ZKU Assignment 4

- I would choose `Stream A`, which is the infrastructure track.

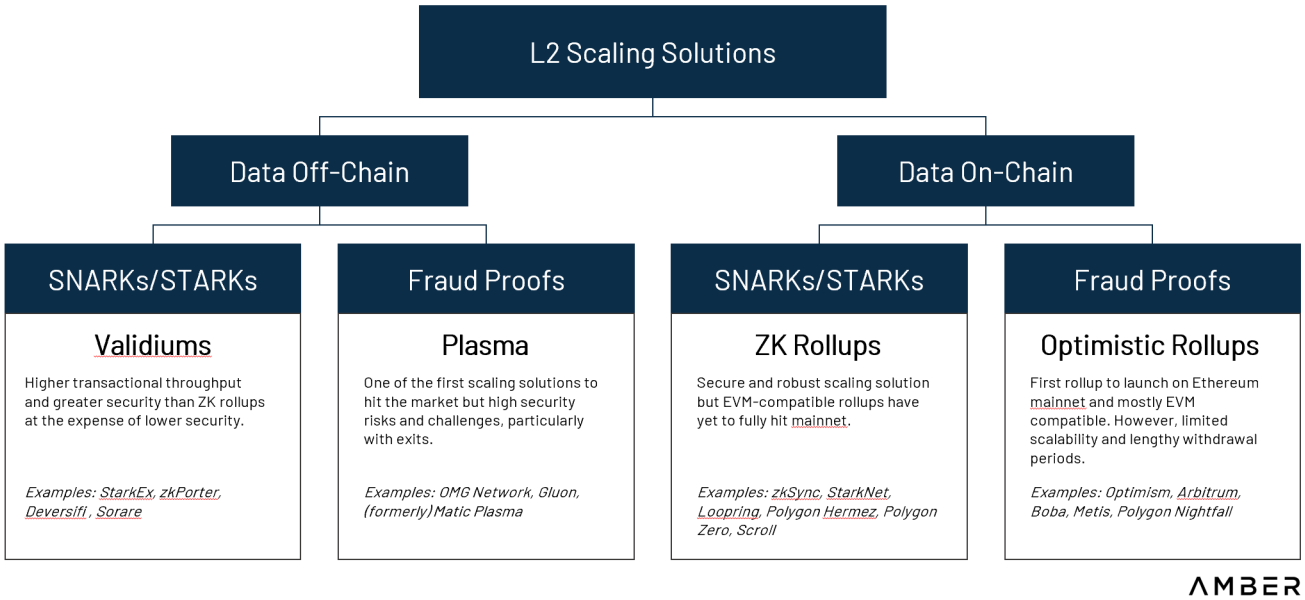
Question 1.

Question 1.1

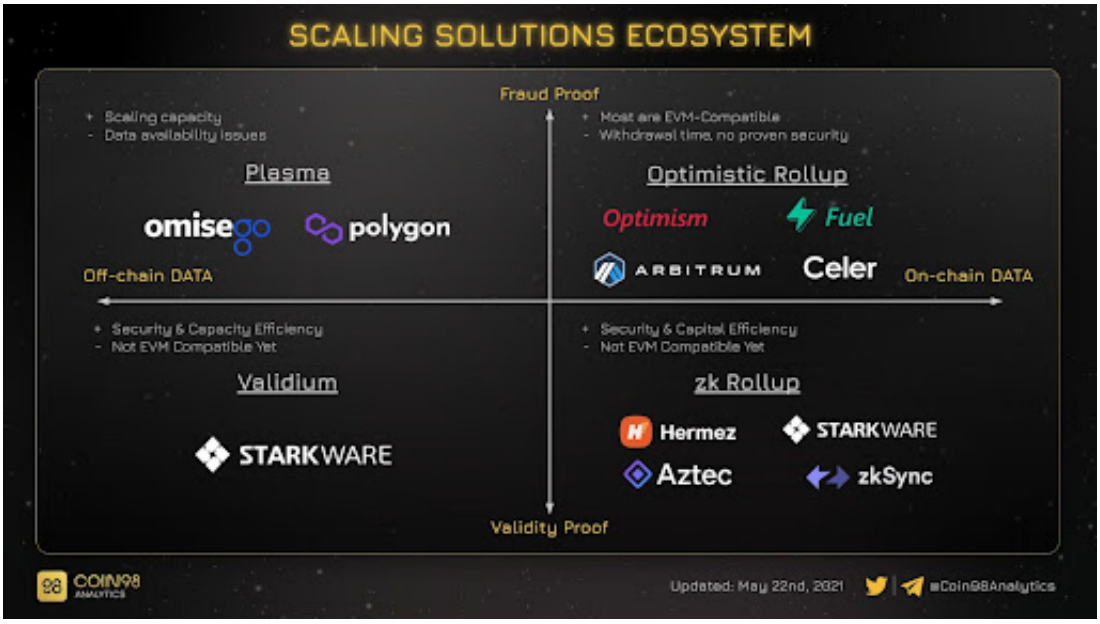
> Based on the above, a number of solutions have been proposed to solve this trilemma. Briefly describe the different scalability solutions and write pros and cons of each approach. What was the biggest problem with the Plasma approach?

	Validity proofs	Fraud proofs
Data on-chain	ZK-Rollup	ORU
Data off-chain	StarkEx	Plasma

Twitter



Twitter



Twitter

State Channels

- The layer 1 blockchain serves as a settlement layer, while the most of the transactions happens in off-chain. By doing that, we could handle the transactions while keeping its costs low with much faster speed. However, each channels cannot communicate with each other. The channel itself fails to transact with the transactions in other channels. In addition, when opening up the channels, the users are required to lock up their liquidity within the channels. This could raise the problem of capital inefficiency when trying to scale the channels. Lastly, the participants are required to actively monitor the network to ensure whether their fund is secure or not.

Sidechains

- It is an basically auxiliary chain that is pegged to Layer 1 blockchain, while utilizing various approaches and consensus algorithms to scale its performances. The problem of sidechain is to give up their decentralization at the expense of speed. The sidechains have their own fatigue to improve the security as much as the one from their own Layer 1 blockchain like Ethereum Mainnet.

Plasma

- Plasma is akin to a sidechain in that it is a set of smart contracts rather than a distinct blockchain from L1. Merkle Tree will be built from transactions, with the root committed to the L1 blockchain for security. However, because an operator is required to publish the merkle root to the L1 blockchain, they will have the potential to use a data availability attack to prevent transaction data from being sent. (i.e., an operator withholding the merkle root to prevent individuals from moving their funds) Because there will be no means to confirm the authenticity of invalid blocks due to a lack of data, invalid blocks may be accepted. Additionally, it does not support general computation other than basic calculations like transfer and swap. The withdrawals are required by some days to get allowed for challenges, which is the same logic as

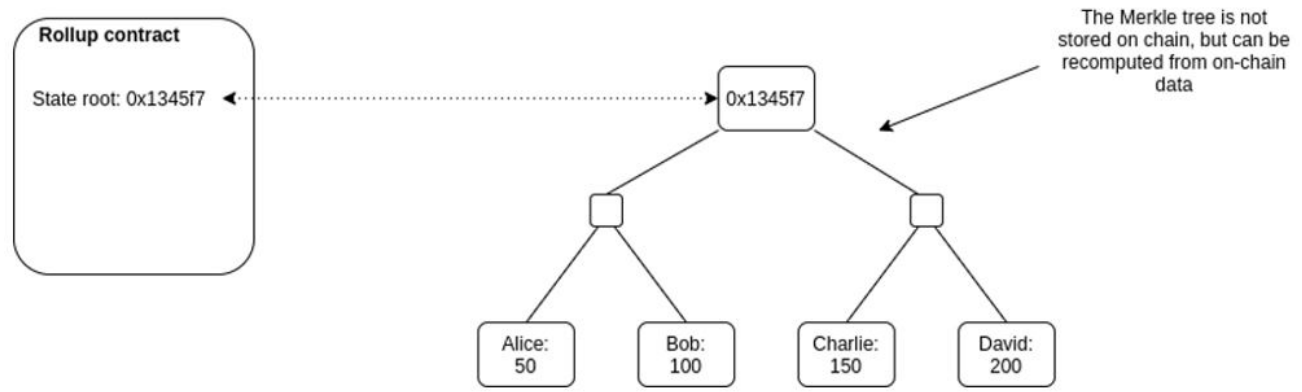
Optimitic Rollups.

Validium

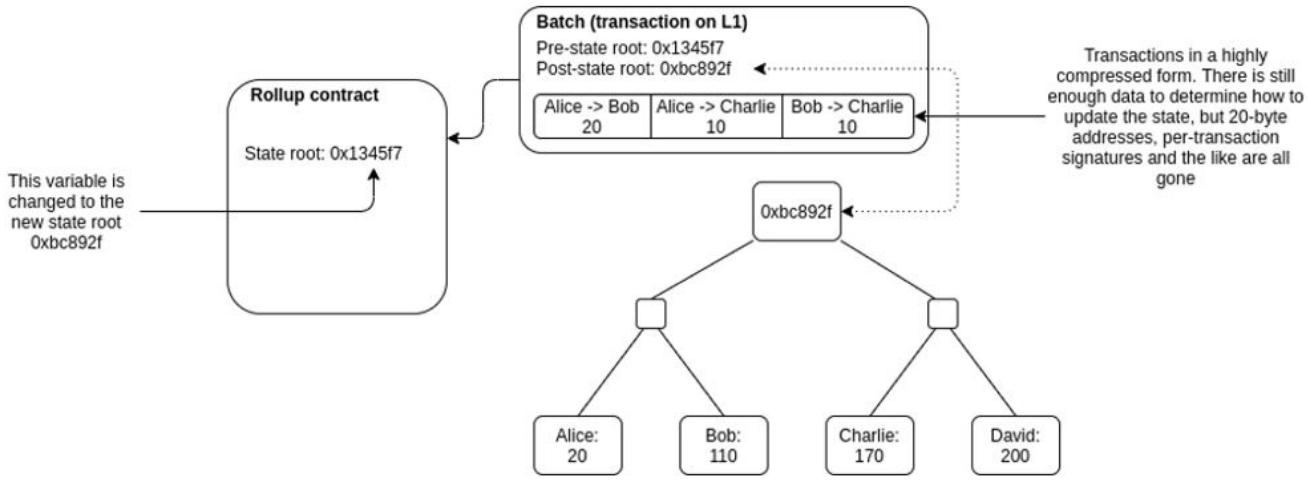
- The protocol employs validity proofs such as ZK-rollups, however data is not kept on the Ethereum main chain's layer 1 but on off-chain. This can result in tens of thousands of transactions per second per validium chain, with numerous chains running in simultaneously. It requires no withdrawal delay or latency to on-chain or cross-chain, this means that the layer could achieve much higher capital efficiency. Since it does use zkp validity proof, no economic vulnerabilities to be faced with when utilizing fraud-proof based systems.

Optimistic Rollups

`Before Updating Rollups, State Root`



`After Updating Rollups, State Root`



- On layer 2, optimistic rollups run parallel to the main Ethereum chain. Because they don't execute any calculation by default, they can increase scalability. Instead, they propose the new state to mainnet following a transaction. Transactions are written to the main Ethereum chain as calldata with optimistic rollups, further optimizing them by lowering the gas cost. Layer 2 Rollup can perform any transactions that is able to be done with Layer 1. Since the data itself is saved at Layer 1 also, it is secure and decentralized. However, it is to note that an operator can influence the ordering of transactions/MEV frauding. It also points that much longer waiting times for on-chain transactions is required to prevent potential fraud challenges.

Zero-knowledge based Rollups

- Hundreds of off-chain transfers are bundled into zero-knowledge rollups, which create cryptographic proofs. These proofs are uploaded to layer 1 and can be in the form of SNARKs or STARKs. The status of all transfers on layer 2 is maintained by the ZK-rollup smart contract, and this state can only be altered with a validity proof.
- This implies that instead of complete transaction data, ZK-rollups simply require the validity evidence. Validating a block using a ZK-rollup is faster and less expensive since less data is provided. The rollup's state is quickly validated after the proofs are submitted to the main chain, resulting in a faster finality time. In addition, the rollup itself is decentralized and secure since the data is stored on layer 1 chain.
- ZK rollup does not have any vulnerabilites to the economic attacks that Optimistic Rollup has. However, the rollup has some computational intensity to compute. It means that the efficiency for the chain which has a relatively lower on-chain activity is low. ZK rollup still has the problem of operator fraud. They can reorder transactions and manipulate them to earn MEV incentives.

The problem of Plasma

- To begin with, the users who wish to withdraw their assets from Plasma to its Layer 1 blockchain must wait at least 7 days for the transactions to be settled. It is required to accept any possible challenges that catches fraud.
- Each plasma chain necessitates the publishing of Merkle root commitments to the mainchain by an operator, which requires putting the faith in a third party to correctly publish the Merkle root commitments to the chain.
- Lastly but not least, Plasma needs the presence of the owners of the assets being transacted (basic transfers).

Question 1.2

> 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.

- For the solution: To begin with, let me explain the key features of zkRollups. It could handle the hundreds of transfers with bundling them in off-chain and produce a cryptographic proof (zk-SNARKs). To verify, it could upload the rollup data and provide evidence to the Layer 1 blcockahin. By doing the calculation in a parallel computing approach, it could achieve decentralization with much lower gas fees but with much faster processing speeds. It also does not have any delayed funding withdrawal periods like 7 days of Optimistic Rollups.
- Against the solution:
    - zk-Rollup is difficult to achieve EVM compatibility when compared to Optimistic rollup.
    - zk-Rollup relies on relatively heavy computation resources and significant hash powers. The programs have relatively limited on-chain activity due to the aformentioned issues. It would be much beneficial to use Optimistic Rollup according to the context.

- The initial setup for zk-Rollup assumes the involvement of a trusted state. The trusted setup cannot be proven in any decentralized ways, since only small bunches of people are invited to initiate the setup phase. Some pundits claim that this nature strongly undermines the decentralization and opens the possibility for dishonest attacks.
- Finally, the operator is not centralized and check-and-balanced. It means that they could initiate MEV-based attacks or transaction reordering easily to earn economic incentives.

Question 1.3

> 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?

- Any Ethereum validators may be used to validate a state transition function such as `STF(current state, new block)` to validate a block. THe current state refers to the block's entire contents, and new block refers to the block that has to be validated.
- This necessitates the validator loading all data onto local disk and replaying each transaction in new block. It also requires a large amount of I/O and processing resources.
- The new state function as `STF_NEW(state_root, new_block, witness)` is required to design a stateless client. `witness` is the zk-based proof that validates the correct execution of transactions in a new block. `state_root` is merkle root of the `current_state` that refers to the entire block contents. Validators can entertain the benefits of new state functions by the following. 1) reduced download times when retrieving `state_root` from the mainnet. 2) The validation logic could be achieved completely through in-memory with lower disk usages and faster performances.
- Thus, we can acknowledge that zero-knowledge technology is crucial to the designing the concept of stateless client. Specifically, the zero-knowledge technolgy could be utilized to implement VRF (Verifiable Random Function) and VDF (Verifiable Delay Function) to randomly choose a lead validator, which utilizes zk-STARKs/SNARKs technolgies to generate proof for the correct execution of new transactions. When verifying those transactions, the stateless clients only require to check the proof, instead of replaying all of the transactions again with heavy computational resources involved.

Question 2.

Question 2.1

> [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) 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.

- To begin with, the RollupNC scales in higher level on Ethereum mainnet by coordinators to periodically batch transactions and publish their merkle tree (storing transactions and balances) proof along with the merkle proof to the Layer 1 chain. The merkle proofs are stored on the L1 chain, the L2 needs to ensure its data availability problem. The setting up coordinator in a decentralized fashion is important, however, the considerable amount of attempt is required to achieve decentralization, by designing security methods and staker incentives.

1. UpdateState (Contract)

- `UpdateState` accepts the SNARK prover inputs that is generated by using the commands like `snarkjs generatecall`. The function is executed by the coordinator attempting to verify the given SNARK proof to appreciate whether the account merkle root is updated correctly by transactions. By checking the output of the circuit matches the current root of the balance merkle tree or not, the smart contract can ensure the fact that the circuit was called with the right initial merkle root.
- To explain the parameter, the `input[0]` is the new merkle tree root after applying the transactions. `input[1]` is the current transaction merkle tree root. Finally, `input[2]` is the old merkle tree root before applying the transactions.
- The circuit updates the state and returns the new balance merkle tree root, and smart contract stores the newly updated merkle tree root after validating the output and verifying the proof.

```
// 1. validate the proof
require(update_verifyProof(a,b,c,input), "SNARK proof is invalid");

// 2. update the root
currentRoot = input[0];

// 3. update a count of numbers updated.
// uint256 public updateNumber;
updateNumber++;

// 4. setting the transaction merkle root to the updates map for checking whether
the txRoot passed is valid when withdrawaling the funds.

// transaction root to updated indexes
// mapping(uint256 => uint256) public updates;
updates[input[1]] = updateNumber;

// 5. emit UpdatedState / input[0] == newRoot, input[1] == txRoot, input[2] ==
oldRoot
```

```
emit UpdatedState(input[0], input[1], input[2]);
```

- Full source code of `updateState` function is the below.

```
function updateState(
            uint[2] memory a,
            uint[2][2] memory b,
            uint[2] memory c,
            uint[3] memory input
        ) public onlyCoordinator {
        require(currentRoot == input[2], "input does not match current root");
        //validate proof
        require(update_verifyProof(a,b,c,input),
        "SNARK proof is invalid");
        // update merkle root
        currentRoot = input[0];
        // save transaction root to onchain map `updates`
        updateNumber++;
        updates[input[1]] = updateNumber;
        emit UpdatedState(input[0], input[1], input[2]); //newRoot, txRoot,
oldRoot
    }
```

2. Deposit (Contract)

- User sends the following to the `payable` smart contract, which is a `payable` smart contract, implying the user must pay an amount that is at least equal to the deposit amount: 1) their public key 2) deposit amount 3) tokenType.
- The function is called when the user tries to deposit ERC20 tokens. When it comes to parameter, `pubkey` is user's public key. `amount` is a deposit amount. Lastly, `tokenType` is 0 (coodinator only), 1 (ETH) and >1 (ERC20 token).
- It verifies the token types and transfer token to the fund pool which is managed by the contract address. The deposit happens in two steps. To begin with, the user calls the smart contracts that deposit the expected tokenType and sends their funds. Next, the coordinator periodically processes the deposits and updates the newly created deposits with updating the merkle tree roots.
- To create a full-scaled merkle tree, the tokenType 0 is used to fill the empty merkle leaves, which means that only the oocrdinator can transact this methods to use the token. The transaction amounts is expected to be initialized to 0 since the token has no value. Meanwhile, if tokenType is 1, it is Ethereum token. If tokenType is large than 1, it is other ERC20 tokens that utilizes the smart contracts of token to deposit the funds.

```
public payable {
        if (tokenType == 0) {
```

```
            require(
                msg.sender == coordinator,
                "tokenType 0 is reserved for coordinator"
            );
        } else if (tokenType == 1) {
            require(
                msg.value > 0 && msg.value >= amount,
                "msg.value must at least equal stated amount in wei"
            );
        } else if (tokenType > 1) {
            require(amount > 0, "token deposit must be greater than 0");
            address tokenContractAddress = tokenRegistry.registeredTokens(
                tokenType
            );
            tokenContract = IERC20(tokenContractAddress);
            require(
                tokenContract.transferFrom(msg.sender, address(this), amount),
                "token transfer not approved"
            );
```

- It hashes `[eddsa_pubkey, amount, nonce = 0, tokenType]` to get the `deposit_leaf`, which is an `account_leaf`.

```
uint[] memory depositArray = new uint[](5);
        depositArray[0] = pubkey[0];
        depositArray[1] = pubkey[1];
        depositArray[2] = amount;
        depositArray[3] = 0;
        depositArray[4] = tokenType;
```

- The function creates a `deposit MiMC hash` using the elements of the given deposit array. which is relatively expensive on-chain activity but optimized for zk-SNARKs execution. As stated on last assignment, the hash function could be replaced for SHA256 which is optimized for EVM gas costs, but it would require more computation resources to verify the zk-SNARKs proof.
- After rendering the hash, it pushes the `deposit_leaf` which is a deposit hash to the pending deposits queue.
- It increments the queueNumber by 1.

```
uint256 depositHash = mimcMerkle.hashMiMC(depositArray);
pendingDeposits.push(depositHash);
emit RequestDeposit(pubkey, amount, tokenType);
queueNumber++;
```

- The deposit array is hashed into on-chain merkle root and stores in `pendingDeposits[0]` and

compute treeHeight. The number of times a user has to hash is equal to the number of times that the `queueNumber` of the deposits, which can be divided by 2. The deposits are hashed together, are deleted, replaced with their new hash.

- The coordinator should put their own efforts to amortize the cost of each transactions and charge a premium cost to depositors to cover the gas fees to evenly distribute among the users. It is to note that users depositing their funds later in the queue is considerably higher than those who deposits much earlier, as they (later depositors) have to remit the expensive gas fees for execute the MiMC hash function more number of times to get a root hash of the merkle tree.
- If the number of transactions exceed the power of 2, it updates the height of merkle tree root.

```
uint tmpDepositSubtreeHeight = 0;
uint tmp = queueNumber;
while(tmp % 2 == 0) {
    uint[] memory array = new uint[](2);
    array[0] = pendingDeposits[pendingDeposits.length - 2];
    array[1] = pendingDeposits[pendingDeposits.length - 1];
    pendingDeposits[pendingDeposits.length - 2] = mimcMerkle.hashMiMC(array);
    removeDeposit(pendingDeposits.length - 1);
    tmp = tmp / 2;
    tmpDepositSubtreeHeight++;
}

if (tmpDepositSubtreeHeight > depositSubtreeHeight) {
    depositSubtreeHeight = tmpDepositSubtreeHeight;
}
```

- Full source code of Deposit function is the following.

```
    // user tries to deposit ERC20 tokens
    function deposit(
        uint[2] memory pubkey,
        uint amount,
        uint tokenType
    ) public payable {
      if ( tokenType == 0 ) {
           require(
         msg.sender == coordinator,
         "tokenType 0 is reserved for coordinator");
           require(
         amount == 0 && msg.value == 0,
         "tokenType 0 does not have real value");
        } else if ( tokenType == 1 ) {
           require(
         msg.value > 0 && msg.value >= amount,
```

```
         "msg.value must at least equal stated amount in wei");
        } else if ( tokenType > 1 ) {
            require(
        amount > 0,
        "token deposit must be greater than 0");
            address tokenContractAddress =
tokenRegistry.registeredTokens(tokenType);
            tokenContract = IERC20(tokenContractAddress);
            require(
                tokenContract.transferFrom(msg.sender, address(this), amount),
                "token transfer not approved"
            );
        }

        uint[] memory depositArray = new uint[](5);
        depositArray[0] = pubkey[0];
        depositArray[1] = pubkey[1];
        depositArray[2] = amount;
        depositArray[3] = 0;
        depositArray[4] = tokenType;

        uint depositHash = mimcMerkle.hashMiMC(
            depositArray
        );
        pendingDeposits.push(depositHash);
        emit RequestDeposit(pubkey, amount, tokenType);
        queueNumber++;
        uint tmpDepositSubtreeHeight = 0;
        uint tmp = queueNumber;
        while(tmp % 2 == 0){
            uint[] memory array = new uint[](2);
            array[0] = pendingDeposits[pendingDeposits.length - 2];
            array[1] = pendingDeposits[pendingDeposits.length - 1];
            pendingDeposits[pendingDeposits.length - 2] = mimcMerkle.hashMiMC(
                array
            );
            removeDeposit(pendingDeposits.length - 1);
            tmp = tmp / 2;
            tmpDepositSubtreeHeight++;
        }
        if (tmpDepositSubtreeHeight > depositSubtreeHeight){
            depositSubtreeHeight = tmpDepositSubtreeHeight;
        }
    }
```

3. processDeposits (Contract)

- Coordinator inserts a specified amount of deposits to the balance tree, and because the deposits are being inserted at a nonzero height, the coordinator must give a subtree index in the tree.
- It checks whether the position to insert the deposit is empty or not, by checking a attribute called `emptySubtreeRoot`.

```
require(currentRoot == mimcMerkle.getRootFromProof(emptySubtreeRoot,
subtreePosition, subtreeProof), "specified subtree is not empty");
```

- It computes the new merkle root of the balance merkle tree by inserting the deposit subtree into the balance merkle tree. The attribute of `pendingDeposits[0]` is the root of the deposit subtree.

```
currentRoot = mimcMerkle.getRootFromProof(pendingDeposits[0], subtreePosition,
subtreeProof);
```

- It removes the deposit from the queue and updates depositNumber, and returns the currentRoot.

```
removeDeposit(0);
queueNumber = queueNumber - 2**depositSubtreeHeight;
return currentRoot;
```

- Full source code of processDeposits is the following.

```
function processDeposits(
        uint subtreeDepth,
        uint[] memory subtreePosition,
        uint[] memory subtreeProof
    ) public onlyCoordinator returns(uint256){
        uint emptySubtreeRoot = mimcMerkle.zeroCache(subtreeDepth); //empty
subtree of height 2
        require(currentRoot == mimcMerkle.getRootFromProof(
            emptySubtreeRoot, subtreePosition, subtreeProof),
            "specified subtree is not empty");
        currentRoot = mimcMerkle.getRootFromProof(
            pendingDeposits[0], subtreePosition, subtreeProof);
        removeDeposit(0);
        queueNumber = queueNumber - 2**depositSubtreeHeight;
        return currentRoot;
    }
```

4. Withdraw (Contract)

- User could call withdraw function by passing the following parameters: 1) `txInfo` : withdrawal info 2) `position` : the transaction's merkle path indices root of the merkle tree 3) `proof` : the transaction's merkle path values root of the merkle tree 4) `recipent` : the recipent address, 5) `a, b, c` : the zk-SNARKs proof.
- To initiate a withdrawal, user needs to send one's tokens to zero address which triggers withdraw contract function. The function firstly verifies the existence of a withdraw transaction, verifies the signature, and transfers money from the pool to the specified address. The user signs EdDSA message specifiying the recipient address, and submits the SNARK proof of the EdDSA signatures which is verified before withdrawal to the receipient address.
- It checkes whether the token type is valid (1 or 2), and the transaction root exists in the main chain (`updates` map).

```
public {
        require(txInfo[7] > 0, "invalid tokenType");
        require(updates[txInfo[8]] > 0, "txRoot does not exist");
```

- To save the gas cost, I would improve the code by using `calldata` instead of `memory`. In addition, the code can reuse `txInfo` array to avoid the cost of copying the `txArray`.

```
uint256[] memory txArray = new uint256[](8);
    for (uint256 i = 0; i < 8; i++) {
        txArray[i] = txInfo[i];
}
```

- MiMC function hashes the details of whole transactions and generate the leaf. And then, it checks the proof of existence of transactions in the transaction merkle tree.

```
uint256 txLeaf = mimcMerkle.hashMiMC(txArray);
require(txInfo[8] == mimcMerkle.getRootFromProof(txLeaf, position, proof),
"transaction does not exist in specified transactions root");
```

- The message is hash of nonce and recipient address. And then, by using given SNARK proofs, verifying the validity of user's EdESA signatures. To save the gas fee, creating array with memory declaration should be avoided. We might be able to pass the parameters directly by modifying the inner signature of MiMC function.

```
uint[] memory msgArray = new uint[](2);
msgArray[0] = txInfo[5];
msgArray[1] = uint(recipient);

// As-is
require(withdraw_verifyProof(a, b, c, [txInfo[0], txInfo[1],
mimcMerkle.hashMiMC(msgArray)]), "eddsa signature is not valid");
```

```
// To-be
require(withdraw_verifyProof(a, b, c, [txInfo[0], txInfo[1],
mimcMerkle.hashMiMC(txInfo[5], uint(recipient))]), "eddsa signature is not
valid");
```

- Let us transfer token on token contract. The transfer supports both ETH or ERC20 tokens. ERC20 tokens should be preliminiarly registered with the smart contract.

```
if (txInfo[7] == 1){
            // ETH
            recipient.transfer(txInfo[6]);
        } else {
            // ERC20
            address tokenContractAddress =
tokenRegistry.registeredTokens(txInfo[7]);
            tokenContract = IERC20(tokenContractAddress);
            require(
                tokenContract.transfer(recipient, txInfo[6]),
                "transfer failed"
            );
        }

        emit Withdraw(txInfo, recipient);
```

5. UpdateStateVerifier (circuit)

- The public input and output of the circuit is the following.
  - `tx_root` : merkle root of a transaction tree sent to the coordinator
  - `current_state` : merkle root of previous accounts tree
  - `out` : merkle root of updated accounts tree
- The circuit is to prove whether transactions were done correctly. It verifies the transactions by the following.
  - checking whether transaction actually exists or not in the `tx_root`
  - checking the transactions signatures to validate whether the transaction is signed by the sender correctly
  - validating that the sender account exists in `current_state`
  - validating that the sender has `sufficient` balance enough to send the transaction
  - the state update triggers whether decrementing the `sender balance` and increasing the `sender nonce`
  - intermediate root update from current state `intermediate_root` to `intermediate_root_1` using new sender leaf
  - receiver state is to be updated by getting incremented `receiver balance`
  - update `intermediate_root_1` with new receiver leaf to render `intermediate_root_2`

- `n` is depth of balance tree, `m` is depth of transactions tree, for each proof, update `2**m` transactions

```
template Main(n,m) {
```

- All transaction information are kept secret, save for the merkle roots of the balance and transaction trees. On the L2 scale, this protects the secrecy of transaction data.
- However, because the transaction data is not saved on-chain, this undermines data availability. The L2 must ensure that the transaction data is securely kept and that its availability should be always guaranteed.
- The root of the new transaction tree is consumed by the smart contract. The contract updates the merkle tree to the L1 chain.

```
// public input
signal input txRoot;

// Belows are private inputs

// Merkle proof for transaction in tx tree
signal private input paths2txRoot[2**m, m];

// Merkle proof for transaction in tx tree
signal private input paths2txRoot[2**m, m];

// binary vector indicating whether node in tx proof is left or right
signal private input paths2txRootPos[2**m, m];
```

- The following variable is the merkle tree root which is yet to be added with the additional transactions. The smart contract consumes the variable and compares it to the merkle root on the L1 chain.
- By verifying this, one can guarantees that the proof is formed correctly from the right state. It also can be verified that the circuit has not been impaired (not spoofing the old merkle root!!!).

```
signal input currentState;
```

- Two intermediate merkle tree roots are computed for each transaction, it is composed of send part and receive part of the transaction. The input array is used to check the validity of intermediate merkle roots.

```
// intermediate roots (two for each tx), final element is last.
signal private input intermediateRoots[2**(m+1)+1];
```

- When it comes to the constant zero address, there is a special transaction to withdraw to the

zero address. It does not change the merkle root, the balance and nonce of `zero_leaf`.

```
var ZERO_ADDRESS_X = 0;
var ZERO_ADDRESS_Y = 0;
```

- The value of the `intermediateRoots[0]` must be same as the merkle root of the old transactions merkle tree.

```
currentState === intermediateRoots[0];
```

- The following loop logic checks the validity of the transactions and signatures, and the validity of the sender/receiver account. When checking them, it also validates the balance, nonce, and tokenTypes of sender's account.
- In addition, it also checks the intermediate merkle root of the balance tree that should be updated after sending the transaction leaf. It also checks the existence of the receiver on the chain and updates the balance of the receiver.
- Finally, the business logic checks the intermediate merkle root of the balance after processing the balance of the receiver.

```
for (var i = 0; i < 2**m; i++) {

        // transactions existence and signature check
        txExistence[i] = TxExistence(m);
        txExistence[i].fromX <== fromX[i];
        txExistence[i].fromY <== fromY[i];
        txExistence[i].fromIndex <== fromIndex[i];
        txExistence[i].toX <== toX[i];
        txExistence[i].toY <== toY[i];
        txExistence[i].nonce <== nonceFrom[i];
        txExistence[i].amount <== amount[i];
        txExistence[i].tokenType <== tokenTypeFrom[i];

        txExistence[i].txRoot <== txRoot;

        for (var j = 0; j < m; j++){
            txExistence[i].paths2rootPos[j] <== paths2txRootPos[i, j] ;
            txExistence[i].paths2root[j] <== paths2txRoot[i, j];
        }

        txExistence[i].R8x <== R8x[i];
        txExistence[i].R8y <== R8y[i];
        txExistence[i].S <== S[i];

        // sender existence check
```

```
        senderExistence[i] = BalanceExistence(n);
        senderExistence[i].x <== fromX[i];
        senderExistence[i].y <== fromY[i];
        senderExistence[i].balance <== balanceFrom[i];
        senderExistence[i].nonce <== nonceFrom[i];
        senderExistence[i].tokenType <== tokenTypeFrom[i];

        senderExistence[i].balanceRoot <== intermediateRoots[2*i];
        for (var j = 0; j < n; j++){
            senderExistence[i].paths2rootPos[j] <== paths2rootFromPos[i, j];
            senderExistence[i].paths2root[j] <== paths2rootFrom[i, j];
        }

// balance checks
        balanceFrom[i] - amount[i] <= balanceFrom[i];
        balanceTo[i] + amount[i] >= balanceTo[i];

        nonceFrom[i] != NONCE_MAX_VALUE;

        //-----CHECK TOKEN TYPES === IF NON-WITHDRAWS-----//
        ifBothHighForceEqual[i] = IfBothHighForceEqual();
        ifBothHighForceEqual[i].check1 <== toX[i];
        ifBothHighForceEqual[i].check2 <== toY[i];
        ifBothHighForceEqual[i].a <== tokenTypeTo[i];
        ifBothHighForceEqual[i].b <== tokenTypeFrom[i];
        //-----END CHECK TOKEN TYPES-----//
```

- In addition to the aforementioned point, I think that `Full Exit` Mode would be viable to withdraw user's funds without need to communicate with operator considering the emergency case where the single point of failure problem arises due to the lack of credibility of single operator.

Question 2.2

> [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). The justification in the event any of the tests fails.

- The unit test fails in `fee_ticker` that attempting to test `coingecko_api`. The runtime from `tokio::test` attempts to create an instance of a new tokio runtime.

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

- Memory Pool is a memory buffer which is used to store transactions from the zkSync network and runs them on server nodes. The pool accepts the transactions, check signatures and validate nonce correctness.
- Transactions are stored in `MempoolTransactionQueue`, which manages the following memory transaction queues.
    - The file is in `zksync/core/bin/zksync_core/src/mempool/mempool_transactions_queue.rs`
    - `reverted_txs`: This is a reverted transaction queue that must be utilized before any other transactions are processed. Unless the server is rebooted after undoing blocks, it will usually be empty. Only popping elements have access to the queue.
    - `priority_ops`: The queue to store priority transactions (i.e. deposit, withdraw)
    - `ready_txs`: The transactions ready for exeuction.
    - `pending_txs`: The transactions that have not yet ready due to the `valid_from` field.
    - Those following transactions or data in memory pool will be completely lost on node shutdown.
    - The queue is initalized from DB when calling the function `MempoolState::restore_from_db()`.
- zkSync transaction request handler runs their own thread with `MempoolTransactionsHandler`. It runs the requests which is the following.
    - The file is in `zksync/core/bin/zksync_core/src/mempool/mod.rs`.
    - `MempoolTransactionRequest::NewTx` : adding new transactions to `MempoolTransactionsQueue` which is the pending queue of transactions. (`pending_txs`).
    - The transactionis moves to `ready_txs` which is ready queue when `valid_from` field is less than block timestamp.
    - `MempoolTransactionRequest::NewTxsBatch` : adding the batch of transactions to `MempoolTransactionsQueue` which is the pending queue of transactions.
    - `MempoolTransactionRequest::NewPriorityOps` : adding priority transactions to `MempoolTransactionsQueue` priority transactions.
- zkSync block request handler runs their own thread with `MempoolBlocksHandler`. It runs the requests which is the following.
    - The file is in `zksync/core/bin/zksync_core/src/mempool/mod.rs`.
    - The MempoolTransactionsQueue is used to create the block in this sequence: 1) `reverted_txs`, 2) `priority_ops`, 3) `ready_txs` 4) `MempoolBlocksRequest::UpdateNonces(AccountUpdates)`.
    - `MempoolBlocksRequest::UpdateNonces(AccountUpdates)` handles the request to update account tree, `MempoolState`. The function creates account, deletes account, updates nonce, and changes account pubkey hash.

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

- The ETHSender module, specified in `core/bin/zksync/eth_sender/src/lib.rs`, may

synchronize ZKSync activities with the Ethereum blockchain by generating transactions from the operations, sending them, and verifying that each transaction is completed and confirmed correctly. The sequence of operations is also preserved that the older operation of `commit` will always be committed before the younger one.

- An event loop, which should be operated on a separate thread, acquires the operations to commit over the channel and then commits them to the Ethereum, guaranteeing that all transactions are correctly included in blocks and processed, is an important aspect of this framework.
- Before transmitting a signed transaction to Ethereum, it is stored to a database to ensure that its state is always recoverable. It takes care of an ongoing operation (perform commitment step()) by verifying its condition and doing the following actions:

1. Stop the execution if the transaction is either pending or finished (as there is nothing to do with the operation yet).
2. Sends a supplement transaction if the transaction is stalled.
3. Handle the failure according to the failure processing policy if the transaction fails.

- The transaction being synchronized to Ethereum blockchain requires the following 4 phases.
    - `CommitBlocks` : The transaction, or the state root, which is committed to Ethereum blockchain through the L1 smart contract is included in ZKSync block.
    - `CreateProofBlocks` : For the transactions in this block, the prover program creates a ZK-SNARK proof.
    - `PublishProofBlocksOnchain` : To validate the block change, a ZK-SNARK proof is given to the L1 smart contract.
    - `ExecuteBlocks` : ZKSync is used to perform transactions which is validated by L1 smart contract in the block.

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

- The essential part of this structure is `maintain` function in an infinite loop and adds data to the database. It awaits `rounds_interval` time between updates. This will generate and store in db witnesses for blocks with indexes `start_block, start_block + block_step, start_block + 2*block_step,` and vice versa.

Question 2.3

> 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.

- Advantages
    - The good news is that users are able to choose either option based on their preferences & the trade-offs presented. Basically, each user is able to choose their own amount of security. zkPorter will offer negligible cost but lower security for trivial transactions. The user can choose which transactions to be executed within an account. Reference

- Shortcomings
    - However, it should be noted that, when utilizing zkPorter, the user is relying on zkSync's internal consensus mechanism. This requires the user to place their trust in `@the_matter_labs` & rely on a far less secure or decentralized layer that leverages L1's consensus mechanism. Reference

Question 3.

Question 3.1

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

- Coda, a fully-succinct blockchain technology that allows clients to validate the whole chain by confirming a short cryptographic proof in the tens of kilobytes, is one of the most well-known at the present. Other blockchain scaling methods have also been proposed, such as recursively demonstrating the validity of Bitcoin's proof of work consensus mechanism and employing limited recursion to efficiently validate aggregated signatures for mobile clients at Celo, which utilizes the developer utility tool to convert binary RLP (Recursive Length Prefix) dumps (data encoding used by the Celo protocol both network as well as consensus wise) to user friendlier hierarchical representation.
- Those who tries to utilize recursive SNARKs to provide a light client experience while maintaining complete node security, has an expectation to still have full node security and inclusive responsibility.
- Recursive SNARKs let a side to offer a proof of a proof, further compressing the data. If Bitcoin adopted this strategy, it would obtain nearly limitless transactional capacity. However, there is a cost to this scalability: if parameters aren't correctly specified, the network's security may be jeopardized. This technology underpins Zcash (ZEC), a privacy-focused cryptocurrency.
- The Fractal would be to completely eliminate the need of elliptic curves in the proof system by designing and employing a proof system that just requires a field F and a cryptographic hash H to be instantiated. This accomplishes `quantum security` in an unexpected way: by eliminating assumptions rather than adding them. We know this approach can be implemented to be secure against a quantum opponent since quantum-resistant hashes exist. Proof sizes, on the other hand, are now fairly big, ranging from 80 to 200 kB.
- Finally, Halo is a newly created proof system that tries to enhance recursion efficiency by using elliptic curve cycles that do not leverage pairings, while also eliminating the need for a trusted setup by employing a non-preprocessing proof system. This is analogous to Bulletproofs in that we are unable to create a concise cryptographic description of our circuit due to a lack of setup. Reference

Question 3.2

> What is Kimchi and how does it improve PLONK?

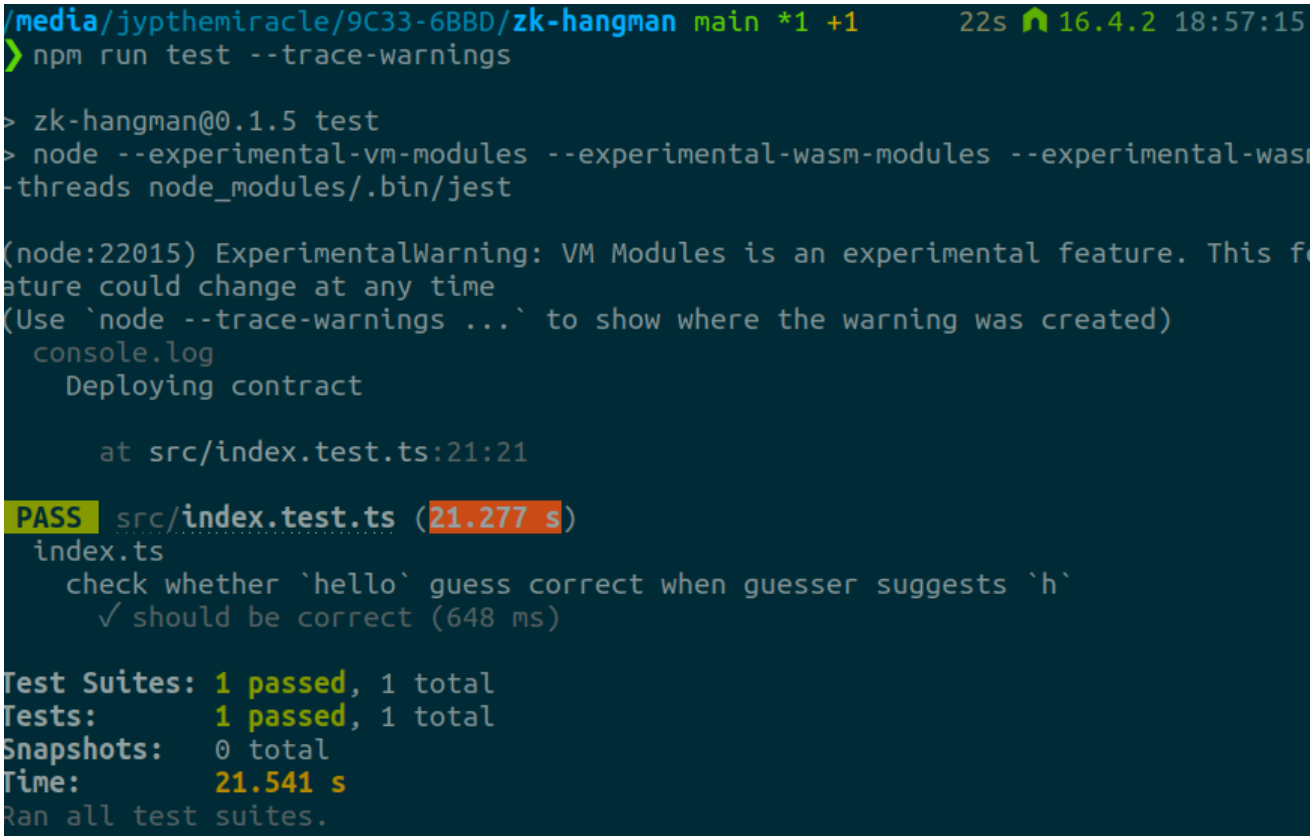
Link

- zkApps (zero-knowledge smart contracts) may be authored in typescript using the snarkyjs library in Mina, and then compiled to an intermediate form using snarky. The program may then be compiled into the prover and verifier indexes using a Kimchi compiler, and both sides can utilize Kimchi's built-in features to create and verify proofs. The verifier index is published on the blockchain, allowing anybody to check evidence contained in transactions that claim to have successfully run a zkApp.
- On top of PLONK, Kimchi is a collection of enhancements, optimizations, and modifications. It avoids PLONK's trusted setup restriction, for example, by including a bulletproof-style polynomial commitment within the protocol. There is no need to trust that the individuals in the trustworthy setting were truthful this manner (if they were not, they could break the protocol). Since we're on the subject of circuits, Kimchi adds 12 more registers to the three that PLONK already had.
- These registers are divided into two groups: IO registers that can be linked together and temporary registers (also known as advice wires) that can only be utilized by the corresponding gate. With more registers, we can now create gates that accept several inputs rather than just one. A scalar multiplication gate, for example, would require at least three inputs (a scalar, and two coordinates for the curve point). Because certain processes occur more frequently than others, they can be rebuilt as new gates to save time. At the time of writing, Kimchi has nine new gates.
- Another Kimchi's notion is that a gate can write its output straight to the registers required by the following gate. This is useful in gates like "poseidon," which must be used several times (11 times in this case) to represent the poseidon hash function. Lookups are another speed optimization made in Kimchi. The size of an XOR table for values of four bits is 28. Because doing this with generic gates would be difficult and time-consuming, Kimchi creates the database and allows gates to execute a simple search into the table to retrieve the operation's outcome.

Question 3.3

> [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.

- Go to the snapp: https://github.com/jypthemiracle/snapp-hangman
- The code of a unit test: https://github.com/jypthemiracle/snapp-hangman/blob/main/src/index.test.ts



Question 3.4

> [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).

- The following processes the rollup deposits.

```
@branch static processDeposit(
    pending: MerkleStack<RollupDeposit>,
    accountDb: AccountDb
  ): RollupProof {
```

- To begin with, the function retrives the current state of the rollup. After that, it pops the element which deposited at the last from the stack. It also asserts to make sure the account you're depositing into doesn't already exist.

```
let before = new RollupState(pending.commitment, accountDb.commitment());
let deposit = pending.pop();
let [{ isSome }, mem] = accountDb.get(deposit.publicKey);
isSome.assertEquals(false);
```

- The function initalizes the rollup account. The account has zero balance, and nonce is set to 0.

```
let account = new RollupAccount(
      UInt64.zero,
      UInt32.zero,
      deposit.publicKey
);
accountDb.set(mem, account);
```

- The function retrieves the state after rollup is finished.

```
let after = new RollupState(pending.commitment, accountDb.commitment());
```

- Finally, the function returns the proof that validates the state transition.

```
return new RollupProof(new RollupStateTransition(before, after));
```

- When it comes to the `transaction` function, it asserts whether the signature provided was signed correctly by the sender for the specified transactions.

```
@branch static transaction(
    t: RollupTransaction,
    s: Signature,
    pending: MerkleStack<RollupDeposit>,
    accountDb: AccountDb
  ): RollupProof {
    s.verify(t.sender, t.toFields()).assertEquals(true);
    let stateBefore = new RollupState(
      pending.commitment,
      accountDb.commitment()
    );
```

- After retrieving the current state of the rollup in `stateBefore`, it asserts whether the sender account exists and the nonce of specified transactions matches as expected. If the case, the function subtracts the amount of sent from the sender's balance and increment the nonce. Finally, it updates the sender account's state as we had made the changes in the block.

```
let stateBefore = new RollupState(
      pending.commitment,
      accountDb.commitment()
    );

let [senderAccount, senderPos] = accountDb.get(t.sender);
    senderAccount.isSome.assertEquals(true);
    senderAccount.value.nonce.assertEquals(t.nonce);

    senderAccount.value.balance = senderAccount.value.balance.sub(t.amount);
    senderAccount.value.nonce = senderAccount.value.nonce.add(1);

    accountDb.set(senderPos, senderAccount.value);
```

- Let us retrieve the account of receiver. The function adds the balance of receiver's account, and update the state of the account.

```
let [receiverAccount, receiverPos] = accountDb.get(t.receiver);
receiverAccount.value.balance = receiverAccount.value.balance.add(t.amount);
accountDb.set(receiverPos, receiverAccount.value);
```

- Retrieving the state of rollup, and returning a proof that validates the state transition.

```
let stateAfter = new RollupState(
      pending.commitment,
      accountDb.commitment()
);

return new RollupProof(new RollupStateTransition(stateBefore, stateAfter));
```

- The following code defines how rollup proofs merged. It asserts whether the result of the first proof goes the second proof's input. It also returns a proof that validates the fact that the transition from the state of first proof goes to the state of second state well.

```
  @branch static merge(p1: RollupProof, p2: RollupProof): RollupProof {
    p1.publicInput.target.assertEquals(p2.publicInput.source);
    return new RollupProof(
      new RollupStateTransition(p1.publicInput.source, p2.publicInput.target)
    );
  }
```

Question 5: Thinking in ZK

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

- I would like to ask the consensus layer in Mina blockchain. Choosing the block producers and managing them would be important when considering the intermittent network failure of Solana. How can Mina protocol improve the developer experience for ordinary developers, since it seems that version changes have dramatically alters the way of coding (i.e. not allowing superceding the constructors of a `Smart Contract` class.)
- To zksync, I would like to ask how they have their roadmap to replace current EVM with zk-supported EVM to integrate zk technologies into the Ethereum mainnet.
- Comparing to Polygon's MidenVM, what edges that zkSync has?