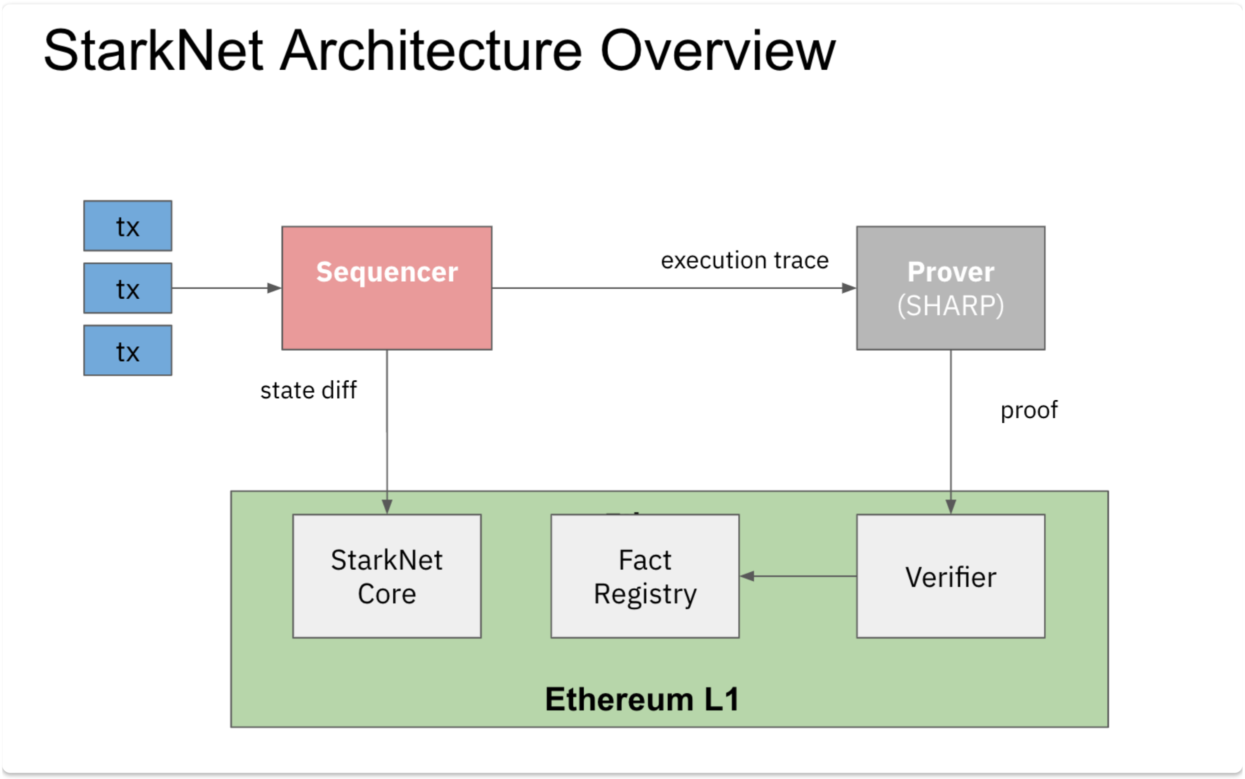
Lesson 6 - Cairo contracts

Starknet Architecture



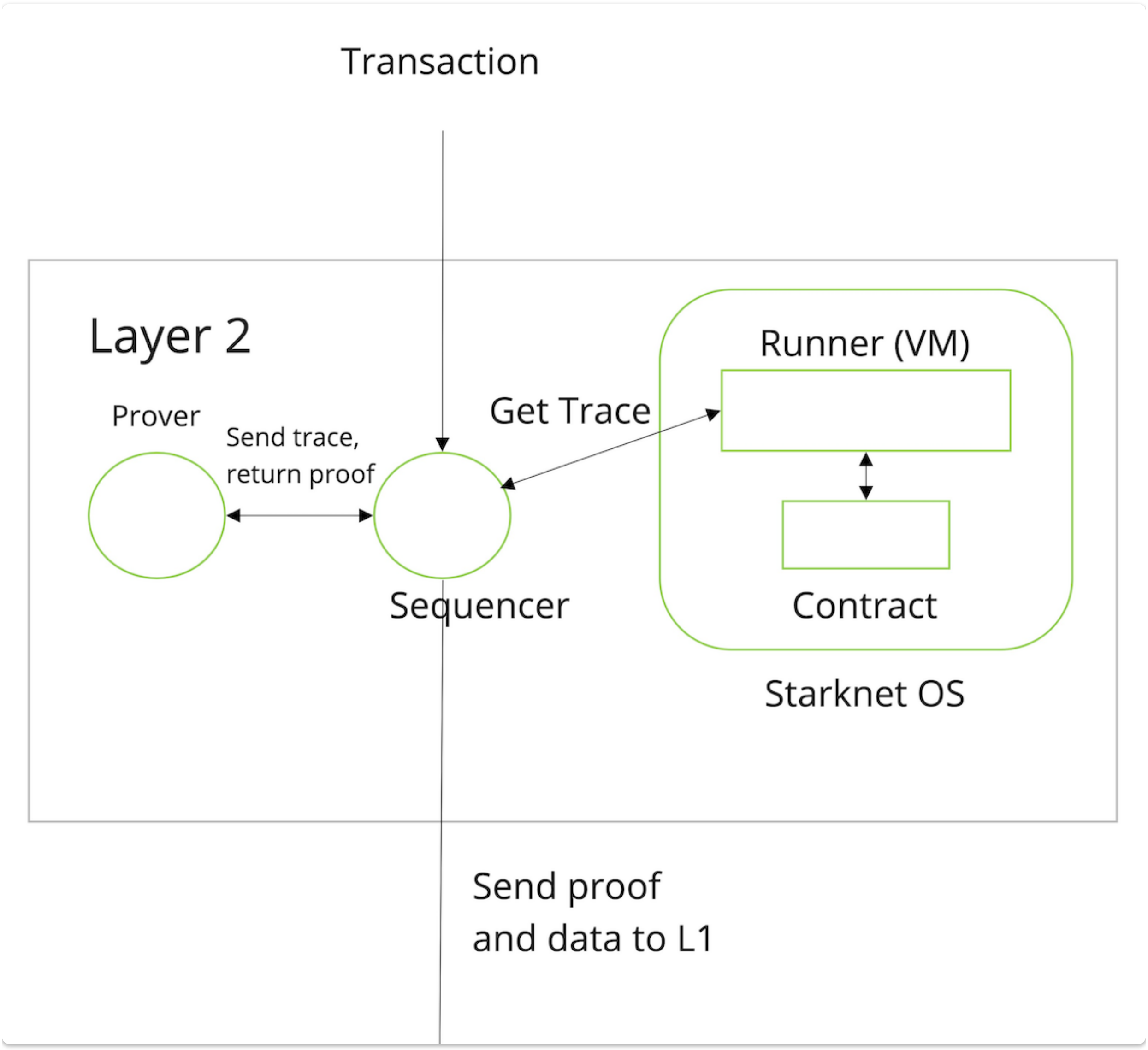
See this article for a good overview

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



Starknet has blocks, see Block structure
which consists of a header and a set of transactions
For further details see Starknet documentation

Transaction Status

1. NOT_RECEIVED

Transaction is not yet known to the sequencer

2. RECEIVED

Transaction was received by the sequencer. Transaction will now either execute successfully or be rejected.

3. PENDING

Transaction executed successfully and entered the pending block.

4. REJECTED

Transaction executed unsuccessfully and thus was skipped (applies both to a pending and an actual created block). Possible reasons for transaction rejection:

- An assertion failed during the execution of the transaction (in StarkNet, unlike in Ethereum, transaction executions do not always succeed).
- The block may be rejected on L1, thus changing the transaction status to `REJECTED`

5. ACCEPTED_ON_L2

Transaction passed validation and entered an actual created block on L2.

6. ACCEPTED_ON_L1
   Transaction was accepted on-chain.

Transaction types

1. Deploy
2. Invoke
3. Declare

Invoke Transaction Structure

Name	Type	Description
`contract_address`	`FieldElement`	The address of the contract invoked by this transaction
`entry_point_selector`	`FieldElement`	The encoding of the selector for the function invoked (the entry point in the contract)
`calldata`	`List<FieldElement>`	The arguments passed to the invoked function
`signature`	`List<FieldElement>`	Additional information given by the caller, representing the signature of the transaction
`max_fee`	`FieldElement`	The maximum fee that the sender is willing to pay for the transaction
`version`	`FieldElement`	The transaction's version [1]

Full Nodes

These run the Pathfinder client to keep a record of all the transactions performed in the rollup and to track the current global state of the system.
Full Nodes receive this information through a p2p network where changes in the global state and the validity proofs associated with it are shared everytime a new block is created.
When a new Full Node is set up it is able to reconstruct the history of the rollup by connecting to an Ethereum node and processing all the L1 transactions associated with StarkNet.

Cairo Contracts

Cairo programs are by default stateless, if we want to write contracts to run on Starknet we need additional context provided by the StarknetOS

We need to add the following to our code to declare it as a contract

```
# Declare this file as a StarkNet contract.
%lang starknet
```

Adding state

To store state within our contract we use a decorator

```
@storage_var
```

We then specify the variable as a function

```
@storage_var
func balance() -> (res : felt){
}
```

The decorator will create the ability for us to read and write to a variable 'balance'
For example

```
balance.read();
```

```
balance.write(1234);
```

You would then write getter and setter functions to interact with the variable as usual.
When the contract is deployed, all the storage is set to 0

Function visibility

Unlike a standalone cairo program, we don't have a main function, instead we can interact with any of the functions in the contract depending on their visibility.

The decorators are
`@external`
`@view`

Both external and view functions can be called from other contracts or externally.
If no decorator is specified, then the function is internal.
Although @view indicates that we are not changing state, this is not currently enforced.

Function implicit arguments

```
@view
func get_score{
    syscall_ptr : felt*,
    pedersen_ptr : HashBuiltin*,
    range_check_ptr,
}() -> (score : felt){
    let (score) = score.read();
    return (score);
}
```

We have an additional implicit argument `syscall_ptr` this allows the OS to handle the storage variables correctly.

Hints

The situation regarding hints is a little more complex, see Hints
You generally do not use hints in your contracts due to security concerns (the user and the operator running the code are likely to be different).
You may see hints in libraries however, and there is a whitelisting mechanism to ensure security.

Constructors

Constructors work in a similar way to Solidity constructors.

- It must be called `constructor`
- It is decorated with `@constructor`
- It is run once, only during deployment.

For example

```
@constructor
func constructor{
    syscall_ptr : felt*,
    pedersen_ptr : HashBuiltin*,
    range_check_ptr,
}(owner_address : felt){
    owner.write(value=owner_address);
    return ();
}
```

More complex storage variables

We can create the equivalent of a Solidity mapping as follows

```
@storage_var
func balance(user : felt) -> (res : felt){
}
```

The corresponding read and write functions have the following signatures

```
func read{
        syscall_ptr : felt*, range_check_ptr,
        pedersen_ptr : HashBuiltin*}(
    user : felt) -> (res : felt)

func write{
        syscall_ptr : felt*, range_check_ptr,
        pedersen_ptr : HashBuiltin*}(
    user : felt, value : felt)
```

Getting the user address

The common.syscalls library allows us to get the address of the user calling a function

```
from starkware.starknet.common.syscalls import get_caller_address

# ...

let (caller_address) = get_caller_address();
```

For reference this is the default contract created by protostar

```
%lang starknet

from starkware.cairo.common.math import assert_nn
from starkware.cairo.common.cairo_builtins import HashBuiltin

@storage_var
func balance() -> (res : felt){
}


@external
```

```
func increase_balance{syscall_ptr : felt*,
pedersen_ptr : HashBuiltin*, range_check_ptr}(

amount : felt){
with_attr error_message("Amount must be positive. Got: {amount}."){

        assert_nn(amount);

        }

let (res) = balance.read();
balance.write(res + amount);
return ();
}



@view
func get_balance{syscall_ptr : felt*,
pedersen_ptr : HashBuiltin*, range_check_ptr}() -> (
res : felt){

let (res) = balance.read();
return (res);

}

@constructor
func constructor{syscall_ptr : felt*,
pedersen_ptr : HashBuiltin*, range_check_ptr}(){
        balance.write(0);
        return ();
}
```

Events in Starknet Contracts

See docs

The mechanism is similar to that on Ethereum.

A contract may emit events throughout its execution. Each event contains the following fields:

- `from_address` : address of the contract emitting the events
- `keys` : a list of field elements
- `data` : a list of field elements

The keys can be used for indexing the events, while the data may contain any information that we wish to log (note that we are dealing with two separate lists of possibly varying size, rather than a list of key-value pairs).

Events can be defined in a contract using the `@event` decorator. Once an event `E` has been defined, the compiler automatically adds the function `E.emit()` . The following example illustrates how an event is defined and emitted:

```
@event
func message_received(a : felt, b: felt){
}
```

...

```
message_received.emit(1, 2);
```

The emit function emits an event with a single key, which is an identifier of the event, given by `sn_keccak(event_name)`

Open Zeppelin cairo contracts

See Repo

and [guide] (https://blog.openzeppelin.com/getting-started-with-openzeppelin-contracts-for-cairo/)

In protostar you need to install the contract libraries

```
protostar install https://github.com/OpenZeppelin/cairo-contracts
```

There are some presets that can be used (Presets are pre-written contracts that extend from the library of contracts. They can be deployed as-is or used as templates for customization.)

- Account
- ERC165
- ERC20Mintable
- ERC20Pausable
- ERC20Upgradeable
- ERC20
- ERC721MintableBurnable
- ERC721MintablePausable
- ERC721EnumerableMintableBurnable

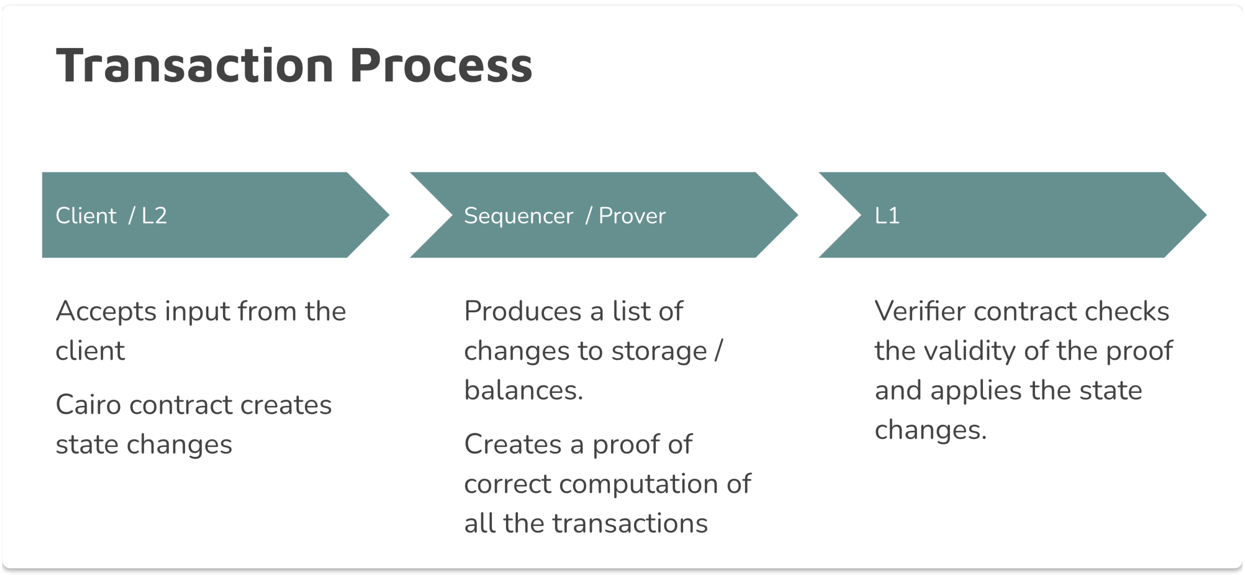
From cairo common library
Signature.cairo
This gives us the function

```
func verify_ecdsa_signature{ecdsa_ptr : SignatureBuiltin*}(

message, public_key, signature_r, signature_s

):
```

Allowing us to verify that the prover knows a signature of the given public_key on the given message.

An example of its use see docs

```
@external
func increase_balance{
    syscall_ptr : felt*,
    pedersen_ptr : HashBuiltin*,
    range_check_ptr,
    ecdsa_ptr : SignatureBuiltin*,
}(user : felt, amount : felt, sig : (felt, felt)){
    # Compute the hash of the message.
    # The hash of (x, 0) is equivalent to the hash of (x).
    let (amount_hash) = hash2{hash_ptr=pedersen_ptr}(amount, 0)

    # Verify the user's signature.
    verify_ecdsa_signature(
        message=amount_hash,
        public_key=user,
        signature_r=sig[0],
        signature_s=sig[1],
    )
```



Transaction Process

Client / L2

Accepts input from the client

Cairo contract creates state changes

Sequencer / Prover

Produces a list of changes to storage / balances.

Creates a proof of correct computation of all the transactions

L1

Verifier contract checks the validity of the proof and applies the state changes.

Data Availability

Currently Starknet is operating in zkrollup mode, this means that upon the acceptance of a state update on-chain, the state diff between the previous and new state is sent as calldata to Ethereum.

This data allows anyone that observes Ethereum to reconstruct the current state of StarkNet.

There are other modes possible :

In ZK-Rollup mode data is published on-chain.
In Validium mode data is stored off-chain.
Volition is a hybrid data availability mode, where the user can choose whether to place data on-chain or off-chain.

Starknet Fee Mechanism

See docs

Users can specify the maximum fee that they are willing to pay for a transaction via the `max_fee` field.

The only limitation on the sequencer (enforced by the StarkNet OS) is that the actual fee charged is bounded by `max_fee`, but for now, StarkWare's sequencer will only charge the fee required to cover the proof cost (potentially less than the max fee).

Presently, the sequencer only takes into account L1 costs involving proof submission. There are two components affecting the L1 footprint of a transaction:

- computational complexity: the heavier the transaction, the larger its portion in the proof verification cost.

- on chain data: L1 calldata cost originating from data availability and L2→L1 messages.

The fee is charged atomically with the transaction execution on L2. The StarkNet OS injects a transfer of the fee-related ERC-20, with an amount equal to the fee paid, sender equals to the transaction submitter, and the sequencer as a receiver.

Reverting transactions and a problem

See documention for adding error messages
For example

```
with_attr error_message("ERC20: decimals exceed 2^8"):
assert_lt(decimals, UINT8_MAX)
end
```

But handling reverted transactions is being debated

Cairo is not natively capable of proving reverted transactions. The reason is that for a transaction to fail, one needs to prove that the transaction resulted in an error.
On the current version of StarkNet Alpha, either a transaction is valid and can be included in a proof or it is unprovable and cannot be included.
The problem this triggers is that an unprovable transaction cannot pay fees and one can imagine sending expensive computation-wise transactions which consume sequencers' resources without compensating for them. This allows DoS attacks on the sequencers.

Suggested solutions

1. Making All Transactions Provable
   This can be achieved by using a new programming language (whose syntax can be very similar to Cairo), in which the compiled contract will include the code to handle exceptions.
2. Economic Solution - "Red/Green"
   The main idea is that a sequencer can choose to include a transaction without executing it and receive a red fee, thus enabling it to ensure it receives a fee payment for their work. We can design the mechanism in such a way that a Sequencer will always be incentivized to execute a provable transaction and take the green fee instead of the red fee.
3. Transaction level PoW
   Users provide a PoW on top of their transactions based on the numbers of reverted txs. In such a way, a user would have to provide a small proof of work or none on the average case, but in the case too many reversions, the PoW will be much larger.

Account Abstraction

See docs from Nethermind
Also this blog from Ethereum and this blog from Gnosis

Why Account Abstraction matters

The shift from EOAs to smart contract wallets with arbitrary verification logic paves the way for a series of improvements to wallet designs, as well as reducing complexity for end users. Some of the improvements Account Abstraction brings include:

- Paying for transactions in currencies other than ETH
- The ability for third parties to cover transaction fees
- Support for more efficient signature schemes (Schnorr, BLS) as well as quantum-safe ones (Lamport, Winternitz)
- Support for multisig transations
- Support for social recovery

Previous solutions relied on centralized relay services or a steep gas overhead, which inevitably fell on the users' EOA.

EIP-4337 is a collaborative effort between the Ethereum Foundation, OpenGSN, and Nethermind to achieve Account Abstraction in a user-friendly, decentralized way.

Account Abstraction on Starknet

An account is a contract address.
With abstract accounts, it's the who (address) that matters, not the how (signature).

A user, will still have an address that can be shared publicly so that someone can send a token to that address.
The user still has a wallet with private keys that are used to sign transactions.

The difference is that the address will be a contract. The contract can contain any code. Below is an example of a stub contract that could be used as an account. The user deploys this contract and calls `initialize()`, storing the public key that their wallet generated.

```
A minimalist Account contract ###

# This function is called once to set up the account contract.

@external
func initialize(public_key){
    store_public_key(public_key);
    return ();
}

# This function is called for every transaction the user makes.
@external
func execute(destination, transaction_data, signature){
    # Check the signature used matches the one stored.
    check_signature(transaction_data, signature);
    # Increment the transaction counter for safety.
    increase_nonce();
    # Call the specified destination contract.
    call_destination(destination, transaction_data);
    return ();
}
```

Then to transfer a token, they put together the details (token quantity, recipient) and sign the message with their wallet. Then `execute()` is called, passing the signed message and intended destination (the address of the token contract).

A user can define what they want their account to "be". For many users, an account contract will perform a signature check and then call the destination.

However, the contract may do anything:

- Check multiple signatures.
- Receive and swap a stable coin before paying for the transaction fee.
- Only agree to pay for the transaction if a trade was profitable.
- Use funds from a mixer to pay for the transaction, separating the deposit from withdrawl.
- Send a transaction on behalf of itself - a DAO.

If you wish to explore this more, this is a useful workshop explaining account abstraction.

Cairo Contract Tools

Developing contracts - tools

An equivalent to the Cairo playground is the Starknet Playground
https://starknet.io/playground/?lesson=starknet_contract

(This is not fully working at the moment)

Protostar

Testing in protostar

See previous notes, in addition the following flags may be useful

```
protostar test test/test_erc20.cairo --disable-hint-validation
protostar test test/test_ex1.cairo --stdout-on-success
```

Cheatcodes

See documentation

There are a number of keywords to add functionality to your tests, such as mocking, deploying contracts. We will cover these in more detail later.