Cairo, Account Abstraction and other superpowers

Uday Jhunjhunwala
Twitter: @udayj

Before we start

ZKX is a permissionless protocol for derivatives built on StarkNet, with a decentralised order book and a unique way to offer complex financial instruments as swaps.

Trustless, **Permissionless**, and **Borderless**.

Prerequisites

This presentation assumes familiarity with

Basic blockchain concepts

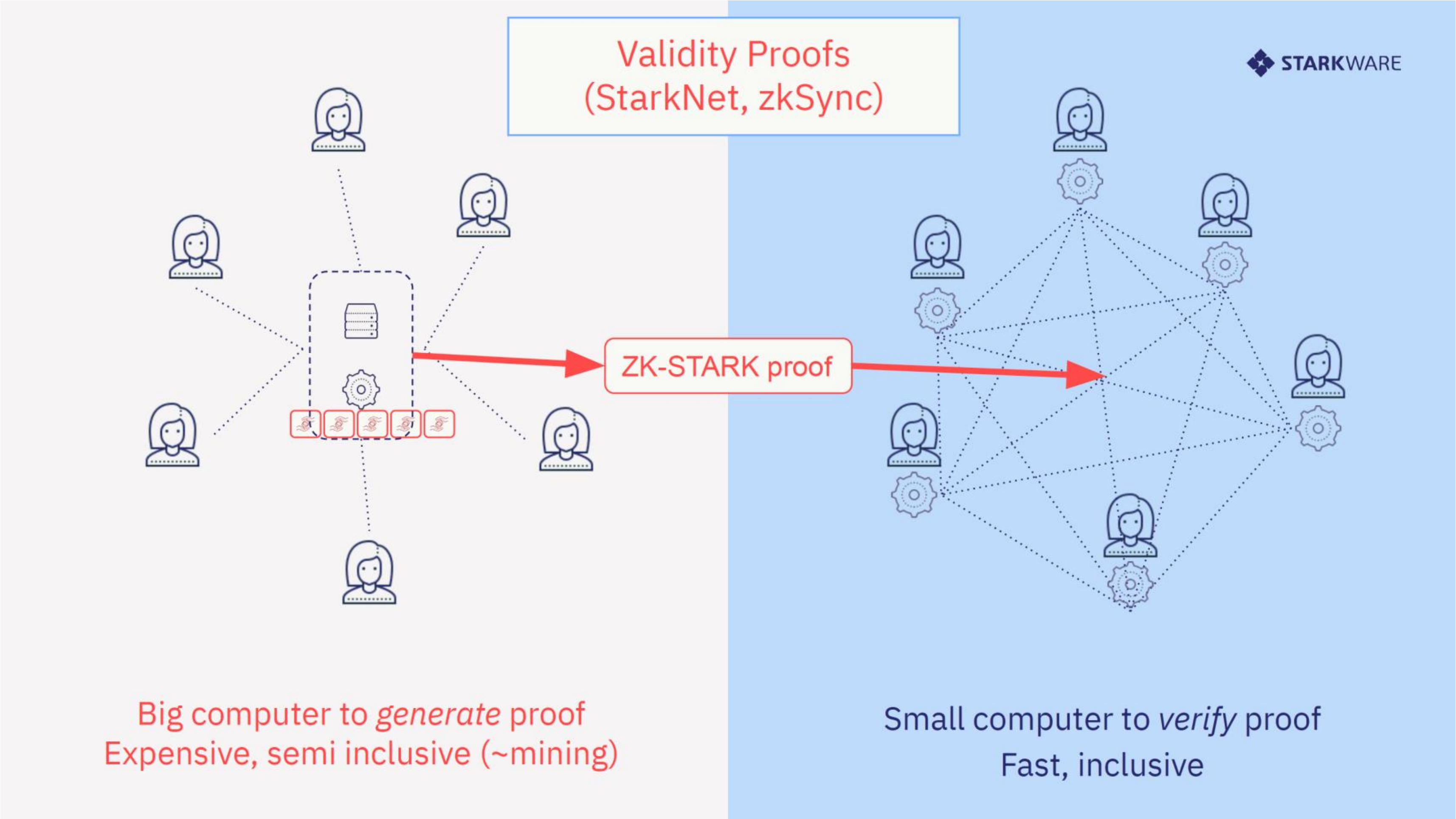
Ethereum as blockchain and world computer

Scalability & the need for rollups

Basic understanding of smart contracts programming

Why Cairo ?

Scaling with inclusive accountability



What do we need ?

We need a way to scale. We need a way to distribute transaction costs amongst many transactions. Starknet performs many transactions outside of Ethereum and sends proof of correct execution to Ethereum for verification. This is known as validity proof.

Trick – Proving takes almost same time as performing the computations but verification is exponentially cheaper.

What is proving of computational integrity

Starknet uses the STARK proof system. It is based on AIR (Algebraic Intermediate Representation). Basically, here computation is expressed as a system of polynomial equations which satisfies certain constraints.

Verifier written in solidity on Ethereum will take a proof and check that the computation it describes is valid.

A compiler that converts computation to such lists of polynomials would help or might not ?

Approach 1 - ASIC

We need a way to convert a program describing a computation to an AIR – which is needed by STARK proof system.

We write a compiler that takes in the computation and outputs an AIR.

This is like an Application Specific Integrated Chip. It is efficient but the set of constraints can vary widely for different computation being converted. Verifier has to be implemented per application to check the validity proof for each AIR. And building recursive proofs becomes difficult.

Approach 2 - CPU

Here we design a single AIR that behaves like a CPU. This AIR represents a single computation – the loop of fetching an instruction from memory and executing that instruction.

This is similar to a general purpose CPU chip rather than an ASIC

- Fixed constraint set
- Single AIR – auditing a new application means just auditing its code in CAIRO
- Single Verifier
- Easier recursive proofs

Noteworthy Features in Cairo

Cairo helps us write arbitrary computation that is ultimately converted to a format suitable for creating validity proofs.

- Modular Arithmetic
- Non-deterministic computation
- Continuous Memory
- AP, FP and PC – memory registers

The learning curve - quirks of Cairo

Absence of boolean expressions

Solidity: *if (x == 2 && y == 2) { … }*

In Cairo, this is not possible!

But there are workarounds like the following:

*assert (x - 1) * (y - 1) = 0 can be used for assert x || y*

*assert x * y = 0 can be used for assert !x || !y*

assert x + y = 2 can be used for assert x && y

Segments

Memory is treated as a list of continuous segments.

Address of a memory cell within a segment is referenced using relocatable values
<segment>:<offset>

AP, FP start in execution segment
PC starts in program segment

Use –print_segments as a compiler option

Builtins

Adding a new instruction to the instruction set is expensive.

Builtins support pre-defined tasks as execution units within Cairo.
Uses memory-mapped I/O.

For e.g. for range-check, write values in memory segment for range-check builtin

Overall cheaper than doing the same thing using pure cairo code.

Starknet and Cairo are amazing!

- The community – join Starknet discord
- The amount of documentation, tutorials, sample code – awesome cairo on github
- Great frameworks for development – Nile, Protostar
- Workshops and hackathons
- Cairo just keeps getting better! – more developer friendly
- No need to mention again, but transactions are cheap!

Account Abstraction

What is account abstraction ?

Ethereum – accounts represented by key-pair (private key signs the transactions)
All user accounts are same – contract accounts are separate.

On Starknet

- Account is a smart contract
- Address is not derived from the Signer
- Can have multiple signatures
- Signatures have to be validated by account – different schemes
- Can use multicalls

How does it work ?

Account (and its rights) are separated from the signer (authorizer)

Interface –

func __validate__
func __execute__

Arbitrary logic inside the validate method – subject to constraints
Enables sequencers to charge fees for reverted transactions
You can support different authentication schemes – even your hardware authenticator

Internals of an Account

getPublicKey – view function
setPublicKey – callable by owner or self
isValidSignature
__validate__
__execute__

Storage holds public key but can be extended to hold an array of keys
Validate function can have arbitrary logic but only access own storage

contract_address := pedersen(“STARKNET_CONTRACT_ADDRESS”, caller_address, salt, pedersen(contract_code), pedersen(constructor_calldata))

Some use cases

It’s a game changing feature for Starknet – and one of the most exciting in the community

- Social Recovery
- Different signing scheme than ECDSA (or a different curve rather than Secp256k1)
- Multiple signatures
- Session Keys
- NFT owned accounts
- Self-Custodial vaults

Session keys

Typically single use / restricted use allowance for dapps

You can give a Dapp the permission to send transactions through your account for a specified duration, for certain kinds of actions and within certain limits (amount spent, no. of transactions sent etc.)

Being used in on-chain gaming on Starknet by various teams

NFT owned accounts

The smart contract representing an Account is free to decide what transaction is valid.

It could then just decide if the sender of a transaction holds a particular NFT then treat it like the owner or co-owner.

You could create a gaming entity and sell the rights to your online avatar by simply transferring the NFT without giving your private keys.

New owner of NFT = New owner of Account

Its essentially the same as rotating the keys but different semantics

Social Recovery

You could have a second key-pair act like a co-owner.

If you lose keys, the co-owner can send transactions or assign primary ownership after a certain wait-period.

Helps with account recovery. Argent-X has a version of this available

Self-custodial vaults

Give limited use co-ownership of assets in an account to holder of a key-pair.

This owner can issue transactions to invest on your behalf into yield bearing protocols. All assets still belong to your account even though someone else is investing for you.

Self-custodial vault strategies.

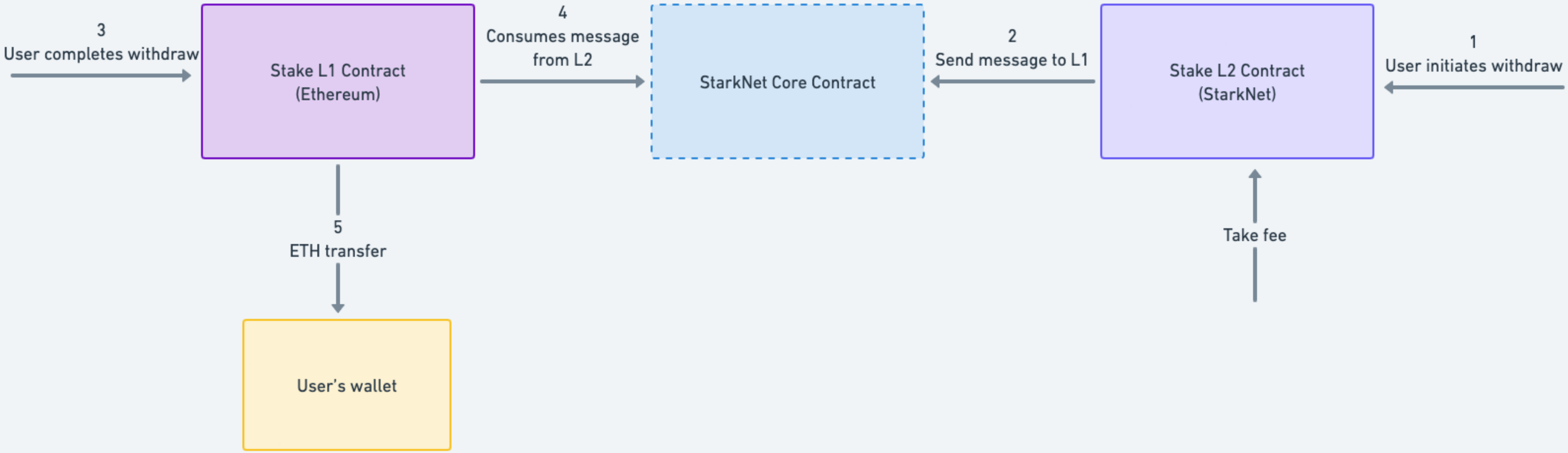
L1-L2 communication

One important property of a good L2 system is the ability to interact with the L1 system it’s built on.

Every StarkNet contract may send and receive messages to/from any L1 contract.

Stake ETH (L1 → L2 message)

Withdraw (L2 → L1 message)



Stake L1 contract

- function stake() - payable
- function withdraw(uint256 amount)

Stake L2 contract

- func deposit(from_address, user_l1_address, amount) - l1 handler
- func withdraw(user_l1_address, amount)
- func take_fee(user_l1_address)
- func get_balance(user_l1_address) - view function

L1 -> L2 messaging

- L1 contract can initiate a message to L2 by calling sendMessageToL2 function on the StarkNet core contract.

- The arguments for this function are:
  - toAddress
  - selector
  - payload

- This invokes function annotated with l1_handler decorator on the target contract.

L2 -> L1 messaging

- L2 contract can send a message to L1 by calling send_message_to_l1 syscall.

- The arguments for this function are:
  - to_address
  - payload_size
  - payload

- After the state update that included this transaction is proved and L1 state is updated, hash of this message is stored on StarkNet core contract.

L2 -> L1 messaging

- Recipient address on L1 can consume this message by providing the message parameters by calling consumeMessageFromL2 function on the StarkNet core contract.

- Arguments needed are:
  - fromAddress
  - payload

Contracts and Functions

L1

```
///////////////////
/// Constructor ///
///////////////////

/// @param starknetCore_ StarknetCore contract address used for L1-to-L2 messaging
/// @param stakeL2Address_ L2 Stake Contract address
constructor(
    IStarknetCore starknetCore_,
    uint256 stakeL2Address_
) {
    require(address(starknetCore_) != address(0));

    starknetCore = starknetCore_;
    stakeL2Address = stakeL2Address_;
}
```

L1

```
/// @dev function to deposit ETH to Stake contract
function stake()
    external
    payable
{

    uint256 senderAsUint256 = uint256(uint160(msg.sender));

    uint256[] memory payload = new uint256[](2);
    payload[0] = senderAsUint256;
    payload[1] = msg.value;

    starknetCore.sendMessageToL2(
        stakeL2Address,
        DEPOSIT_SELECTOR,
        payload
    );
}
```

L2

```
##############
# L1 Handler #
##############

# @notice Function to handle deposit from Stake L1 contract
# @param from_address_ - The address from where deposit function is called
# @param user - User's l1 address
# @param amount - The Amount of funds that user wants to deposit
@l1_handler
func deposit{syscall_ptr : felt*, pedersen_ptr : HashBuiltin*, range_check_ptr}(
    from_address : felt, user_l1_address_ : felt, amount_ : felt
):
    let (l1_address) = stake_l1_address.read()
    assert from_address = l1_address

    let (current_balance) = user_balance.read(user_l1_address_)
    user_balance.write(user_l1_address=user_l1_address_, value=current_balance + amount_)

    return ()
end
```

L2

```
# @notice Reduce a constant amount from user's balance
# @param user_l1_address_ - l1 address of user
@external
func take_fee{syscall_ptr : felt*, pedersen_ptr : HashBuiltin*, range_check_ptr}(
    user_l1_address_ : felt
):
    let (balance) = user_balance.read(user_l1_address_)

    assert_le(FEE, balance)

    user_balance.write(user_l1_address=user_l1_address_, value=balance - FEE)

    return ()
end
```

L2

```
# @notice Withdraw amount from this contract
# @param user_l1_address_ - l1 address of user
# @param amount_ - amount to be withdrawn
@external
func withdraw{syscall_ptr : felt*, pedersen_ptr : HashBuiltin*, range_check_ptr}(
    user_l1_address_ : felt, amount_ : felt
):
    let (balance) = user_balance.read(user_l1_address_)

    assert_le(amount_, balance)

    user_balance.write(user_l1_address=user_l1_address_, value=balance - amount_)

    let (l1_address) = stake_l1_address.read()

    # Send the withdrawal message.
    let (message_payload : felt*) = alloc()
    assert message_payload[0] = MESSAGE_WITHDRAW
    assert message_payload[1] = user_l1_address_
    assert message_payload[2] = amount_

    # Send Message to L1
    send_message_to_l1(to_address=l1_address, payload_size=3, payload=message_payload)

    return ()
end
```

L1

```
/// @dev function to withdraw funds from an L2 Account contract
/// @param amount_ - The amount of tokens to be withdrawn
function withdraw(
    uint256 amount_
) external {
    uint256 senderAsUint256 = uint256(uint160(msg.sender));

    uint256[] memory payload = new uint256[](3);
    payload[0] = WITHDRAWAL_INDEX;
    payload[1] = senderAsUint256;
    payload[2] = amount_;

    // Consume call will revert if no matching message exists
    starknetCore.consumeMessageFromL2(
        stakeL2Address,
        payload
    );

    payable(msg.sender).transfer(amount_);
}
```

VOYAGER

Search by blocks / transactions / contracts ⌘/

Goerli

Blocks Transactions Contracts More

CONTRACT

0x00ee9d0dd4011bef4215871b35df4478aa6961e2e63383d264b566657481d81a

BALANCE	SOURCE CODE STATUS	TRANSACTIONS	CREATED ON
0.0 ETH	Not Verified	2	Aug 23rd 2022 5:30:00 AM

Transactions 2 Events Messages Code Read Contract Write Contract

MESSAGE HASH	DIRECTION	FROM	TO
0x2eb2fa2cfd362...	L1 → L2	0x8ba72c77b5eb9...	0x00ee9d0dd4011...

Show 10 records

First Page 1 of 1 Last

Message

? MESSAGE HASH:	0×2eb2fa2cfd36279cba12e0a154fd4e7600ea3b3074df299b0fa5eeae3f0f4ddf
? DIRECTION:	From L1
? FROM:	0×8ba72c77b5eb91b62f10d3d019be288c519345b6
? TO:	0×00ee9d0dd4011bef4215871b35df4478aa6961e2e63383d264b566657481d81a

? STATUSES:

Sent	Canceled	Pending	Consumed
0	0	1	0

? PAYLOAD:

```
[0]: 342391293183593373432722476863500651063588264326
[1]: 10000000000000
```

? SELECTOR:	0xc73f681176fc7b3f9693986fd7b14581e8d540519e27400e88b8713932be01

L2 Info

? BLOCK HASH:	0×77385535c4fb92a10a5b24f48fd3d51897ee60a4c25b4813e9a1106fb61d285
? TRANSACTION HASH:	0×72499503b9a3557269db672a184327cc8cbcb3927d485648e5b105a4a1a0738
? TIMESTAMP:	12 mins ago (Aug 23rd 2022 5:34:56 PM)

CONTRACT

0x00ee9d0dd4011bef4215871b35df4478aa6961e2e63383d264b566657481d81a

BALANCE	SOURCE CODE STATUS	TRANSACTIONS	CREATED ON
0.0 ETH	Not Verified	2	Aug 23rd 2022 5:30:00 AM

Transactions 2 | Events | Messages | Code | Read Contract | Write Contract

✓ Connected Braavos: 0x4f0753f8b0afddf449d0c00d7cad63eb2913823f70e667de087160edf2dbd36 Change Provider

The contract ABI is not verified and is not guaranteed to be correct.

1. get_balance

user_l1_address_ (integer or hex, e.g.: 1 or 0x1)

342391293183593373432722476863500651

Response format:

- (●) Decimal
- ( ) Hex
- ( ) Text

Query

↳ balance: 10000000000000

2. get_l1_address

References

- Stake dApp Github - https://github.com/StarkCon/workshop01
- StarkNet documentation - https://cairo-lang.org/docs/hello_starknet/index.html
- L1 <-> L2 communication - https://cairo-lang.org/docs/hello_starknet/l1l2.html
- Cairo Whitepaper - https://eprint.iacr.org/2021/1063.pdf
- Account Abstraction - https://www.argent.xyz/blog/wtf-is-account-abstraction/

Thank you!

Any questions?
E-Mail: priority@zkx.fi
Twitter: @zkxprotocol
Website: zkx.fi

zkx.workable.com