Assignment 4

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

[ANSWER] A

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 few 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?*

   [ANSWER]

- **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.
- **SIDECHAINS**: A sidechain is a separate blockchain which runs in parallel to Ethereum 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:

        - Established technology.
        - 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).
        - A quorum of sidechain validators can commit fraud.
- **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.
    - Pros:
        - High throughput, low cost per transaction.
        - Good for transactions between arbitrary users (no overhead per user pair if both are established on the plasma chain).
    - Cons:
        - Does not support general computation. Only basic token transfers, swaps, and a few other transaction types are supported via predicate logic.
        - Need to periodically watch the network (liveness requirement) or delegate this responsibility to someone else to ensure the security of your funds.
        - Relies on one or more operators to store data and serve it upon request.
        - 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.
- **VALIDIUM:** Uses validity proofs like ZK-rollups but data is not stored on the main layer 1 Ethereum chain. This can lead to 10k transactions per second per validium chain and multiple chains can be run in parallel.
        - Pros:
            - No withdrawal delay (no latency to on-chain/cross-chain tx); consequent greater capital efficiency.
            - Not vulnerable to certain economic attacks faced by fraud-proof based systems in high-value applications.
        - Cons:
            - Limited support for general computation/smart contracts; specialized languages required.
            - High computational power required to generate ZK proofs; not cost effective for low throughput applications.
            - Slower subjective finality time (10-30 min to generate a ZK proof) (but faster to full finality because there is no dispute time delay).

        - Generating a proof requires off-chain data to be available at all times.
- **OPTIMISTIC ROLLUPS:** Optimistic rollups sit 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 transaction due to potential fraud challenges.
        - An operator can influence transaction ordering.
- **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.

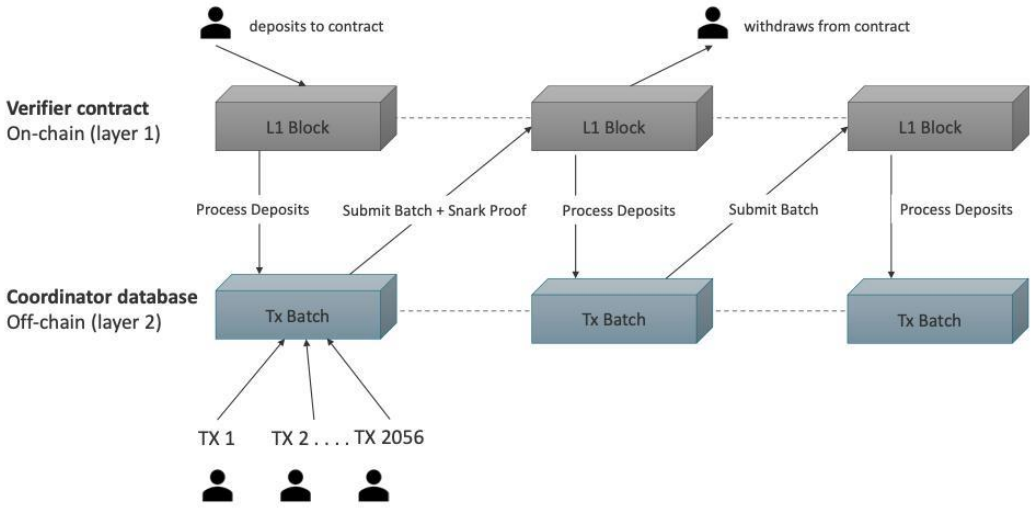
    - Pros:
        - Faster finality time since the state is instantly verified once the proofs are sent to the main chain.
        - Not vulnerable to the economic attacks that Optimistic rollups can be vulnerable to.
        - Secure and decentralized, since the data that is needed to recover the state 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

The biggest problem with the Plasma approach as described above and summarize as following:

- When users want to withdraw their assets from Plasma to the Ethereum, they need to wait seven days to settle the transaction.
- 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.

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.*

[ANSWER]



Key features:

- 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 L2 block chain, it can extend mainchain's scalability, increase processing speed and reduce gas fees more than 100 times.

Arguments:

- ZK-rollup is related new technology with less proven use cases compared with other L2 solutions (Optimistic rollup, Sidechains, Plasma).
- ZK-rollup relies on significant hash power to compute. Therefore, programs that have limited on-chain activity may find it more beneficial to use optimistic rollup solutions.
- ZK-compatible EVM is more difficult to build compared with other L2 solutions (Optimistic roll-up).
- Security concerns:
  - The initial setup of ZK-Rollups is assumed to be a trusted state when this trust cannot be proven. A small group of developers will be subject matter experts on the initial trusted state. This undermines decentralization and opens the risk of attacks by dishonest developer.
  - Crypto used in ZK-Snark is not quantum resistant.

    However, both problems can be mitigated by ZK-STARK based ZK-Rollup, like StarkNet.

- An operator can influence transaction ordering which could cause concerns in decentralized environment.

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 client, and how they help resolve this issue? Explain how Zero-Knowledge improves on the concept of stateless client?*

   [ANSWER] Today, to validate a block, a Ethereum validator can be considered to validate a state transition function as: STF(current_state, new_block). Where current_state is all block data and new_block is the block to be verified. This means the validator need to load all data in local disk and replay each transaction in the new_block. It requires lot of io and computation resources.

   Stateless client is designed to make a new state transition function as: STF_New(state_root, new_block, witness). Where state_root is merkle root of the current_state (as described above), witness is the zk-proof that prove the correct execution of transactions in new_block. By taking this approach,

   - Validator now long need to download all block data and only need to retrieve state_root from block chain.
   - The validation can be done completely in memory.

This means less disk space and io usage, and faster validation.

Zero-knowledge is key technology to enable stateless client. Specifically, ZKP can be used to:

1. Use VRF (Verifiable Random Function) and VDF (Verifiable Delay Function) to randomly select a lead validator.
2. The lead validator uses ZK-SNARK or ZK-STARK techniques to generate proof for the correct execution of new transactions. The proof then submitted on chain.
3. Other (stateless client) validators verify the block by just checking the proof (instead of recomputing all the transactions).

2. Roll the TX up

1. *[Infrastructure Track only] 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)*
    1. *UpdateState (Contract)*
    2. *Deposit (Contract)*
    3. *Withdraw (Contract)*
    4. *UpdateStateVerifier (Circuit)*

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

[ANSWER] Comment inline,
https://github.com/geesimon/zku/blob/main/week4/RollupNC.sol
https://github.com/geesimon/zku/blob/main/week4/update_state_verifier.circom

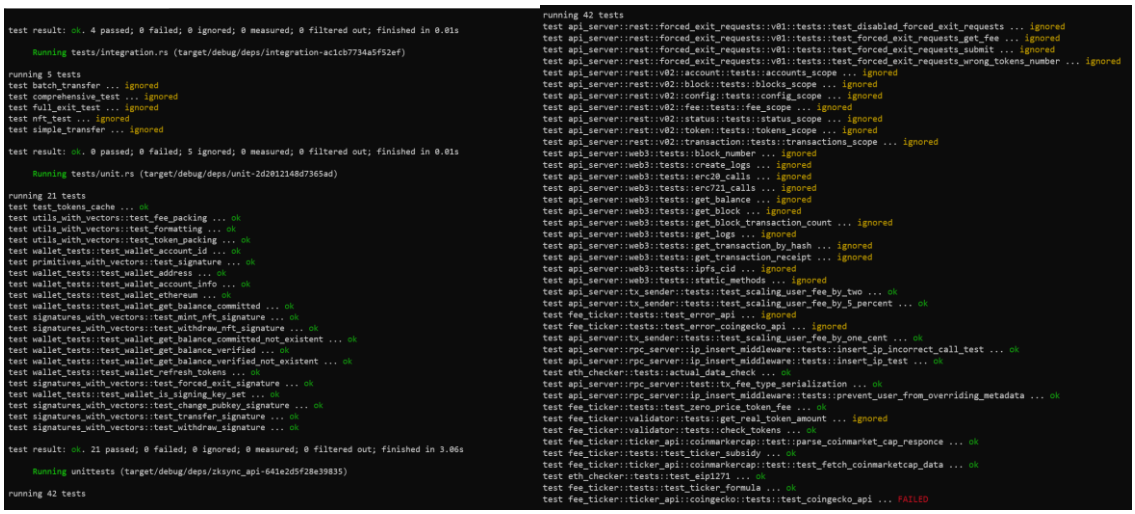
For improvement,

- This implementation assume data availability guarantees from coordinator which causes single point of failure. In case coordinator is not available or censor the transaction, user won't be able to deposit and withdraw token timely. A decentralized coordinator mechanism could help resolve this problem. Even better, we could add a `Full Exit` mode like the one implemented in zkSync to allow user to withdraw fund without interaction with coordinator.
- The accounts and transactions tree are maintained in a node.js process which is not secure, a better solution is to make L2 chain to store such info.

2. *[Infrastructure Track only] 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*

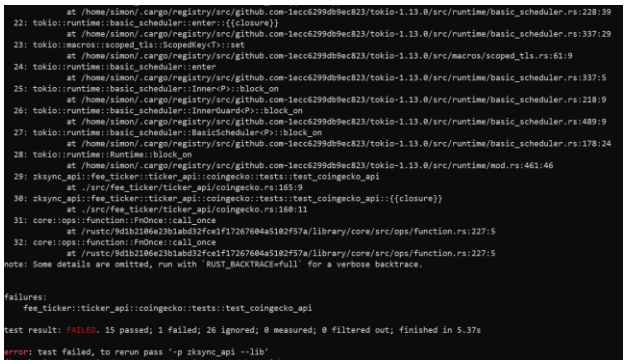
for the below functionalities Docs (You can use either pseudocode or comment inline):

[ANSWER]



```
test result: ok. 4 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.01s

     Running tests/integration.rs (target/debug/deps/integration-ac1cb7734a5f52ef)

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 0.01s

     Running tests/unit.rs (target/debug/deps/unit-2d2012148d7365ad)

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_address ... ok
test wallet_tests::test_wallet_account_info ... ok
test wallet_tests::test_wallet_ethereum ... ok
test wallet_tests::test_wallet_get_balance_committed ... ok
test signatures_with_vectors::test_mint_nft_signature ... ok
test signatures_with_vectors::test_withdraw_nft_signature ... ok
test wallet_tests::test_wallet_get_balance_committed_not_existent ... ok
test wallet_tests::test_wallet_get_balance_verified ... ok
test wallet_tests::test_wallet_get_balance_verified_not_existent ... ok
test wallet_tests::test_wallet_refresh_tokens ... ok
test signatures_with_vectors::test_forced_exit_signature ... ok
test wallet_tests::test_wallet_is_signing_key_set ... ok
test signatures_with_vectors::test_change_pubkey_signature ... ok
test signatures_with_vectors::test_transfer_signature ... ok
test signatures_with_vectors::test_withdraw_signature ... ok

test result: ok. 21 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 3.06s

     Running unittests (target/debug/deps/zksync_api-641e2d5f28e39835)

running 42 tests
```

```
running 42 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_5_percent ... ok
test fee_ticker::tests::test_error_api ... ignored
test fee_ticker::tests::test_error_coingecko_api ... ignored
test api_server::tx_sender::tests::test_scaling_user_fee_by_one_cent ... 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 eth_checker::tests::actual_data_check ... 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::tests::test_zero_price_token_fee ... ok
test fee_ticker::validator::tests::get_real_token_amount ... ignored
test fee_ticker::validator::tests::check_tokens ... ok
test fee_ticker::ticker_api::coinmarkercap::test::parse_coinmarket_cap_responce ... ok
test fee_ticker::tests::test_ticker_subsidy ... ok
test fee_ticker::ticker_api::coinmarkercap::test::test_fetch_coinmarketcap_data ... ok
test eth_checker::tests::test_eip1271 ... ok
test fee_ticker::tests::test_ticker_formula ... ok
test fee_ticker::ticker_api::coingecko::tests::test_coingecko_api ... FAILED
```



```
          at /home/simon/.cargo/registry/src/github.com-1ecc6299db9ec823/tokio-1.13.0/src/runtime/basic_scheduler.rs:228:39
  22: tokio::runtime::basic_scheduler::enter::{{closure}}
          at /home/simon/.cargo/registry/src/github.com-1ecc6299db9ec823/tokio-1.13.0/src/runtime/basic_scheduler.rs:337:29
  23: tokio::macros::scoped_tls::ScopedKey<T>::set
          at /home/simon/.cargo/registry/src/github.com-1ecc6299db9ec823/tokio-1.13.0/src/macros/scoped_tls.rs:61:9
  24: tokio::runtime::basic_scheduler::enter
          at /home/simon/.cargo/registry/src/github.com-1ecc6299db9ec823/tokio-1.13.0/src/runtime/basic_scheduler.rs:337:5
  25: tokio::runtime::basic_scheduler::Inner<P>::block_on
          at /home/simon/.cargo/registry/src/github.com-1ecc6299db9ec823/tokio-1.13.0/src/runtime/basic_scheduler.rs:218:9
  26: tokio::runtime::basic_scheduler::InnerGuard<P>::block_on
          at /home/simon/.cargo/registry/src/github.com-1ecc6299db9ec823/tokio-1.13.0/src/runtime/basic_scheduler.rs:489:9
  27: tokio::runtime::basic_scheduler::BasicScheduler<P>::block_on
          at /home/simon/.cargo/registry/src/github.com-1ecc6299db9ec823/tokio-1.13.0/src/runtime/basic_scheduler.rs:178:24
  28: tokio::runtime::Runtime::block_on
          at /home/simon/.cargo/registry/src/github.com-1ecc6299db9ec823/tokio-1.13.0/src/runtime/mod.rs:461:46
  29: zksync_api::fee_ticker::ticker_api::coingecko::tests::test_coingecko_api
          at ./src/fee_ticker/ticker_api/coingecko.rs:165:9
  30: zksync_api::fee_ticker::ticker_api::coingecko::tests::test_coingecko_api::{{closure}}
          at ./src/fee_ticker/ticker_api/coingecko.rs:160:11
  31: core::ops::function::FnOnce::call_once
          at /rustc/9d1b2106e23b1abd32fce1f17267604a5102f57a/library/core/src/ops/function.rs:227:5
  32: core::ops::function::FnOnce::call_once
          at /rustc/9d1b2106e23b1abd32fce1f17267604a5102f57a/library/core/src/ops/function.rs:227:5
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose backtrace.


failures:
    fee_ticker::ticker_api::coingecko::tests::test_coingecko_api

test result: FAILED. 15 passed; 1 failed; 26 ignored; 0 measured; 0 filtered out; finished in 5.37s

error: test failed, to rerun pass '-p zksync_api --lib'
```

One test failure:

1. infee_ticker::ticker_api::coingecko::tests::test_coingecko_api: caused by reqwest::Url is trying to create a tokio runtime in another runtime (created by tokio::test).

1. *How the core server maintains transactions memory pool and commits new blocks?*

   [ANSWER]

   - MempoolTransactionsQueue (defined in zksync/core/bin/zksync_core/src/mempool/mempool_transactions_queue.rs):

     Memory pool is simple in memory buffer to store zkSync transactions and runs inside server application. It accepts transactions from api, check signatures and basic nonce correctness (nonce not too small).

Transactions are stored in MempoolTransactionsQueue. It manages 4 in memory transaction queues:

- reverted_txs: reverted transactions queue that must be used before processing any other transactions. Most of the time it will be empty unless the server is restarted after reverting blocks. The queue is only accessible for popping elements.
- priority_ops: queue to store priority transactions, like deposit, withdraw.
- ready_txs: transactions ready for execution.
- pending_txs: transactions that are not ready yet because of the `valid_from` field.

Data in memory pool will be lost on node shutdown. MempoolTransactionsQueue is initialized from DB by function MempoolState::restore_from_db().

- MempoolTransactionsHandler (defined in zksync/core/bin/zksync_core/src/mempool/mod.rs)

  ZKSync transaction request handler runs in its own thread. Specifically, it handles and stores transaction request as:

  - MempoolTransactionRequest::NewTx: add new transitions to MempoolTransactionsQueue pending queue (pending_txs). The transaction will move to ready queue (ready_txs) once valid_from is less than block timestamp.
  - MempoolTransactionRequest::NewTxsBatch: add batch of transaction to MempoolTransactionsQueue pending queue.
  - MempoolTransactionRequest::NewPriorityOps: add priority transaction to MempoolTransactionsQueue priority transactions.

- MempoolBlocksHandler (defined in zksync/core/bin/zksync_core/src/mempool/mod.rs)

  ZKSync block request handler runs in its own thread. Specifically, it handles requests as:

  - MempoolBlocksRequest::GetBlock: request propose a new block. The block is prepared from the MempoolTransactionsQueue following this order:
    - reverted_txs
    - priority_ops
    - ready_txs
  - MempoolBlocksRequest::UpdateNonces(AccountUpdates): handle request to update account tree (MempoolState). Specifically,

- Create account
- Delete account
- Update nonce
- Change account public key hash

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

   [ANSWER]

   ETHSender module (defined in core/bin/zksync_eth_sender/src/lib.rs) can synchronize the operations occurring in ZKSync with the Ethereum blockchain 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 the 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:

   1. CommitBlocks: ZKSync block contains such transaction is committed to Ethereum through L1 smart contract. To clarify, just state root is committed to ETH block. Transaction data is passed as calldata and stored in ETH log.
   2. CreateProofBlocks: prover application generates a ZK-SNARK proof for the transactions of this block
   3. PublishProofBlocksOnchain: ZK-SNARK proof is sent to through L1 smart contract to verify the block change.
   4. ExecuteBlocks: transactions in the block are executed in ZKSync.

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

   ETHSender loads operations from DB (load_new_operations()) and try to commit operation to ETH (in initialize_operation()), it selects nonce, gas price, sign transaction and watch for its confirmations. If transaction is not confirmed for a while, it increases the gas price and do 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, sends a supplement transaction for it.
- If the transaction is failed, handles the failure according to the failure processing policy.

3. *[Bonus] How the witness_generator creates input data required for provers to prove blocks?*

3. ***[All Tracks]** 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.*

[ANSWER]

Advantage

- **Scalability**: ZKPorter can scale from 3k TPS (zkRollup shard) to 20k TPS (Guardian shard) or even more (Protocol X shard) while the state validity is enforced by means of zero-knowledge proof.
- **Cost**: reduce the cost by 100x to1000x compared to main net.
- **Privacy**: unlike ZKSync that needs to post all transactions through calldata. ZKPorter can offer better privacy by leveraging ZKP and only send proof to main net.
- **Flexibility**: flexible data availability is a core design goal for zkPorter. By allowing protocols to design their own policies, zkPorter enables a broad range of possible solutions.

Shortcoming

- Security: ZKPorter is more like a sidechain solution and don't totally rely on the security of ETH. It is considered less secure compared with ZKSync and has data availability issue (like other side chain solution).
- ETH Interoperability: although shard in ZKPorter can interop. The integration between ZKPorter and ETH is weaker compared with ZKSync/ETH. Given the business incentive to have its own chain and token, my worry is ZKPorter could become more and more isolated from ETH.

3. Recursive SNARK's

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

[ANSWER] Verification of one SNARK proof is cheap. However, in block chain scenario, it is still too expensive if we need to verify every block (when a node joins the network). With recursive SNARK, we can make use of one proof to verify the whole block chains. Consider it as proof for one execution and correctness of prior proofs.

In addition, recursive SNARK can circumvent the max constraint limitation (for example: circom can only support 1M constraints). Instead of building a proof for a huge circuit, we can split the entire circuit to smaller piece of circuits and use recursive proof to accumulatively verify every small circuit. Once done, generate a proof for the entire circuit.

In practice, the security drawback is recursive SNARK needs a new protocol to make sure the proof chain are generated correctly.

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

[ANSWER]

Setup

Kimchi 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).

Circuit

Kimchi adds 12 registers to the 3 registers PLONK already had. These registers are split into two types of registers: the IO registers, which can be wired to one another, and temporary registers (sometimes called advice wires) that can be used only by the associated gate. More registers means that we now can have gates that take multiple inputs instead of just two.

As some operations happen more often than others, they can be rewritten more efficiently as new gates. Kimchi offers 9 new gates.

Kimchi gate can directly write its output on the registers used by the next gate. This is useful in gates like “poseidon”, which need to be used several times in a row (11 times, specifically) to represent the poseidon hash function.

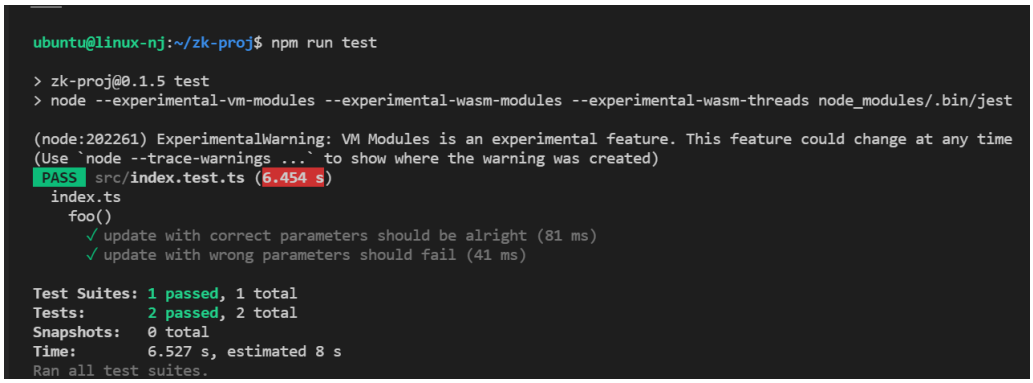
Kimchi support lookups. For example, an XOR table: An XOR table for values of 4 bits is of size 28. Implementing this with generic gates would be hard and lengthy, so instead Kimchi builds the table and allows gates (so far only Chacha uses it) to simply perform a lookup into the table to fetch for the result of the operation.

3. ***[Infrastructure Track only]*** *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.*

   [ANSWER] Code:

   snapp: https://github.com/geesimon/zku/blob/main/week4/index.ts

   Jest: https://github.com/geesimon/zku/blob/main/week4/index.test.ts



4. *[Bonus] In bonus.ts we can see the implementation of zkRollup with the use of recursion. Explain 87-148 lines of code (comment the code inline).*

   [ANSWER] Comment inline the Code:
   https://github.com/geesimon/zku/blob/main/week4/bonus.ts

5. Thinking in ZK

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

   [ANSWER]
   ZKSync:

   - Polygon just released Miden which is a STARK-based, EVM-compatible scaling solution. What advantage/disadvantage ZKSync has over this solution?
   - Is it possible to replace ETH's current EVM by zkEVM? What could be the potential impact and difficulties?

   Mina

   - Currently, snarkyjs's grammar seems quite limited compared with Solidity and may not be tuning-complete. What is the technical difficulty to make a JVM-like VM that can run very complicated and provable application?