Zero Knowledge University (Assignment-1)

This is the first assignment report in the first week of Zero-Knowledge University (ZKU) cohort. In the first week of the cohort, ZKU teaches the student about the fundamental aspect of Zero-Knowledge, Merkle Tree, Circom, and NFT. The assignment consist of 3 (three) question which is answered in this report.

Based on my attempt to complete the assignment, I write this report by grouping the question into 3 (three) headings below.

Intro to Circom

According to the project's Readme, circom is a language for compiling circuit that is needed to compute different Zero-Knowledge proof. With that said, the first section of the assignment is to be familiar with the circom language to construct a circuit that takes a list of numbers input as leaves of Merkle tree and outputs the Merkle root.

Before we jump to the compiling process and the output, I will explain a little bit about the code that is written to complete the task. The circuit the I built using circom is based on signal operation that can be named with an identifier. The key to create Merkle tree function is to use the hashing function which in this case, I use `MiMCsponge` hash function from circomlib. Since I only need that one function, I can just copy the function and create some local file to be imported later.

In this task, I only create 2 (two) function called `MiMCHash` and `MerkleTree`. The `MiMCHash` as shown below, is the template/function that will hash 2 leafs into 1 hashed output. It is basically a wrapper to `MiMCsponge` function with predefined parameter.

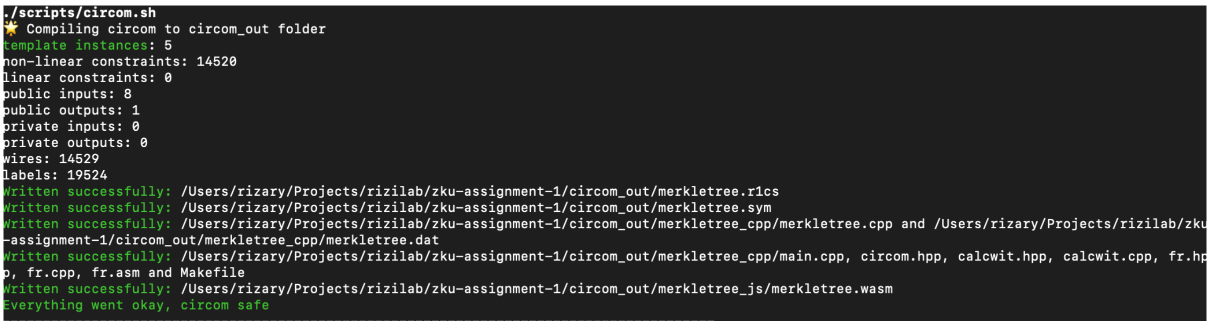
```
pragma circom 2.0.0;

// import mimcsponge.circom to get MiMCSponge template
include "mimcsponge.circom";

// MiMCHash template will hash 2 leafs into 1 hashed output
template MiMCHash(){
  signal input left;
  signal input right;
  signal output hash;

  component hashFunc = MiMCSponge(2, 220, 1);
  hashFunc.ins[0] <== left;
  hashFunc.ins[1] <== right;
  hashFunc.k <== 0;
  hash <== hashFunc.outs[0];
}
```

The next function is the `MerkleTree` function where the generation of merkle tree happened. The idea behind the function is to generate merkle tree using two iterations. The first iteration is to hash all the input and after that, generate

```
// Consturct MerkleTree with predefined input size -len-
//
template MerkleTree(len) {
    signal input leaves[len];
    signal output out;

    assert(len % 2 == 0);
    var arrLength = 2 * len - 1;
    signal hashResult[arrLength];
    var subs = len;
    var l = 0;
    var r = 1;
    component singleHash[len];
    component doubleHash[arrLength];

    var i;
    for(i = 0; i < len; i++) {
            singleHash[i] = MiMCSponge(1,220,1);
            singleHash[i].ins[0] <== leaves[i];
            singleHash[i].k <== 0;
```

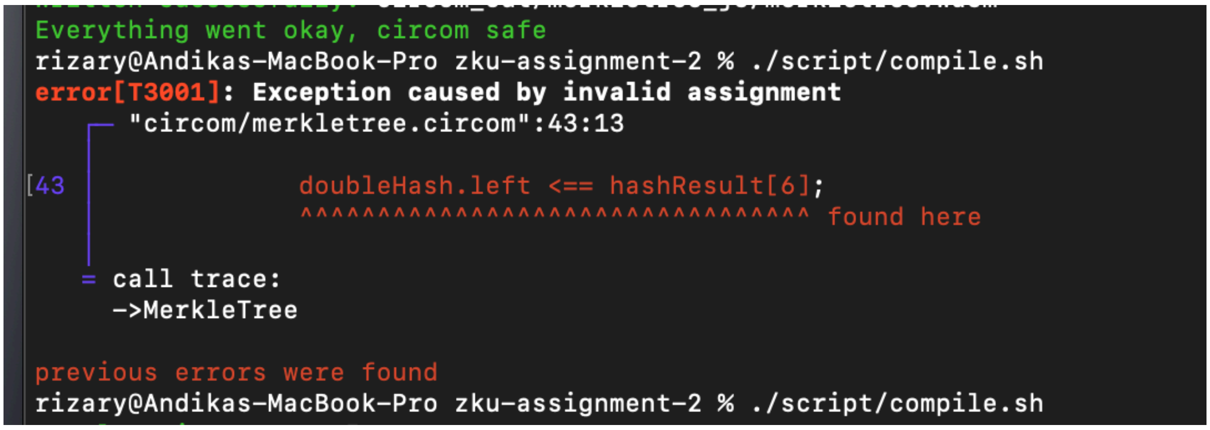
the merkle tree using the result of the first iteration. After that, the final output of the generation process will be the function output, or we usually called "Merkle Root". For further study, the complete code can be seen in: https://github.com/Rizary/zero-knowlegde-university-report/blob/zku-assignment-1/circom/merkletree.circom.

In order to test the function, I need to generate the proof using a list of 8 numbers, for example `{“leaves”:[1,2,3,4,5,6,7,8]}`. Following the circom documentation, the first step is to compile the circom and the output will be the following image:

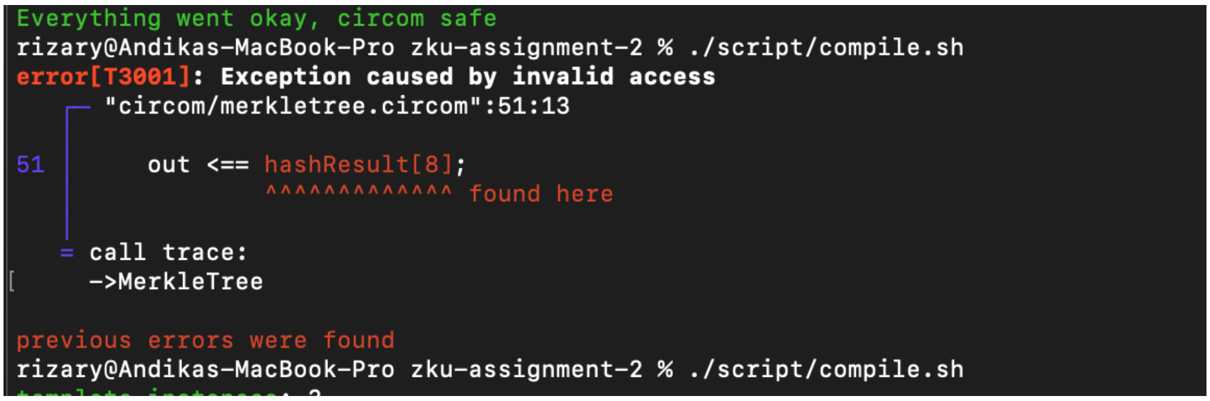


```
./scripts/circom.sh
🌟 Compiling circom to circom_out folder
template instances: 5
non-linear constraints: 14520
linear constraints: 0
public inputs: 8
public outputs: 1
private inputs: 0
private outputs: 0
wires: 14529
labels: 19524
Written successfully: /Users/rizary/Projects/rizilab/zku-assignment-1/circom_out/merkletree.r1cs
Written successfully: /Users/rizary/Projects/rizilab/zku-assignment-1/circom_out/merkletree.sym
Written successfully: /Users/rizary/Projects/rizilab/zku-assignment-1/circom_out/merkletree_cpp/merkletree.cpp and /Users/rizary/Projects/rizilab/zku
-assignment-1/circom_out/merkletree_cpp/merkletree.dat
Written successfully: /Users/rizary/Projects/rizilab/zku-assignment-1/circom_out/merkletree_cpp/main.cpp, circom.hpp, calcwit.hpp, calcwit.cpp, fr.hp
p, fr.cpp, fr.asm and Makefile
Written successfully: /Users/rizary/Projects/rizilab/zku-assignment-1/circom_out/merkletree_js/merkletree.wasm
Everything went okay, circom safe
```

The common mistake that I encountered was `invalid assignment` and `invalid access` error like the following:



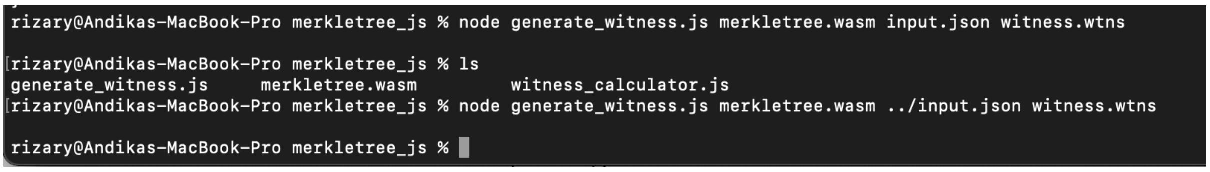
```
Everything went okay, circom safe
rizary@Andikas-MacBook-Pro zku-assignment-2 % ./script/compile.sh
error[T3001]: Exception caused by invalid assignment
    ┌─ "circom/merkletree.circom":43:13
    │
 43 │             doubleHash.left <== hashResult[6];
    │             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ found here
    │
    = call trace:
      ->MerkleTree

previous errors were found
rizary@Andikas-MacBook-Pro zku-assignment-2 % ./script/compile.sh
```



```
Everything went okay, circom safe
rizary@Andikas-MacBook-Pro zku-assignment-2 % ./script/compile.sh
error[T3001]: Exception caused by invalid access
    ┌─ "circom/merkletree.circom":51:13
    │
 51 │     out <== hashResult[8];
    │             ^^^^^^^^^^^^^ found here
    │
    = call trace:
      ->MerkleTree

previous errors were found
rizary@Andikas-MacBook-Pro zku-assignment-2 % ./script/compile.sh
```

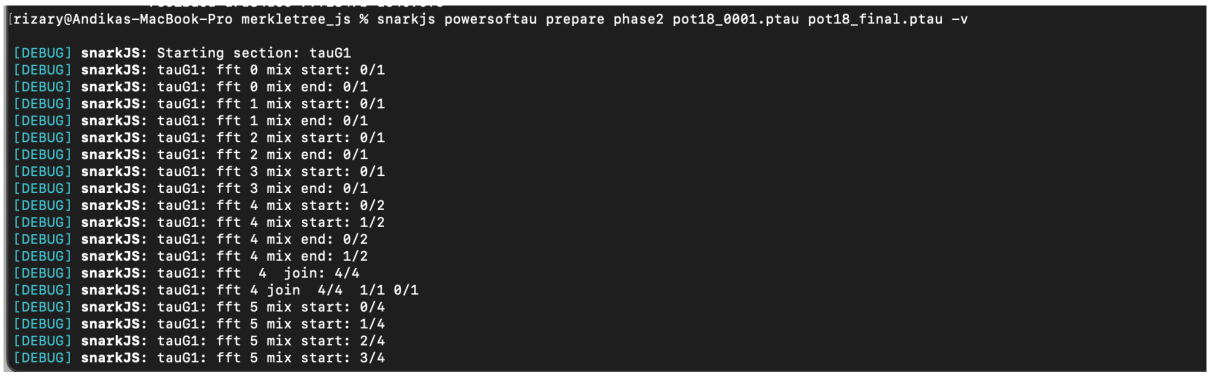
For invalid access, one thing that I learned is to make sure the element is existed inside the array. For invalid assignment, usually it happened because I try to put the signal result into non-signal variable.

After we compile the file, there is folder that called `merkletree.js` and we need to generate witness from it. The output of generating witness is like the following picture:



```
rizary@Andikas-MacBook-Pro merkletree_js % node generate_witness.js merkletree.wasm input.json witness.wtns

rizary@Andikas-MacBook-Pro merkletree_js % ls
generate_witness.js     merkletree.wasm         witness_calculator.js
rizary@Andikas-MacBook-Pro merkletree_js % node generate_witness.js merkletree.wasm ../input.json witness.wtns

rizary@Andikas-MacBook-Pro merkletree_js % 
```

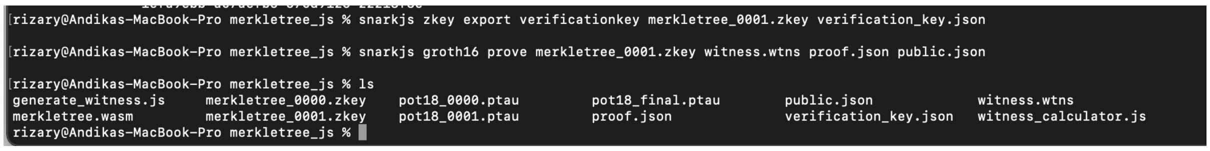
The next step is to preparing powers of tau ceremony and phase 2 using the snarkjs command.



```
rizary@Andikas-MacBook-Pro merkletree_js % snarkjs powersoftau prepare phase2 pot18_0001.ptau pot18_final.ptau -v

[DEBUG] snarkJS: Starting section: tauG1
[DEBUG] snarkJS: tauG1: fft 0 mix start: 0/1
[DEBUG] snarkJS: tauG1: fft 0 mix end: 0/1
[DEBUG] snarkJS: tauG1: fft 1 mix start: 0/1
[DEBUG] snarkJS: tauG1: fft 1 mix end: 0/1
[DEBUG] snarkJS: tauG1: fft 2 mix start: 0/1
[DEBUG] snarkJS: tauG1: fft 2 mix end: 0/1
[DEBUG] snarkJS: tauG1: fft 3 mix start: 0/1
[DEBUG] snarkJS: tauG1: fft 3 mix end: 0/1
[DEBUG] snarkJS: tauG1: fft 4 mix start: 0/2
[DEBUG] snarkJS: tauG1: fft 4 mix start: 1/2
[DEBUG] snarkJS: tauG1: fft 4 mix end: 0/2
[DEBUG] snarkJS: tauG1: fft 4 mix end: 1/2
[DEBUG] snarkJS: tauG1: fft  4  join: 4/4
[DEBUG] snarkJS: tauG1: fft 4 join  4/4  1/1 0/1
[DEBUG] snarkJS: tauG1: fft 5 mix start: 0/4
[DEBUG] snarkJS: tauG1: fft 5 mix start: 1/4
[DEBUG] snarkJS: tauG1: fft 5 mix start: 2/4
[DEBUG] snarkJS: tauG1: fft 5 mix start: 3/4
```

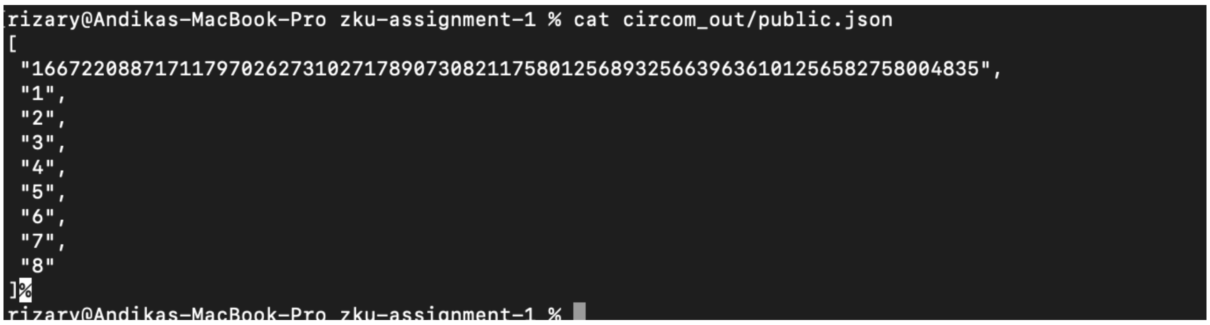
I should run phase2 after completing the powers of tau ceremony for specific circuit. Phase 2 start with generating a `.zkey` file that will contain the proving and verification keys together with all phase 2 contributions. In this process, we contribute to phase 2 of the ceremony using the generated `.zkey`. Once I finished with the ceremony, I generate the verification key by exporting it using snarkjs command.

The next step is to prove it by generate a zk-proof associated to the circuit and the witness after it is computed and the trusted setup is already executed. This will generate `public.json` and `proof.json` like the following picture:



```
rizary@Andikas-MacBook-Pro merkletree_js % snarkjs zkey export verificationkey merkletree_0001.zkey verification_key.json

rizary@Andikas-MacBook-Pro merkletree_js % snarkjs groth16 prove merkletree_0001.zkey witness.wtns proof.json public.json

rizary@Andikas-MacBook-Pro merkletree_js % ls
generate_witness.js     merkletree_0000.zkey    pot18_0000.ptau         pot18_final.ptau        public.json             witness.wtns
merkletree.wasm         merkletree_0001.zkey    pot18_0001.ptau         proof.json              verification_key.json   witness_calculator.js
rizary@Andikas-MacBook-Pro merkletree_js % 
```

In order to verify our proof, we can run snarkjs command and if the output is “OK”, then our proof is correct. The following image shows the generated public.json that contain the inputs and the outputs of generating proof.



```
rizary@Andikas-MacBook-Pro zku-assignment-1 % cat circom_out/public.json
[
 "16672208871711797026273102717890730821175801256893256639636101256582758004835",
 "1",
 "2",
 "3",
 "4",
 "5",
 "6",
 "7",
 "8"
]%
rizary@Andikas-MacBook-Pro zku-assignment-1 % 
```

The next question of this section assignment is, “***Do we really need zero-knowledge proof for this? Can a publicly verifiable smart contract that computes Merkle root achieve the same?***”.

My answer is “YES” to this question. We need a zero-knowledge proof for this even though publicly verifiable smart contract can achieve the same. Smart contract that computes Merkle root cannot achieve the same result compared to zero-knowledge proof because in order to proof and verify the Merkle tree, smart contract can or usually exposing all the data required to the public. Sometime for certain activity, exposing all the data and process just to proof it might be dangerous.

One scenario that Zero-Knowledge proofs like this will be useful is when we want to draw some money without having to show additional data to proof that the money that we draw is our money. Technology that already implementing this is called “Tornado Cash”. Basically, Tornado Cash is a decentralized finance application built on top of Ethereum network with the ability to help people anonymize their transaction. All the Ethereum transaction is public, but Tornado Cash can make it anonymous by using Zero-Knowledge proof technology.

If we look into Tornado Cash code here: https://github.com/tornadocash/tornado-core/blob/master/circuits/withdraw.circom, there is template/function that called `template Withdraw`. This function is used to verifies all the secret and nullifier is already in the merkle tree of the deposits. In this case, we don’t need to know the secret or nullifier to find it, we just use Zero-Knowledge proof technique to make sure that both secret and nullifier are there.

In this project, I created some of bash script in order to help automate the process from compiling to generating proof. I also use `just` command to help simplify the process. Please see the `justfile` in the root folder of the project.

NFT and Merkle Tree

The next assignment section is about creating an NFT and its implementation with Merkle Tree. For completing this task, I use Remix because of its simplicity and clean UI. Some feature of Remix is really good for speed up the smart contract development process. I use solidify pragma version ^0.8.0 and since this is NFT contract, I use openzeppelin library for ERC721.

The first thing that I create is a `MerkleTree` contract before the `ZkuNftToken` contract that inherit ERC721 contract. All the source code of the contract is available in https://github.com/Rizary/zero-knowlegde-university-report/blob/

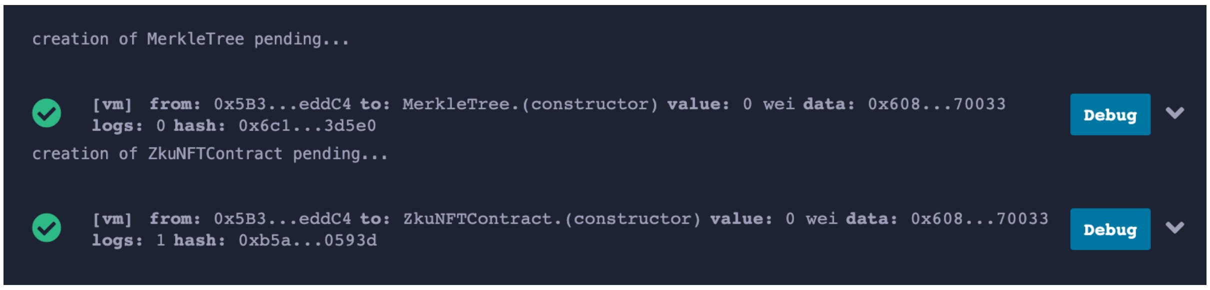
zku-assignment-1/contracts/ZkuNftToken.sol. However, in this report I will explain about the ZkuNftToken contract first.

The MerkleTree contract consist of 4 (four) main functions as shown below. There are 2 (two) minting function called `mint` for minting the NFT and send it to the message sender, and `mintForAddress`, a function for minting the NFT and send it to the destined address.

```
contract ZkuNFTContract is ERC721URIStorage, Ownable {
    using Strings for uint256;
    string public uriPrefix = "";
    string public uriSuffix = ".json";
    using Counters for Counters.Counter;
    Counters.Counter private _tokenId;

    constructor() ERC721("ZkuNftToken", "ZNT") {}

    // The mint function is used to mint NFT and give the result to this function caller.
    // Once the NFT is minted, this function will hashed the relevant information and
    // then update the transaction using MerkleTree's `updateTransaction`.
    function mint(address _merkleTree) public payable {
        _tokenId.increment();
        uint256 newTokenID = _tokenId.current();
        string memory _tokenURI = this.tokenURI();
        _safeMint(_msgSender(), newTokenID);
        bytes32 newLeaf = keccak256(abi.encodePacked(_msgSender(), _msgSender(),
newTokenID, _tokenURI));
        MerkleTree(_merkleTree).updateTransaction(newLeaf);
        MerkleTree(_merkleTree).generateTree();

    }

    // we call MerkleTree to move the simple getRoot() and be available in this file.
    function getMerkleRoot(address _merkleTree) external view returns (bytes32 root) {
        root = MerkleTree(_merkleTree).getRoot();
    }

    // mintForAddress is used by contract owner to mint an NFT address and
    // send it to the destination address.
    function mintForAddress(address _merkleTree, address _receiver) public onlyOwner {
        _tokenId.increment();
        uint256 newTokenID = _tokenId.current();
        string memory _tokenURI = this.tokenURI();
        _safeMint(_receiver, newTokenID);
        bytes32 newLeaf = keccak256(abi.encodePacked(_msgSender(), _receiver, newTokenID,
_tokenURI));
        MerkleTree(_merkleTree).updateTransaction(newLeaf);
        MerkleTree(_merkleTree).generateTree();
    }

    // TokenURI is the custom function to generate TokenURI on-chain and having the name
    // and description attached to it.
    function tokenURI() public view virtual returns (string memory) {
        uint256 tokenId = _tokenId.current();
        bytes memory dataURI = abi.encodePacked(
            “{",
                '"name": "ZkuNftToken #', tokenId.toString(), '",',
                '"description": "NFT Token for ZKU students around the world"',
            "}"
        );

        return string(
            abi.encodePacked(
                "data:application/json;base64,",
                Base64.encode(dataURI)
            )
        );
    }

    function _baseURI() internal view virtual override returns (string memory) {
        return uriPrefix;
    }
}
```

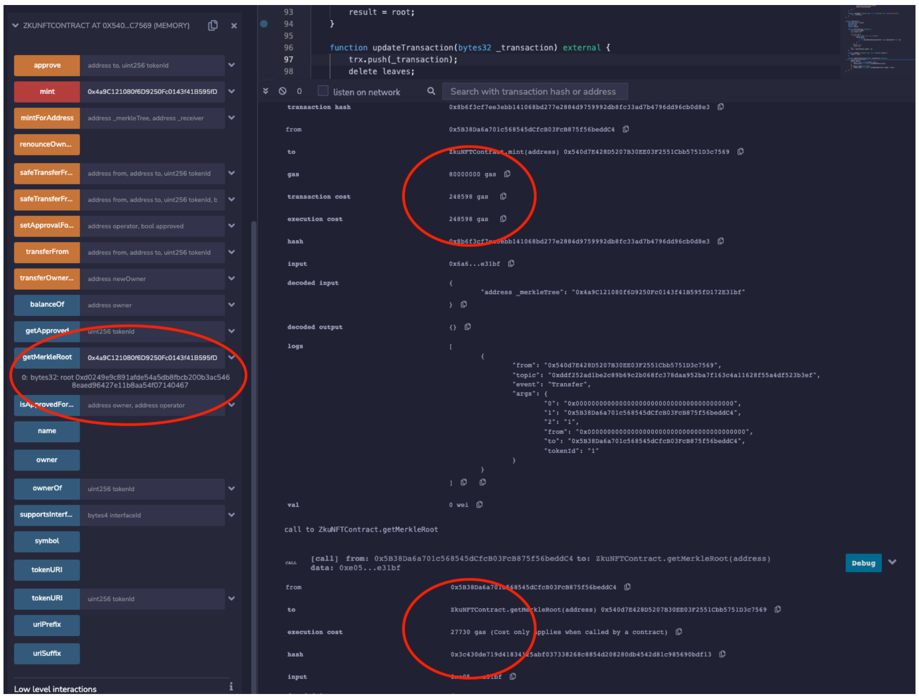
How `mint` and `mintForAddress` work are similar. The first step is to generate the `tokenId` and the `tokenURI()`. After that, the _safeMint() function obtained from openzeppelin library is used to mint the NFT. After minting the NFT, I commit the msg.sender, receiver address, tokenID, and tokenURI to a Merkle tree using the `keccak256` hash function and then updated the Merkle tree. All update and generate the Merkle tree is happened in the MerkleTree contract as you can see from the given link above.

Once the contract is compiled without any error, I began to test the contract in Remix IDE. The first is to test the mint function. If you look at the below image, the contract was successfully compiled.



```
creation of MerkleTree pending...

[vm] from: 0x5B3...eddC4 to: MerkleTree.(constructor) value: 0 wei data: 0x608...70033
logs: 0 hash: 0x6c1...3d5e0
creation of ZkuNFTContract pending...

[vm] from: 0x5B3...eddC4 to: ZkuNFTContract.(constructor) value: 0 wei data: 0x608...70033
logs: 1 hash: 0xb5a...0593d
```

I try to run the minting twice to make sure that the Merkle tree updated by looking at the Remix interface. The first attempt is to mint an NFT and send it to the owner. Finally, I run getRoot function to find the Merkle root value (please see image below).

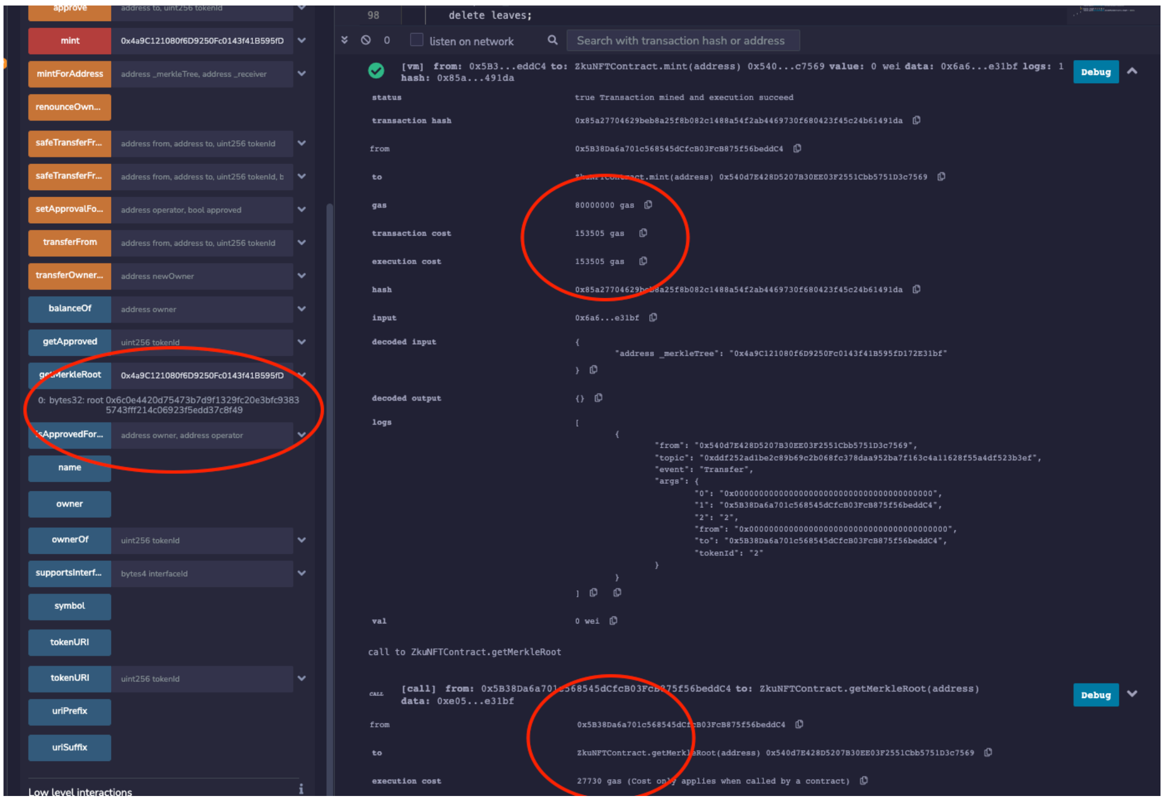


```
mint           0x4a9C121080f6D9250Fc0143f41B595fD
getMerkleRoot  0x4a9C121080f6D9250Fc0143f41B595fD
0: bytes32: root 0xd0249e9c891afde54a5db8fbcb200b3ac546
8eaed96427e11b8aa54f07140467
```

```
        result = root;
    }

    function updateTransaction(bytes32 _transaction) external {
        trx.push(_transaction);
        delete leaves;
```

```
transaction hash    0x8b6f3cf7ee3ebb141068bd277e2884d9759992db8fc33ad7b4796dd96cb0d8e3
from                0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
to                  ZkuNFTContract.mint(address) 0x540d7E428D5207B30EE03F2551Cbb5751D3c7569
gas                 80000000 gas
transaction cost    248598 gas
execution cost      248598 gas
hash                0x8b6f3cf7ee3ebb141068bd277e2884d9759992db8fc33ad7b4796dd96cb0d8e3
input               0x6a6...e31bf
decoded input       {
                        "address _merkleTree": "0x4a9C121080f6D9250Fc0143f41B595fD172E31bf"
                    }
decoded output      {}
logs                [
                        {
                            "from": "0x540d7E428D5207B30EE03F2551Cbb5751D3c7569",
                            "topic": "0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef",
                            "event": "Transfer",
                            "args": {
                                "0": "0x0000000000000000000000000000000000000000",
                                "1": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",
                                "2": "1",
                                "from": "0x0000000000000000000000000000000000000000",
                                "to": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",
                                "tokenId": "1"
                            }
                        }
                    ]
val                 0 wei

call to ZkuNFTContract.getMerkleRoot

CALL  [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: ZkuNFTContract.getMerkleRoot(address)
      data: 0xe05...e31bf
from                0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
to                  ZkuNFTContract.getMerkleRoot(address) 0x540d7E428D5207B30EE03F2551Cbb5751D3c7569
execution cost      27730 gas (Cost only applies when called by a contract)
hash                0x3c430de719d41834125abf037338268c8854d208280db4542d81c985690bdf13
input               0xe05...e31bf
```

The second attempt is conducted to proof that the Merkle root is changed because of different input. For further knowledge, please refer to the below image:

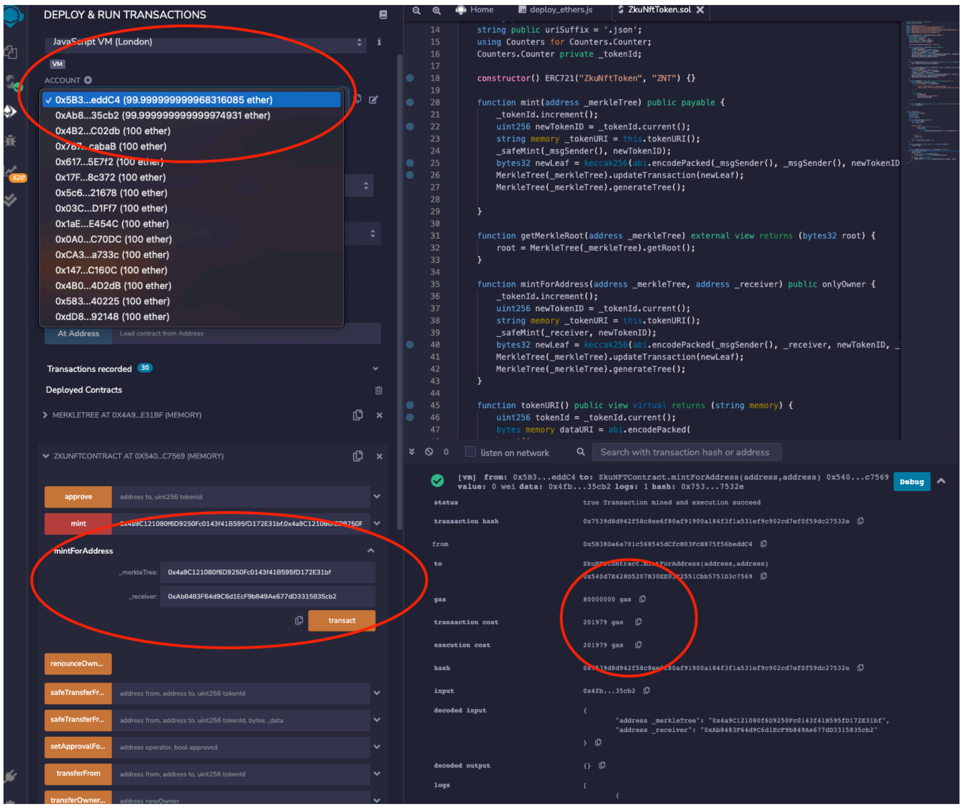


```
mint           0x4a9C121080f6D9250Fc0143f41B595fD
getMerkleRoot  0x4a9C121080f6D9250Fc0143f41B595fD
0: bytes32: root 0x6c0e4420d75473b7d9f1329fc20e3bfc9383
5743fff214c06923f5edd37c8f49
```

```
        delete leaves;
```

```
[vm] from: 0x5B3...eddC4 to: ZkuNFTContract.mint(address) 0x540...c7569 value: 0 wei data: 0x6a6...e31bf logs: 1
hash: 0x85a...491da
status              true Transaction mined and execution succeed
transaction hash    0x85a27704629beb8a25f8b082c1488a54f2ab4469730f680423f45c24b61491da
from                0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
to                  ZkuNFTContract.mint(address) 0x540d7E428D5207B30EE03F2551Cbb5751D3c7569
gas                 80000000 gas
transaction cost    153505 gas
execution cost      153505 gas
hash                0x85a27704629beb8a25f8b082c1488a54f2ab4469730f680423f45c24b61491da
input               0x6a6...e31bf
decoded input       {
                        "address _merkleTree": "0x4a9C121080f6D9250Fc0143f41B595fD172E31bf"
                    }
decoded output      {}
logs                [
                        {
                            "from": "0x540d7E428D5207B30EE03F2551Cbb5751D3c7569",
                            "topic": "0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef",
                            "event": "Transfer",
                            "args": {
                                "0": "0x0000000000000000000000000000000000000000",
                                "1": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",
                                "2": "2",
                                "from": "0x0000000000000000000000000000000000000000",
                                "to": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",
                                "tokenId": "2"
                            }
                        }
                    ]
val                 0 wei

call to ZkuNFTContract.getMerkleRoot

CALL  [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: ZkuNFTContract.getMerkleRoot(address)
      data: 0xe05...e31bf
from                0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
to                  ZkuNFTContract.getMerkleRoot(address) 0x540d7E428D5207B30EE03F2551Cbb5751D3c7569
execution cost      27730 gas (Cost only applies when called by a contract)
```

After testing the mint function that send the NFT to the owner, then I run the mintForAddress function to make sure that we can send the NFT to other address and update the Merkle tree accordingly. Running the mintForAddress the second time is necessary to make sure the root generated is different.



DEPLOY & RUN TRANSACTIONS

JavaScript VM (London)

VM

ACCOUNT

```
✓ 0x5B3...eddC4 (99.999999999968316085 ether)
  0xAb8...35cb2 (99.999999999999974931 ether)
  0x4B2...C02db (100 ether)
  0x787...cabaB (100 ether)
  0x617...5E7f2 (100 ether)
  0x17F...8c372 (100 ether)
  0x5c6...21678 (100 ether)
  0x03C...D1Ff7 (100 ether)
  0x1aE...E454C (100 ether)
  0x0A0...C70DC (100 ether)
  0xCA3...a733c (100 ether)
  0x147...C160C (100 ether)
  0x4B0...4D2dB (100 ether)
  0x583...40225 (100 ether)
  0xdD8...92148 (100 ether)
```

At Address — Load contract from Address

Transactions recorded 30

Deployed Contracts

MERKLETREE AT 0X4A9...E31BF (MEMORY)

ZKUNFTCONTRACT AT 0X540...C7569 (MEMORY)

```
approve            address to, uint256 tokenId
mint               0x4a9C121080f6D9250Fc0143f41B595fD172E31bf,0x4a9C121080f6D9250F
mintForAddress
    _merkleTree:   0x4a9C121080f6D9250Fc0143f41B595fD172E31bf
    _receiver:     0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2
                                                   transact
renounceOwn...
safeTransferFr...  address from, address to, uint256 tokenId
safeTransferFr...  address from, address to, uint256 tokenId, bytes _data
setApprovalFo...   address operator, bool approved
transferFrom       address from, address to, uint256 tokenId
transferOwner...   address newOwner
```

Home | deploy_ethers.js | ZkuNftToken.sol

```solidity
    string public uriSuffix = '.json';
    using Counters for Counters.Counter;
    Counters.Counter private _tokenId;

    constructor() ERC721("ZkuNftToken", "ZNT") {}

    function mint(address _merkleTree) public payable {
        _tokenId.increment();
        uint256 newTokenID = _tokenId.current();
        string memory _tokenURI = this.tokenURI();
        _safeMint(_msgSender(), newTokenID);
        bytes32 newLeaf = keccak256(abi.encodePacked(_msgSender(), _msgSender(), newTokenID
        MerkleTree(_merkleTree).updateTransaction(newLeaf);
        MerkleTree(_merkleTree).generateTree();

    }

    function getMerkleRoot(address _merkleTree) external view returns (bytes32 root) {
        root = MerkleTree(_merkleTree).getRoot();
    }

    function mintForAddress(address _merkleTree, address _receiver) public onlyOwner {
        _tokenId.increment();
        uint256 newTokenID = _tokenId.current();
        string memory _tokenURI = this.tokenURI();
        _safeMint(_receiver, newTokenID);
        bytes32 newLeaf = keccak256(abi.encodePacked(_msgSender(), _receiver, newTokenID, _
        MerkleTree(_merkleTree).updateTransaction(newLeaf);
        MerkleTree(_merkleTree).generateTree();
    }

    function tokenURI() public view virtual returns (string memory) {
        uint256 tokenId = _tokenId.current();
        bytes memory dataURI = abi.encodePacked(
```

listen on network — Search with transaction hash or address

```
[vm]  from: 0x5B3...eddC4 to: ZkuNFTContract.mintForAddress(address,address) 0x540...c7569
value: 0 wei data: 0x4fb...35cb2 logs: 1 hash: 0x753...7532e
status              true Transaction mined and execution succeed
transaction hash    0x7539d8d942f58c8ee6f80af91900a184f3f1a531ef9c902cd7ef0f59dc27532e
from                0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
to                  ZkuNFTContract.mintForAddress(address,address)
                    0x540d7E428D5207B30EE0[illegible]2551Cbb5751D3c7569
gas                 80000000 gas
transaction cost    201979 gas
execution cost      201979 gas
hash                0x7539d8d942f58c8ee6f80af91900a184f3f1a531ef9c902cd7ef0f59dc27532e
input               0x4fb...35cb2
decoded input       {
                        "address _merkleTree": "0x4a9C121080f6D9250Fc0143f41B595fD172E31bf",
                        "address _receiver": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2"
                    }
decoded output      {}
logs                [
                        {
```

All the gas costs are circled in red.

Due to time constraint, I couldn't finished the bonus question for this task.

Ideas on ZK Technologies

zk-SNARKs and zk-STARKs differences

The last topic on the assignment is around ZK Technologies. Before we jump into the idea and how we can apply ZK in some activity, we should know the key differences between SNARKs and STARKs in application.

zk-SNARKs and zk-STARKs are two most compelling zero-knowledge technologies in the market. Even though both technologies are non-interactive by nature (the code can be deployed and act autonomously), they have some key differences. The first different in applying both technologies is that zk-

SNARKs documentation is better than zk-STARKs. Documentation is one of the key of success for every technology. Good documentation can lead to easy adoption and attract more developer to try and build application on top of it. Zk-SNARKs first advantage is they have good documentation and that makes it easy for developer to build an application on top of it. Two company that is already use zk-SNARKs are ZCash and JP Morgan's payment system.

The next is differences in the application is that zk-STARKs do not required an initial trusted setup because it rely on leaner cryptography through collision-resistant hash function. In this case zk-STARKs does not require initial trusted setup, unlike zk-SNARKs. Due to this, the third differences is that zk-SNARKs size of the proofs is smaller (which mean it would take less on-chain storage) than zk-STARKs. The bigger size of the proofs can lead some limitation when it gets implemented for some context.

Lastly, zk-SNARKs has only require 24% of that gas that STARKs would require. It means that zk-SNARKs is cheaper than zk-STARKs in term of gas fee. This will be advantages from the end-user point of view.

Groth16 and PLONK trusted setup

Groth16 trusted setup is related to the proving system, which is circuit-specific common reference string. This means that in the setup phase for the prover, the proof system of Groth16 can only support a fixed circuit. Additionally, when the proof system is used by other applications, Groth16 user must re-run the setup phase with different parameters. Thus, if we change or modify private smart contract (in this case is the circuit), we need to make a new trusted setup.

On the other hand, PLONK can be validated with just one trusted setup. It is because PLONK uses a new circuit description that consists of gates, multiplication (x) and additions (+). With this advantage, we do not need to make new trusted setup every time we change or modify our smart contract.

Applying Zero-Knowledge to Unique NFTs

NFT can be a new way to identified personal ownership of some digital asset, or digital identity. Applying Zero-Knowledge to NFT that I can think of is when some country or multinational company wants to have their citizen or employee own digital identity through NFT. Every time the government or the company created NFT for the employee, the metadata in their NFT can be submitted to Zero-Knowledge technologies and once it is inside the Merkle tree, people do not have to show their ID or Social Number whenever they want to proof that they work or a citizen of the organizations.

Zero-Knowledge can also be applied to fans of some sports club or an artist. By implementing Zero-Knowledge to NFTs, fans can be more comfortable when they can proof that they are part of the community, without having to show anything that is secret for them.

Applying Zero-Knowledge for DAO Tooling

Decentralized Autonomous Organizations (DAO) is the next step of bringing freedom to running an organizations. In order to achieved full decentralization, DAO needs some tools that can support their day-to-day activity. One idea that I have about the implementation of Zero-Knowledge for DAO tooling is regarding DAO membership and contribution tracker.

In the sense of membership, people should buy NFT that is issue by the DAO. This NFT already use Zero-Knowledge to keep the metadata and people is required to have it before joining the DAO. Once joining the DAO, they can participate on DAO activity without having to show the important data that they are a member. The tooling will achieve that so DAO can be organized and more trusted.

In the sense of contribution tracker, every contribution is recorded in the Merkle tree, and every time people want to achieve some reward, people just need to proof their contribution. This is also helpful when some person cannot contribute due to the constraint from their current employer.