ZKU Assignment 2

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

Question:1 Privacy and ZK VMs:

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?

As explained in the zCloak Network overview, present blockchains grab inputs in the form of a transaction and depending on some predefined rules, change the general state. This process is replicated across all distributed nodes by re-execution of the State Transition Function (STF).
We get: `STF(current_state, input_data) = new_state`

With verification over re-execution, you get the following advantages:

- **Privacy:** Users might not want to expose their input data y as part of the STF(x, y) computation. Some zkVM allow for this enhanced user privacy. In Polygon Hermez, the state as well as user inputs are public when performing zkRollups, therefore privacy is optional in this case. Someone creating a zkVM can choose to make these input data private or not.
  Note: Polygon Hermez does not have a zkEVM in mainnet, it only has native support like payments and such.
- **Scalability:** Re-execution is computationally quite expensive for the nodes across a network. Verification would be extremely cheaper in terms of computation usage.
- **Computation restriction:** One could compute something extremely costly off-chain and submit on-chain the "proof of computation", i.e. a ZKP proving that the calculation is correct and genuine.
- **Storage:** If a node could boot up using the latest state of the network and a proof that this state results from the full history of the blockchain, then the community could circumvent the risk of excluding smaller nodes from the network.

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

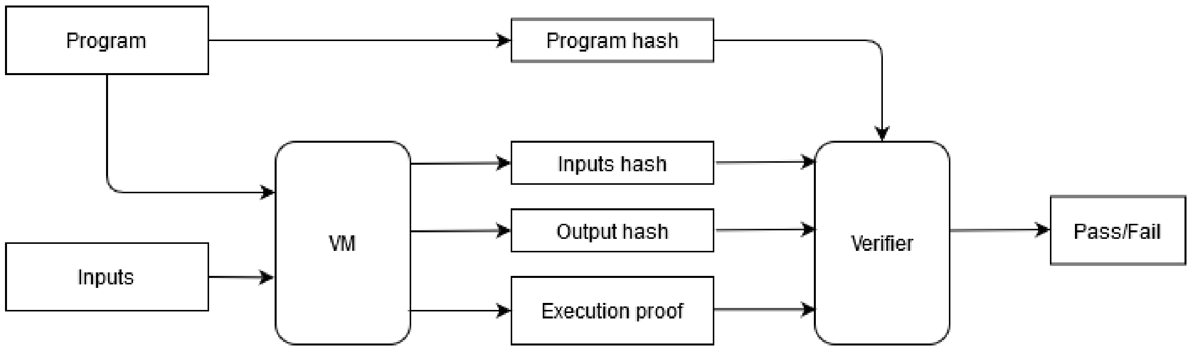
1. Give examples of certain projects building Zk VMs (at-least 2-3 projects). Describe in brief, key differences in their VMs.

Defn: “A ZK VM is a circuit (a representation of a program) that executes bytecode. It 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.” ……[Reference]

Notably, the bytecode (or program) is not the circuit itself, but part of its inputs. As such, a prover can use the same circuit to create proofs of validity of execution for arbitrary programs, as long as said programs fit the underlying ZK VM circuit.

zkVM	Architecture	Language	ZKP Type	Projects
Miden VM	Upgrade of Distaff VM	Rust + assembly-like	zk-Stark	Polygon
Snark VM	EVM	Leo	zk-Snark	Aleo
Distaff VM	Distaff VM	Rust + assembly-like	zk-Stark	zCloak
Cairo VM	Cairo VM	Cairo (assembly-like)	zk-Stark	Starkware

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



In the above:

1. The VM does the following:
   a. Takes 2 inputs: a program and a set of inputs for the program,
   b. Executes the program with the given set of inputs,
   c. Outputs hash of the inputs, hash of the outputs generated by the program, and a STARK proof attesting to the correct execution of the program.
2. The verifier does the following:
   a. Takes the hash of the program, hash of the inputs, and hash of the outputs, and uses them to verify the STARK proof.

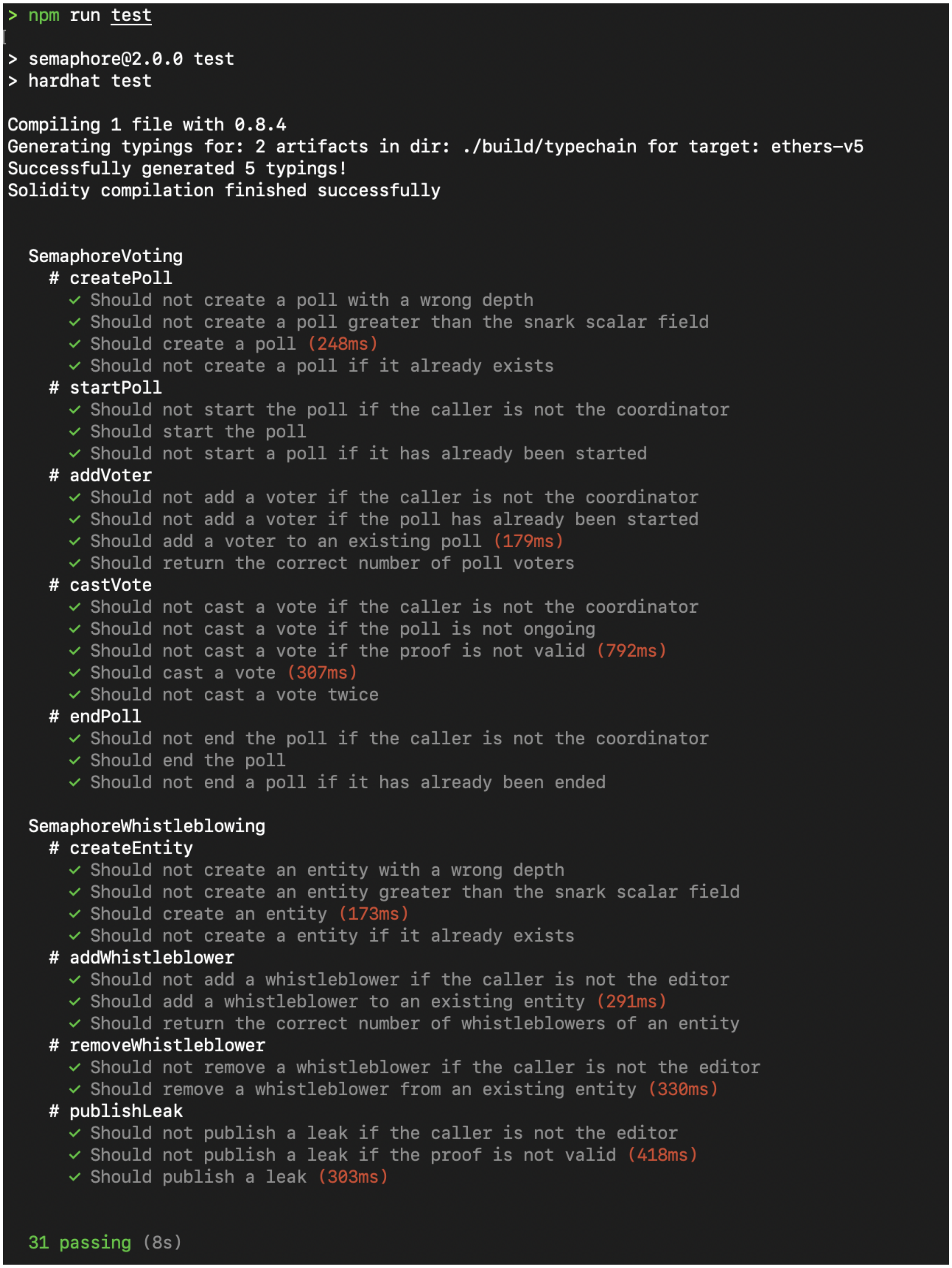
Question 2. Semaphore

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

Semaphore is a Zero-Knowledge tool that allows Ethereum users to prove their membership of a set without revealing their original identity. At the same time, it allows users to signal their endorsement of an arbitrary string and provides a simple built-in mechanism to prevent double-signaling or double-spending. It is designed to be a simple and generic privacy layer for Ethereum DApps.
Use cases include private voting, whistleblowing, mixers (Tornado Cash) and anonymous authentication.

2. Clone the semaphore repo ([3bce72f](3bce72f)).

1. Run the tests and add a screenshot of all the test passing.



```
> npm run test

> semaphore@2.0.0 test
> hardhat test

Compiling 1 file with 0.8.4
Generating typings for: 2 artifacts in dir: ./build/typechain for target: ethers-v5
Successfully generated 5 typings!
Solidity compilation finished successfully


  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 (248ms)
      ✓ 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
      ✓ 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 (179ms)
      ✓ 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 (792ms)
      ✓ Should cast a vote (307ms)
      ✓ 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 (173ms)
      ✓ 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 (291ms)
      ✓ Should return the correct number of whistleblowers of an entity
    # removeWhistleblower
      ✓ Should not remove a whistleblower if the caller is not the editor
      ✓ Should remove a whistleblower from an existing entity (330ms)
    # publishLeak
      ✓ Should not publish a leak if the caller is not the editor
      ✓ Should not publish a leak if the proof is not valid (418ms)
      ✓ Should publish a leak (303ms)


  31 passing (8s)
```

2. Explain code in the semaphore.circom file (including public, private inputs).

- CalculateSecret() : this circuit uses the poseidon hashing algorithm twice to take someone's private information and obfuscate it. More detailed information is available [here].

- CalculateIdentityCommitment() : The circuit hashes the hash of the identity nullifier and the identity trapdoor to generate an identity commitment. It then verifies the Merkle proof against the Merkle root and the identity commitment.

- CalculateNullifierHash() : The circuit hashes the identity nullifier and the external nullifier, then it checks that it matches the given nullifiers hash. Additionally, the smart contract ensures that it has not previously seen this nullifiers hash. This way, double-signaling is impossible.

- Semaphore() : 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 (MT Verifier). This hash is called an identity commitment, and the random secrets are the identity nullifier and identity trapdoor. Broadcasting a signal is a little more complex.

- Inputs and outputs :
    - public input: signalHash, externalNullifier, private inputs: identityNullifier, identityTrapdoor, treeSiblings, treePathIndices.
    - `poseidon(identityNullifier,identityTrapdoor) => secret`
    - `poseidon(secret) => inclusionProof.leaf`
    - `poseidon(externalNullifier, identityNullifier) => nullifierHash`
    - `MerkleTreeInclusionProof(inclusionProof.leaf, treeSiblings, treePathIndices) => root`
    - outputs: `root` and `nullifierHash`

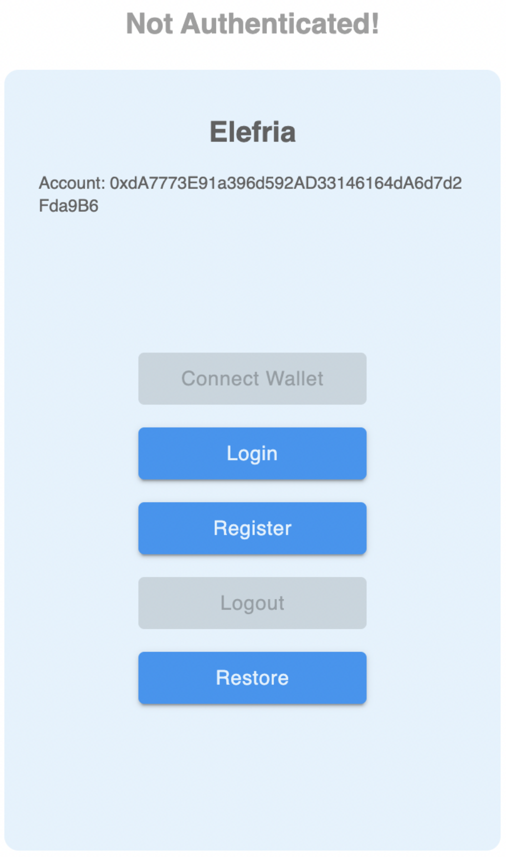
- There are two parts to it:
    - (a) anonymously proving membership of the set of registered users, and
    - (b) preventing double-signaling via an external nullifier.

  This is not done in the circuit, though.

3. Use Elefria protocol on the Harmony Testnet, try to generate a ZK identity and authenticate yourself as a user.

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

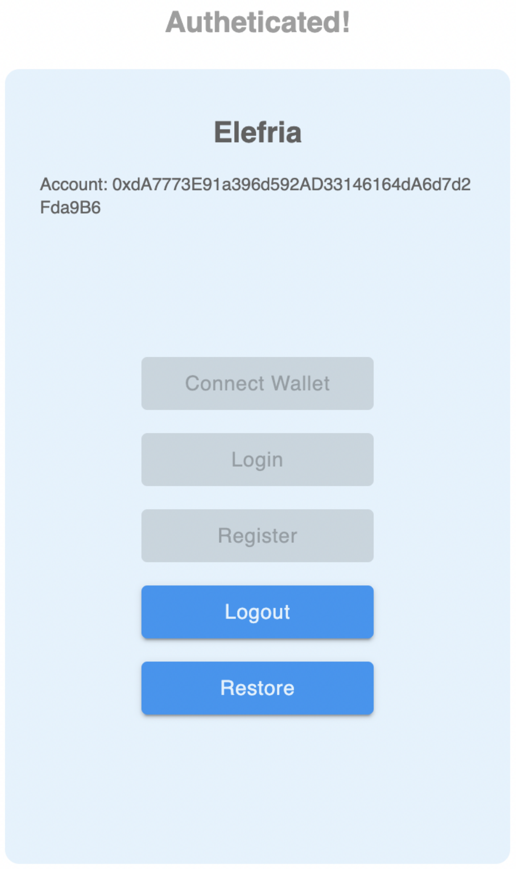
The elefria auth system works like this: you generate a zk identity, and submit the identity to the contract, which verifies the identity, and generates a token. You then can use the token on any other platforms you are trying to access rather than using your personal address, as long as the platforms enable validation through the elefria smart contracts. This will be their main challenge, as zk systems are hard, and most web3 devs do not know how to audit such a system. In the case of private voting, it will be hard to verify that someone is a unique voter, since we still would need a more fundamental `proof of unique-person` protocol for this to work.



Not Authenticated!

Elefria

Account: 0xdA7773E91a396d592AD33146164dA6d7d2 Fda9B6

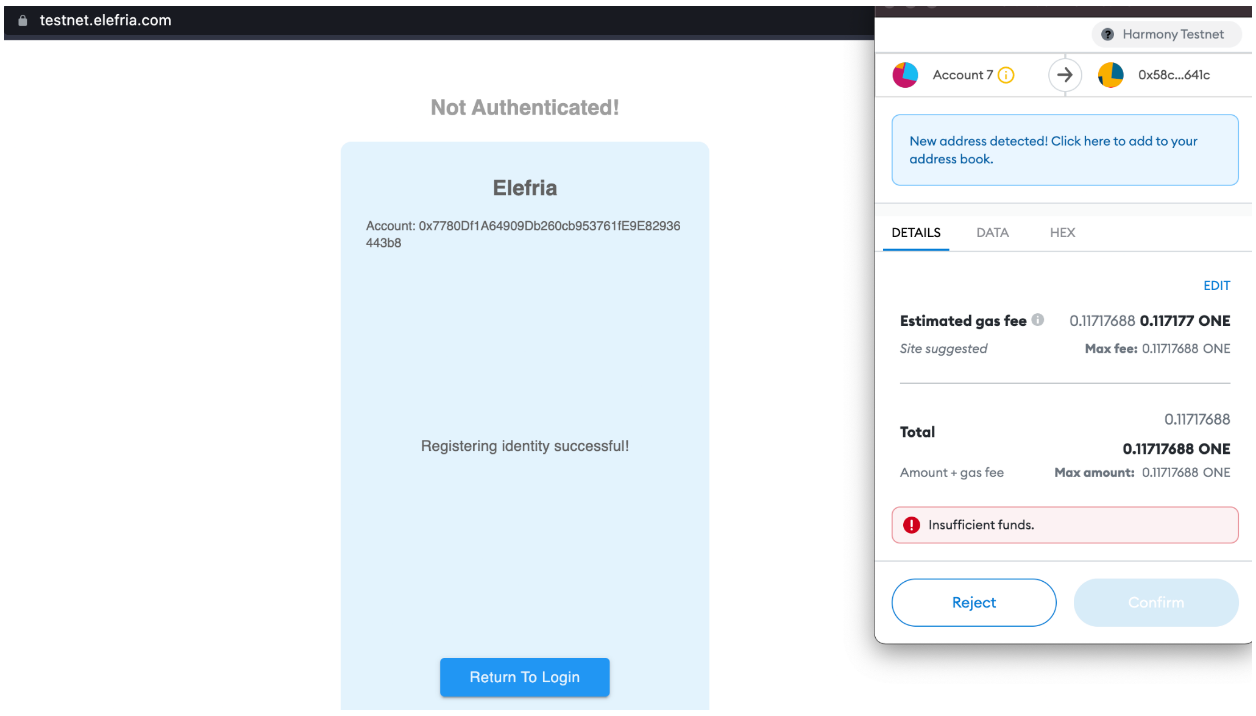


Autheticated!

Elefria

Account: 0xdA7773E91a396d592AD33146164dA6d7d2 Fda9B6

Note (Possible Bug): Initially I had no funds to confirm the transaction for registering but on rejecting the transaction it still showed on the UI that Registering for Identity is successful.


testnet.elefria.com

Not Authenticated!

Elefria

Account: 0x7780Df1A64909Db260cb953761fE9E82936443b8

Registering identity successful!

Return To Login

Harmony Testnet

Account 7 → 0x58c...641c

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

DETAILS DATA HEX

EDIT

Estimated gas fee 0.11717688 **0.117177 ONE**
Site suggested **Max fee:** 0.11717688 ONE

Total 0.11717688 **0.11717688 ONE**
Amount + gas fee **Max amount:** 0.11717688 ONE

Insufficient funds.

Reject Confirm

Question 3. Tornado Cash

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.

It appears former [Tornado pools](#) allowed a per poor fixed amount of deposit/withdrawal. As of Tornado Cash [Nova](#), users can deposit and withdraw arbitrary amounts inside the tornado cash pools. These pools are in beta and max amount of deposit is 1 ETH. Users can now do shielded transfers within the pool without having to withdraw and redeposit. Thus, they can do intrapool transactions.

Tornado Cash Nova chose Gnosis Chain (former xDai) as L2 for their low gas fees, fast withdrawal time (minutes). They utilize under the hood the ETH <> WETH L2 Gnosis Bridge. Users might therefore consider the risk of using Tornado Cash to be slightly increased: from only zk-proof technology risk (greenness of the tech) to now bridge technology risk on top of it. Nonetheless, this is an acceptable risk as this bridge is part of the major bridges and has been audited.

2. Check out the tornado-trees repo

1. **The circuit [circuits/TreeUpdateArgsHasher.circom]() and the smart contract: [contracts/TornadoTrees.sol]()**

- ***Clarifications to help understand the underlying structure:***
  - Elements to be added to merkle tree: information relevant to the users' action (deposit, withdrawal, transaction)
  - Tornado Cash's priorities are privacy and gas optimization. They will use many smart and crafty operations to reduce gas usage.
  - Although elements that are added to the merkle tree are not included in the EVM side as storage, they are included as calldata (16 gas vs. 625 gas per byte of data).

Tornado Cash uses zkSnarks for its compression capabilities and privacy enabling. Tornado Cash uses an updatable merkle tree to upload an array of authorized users to the blockchain. These users are stored in a merkle root and will then be able to withdraw some amount from a pool. The zkSnarks are used to do the computation off-chain in order to save gas, and prove onchain the correctness of the elements added.

The circom circuit will pack all the elements and hash them using sha256.

- ***How does it work?***
  - User constructs the proof (.circom related files) of their elements being added to the Tornado pool (merkle tree).
  - Call is made to the smart contract (.sol, EVM related process) where the proof is submitted on-chain, the elements are submitted as call-data and packed as bytes and then rechecked. Then, checks are made to see that everything matches and correctness of the submitted elements.
  - User is successfully added to the withdrawal tree.

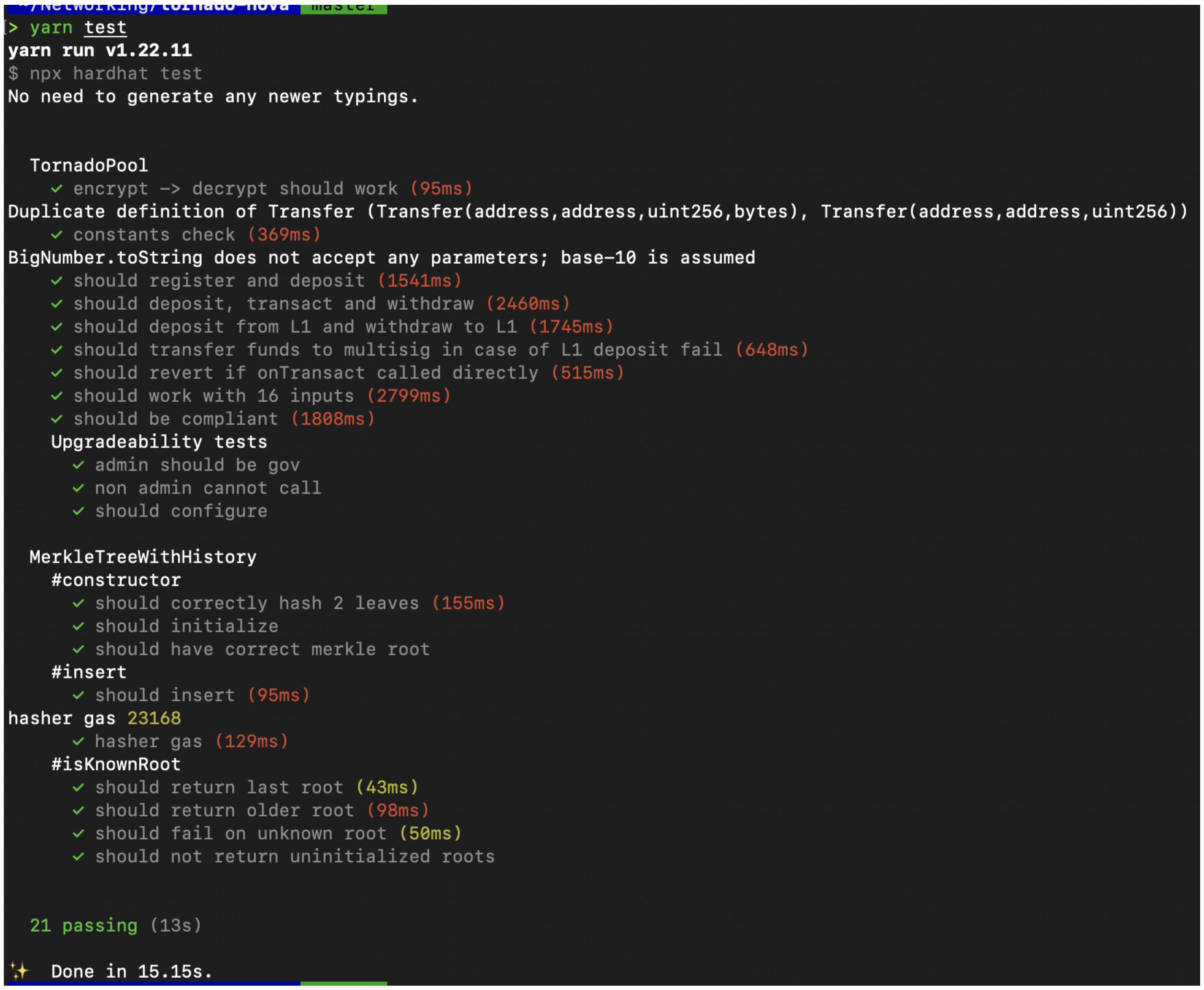
All the SNARK public inputs are hashed together to be submitted as only one public input to the verifier. This is aimed to minimize elliptic curve computations, thus saving a lot of gas. The tradeoff is to have more computation on the SNARK side.

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

   SHA256 is relatively light gas-wise on the EVM, but on-chain Poseidon is very expensive. On the contrary, SHA256 is very expensive on circom and Poseidon isn't. Since Tornado wanted to save money through gas optimization, they chose to put the computation pressure on the side of the SNARK infrastructure.

3. Clone/fork the tornado-nova repo:

1. Run the tests and add a screenshot of all the tests passing:

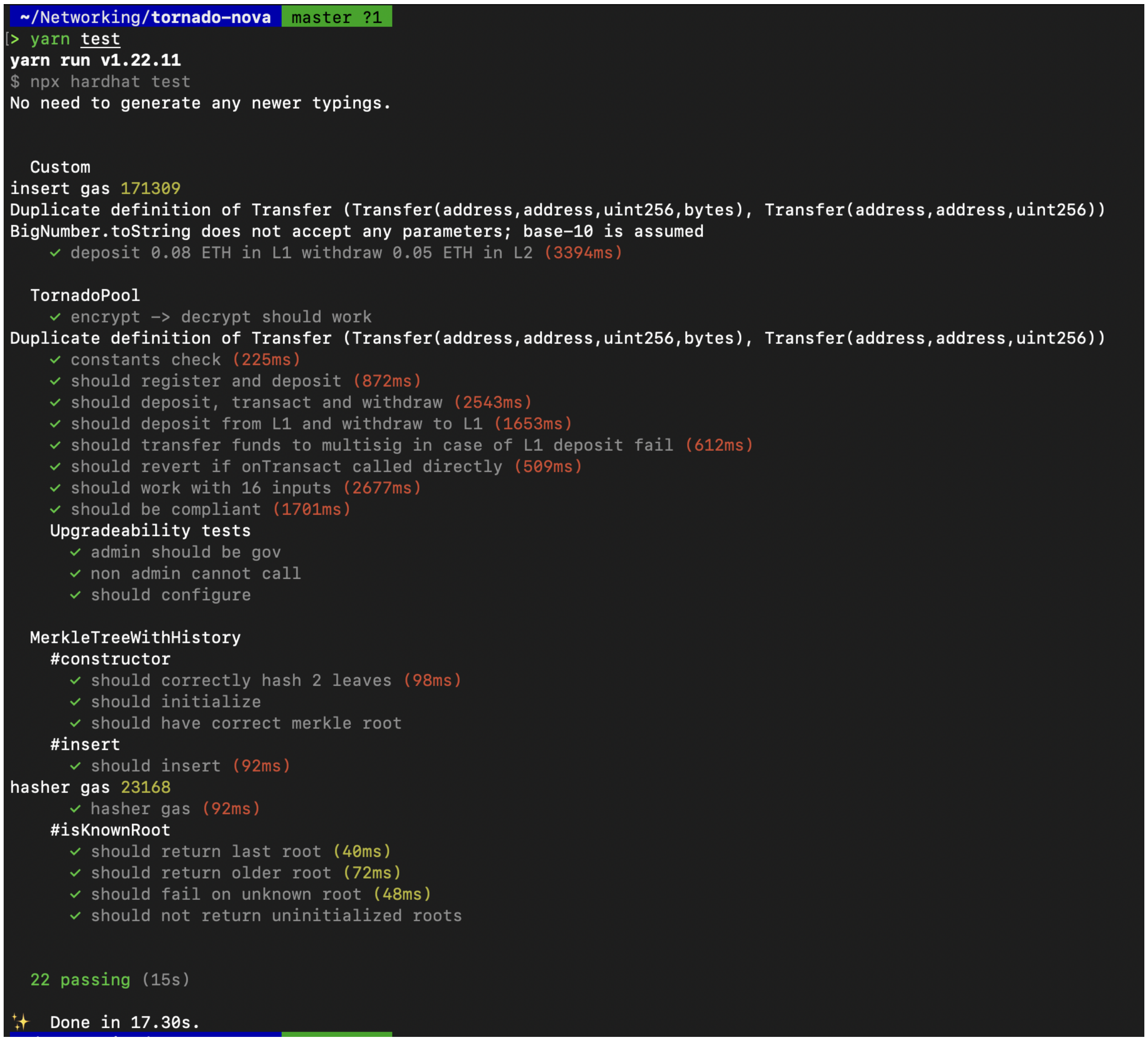


```
~/Networking/tornado-nova  master
> yarn test
yarn run v1.22.11
$ npx hardhat test
No need to generate any newer typings.


  TornadoPool
    ✓ encrypt -> decrypt should work (95ms)
Duplicate definition of Transfer (Transfer(address,address,uint256,bytes), Transfer(address,address,uint256))
    ✓ constants check (369ms)
BigNumber.toString does not accept any parameters; base-10 is assumed
    ✓ should register and deposit (1541ms)
    ✓ should deposit, transact and withdraw (2460ms)
    ✓ should deposit from L1 and withdraw to L1 (1745ms)
    ✓ should transfer funds to multisig in case of L1 deposit fail (648ms)
    ✓ should revert if onTransact called directly (515ms)
    ✓ should work with 16 inputs (2799ms)
    ✓ should be compliant (1808ms)
    Upgradeability tests
      ✓ admin should be gov
      ✓ non admin cannot call
      ✓ should configure

  MerkleTreeWithHistory
    #constructor
      ✓ should correctly hash 2 leaves (155ms)
      ✓ should initialize
      ✓ should have correct merkle root
    #insert
      ✓ should insert (95ms)
hasher gas 23168
      ✓ hasher gas (129ms)
    #isKnownRoot
      ✓ should return last root (43ms)
      ✓ should return older root (98ms)
      ✓ should fail on unknown root (50ms)
      ✓ should not return uninitialized roots


  21 passing (13s)

✨  Done in 15.15s.
```

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

Link to the GitHub code: [custom.test.js]

- estimate and print gas needed to insert a pair of leaves to `MerkleTreeWithHistory`
- deposit 0.08 ETH in **L1**
- withdraw 0.05 ETH in **L2**
- assert recipient, omniBridge, and tornadoPool balances are correct



```
 ~/Networking/tornado-nova  master ?1
[> yarn test
yarn run v1.22.11
$ npx hardhat test
No need to generate any newer typings.


  Custom
insert gas 171309
Duplicate definition of Transfer (Transfer(address,address,uint256,bytes), Transfer(address,address,uint256))
BigNumber.toString does not accept any parameters; base-10 is assumed
    ✓ deposit 0.08 ETH in L1 withdraw 0.05 ETH in L2 (3394ms)

  TornadoPool
    ✓ encrypt -> decrypt should work
Duplicate definition of Transfer (Transfer(address,address,uint256,bytes), Transfer(address,address,uint256))
    ✓ constants check (225ms)
    ✓ should register and deposit (872ms)
    ✓ should deposit, transact and withdraw (2543ms)
    ✓ should deposit from L1 and withdraw to L1 (1653ms)
    ✓ should transfer funds to multisig in case of L1 deposit fail (612ms)
    ✓ should revert if onTransact called directly (509ms)
    ✓ should work with 16 inputs (2677ms)
    ✓ should be compliant (1701ms)
    Upgradeability tests
      ✓ admin should be gov
      ✓ non admin cannot call
      ✓ should configure

  MerkleTreeWithHistory
    #constructor
      ✓ should correctly hash 2 leaves (98ms)
      ✓ should initialize
      ✓ should have correct merkle root
    #insert
      ✓ should insert (92ms)
hasher gas 23168
      ✓ hasher gas (92ms)
    #isKnownRoot
      ✓ should return last root (40ms)
      ✓ should return older root (72ms)
      ✓ should fail on unknown root (48ms)
      ✓ should not return uninitialized roots


  22 passing (15s)

✨  Done in 17.30s.
```

Question 4. Thinking In ZK

Questions for tornado Cash:

- Currently, Tornado Cash focuses on shielding financial transactions, any plans for increasing the use-cases and scope for the application?