Lesson 7

This Week

- Starknet and Cairo
- Cairo and Warp
- zk Rollups
- zkSync / StarkEx

RESOURCES

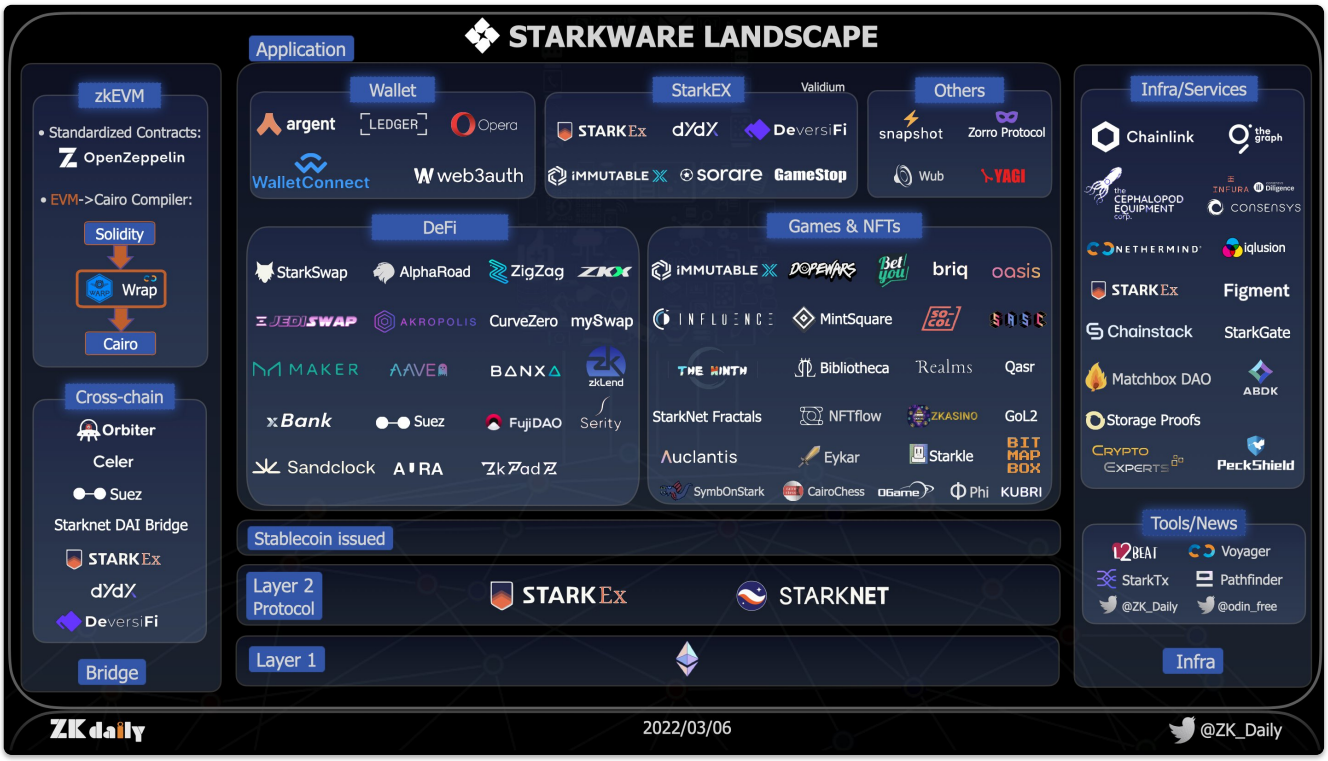
https://github.com/ZkPad-Labs/starknet-libs

Starknet

"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

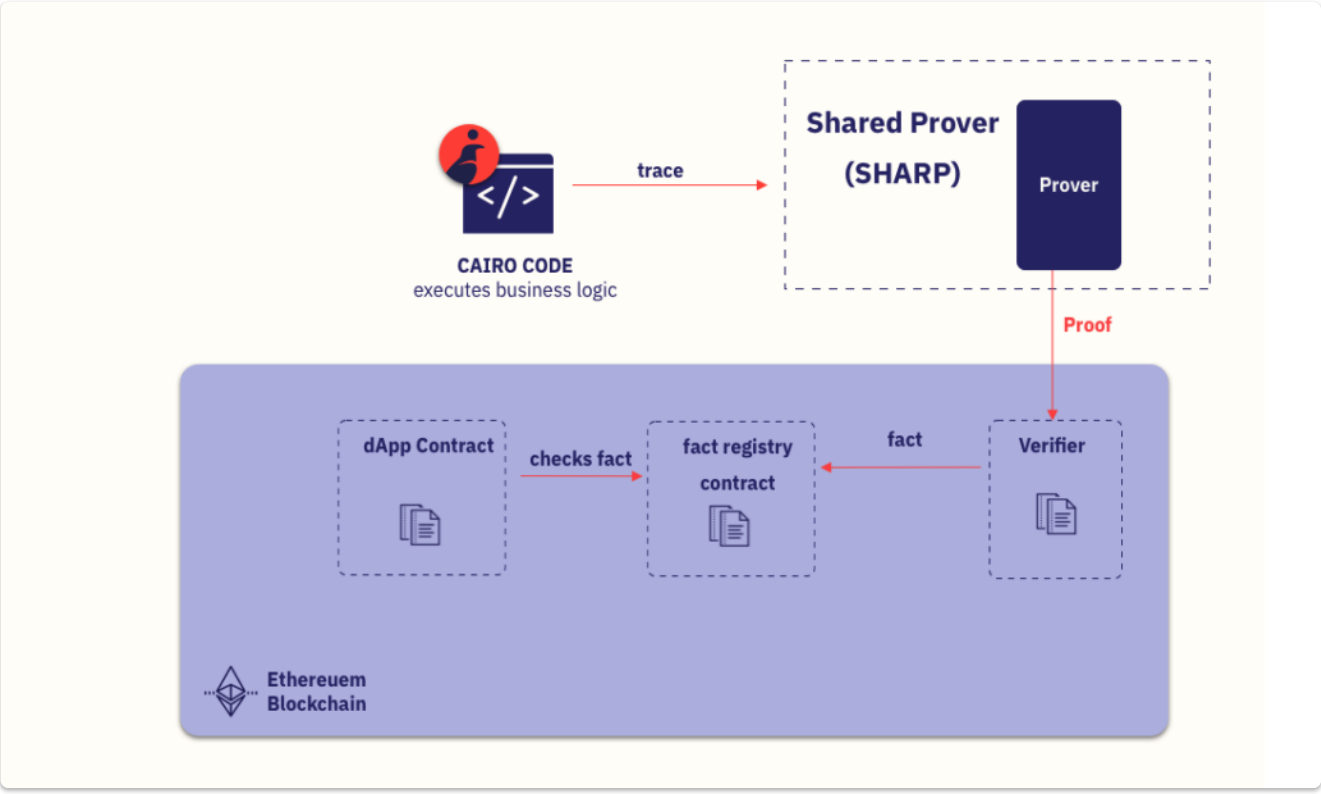
GAME OF LIFE ON STARKNET

https://www.gol2.io/nav

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)

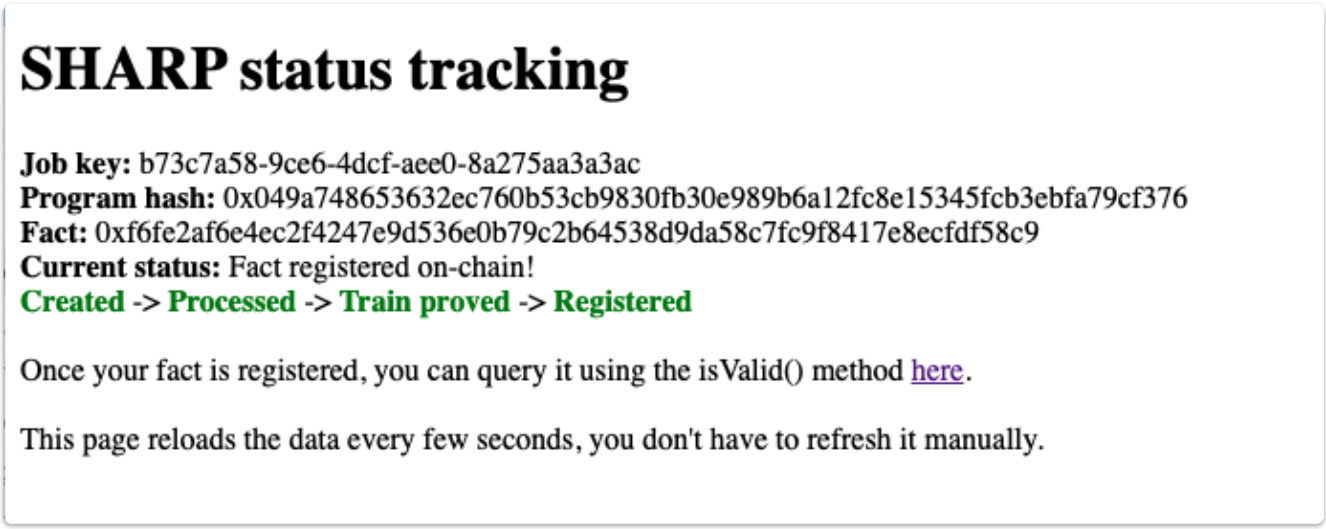
The Cairo SHARP collects several (possibly unrelated) programs and creates a STARK proof that they ran successfully. Such a batch of programs is called a "train". Just like a train doesn't leave the station on-demand, a Cairo train may take a while to be dispatched to the prover. It will wait for a large 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.

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