Lesson 16

Mina updates
https://minaprotocol.com/blog/ecosystem-updates-march-2022

Zktech funding round
https://medium.com/zero-knowledge-validator/zk-tech-gitcoin-gr13-side-round-live-on-9th-march-7e90f44ae715

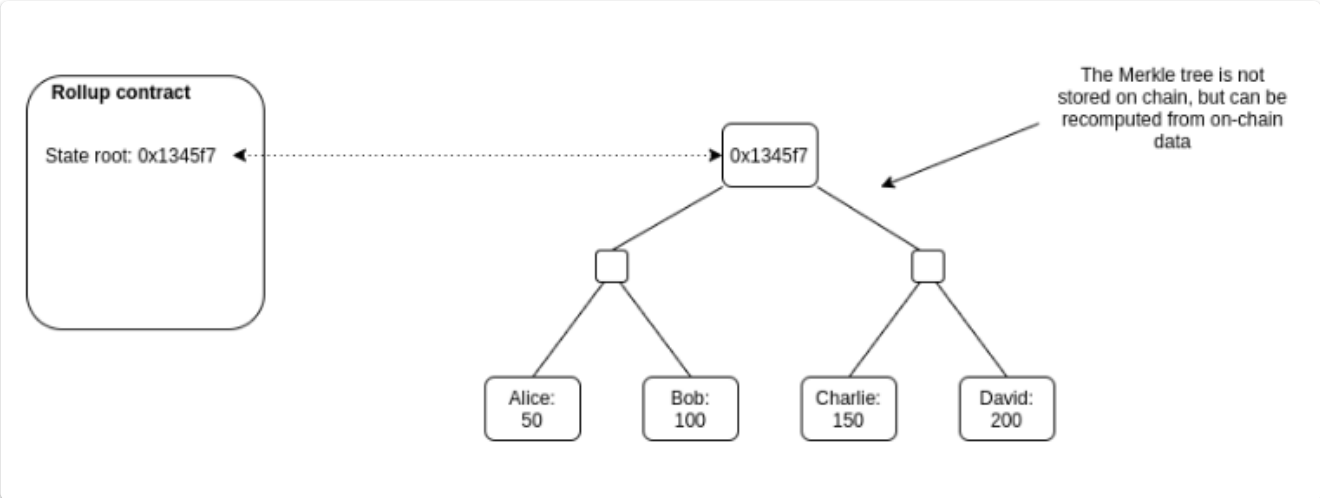
Chainsafe is currently working on a Minimum Viable Product (MVP) demo of a Mina node that can run in a browser. For details on the current development architecture, you can see the overview of the node here. As the next iteration of this MVP, we're working on a browser extension with a web Graphical User Interface (GUI), with a small set of functionalities including displaying node status/ chain information in the Mina network.

Rollups recap

Zkproofs have been used very successfuly to address scalability on L1 by providing a rollup mechanism on a L2 chain.
The role of the proof is typically a proof of computation, to show that a set of transactions have been correctly applied to give a certain state transition. In some L2 projects such as Aztec and zkopru zkps are also used to give privacy.

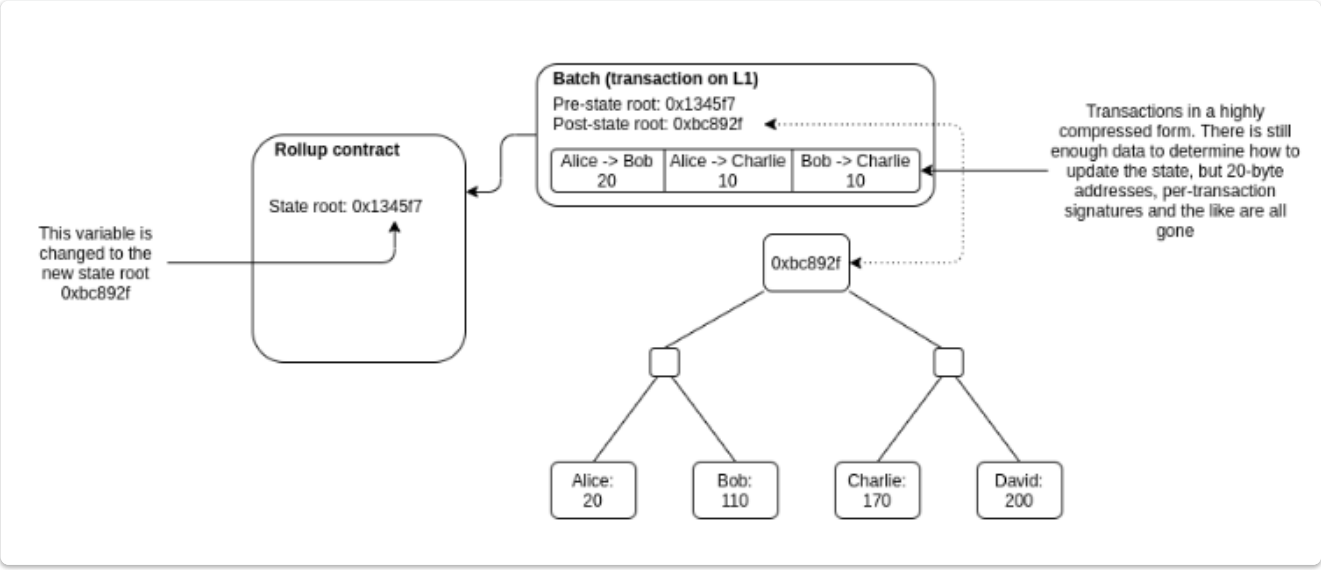
The main chain L1 holds funds and commitments to the side chain L2
The L2 holds additional state and performs execution
There needs to be some proof, either a fraud proof (Optimistic) or a validity proof (zk)

Rollups require "operators" to stake a bond in the rollup contract. This incentivises operators to verify and execute transactions correctly.



L2s can be permissioned, or allow anyone to be an operator and publish a collection of transactions.
These are in a highly compressed form together with the previous state root and the new state root (the Merkle root after processing the transactions). The L1 contract (verifier) checks that

the previous state root in the batch matches its current state root; if it does, it switches the state root to the new state root.



Systems will often use a mixture of commitments and proofs, usually using composition to provide an overall proof.

It was simpler initially to create rollup projects where the transaction and proof were simple, such as a native token transfer, more general solutions that need to prove the correctness of a smart contract have been achieved with projects such as Cairo / Warp and the various zkEVM solutions.

Ethereum sees their mid term future as being roll up centric, and the scalability benfits that they bring have reduced the urgency to introduce sharding

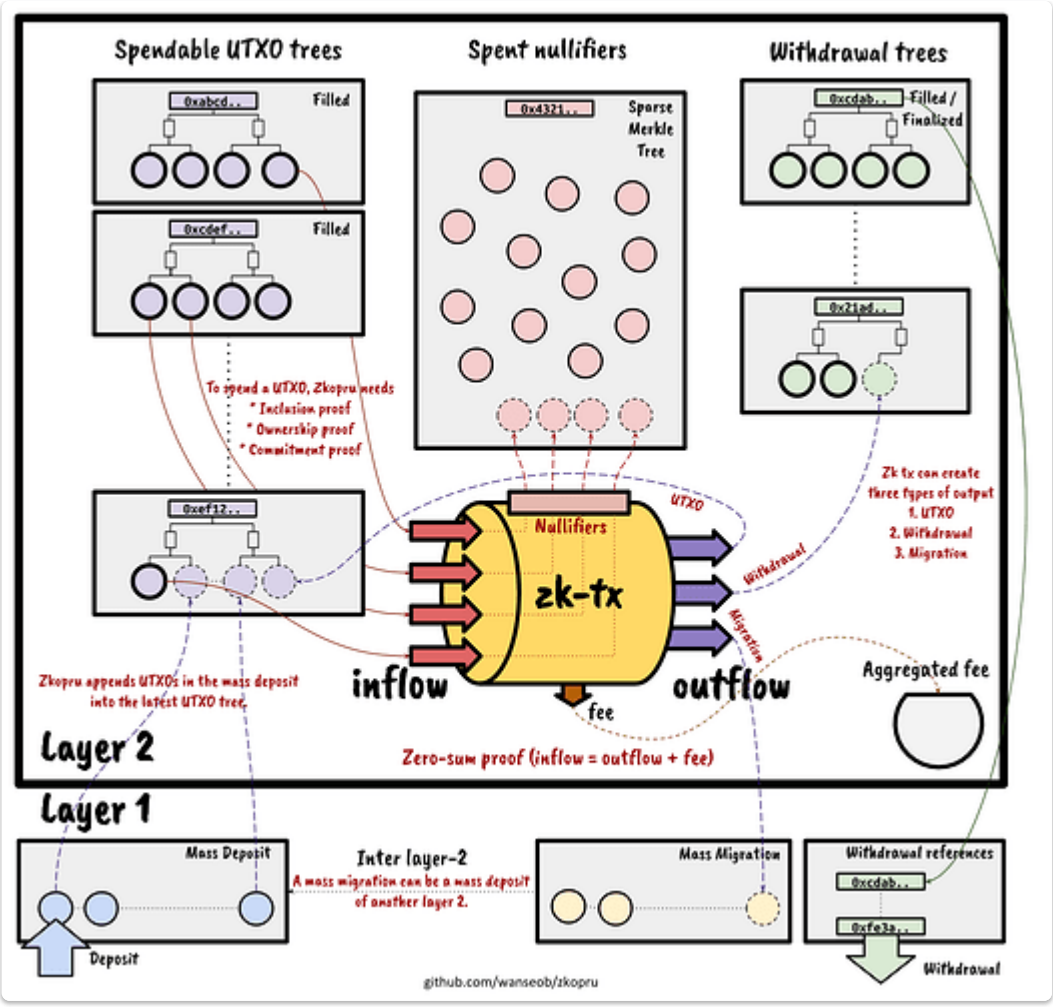
SECURITY

Recent article about fraud proofs and possible attack vectors.

zkopru

See docs
and docs2

zkopru is a L2 with a combination of optimistic rollups, and zk proofs for privacy.
It uses a similar model to Zcash and Aztec in its use of a UTXO model, notes and nullifiers.



The proofs used are

- UTXO membership proof

The tx builder submits Merkle proofs of every UTXO to prove its existence. For an effective SNARK computation, the UTXO tree uses Poseidon for its hash function.

- Ownership proof

Only the owner should be possible to spend the UTXO. For this condition, every note has a public key field, a Babyjubjub point. Using the paired private key, the owner can create an EdDSA signature to prove its ownership.

- Commitment proof

The circuit should have detailed information about the input UTXOs to calculate the total sum of the inflow. Therefore, the owner should provide the details, and its Poseidon hash should equal the leaf hash of the Merkle proof and the ownership proof.

- Nullifier proof

The given nullifiers should be correctly derived from the input UTXOs.

- Zero-sum proof
  Finally, the zk transaction should guarantee that the inflow equals the outflow, including the fee.

To use zkopru you need to run a node, there is a simple [setup] (https://github.com/zkopru-network/docs/blob/burrito/getting-started/coordinator.md), and a docker image is also available.

Block producers

They use a proof of burn method to select the block producer
"Zkopru uses a burn auction to determine who is allowed to propose blocks in the network. Coordinators must spend their Ether in exchange for the right to propose blocks and collect fees. Auction Ether is collected by Zkopru with the intent of spending on public good services."

Trusted sestup

They completed a trusted setup using powers of tau, you can see the details here

Hop protocol

Designed to allow bridges beween L2s
This works by leveraging Bonders, who earn a small fee by fronting the liquidity at the destination chain.
At a very high level, a user sends tokens to the Bonder on the source chain and the Bonder sends tokens to the user on the destination chain.

The complicated part of the protocol is to ensure that the user does not have to trust the Bonder. The protocol wants to prevent the Bonder from receiving tokens on the source chain but never sending something on the destination chain.

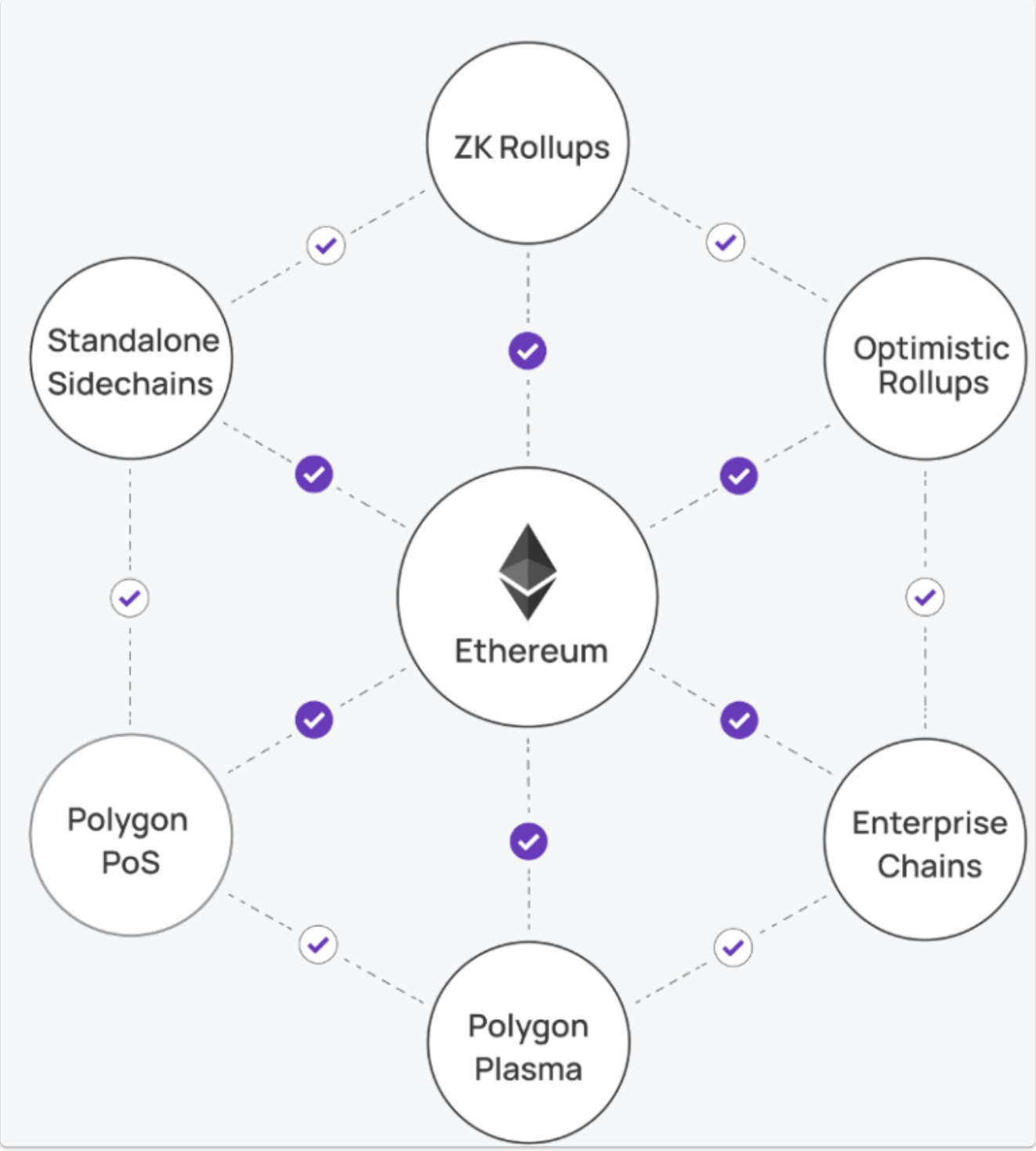
Polygon projects

Nightfall 3

This is a collaboration with EY, continuing the development of Nightfall.
See article
Nightfall 3, combines zero-knowledge proofs with optimistic rollups. The combined protocol is known as a ZK-Optimistic Rollup.
The system assumes the transactions to be valid unless proven otherwise and eliminates the process of having all participants verify all transactions.

Polygon SDK

The Polygon SDK vision is to effectively transform Ethereum into a full-fledged multi-chain system



It has as components

- L2 chains
- Standalone chains (interacting with Ethereum via bridges)

See polygon edge documentation

Avail

Avail is a general-purpose, scalable data availability-focused blockchain targeted for standalone chains, sidechains, and off-chain scaling solutions.
Its objectives are

- Enable standalone chains or sidechains with arbitrary execution environments to bootstrap validator security without needing to create and manage their own validator set by guaranteeing transaction data availability
- Layer-2 solutions such as Validiums to offer increased scalability throughput by using Avail as an off-chain data availability layer

Miden

Miden is a STARK-based, EVM-compatible scaling solution.
Miden VM is the first fully open-source STARK-based virtual machine. It supports arbitrary logic and transactions and has one important additional feature - for any program executed on the VM a small cryptographic proof of correctness is automatically generated. This proof can then be used by anyone to verify that the program was executed correctly, without the need for re-executing the program or even knowing what the program was. With Miden VM, it becomes feasible to build a ZK Rollup that can execute any transaction and program, including those currently living on Ethereum.

ZkSync

ZkSync provide a javascript SDK

Example to deposit tokens to zkSync

```
import * as zksync from "zksync-web3";
import { ethers } from "ethers";

const PRIVATE_KEY =
"0xc8acb475bb76a4b8ee36ea4d0e516a755a17fad2e84427d5559b37b544d9ba5a";

const zkSyncProvider = new zksync.Provider("https://zksync2-
testnet.zksync.dev/");
const ethereumProvider = ethers.getDefaultProvider("goerli");
const wallet = new zksync.Wallet(PRIVATE_KEY, zkSyncProvider, ethereumProvider);

const USDC_ADDRESS = "0xd35cceead182dcee0f148ebac9447da2c4d449c4";
const usdcDepositHandle = await wallet.deposit({
  token: USDC_ADDRESS,
  amount: "10000000",
  approveERC20: true,
});
// Note that we wait not only for the L1 transaction to complete but also for it
to be
// processed by zkSync. If we want to wait only for the transaction to be
processed on L1,
// we can use `await usdcDepositHandle.waitL1Commit()`
await usdcDepositHandle.wait();

const ethDepositHandle = await wallet.deposit({
  token: zksync.utils.ETH_ADDRESS,
```

```
  amount: "10000000",
});
// Note that we wait not only for the L1 transaction to complete but also for it
to be
// processed by zkSync. If we want to wait only for the transaction to be
processed on L1,
// we can use `await ethDepositHandle.waitL1Commit()`
await ethDepositHandle.wait();
```

Integration with hardhat dev environment.

They provide 2 plugins

- `@matterlabs/hardhat-zksync-solc` for smart contract compilation.
- `@matterlabs/hardhat-zksync-deploy` for smart contract deployment.

ZKPs and neural nets

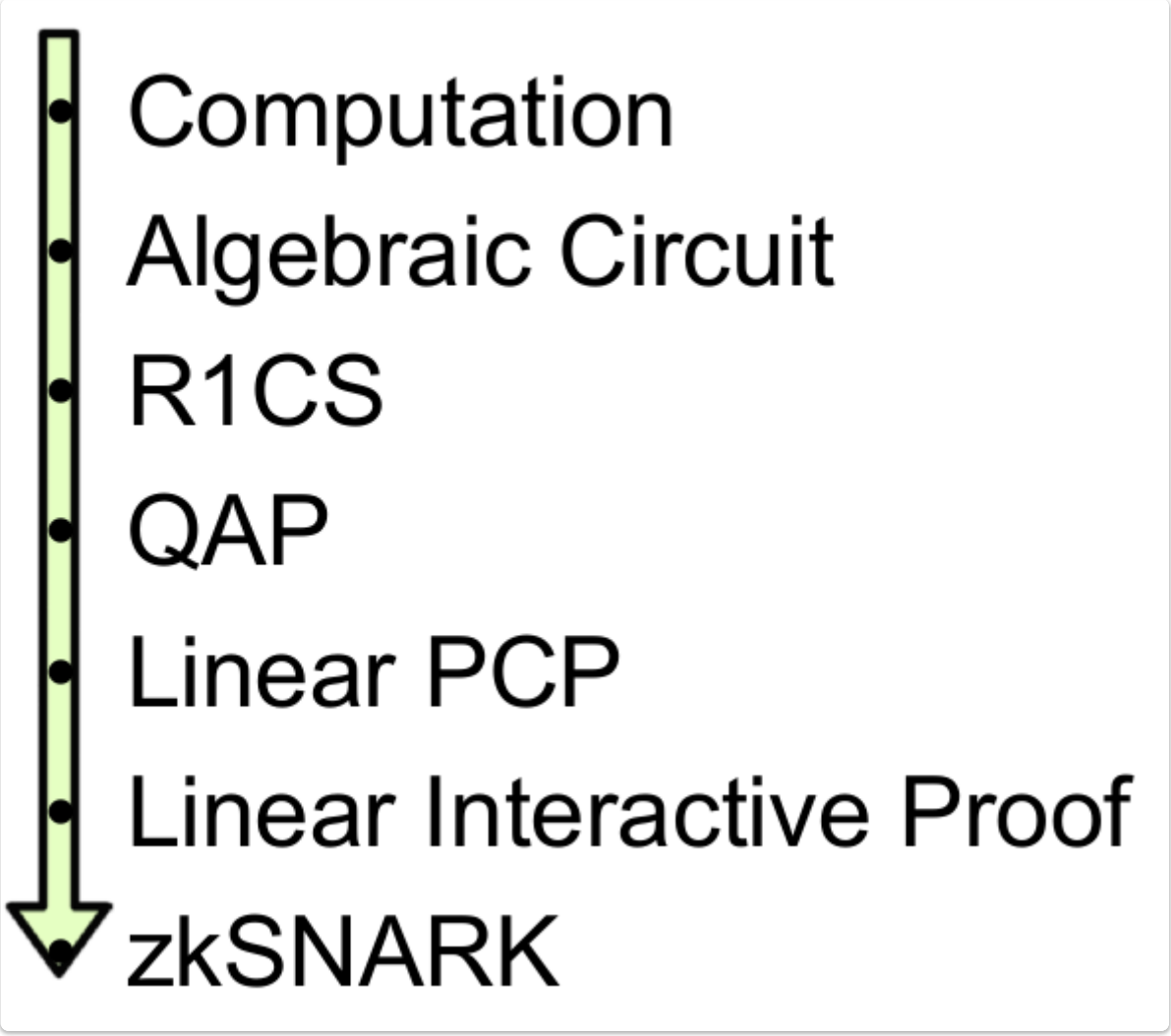
https://github.com/ethereum/research/issues/3

It will include a hash of the weights as a public input, so you can be sure that multiple applications use the same weights. You can also publish the weights and then verify the hash, but that does not have to be done on the blockchain.

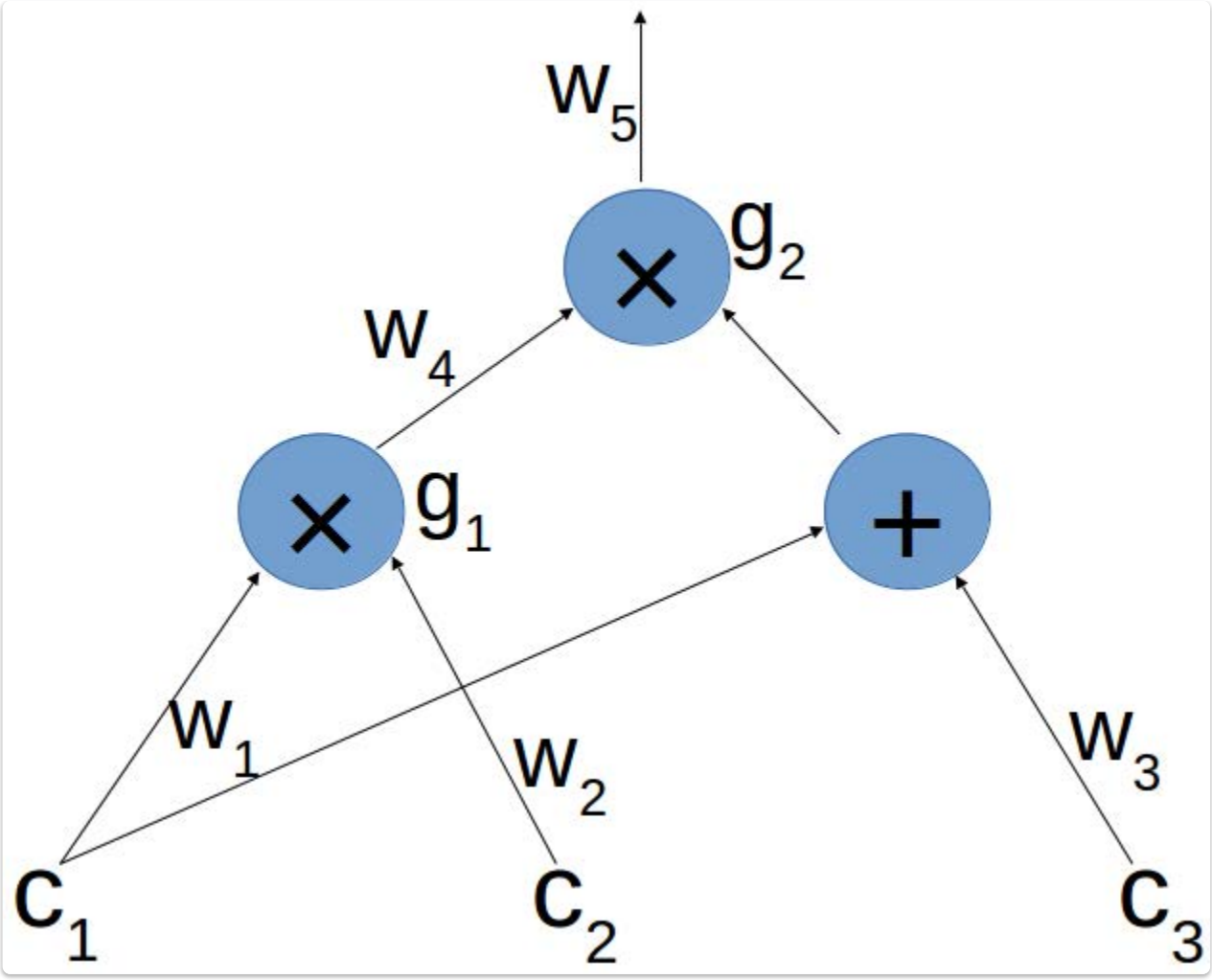
If course you have to put trust into someone training the network correctly (if there is such a thing as a correctly trained network). You can do some random samples to check the network, though.

Oursourcing Private Machine Learning via Lightweight Secure Arithmetic Computation
https://arxiv.org/pdf/1812.01372.pdf

Recap of zkSNARK process



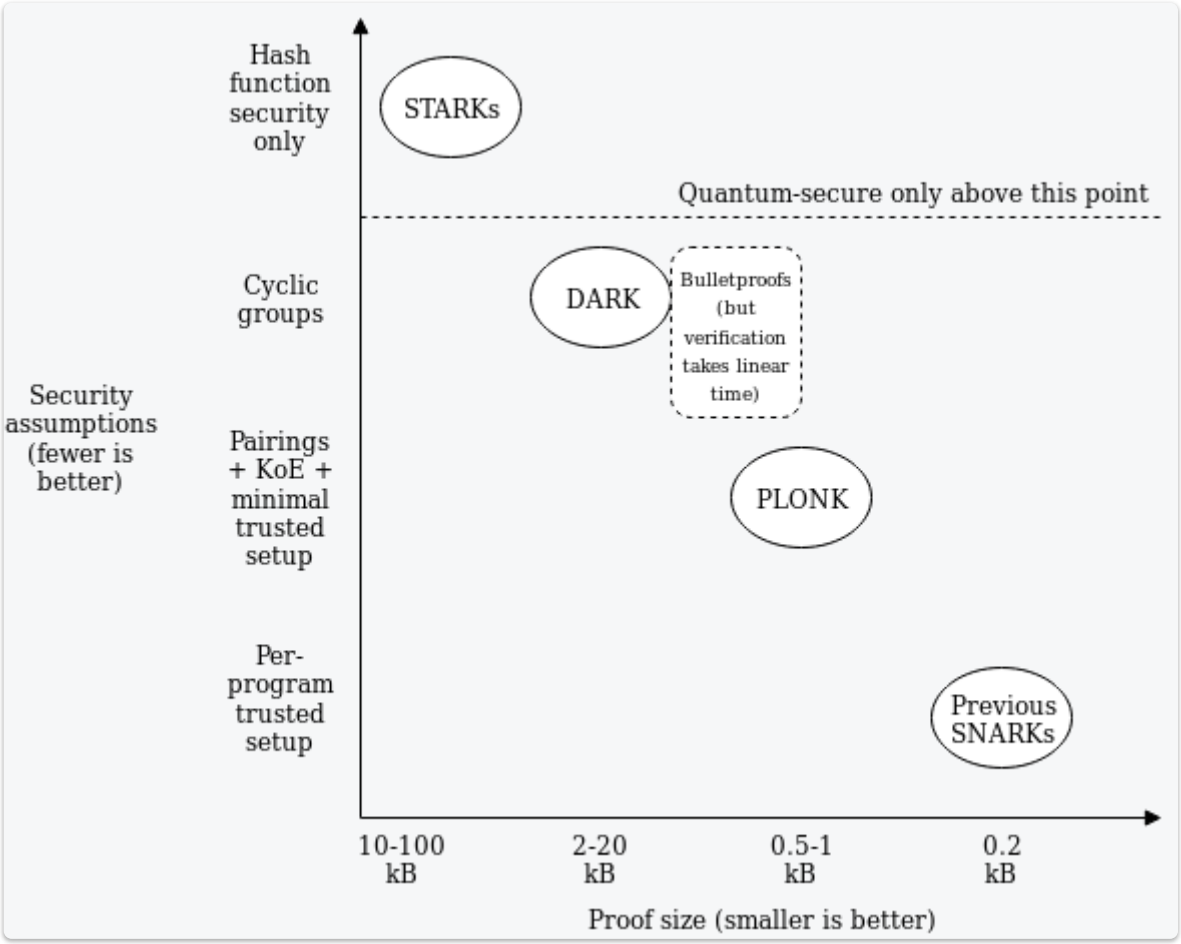
1. Trusted Setup
2. High level description -> arithmetic circuit



3. R1CS is created, and transformed to a QAP
4. Polynomials are transformed to give a non interactive proof
5. Zero knowledge is added via homomorphic hiding

This process has been adapted, more recent SNARK development has lead to a more modular approach wher polynomial commitments can be used instead of homomorphic hiding, at the

expense of the proof size.

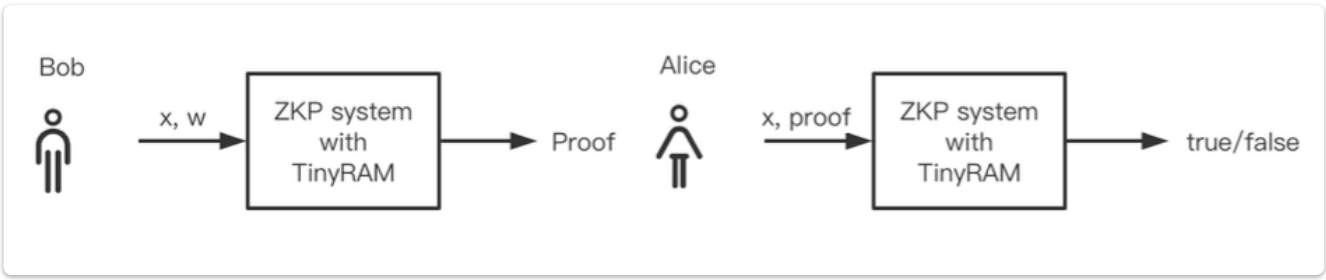


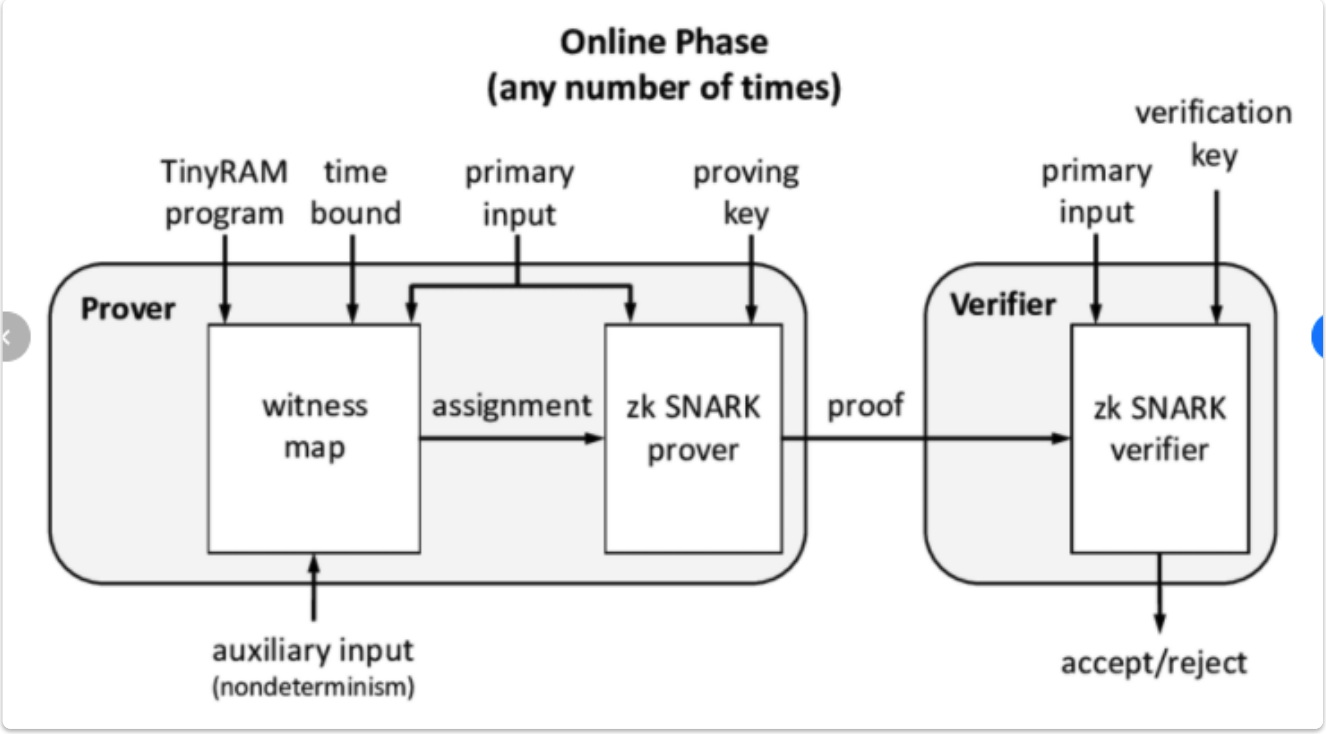
The proving systems have developed and use a mixture of cryptographic primitives / approaches, performance is still an issue for them, hence there is much research around optimisation, of for examle hash functions.

For example see this article about Halo and this article about PLONK

TinyRam

TinyRAM is a Reduced Instruction Set Computer (RISC) with byte-level addressable random-access memory.
It strikes a balance between two opposing requirements — “sufficient expressibility” and “small instruction set”:





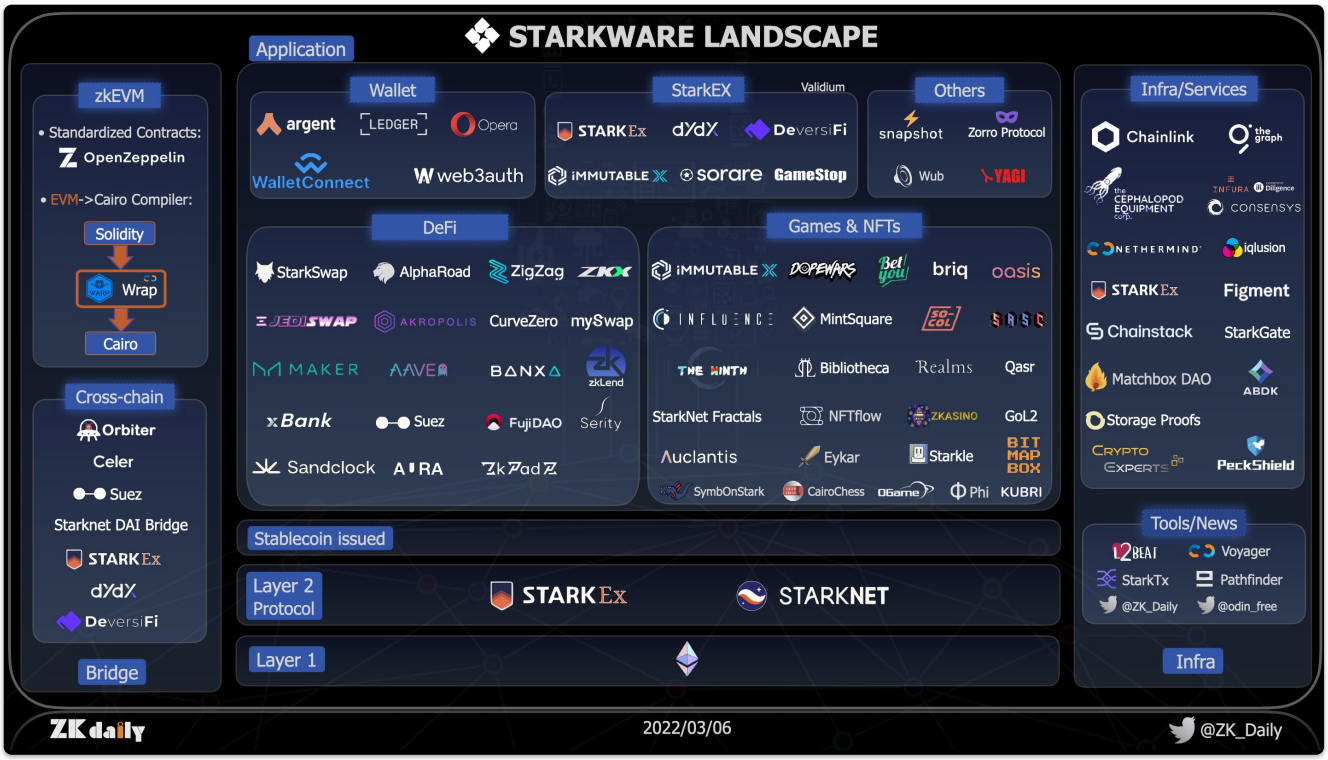
	A	B	C	D	E
1		hv - Edward Curves, 80 bits	vn - Edward Curves, 80 bits	hv - BN Curves, 128 bits	vn - BN Curves, 128 bits
2	key generator	306s	97s	123s	117s
3	prover	351s	115s	784s	147s
4	verifier	66.1ms	4.9ms	9.2ms	5.1ms
5	proof size	332B	230B	288B	288B

Starknet and Cairo Recap

"StarkNet is a permissionless decentralized 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."

StarkNet Milestones

	Availability
Smart contracts support general computation	✓
Smart contracts can interact with each other, allowing composability	✓
L1<>L2 interoperability	✓
Full L1 security through on-chain data (Rollup)	✓
Solidity to Cairo Compiler	✓
StarkNet Full Nodes	**Coming Soon**
Range of Data Availability Solutions	**Coming Soon**
Permissionless Sequencer and Prover	**Coming Soon**



https://www.starknet-ecosystem.com/
The area is under heavy development for example this was announced today
rpc adapter

From Starknet Documentation

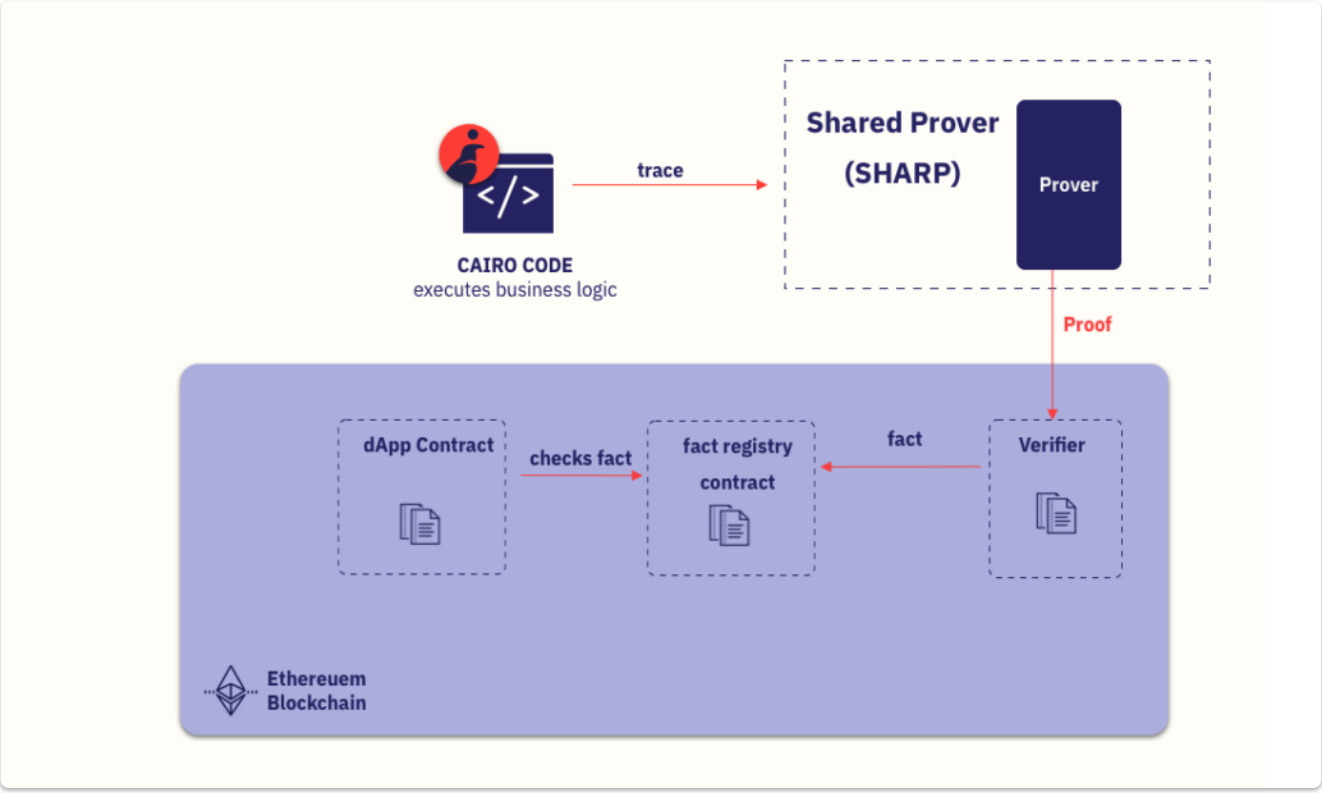
Starknet Components

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

Cairo

Details from documentation
Overview for blockchain developers

Cairo is a language for creating STARK-provable programs for general computation. Cairo powers StarkEx, which scales applications on Mainnet (including dYdX, Sorare, Immutable X, and DeversiFi). See Documentation



Cairo is the programming language used for StarkNet It aims to validate computation and includes the roles of prover and verifier.
It is a Turing complete language.

"In Cairo programs, you write what results are **acceptable**, not **how to** come up with results."

In solidity we might write a stament to extract an amount from a balance, in Cairo we would write a statement to check that for the parties involved the sum of the balances hasn't changed

Data types - the field element

In Cairo, the basic data type is an integer in the range $0 \leq x < P$ where P is a prime number. All the computations are done modulo P.

Problems working with modulo P :

In Cairo when you don't specify a type of a variable/argument, its type is a **field element** (represented by the keyword `felt`). In the context of Cairo, when we say "a field element" we mean an integer in the range $-P/2 < x < P/2$ where P is a very large (prime) number (currently it is a 252-bit number, which is a number with 76 decimal digits). When we add, subtract or multiply and the result is outside the range above, there is an overflow, and the appropriate multiple of P is added or subtracted to bring the result back into this range (in other words, the result is computed modulo P).

The most important difference between integers and field elements is **division**: Division of field elements (and therefore division in Cairo) *is not* the integer division you have in many programming languages, where the integral part of the quotient is returned (so you get `7 / 3 =`

`2`). As long as the numerator is a multiple of the denominator, it will behave as you expect (`6 / 3 = 2`). If this is not the case, for example when we divide `7/3`, it will result in a field element `x` that will satisfy `3 * x = 7`. It won't be `2.3333` because `x` has to be an integer. If this seems impossible remember that if `3 * x` is outside the range $-P/2 < x < P/2$ an overflow will occur which can bring the result down to 7. It's a well-known mathematical fact that unless the denominator is zero, there will always be a value `x` satisfying `denominator * x = numerator`.

Memory model

Cairo supports a read-only nondeterministic memory, which means that the value for each memory cell is chosen by the prover, but it cannot change over time (during a Cairo program execution). We use the syntax $[x]$ to represent the value of the memory at address x. The above implies, for example, that if we assert that $[0] = 7$ at the beginning of a program, then the value of $[0]$ will be 7 during the entire run.

It is usually convenient to think of the memory as a write-once memory: you may write a value to a cell once, but you cannot change it afterwards. Thus, we may interpret an instruction that asserts that $[0] == 7$ either as "read the value from the memory cell at address 0 and verify that you got 7" or "write the value 7 to that memory cell" depending on the context (in the read-only nondeterministic memory model they mean the same thing).

Registers

The only values that may change over time are held within designated registers:

- `ap` (allocation pointer) - points to a yet-unused memory cell.
- `fp` (frame pointer) - points to the frame of the current function. The addresses of all the function's arguments and local variables are relative to the value of this register. When a function starts, it is equal to `ap`. But unlike `ap`, the value of `fp` remains the same throughout the scope of a function.
- `pc` (program counter) - points to the current instruction.

A simple Cairo program

```
[ap] = [ap - 1] * [fp]; ap++
```

Adding some syntactic sugar

Some simple programs

```
%builtins output

from starkware.cairo.common.serialize import serialize_word

func main{output_ptr : felt*}():
    serialize_word(1234)
    serialize_word(4321)
    return ()
end
```

```
# Computes the sum of the memory elements at addresses:
# arr + 0, arr + 1, ..., arr + (size - 1).
func array_sum(arr : felt*, size) -> (sum):
    if size == 0:
        return (sum=0)
    end

    # size is not zero.
    let (sum_of_rest) = array_sum(arr=arr + 1, size=size - 1)
    return (sum=[arr] + sum_of_rest)
end
```

Using the Shared Prover (SHARP)

The Cairo SHARP collects several (possibly unrelated) programs and creates a STARK proof that they ran successfully. Such a batch of programs is called a "**train**". Just like a train doesn't leave the station on-demand, a Cairo train may take a while to be dispatched to the prover. It will wait for a large enough batch of program traces to accumulate or a certain amount of time to pass – whichever happens first.

Once the STARK proof was created, the SHARP sends it to be verified on-chain (ON GOERLI, FOR NOW). For each program in the train, the SHARP contract writes a fact in the Fact Registry attesting to the validity of the run with its particular output.

Creating starknet contracts

An additional declaration `%lang starknet` is used to indicate that we are writing code as a starknet contract rather than a stand alone cairo program. We also use a different compiler command
`starknet-compile`

Cairo and storage

Cairo is stateless – meaning any Cairo run is independent of an external state, and of other Cairo runs.
This allows for ordering proofs in any order we want, creating multiple proofs concurrently.
The way that we deal with state (which we will probably need for any useful application) is to keep the state stored on Ethereum, but to "compress" it by using a merkle tree.

A useful example is given in the documentation
Imagine we are implementing a voting system, on receiving a batch of votes, the Cairo program will calculate the new Merkle root, representing the new state of voters, and output the number of yes and no votes in the batch and the two roots – both the old one and the new one.
The SHARP will now generate a STARK proof that asserts these 2 roots represent a valid state transition with the given vote counts, and will write a fact on chain

Starknet contracts and Storage

Starknet adds to this by providing storage for contracts
See Contracts

The most common way for interacting with a contract's storage is through storage variables.

The '@storage_var' decorator declares a variable that will be kept as part of the contract storage.
The variable can consist of a single felt, or it can be a mapping from multiple arguments to a tuple of felts or structs.

The StarkNet contract compiler generates the Cairo code that maps the storage variable's name and argument values to an address – so that it can be part of the generated proof.

Language Features

See the documentation

TYPES

The following are available

- `felt` – a field element (see Field elements).
- `MyStruct` where `MyStruct` is a struct name.
- A tuple – For example `(a, b)` where `a` and `b` are types (see Tuples).
- `T*` where `T` is any type – a pointer to type `T`. For example: `MyStruct*` or `felt**`.

REFERENCES

You can use

```
let a = 23
```

There are more restrictions on the compile time behaviour
For example

```
func foo(x):
    # The compiler cannot deduce whether the if or the else
    # block will be executed.
    if x == 0:
        let a = 23
    else:
        let a = 8
    end

    # 'a' cannot be accessed, because it has
    # conflicting values: 23 vs 8.

    return ()
end
```

To get round this local variables can be used

```
local a = 3
```

FUNCTIONS

Functions are defined as follows

```
func func_name{implicit_arg1 : felt, implicit_arg2 : felt*}(
        arg1 : felt, arg2 : MyStruct*) -> (
        ret1 : felt, fet2 : felt):
    # Function body.
end
```

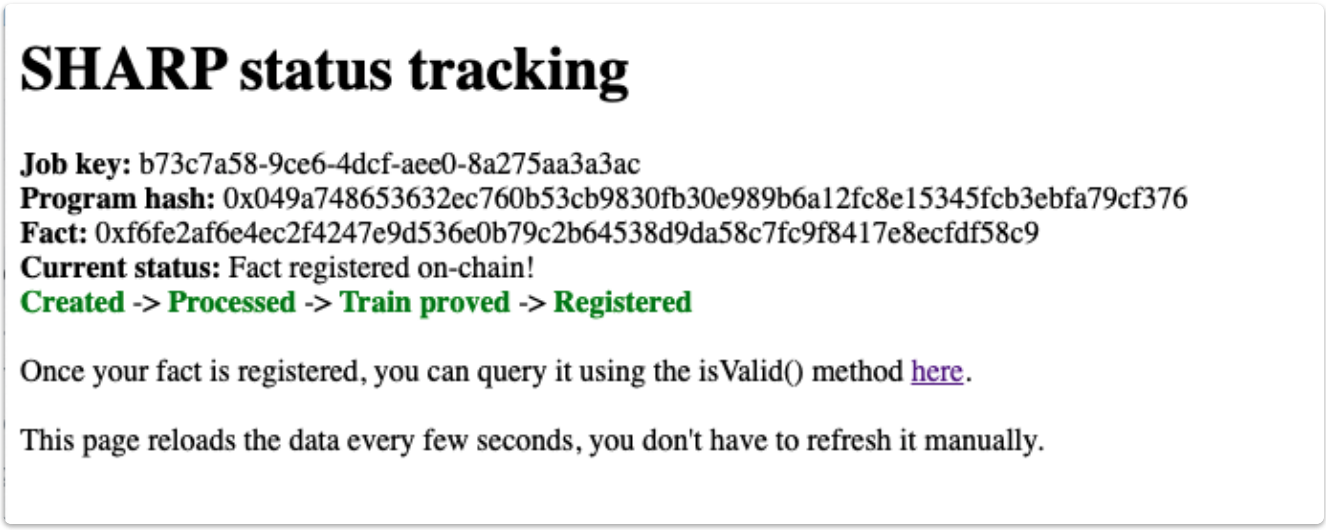
OUTPUT

Cairo programs can share information with the verifier using outputs. Whenever the program wishes to communicate information to the verifier, it can do so by writing it to a designated memory segment which can be accessed by using the output builtin. Instead of directly handling the output pointer, one can call the `serialize_word()` library function which abstracts this from the user.

Cairo Playground

An online tool to allow you to try out Cairo

SHARP STATUS



Starknet Voyager

https://voyager.online/contract/0x01de347b0277deef23915daddc92b3ce85ac55f6520d8f650be75eb368bd4ba4#readContract

Open Zeppelin Projects

- https://github.com/OpenZeppelin/nile
- Open Zeppelin Cairo contracts : https://github.com/OpenZeppelin/cairo-contracts

For example ERC20