ZKU Week 2 Assignment

Question 1.1

> Explain in brief, how does the existing blockchain state transition to a new state? What is the advantage of using zk based verification over re-execution for state transition?

- The blockchain state transition from existing state to new state happens by a consensus mechanism, which means that the list of transactions from executing node are “approved” and “written” into the block by validating node. The state transition function, called `STF(current_state, input_data) = new_state` is used to replicate all across distributed nodes by re-executing the transactions. Thus, this process by validating nodes renders the duplication of validity, which leads to the lack of privacy, and scalability problem. ZKP could solve this problem by suggesting cryptographic technology. First of all, with ZKP technology, one can prove that he/she knows something without revealing about the secret. In addition, One can verify that the bunch of complex computations had worked correctly, without re-executing those computations.

Question 1.2.1

> Explain in brief what is a ZK VM (virtual machine) and how it works?

From Koh Wei Jie (https://medium.com/zeroknowledge/zero-knowledge-virtual-machines-the-polaris-license-and-vendor-lock-in-ab2c631cf139), he mentioned that ZK-VM is a circuit that executes bytecode, which allows a prover to show that one have correctly executed the program code on given a set of inputs and some bytecodes. Yes, a ZK-VM is a trans-compiler which means that the abstraction layer makes it possible for developers to create solidity contracts and iterate quickly when dealing with zkp programs. There are two viable projects, 1) zkSync 2) MidenVM from Polygon.

Question 1.2.1

> Give examples of certain projects building Zk VMs (at-least 2-3 projects). Describe in brief, key differences in their VMs.

1. MidenVM (https://github.com/maticnetwork/miden): Polygon Miden is a ZK-VM written in Rust, which allows any programs could retrieve a STARK-based proof automatically. The proof can be used by anyone to verify that a program had executed spot on without the

need for re-executing related transactions. It works as a stack based machine, which has ‘initalize’, ‘push’ and ‘read’ values from the stack.

2. zkSync (https://github.com/matter-labs/zksync): zkSync is based on zk Rollup, which has 1) a smart contract in L1 blockchain, 2) a prover/worker application in off-chain network, 3) a server application that runs the zkSync network. To specify, zkSync contract which is deployed on the Ethereum blockchain manages the user’s balances and verifies the correctness of operations in zkSync network. Second of all, prover application creates a proof for an executed block by polling server application for available its jobs. If there is a new block, server application provides a witness to prover. The prover generates proof and reports to the Server application. The server application publisdhed the proof to the smart contract. The server application does the following things, 1) monitoring the smart contract for the onchain operations 2) accepting transactions 3) generating zkSync chain blocks 4) requesting proofs for executed blocks 5) publishing data to the smart contract.

Question 1.2.2

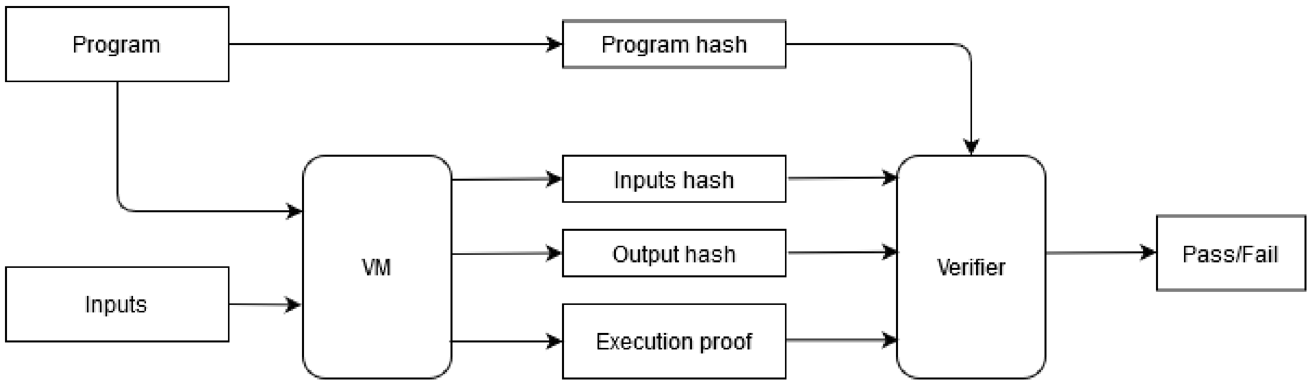
> What are the advantages and disadvantages of some of the existing Zk VMs?

- Miden M: The zk-VM has its advantages when it comes to the ability to automatically generates proof when putting a program into the machine. However, the developer needs to learn a new language called `Miden Assembly`. The team has pledged to support ‘Ethereum-compatible’ to entertain more developers into the project.
- zkSync: The project allows developers to deploy zk-based smart contracts in Solidity language. In addition, it states that they were very conservative with respect to security choices made in the protocol, while every components relies execlusively on well-established cryptographic assumptions. However, since zkSync has its own mainnet, the network should be checked in decentralized fashion since it could misuse its ability by preventing specific transactions not to be included in a block (e.g. malicious MEV or etc.)

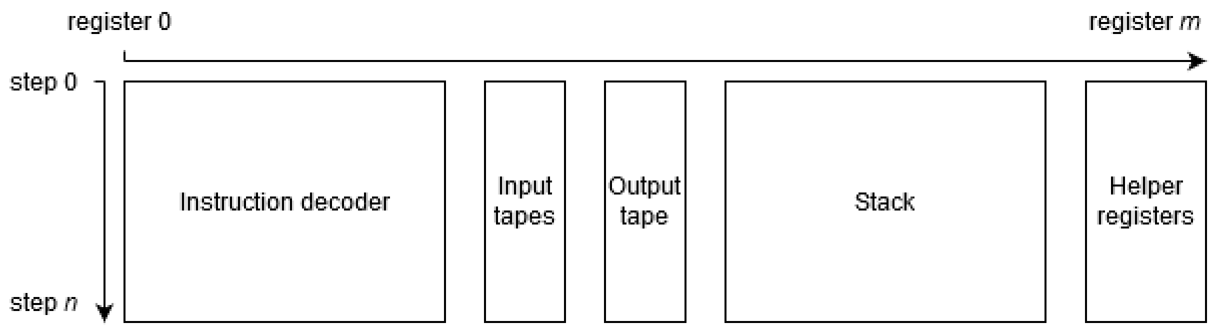
Question 1.2.3

> Explain in detail one of the Zk VM architectures using diagrams.

Miden VM



According to bobbinth (https://ethresear.ch/t/a-sketch-for-a-stark-based-vm/7048), the ZK-VM does the following. To begin with, it takes two inputs which is 1) a program 2) and a set of inputs for the program. After taking those inputs, the VM executes the program with the given set of inputs. The program returns 1) hash of the inputs, 2) hash of the outputs generated by the program, 3) a STARK proof's hash pertaining to the program's proper execution. On the other side, the verifier takes the following: 1) the hash of the program, 2) the hash of the inputs, 3) the hash of the outputs, and uses them to verify the STARK proof.



To be honest, basically, the ZK-VM is merely a stack-based machine. It basically has 1) instruction decoder, which is in charge of decoding instructions into binary form and constructing a continuous hash of the instructions that have been performed that takes around 14 registers. 2) two input tapes, one contains inputs to be pushed onto the stack, and other ones to aid in execution of certain instructions. They are eventually hashed into one single text. 3) output tape, which is the tape that defines output values of the program. 4) stack registers, that represents stack e.g. the first register being the top of the stack, and second one being the second from the top. 5) Lastly but not least, helper registers is a small set of registers that aids in execution of certain complex instructions.

To simply put, the above image from zk Hack Mini (https://www.youtube.com/watch?v=Q8xYSOx82U0) shows how the miden VM architecture works: You compose an assembly program (MAST: master abstract syntax tree), which is then compiled into bytecode by the assembler. The bytecode is a series of digits that individually lead to an opcode, some restrictions, some data through a location pointer, and flags. After being compiled into bytecode, the program is executed via a Turing complete machine, which delivers the program's final state as a hash.

Question 2.1

> What is Semaphore? Explain in brief how it works? What applications can be developed using Semaphore (mention 3-4)?

Users of Ethereum can use Semaphore to authenticate their membership in a set without disclosing their true identity. At the same time, it allows users to indicate their approval of any string. It's intended to provide a straightforward and general privacy layer for Ethereum dApps. Private voting, whistleblowing, mixers, and anonymous authentication are examples of use cases.

Question 2.2.1

Question 2.2.2

> Explain code in the sempahore.circom file (including public, private inputs).

- `CalculateSecret()` : The circuit uses `Poseidon` algorithm to get private information after hashing it and blur it. DUSK Network's Poseidon (https://dusk.network/news/poseidon-the-most-efficient-zero-knowledge-friendly-implementation) is a hash function for zero-knowledge proofs to be generated and validated. It says that the algorithm uses up to 8x fewer constraints per message bit than its competitor hash functions, such as Pedersen Hash. Specifically, it is used at Semaphore function to 1) calculate the secret based on the given inputs (identifyNullifier, identityTrapdoor) 2) initiate signal and puts it into the calculateSecret output.

```
component calculateSecret = CalculateSecret();
calculateSecret.identityNullifier <== identityNullifier;
calculateSecret.identityTrapdoor <== identityTrapdoor;

signal secret;
secret <== calculateSecret.out;
```

- `CalculateIdentityCommitment()` : The circuit hashes two of the following: 1) The hash of the identify nullfier 2) The identity trapdoor that generates the identity commitment based on secret, at Semaphore function.

```
component calculateIdentityCommitment = CalculateIdentityCommitment();
calculateIdentityCommitment.secret <== secret;
```

- `CalculateNullifierHash()` : The circuit hashes the identifier and the external nullifer, and it ensures that the given hash matches the same, and the nullifiers hash has not been suggested so that it can prevent double-signaling.

```
// Semaphore()
// The nullifier hash is calculated based on given external nullifier and ident
component calculateNullifierHash = CalculateNullifierHash();
calculateNullifierHash.externalNullifier <== externalNullifier;
calculateNullifierHash.identityNullifier <== identityNullifier;

// returns the nullifier hash
nullifierHash <== calculateNullifierHash.out;
```

- `Semaphore()` : The function provides the hash of an public key - identity commitment - and random secrets - identity nullifier and identity trapdoor - to the contract by storing them into a Merkle tree, using all above functions.

```
component inclusionProof = MerkleTreeInclusionProof(nLevels);
inclusionProof.leaf <== calculateIdentityCommitment.out;

 for (var i = 0; i < nLevels; i++) {
        inclusionProof.siblings[i] <== treeSiblings[i];
        inclusionProof.pathIndices[i] <== treePathIndices[i];
}
```

- This action happens after a user registers his/her identity. The function broadcasts a signal by proving membership that one is a registered user set, without revealing one's identity while preventing double-spending.

Question 2.3.1

> What potential challenges are there to overcome in such an authentication system?

- This authentication system works by the following. 1) generate a zk identity 2) submit the identity to the contract by verifying the identity and generating a token. The key is that the user can use this given token for accessing other platforms by **not leaking one's personal address or sensitive information**. It is clear that other web3 or web2 platforms should support the elefria smart contracts.

- It is obvious that 1) how to make outer platforms appreciate zk-based identification system? It is crucial for the project to make its edge over another authentication system like OAuth 2.0. For example, it could provide KYC process for the address so that one can confirm that only unique but only one address can be given to the unique person. (e.g. it could be used by democratic but private voting.) 2) This leads the project's burden to be verified and audited by its consumers. Since the understanding of zk based system is difficult, more convenient materials for developers are seriously required.

Question 3.1.

> Compare and contrast the circuits and contracts in the two repositories above (or consult this article), summarize the key improvements/upgrades from tornado-trees to tornado-nova in 100 words.

- According to Tornado Cash Medium (https://tornado-cash.medium.com/tornado-cash-introduces-arbitrary-amounts-shielded-transfers-8df92d93c37c), Tornado Nova allows user to deposit and withdraw arbitrary amounts of ETH. In contrast, before Tornado Nova, All Tornado Cash pools render users to only deposit and withdraw a fixed amount of a given token within each pool. Tornado Cash Nova uses the Gnosis Chain as a Layer2, which means that users can benefit from cheaper fees and enjoy fast transactions. Users can either connect one's wallet to transfer ETH or uses a Relayer. When using a Relayer, one could wash away worries since it prevents the possibility of compromising the anonymity of the transfer, by severing the link with used ETH and his/her identity.

Question 3.2.1.

> Take a look at the circuits/TreeUpdateArgsHasher.circom and contracts/TornadoTrees.sol. Explain the process to update the withdrawal tree (including public, private inputs to the circuit, arguments sent to the contract call, and the on-chain verification process).

- To begin with, the circom file hashes all of the given arguments, and compress them into the merkle tree, which is to reduce gas fees.
- The contract has a deposit merkle tree, and a withdrawal merkle tree. The contract uses two trees to validate with zk proof that user had sent a proper withdrawl request with correct amount of funds when doing deposits and withdrawals.
- The updateDepositTree method of tornado contract takes the following public inputs: 1) _currentRoot which is current merkle tree root, 2) _newRoot which is updated merkle tree root, 3) _pathIndices which is Merkle path to inserted batch, 4) _events which is A batch of inserted events (leaves). In addition, this also has private inputs including _proof which is a snark proof that elements were inserted correctly and _argsHash which is a hash of snark inputs. They are all private inputs.

- According to the business logic of the contract, it checks the inputs that the proposed updates are available. If so, the tree deposit data is added to the tree. Then, the tree is updated. The arguments hash communicates with the circom verifier to be verified, and if it passes all, the new root is calculated and uploaded to the blockchain.

Question 3.2.2.

> Why do you think we use the SHA256 hash here instead of the Poseidon hash used elsewhere?

- To begin with, when it comes to a speed issue, bobbinth from ethresear.ch (https://ethresear.ch/t/performance-of-rescue-and-poseidon-hash-functions/7161) claims that Poseidon is almost 3x faster than Rescue, and both are significantly slower than sha256 (this was expected). Specifically, Poseidon is about 30x slower, and Rescue is about 80x slower than sha256. Since Torando Cash deems the speed issue is one of its priority, the project should utilize SHA256 over the Poseidon.
- It is to note that Poseidon is dedicated for merkle tree transactions (https://www.usenix.org/system/files/sec21summer_grassi.pdf). zk-SNARK computation has a very large amount of multiplication so that Poseidon is better choice than SHA256. However, Solidity does basic addition and multiplication features that leads to choose SHA256. To reduce gas fee for EVM choices, the project chose SHA256 in expense for the costs for circuit generations.

Question 3.3.1

Question 3.3.2

> Add a script named custom.test.js under test/ and write a test for all of the followings in a single it function.

- Here is the personal gist link (https://gist.github.com/jypthemiracle/c22c51ff803f0466ce46b17cb6551054) for custom.test.js

Question 4.1

> If you have a chance to meet with the people who built Tornado Cash & Semaphore, what questions would you ask them about their protocols?

I would ask how to solve the problem of 1) data availability 2) the centralization of relayers. To begin with, when depositing zkp proofs, it requires some large data storages by using shard or another modular blockchain like celestia. What solutions to put its rollup to which places would be appropriate? Secondly, as the universe of Layer 2 grows, there would be more needs to communicate among Layer 2 chains. If relayers that helps chain to communicate each other are centralized to priortize performances over security, how we can solve this problem with ruling out more complexity on its architectures?

I'd like to know how to address the issue of 1) data availability. 2) Relayer centralized control. To begin, utilizing shard or another modular blockchain like celestia to deposit zkp proofs necessitates certain big data storages. What options would be ideal for putting its rollup in which locations? Second, as the Layer 2 world expands, more communication between Layer 2 chains will be required. How can we tackle this problem without adding additional complexity to its structures if the relayers that assist the chain connect with each other are centralized to prioritize efficiency above security?

Question 4.2

> Regarding writing and maintaining circuits for each dapp separately, what are your thoughts about using just one circuit for all dapps? Is that even possible? What is likely to be a standard in the future for developing Zk dapps?

I believe that as the position of OpenZepplien in Ethereum ecosystem, there would be 'certified' circuits that is applicable for all dApps for zk products. In addition, if zk circuits could be compiled into any solidity codes, dApp developers would consider more optimistically in applying zkp in their products.