e-mail : dalgarim@protonmail.com
discord : 구자현#7593
github : https://github.com/dalgarim/zkuAssignment2

Question 1: Privacy & ZK VMs

Q1. Explain in brief, how does the existing blockchain state transition to a new state? What is the advantage of using verification over re-execution?

A1.

- State transition functions can be expressed as follows,

  APPLY(S,TX) -> S'
  S := current state (before transaction)
  TX := signed data package that stores a message to be sent from an externally owned account
  S':= state after transaction

- State transition process is generally as follows,
  1. Check the transaction(TX) is valid and well-formed.
  2. Validate the sender has enough amount of transaction cost(GAS) using current state(S)
  3. Validate the sender has enough amount to transfer value to the receiver.
  4. Transfer value and execute smart contract code.
  5. If the value transfer failed (sender did not have enough money, ran out of gas), revert all state changes.
  6. Return new state(S)
- If a node has to re-execute all state changes(TX) to check if the state is valid or not, it requires excessive computing resources to execute all transactions from genesis state(S^0) to the current state which demeans the availability of the system.
- By verifying the state of block(S) before transaction, nodes can validate the state of account in an efficient time and computing resources.

Q2. Explain in brief what is a ZK VM (virtual machine) and how it works?

A2.

A Virtual Machine is an encapsulated environment that provides a separate execution layer. ZK VM is a virtual machine that adopts zero-knowledge proof such that it can verify the execution of bytecode(smart contract) and its inputs and outputs. ZK VM enables one to use a single circuit or AIR(algebraic intermediate representations) to verify execution which makes ease of development and key management.

A user provides bytecode and inputs to the VM and hash of bytecode to the verifier then the VM executes the byte code using the inputs. After execution, VM sends the execution proof and input hash, output hash to the verifier. The verifier validate given parameters and determines whether the execution is passed or not.

Q2.1 Give examples of certain projects building Zk VMs (at-least 2-3 projects). Describe in brief, key differences in their VMs.

A2.1

- Cairo : A general-purpose ZK-STARK virtual machine developed by StarkWare. It provides Cairo programming language both for its infrastructure and for writing contracts.
- ZENROOM : Attribute based credentials (ABC) and non-interactive zero knowledge proof (zk-SNARKS) based VM written in C. Using domain specific language called zencode, users are able to execute cryptographic operations and smart contracts in a multiplatform environment.
- Distaff : A general-purpose ZK-STARK virtual machine written in rust. This VM resides in the zCloak network and is compatible with the zColoak’s extensions and components.

Question 2. Semaphore

Q1. What is Semaphore? Explain in brief how it works? What applications can be developed using Semaphore (mention 3-4)?

A1.

Semaphore is a zero-knowledge instrument that enables Ethereum users to prove their inclusion in a group that they previously joined without sharing their personal identity It is intended to be a simplistic and generic privacy layer for Ethereum dApps. Using the Semaphore, user can register their unique identity and broadcast a signal to prove their registration or store a string to smart contract if their external nullifier is valid.

Users can insert their identity(EdDSA private key, random 32 bytes nullifier and trapdoor). The identity commitment is a hash of corresponding public key and nullifier and trapdoor. User easily register the identity commitment using the 'insertIdentity(uint256 _identityCommitment)' function.

To broadcast a signal, the user must invoke this Semaphore contract function:

```
broadcastSignal(
//the signal to broadcast.
    bytes memory _signal,

//a zk-SNARK proof
    uint256[8] memory _proof,

//The root of the identity tree, where the user's identity commitment is the last-inserted
leaf.
    uint256 _root,

//A uniquely derived hash of the external nullifier, user's identity nullifier, and the
Merkle path index to their identity commitment.
    uint256 _nullifiersHash,

//The external nullifier at which the signal is broadcast.
    uint232 _externalNullifier
)
```

broadcastSignal function

Common use case of Semaphore is

1. Mixer : transports ETH or ERC20 tokens from one address to another in such a way that only the sender knows the addresses are related
2. Anonymous authentication
3. Anonymous voting

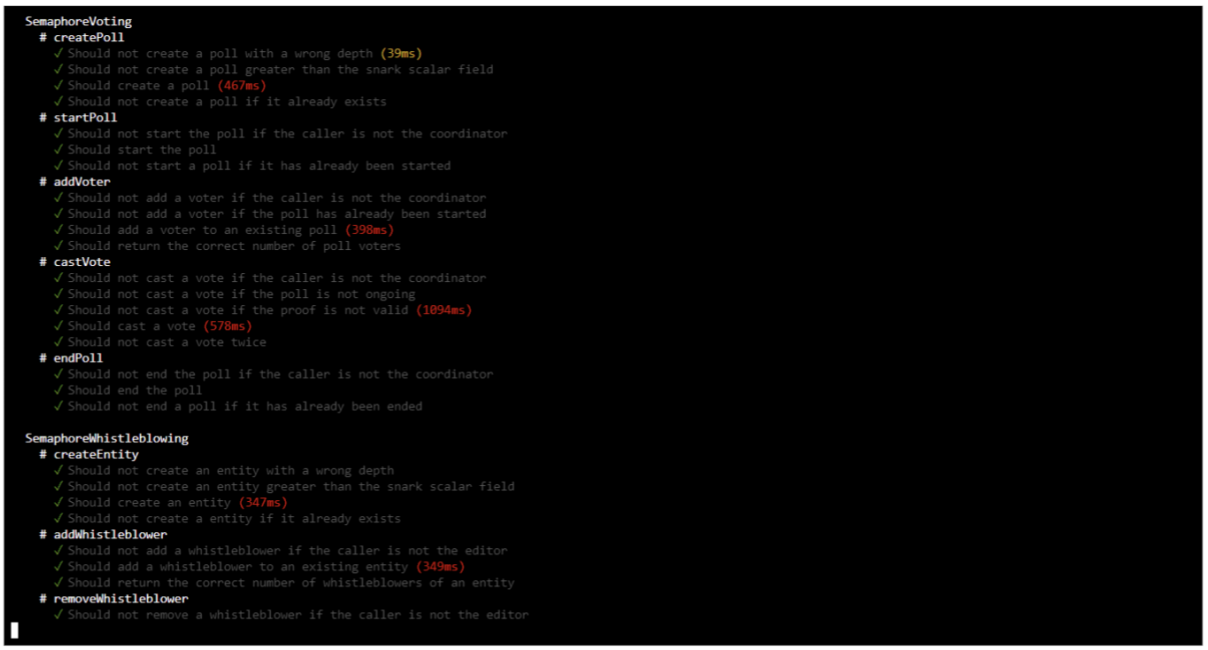
4. Whistleblowing
5. Rate limiting for spam protection

Q2. Clone the semaphore repo (3bce72f).

Q2.1. Run the tests and add a screenshot of all the test passing.

A2.1.

As the given commit(3bce72f) has 2 failed test cases, I tested on another(3d846da) commit as TA Lovelace stated on the discord message



Q2.2. Explain code in the sempahore.circom file (including public, private inputs).

A2.2.

Pubic inputs are given

- signalHash : the hash of the signal to broadcast
- externalNullifier : the 29-byte external nullifier

External nullifier prevent particular users from broadcasting multiple times using the same external nullifier. Default parameter of semaphore(nLevels) is 20 which is number of sibling hashes of proof and path indices.

```
component main {public [signalHash, externalNullifier]} = Semaphore(20);
```

semaphore.circom - public inputs

Private input are given

- identityNullifier : a random 32-byte value which represents the user's identity.
- identityTrapdoor : a random 32-byte value also the user should save.
- treePathIndices[nLevels] : binary expression of hash direction, 0 choose left node , 1 choose right node.
- treeSiblings[nLevels] : merkle proof hashes.

```
template Semaphore(nLevels) {
    signal input identityNullifier;
    signal input identityTrapdoor;
    signal input treePathIndices[nLevels];
    signal input treeSiblings[nLevels];
```

semaphore.circom - private inputs

Two outputs are as follows.

- root : calculated hash of root node.
- nullifierHash : square of signalHash to prevent tampering signalHash.

```
    signal output root;
    signal output nullifierHash;
```

semaphore.circom - outputs

Then, calculate the secret which is the Poseidon hash of identityNullifier and identityTrapdoor.

```
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;
}
```

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

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

semaphore.circom - secre calculation

The calculated secret hashed again using the Poseidon algorithm.

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

    signal output out;

    component poseidon = Poseidon(1);

    poseidon.inputs[0] <== secret;
```

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

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

semaphore.circom - secret calculation

The calculated secret hashed again using the Poseidon algorithm to identity commitment.

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

    signal output out;

    component poseidon = Poseidon(1);

    poseidon.inputs[0] <== secret;

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

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

semaphore.circom - identity commitment calculation

The nullifier hash is calculated by hashing the external and identity nullifier. Then, save it as output.

```
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;
}
```

```
template MerkleTreeInclusionProof(nLevels) {
    signal input leaf;
    signal input pathIndices[nLevels];
    signal input siblings[nLevels];
```

```
    signal output root;

    component poseidons[nLevels];
    component mux[nLevels];

    signal hashes[nLevels + 1];
    hashes[0] <== leaf;

    for (var i = 0; i < nLevels; i++) {
        pathIndices[i] * (1 - pathIndices[i]) === 0;

        poseidons[i] = Poseidon(2);
        mux[i] = MultiMux1(2);

        mux[i].c[0][0] <== hashes[i];
        mux[i].c[0][1] <== siblings[i];

        mux[i].c[1][0] <== siblings[i];
        mux[i].c[1][1] <== hashes[i];

        mux[i].s <== pathIndices[i];

        poseidons[i].inputs[0] <== mux[i].out[0];
        poseidons[i].inputs[1] <== mux[i].out[1];

        hashes[i + 1] <== poseidons[i].out;
    }

    root <== hashes[nLevels];
}
```

```
    component calculateNullifierHash = CalculateNullifierHash();
    calculateNullifierHash.externalNullifier <== externalNullifier;
    calculateNullifierHash.identityNullifier <== identityNullifier;
...
    nullifierHash <== calculateNullifierHash.out;
```

semaphore.circom - nullifierHash calculation

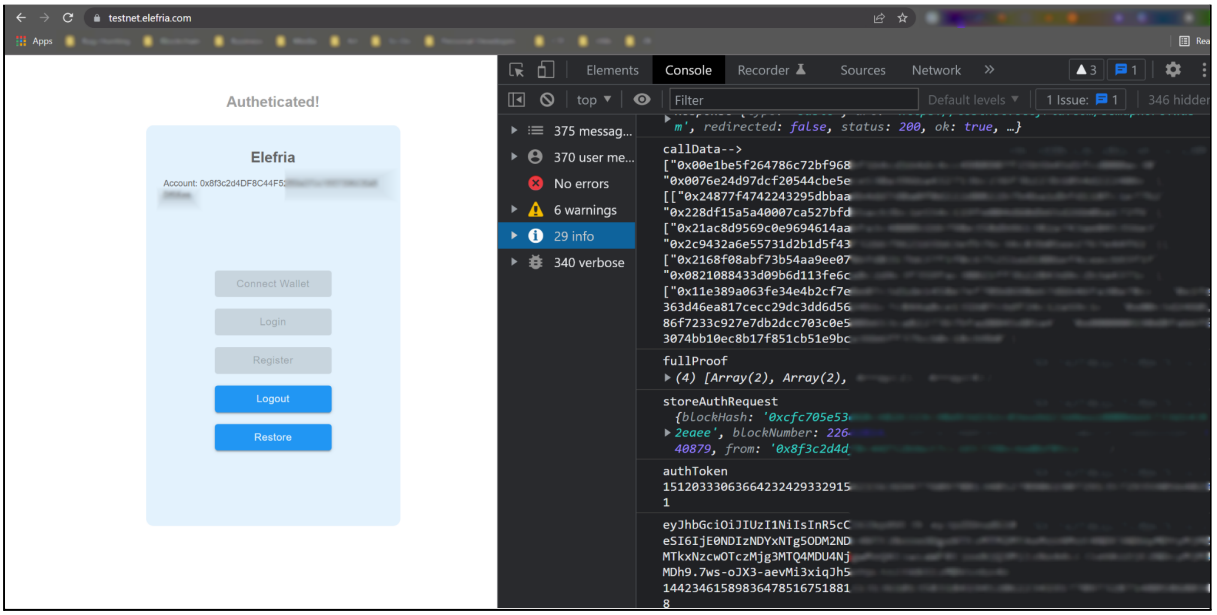
Calculate the root hash using previously calculated identity commitment as leaf node and path indices and sibling hashes given as private inputs. The calculated hash of the root is saved in the output.

```
    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;
```

semaphore.circom - outputs

Q3. Use Elefria protocol on the Harmony Testnet, try to generate a ZK identity and authenticate yourself as a user.

A3.



semaphore.circom - outputs

Q3.1 What potential challenges are there to overcome in such an authentication system?

A3.1

1. Registration and authentication speed is dependent on the transactions' inclusion on the blockchain network. Fast network and transaction throughput are prerequisite for the user experience.
2. The JWT token is issued to the user when the authentication procedure is successful. Because this token is such an important part of a user's identification, there should be no security vulnerabilities on the frontend web, and security measures to avoid a man-in-the-middle attack should be implemented.

Question 3. Tornado Cash

You will need these resources:

- tornadocash/tornado-trees
- tornadocash/tornado-nova
- *[Bonus]* Lecture from Roman Semenov, Co-founder of Tornado Cash

Q1. 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.

A1.

The tornado nova adopts commitment and nullifier.

The commitment is a hash of amount, public key, and random number. By checking the amount to transfer in transactions and public amounts, a user can transfer an arbitrary amount within the public amount. Before this, the user has to transfer a predetermined amount of token because the user could be traced if transfers a unique amount of token. By using the tornado nova user can preserve privacy by splitting the withdrawal amount.

The Nullifier checks whether the inputs are duplicated. This prevents double spending issues by checking the nullifier is submitted before.

Q2. Check out the tornado-trees repo

Q2.1. 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).

A2.1.

The tree height is set to 8 and chunk size is set to 2**8. To withdraw the deposited value and update the tree, the following parameter is required.

- _proof : This parameter used to verify the snark inputs(_pathIndices, _newRoot , _currentRoot, _events) were accurately supplied.
- _argsHash : A hash of snark inputs(_pathIndices, _newRoot , _currentRoot) mod SNARK_FIELD(defined as constant).
- _currentRoot : Current merkle tree root
- _newRoot : Merkle tree root after withdrawal
- _pathIndices : Merkle path of batch proofs.
- _events : A batch of inserted events (leaves). This parameter is stored in calldata to save the gas cost.

```
  uint256 public constant CHUNK_TREE_HEIGHT = 8;
  uint256 public constant CHUNK_SIZE = 2**CHUNK_TREE_HEIGHT;
  uint256 public constant ITEM_SIZE = 32 + 20 + 4;
  uint256 public constant BYTES_SIZE = 32 + 32 + 4 + CHUNK_SIZE * ITEM_SIZE;
  uint256 public constant SNARK_FIELD =
21888242871839275222246405745257275088548364400416034343698204186575808495617;
```

```
  function updateWithdrawalTree(
    bytes calldata _proof,
    bytes32 _argsHash,
    bytes32 _currentRoot,
    bytes32 _newRoot,
    uint32 _pathIndices,
    TreeLeaf[CHUNK_SIZE] calldata _events
  )
```

TornadoTree.sol - arguments of updateWithdrawalTree

Checks proof path index is valid and given _currentRoot hash is saved hash of root node.

```
    uint256 offset = lastProcessedWithdrawalLeaf;
    require(_currentRoot == withdrawalRoot, "Proposed withdrawal root is invalid");
    require(_pathIndices == offset >> CHUNK_TREE_HEIGHT, "Incorrect withdrawal insert
index");
```

TornadoTree.sol - _pathIndices, _currentRoot check

Save the _pathIndices, _newRoot, _currentRoot and leaf hashes of proofs(instance, hash, blockNumber) to the data variable.

```
    bytes memory data = new bytes(BYTES_SIZE);
    assembly {
```

```
      mstore(add(data, 0x44), _pathIndices)
      mstore(add(data, 0x40), _newRoot)
      mstore(add(data, 0x20), _currentRoot)
    }
    for (uint256 i = 0; i < CHUNK_SIZE; i++) {
      (bytes32 hash, address instance, uint32 blockNumber) = (_events[i].hash,
_events[i].instance, _events[i].block);
      bytes32 leafHash = keccak256(abi.encode(instance, hash, blockNumber));
      bytes32 deposit = offset + i >= depositsV1Length ? deposits[offset + i] :
tornadoTreesV1.deposits(offset + i);
      require(leafHash == deposit, "Incorrect deposit");
      assembly {
        let itemOffset := add(data, mul(ITEM_SIZE, i))
        mstore(add(itemOffset, 0x7c), blockNumber)
        mstore(add(itemOffset, 0x78), instance)
        mstore(add(itemOffset, 0x64), hash)
      }
      if (offset + i >= depositsV1Length) {
        delete deposits[offset + i];
      } else {
        emit DepositData(instance, hash, blockNumber, offset + i);
      }
    }
```

TornadoTree.sol - proof element construction

Checks the hash of the structured proof element is valid and executes verifyProof function through external call. If verification succeeds, a new state is saved on the chain.

```
    uint256 argsHash = uint256(sha256(data)) % SNARK_FIELD;
    require(argsHash == uint256(_argsHash), "Invalid args hash");
    require(treeUpdateVerifier.verifyProof(_proof, [argsHash]), "Invalid deposit tree
update proof");

    previousDepositRoot = _currentRoot;
    depositRoot = _newRoot;
    lastProcessedDepositLeaf = offset + CHUNK_SIZE;
  }
```

TornadoTree.sol - verify proof and save new state

The arguments of the circuit are the same as the structure proof element of solidity code.

```
template TreeUpdateArgsHasher(nLeaves) {
    signal input oldRoot;
    signal input newRoot;
    signal input pathIndices;
    signal input instances[nLeaves];
```

```
    signal input hashes[nLeaves];
    signal input blocks[nLeaves];
    signal output out;
```

TornadoTree.sol - verify proof and save new state

The merkle root is calculated using inputs and the sha256 hash of inputs.

```
    var index = 0;
    hasher.in[index++] <== 0;
    hasher.in[index++] <== 0;
    for(var i = 0; i < 254; i++) {
        hasher.in[index++] <== bitsOldRoot.out[253 - i];
    }
    hasher.in[index++] <== 0;
    hasher.in[index++] <== 0;
    for(var i = 0; i < 254; i++) {
        hasher.in[index++] <== bitsNewRoot.out[253 - i];
    }
    for(var i = 0; i < 32; i++) {
        hasher.in[index++] <== bitsPathIndices.out[31 - i];
    }
    for(var leaf = 0; leaf < nLeaves; leaf++) {
        // the range check on hash is optional, it's enforced by the smart contract anyway
        bitsHash[leaf] = Num2Bits_strict();
        bitsInstance[leaf] = Num2Bits(160);
        bitsBlock[leaf] = Num2Bits(32);
        bitsHash[leaf].in <== hashes[leaf];
        bitsInstance[leaf].in <== instances[leaf];
        bitsBlock[leaf].in <== blocks[leaf];
        hasher.in[index++] <== 0;
        hasher.in[index++] <== 0;
        for(var i = 0; i < 254; i++) {
            hasher.in[index++] <== bitsHash[leaf].out[253 - i];
        }
        for(var i = 0; i < 160; i++) {
            hasher.in[index++] <== bitsInstance[leaf].out[159 - i];
        }
        for(var i = 0; i < 32; i++) {
            hasher.in[index++] <== bitsBlock[leaf].out[31 - i];
        }
    }
    component b2n = Bits2Num(256);
    for (var i = 0; i < 256; i++) {
        b2n.in[i] <== hasher.out[255 - i];
    }
    out <== b2n.out;
```

TornadoTree.sol - verify proof and save new state

Q2.2. Why do you think we use the SHA256 hash here instead of the Poseidon hash used elsewhere?

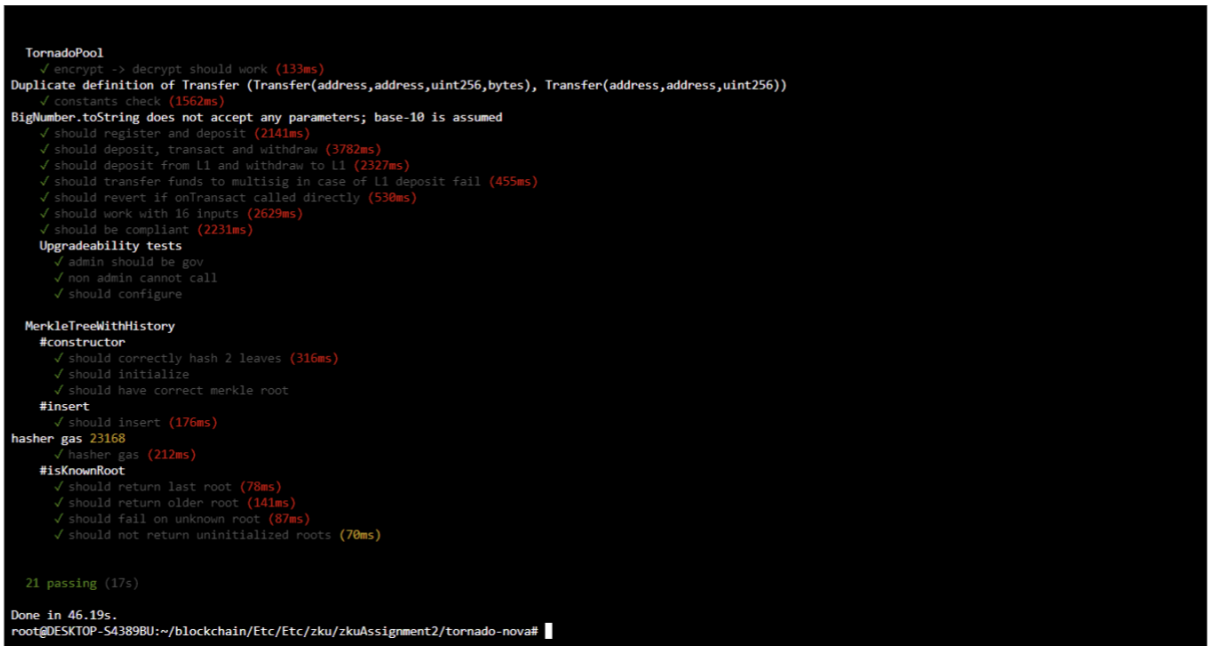
A2.2.

Implementing a specific hash algorithm that is not provided as a native function needs more time and effort to develop logic. In solidity, SHA256 hashing is natively usable. Furthermore, It can save gas using the native function rather than implementing a custom hash algorithm.

Q3. Clone/fork the tornado-nova repo

Q3.1. Run the tests and add a screenshot of all the tests passing.

A3.1.



Q3.2. Add a script named `custom.test.js` under `test/` and write a test for all of the followings in a single `it` function

Q3.2.1. estimate and print gas needed to insert a pair of leaves to `MerkleTreeWithHistory`

```
...
    const { merkleTreeWithHistory } = await loadFixture(fixture)
    var gas = await merkleTreeWithHistory.estimateGas.insert(toFixedHex(123),
toFixedHex(456))
    console.log('gas needed to insert a pair of leaves: ', gas.toString())
...
```

```
...
gas needed to insert a pair of leaves:  192309
...
```

Gas Estimation of inserting a pair of leaves

Q3.2.2. deposit 0.08 ETH in L1

```
async function fixture()
...
const token = await deploy('PermittableToken', 'Wrapped ETH', 'WETH', 18, l1ChainId)
    await token.mint(sender.address, utils.parseEther('10000'))
...
```

```
    const { tornadoPool, token, omniBridge } = await loadFixture(fixture)
    const depositorKeypair = new Keypair()

    const depositorDepositAmount = utils.parseEther('0.08')
    const depositorDepositUtxo = new Utxo({ amount: depositorDepositAmount, keypair:
depositorKeypair })
    const { args, extData } = await prepareTransaction({
      tornadoPool,
      outputs: [depositorDepositUtxo],
    })

    const onTokenBridgedData = encodeDataForBridge({
      proof: args,
      extData,
    })

    const onTokenBridgedTx = await tornadoPool.populateTransaction.onTokenBridged(
      token.address,
      depositorDepositUtxo.amount,
      onTokenBridgedData,
    )
```

```
    await token.transfer(omniBridge.address, depositorDepositAmount)
    const transferTx = await token.populateTransaction.transfer(tornadoPool.address,
depositorDepositAmount)

    await omniBridge.execute([
      { who: token.address, callData: transferTx.data },
      { who: tornadoPool.address, callData: onTokenBridgedTx.data },
    ])
```

deposit from L1 to L2

Q3.2.3. withdraw 0.05 ETH in L2

```
    const depositorWithdrawAmount = utils.parseEther('0.05')
    const recipient = '0x13b75E274Acb0eb9cEd75FcF7eA9AA28E5C7aa42'
    const depositorChangeUtxo = new Utxo({
      amount: depositorDepositAmount.sub(depositorWithdrawAmount),
      keypair: depositorKeypair,
    })

    await transaction({
      tornadoPool,
      inputs: [depositorDepositUtxo],
      outputs: [depositorChangeUtxo],
      recipient: recipient,
      isL1Withdrawal: true,
    })
```

```
async function transaction({ tornadoPool, ...rest }) {
  const { args, extData } = await prepareTransaction({
    tornadoPool,
    ...rest,
  })

  const receipt = await tornadoPool.transact(args, extData, {
    gasLimit: 2e6,
  })
  return await receipt.wait()
}
```

Withdraw from L2 to L1

Q3.2.4. assert recipient, omniBridge, and tornadoPool balances are correct

```
...
    const recipientBalance = await token.balanceOf(recipient)
    expect(recipientBalance).to.be.equal(0)
    const omniBridgeBalance = await token.balanceOf(omniBridge.address)
    expect(omniBridgeBalance).to.be.equal(utils.parseEther('0.05'))
    const tornadoPoolBalance = await token.balanceOf(tornadoPool.address)
    expect(tornadoPoolBalance).to.be.equal(utils.parseEther('0.03'))
...
```

```
...
✓ Q3.2 (4799ms)
  1 passing (5s)
...
```

Balance correction check after transaction

Question 4. Thinking In ZK

Q1. If you have a chance to meet with the people who built Tornado Cash & Semaphore, what questions would you ask them about their protocols?

A1.

1. Recently, the hacker who hacked parity wallet in 2017 has been spotted by researchers to move the stolen eth to tornado's service. Is this ethically right to open anonymous financial service access to random people? Is there community or technical measure available in the privacy platform such as blacklisting known criminal-related accounts by governance? [Ref link]
2. When considering a cryptographic algorithm to implement ZK proof, what is your priority to choose such an algorithm? (Ex, Gas cost, ease of development, reference code, mathematical strengths)