instructions about CPU operations and is a series of instructions used to instruct the CPU to perform operations.

zkEVM Introduction

A zkEVM is a virtual machine designed and developed to emulate the Ethereum Virtual Machine (EVM) by recreating all existing EVM opcodes for transparent deployment of existing Ethereum smart contracts.

The zkEVM, acts as a state machine, processes state transitions stemming from the execution of Ethereum's Layer 2 transactions, which users transmit to the network.
After this, it generates validity proofs that confirm the accuracy of the off-chain state change computations.

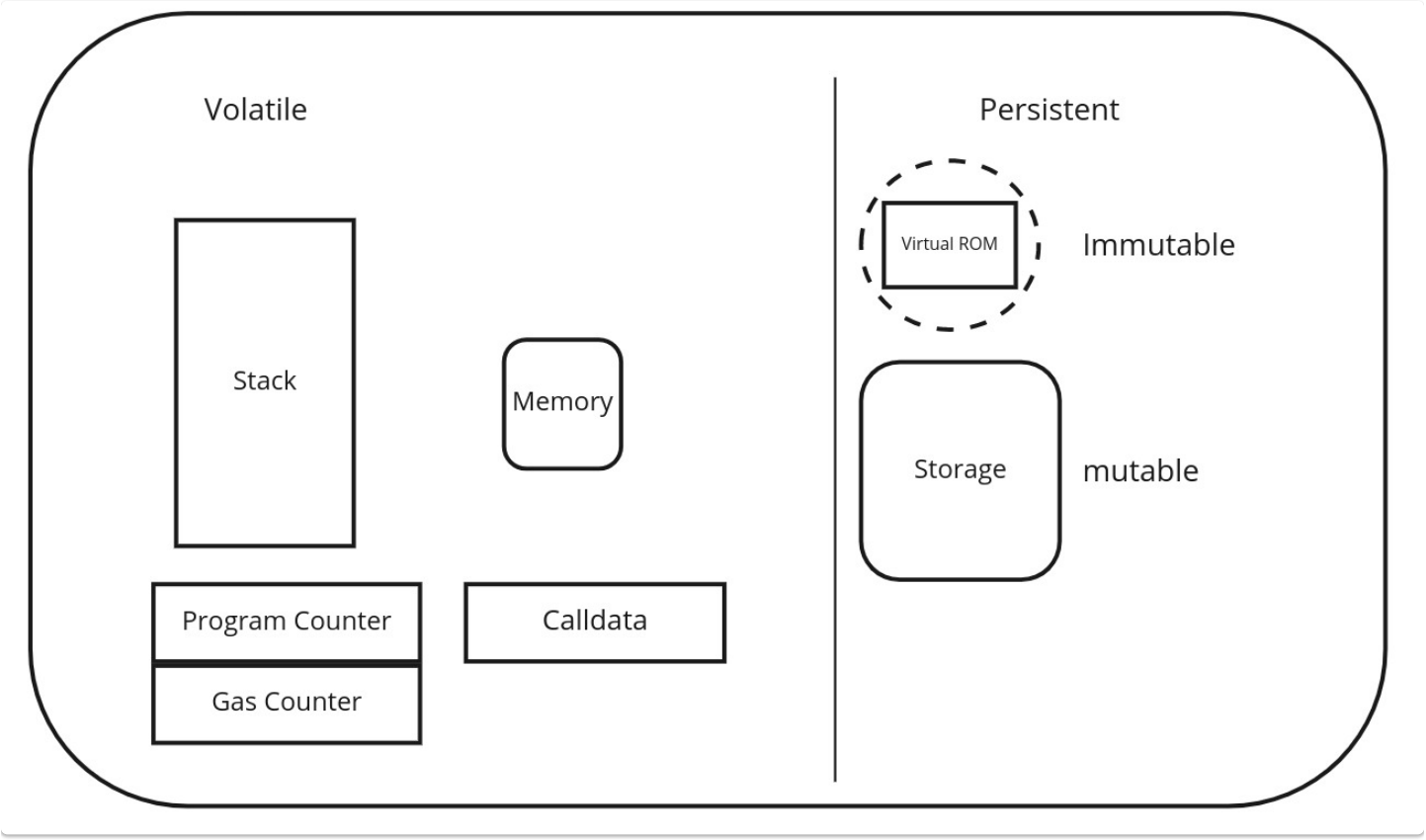
Building general DApps in zk-Rollup

From Scroll Docs

Of the possible ways to build a zkEVM we shall concemtrate on those which will run general DApps, rather than being application specific.

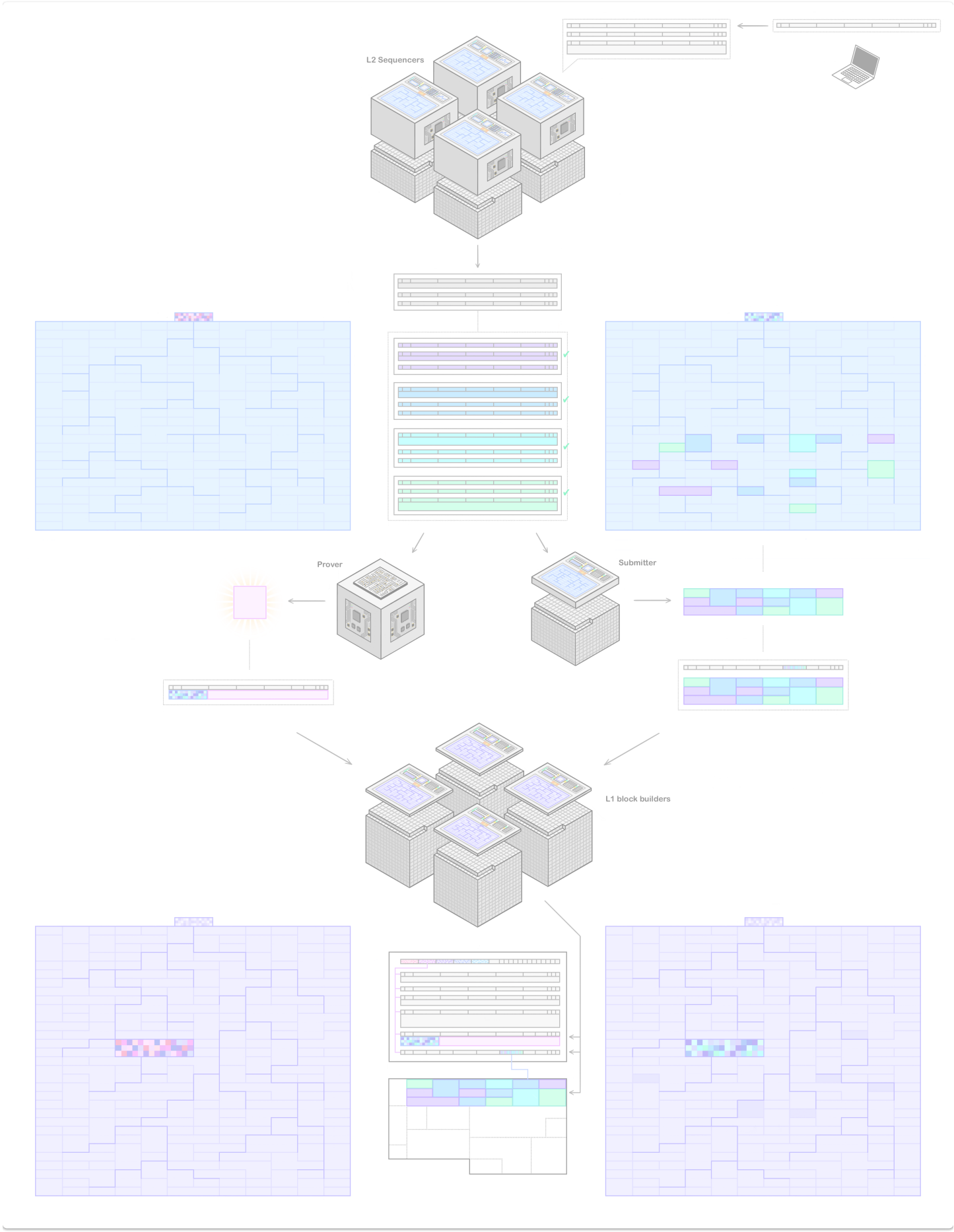
The architecture of the zkEVM Layer 2 will follow the architecture we have already seen, having a sequencer and prover (or multiples of each) , L1 <-> L2 messaging capabilities and a contract on the L1.

The EVM Architecture



The opcode of the EVM needs to interact with Stack, Memory, and Storage during execution. There should also be some contexts, such as gas/program counter, etc.
Stack is only used for Stack access, and Memory and Storage can be accessed randomly.
A zkEVM would seek to emulate all of these components and their interactions and create proofs that the interactions were correct.

zkEVM Rollup Architecture



Timeline

2013 - TinyRAM
2018 - Spice
2020 - zkSync
2021 - Cairo VM
2022 - Polygon zkEVM / Scroll / Risc Zero
2023 - Powdr

zkEVM Phases

Proof focussed

- Circuit creation
- Setup
    - Parameters created - gives proving key and verification key
- Proof creation
- Proof aggregation
- Proof acceptance on L1 and verification

L2 aspects

As we have seen for validity rollups we also have processes to

- Submit data to the DA layer
- Allow L1 <-> L2 messaging
- Provide an escape hatch via forced transactions

zkEVM workflow

The general workflow for an zkEVM would be

- Receive a transaction
- Execute the relevant bytecode
- Make state changes and transaction receipts
- Using the zkEVM circuits with the execution trace as input produce a proof of correct execution
- Aggregate proofs for a bundle of transactions and submit them to L1
- Submit data to the appropriate data availability layer.

Comparing the L1 and L2 processing

Even though the external workflow remains unchanged, there are significant differences in the underlying processing procedures for Layer 1 and Layer 2:

- Layer 1 relies on the re-execution of smart contracts.
- Layer 2 relies on the validity proof of zkEVM circuits.

In Layer 1, the bytecode of deployed smart contracts are stored in Ethereum storage. Transactions are broadcasted across a peer-to-peer network. For each transaction, every full node must load the corresponding bytecode and execute it on the Ethereum Virtual Machine (EVM) to reach the same state, using the transaction as input data.

In Layer 2, bytecode is also stored in storage, and users follow a similar process. Transactions are sent off-chain to a centralized zkEVM node. Instead of directly executing the bytecode, zkEVM generates a succinct proof demonstrating that the states are correctly updated after applying the transactions. Finally, the Layer 1 contract verifies the proof and updates the states without re-executing the transactions.

On L1 the EVM loads the bytecode and executes the opcodes within it one by one.

Each opcode can be broken down into three sub-steps:

- Reading elements from the stack, memory, or storage,
- Performing computations on these elements, and
- Writing back the results to the stack, memory, or storage. For instance, the "add" opcode reads two elements from the stack, adds them together, and writes the result back to the stack.

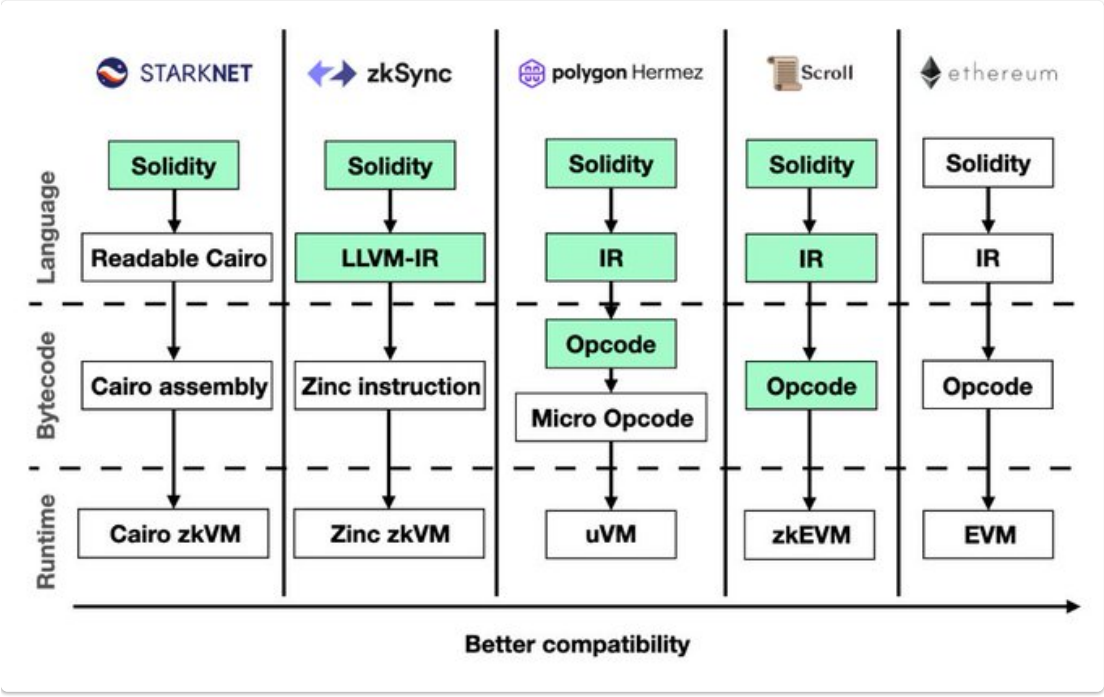
Thus, it is evident that zkEVM's proof must encompass the following aspects related to the execution process:

- Confirming that the bytecode is accurately loaded from persistent storage (ensuring the correct opcode is loaded from a specified address).
- Demonstrating that the opcodes in the bytecode are executed sequentially without missing or skipping any opcode.
- Verifying the correct execution of each opcode, including the proper execution of the three sub-steps within each opcode (Read/Write and computation
  We will see this in much more detail in Lesson 12.

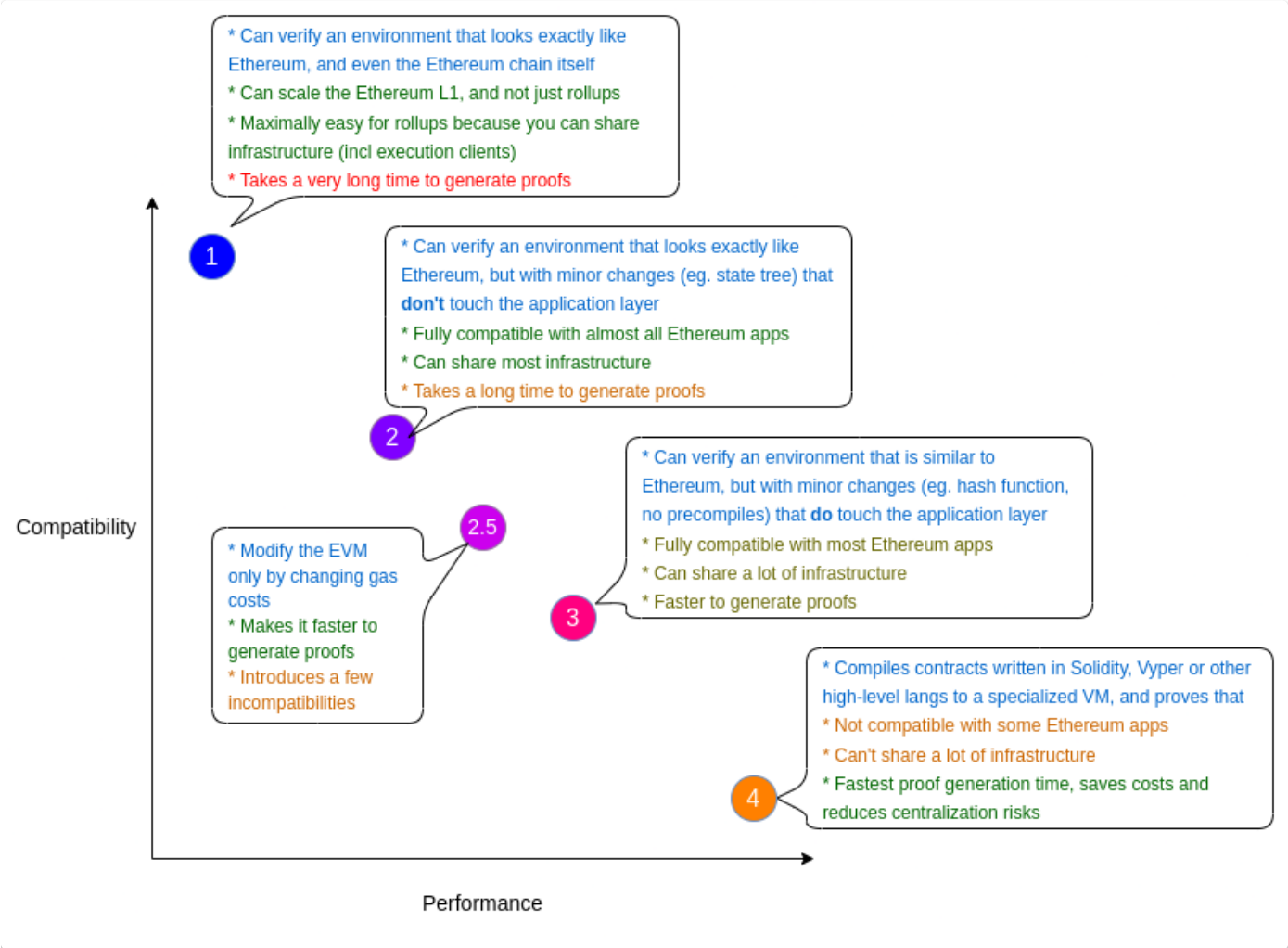
zkEVM taxonomy

From article and article

Approaches to zkRollups on Ethereum



See Vitalik's article



Type 1 (fully Ethereum-equivalent)

See zkEVM research team

Type 2 (fully EVM-equivalent)

(not quite Ethereum-equivalent)

Type 2.5 (EVM-equivalent, except for gas costs)

Type 3 (almost EVM-equivalent)

Type 4 (high-level-language equivalent)

Also see
The ZK-EVM Community Edition (bootstrapped by community contributors including Privacy and Scaling Explorations, the Scroll team, Taiko and others) is a Tier 1 ZK-EVM.

zkEVM Transaction Lifecycle

A good overview is given by Polygon

zkEVM Proving System

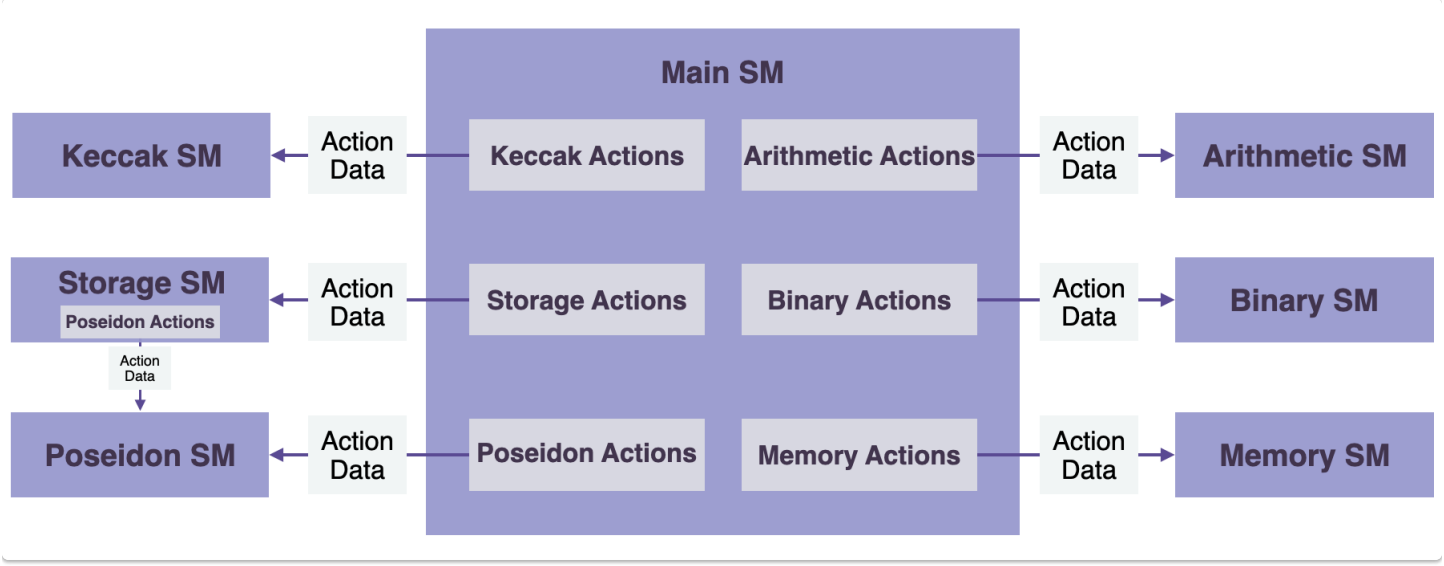
An overview is provided by Polygon

The prover is responsible for ensuring that all the rules for a transaction to be valid are enforced, otherwise the proof would have no meaning.

The circuit / proof is for any general contract, and not specific to a particular application such as a DEX.
We need to check that among other things that

- the contract has been loaded correctly,
- transaction signatures are correct
- the changes in state are correct
- the execution proceeded correctly.

A useful abstraction for the Ethereum blockchain is to see it as a state machine, we can in fact take this further and see the EVM as a number of state machines.
The interaction between parts within the EVM is complex as shown in this diagram from Polygon



The PSE group from the EF have published specifications of the EVM.
They see two components to the proof :

1. State proof: State/memory/stack ops have been performed correctly. This does not check if the correct location has been read/written. We

allow our prover to pick any location here and in the EVM proof confirm it is correct.

2. EVM proof: This checks that the correct opcode is called at the correct time. It checks the validity of these opcodes and confirms that each of these opcodes and the state proof both performed the correct operations.

Main zkEVM Projects

From L2Beat

TVL for all L2s

Value Locked

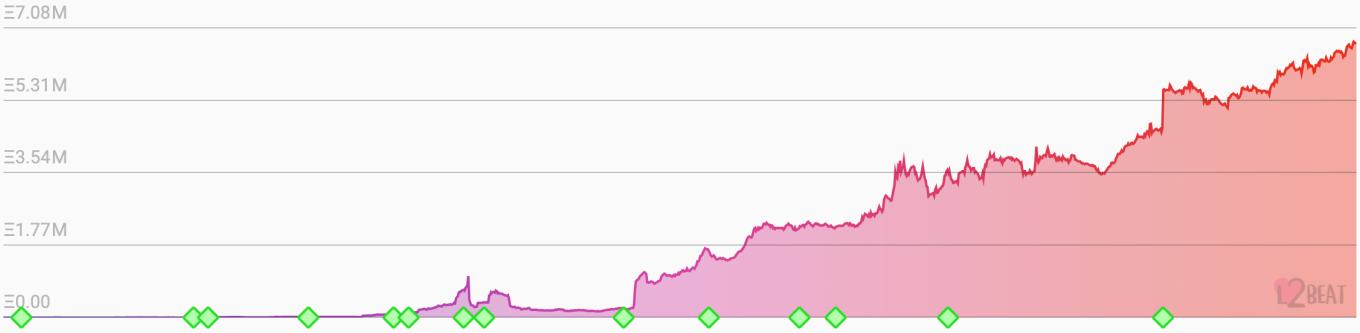
Sum of all funds locked on Ethereum converted to USD

$10.37B

▼ 3.97% / 7 days

2019 Nov 15 – 2023 Oct 14

7D 30D 90D 180D 1Y MAX



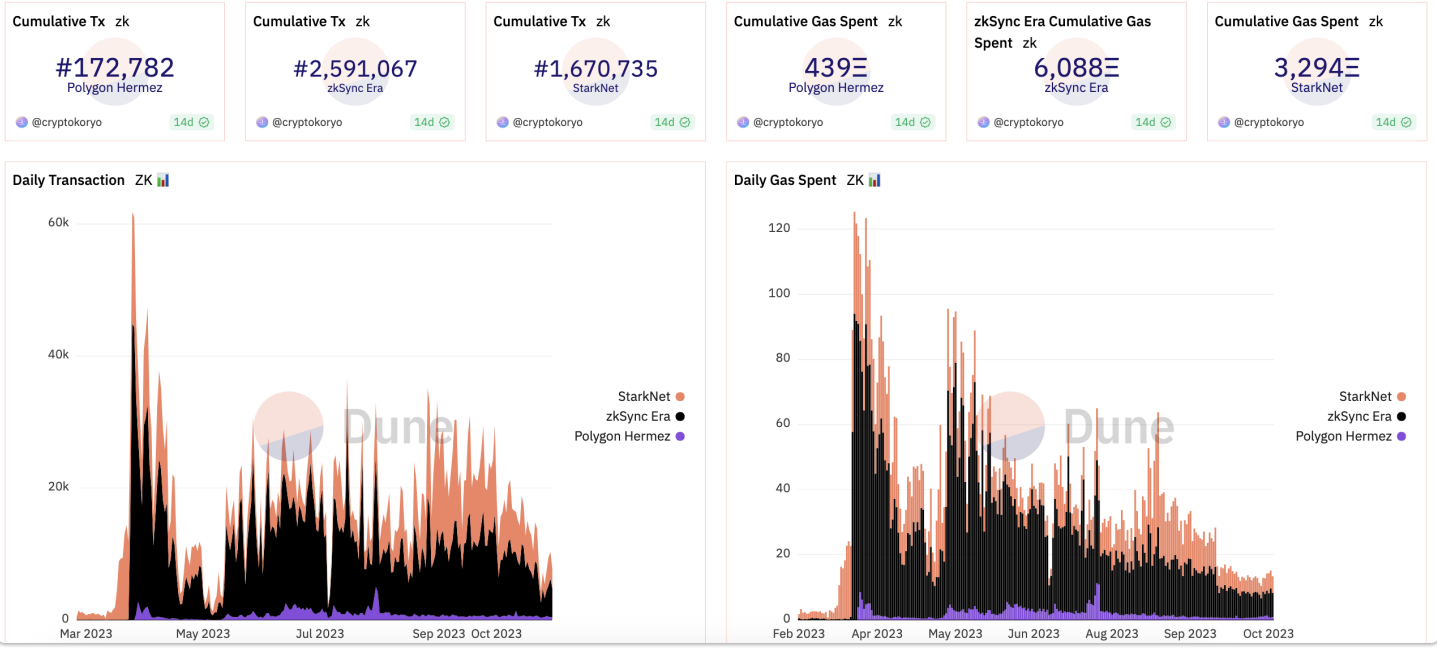
USD ETH

LOG LIN

#	NAME	RISKS	TECHNOLOGY	STAGE	PURPOSE	TOTAL	MKT SHARE
1	Arbitrum One		Optimistic Rollup	STAGE 1	Universal	$5.70B ▼ 3.33%	54.94%
2	OP Mainnet		Optimistic Rollup	STAGE 0	Universal	$2.57B ▼ 6.09%	24.79%
3	Base		Optimistic Rollup	STAGE 0	Universal	$546M ▼ 2.13%	5.26%
4	zkSync Era		ZK Rollup	STAGE 0	Universal	$399M ▼ 10.60%	3.85%
5	dYdX		ZK Rollup	STAGE 1	Exchange	$325M ▼ 5.42%	3.14%
6	Starknet		ZK Rollup	STAGE 0	Universal	$131M ▼ 7.41%	1.26%
7	Loopring		ZK Rollup	STAGE 0	Tokens, NFTs, AMM	$81.31M ▼ 4.41%	0.78%
8	zkSync Lite		ZK Rollup	STAGE 1	Payments, Tokens	$66.23M ▼ 9.35%	0.64%
9	Linea		ZK Rollup	STAGE 0	Universal	$66.22M ▼ 3.79%	0.64%
10	Polygon zkEVM		ZK Rollup	STAGE 0	Universal	$52.47M ▲ 3.85%	0.51%

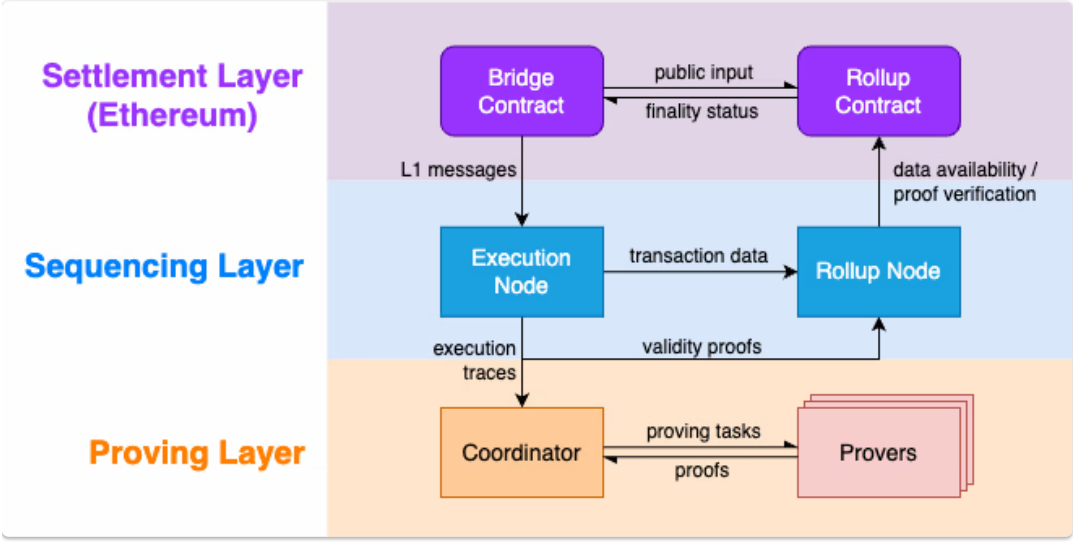
11	ZKSpace	ZK Rollup	STAGE 0	Tokens, NFTs, AMM	$19.56M ▼ 5.75%	0.19%
12	Manta Pacific	Optimistic Rollup	STAGE 0	Universal	$10.24M ▲ 7.78%	0.10%
13	Boba Network	Optimistic Rollup	STAGE 0	Universal	$8.54M ▼ 3.75%	0.08%
14	Aevo	Optimistic Rollup	STAGE 0	DEX	$6.71M ▼ 0.43%	0.06%
15	Zora	Optimistic Rollup	STAGE 0	Universal, NFTs	$6.25M ▼ 8.30%	0.06%
16	Aztec Connect	ZK Rollup	STAGE 0	Private DeFi	$4.32M ▼ 5.77%	0.04%
17	DeGate V1	ZK Rollup	STAGE 2	Exchange	$4.04M ▲ 31.15%	0.04%
18	Aztec	ZK Rollup	STAGE 0	Private payments	$2.03M ▼ 5.29%	0.02%
19	Scroll	ZK Rollup	IN REVIEW	Universal	$624K ▲ 0.00%	0.01%
20	Public Goods Network	Optimistic Rollup	STAGE 0	Universal	$537K ▼ 3.08%	0.01%
21	Kroma	Optimistic Rollup	STAGE 0	Universal	$266K ▼ 4.36%	0.00%
22	Honeypot (Cartesi)	Optimistic Rollup	STAGE 0	Bug bounty	$5.30K ▲ 10.49%	0.00%
23	Fuel v1	Optimistic Rollup	STAGE 2	Payments	$406 ▼ 5.38%	0.00%

See Dune dashboard



Scroll Overview

See Docs



Settlement Layer

This layer ensures data availability and ordering for the primary Scroll chain. It also validates validity proofs and enables users and decentralized applications (dapps) to exchange messages and assets between Ethereum and Scroll. Ethereum serves as the Settlement Layer, and we deploy the bridge and rollup contracts onto the Ethereum platform.

Sequencing Layer

This layer comprises an Execution Node, responsible for executing transactions submitted to the Scroll sequencer and those submitted to the L1 bridge contract. It produces L2 blocks. Additionally, it includes a Rollup Node that batches transactions, publishes transaction data and block information to Ethereum for data availability, and submits validity proofs to Ethereum for finality.

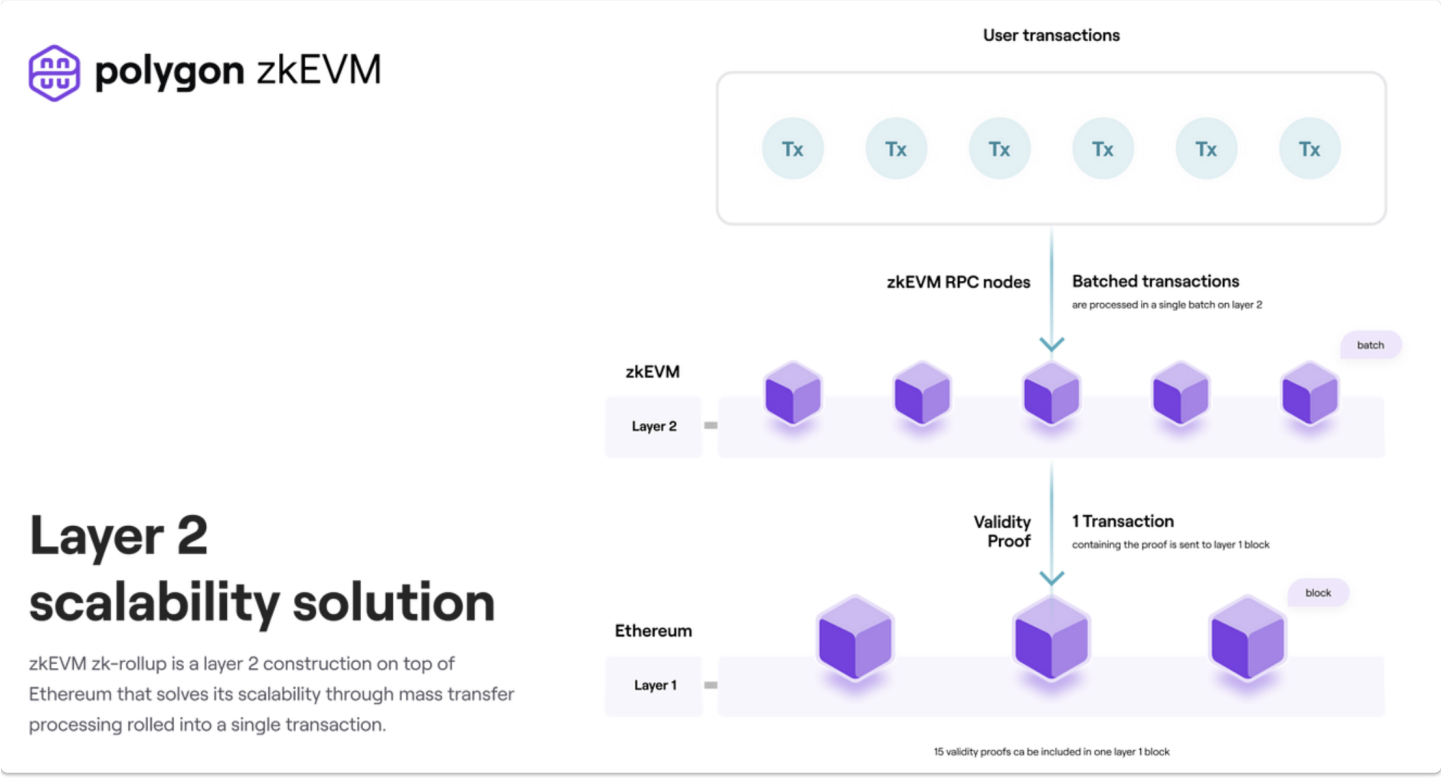
Proving Layer

This layer consists of a pool of provers with the task of generating zkEVM validity proofs that verify the correctness of L2 transactions. A coordinator manages the dispatch of proving tasks to the provers and relays the resulting proofs to the Rollup Node for finalization on the Ethereum network.

Polygon zkEVM Overview

See Docs

Useful article

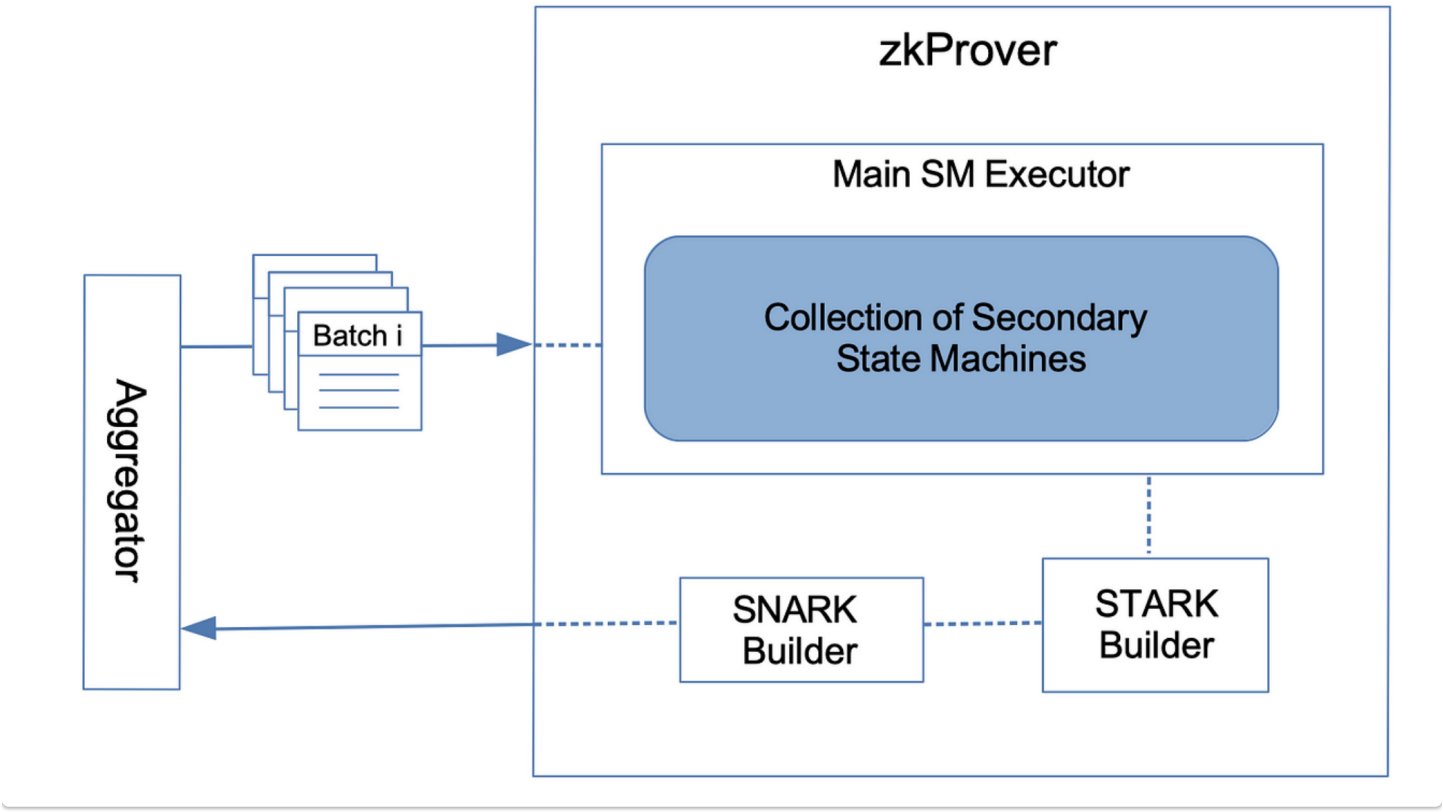


From the docs

"The Trusted Sequencer reads transactions from the pool and decides whether to discard them or order and execute them. Transactions that have been executed are added to a transaction batch, and the Sequencer's local L2 State is updated.

Once a transaction is added to the L2 State, it is broadcast to all other zkEVM nodes via a broadcast service. It is worth noting that by relying on the Trusted Sequencer, we can achieve fast transaction finality (faster than in L1). However, the resulting L2 State will be in a trusted state until the batch is committed in the Consensus Contract."

Proof process


zkProver
Main SM Executor
Collection of Secondary State Machines
Batch i
Aggregator
SNARK Builder
STARK Builder

zkSync introduction

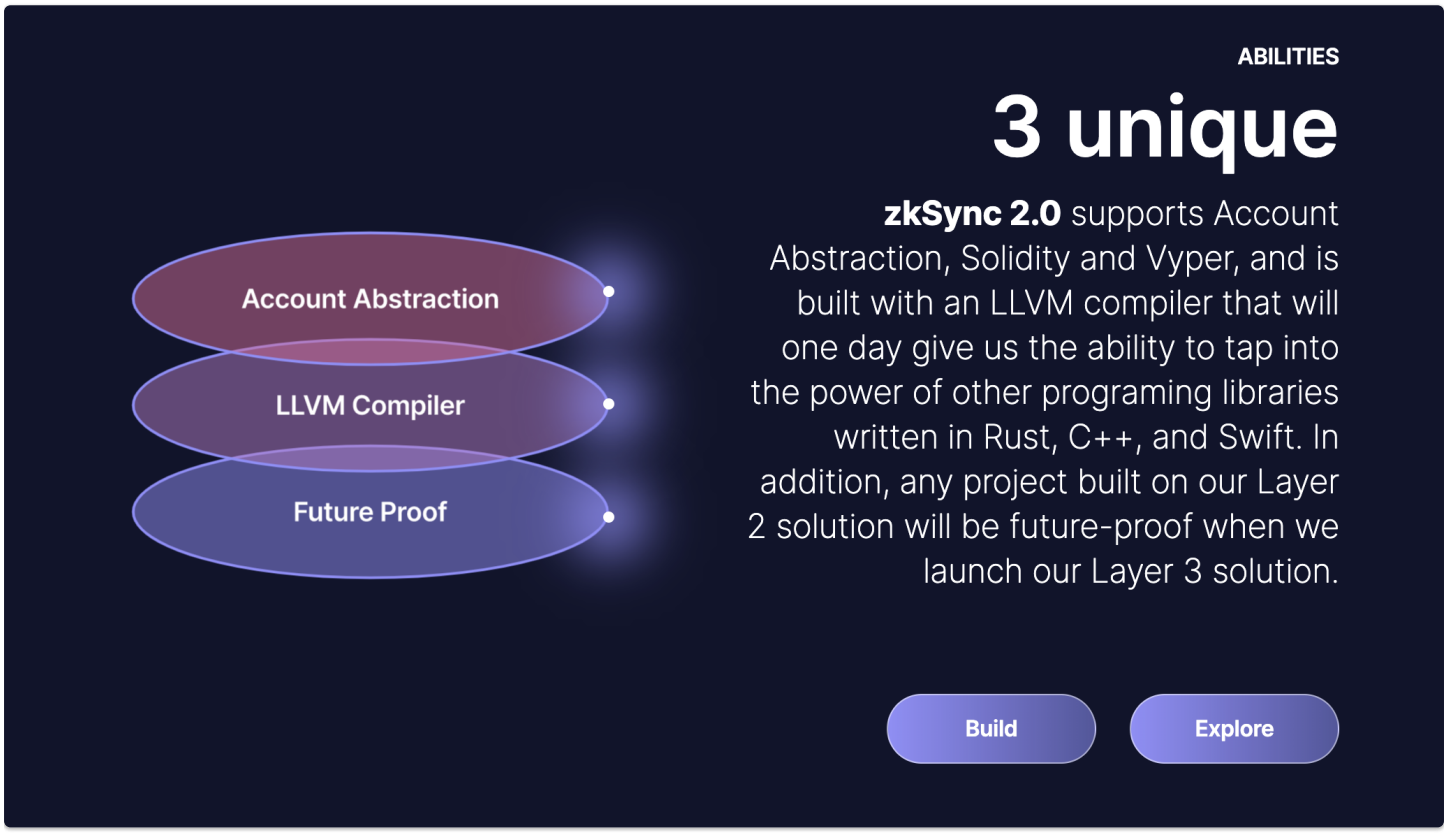
Resources

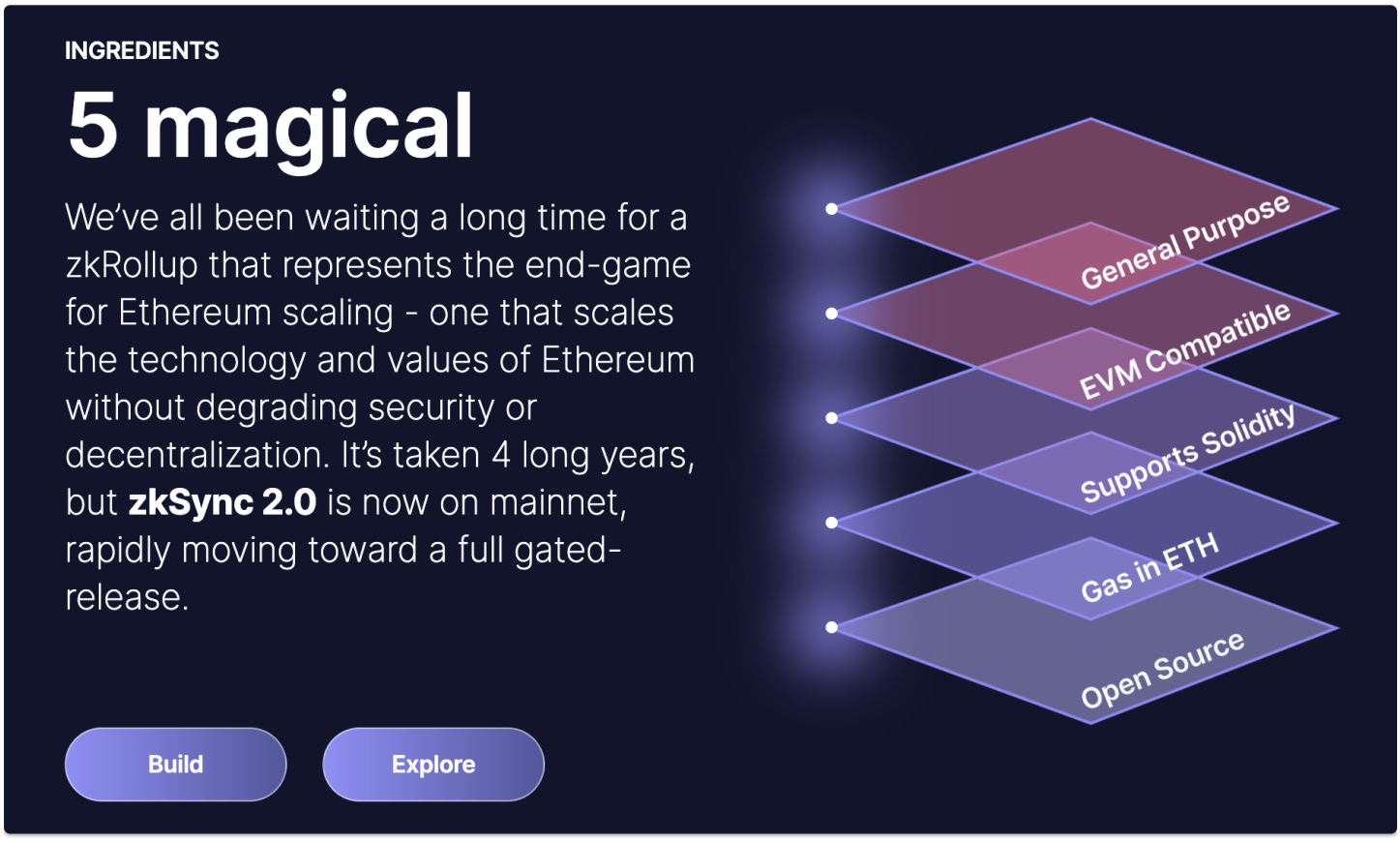
[Web]

[Twitter]

[Medium]

[Docs]

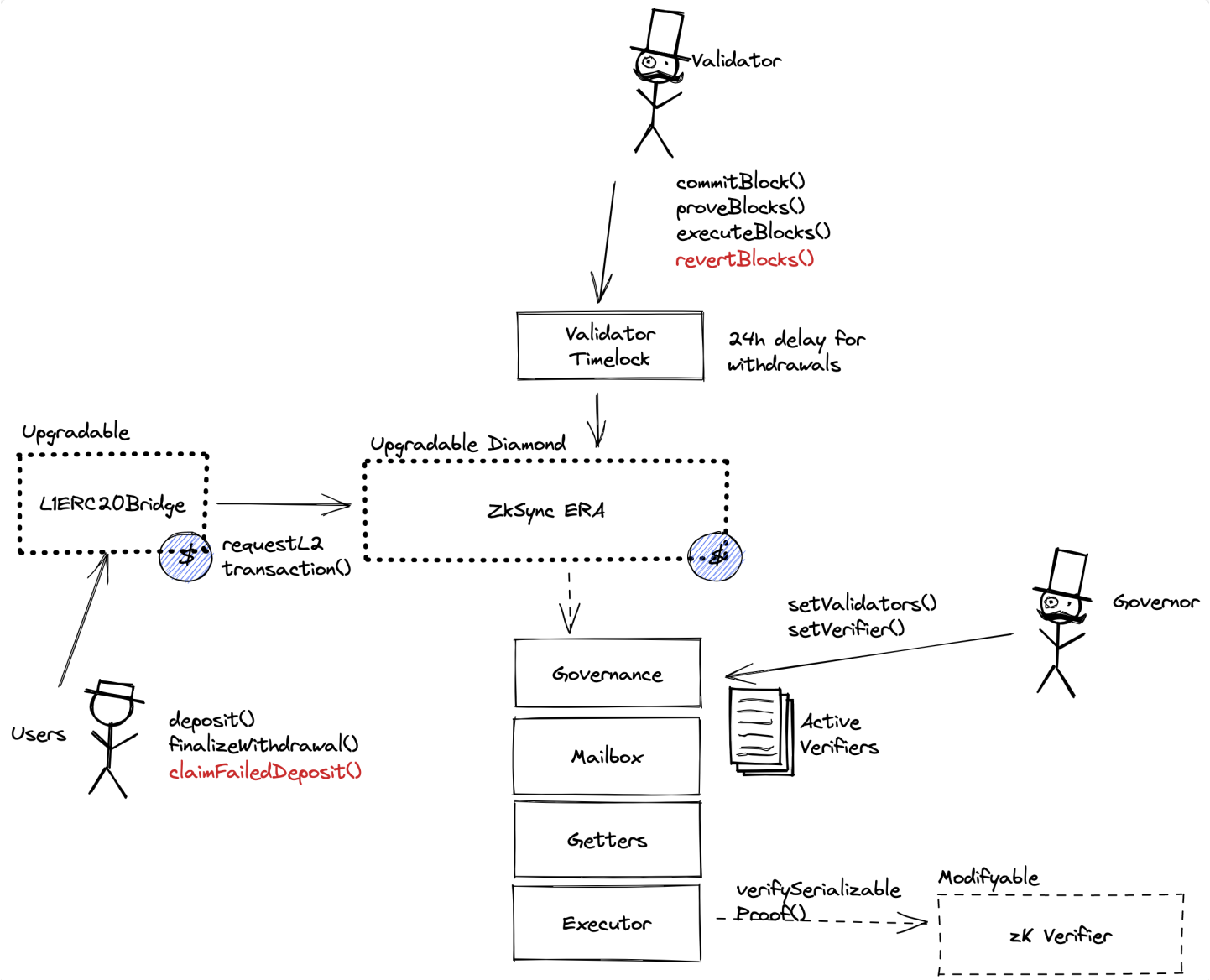


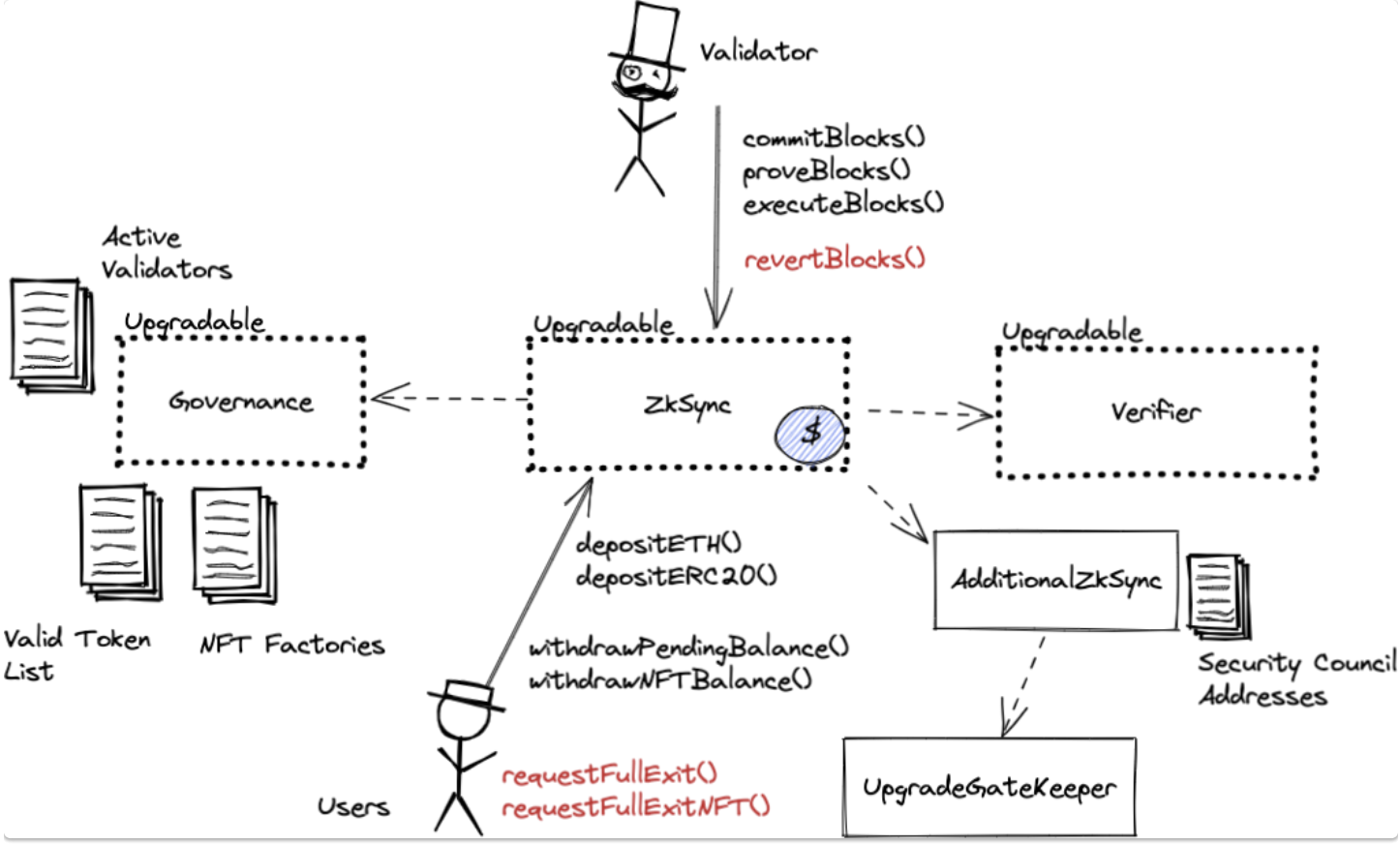


zkSync Architecture

From zkSync Docs :
A layer 2 relies on the L1 for security guarantees :

- The Rollup validator(s) can never corrupt the state or steal funds (unlike Sidechains).
- Users can always retrieve the funds from the Rollup even if validator(s) stop cooperating because the data is available (unlike Plasma).
- Thanks to validity proofs, neither users nor a single other trusted party needs to be online to monitor Rollup blocks in order to prevent fraud (unlike payment channels or Optimistic Rollups).




Validator
commitBlocks()
proveBlocks()
executeBlocks()
revertBlocks()
Active Validators
Upgradable
Governance
Upgradable
ZkSync
$
Upgradable
Verifier
depositETH()
depositERC20()
withdrawPendingBalance()
withdrawNFTBalance()
AdditionalZkSync
Security Council Addresses
Valid Token List
NFT Factories
Users
requestFullExit()
requestFullExitNFT()
UpgradeGateKeeper

Starknet overview

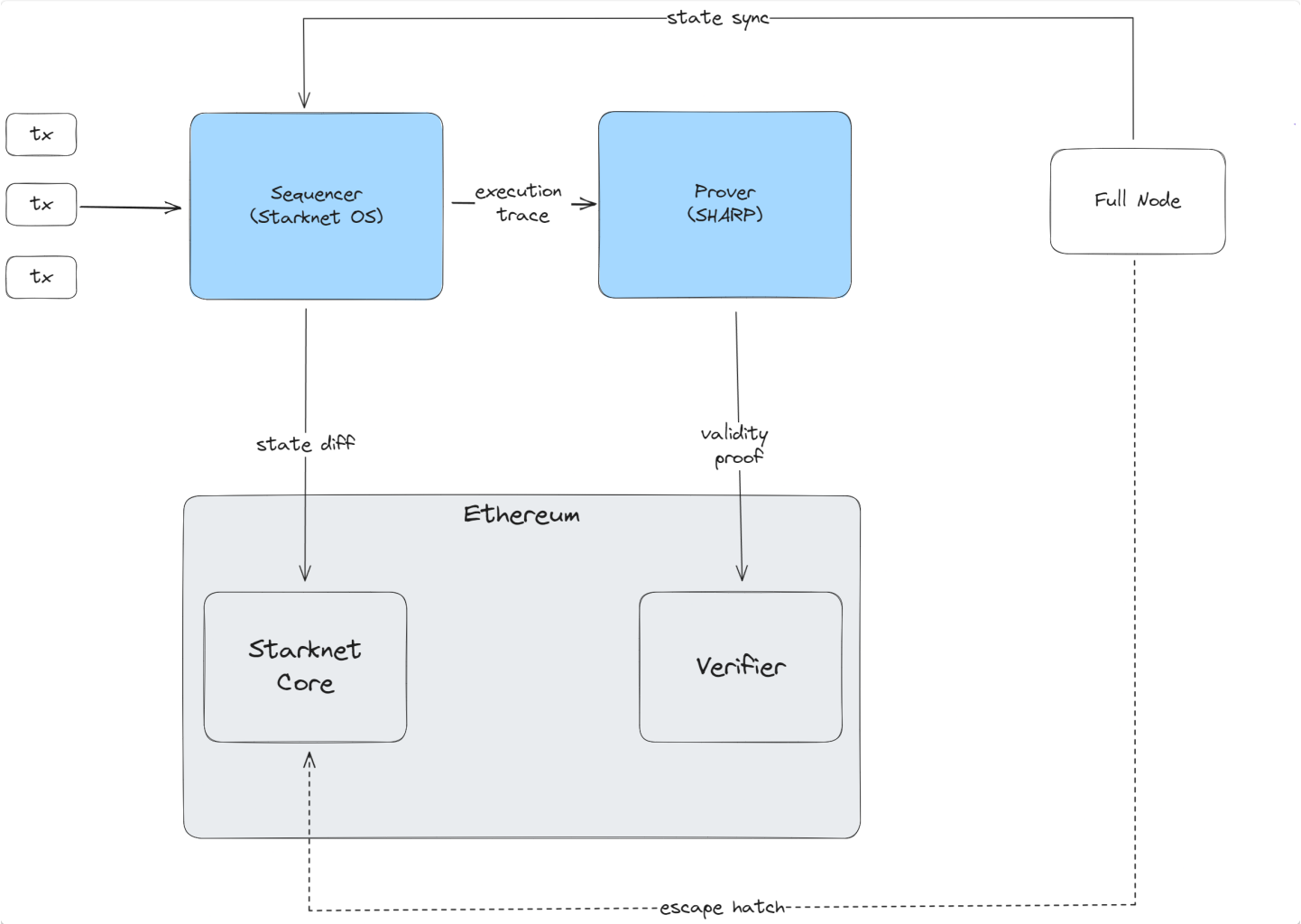
See Mindmap

See Starknet documentation

"Starknet is a permissionless decentralised ZK-Rollup. It operates as an L2 network over Ethereum, enabling any dApp to achieve unlimited scale for its computation – without compromising Ethereum's composability and security."

See ecosystem

Starknet architecture overview

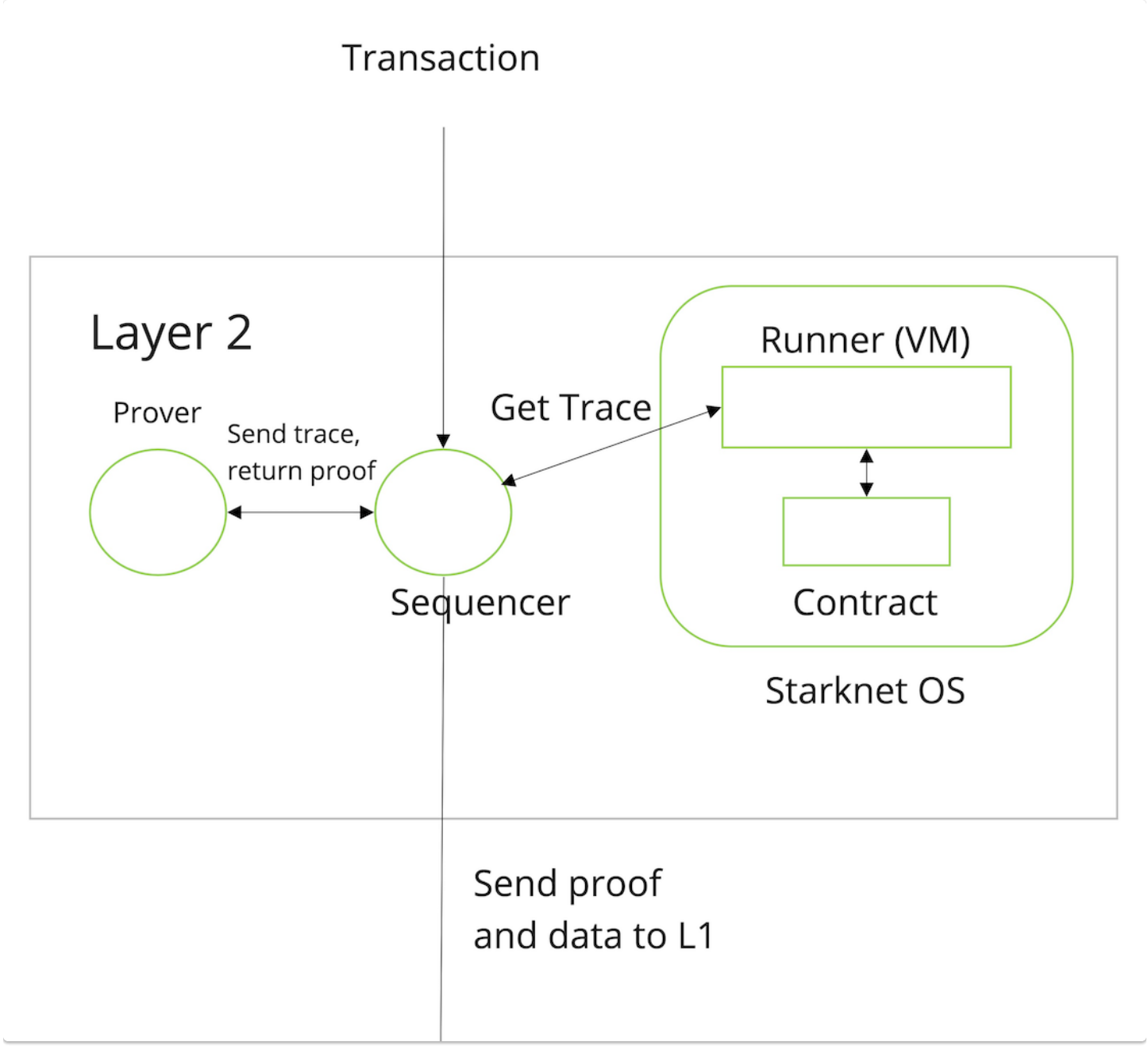


See this article for a good overview

Starknet Components

1. Prover: A separate process (either an online service or internal to the node) that receives the output of Cairo programs and generates STARK proofs to be verified. The Prover submits the STARK proof to the verifier that registers the fact on L1.
2. StarkNet OS: Updates the L2 state of the system based on transactions that are received as inputs. Effectively facilitates the execution of the (Cairo-based) StarkNet contracts. The OS is Cairo-based and is essentially the program whose output is proven and verified using the STARK-proof system. Specific system operations and functionality available for StarkNet contracts are available as calls made to the OS.
3. StarkNet State: The state is composed of contracts' code and contracts' storage.

4. StarkNet L1 Core Contract: This L1 contract defines the state of the system by storing the commitment to the L2 state. The contract also stores the StarkNet OS program hash – effectively defining the version of StarkNet the network is running.
   The committed state on the L1 core contract acts as provides as the consensus mechanism of StarkNet, i.e., the system is secured by the L1 Ethereum consensus. In addition to maintaining the state, the StarkNet L1 Core Contract is the main hub of operations for StarkNet on L1.
   Specifically:
   - It stores the list of allowed verifiers (contracts) that can verify state update transactions
   - It facilitates L1 ↔ L2 interaction
5. Starknet Full Nodes: Can get the current state of the network from the sequencer. If the connection between the Sequencer and the Full Node fails for some reason, you can recreate the L2 current state by indexing date from the Starknet L1 Core Contract independently



For details of Starknet blocks, see Block structure

Risc Zero

Introduction

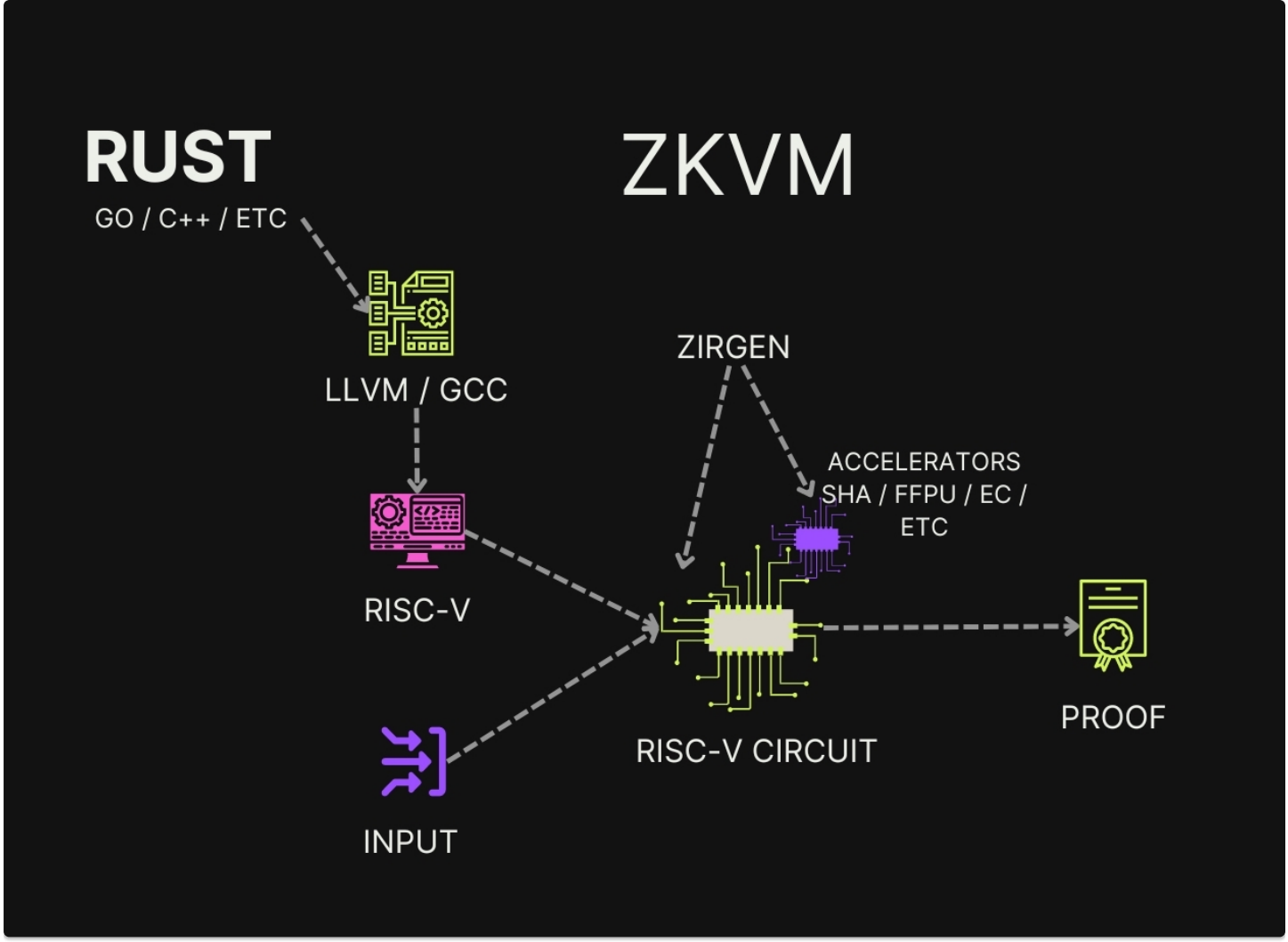
The RISC Zero zkVM is an open-source, zero-knowledge virtual machine designed for constructing trustless, verifiable software applications.

Risc Zero's goal is to integrate existing programming languages and developer tools into the zero-knowledge realm. This is accomplished through a high-performance ZKP prover, which provides the performance allowance necessary to create a zero-knowledge virtual machine (zkVM) implementing a standard RISC-V instruction set.

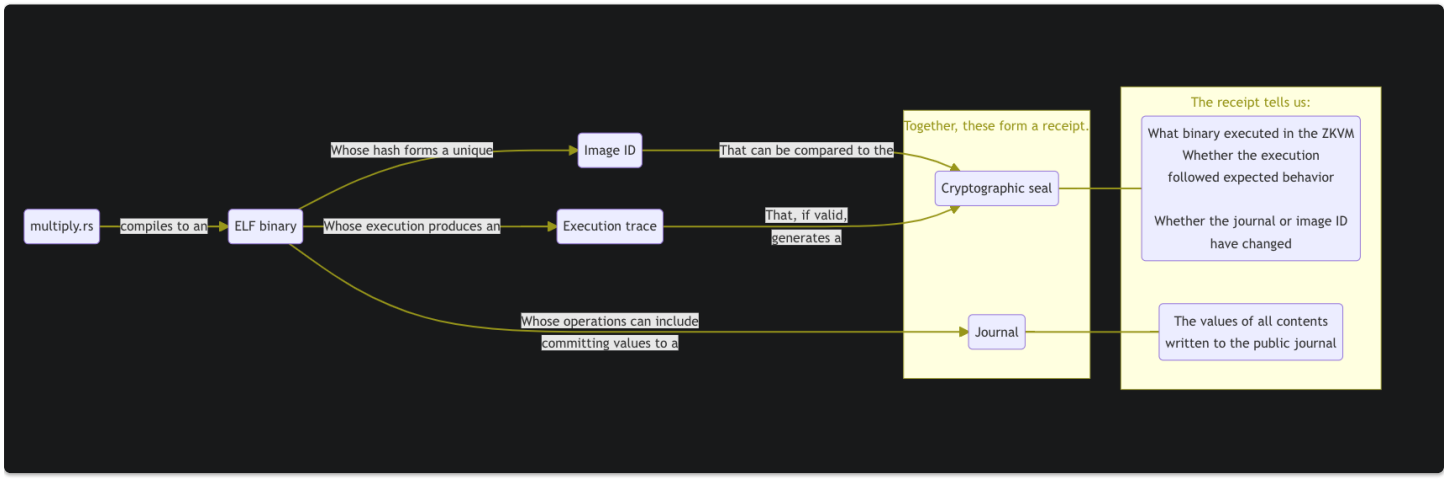
In practical terms, this allows for smooth integration between "host" application code, written in high-level languages running natively on host processors (e.g., Rust on arm64 Mac), and "guest" code in the same language executing within the zkVM.

A zero-knowledge virtual machine is a virtual machine that runs trusted code and generates proofs that authenticate the zkVM output.
RISC Zero's zkVM implementation, based on the RISC-V architecture, executes code and produces a computational receipt.

Risc-V is an open standard instruction set architecture (ISA) based on established reduced instruction set computer (RISC) principles


RUST
GO / C++ / ETC
ZKVM
LLVM / GCC
ZIRGEN
ACCELERATORS
SHA / FFPU / EC /
ETC
RISC-V
PROOF
RISC-V CIRCUIT
INPUT

In a RISC Zero zkVM program, guest code written for the zkVM is compiled to an ELF binary and executed by the `prover`, which returns a computational receipt to the host program
Anyone possessing a copy of this receipt can verify the program's execution and access its publicly shared outputs.



Before being executed on the zkVM, guest source code is converted into a RISC-V ELF binary. The binary file is hashed to create a `image ID` that uniquely identifies the binary being executed. The binary may include code instructions to publicly commit a value to the `journal`. Later, the journal contents can be read by anyone with the receipt.

After the binary is executed, an execution trace contains a complete record of zkVM operation. The trace is inspected and the ELF file's instructions are compared to the operations that were actually performed. A valid trace means that the ELF file was faithfully executed according to the rules of the RISC-V instruction set architecture.

The execution trace and the journal are then used to generate a seal, a blob of cryptographic data that shows the receipt is valid.
The seal has properties that reveal whether itself or the journal have been altered.
When the receipt is verified, the seal will be checked to confirm the validity of the receipt.
To check whether the correct binary was executed, the seal can be compared to the image ID of the expected ELF file.

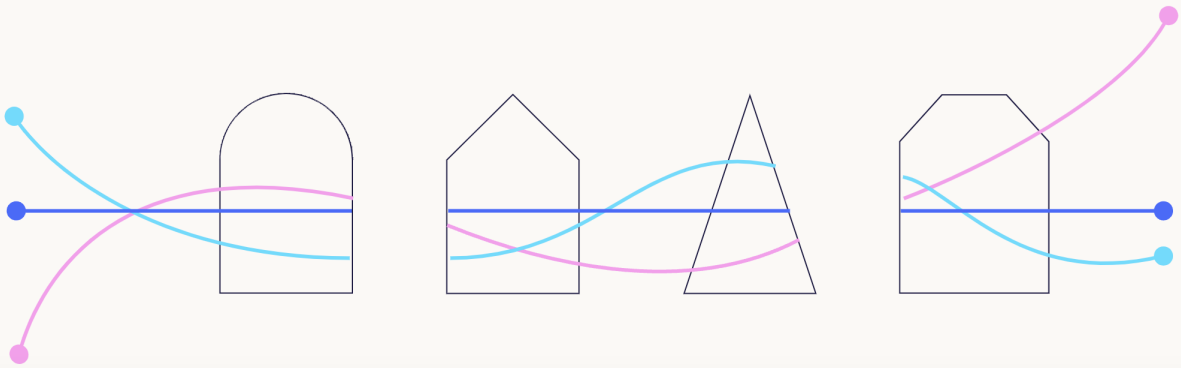
Powdr

Powdr brings modularity, flexibility, security and excellent developer experience to zkVMs.

How it works

Design a new zkVM in hours, through a user-defined ISA, which powdr compiles into a zkVM.

Generate proofs using eSTARK, Halo2, Nova, and whatever comes next.



See Docs

powdr is a modular compiler stack to build zkVMs. It is ideal for

implementing existing VMs and experimenting with new designs with minimal boilerplate.

- Domain specific languages are used to specify the VM and its underlying constraints, not low level Rust code
- Automated witness generation
- Support for multiple provers as well as aggregation schemes
- Support for hand-optimized co-processors when performance is critical
- Built in Rust 🦀

See video from Christian Reitwiessner

Installation

Rust is a prerequisite
Instructions are here

Front Ends

A Risc V frontend is available and others are under development.

Backends

Halo 2 and eSTARK are supported

Polygon Miden overview

Polygon Miden is a general-purpose, STARK-based ZK rollup with EVM compatibility.

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."

Comparing different proving systems

	SNARKs	STARKs	Bulletproofs
Algorithmic complexity: prover	O(N * log(N))	O(N * poly-log(N))	O(N * log(N))
Algorithmic complexity: verifier	~O(1)	O(poly-log(N))	O(N)
Communication complexity (proof size)	~O(1)	O(poly-log(N))	O(log(N))
- size estimate for 1 TX	Tx: 200 bytes, Key: 50 MB	45 kB	1.5 kb
- size estimate for 10.000 TX	Tx: 200 bytes, Key: 500 GB	135 kb	2.5 kb
Ethereum/EVM verification gas cost	~600k (Groth16)	~2.5M (estimate, no impl.)	N/A
Trusted setup required?	YES 😒	NO 😄	NO 😄
Post-quantum secure	NO 😒	YES 😄	NO 😒
Crypto assumptions	Strong 😒	Collision resistant hashes 😄	Discrete log 😏

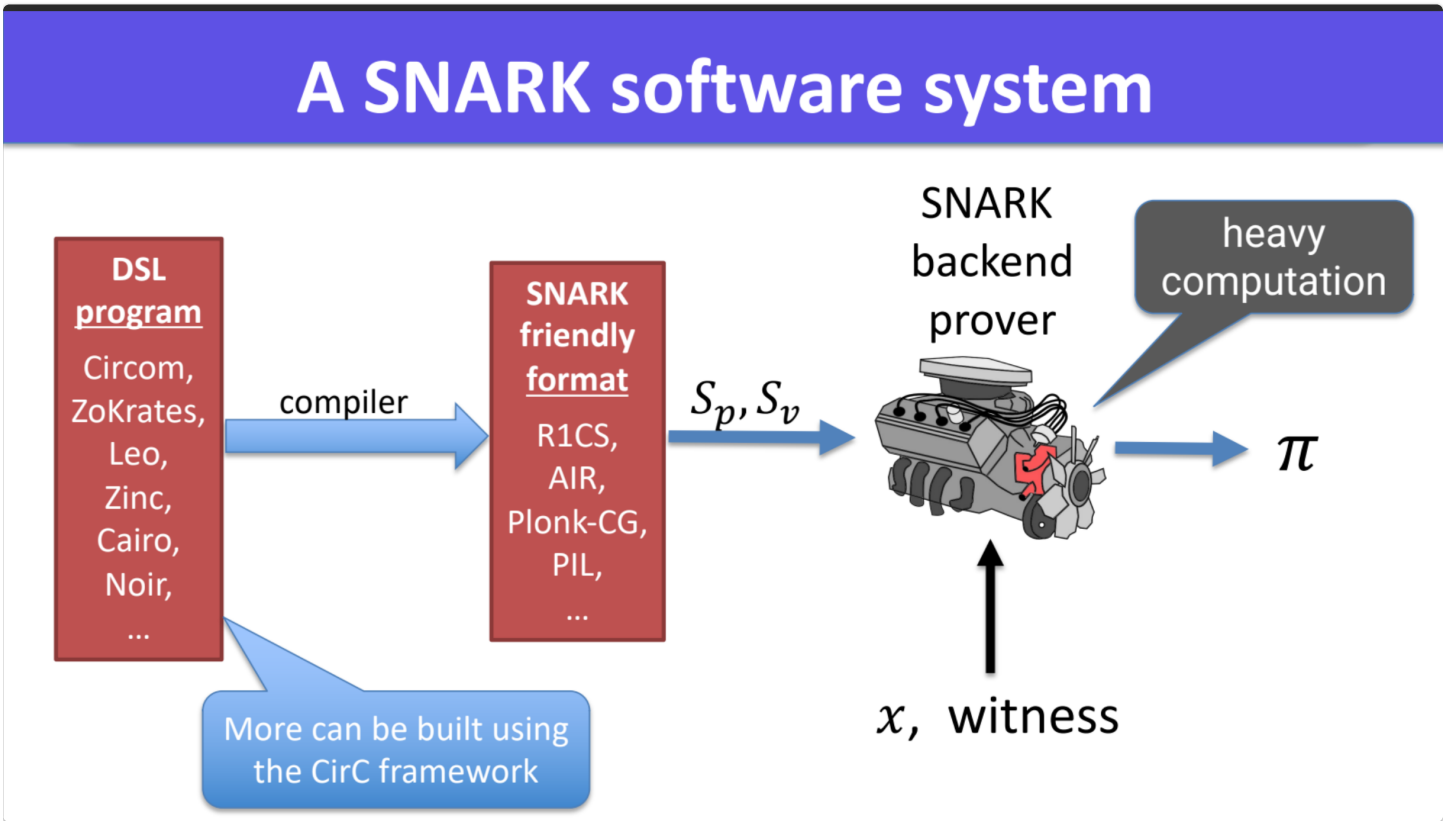
Taxonomy from Celer

Framework	Arithmetization	Commitment Scheme	Field	Other Configs
Circom + snarkjs / rapidsnark	**R1CS**	**Groth16**	**BN254 scalar**	
gnark	**R1CS**	**Groth16**	**BN254 scalar**	
Arkworks	**R1CS**	**Groth16**	**BN254 scalar**	
Halo2 (KZG)	**Plonkish**	**KZG**	**BN254 scalar**	
Plonky2	**Plonk**	**FRI**	**Goldilocks**	**blowup factor = 8 proof of work bits = 16 query rounds = 28 num_of_wires = 60 num_routed_wires = 60**
Starky	**AIR**	**FRI**	**Goldilocks**	**blowup factor = 2 proof of work bits = 10 query rounds = 90**
Boojum	**Plonk**	**FRI**	**Goldilocks**	

Video analysing EVM Design

Arithmetic circuit introduction

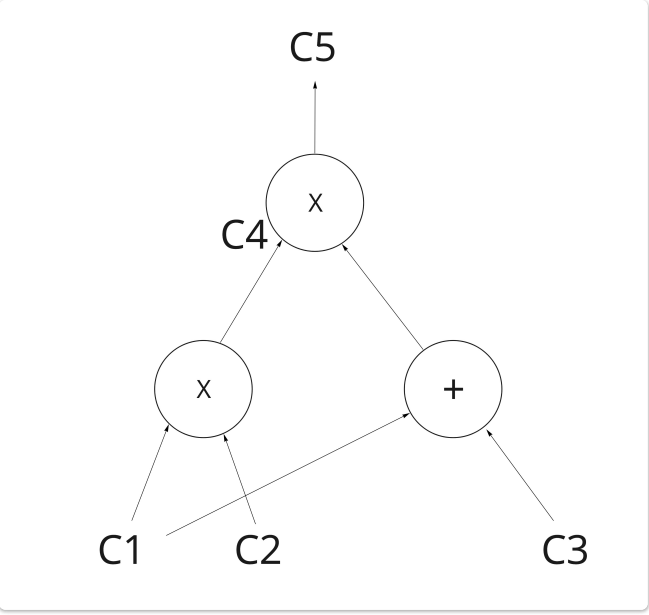
Slide from Stanford Web3 MOOC



An arithmetic circuit is one way to represent our DSL.
Arithmetic circuits are useful as they represented as a rank-1 constraint system, which can be transformed into polynomials, which is the next stage in the creation of the proof.

General form of an arithmetic circuit

We can visualise our circuit as a number of addition and multiplication gates.
These simple elements can be combined to represent any polynomial.



It is important to realise that this is not an algorithm, it is more like a hardware circuit. We can think of the circuit as having constraints, which will be satisfied if the correct inputs to the gates are supplied, i.e. we have the correct witness.

When we implement a circuit we use large tables, where a gate is represented by a row in the table.
When we are considering the interaction between the rows, this may be limited to nearby rows, but this varies with different aithmetisations.