ZKU Week 5 Assignment

Email: qadir@xord.com
Discord Id: AbdulQadir#0432

Section A: Theoretical Questions

Part 1: Scalability

Question 1:

Currently, there are a lot of projects working on the blockchain scaling solutions, or I say smart-contract rollup chains, such as - Arbitrum, Optimistic Ethereum, zkSync 2.0, Polygon Hermez, Cartesi, and StarkNet. We will talk about the ZkSync and Arbitrum comparison and tradeoffs.

1. **Proof System:**

   Arbitrum:

   Arbitrum is an optimistic rollup with a multi-round interactive dispute mechanism for fraud proofs. In Arbitrum, we can batch all the transactions and publish them on the main chain and call data to the main chain to verify. Arbitrum comes with the excellent property of only posting one state proof of many transactions.

   ZkSync:

   ZkSync is a ZK-Rollup. A zero-knowledge proof (snark) is generated for every state transition, which is verified by the main rollup contract deployed on the main chain. Thus, no validator can ever move the system into an incorrect state or take users' funds.

2. Cost Factor:

ZkSync:

Due to zero-knowledge in zkSync, we need to produce a complex cryptographic proof on every transaction to verify state transition, which requires heavy machines and an inherent cost.

Arbitrum:

Arbitrum is cost-effective compared to the zksync as it does not require running complex cryptographic functions and doesn't need to verify them on-chain. Besides this, Arbitrum uses the ArbGas inspired by EIP-1559 to offer users predictable gas fees. You pay gas in ETH.

3. EVM compatibility:

ZkSync:

zkSync 2.0 has the custom VM based on the LLVM. It allows smart-contract deployments in an EVM-compatible environment. ZkSync 2 has also introduced ZkEVM, which allows ethereum smart contracts to be deployed on zkRollUps. ZkEVM testnet is currently also live.

Arbitrum:

Arbitrum, on the other hand, also used its own customized VM named ArbOs. ArbOs provide the functionality to emulate the execution of an Ethereum-compatible chain. It tracks accounts, executes contract code, and handles the details of creating and running EVM code. Clients can submit EVM transactions, including those that deploy contracts, and ArbOS ensures that the submitted transactions run in a compatible manner.

4. Instant Withdrawals:

Arbitrum:

Due to the use of the validity proof, Arbitrum has a long period of 7 days withdrawal period; however, multiple projects like hop,connext, celer, etc. are working on mitigating it using different architecture.

Zksync:

ZkSync uses advanced aggregation techniques and zero-knowledge proof to give instant withdrawals.

5. Privacy:

Arbitrum:

As Arbitrum is based on optimistic rollups, it can support any privacy solution available on the L2 Ethereum. And if we build any privacy solution over the arbitrum, it will be termed the L3, which renders the utility of privacy very low. To add some privacy like zcash, we need to build an extra layer on the top of the Arbitrum.

ZKSync:

We can achieve real privacy as the system supports privacy by default. ZkSync can support any confidential transaction system like z-cash without building any layer on the top level.

6. Latency:

Arbitrum:

Currently, The Arbitrum batch of transactions is considered sage within a 1-2 week fraud-proof challenge window. Due to which transactions are not considered final, the user can not withdraw his funds after a week. No instant withdrawal, which impacts the latency issue in the system.

Time-to-finality (under PoW): 2 weeks.
Time-to-finality (under PoS): 1 week.

ZkSync:

Currently, ZKPs are quite computationally intense. For a block of 1000 transactions, we can have a 20-minute proof generation time on ordinary server hardware.

Time-to-finality (now): 20 min.
Time-to-finality (future): under 1 min.

Question 2:

ZkSync and StarkNet, The two of the most promising zkRollups currently available with differing strategies.

Differences:

Name	zksync	StarkNet
EVM compatible and language support	Yes, ZkSync is EVM compatible and has its language named zinc.	StarkNet is not EVM compatible. Using its language Cairo they are working on translating Cairo to Solidity.
ZK/validity proofs	ZkSync uses Zk-Snarks as proof construction.	StarkNet uses the Zk-Starks for the proof system. A side benefit of the k-Starks is that they are quantum secure.
Data Availability problem-solving	ZkSync is currently working on the ZkPorter to resolve the problem of data availability. ZkPorter is a fragmented infrastructure that works seamlessly and parallels with zkSync's zk-rollup solution.	StarNet uses the volition system to solve the Data Availability problem. Volition allows end-users to choose between the rollup scheme (data available on the chain) and the validium scheme (data availability off the chain) for each transaction.
Commitment Schemes	Kate Commitments	FRI commitments

Zk-Rollup War:

STARKS technology is relatively new and not as mature as SNARKS. It is difficult to make it compatible with the EVM. However, starkWave has created a specific Programming language Cairo to run autonomous programs supported by STARKs. They are also working alongside the

teams to convert Solidity smart contracts for the deployment on the EVM-based chains. Altho on the Zk-Snarks uses the trusted setup while the Zk-Starks use the transparent setup. Also, ZkStarks is quantum secure by using hash functions compared to the Zk-Snarks.

Zk-Snarks will most likely win the ZKRollups due to no factors, including the EVM compatibility, trusted setups, language support, and maturity. Yes, it is not quantum secured, but we are currently looking into the scalability problem that quantum secure can cater to later.

Part 2: Interoperability

Question 1:

Interoperability between the different blockchains is one of the major problems people are trying to solve. One way to do this is by creating trustless cross-chain bridges. Token transfer bridges are the most common these days to transfer funds from one blockchain to another.

Here is the list of components needed to create a Proof of Authority-based bridge, assuming both blockchains are EVMased and the bridge is used for the token transfers.

1. **Smart Contracts:**

   Simple Erc20 token contract on the main chain and Erc20 burnable contract on the destination chain. Burn and Mint can only be one by the bridge modifier.

2. **Web App:**

   Frontend that users will interact with the smart contract and lock their funds on the main chain and claim tokens on the destination chain.

3. **Backend Job:**

   we also need a process of listening to tokens received in the bridge wallet and minting/burning on the destination chain. This cause some privacy issues as the servers store the bridge wallet's private key, which is used to sendMint and burnFrom. In case of a leak, the attacker can create tokens out of thin air.

Question 2:

Instead of using the balances, Aztec uses the UTXO model similar to the bitcoin to shield both the native assets and assets like the Erc20 token. The protocol uses Notes (UTXOs). A note encrypts a number that represents a value like the number of tokens the user holds. Each Note has an owner, defined via an Ethereum address. To *spend* a Note, the owner must provide a valid ECDSA signature.

Aztec also has a Note Registry that manages the state of notes for any given asset. The user's balance of any AZTEC asset is made up of the sum of all of the valid notes their address owns in a given Note Registry. The Notes model achieves privacy for native and non-native tokens in the blockchain, and users can shield transactions using the Notes model. Users can transfer the tokens and use defi protocols without revealing their identity.

Aztec allows the user to deposit the funds on layer 1, and then it mints the new set of Notes for the user once the amount is deposited. The user can then use those notes by verifying the ECDSA signature signed to it and withdrawing funds or using the defi protocols without revealing the identity. Under the hood, Aztec uses the SDKs to pick the right note from the notes registry for the transaction. Hence all aspects of a transaction remain hidden from the public or third parties.

Reference:

FAQ - AZTEC documentation.
https://aztec-protocol.gitbook.io/aztec-documentation/support/archive

Part 3: Final Project

ZK PoolTogether

Introduction:

This document explains the working of the ZK PoolTogether project. The zkp version of the original application PoolTogether builds on Ethereum, avalanche, and polygon.

This document will include the working of PoolTogether, the importance of Zk in PoolTogether from a holistic perspective, and a Comparison to similar projects. It will explain the components of the Zk PoolTogether in detail.

What circuits will it include? What will be the role of the smart contract? And most importantly, what is the scope of the project?

PoolTogether:

PoolTogether is a crypto-powered savings protocol based on Premium Bonds. Save money and have a chance to win every week. In PoolTogether, the user deposits funds in the pool and has a chance to win a prize awarded every day. If the user doesn't win, he can withdraw the deposited money.

PoolTogether User Journey:

- Users can connect to which blockchain they want to deposit.
- Then, users can deposit USDC to the pool, representing the share of your token in the pool.
- The amount can be arbitrary.
- Amounts are then invested in strategies to gain yield.
- After every 24 hours, the VRF function from chainlink picks a random user who wins the lottery price.
- The winner can claim the winning price within 60 days.

The overview of the work looks like this, but there are some minor details as well, such as pool sizes, etc., which relate to the amount you have deposited and the average time you have deposited.

But we won’t be diving deep into those things.

Scope of zkPoolTogether [Advantages]:

ZkPoolTogether is the zkp version of PoolTogether; usage of zero-knowledge proof systems will ***protect users' identity, and the amount users are depositing***. For example, users can ***prove that they have deposited X amount in the pool and claim the reward without revealing any knowledge***.

Working of the ZkPoolTogether:

1. Actors:

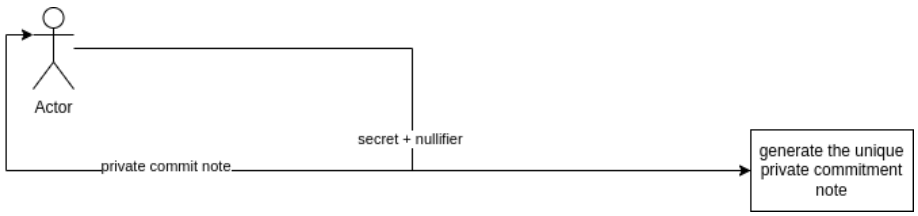
Actors include users involved in the overall process of ZkPoolTogether and entities that interact with the smart contract and Frontend of the system.

- **Depositor:** User that deposits the amount in the pool and creates the commitment hash using the unique secret and nullifier.
- **Relayer Node:** A central server that triggers the yield collection selects the winner through a VRF function in a smart contract.
- **Withdrawer:** User who withdraws the winning amount from the pool by proving the commitment hash.

2. Case Scenarios:

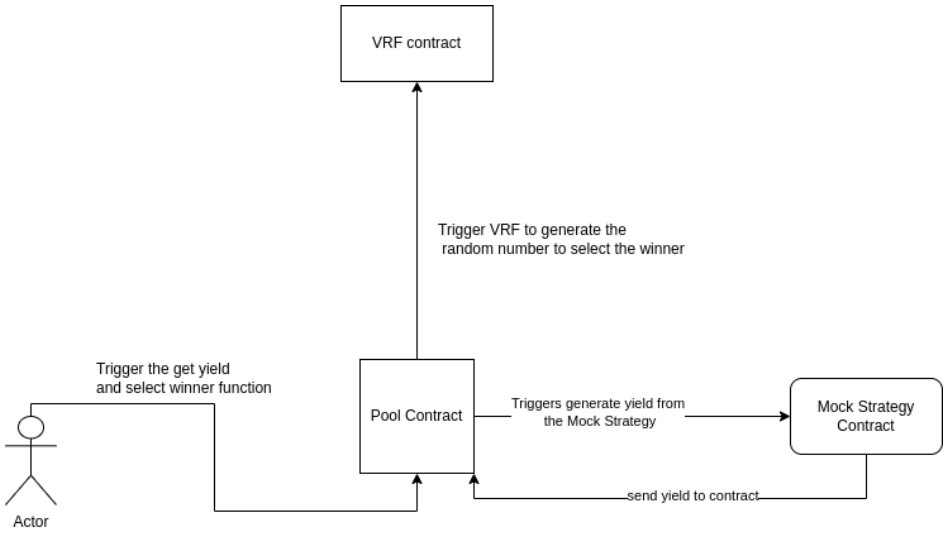
Below are the high-level case scenarios through which actors will pass and interact with the system Frontend, Smart Contract, and circuits.

- **Generating Commitment Hash:**

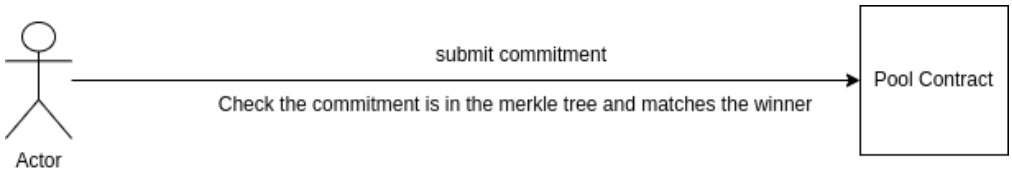


- **Depositing in the Pool:**

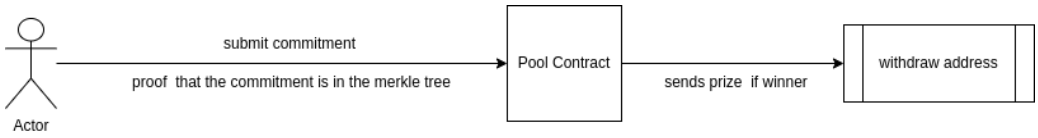
- **Relayer Triggering the get yield and selecting the winner:**



- **Winner checks that he is eligible for the prize:**

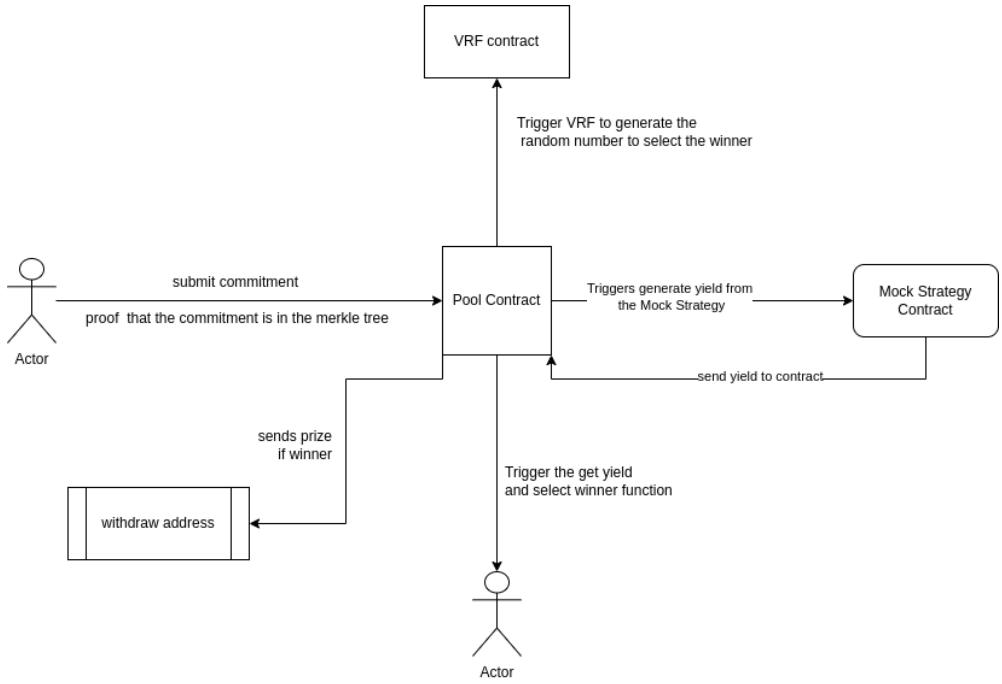


- **Withdraw the prize to another address:**



3. Architecture Overview:

There are five main components of the system: Pool Contract, VRF Contract, Mock Strategy Contract, Circuits, Relayer, and the Frontend (denoted as a user). In the Future, Indexer will also be a part of the architecture.



Protocol:

Contracts:

This section describes the smart contracts and their high-level working and usage in the ZkPoolTogether.

1. Strategy Mock Contract:

Strategy Mock Contract will generate a random yield after a fixed amount of time, sent to the pool contract to give the winning prize to the user who won the lottery.

2. Harmony VRF:

Harmony VRF function will be used here to generate the true randomness to select the winner after the end time of the pool.

Harmony brings the technology of VRF (Verifiable Random Function) natively on a chain to create an optimal solution for the randomness that is unpredictable, unbiased, verifiable, and immediately available.

3. Pool Contract:

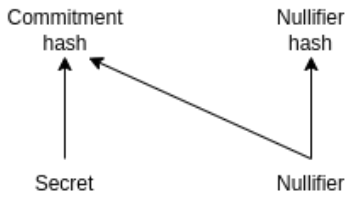
The core contract will be the updated tornado cash classic pool version. This contract allows users to deposit into the current Pool. After a fixed amount of time, a yield generated from the mock strategy contract is taken to the pool contract, and a winner is picked using the VRF function.

The user can also withdraw by proving his commitment exists in the Merkle root and checking whether he is the winner or not to withdraw funds to another address.

4. Merkle Tree History Contract:

When the User deposits an amount on the Pool contract, the user must provide a commitment hash. This commitment hash will be stored in a Merkle tree. Then, when the user withdraws this amount and prize with a different account, you have to provide two zero-knowledge proofs. The first proves that the Merkel tree contains your commitment. This proof is a zero-knowledge proof of a Merkle proof. But this is not enough because the user should be allowed to withdraw the deposited amount and prize only once. Because of this, the user has to provide a unique nullifier for the commitment. The contract stores this nullifier; This ensures that users don't be able to withdraw the deposited money more than one time.

The nullifier is guaranteed by the commitment generation method. The commitment is generated from the nullifier and a secret by hashing. If you change the nullifier, so one nullifier can be used for only one commitment. Because of the one-way nature of hashing, it’s not possible to link the commitment and the nullifier, but we can generate a ZKP.



Relayer:

A center relayer triggers the Pool Contract **EndDraw function** transaction after fixed amounts of time, triggering the GenerateYield on Mock Strategy Contract to create the yield and triggering the VRF contract function to select the random nullifier as a winner.

Circuits:

The circuit defines a set of constraints that must be satisfied to accept a given result in the Smart Contract.

1. Merkle Tree Checker Circuit:

Merkle Tree Checker circuit is used to verify that the given path elements and root of the provided leaf is present in the Merkle tree. This circuit is the same as the tornado cash classic Merkle tree circuit used to prove the commitment is the part of the Merkle tree.

2. Commitment hasher:

It generates the steadfast commitment and nullifier hash using the private secret and nullifier. In this case, a secret and nullifier are not exposed to the external server and smart contract. Then, as I wrote before, the template calculates the nullifier and commitment hash, which is a hash of the nullifier and the secret.

3. Withdraw

It has two public inputs, the Merkle root, and the nullifierHash. The Merkle root is needed to verify the Merkle proof, and the nullifierHash is required by the smart contract to store it. The private input parameters are the nullifier, the secret, and the path elements and pathIndices of the Merkle proof. The circuit checks the nullifier by generating the commitment from it and the secret and matches the given Merkle proof. If everything is fine, the zero-knowledge proof will be generated the TC smart contract can validate that.

Frontend [User Interface]:

Frontend Includes all the user interfaces to interact with the smart contracts and generate the proofs using the wasm and Zkey generated files of the circuits. Following are the components of the frontend user interface:

- Main Page: Display the details and information about the ZkPoolTogether
- App Page: Main Application page allows users to navigate the deposit, withdraw, and prizes pages.
- Deposit Page: Allowing the user to create a commitment and deposit amount in the pool
- Withdraw Page: This allows the user to remove the amount and generate proof of valid commitment.
- Prizes Page: Allows the user to claim the prize by proofing the commitment
- Draw Pages: Display data regarding all the going and past draws of ZkPoolTogether

Indexer:

The indexer will be used to get the real-time data of the draws from the smart contracts and the draw winners on display on the front end.

For Indexer, we will use the Graph, an indexing protocol for querying networks like Ethereum and IPFS. Anyone can build and publish open APIs, called subgraphs, making data easily accessible.

Implementation [Tech Stack]:

As mentioned earlier, we implemented the proposed design of

- Circuits: zkSNARK (Groth16) Circom circuit
- Smart contracts: solidity on-chain registry of processes.
- Foundry: a modular toolkit for Ethereum application development
- Node: Relayer node, typescript based.
- Client lib: typescript library used in the user's browser to create keys,
- NextJs: for the development of the user interface