Lesson 7

This week

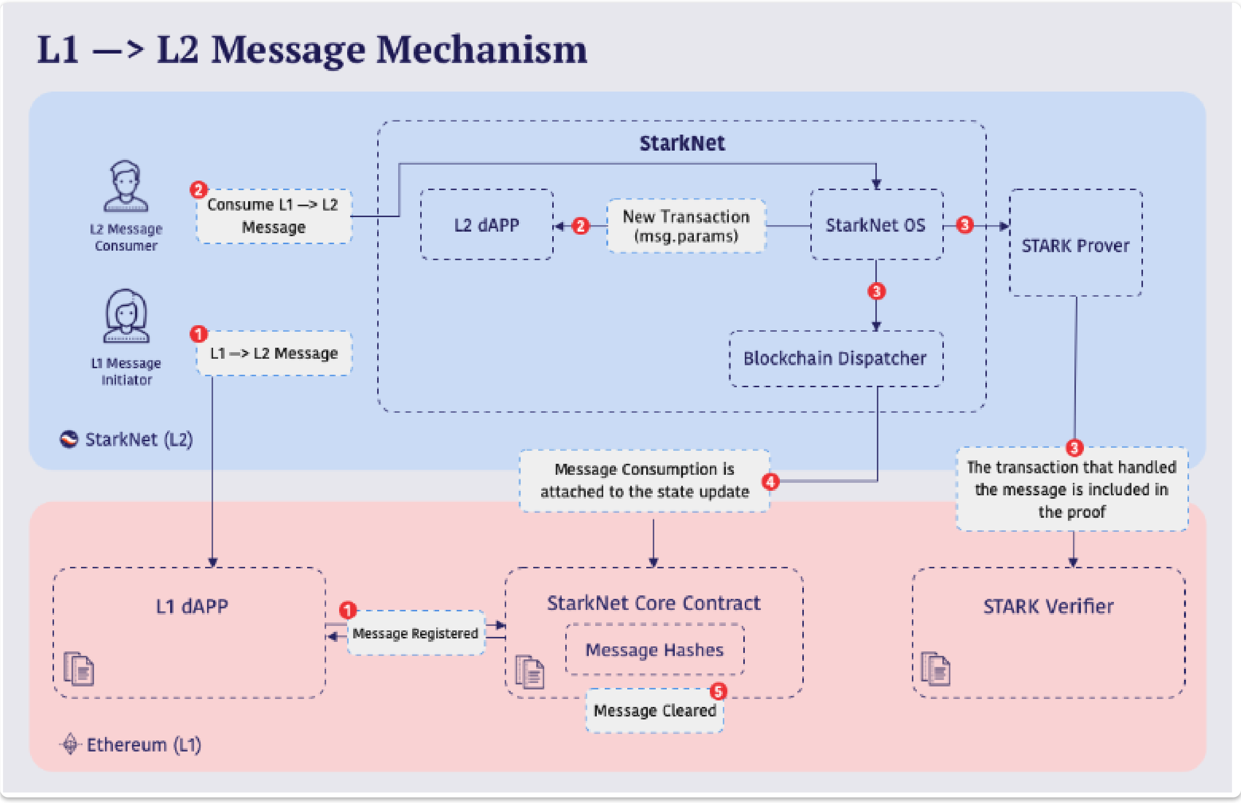
- Cairo Contracts continued
- Starknet JS
- DeFi - ZCash / Aztec
- Noir
- Mina

Today

- Contracts continued
- Warp
- Non deterministic computation
- Starknet JS

Starknet Architecture continued

L1 to L2 Messaging

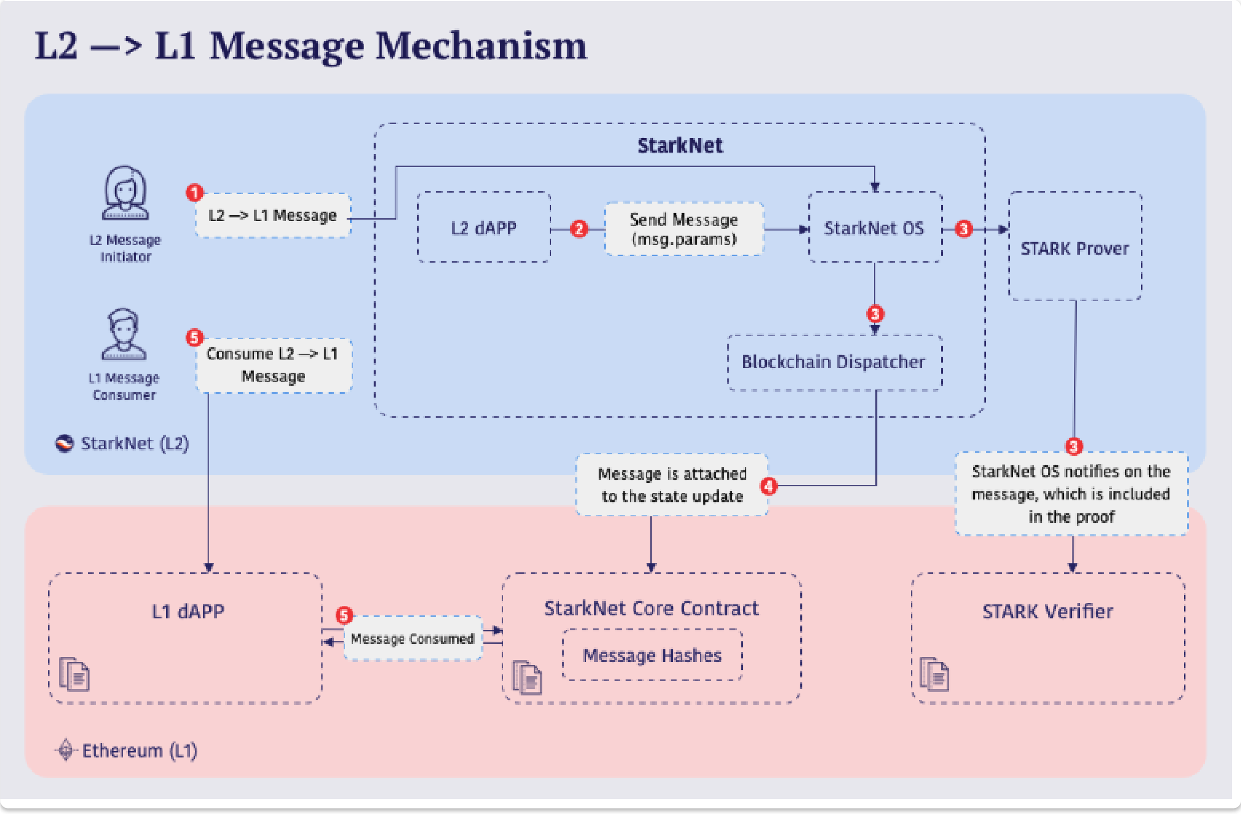


1. The L1 contract calls (on L1) the `send_message()` function of the StarkNet core contract, which stores the message. In this case the message includes an additional field - the "selector", which determines what function to call in the corresponding L2 contract.
2. The StarkNet Sequencer automatically consumes the message and invokes the requested L2 function of the contract designated by the "to" address.

This direction is useful, for example, for "deposit" transactions.

Note that while honest Sequencers automatically consume L1 -> L2 messages, it is not enforced by the protocol (so a Sequencer may choose to skip a message). This should be taken into account when designing the message protocol between the two contracts.

L2 to L1 Messaging



Messages from L2 to L1 work as follows:

1. The StarkNet (L2) contract function calls the library function `send_message_to_l1()` in order to send the message. It specifies:

   > 1. The destination L1 contract ("to"),
   > 2. The data to send ("payload")

   The StarkNet OS adds the "from" address, which is the L2 address of the contract sending the message.

2. Once a state update containing the L2 transaction is accepted on-chain, the message is stored on the L1 StarkNet core contract, waiting to be consumed.

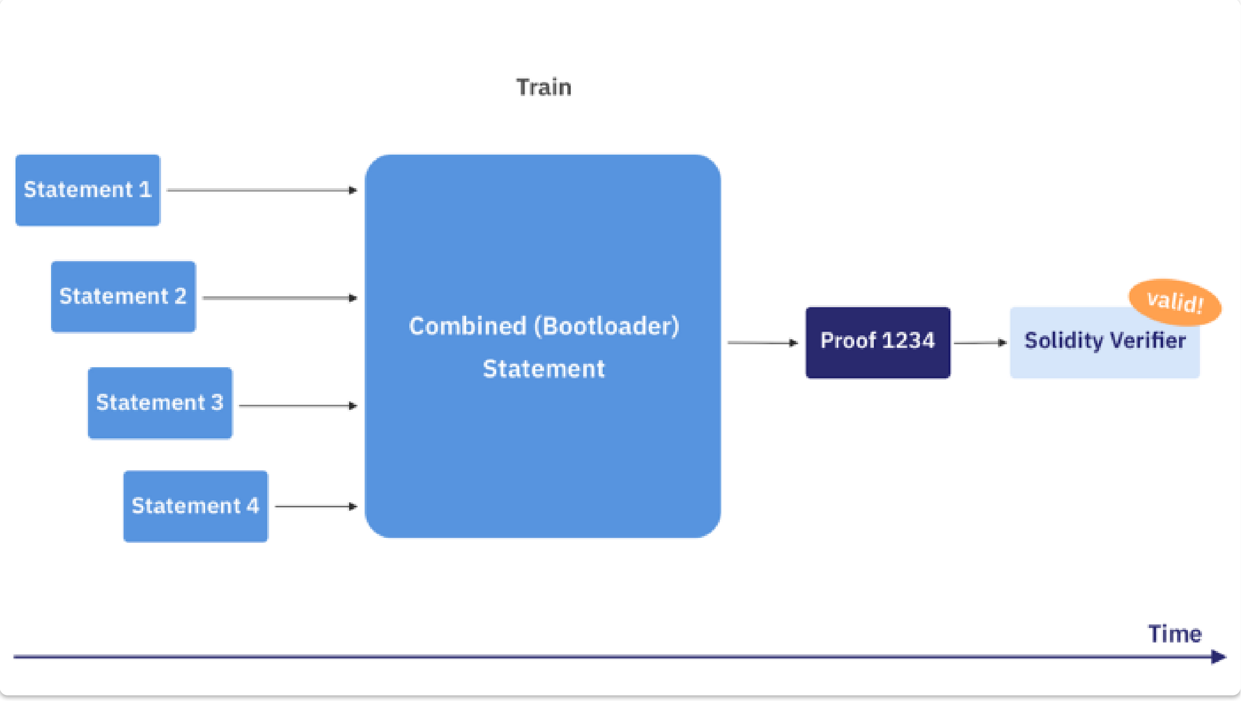
3. The L1 contract specified by the "to" address invokes the `consumeMessageFromL2()` of the StarkNet core contract.

Note: Since any L2 contract can send messages to any L1 contract it is recommended that the L1 contract check the "from" address before processing the transaction.

Recursive STARKS

See post

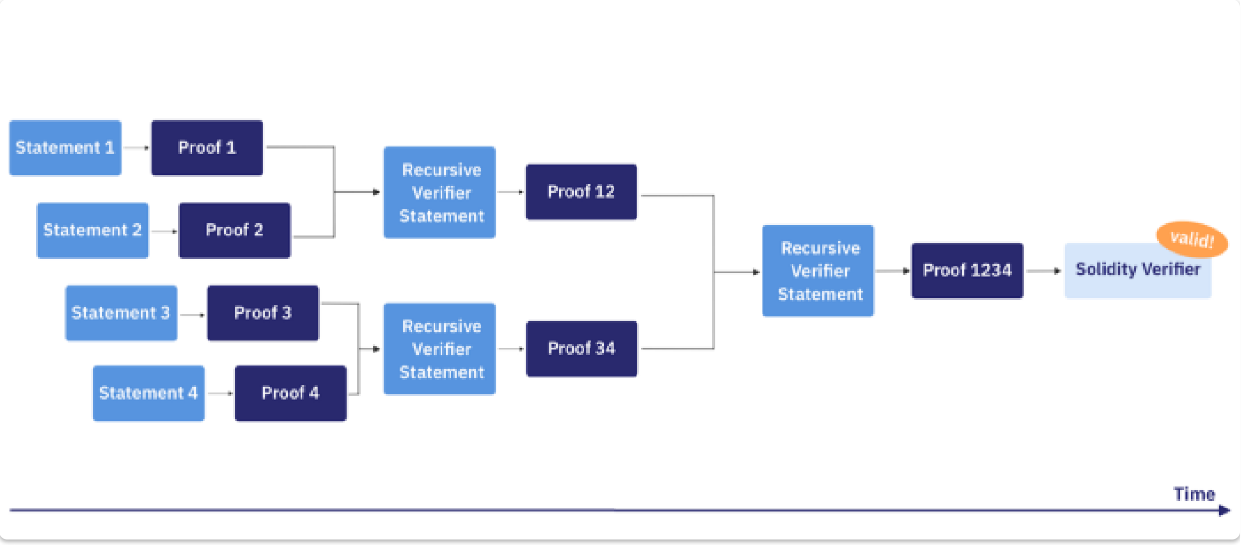
Initially SHARP (The shared prover) would process proofs from applications sequentially, oncce a threshold number of tranactions had arrived, a proof would be generated for all of them.



The amount of memory needed to to generate the proof was a limiting factor.

STARKs have roughly linear proving time and log validation time.

Recursive proofs



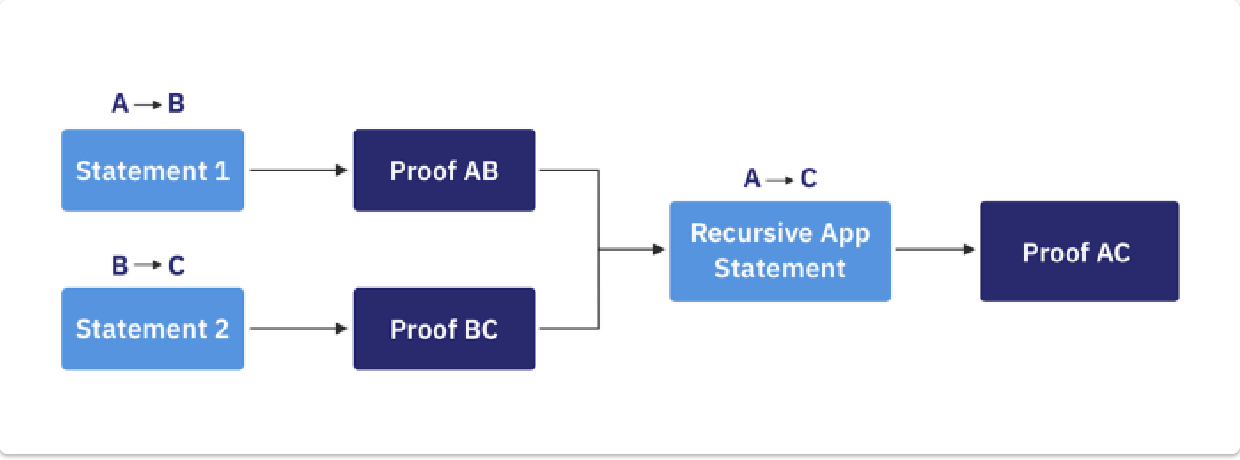
Here the proofs are calculated in parallel, then combined in pairs and a proof created and so on.

This results in

1. Reduced on chain cost, and memory requirements
2. Reduced latency since the proofs can be computed in parallel and we don't need to wait for the final proof to arrive.

Application recursion

Each STARK proof attests to the validity of a statement applied to some input
STARK recursion compresses two proofs with *two* inputs into *one* proof with two inputs. In other words, while the number of proofs is reduced, the number of inputs is kept constant. If the recursive statement is allowed to be *application-aware*, i.e. recognizes the semantics of the application itself, it can both compress two proofs into one *as well as* combine the two inputs into one.



Cairo Contracts Continued

Comparison operations

There is ambiguity about the functions to use for comparison, see this issue
Open Zeppelin have therefore created 2 libraries
Safe_cmp
and
FeltMath

Namespaces

To allow modularity in our contracts we have the namespace keyword

```
namespace encode  {
        func homework1(a: felt, b:felt) -> (c: felt){
                return (a*b);
        }
}
```

We can then reference this function as

```
encode.homework1(11, 13);
```

Contract Classes

A recent addition to starknet, since version 0.9
From medium article
"Taking inspiration from object-oriented programming, we distinguish between the contract code and its implementation. We do so by separating contracts into classes and instances."

The way it works is similar to the proxy pattern in Ethereum.

A contract class is the definition of the contract: Its Cairo bytecode, hint information, entry point names, and everything necessary to unambiguously define its semantics. Each class is identified by its class hash.

A contract instance, is a deployed contract corresponding to some class. Note that only contract instances behave as contracts, i.e., have their own storage and are callable by transactions/other contracts.
A contract class does not necessarily have a deployed instance in StarkNet.

The declare transaction type declares a class but does not deploy an instance of that class.

The deploy system call takes 3 arguments

- The class hash
- Salt
- Constructor arguments

This will deploy a new instance of the contract whose address depends on the above arguments, this is similar to the CREATE2 op code on Ethereum.

Declaring the contract

With the starknet CLI

```
starknet declare --contract contract_compiled.json
```

With protostar

```
protostar declare ./build/main.json --network testnet
```

Deploying contracts

If you are using the starknet CLI you can use

```
starknet deploy --class_hash $CLASS_HASH
```

If you are using protostar, see docs

```
protostar deploy 0xdeadbeef --network testnet
```

Contract Extensibility

See Forum post

Currently

- Cairo has no explicit smart contract extension mechanisms such as inheritance or composability
- There's no function overloading making function selector collisions very likely – more so considering selectors do not take function arguments into account
- Any `@external` function defined in an imported module will be automatically re-exposed by the importer (i.e. the smart contract)
- Builtins cannot be imported more than once in the entire imports hierarchy, resulting in errors on import (or errors on compilation if not added) – and most contracts will need the same common set of builtins such as `pedersen`, `range_check`, etc.

Contracts and libraries

Libraries define behavior and storage while contracts build on top of libraries. Contracts can be deployed – libraries cannot.

Guidelines from Open Zeppelin when using libraries, see tips from Nethermind below

Considering the following types of functions:

- `private`: private to a library, not meant to be used outside the module or imported
- `public`: part of the public API of a library
- `internal`: subset of `public` that is either discouraged or potentially unsafe (e.g. `_transfer` on ERC20)
- `external`: subset of `public` that is ready to be exported as-is by contracts (e.g. `transfer` on ERC20)
- `storage`: storage variable functions

Then:

- Must implement `public` and `external` functions under a namespace
- Must implement `private` functions outside the namespace to avoid exposing them
- Must prefix `internal` functions with an underscore (e.g. `ERC20._mint`)
- Must not prefix `external` functions with an underscore (e.g. `ERC20.transfer`)
- Must prefix `storage` functions with the name of the namespace to prevent clashing with other libraries (e.g. `ERC20balances`)
- Must not implement any `@external`, `@view`, or `@constructor` functions
- Can implement initializers (never as `@constructor` or `@external`)

> - Must not call initializers on any function
>
> Namespaces allow us to better distinguish between four types of library functions and how to approach each of them for secure development:

Tips and best practices

There are some useful tips here
Some items from the coding guideline from Nethermind

Split the contract into a logic file and a contract file

- A library file (or logic file), named `my_contract_library.cairo`, contains the logic code of the contract. Namely, it contains: (i) internal and external functions encapsulated in a namespace, and (ii) storage variables and events defined outside the namespace.
- A contract file, named `my_contract.cairo`, exposes external functions from its corresponding library file and other library files. For instance, an implementation of cToken that inherits from an ERC20 contract would expose both functions in `c_token_library.cairo` and `erc20_library.cairo`.

Error messages

Use `with_attr error_message(...)` as shown in yesterday's notes, make sure only one thing can fail in the block.

Passing arrays in calldata

To pass an array of felt to a function, the usual pattern is to pass a pointer of felt and the array’s length. We recommend encapsulating this array in a struct `MyStruct` and use `MyStruct.SIZE` for the array length.

Recursion

For each loop, we recommend defining an internal function suffixed with `_inner` or `_loop` to do the job.

```
func sum_array(array_len : felt, array : felt*) -> felt {
    let sum = 0;
    let (res) = _sum_array_inner{array_len=array_len, array=array,
sum=sum}(0);
    return res;
}



func _sum_array_inner{array_len : felt, array : felt*, sum : felt}
(current_index : felt) -> felt {
    if (current_index - array_len == 0) {
        return (sum);
    }
    let sum = sum + array[current_index];
    return _sum_array_inner(current_index + 1);
}
```

Variable names

To avoid collisions, prefix variable names with the namespace that was specified with the `namespace` keyword

```
// in my_contract_library.cairo
@storage_var
func MyContract_name() {
}

namespace MyContract {
    ...
}
```

Calling functions in other contracts

You can call external functions in other contracts, but to do this you need to provide an interface, for this we use the `@contract_interface` decorator.
The body of the function and implicit arguments are not needed.

For example

```
@contract_interface
namespace IBalanceContract {
    func increase_balance(amount: felt) {
    }

    func get_balance() -> (res: felt) {
    }
}
```

This can be called from another contract as follows.
We need to pass the contract address as an additional argument.

```
@external
func call_increase_balance{syscall_ptr: felt*, range_check_ptr}(
    contract_address: felt, amount: felt
) {
    IBalanceContract.increase_balance(
        contract_address=contract_address, amount=amount
    );
    return ();
}
```

Upgradable Contracts

See this article

We can use a proxy pattern, this involves a proxy contract forwarding calls to an 'implementation' contract.
It uses a function `__default__` that is the equivalent of the fallback function in Solidity.
The flow is then

Tx -> Proxy contract -> delegates to -> Implementation contract.

There are preset contracts from Open Zeppelin to help you with this.

Warp

Warp allows you transpile Solidity contracts into Cairo

Installation Instructions

See Warp installation instructions

1. On macos:

```
brew install z3
```

2. On ubuntu:

```
sudo apt install libz3-dev
```

Make sure that you have the `venv` module for your python installation.

Installation

Without any virtual environment activated run the following in order:

```
yarn global add @nethermindeth/warp
```

Run the following to see the version and that it was installed:

```
warp version
```

Finally run the following to install the dependencies:

```
warp install
```

Test installation works by transpiling an example ERC20 contract:

`warp transpile example_contracts/ERC20.sol`

Using Docker

`docker build -t warp .`

`docker run --rm -v $PWD:/dapp --user $(id -u):$(id -g) warp transpile example_contracts/ERC20.sol`

Using Warp

`warp transpile example_contracts/ERC20.sol`

`warp transpile example_contracts/ERC20.sol --compile-cairo`

You can then deploy your cairo code to the network, with the following commands you need to specify the network, in our case alpha-goerli

```
warp deploy test.json --network alpha-goerli
```

```
Deploy transaction was sent.
Contract address:
0x0403bd2f0abdd765398d6a50ff89cfe9ac48760f3b94ba2728bfbacdaff9f59a
Transaction hash:
0x32ca42d1341703cc957845ea53a71b3eb2e762ff148cb9dc522322eede94b65
```

You can invoke a transaction on your contract

```
warp invoke --program test.json --address
0x0403bd2f0abdd765398d6a50ff89cfe9ac48760f3b94ba2728bfbacdaff9f59a  --
network
alpha-goerli --function store --inputs [13]
```

```
Invoke transaction was sent.
Contract address:
0x0403bd2f0abdd765398d6a50ff89cfe9ac48760f3b94ba2728bfbacdaff9f59a
Transaction hash:
0x1d1ec8278ccf41452737e80a54e7626299e598528363ced7a527d810f9d6881
```

And check the status

> ```
> warp status
> 0x1d1ec8278ccf41452737e80a54e7626299e598528363ced7a527d810f9d6881 --
> network alpha-goerli
> ```
>
> which will give a answer similar to
>
> ```
>         {
>                 "block_hash":
> "0x1c55254f16d087f0bf7776183c4d38549680e68600394167f304f1afe5a035e",
>                 "tx_status": "ACCEPTED_ON_L1"
>         }
>
>
> ```
>
> You should be able to see the details on the block explorer
> Voyager Block Explorer
>
> There is also now a vyper transpiler

Non Determinism and under constrained code

From Perama's notes

AIR

We are interested in computational integrity, and as we will see in later lessons, all the steps within a computation can be represented as polynomials.
This form is called the algebraic intermediate representation (AIR).
The process has been optimised so that the AIR can be tested quickly, and a proof generated quickly
Building blocks of computation represented as an AIR can be combined together, which is the basis for Cairo.

To use a hardware analogy

- ASIC (AIR)
- CPU (Multiple AIRs)

The name Cairo comes from: a CPU built from AIRs (CPU-AIR, Oh nice –> CAIRO).

CAIRO is a non-deterministic, turing complete, functional high level language.
It has a register-based memory model and a compiler. The compiler produces a table of computational steps called a trace.
The trace is used by the prover to construct AIRs which are combined, and converted into a STARK proof.

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

```
func main{}() {
    alloc_locals;
    local x;

    assert x + 3 = 10;
    return ();
}
```

This is expecting the prover to provide a value for x

We can add a hint as follows

```
func main{}() {
    alloc_locals;
    local x;

    %{
    ids.x = 4
    %}

    assert x + 3 = 10;
    return ();
}
```

so this would fail
but if we produce an acceptable hint

```
    %{
    ids.x = 7
    %}
```

Then our code will succeed

Under constrained code

We then have to be careful that our asserts are sufficient that only a correct assignment will be accepted.
For example

```
func main{}() {
    alloc_locals;
    local x;

    %{
    ids.x = 5
    %}

    assert x * x = 25;
    return ();
}
```

will work, but the following also works, which may not be what we wanted.

```
func main{}() {
    alloc_locals;
    local x;

    %{
```

> ```
>     ids.x = -5
>     %}
>
>     assert x * x = 25;
>     return ();
> }
> ```

Starknet JS

Introduction

Documentation
https://www.starknetjs.com/docs/API/provider

This is modelled on libraries such as Web3.js

The main areas are

- Provider API - connecting to starknet
- Account API - connection with an account
- Signer API - allows signatures
- Contract API - an object representing a contract
- Utils API - Utility methods

Installation

```
npm install starknet@next
```

Provider API

You can create a provider with

```
const provider = new starknet.Provider()
```

or if you want specify the network

```
const provider = new starknet.Provider({
sequencer: {
        network: 'mainnet-alpha' // or 'goerli-alpha'
        }
})
```

To interact with a contract we use the provider we set up

Provider methods

callContract

```
provider.callContract(call [ , blockIdentifier ]) => _Promise
```

The call object has the following structure

- call.contractAddress - Address of the contract
- call.entrypoint - Entrypoint of the call (method name)
- call.calldata - Payload for the invoking method

Response

```
{
result: string[];
}
```

getTransactionReceipt

```
provider.getTransactionReceipt(txHash) => _Promise
```

Response

```
{
transaction_hash: string;
status: 'NOT_RECEIVED' | 'RECEIVED' | 'PENDING' | 'ACCEPTED_ON_L2' |
'ACCEPTED_ON_L1' | 'REJECTED';
actual_fee?: string;
status_data?: string;
messages_sent?: Array<MessageToL1>;
events?: Array<Event>;
l1_origin_message?: MessageToL2;
}
```

Deploy Contract

```
provider.deployContract(payload [ , abi ]) => _Promise
```

Response

```
{
transaction_hash: string;
contract_address?: string;
};
```

Wait For Transaction

`provider.waitForTransaction(txHash [ , retryInterval]) => Promise < void >`

> Wait for the transaction to be accepted on L2 or L1.

Other methods

- getBlock
- getClassAt
- getStorageAt
- getTransaction
- declareContract
- waitForTransaction

A useful library is get-starknet which provides connection methods.
If you are connecting with a wallet use the connect method from the get-starknet module

```
const starknet = await connect()
// connect to the wallet
await starknet?.enable({ starknetVersion: "v4" })
const provider = starknet.account
```

Signer API

The Signer API allows you to sign transactions and messages

You can generate a key pair by using the utility functions

`ec.genKeyPair()`
or
`getKeyPair(private_key)`

The signer object is then created with

`new starknet.Signer(keyPair)`

You can then sign messages

```
signer.signMessage(data, accountAddress) => _Promise
```

Code Example

```
const privateKey = stark.randomAddress();
const starkKeyPair = ec.genKeyPair(privateKey);
const starkKeyPub = ec.getStarkKey(starkKeyPair);
```

Account API

The Account object extends the Provider object
To create the account object, an account contract needs to have been deployed, see below for guide to deploy an account contract.

```
const account  = new starknet.Account(Provider, address,
starkKeyPair)
```

Account Properties

```
account.address =>string
```

Account Methods

```
account.getNonce() => Promise
account.estimateFee(calls [ , options ]) => _Promise
account.execute(calls [ , abi , transactionsDetail ]) => _Promise
account.signMessage(typedData) => _Promise
account.hashMessage(typedData) => _Promise
account.verifyMessageHash(hash, signature) => _Promise
account.verifyMessage(typedData, signature) => _Promise
```

See guide to creating and deploying an account

Contract

Creating the contract object

```
new starknet.Contract(abi, address, providerOrAccount)

contract.attach(address)` _for changing the address of the connected
contract_

contract.connect(providerOrAccount)` _for changing the provider or
account_
```

Contract Properties

```
contract.address => string
contract.providerOrAccount => ProviderInterface | AccountInterface
contract.deployTransactionHash => string | null
contract.abi => Abi
```

Contract Interaction

1. View Functions

contract.METHOD_NAME(...args [, overrides]) => Promise < Result >

The type of the result depends on the ABI.
The result object will be returned with each parameter available positionally and if the parameter is named, it will also be available by its name.

The override can identify the block : `overrides.blockIdentifier`

Code Example

`const bal = await contract.get_balance()`

2. Write Functions

contract.METHOD_NAME(...args [, overrides]) => Promise < AddTransactionResponse >

Overrides can be

- overrides.signature - Signature that will be used for the transaction
- overrides.maxFee - Max Fee for the transaction

- overrides.nonce - Nonce for the transaction

Code Example

```
await contract.increase_balance(13)
```

Utils

Useful Methods

- toBN

  `toBN(number: BigNumberish, base?: number | 'hex'): BN`

  Converts BigNumberish to BN.
  Returns a BN.

- uint256ToBN

  `uint256ToBN(uint256: Uint256): BN`

  Function to convert `Uint256` to `BN` (big number), which uses the `bn.js` library.

- getStarkKey

  `getStarkKey(keyPair: KeyPair): string`

  Public key defined over a Stark-friendly elliptic curve that is different from the standard Ethereum elliptic curve

- getKeyPairFromPublicKey

  `getKeyPairFromPublicKey(publicKey: BigNumberish): KeyPair`

  Takes a public key and casts it into `elliptic` KeyPair format.
  Returns keyPair with public key only, which can be used to verify signatures, but can't sign anything.

- sign

  `sign(keyPair: KeyPair, msgHash: string): Signature`

  Signs a message using the provided key.
  keyPair should be an KeyPair with a valid private key.
  Returns an Signature.

- verify

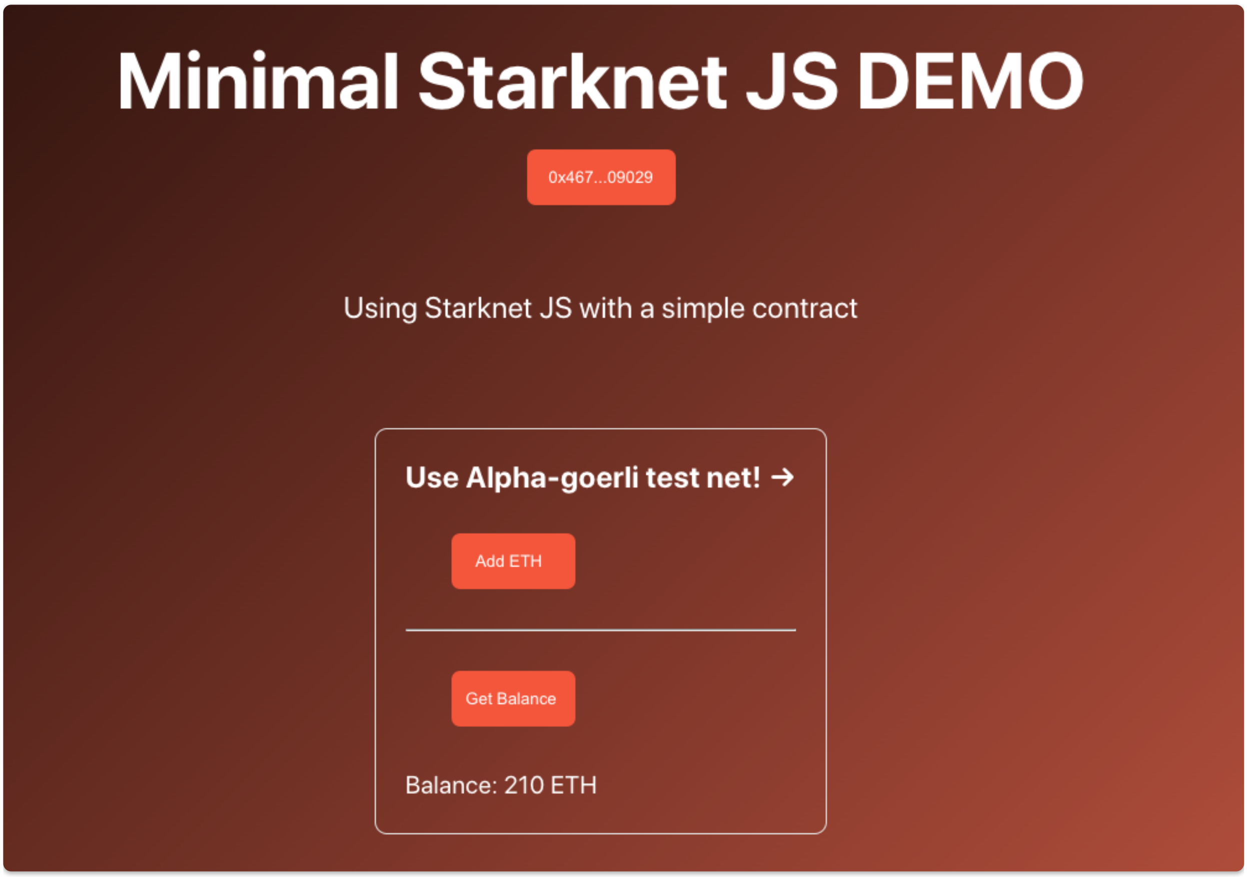
  `verify(keyPair: KeyPair | KeyPair[], msgHash: string, sig: Signature): boolean`

Verifies a message using the provided key.
keyPair should be an KeyPair with a valid public key.
sig should be an Signature.
Returns true if the verification succeeds.

Example in repo

Code

Based on tutorial from @darlingtonnnam



Minimal Starknet JS DEMO

Using Starknet JS with a simple contract

Use Alpha-goerli test net! →

Balance: 210 ETH

Links

Starknet.js workshop : https://github.com/0xs34n/starknet.js-workshop
Tutorial on medium :https://medium.com/@darlingtonnnam/an-in-depth-guide-to-getting-started-with-starknet-js-a55c04d0ccb7

Alternatives to starknet.js

Rust library
Python library