ZKU Assignment 4

Name: Shyam Patel
Discord: L_guy#4297
Email: shyampkira@gmail.com
GitHub Repo: [Link]

Question 0. Which stream do you choose (answer with A or B)?

I choose stream A but would like to keep the option Stream B open for building a new dApp or porting an existing project. Currently I don't have any ideas for the same.

Question 1. Scaling the future

In the blockchain there is a well known scalability trilemma. We can't have decentralized, secure and scalable L1 blockchain. Many blockchains tend to be secure and decentralized, but they lack scalability.

1. Based on the above, a number of solutions have been proposed to solve this trilemma. Briefly describe the different scalability solutions and write pros and cons of each approach. What was the biggest problem with the Plasma approach?

Small Comparison table among some of the solutions:

	State channels	Side chains	Plasma	Rollups
Decentralization	Low	Moderate	Moderate	High
Security	High	Moderate-Low	Moderate-Low	High
Scalability	High	High	High	Moderate
Generalizability	Low	High	Moderate	High

1. State Channels

State channels allow participants to transact x number of times off-chain while only submitting two on-chain transactions to the Ethereum network. This allows for extremely high transaction throughput.

pros:

- Instant withdrawal/settling on mainnet (if both parties to a channel cooperate).
- Extremely high throughput is possible.
- Lowest cost per transaction - good for streaming micropayments.

cons:

- Time and cost to set up and settle a channel - not so good for occasional one-off transactions between arbitrary users.
- Need to periodically watch the network (liveness requirement) or delegate this responsibility to someone else to ensure the security of your funds.
- Have to lockup funds in open payment channels.
- Don't support open participation.

2. Rollups

A rollup is a type of scaling solution that works by executing transactions outside of Layer 1 but posting transaction data on Layer 1. This allows the rollup to scale the network and still derive its security from the Ethereum consensus.

pros:

They will also enable a new breed of applications that require cheaper transactions and faster confirmation time. All of this while being fully secured by the Ethereum consensus.

cons:

Composability is one of them. In order to compose a transaction that uses multiple protocols, all of them would have to be deployed on the same rollup. Another challenge is fractured liquidity. Lower liquidity usually means higher slippage and worse trade execution.

2.1 OPTIMISTIC ROLLUPS:

Optimistic roll ups kind of run in parallel to the main Ethereum chain on layer 2. They can offer improvements in scalability because they don't do any computation by default. Instead, after a transaction, they propose the new state to mainnet. With Optimistic rollups, transactions are written to the main Ethereum chain as calldata, optimizing them further by reducing the gas cost.

Pros:

- Anything you can do on Ethereum layer 1, you can do with Optimistic rollups as it's EVM and Solidity compatible.
- All transaction data is stored on the layer 1 chain, meaning it's secure and decentralized.

Cons:

- Long wait times for on-chain transactions due to potential fraud challenges.
- An operator can influence transaction ordering.

2.2 ZERO-KNOWLEDGE ROLLUPS:

Zero-knowledge rollups (ZK-rollups) bundle (or "roll-up") hundreds of transfers off-chain and generate a cryptographic proof. These proofs can come in the form of SNARKs or STARKs and get posted to layer 1. The ZK-rollup smart contract maintains the state of all transfers on layer 2, and this state can only be updated with a validity proof. This means that ZK- rollups only need the validity proof instead of all transaction data. With a ZK- rollup, validating a block is quicker and cheaper because less data is included.

pros:

- Faster finality time since the state is instantly verified once the proofs are sent to the main chain.
- Not vulnerable to the economic attacks that Optimistic roll ups can be vulnerable to.
- Secure and decentralized, since the data that is needed to recover the state is stored on the layer 1 chain.

cons:

- Some don't have EVM support.
- Validity proofs are intense to compute – not worth it for applications with little on-chain activity.
- An operator can influence transaction ordering

3. Sidechains:

A sidechain is a separate blockchain which runs in parallel to Mainnet and operates independently. It has its own consensus algorithm (e.g. proof-of-authority, Delegated proof-of-stake, Byzantine fault tolerance). It is connected to Mainnet by a two-way bridge.

pros:

- Supports general computation, EVM compatibility.

cons:

- Less decentralized.
- Uses a separate consensus mechanism.
- Not secured by layer 1 (so technically it's not layer 2).

4. Plasma:

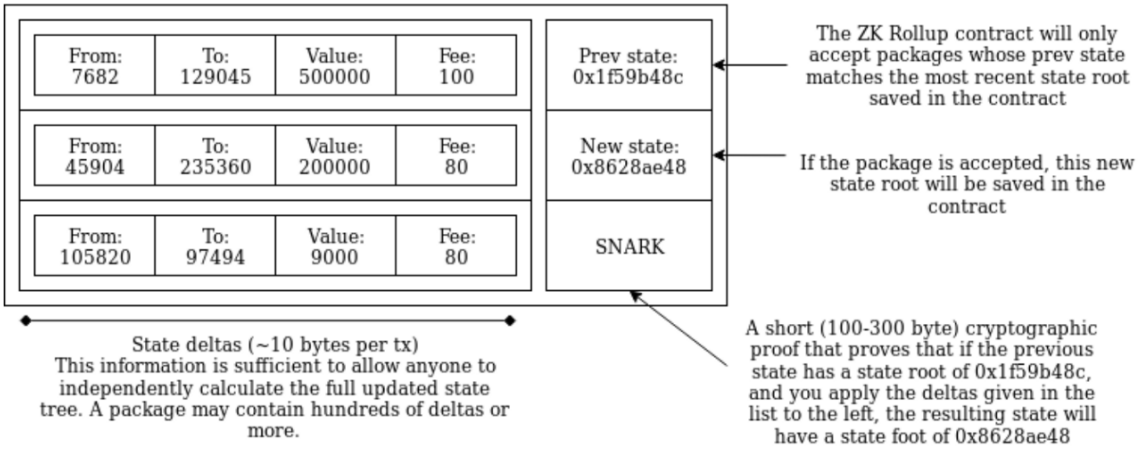
A plasma chain is a separate blockchain that is anchored to the main Ethereum chain, and uses fraud proofs (like optimistic rollups) to arbitrate disputes. These chains are sometimes referred to as "child" chains as they are essentially smaller copies of the Ethereum mainnet. Merkle trees enable creation of a limitless stack of these chains that can work to offload bandwidth from the parent chains (including mainnet). These derive their security through fraud proofs, and each child chain has its own mechanism for block validation.

Challenges:

- Each Plasma chain requires an operator posting the Merkle root commitments to the mainchain. This requires us to trust a third party to accurately post the Merkle root commitments on the chain.
- Plasma requires that owners of transacting assets be present, and it works best for simple transfers.
- Does not support general computation. Only basic token transfers, swaps, and a few other transaction types are supported via predicate logic.
- Withdrawals are delayed by several days to allow for challenges. For fungible assets this can be mitigated by liquidity providers, but there is an associated capital cost.
- Relies on one or more operators to store data and serve it upon request.

2. One of the solutions that has been gaining a lot of traction lately is zkRollups. With the use of a diagram explain the key features of zkRollups. Argue for or against this solution highlighting its benefits or shortcomings with respect to other solutions proposed or in use.

Schema representing Anatomy of ZK rollup:



Anatomy of a ZK Rollup package that is published on-chain. Hundreds of "internal transactions" that affect the state (ie. account balances) of the ZK Rollup system are compressed into a package that contains ~10 bytes per internal transaction that specifies the state transitions, plus a ~100-300 byte SNARK proving that the transitions are all valid.

Basic points for above anatomy:

- A smart contract deployed on Ethereum represents the rollup's on-chain state. It has the balances of accounts present on the L2.
- Transactions' computation is done off-chain. Only the proof and the state delta needs to be submitted to update the state of the smart contract.
- This smart contract on L1 stores the merkle root of its present global state.
- Producing a new block consists in submitting a new batch, containing: state delta, current merkle root to verify that this batch corresponds to the current state of the L2 rollup and a SNARK proof (this zkp proves that the state delta is correct).

How it works:

For every ZK-Rollup block, information required to reconstruct the changes in the state must be submitted as call-data of the Ethereum transaction — otherwise the ZK-Rollup smart contract will refuse to make the state transition. State changes on zkRollups incurs a small gas cost per transaction which grows linearly with the number of transactions. With the Merkle tree data at hand, users who are being censored always have the ability to claim their funds directly from the zkRollup contract on the mainnet. All they need to do is to provide a Merkle proof of ownership on their account. Thus, on-chain data availability serves as a guarantee that nobody (including zkRollup operators) can freeze nor capture users' funds.

Some major pros are:

- Bundle hundreds of transfers off-chain and generate a cryptographic proof (ZK-SNARK). Then post the rollup data and proof to Ethereum blockchain for verification. The computation can be in a parallel computing model which encourages decentralization.
- Unlike Optimistic Rollup, it has no delays when moving funds from layer 2 to layer 1 because a validity proof accepted by the ZK-rollup contract has already verified the funds.
- Since the transaction and processing is handled in the L2 block chain, it can extend the mainchain's scalability, increase processing speed and reduce gas fees more than 100 times.

3. Ethereum is a state machine that moves forward with each new block. At any instance, it provides a complete state of Ethereum consisting of the data related to all accounts and smart contracts running on the EVM. The state of Ethereum modifies whenever a transaction is added to the block by changing the balances of accounts. Based on the massive adoption of Ethereum across the globe, this state has become a bottleneck for validators trying to sync with the network as well as validate transactions. Briefly describe the concept of stateless clients, and how they help resolve this issue? Explain how Zero-Knowledge improves on the concept of stateless client?

The EVM is a big state machine. Its interaction with blocks can be represented as follows, with `stf` being the state transition function:

```
stf(current_state, block) = new_state
```

For Ethereum, running a full node (as opposed to stateless) has become very complex: heavy, a lot of computation, order is important so it's longer to bootstrap a node.

A stateless client solves this issue by changing the parameters needed to validate a new state:

```
stf(current_state_merkle_root, block, witness) = new_state_root
```

Firstly, this allows for actual validation and does not require any form of so-called optimism. Moreover, out of order validation is also made possible and might come in handy for fast bootstrap of nodes in the event of shard reshuffling (Eth2.0 sharding phase). The witness is used as a proof that the updated merkle root is a result of the block's change in state.

Nevertheless, the part of the state that is read and updated needs to be posted as a part of the witness. This means that several merkle branches need to be committed as part of the new block validation. There is a need for even lighter (in terms of bytes size) clients. There comes zero-knowledge proofs.
A stateless node can be correct in its validation with the following parameters:

- the merkle root of the current state,
- the new block
- a zero-knowledge proof that proves the new state root is a result of updating the current state merkle tree with the new block's information.

Zero-knowledge proofs improve on the stateless client concept as they allow for a fully stateless validation process. Without zero-knowledge technology (or polynomial commitments), the state transition function in a stateless client has to incorporate a "semi-stateful" witness (i.e. parts of the state that are read and modified). By providing only current and future merkle root of the state, the new block and a zero-knowledge proof, nodes can now simply verify zero-knowledge proofs and expect that the proof will return true, if and only if the new block is valid.

Q2 Roll the TX up

1. Review the RollupNC source code in the learning resources focusing on the contract and circuit and explain the below functions (Feel free to comment inline)

- UpdateState (Contract)
- Deposit (Contract)
- Withdraw (Contract)
- UpdateStateVerifier (Circuit)

Propose possible changes that can be made to the rollup application to provide better security and functionalities to the users

Comments and explanation for RollupNC.sol: [Link]
Comments and explanation for update_state_verifier.circom: [Link]

2. Clone a copy of the ZKSync source code and run the unit test on it. Submit a screenshot of the test result and justification in the event any of the tests fails. Review the source code of ZKSync and provide detailed explanations for the below functionalities Docs (You can use either pseudocode or comment inline):

Attached log file for the test results: [Link]

```
test result: ok. 4 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

     Running tests/integration.rs (target/debug/deps/integration-8d91782746e83ebf)

running 5 tests
test batch_transfer ... ignored
test comprehensive_test ... ignored
test full_exit_test ... ignored
test nft_test ... ignored
test simple_transfer ... ignored

test result: ok. 0 passed; 0 failed; 5 ignored; 0 measured; 0 filtered out; finished in 0.00s

     Running tests/unit.rs (target/debug/deps/unit-7288a428494053ac)

running 21 tests
test test_tokens_cache ... ok
test utils_with_vectors::test_fee_packing ... ok
test utils_with_vectors::test_formatting ... ok
test utils_with_vectors::test_token_packing ... ok
test wallet_tests::test_wallet_account_id ... ok
test primitives_with_vectors::test_signature ... ok
test wallet_tests::test_wallet_account_info ... ok
test wallet_tests::test_wallet_address ... ok
test wallet_tests::test_wallet_get_balance_committed ... ok
test wallet_tests::test_wallet_ethereum ... ok
test wallet_tests::test_wallet_get_balance_committed_not_existent ... ok
test wallet_tests::test_wallet_get_balance_verified ... ok
test signatures_with_vectors::test_withdraw_nft_signature ... ok
test signatures_with_vectors::test_mint_nft_signature ... ok
test wallet_tests::test_wallet_get_balance_verified_not_existent ... ok
test signatures_with_vectors::test_transfer_signature ... ok
test wallet_tests::test_wallet_refresh_tokens ... ok
test wallet_tests::test_wallet_is_signing_key_set ... ok
test signatures_with_vectors::test_withdraw_signature ... ok
test signatures_with_vectors::test_forced_exit_signature ... ok
test signatures_with_vectors::test_change_pubkey_signature ... ok

test result: ok. 21 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 2.19s
```

```
     Running unittests (target/debug/deps/zksync_api-b7c8dc595541ac3a)

running 39 tests
test api_server::rest::forced_exit_requests::v01::tests::test_disabled_forced_exit_requests ... ignored
test api_server::rest::forced_exit_requests::v01::tests::test_forced_exit_requests_get_fee ... ignored
test api_server::rest::forced_exit_requests::v01::tests::test_forced_exit_requests_submit ... ignored
test api_server::rest::forced_exit_requests::v01::tests::test_forced_exit_requests_wrong_tokens_number ... ignored
test api_server::rest::v02::account::tests::accounts_scope ... ignored
test api_server::rest::v02::block::tests::blocks_scope ... ignored
test api_server::rest::v02::config::tests::config_scope ... ignored
test api_server::rest::v02::fee::tests::fee_scope ... ignored
test api_server::rest::v02::status::tests::status_scope ... ignored
test api_server::rest::v02::token::tests::tokens_scope ... ignored
test api_server::rest::v02::transaction::tests::transactions_scope ... ignored
test api_server::web3::tests::block_number ... ignored
test api_server::web3::tests::create_logs ... ignored
test api_server::web3::tests::erc20_calls ... ignored
test api_server::web3::tests::erc721_calls ... ignored
test api_server::web3::tests::get_balance ... ignored
test api_server::web3::tests::get_block ... ignored
test api_server::web3::tests::get_block_transaction_count ... ignored
test api_server::web3::tests::get_logs ... ignored
test api_server::web3::tests::get_transaction_by_hash ... ignored
test api_server::web3::tests::get_transaction_receipt ... ignored
test api_server::web3::tests::ipfs_cid ... ignored
test api_server::web3::tests::static_methods ... ignored
test api_server::tx_sender::tests::test_scaling_user_fee_by_two ... ok
test api_server::tx_sender::tests::test_scaling_user_fee_by_one_cent ... ok
test api_server::tx_sender::tests::test_scaling_user_fee_by_5_percent ... ok
test api_server::rpc_server::ip_insert_middleware::tests::insert_ip_incorrect_call_test ... ok
test api_server::rpc_server::ip_insert_middleware::tests::insert_ip_test ... ok
test api_server::rpc_server::test::tx_fee_type_serialization ... ok
test api_server::rpc_server::ip_insert_middleware::tests::prevent_user_from_overriding_metadata ... ok
test fee_ticker::ticker_api::coinmarkercap::test::parse_coinmarket_cap_responce ... ok
test fee_ticker::tests::test_zero_price_token_fee ... ok
test fee_ticker::validator::tests::check_tokens ... ok
test eth_checker::tests::actual_data_check ... ok
test fee_ticker::ticker_api::coinmarkercap::test::test_fetch_coinmarketcap_data ... ok
test fee_ticker::ticker_api::coingecko::tests::test_coingecko_api ... ok
test fee_ticker::tests::test_ticker_subsidy ... ok
test eth_checker::tests::test_eip1271 ... ok
test fee_ticker::tests::test_ticker_formula ... ok

test result: ok. 16 passed; 0 failed; 23 ignored; 0 measured; 0 filtered out; finished in 0.16s

     Running unittests (target/debug/deps/dev_liquidity_token_watcher-57bfd61198a430c9)

running 2 tests
test tests::get_volume_for_whitelisted ... ok
test tests::get_volume_for_blacklisted ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

1. How does the core server maintain transactions memory pool and commit new blocks?

Comments and explanation inside the code: [mempool_transactions_queue.rs] and [mod.rs]

2. How does the eth_sender finalizes the blocks by sending corresponding Ethereum transactions to the L1 smart contract?

The eth_Sender module (defined in core/bin/zksync_eth_sender/src/lib.rs) can synchronize the operations occurring in ZKSync with the Ethereum blockchain. This is done by creating transactions from the operations, sending them, and ensuring that every transaction is executed successfully and confirmed.
The essential part of this structure is an event loop (which is supposed to be run in a separate thread), which obtains the operations to commit through the channel and then commits them to Ethereum, ensuring that all the transactions are successfully included in blocks and executed.

A transaction (transfer, deposit) in ZKSync is considered sync to ETH after 4 phases:

- **CommitBlocks:** ZKSync block contains such transactions committed to Ethereum through L1 smart contract. Obviously, Just the state root is committed to the ETH block. Transaction data is passed as calldata and stored in ETH log.
- **CreateProofBlocks:** Prover application generates a ZK-SNARK proof for the transactions of this block
- **PublishProofBlocksOnchain:** ZK-SNARK proof is sent through L1 smart contract to verify the block change.
- **ExecuteBlocks:** Transactions in the block are executed in ZKSync.

eth_sender preserves the order of operations: it guarantees that operations are committed in FIFO order, meaning that until the older operation of a certain type (e.g., commit) will always be committed before the newer one.

eth_sender loads operations from DB (load_new_operations()) and tries to commit operation to ETH (in initialize_operation()), it selects nonce, gas price, sign transaction and watches for its confirmations. If a transaction is not confirmed for a while, it increases the gas price and does the same, but it keeps the list of all sent transaction hashes for one operation, since it can't be sure which one will be committed. Signed transaction is saved to DB before sending it to ETH, this is to make sure that state is always recoverable.

It handles ongoing operation (perform_commitment_step()) by checking its state and doing the following:

- If the transaction is either pending or completed, stops the execution (as there is nothing to do with the operation yet).
- If the transaction is stuck, send a supplement transaction for it.
- If the transaction is failed, handle the failure according to the failure processing policy.

3. ZKSync 2.0 was recently launched to testnet and has introduced ZKPorter. Argue for or against ZKPorter, highlighting the advantages or shortcomings in this new protocol.

(Every user is free to opt into their own security threshold. Any user who wants all data available on-chain can stay completely on the rollup side. But if you are a fee-sensitive user, you can choose to make zkPorter your home. (We suspect that traders and new users will most likely use zkPorter.) *- Matter Labs*

ZKPorter allows for large scale network scalability on an exponential scale, and also allows for composability of applications and protocols.
ZKPorter separates the concerns of state validity and data availability. State validity — that the transition from one state to the next is always valid — is uniformly enforced by means of zero-knowledge proofs, which offer exponential scalability while inheriting the security

guarantees of the underlying L1. In contrast, data availability is delegated to individual shards, which are free to experiment with different solutions.
On the other hand, zk-based scaling cannot directly solve the data availability problem: if the state data becomes unavailable, funds are frozen. ZK-Rollup and Validium handle the data availability problem in different ways.

Q3 Recursive SNARK's

1. Why would someone use recursive SNARK's? What issues does it solve? Are there any security drawbacks?

There are a number of use cases for recursive composition. One of the most well-known at the moment is that of Coda, a fully-succinct blockchain protocol allowing clients to validate the entire chain by checking a small cryptographic proof in the tens of kilobytes. Recursive proofs have also been proposed for other blockchain scaling solutions, such as recursively proving validity of Bitcoin's proof of work consensus algorithm, and at Celo using bounded recursion to efficiently verify aggregated signatures for mobile clients.

2. What is Kimchi and how does it improve PLONK?

In Mina, ZK-Dapps (zero-knowledge smart contracts) can be written in typescript using the snarkjs library, and then compiled down to some intermediary representation with snarky. A Kimchi compiler can then be used to compile the program into the prover and verifier indexes, and both sides can use Kimchi provided functionalities to produce proofs as well as verify them. The verifier index is uploaded on chain, which allows anyone to verify proofs contained in transactions that claim that they executed a zkApp correctly.

Kimchi is a collection of improvements, optimizations, and alterations made on top of PLONK. For example, it overcomes the trusted setup limitation of PLONK by using a bulletproof-style polynomial commitment inside of the protocol. This way, there is no need to trust that the participants of the trusted setup were honest (if they were not, they could break the protocol)

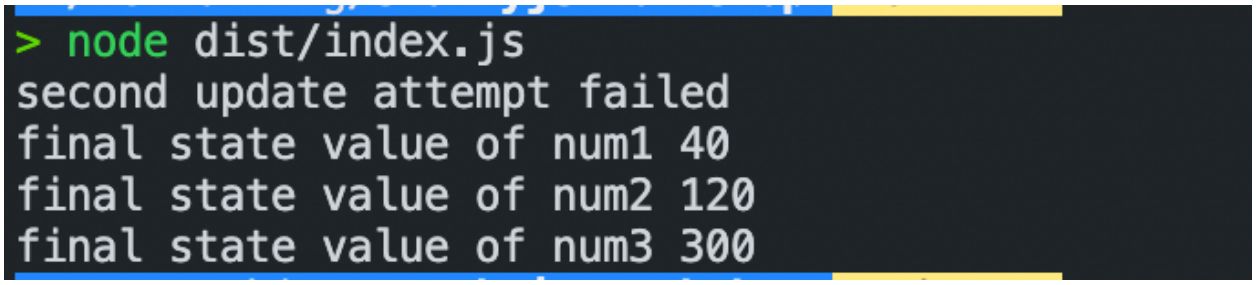
The two last important aspects I understood from Kimchi are:

- That gates can directly write their outputs to the registers of the next gate, this allows for faster iterative computation like that of Poseidon's.
- Lookup tables allow for large logic gates to be expressed as a table, and operations within it to be looked up. This makes operations way faster. The example Mina gives is that a XOR table over 4 bits elements is of size 2^8. Implementing it with generic gates would be tedious.

Pickles now uses an upgraded proof system: Kimchi. Kimchi brings several optimizations and quality-of-life improvements to circuit builders. This should allow for faster provers, larger circuits, and potentially shorter proof sizes!

3. Clone [github repo](). Go through the snapps from the src folder and try to understand the code. Create a new snapp that will have 3 fields. Write an update function which will update all 3 fields. Write a unit test for your snapp.

Link to code: [link]



Question 5: Thinking in ZK

1. If you have a chance to meet with the people who built ZKSync and Mina, what questions would you ask them about their protocols?

ZK-Sync:

- What's the worst-case scenario for zk-sync? Will the pros outweigh the cons for the same?
- What do you think about upcoming solutions from polygon in the same area?

Mina:

How did you come up with an idea like this?

Both:

What can be done to bridge the gap between understanding of workings and tx explorers from an end users perspective.