Zero Knowledge University (Assignment-2)

This is the second assignment report in the second week of Zero-Knowledge University (ZKU) cohort. In the second week of the cohort, ZKU teaches the student about Privacy, Zero Knowledge Virtual Machine, Semaphore, Tornado Cash, Tornado Nova, Tornado Trees. The assignment consist of 4 (four) question which is answered in this report.

Based on my attempt to complete the assignment, I write this report by grouping the question into 4 (four) headings below. All of my work can be found at https://github.com/Rizary/zero-knowlegde-university-report in the **zku-assignment-2** branch.

Privacy and ZK VMs

Blockchain state transition, and advantages of verification over re-execution

A. State transition in the Blockchain

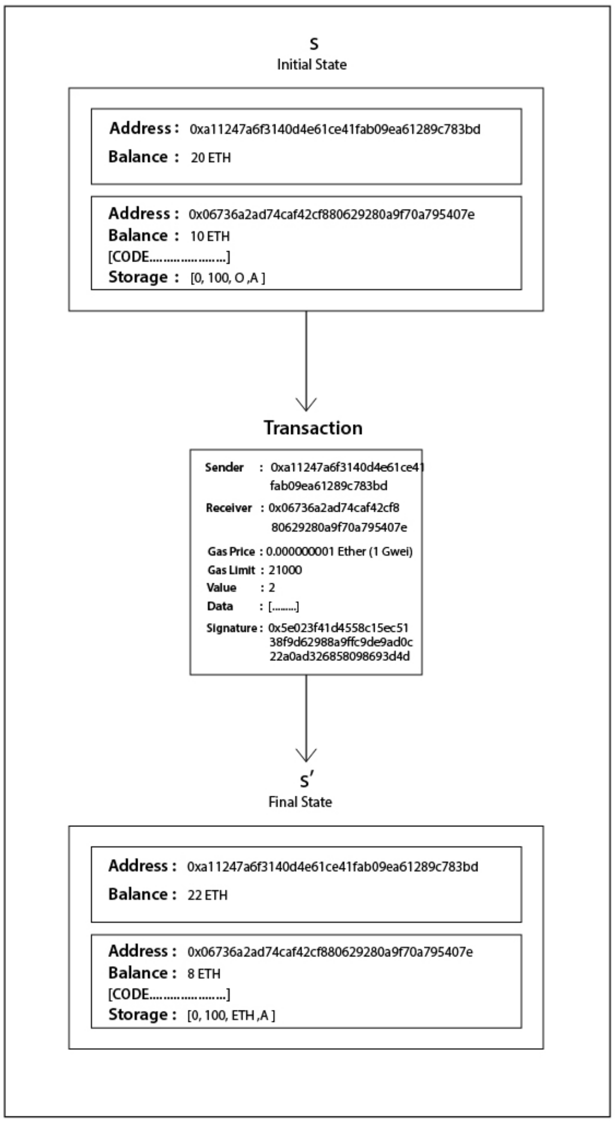
In general, every blockchain is a replicated state machine. It start with the genesis state, or the first state captured as a start of the blockchain. That is why if we want to create blockchain, there will be a concept called genesis block. In order to change the blockchain state, there is a helper function called **State Transition Function** (**STF**) which takes 2 parameter, the "current state" and "input data". The output of this function is a "new state".

STF can be implemented either as on-chain logic or as user defined logic (smart contract running on virtual machine). The step needed to make a change of blockchain state are:

1. Submitting input data as a transaction to the blockchain network;

2. Verify the transaction by a group of people (usually miners for proof of work, and validator for proof of stake) to check the validity;

3. If the majority of them (the group) reach an agreement or consensus, the chain state will be altered and the transaction will be permanently stored in the blockchain.


S
Initial State
Address : 0xa11247a6f3140d4e61ce41fab09ea61289c783bd
Balance : 20 ETH
Address : 0x06736a2ad74caf42cf880629280a9f70a795407e
Balance : 10 ETH
[CODE.....................]
Storage : [0, 100, O ,A]
Transaction
Sender : 0xa11247a6f3140d4e61ce41fab09ea61289c783bd
Receiver : 0x06736a2ad74caf42cf880629280a9f70a795407e
Gas Price : 0.000000001 Ether (1 Gwei)
Gas Limit : 21000
Value : 2
Data : [........]
Signature : 0x5e023f41d4558c15ec5138f9d62988a9ffc9de9ad0c22a0ad326858098693d4d
S′
Final State
Address : 0xa11247a6f3140d4e61ce41fab09ea61289c783bd
Balance : 22 ETH
Address : 0x06736a2ad74caf42cf880629280a9f70a795407e
Balance : 8 ETH
[CODE.....................]
Storage : [0, 100, ETH ,A]

Ethereum State Transition function

The above picture is an example of how the STF in the Ethereum network works. In this case, any newcomers can independently generate the current state of the blockchain by re-executing all the transaction. It means, verifying the transaction is done by the group through re-executing the STF using the user input data and comparing the results with each other. However, there are some problems in doing the re-executions.

B. Verification over re-execution

The first problem in doing re-execution is about the “**Privacy**”. It is because both the input data and the STF are exposed to the public, so we cannot send private data in the blockchain. The second is about the “**Scalability**”. The reason is because doing a re-execution on each transaction will cause an overhead due to the time it takes to verify each transaction. The next problem is about “**Computation restriction**” which in general (i.e. Ethereum), there is some limitation on smart contract functionality in the blockchain when it comes to perform complex scientific computation. This computation can cost very high gas fee, especially if every node need to be re-executed for verifying the transaction. The last problem is about “**State Explosion**”. State explosion is happened because the need of storage whenever the re-execution happened. It means that doing the re-execution will need hundreds of GB storage or more to store the transaction history which subsequently will caused small node runners quitting the network due to high storage cost. This will hurts the essence of decentralization as only big player can participate.

The solution to these problem that happened in re-execution is by doing verification using Zero-Knowledge Proof. The advantage of doing verification instead of re-execution is about **zero knowledge** and **succinctness**. With verification, we can prove that we know a secret without revealing any knowledge about the secret as well as verifying that a large and complex computation is done correctly with less efforts needed to perform the computation. For example, given the STF execution, we can prove the integrity of the transaction without disclosing the input data or the computation process. We also can do long complex computation off-chain and then verify it in on-chain using the succinct proof we have.

ZK Virtual Machines (VMs) and how it works

A ZK VM is a circuit that allows a prover to show that, given a set of inputs, as well as some bytecode, they have correctly executed the program code on said inputs. It works by receiving an inputs from the outside and also inputs from the program.

The program then put a program hash to the verifier. Meanwhile, the VM calculating the inputs from both program and the inputs and produce its own “inputs hash”, “Output hash”, and execution proof. Once it is produced, all the inputs output hash and execution proof are transferred to the verifier for the examination.

A. Example of projects building ZK VMs

There are couple of projects built using ZK VM:

1. StarkNet - a decentralized, permission-less and censorship-resistant STARK-powered L2 ZK-Rollup that supports general-computation over Ethereum. In the StarkNet VM, there is a persistent state that smart contracts can access and modify. These smart contracts can store variables, communicate with other contracts, and send/receive messages to/from L1

2. Miden - The project is led by Bobbin Threadbare, former Facebook's core ZK researcher who led the development of Winterfell. Miden VM supports arbitrary logic and transactions and has one important additional feature - for any program executed on the VM a STARK-based proof of execution is automatically generated. Programs in Miden VM are represented by Merklized Abstract Syntax Trees (MAST), which makes it possible to selectively reveal parts of a program while keeping the rest of the program private.

3. ZkSync - ZkSync is a Layer 2 scaling solution on Ethereum that offers low gas and fast transactions, without compromising on security. ZkSync has better security, cheaper and scalable in term of the other ZK VM.

B. Advantages and disadvantages the existing ZK VMs

The advantages of using the existing ZK VMs are:

1. Because of ease of development. It means that it is easier to write code in the the VM than wiring a circuit by hand;

2. Simpler proving and verification key management. Since a ZK VM is a single circuit that accepts arbitrary program bytecode, developers do not need to handle proving and verifying keys when writing application-specific circuits.

3. Compatibility with other VMs. A compromise can be reached, such that the ZK VM only implements a subset of the EVM, so it can fulfill most use cases.

There are some advantages as follows:

1. Even though the aim is for ease of development, learning a language that targets a ZK VM will be quite difficult.

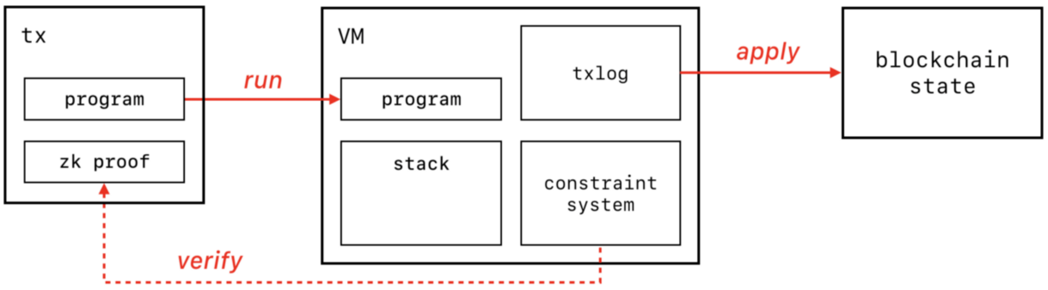
2. The security of programs written for a ZK VM will inherit any underlying security flaws of the ZK VM circuit. This creates a single point of failure, which may not be acceptable in some circumstances or for some use cases.

3. A compromise can be reached, such that the ZK VM only implements a subset of the EVM, so it can fulfill most use cases.

4. Vendor lock-in occurs when the developers of an application rely on tools, frameworks, or platforms which a third party controls so tightly that developers cannot easily switch

C. ZK VM architectures diagrams

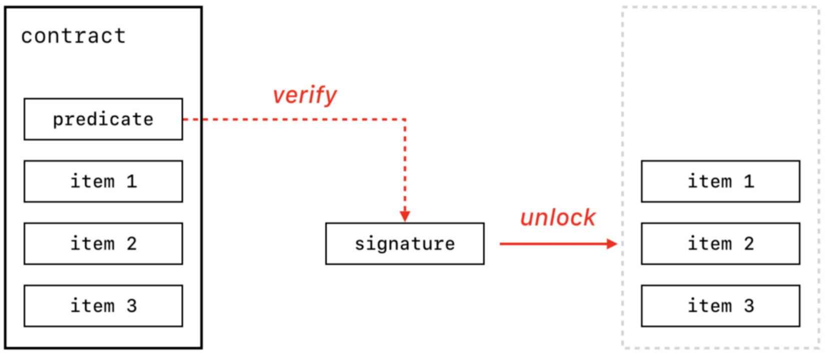
The following architectures is an ZK VM called ZkVM's stellar.



ZkVM uses a novel transaction format pioneered in TxVM: a transaction is represented as a *program* that manipulates flow of assets directly as first-class objects and emits the necessary updates to the blockchain state in the form of a *transaction log*. The transaction log then can be applied to the state of the system separately from transaction validation, which permits a highly scalable design, while offering a powerful, yet safe environment for custom contracts.

ZkVM uses a *utxo model* to represent balances: all balances are represented by a set of "unspent transaction outputs" (abbreviated as *utxos*) that can be created and destroyed. Each transaction proves ownership of the asset values in the previously created outputs, destroying those outputs, then issues, splits and merges the values, and creates new unspent outputs.

ZkVM uses a variant of Taproot design to allow unlocking the contracts with either an ordinary cryptographic *signature*, or by revealing and executing an embedded *sub-program* that verifies custom conditions.



Semaphore

About semaphore and application built using it

A. What is Semaphore

Semaphore is a system which allows any Ethereum user to signal their endorsement of an arbitrary string, revealing only that they have been previously approved to do so, and not their specific identity.

it lets a user:

- *Register* their identity in a smart contract, and then:
- *Broadcast* a signal — that is:
  (i) anonymously prove that their identity is in the set of registered identities, and at the same time:
  (ii) Publicly store an arbitrary string in the contract, if and only if that string is unique to the user and the contract's current *external nullifier* (more on this below). This means that double-signalling the same message under the same external nullifier is not possible.

B. How it works

Semaphore consists of a smart contract and zero-knowledge proof components which work in tandem. The smart contract handles state, permissions, and proof verification on-chain. The zero-knowledge components work off-chain to allow the user to generate proofs, which allow the smart contract to update its state if these proofs are valid.

In this case, we have to prove the following:

1. The user is authorized to do the signaling
2. The signal is unique (depending on the use case)

When a user registers their identity, they simply send a hash of an EdDSA public key and two random secrets to the contract, which stores it in a Merkle tree. This hash is called an *identity commitment*, and the random secrets are the *identity nullifier* and *identity trapdoor*.

In this case, Semaphore works by utilizing **smart contract** and **circom**. For smart contract, it does:

1. Register the identityCommitment by hashing identityTrapdoor and identityNullifier);

2. Verify Proof;

3. Store the nullifierHash (a hash on externalNullifier and identityNullifier) to prevent double-signalling.

For the circom part, it does:

1. Private inputs which consist of identityNullifier, identityTrapdoor, pathIndices, and treeSiblings;

2. Public inputs which is a merkleRoot itself.

In summary, there are two parts to broadcasting a signal: (a) anonymously proving membership of the set of registered users, and (b) preventing double-signalling via an external nullifier.

C. Applications that can be developed using Semaphore

There are at least 3 to 4 applications example that can be developed using Semaphore:

1. Authentication based application, where user can authenticate themselves without the need to provide the password to some applications instead using Semaphore as the verification layer;

2. Voting apps, where user can vote without revealing who they are through Semaphore system;

3. DAO membership, where user can be verified as DAO member without having to prove the identity and personal data that they are part of the DAO;

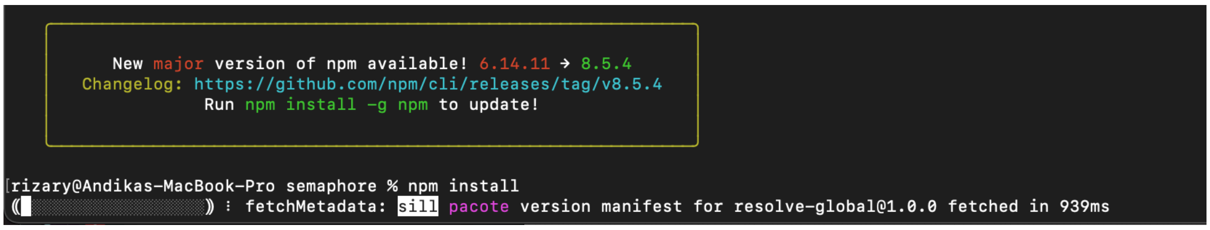
4. Private transaction like Mixers, Tornado Cash can be helped by Semaphore system when dealing with verification and proof.

Semaphore repository overview

Semaphore is an open source project located at: https://github.com/appliedzkp/semaphore. In this overview, I use the specific commit “3bce72febeba48454cb618a1f690045c04809900” as my base commit.

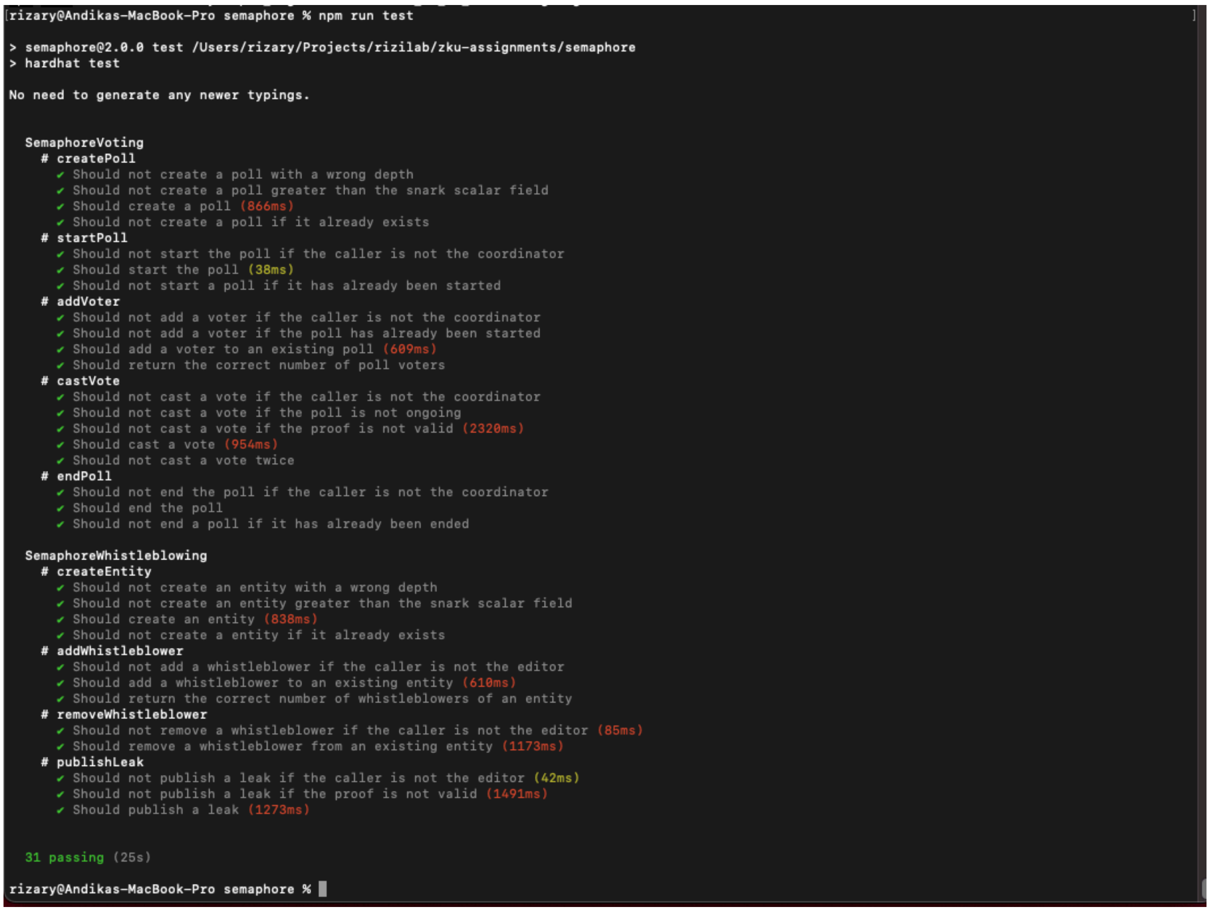
A. Running the test on the Semaphore project repository

Before running the test, make sure to run `npm install` on the root folder of Semaphore project and wait couple of minutes until it is finished installing the dependencies (please refer to the below screenshot).



```
   New major version of npm available! 6.14.11 → 8.5.4
Changelog: https://github.com/npm/cli/releases/tag/v8.5.4
             Run npm install -g npm to update!

[rizary@Andikas-MacBook-Pro semaphore % npm install
((█░░░░░░░░░░░░░░░░░)) : fetchMetadata: sill pacote version manifest for resolve-global@1.0.0 fetched in 939ms
```

Once it finishes installing, run `npm run test` to run the test.

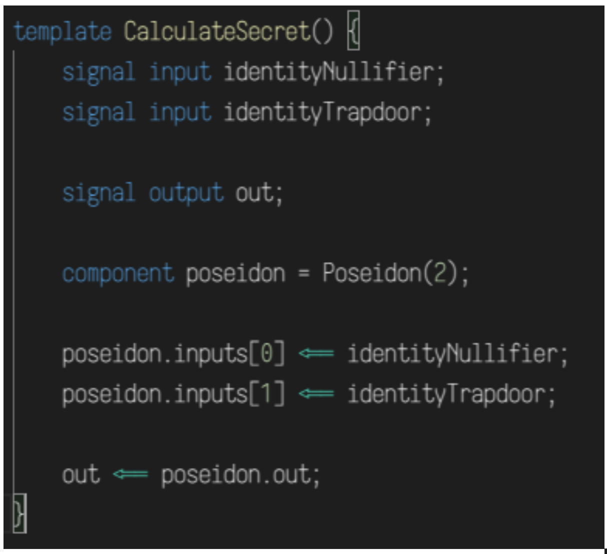


```
[rizary@Andikas-MacBook-Pro semaphore % npm run test

> semaphore@2.0.0 test /Users/rizary/Projects/rizilab/zku-assignments/semaphore
> hardhat test

No need to generate any newer typings.


  SemaphoreVoting
    # createPoll
      ✔ Should not create a poll with a wrong depth
      ✔ Should not create a poll greater than the snark scalar field
      ✔ Should create a poll (866ms)
      ✔ Should not create a poll if it already exists
    # startPoll
      ✔ Should not start the poll if the caller is not the coordinator
      ✔ Should start the poll (38ms)
      ✔ Should not start a poll if it has already been started
    # addVoter
      ✔ Should not add a voter if the caller is not the coordinator
      ✔ Should not add a voter if the poll has already been started
      ✔ Should add a voter to an existing poll (609ms)
      ✔ Should return the correct number of poll voters
    # castVote
      ✔ Should not cast a vote if the caller is not the coordinator
      ✔ Should not cast a vote if the poll is not ongoing
      ✔ Should not cast a vote if the proof is not valid (2320ms)
      ✔ Should cast a vote (954ms)
      ✔ Should not cast a vote twice
    # endPoll
      ✔ Should not end the poll if the caller is not the coordinator
      ✔ Should end the poll
      ✔ Should not end a poll if it has already been ended

  SemaphoreWhistleblowing
    # createEntity
      ✔ Should not create an entity with a wrong depth
      ✔ Should not create an entity greater than the snark scalar field
      ✔ Should create an entity (838ms)
      ✔ Should not create a entity if it already exists
    # addWhistleblower
      ✔ Should not add a whistleblower if the caller is not the editor
      ✔ Should add a whistleblower to an existing entity (610ms)
      ✔ Should return the correct number of whistleblowers of an entity
    # removeWhistleblower
      ✔ Should not remove a whistleblower if the caller is not the editor (85ms)
      ✔ Should remove a whistleblower from an existing entity (1173ms)
    # publishLeak
      ✔ Should not publish a leak if the caller is not the editor (42ms)
      ✔ Should not publish a leak if the proof is not valid (1491ms)
      ✔ Should publish a leak (1273ms)


  31 passing (25s)

rizary@Andikas-MacBook-Pro semaphore % █
```

If all the test is passed, then it means Semaphore is ready to be used by other application.

B. Code explanation on Semaphore's circom file

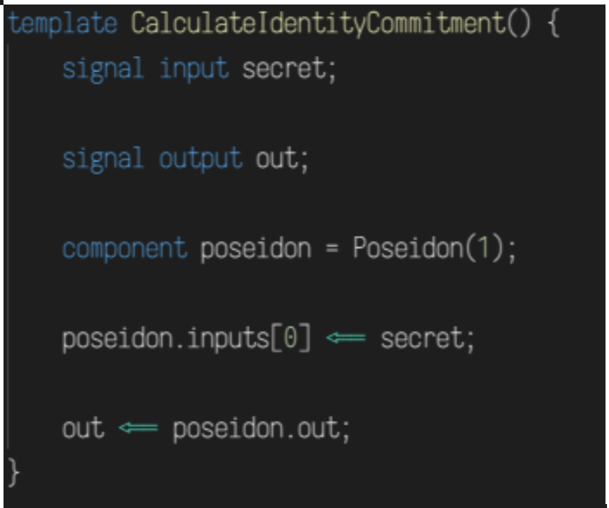
The circuit used by Semaphore located in `./circuit` folder. It contains 4 (four) templates, which are `CalculateSecret()`, `CalculateIdentityCommitment()`, `CalculateNullifierHash()`, and `Semaphore()`.



```
template CalculateSecret() {
    signal input identityNullifier;
    signal input identityTrapdoor;

    signal output out;

    component poseidon = Poseidon(2);

    poseidon.inputs[0] <== identityNullifier;
    poseidon.inputs[1] <== identityTrapdoor;

    out <== poseidon.out;
}
```

The first function is called \`CalculateSecret()\`, which used to create hash using identityNullifier and identityTrapdoor. In Semaphore, it uses **Poseidon** hash function, a hash function that is designed to minimize prover and verifier complexities when zero-knowledge proofs are generated and validated.

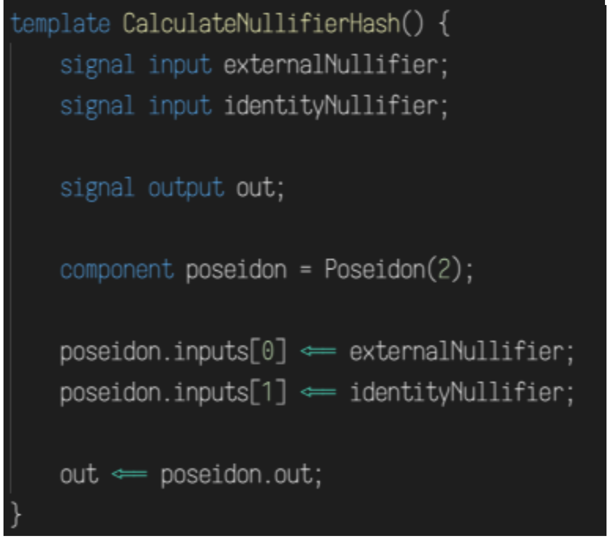
The inputs of this function is identityNullifier and identityTrapdoor.



```
template CalculateIdentityCommitment() {
    signal input secret;

    signal output out;

    component poseidon = Poseidon(1);

    poseidon.inputs[0] <== secret;

    out <== poseidon.out;
}
```

The second function is called \`CalculateIdentityCommitment()\`. This function input is a secret obtained from secret. It also uses **Poseidon** hash function to compute the secret and create a hash out of it.

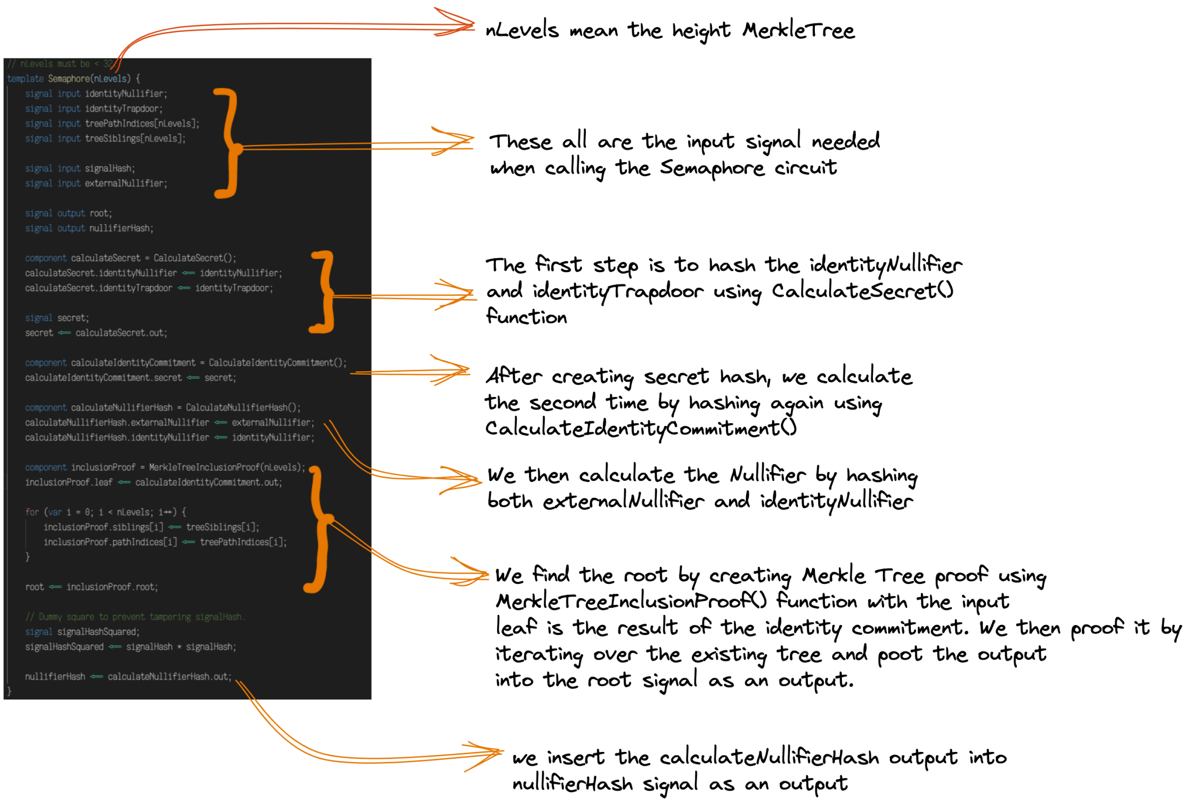
The third function is called \`CalculateNullifierHash()\`, which is a function that is used to create a hash from both identityNullifier and externalNullifier.



```
template CalculateNullifierHash() {
    signal input externalNullifier;
    signal input identityNullifier;

    signal output out;

    component poseidon = Poseidon(2);

    poseidon.inputs[0] <== externalNullifier;
    poseidon.inputs[1] <== identityNullifier;

    out <== poseidon.out;
}
```

All the first three functions are a utility functions that will help the \`Semaphore()\` function to build its system. It combined all of three function to create a zero-knowledge proof system. The step-by-step explanation on it is shown in the below picture.

Lastly, we have \`main\` component which specifies the nLevels to 20 and set the signalHash and externalNullifier to be public



```
// nLevels must be < 32.
template Semaphore(nLevels) {
    signal input identityNullifier;
    signal input identityTrapdoor;
    signal input treePathIndices[nLevels];
    signal input treeSiblings[nLevels];

    signal input signalHash;
    signal input externalNullifier;

    signal output root;
    signal output nullifierHash;

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

    signal secret;
    secret <== calculateSecret.out;

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

    component calculateNullifierHash = CalculateNullifierHash();
    calculateNullifierHash.externalNullifier <== externalNullifier;
    calculateNullifierHash.identityNullifier <== identityNullifier;

    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];
    }

    root <== inclusionProof.root;

    // Dummy square to prevent tampering signalHash.
    signal signalHashSquared;
    signalHashSquared <== signalHash * signalHash;

    nullifierHash <== calculateNullifierHash.out;
}
```

nLevels mean the height MerkleTree

These all are the input signal needed when calling the Semaphore circuit

The first step is to hash the identityNullifier and identityTrapdoor using CalculateSecret() function

After creating secret hash, we calculate the second time by hashing again using CalculateIdentityCommitment()

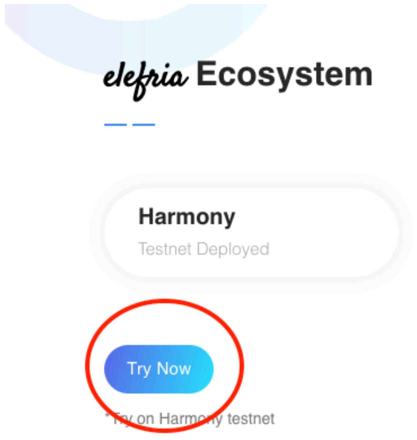
We then calculate the Nullifier by hashing both externalNullifier and identityNullifier

We find the root by creating Merkle Tree proof using MerkleTreeInclusionProof() function with the input leaf is the result of the identity commitment. We then proof it by iterating over the existing tree and poot the output into the root signal as an output.

we insert the calculateNullifierHash output into nullifierHash signal as an output

Generating ZK identity using Elefria on Harmony Testnet

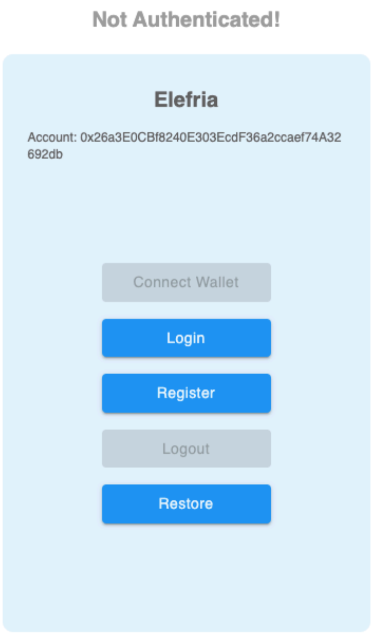
In order to use Elefria, go to elefria.com and click the `Try Now` button as shown below.



Once it is done, it will redirect the user to the login page which is showing that the user is not authenticated yet.

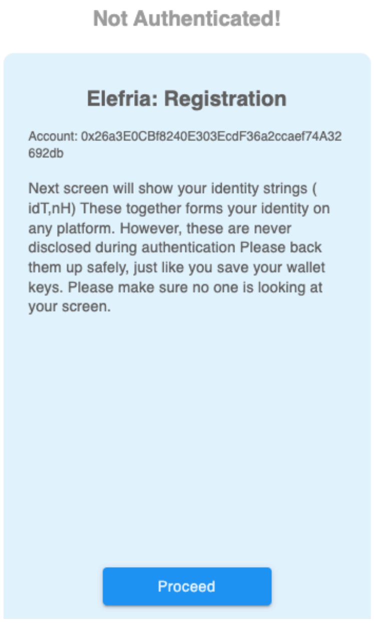
Connect the wallet using metamask and once you are connected (see below), click register and follow along with the registration process.

We then login by providing a password so that elefria can verify it by generating witness.


Not Authenticated!

Elefria

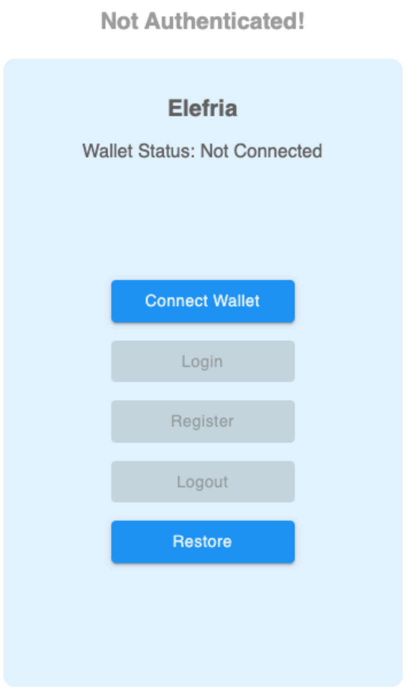
Account: 0x26a3E0CBf8240E303EcdF36a2ccaef74A32692db


Not Authenticated!

Elefria: Registration

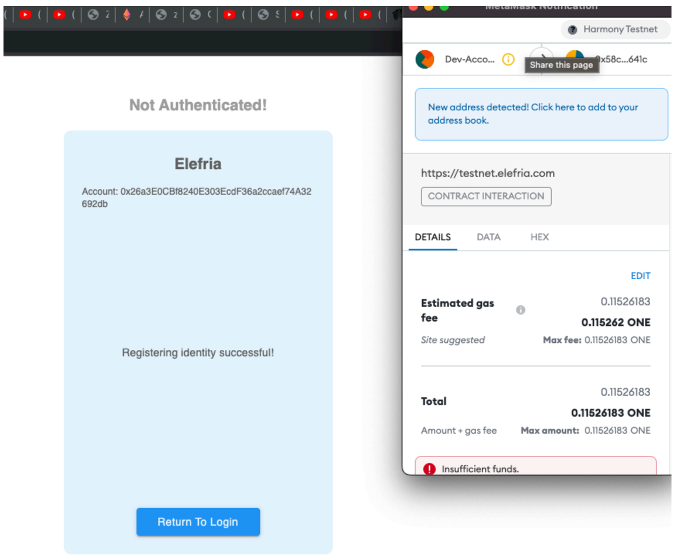
Account: 0x26a3E0CBf8240E303EcdF36a2ccaef74A32692db

Next screen will show your identity strings (idT,nH) These together forms your identity on any platform. However, these are never disclosed during authentication Please back them up safely, just like you save your wallet keys. Please make sure no one is looking at your screen.


Not Authenticated!

Elefria

Wallet Status: Not Connected


Not Authenticated!

Elefria

Account: 0x26a3E0CBf8240E303EcdF36a2ccaef74A32692db

Registering identity successful!

MetaMask Notification

Harmony Testnet

Dev-Acco... 0x58c...641c

New address detected! Click here to add to your address book.

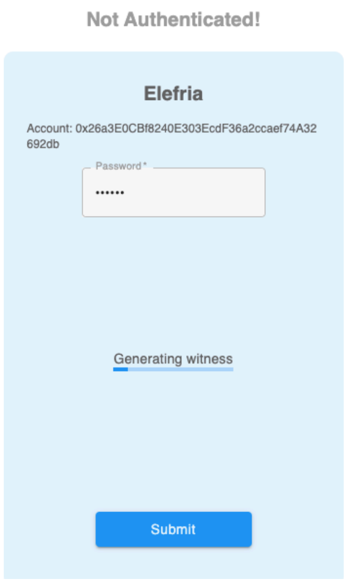
https://testnet.elefria.com

CONTRACT INTERACTION

Estimated gas fee 0.11526183
0.115262 ONE
Site suggested Max fee: 0.11526183 ONE

Total 0.11526183
0.11526183 ONE
Amount + gas fee Max amount: 0.11526183 ONE

Insufficient funds.


Not Authenticated!

Elefria

Account: 0x26a3E0CBf8240E303EcdF36a2ccaef74A32692db

Password * ••••••

Generating witness

A. Potential challenges to overcome

The biggest potential challenge when I try Eferia is about the synchronization between user wallet and the registration state in the UI. So, the case that I am facing is as follows:

1. Click “Try Now” to get directed to login page

2. Click “Connect Wallet” button and the Register

3. In the registration page, we put the password and click register

4. Make sure we don’t have enough balance in our wallet balance

5. Click reject in the wallet (in this case is Metamask)

6. Look into the Elefria, it shows “Registering identity successful!”

7. Go back login page.

Up to this step, the user shouldn’t be registered because I reject the transaction in Metamask. However, instead of informing the user that registration is unsuccessful, it says it is **successful** instead.

This caused synchronization problem, as when I return to login page and click login, I got stuck generating witness after inserting the correct password.

B. Potential improvements to simplify the Elefria authentication protocol

Some of potential improvements from my point of view to simplify this protocol is detecting whether the user is registered or not once connected to the wallet. If the user is not registered, then it gets redirected to registration page. If not, then it stays in the login page, and user can just login. Removing the “register” button and put the checking will simplify the protocol.

Another thing to simplify the protocol is make the generating witness to the background process and let the user know earlier if the user is incorrect or even the user is not registered yet (which is if the first paragraph is implemented, then this is not needed).

Tornado Cash

Comparison and key improvement between tornado-trees and tornado-nova

Looking at the repository between tornado-trees and tornado-nova, there are some addition file circuit in the tornado-nova which is `transaction.circom`. This file contains some update such as Instead of store 2 numbers in the commitment, tornado-nova now store three different number, first is amount, then public key (owner of this output) and then some number random. This random number is called blinding cryptography. Tornado-nova now has more input for nullifier like commitment, merklePath, and sign of privKey, commitment, merklePath. In this case, tornado-nova will allow users to deposit & withdraw arbitrary amounts of ETH and also a shielded transfers of deposited tokens while staying within the pool.

The MerkleTree template circuits is also changed and simpler than tornado-trees. In tornado-nova, the merkle tree created by computing hashes of the new tree layer and the input is an array of leaves instead of the single leaf and intermediary tree (look into tornado-trees' RewMerkleTree() circuit functions).

In tornado-nova, user can remain anonymous by choosing the common predetermined amounts of funds when they want to withdraw their funds. However, user can choose custom amount completely. In addition, tornado-nova will be using the Gnosis chain or formerly called xDAI chain as a Layer-2 so that all related transactions are cheaper and also faster (see the CrossChainGuard contracts).

Checking tornado-trees repository

A. Update the withdrawal tree process

There is a function call `updateWithdrawalTree()` in the `TornadoTrees.sol` solidity file of tornado-trees. This function takes 6 parameter:

1. _proof: A snark proof that elements were inserted correctly;
2. _argHash: A hash of snark inputs
3. _currentRoot: current merkle tree root
4. _newRoot: the root of merkle tree after update finished
5. _pathIndices: marble path to inserted batch
6. _events: a batch of inserted event

The first process of withdrawal is checking whether the current merkle tree root is the same as withdrawalRoot in the tornado trees main function. We also check if the merkle tree path is in the correct position. This function then store the _pathIndices, _newRoot, and _currentRoot to the memory.

After asserting the above variable, the function iterate the tree to update the leaf by doing the following step:

1. It gets the event's hash, instance and block;
2. Then it hash the above using keccak256 function
3. The function create withdrawal by checking if the sum of offset and the current iteration index is greater than or lower than length of the withdrawals
4. Lastly, if the sum of offset + iteration index is greater than withdrawals length, it will delete the offset + iteration index. If not, then it creates the withdrawal data

After updating the tree, the function called verifyProof function to verify between the _proof and the hashed data. The function to verify is obtained from `TreeUpdateArgsHasher()` in the circom circuit by batching it using the batch tree update function specified in the BatchTreeUpdate circuit function.

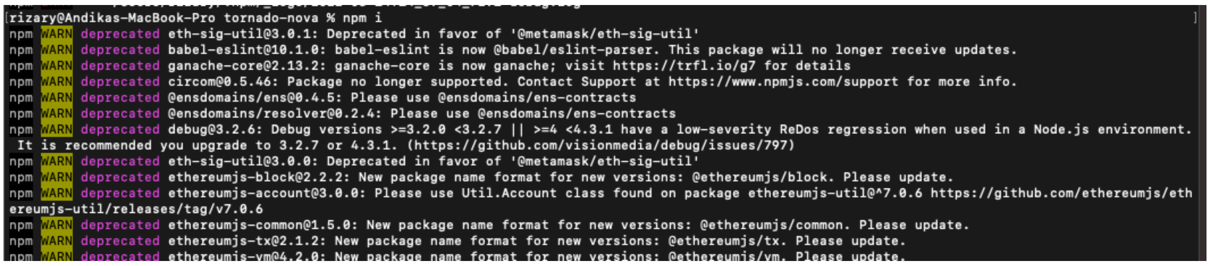
B. SHA256 vs Poseidon on withdrawal tree

Poseidon is a hash function designed to minimize prover and verifier complexities when zero-knowledge proofs are generated and validated. However, in the updateWithdrawalTree function, it uses sha256 instead of Poseidon to hash the data. The reason is because sha256 is faster than Poseidon, and thus updating the tree require faster evaluation since this function deals with the existing tree which is usually already have existing data.

Test tornado-nova repository

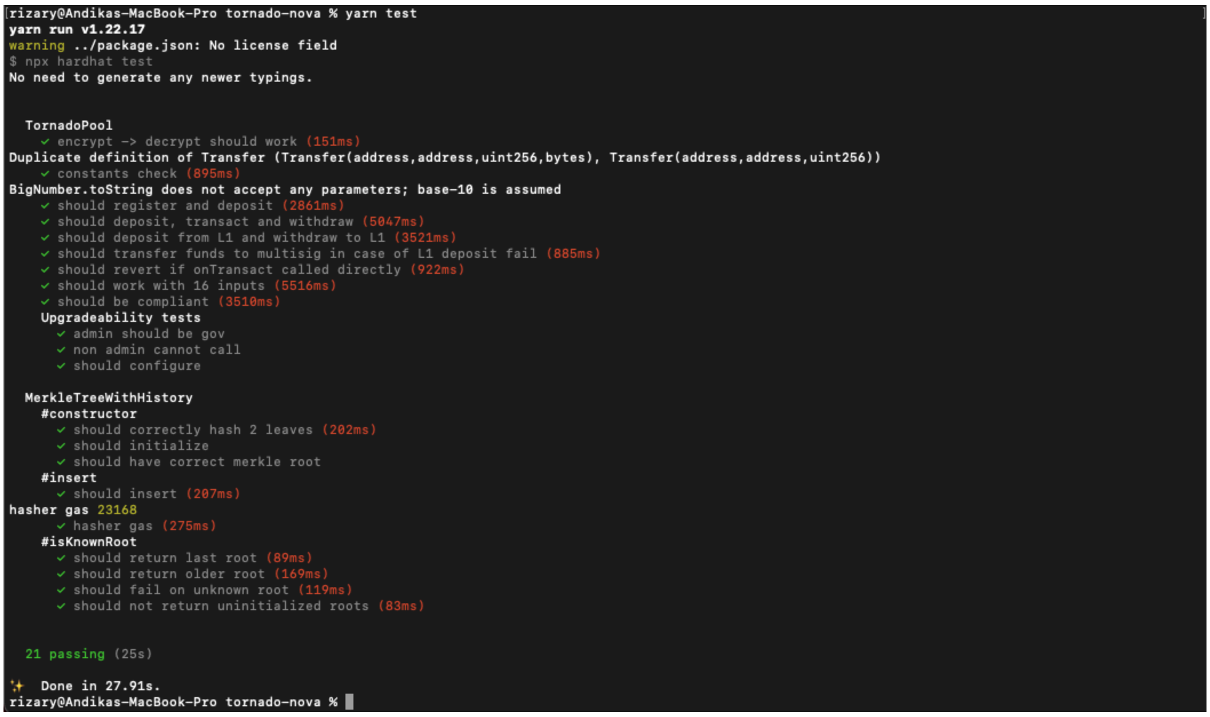
A. Running the test on tornado-nova repository

Before running the test, make sure to run `yarn install` on the root folder of tornado-nova project and wait couple of minutes until it is finished installing the dependencies (please refer to the below screenshot).



```
rizary@Andikas-MacBook-Pro tornado-nova % npm i
npm WARN deprecated eth-sig-util@3.0.1: Deprecated in favor of '@metamask/eth-sig-util'
npm WARN deprecated babel-eslint@10.1.0: babel-eslint is now @babel/eslint-parser. This package will no longer receive updates.
npm WARN deprecated ganache-core@2.13.2: ganache-core is now ganache; visit https://trfl.io/g7 for details
npm WARN deprecated circom@0.5.46: Package no longer supported. Contact Support at https://www.npmjs.com/support for more info.
npm WARN deprecated @ensdomains/ens@0.4.5: Please use @ensdomains/ens-contracts
npm WARN deprecated @ensdomains/resolver@0.2.4: Please use @ensdomains/ens-contracts
npm WARN deprecated debug@3.2.6: Debug versions >=3.2.0 <3.2.7 || >=4 <4.3.1 have a low-severity ReDos regression when used in a Node.js environment.
 It is recommended you upgrade to 3.2.7 or 4.3.1. (https://github.com/visionmedia/debug/issues/797)
npm WARN deprecated eth-sig-util@3.0.0: Deprecated in favor of '@metamask/eth-sig-util'
npm WARN deprecated ethereumjs-block@2.2.2: New package name format for new versions: @ethereumjs/block. Please update.
npm WARN deprecated ethereumjs-account@3.0.0: Please use Util.Account class found on package ethereumjs-util@^7.0.6 https://github.com/ethereumjs/eth
ereumjs-util/releases/tag/v7.0.6
npm WARN deprecated ethereumjs-common@1.5.0: New package name format for new versions: @ethereumjs/common. Please update.
npm WARN deprecated ethereumjs-tx@2.1.2: New package name format for new versions: @ethereumjs/tx. Please update.
npm WARN deprecated ethereumjs-vm@4.2.0: New package name format for new versions: @ethereumjs/vm. Please update.
```

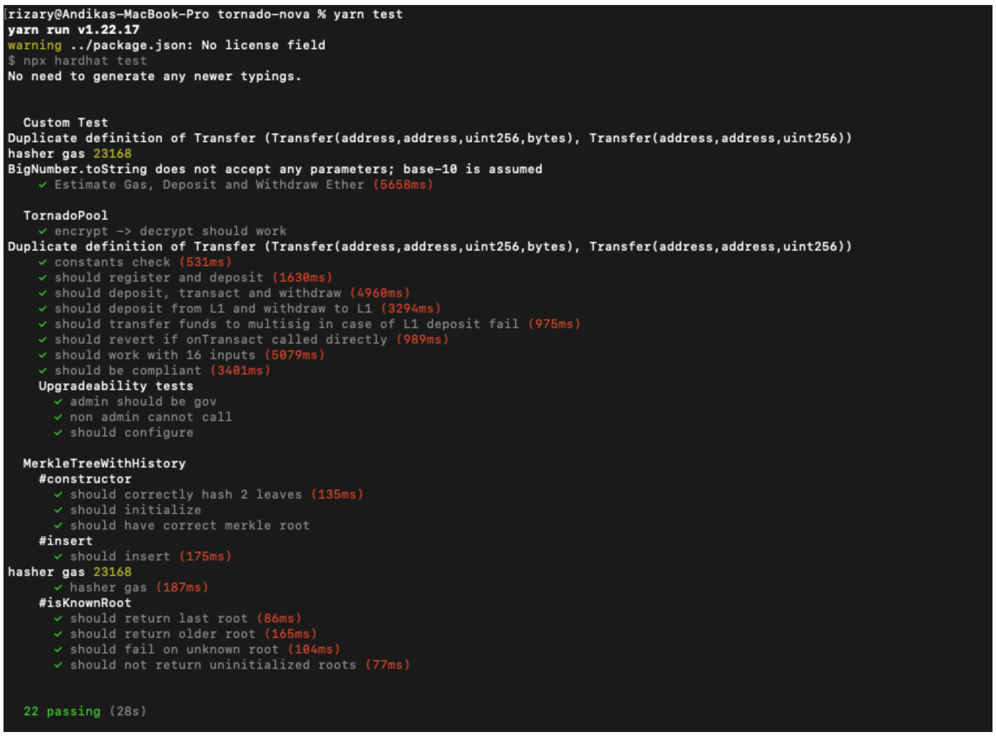
Once it finishes installing, run `yarn build && yarn test` to run the test.



```
rizary@Andikas-MacBook-Pro tornado-nova % yarn test
yarn run v1.22.17
warning ../package.json: No license field
$ npx hardhat test
No need to generate any newer typings.


  TornadoPool
    ✓ encrypt -> decrypt should work (151ms)
Duplicate definition of Transfer (Transfer(address,address,uint256,bytes), Transfer(address,address,uint256))
    ✓ constants check (895ms)
BigNumber.toString does not accept any parameters; base-10 is assumed
    ✓ should register and deposit (2861ms)
    ✓ should deposit, transact and withdraw (5047ms)
    ✓ should deposit from L1 and withdraw to L1 (3521ms)
    ✓ should transfer funds to multisig in case of L1 deposit fail (885ms)
    ✓ should revert if onTransact called directly (922ms)
    ✓ should work with 16 inputs (5516ms)
    ✓ should be compliant (3510ms)
    Upgradeability tests
      ✓ admin should be gov
      ✓ non admin cannot call
      ✓ should configure

  MerkleTreeWithHistory
    #constructor
      ✓ should correctly hash 2 leaves (202ms)
      ✓ should initialize
      ✓ should have correct merkle root
    #insert
      ✓ should insert (207ms)
hasher gas 23168
      ✓ hasher gas (275ms)
    #isKnownRoot
      ✓ should return last root (89ms)
      ✓ should return older root (169ms)
      ✓ should fail on unknown root (119ms)
      ✓ should not return uninitialized roots (83ms)


  21 passing (25s)

✨  Done in 27.91s.
rizary@Andikas-MacBook-Pro tornado-nova %
```

B. Run custom test

First, create a new file called `custom.test.js` in the test/ folder and write the code similar to https://github.com/Rizary/zero-knowlegde-university-report/blob/zku-assignment-2/custom.test.js and run the test again until all the test passed like the following picture:



```
rizary@Andikas-MacBook-Pro tornado-nova % yarn test
yarn run v1.22.17
warning ../package.json: No license field
$ npx hardhat test
No need to generate any newer typings.


  Custom Test
Duplicate definition of Transfer (Transfer(address,address,uint256,bytes), Transfer(address,address,uint256))
hasher gas 23168
BigNumber.toString does not accept any parameters; base-10 is assumed
    ✓ Estimate Gas, Deposit and Withdraw Ether (5658ms)

  TornadoPool
    ✓ encrypt -> decrypt should work
Duplicate definition of Transfer (Transfer(address,address,uint256,bytes), Transfer(address,address,uint256))
    ✓ constants check (531ms)
    ✓ should register and deposit (1630ms)
    ✓ should deposit, transact and withdraw (4960ms)
    ✓ should deposit from L1 and withdraw to L1 (3294ms)
    ✓ should transfer funds to multisig in case of L1 deposit fail (975ms)
    ✓ should revert if onTransact called directly (989ms)
    ✓ should work with 16 inputs (5079ms)
    ✓ should be compliant (3401ms)
    Upgradeability tests
      ✓ admin should be gov
      ✓ non admin cannot call
      ✓ should configure

  MerkleTreeWithHistory
    #constructor
      ✓ should correctly hash 2 leaves (135ms)
      ✓ should initialize
      ✓ should have correct merkle root
    #insert
      ✓ should insert (175ms)
hasher gas 23168
      ✓ hasher gas (187ms)
    #isKnownRoot
      ✓ should return last root (86ms)
      ✓ should return older root (165ms)
      ✓ should fail on unknown root (104ms)
      ✓ should not return uninitialized roots (77ms)


  22 passing (28s)
```

L1Unwrapper contract on Tornado.cash proposal

The purpose of the newly deployed L1Unwrapper contract is because there is some limitation with the Omnibridge that need to be fixed. Another reason is because xDai team that cannot subsidize the withdrawals any longer.

The main goal is to allow Tornado Cash to continue running without xDai intervention. Another reason is so that users can pay for the L1 fees themselves.

Thinking in ZK

The thought on questions for Tornado Cash and Semaphore founders

A. Tornado Cash

1. How do we maintain the correctness of transaction inputs without verifying it in the transaction.circom?

2. Do we really need zero-knowledge for just small transaction? Is it possible to avoid it just for small transaction?

B. Semaphore

1. What is the future on Semaphore implementation in real-life?

2. Is there any possibility to integrate Semaphore into Decentralized Identity (DID)

The thought on using just one circuit for all dApps and the future of developing Zk apps

A. One circuit for all dApps

It is still possible to use one circuit for all dApps because some of the Zk implementation using almost identical circuit. However, there are some variation of circuit needed by the developer dApps for their specific use case.

Making one circuit for all dApps is also hard to maintain, because people can easily fork and use that technology. If that happened, then there is no point to have single circuit for all dApps.

B. The future of developing Zk dApps

It is likely to be some standard on developing Zk dApps in the future. Several thing that comes to my attention is:

1. Having a template for Zk dApps skeleton or certain feature.
2. Make some Zk dApps coding standard and guideline