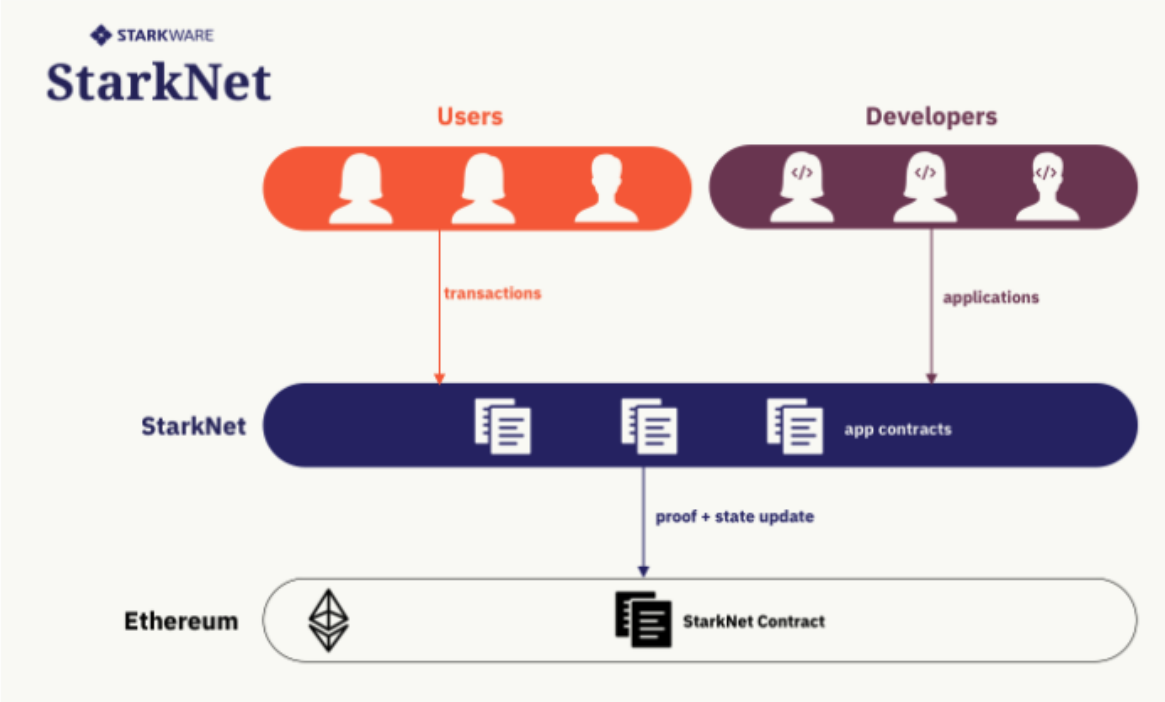
Lesson 8 Cairo and Warp

Cairo Language continued

Recap
Cairo is a language for writing provable programs: running a Cairo program produces a *trace* that can then be sent to a trustless prover, which generates a STARK *proof* for the validity of the statement or computation represented by the Cairo program. The proof can then be verified using a *verifier* (which may or may not be on-chain).



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.

STARK Overview

ZK-STARKs (Zero-Knowledge Scalable Transparent ARguments of Knowledge)

Original Paper for reference

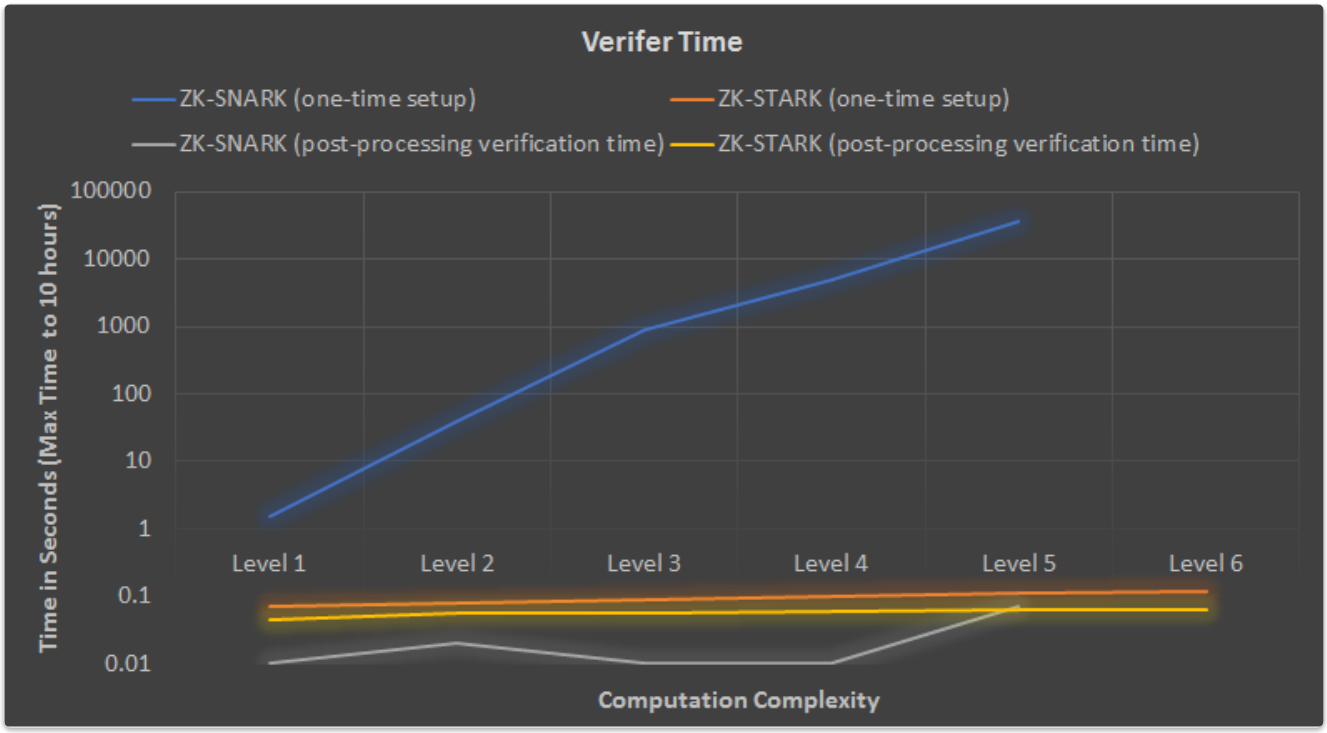
Differences between ZK-SNARKs and ZK-STARKs

1. ZK-SNARKs require a trusted setup phase whereas ZK-STARKs use publicly verifiable randomness to create trustlessly verifiable computation systems.
2. ZK-STARKs are more scalable in terms of computational speed and size when compared to ZK-SNARKs.
3. ZK-SNARKs are vulnerable to attacks from quantum computers due to the cryptography they use. ZK-STARKs are currently quantum-resistant.

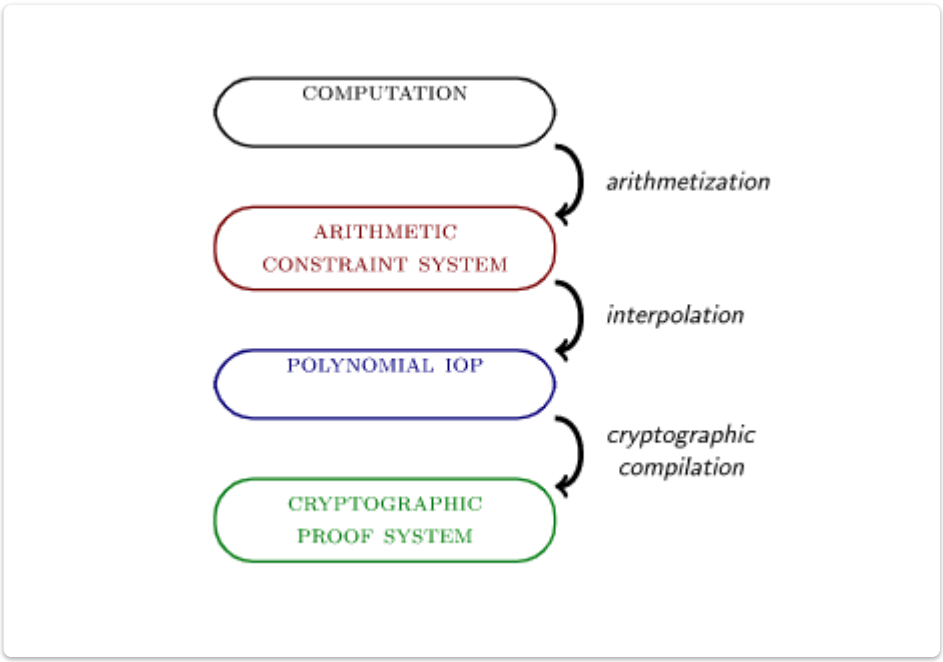
The performance of STARKS is on the whole worse than SNARKS

From Overview

	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 😒



They have a similar process of transformations that SNARKS have

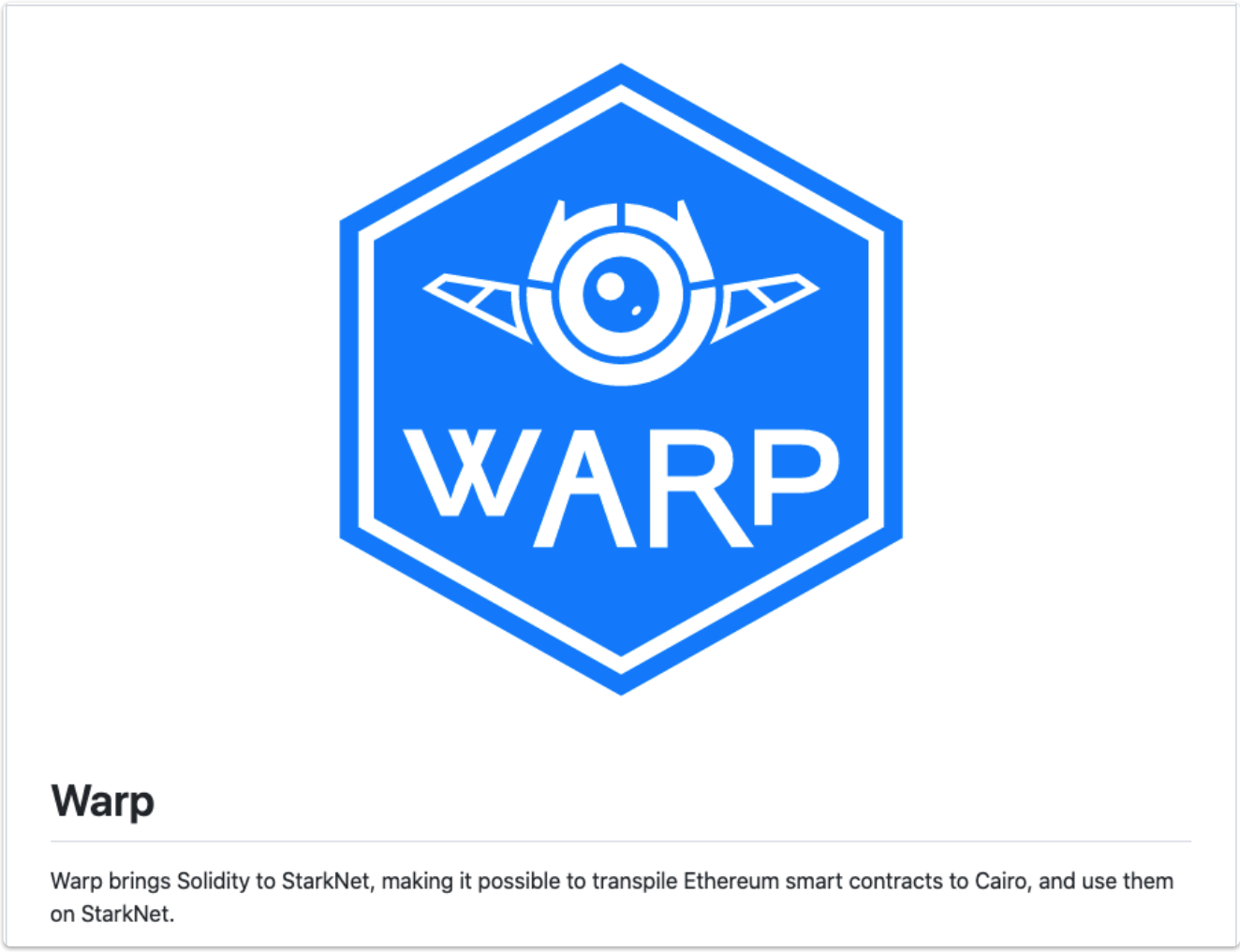


Steps :

1. Computation - for example our Cairo program
2. Arithmetization and Arithmetic Constraint System
   1. The output is an arithmetic constraint system, essentially a bunch of equations with coefficients and variables taking values from the finite field.
   There are 2 types of constraints
   - Boundary constraints*: at the start or at the end of the computation an indicated register has a given value.*
   -Transition constraints: any two consecutive state tuples evolved in accordance with the state transition function.
   Collectively, these constraints are known as the algebraic intermediate representation, or AIR

3. Interpolation and Polynomial IOPs
   interpolation means finding a representation of the arithmetic constraint system in terms of polynomials. The resulting object is not an arithmetic constraint system but an abstract protocol called a Polynomial IOP.
4. Cryptographic Compilation with FRI (Fast Reed-Solomon IOP of Proximity)
   FRI is a key component of a STARK proof that achieves this task by using Merkle trees of Reed-Solomon Codewords to prove the boundedness of a polynomial's degree.

WARP



Warp allows you transpile Solidity contracts into Cairo

Process

See details in their repo

The process you need to go through is

1. Setup the dependencies and install warp
2. Transpile the contract with
   ```js
   ```

```
warp transpile FILE_PATH CONTRACT_NAME
```

        For example
        ```
        warp transpile test.sol test
        ```

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
```

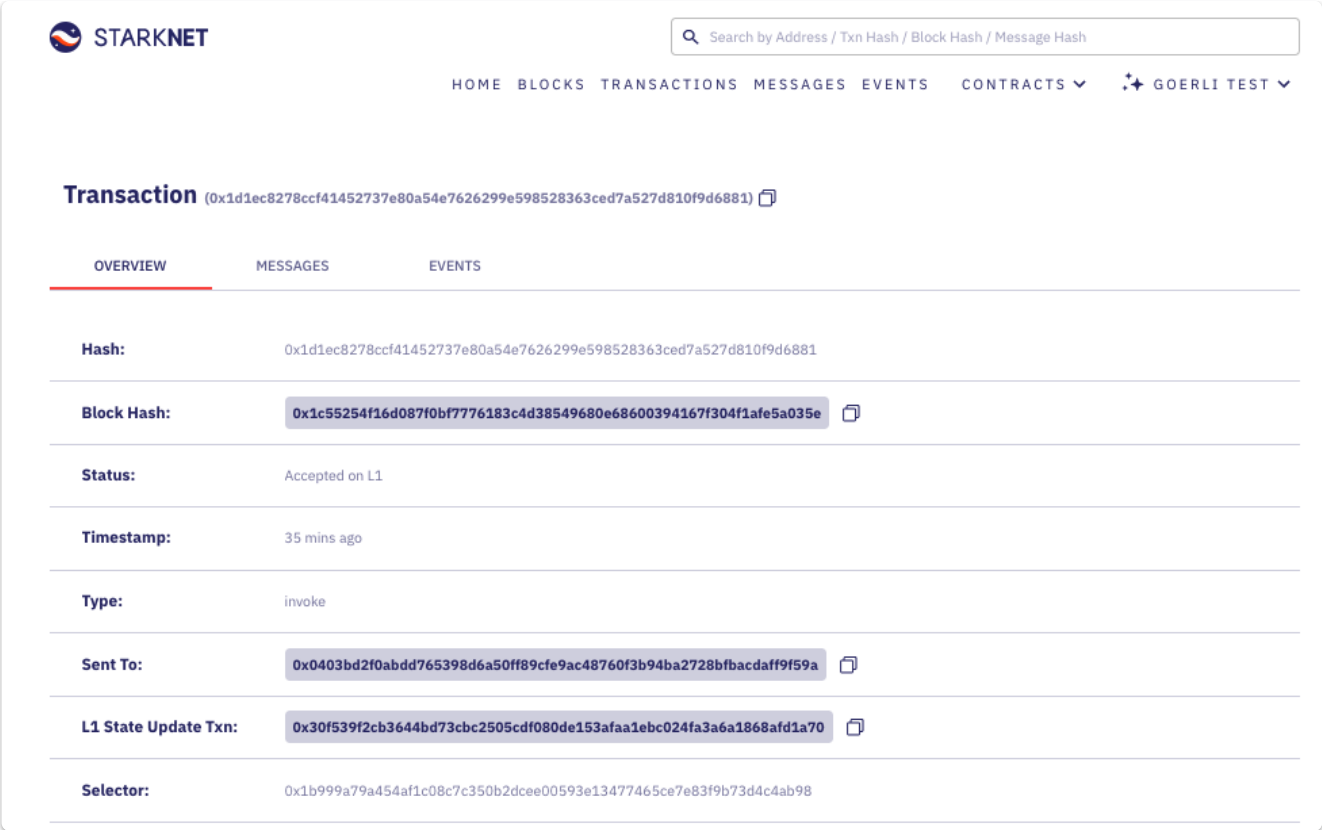
```
warp invoke --program test.json --address
0x0403bd2f0abdd765398d6a50ff89cfe9ac48760f3b94ba2728bfbacdaff9f59a  --network
alpha-goerli --function store --inputs [13]
```

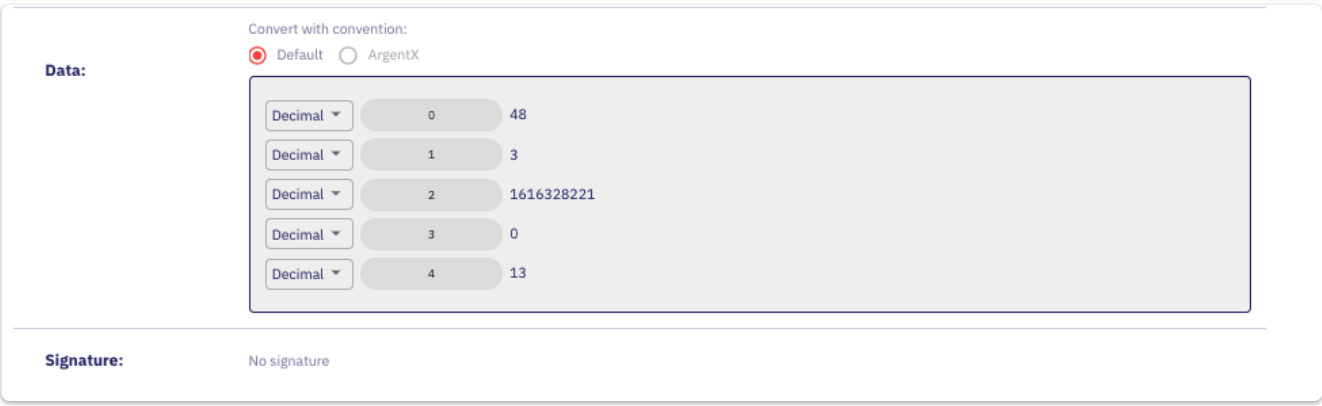
```
Invoke transaction was sent.
Contract address:
0x0403bd2f0abdd765398d6a50ff89cfe9ac48760f3b94ba2728bfbacdaff9f59a
Transaction hash: 0x1d1ec8278ccf41452737e80a54e7626299e598528363ced7a527d810f9d6881
```

```
warp status 0x1d1ec8278ccf41452737e80a54e7626299e598528363ced7a527d810f9d6881 --
network alpha-goerli
```

```
{
        "block_hash":
"0x1c55254f16d087f0bf7776183c4d38549680e68600394167f304f1afe5a035e",
        "tx_status": "ACCEPTED_ON_L1"
}
```

You should be able to find the contract and transactions on the block explorer





TRANSACTION STATUS – TRANSACTION LIFECYCLE

1. `NOT_RECEIVED`
    - Transaction is not yet received (i.e., not written to storage.)

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`

Limitations

Solidity Constructs Currently Not Supported

Support Status	Symbol
Will likely never be supported	❌
Support will land soon	⚒
Will be supported in the future	❗
Currently Unkown	❓

Solidity	Support Status
try/catch	❓
msg.value	❌
tx.origin	❗
tx.gasprice	❓
block.basefee	❌
block.chainid	❗
block.coinbase	❓
block.difficulty	❌
block.gaslimit	❓
gasleft()	❓
functions as data	❌
precompiles	❗
create/create2	❗
Selfdestruct	❌
BlockHash	❗

Yul	Support Status
linkersymbol	❓
codeCopy	❓
codeSize	❓

Run `solc --optimize --ir-optimized <file>` to see if your Solidity results in any of these YUL constructs.

STARKNET PROJECTS ARE VERY ACTVE

Project to make output more readable
https://github.com/kootsZhin/warp-to-cairo

Rust libraries to interact with starknet which now have support for Web Assembly
https://github.com/xJonathanLEI/starknet-rs

Layer Swap, allowing transfer of funds from CEXs to Starknet
https://twitter.com/i/status/1500905618982555660

Pooled Withdrawl Scheme
https://github.com/agolajko/pooledwithdrawal

Other Resources

Details about EVM as against Cairo
https://medium.com/nethermind-eth/field-elements-all-the-way-down-59f02387d1d6