e-mail : dalgarim@protonmail.com
discord : 구자현#7593
github : https://github.com/dalgarim/zkuAssignment1

Question 1: Intro to circom

Q 1.1

Construct a circuit using circom that takes a list of numbers input as leaves of a Merkle tree (Note that the numbers will be public inputs) and outputs the Merkle root. For the Merkle hash function, you may use the MiMCsponge hash function from circomlib. For simplicity, you may assume that the number of leaves will be a power of 2 (say 4) and the input will look like this {“leaves”:[1,2,3,4]}

A 1.1

- The assumption that the number of leaves is the power of 2 is given, we can fix the number of nodes as (N*2-1).
- As there is no condition on the MIMC key, I added an additional input signal k for test purpose, which is used in MIMC hashing.

```
pragma circom 2.0.0;

include "./mimcsponge.circom";

template MerkleHashGenerator(N) {
    signal input in[N];
    signal input k;
    signal output out;
    signal nodeHashes[(N*2)-1];

    component mimc1[N];
//Generate the hashes of leaf nodes
    for(var i = 0; i < N; i++) {
        mimc1[i] = MiMCSponge(1, 220, 1);
        mimc1[i].ins[0] <== in[i];
        mimc1[i].k <== k;
        nodeHashes[i] <== mimc1[i].outs[0];
    }

    component mimc2[N-1];

    var n = N;
    var offset = 0;
    var j = N;
    var q = 0;

//Generate the hashes of branch nodes
    while( n > 0 ) {
        for (var i = 0; i < n - 1; i += 2) {
            mimc2[q] = MiMCSponge(2, 220, 1);
            mimc2[q].ins[0] <== nodeHashes[offset + i];
```

```
            mimc2[q].ins[1] <== nodeHashes[offset + i + 1];
            mimc2[q].k <== k;
            nodeHashes[j] <== mimc2[q].outs[0];
            j++;
            q++;
        }
        offset += n;
        n = n / 2;
    }

    out <== nodeHashes[j-1];
}

component main = MerkleHashGenerator(8);
```

MerkleHashGenerator.circom

- The input.json is as follows.

```
{"in": [1, 2, 3, 4, 5, 6, 7, 8], "k": 0}
```

input.json

- The public.json is as follows.

```
[
"19839177484708313268194175574865510200583871330606097214497167062829556362280"
]
```

public.json

```
[0] 879224641071972007407379435558085566277229243840993668898356441948678255658 7
[1] 202322639608987835421883279913822405963043419098932782832760378988874916335 55
[2] 174309981911130438993543681286064362627376439356990835812974072479367745835 2
[3] 773158273373985560748626183763290158409132413003292594497379628923616821860 4
[4] 166033238090312530721875437163676316047372998666665721541368449347401258371 16
[5] 162683859563410646402915533851663941077092287493848191523619467772270193968 28
[6] 178025647497864216018747029463094758102880399529650708971400600895080933235 6
[7] 818698756701931358328102386496342858713807990133400474094575712998275156 40
[8] 192219742594157833068360760859000601910940808015609388947370071689478782177 12
[9] 199347451252090221337819248120860082592159693295258493720803838624436888202 50
[10] 193744524619244467158023513587738688253981065540940472627594983265466426557 96
[11] 136283638173748072351990005115498774277108372446838198534783570697123180071 74
[12] 192452946455280447498755690084161939830072476048134663782831916236002866966 13
[13] 166722088717117970262731027178907308211758012568932566396361012565827580048 35
[14] 198391774847083132681941755748655102005838713306060972144971670628295563622 80
```

Debug log

Q 1.2

Now try to generate the proof using a list of 8 numbers. Document any errors (if any) you encounter when increasing the size and explain how you fixed them.

A 1.2

- While generating zkey, the following error has occurred.
- I fixed the error by changing the power number of ceremony.

```
#> snarkjs groth16 setup merklehashgenerator.r1cs pot13_final.ptau
merklehashgenerator_0000.zkey

[ERROR] snarkJS: circuit too big for this power of tau ceremony. 14520*2 > 2**13
```

error message

Q 1.3

Do we really need zero-knowledge proof for this? Can a publicly verifiable smart contract that computes Merkle root achieve the same? If so, give a scenario where Zero-Knowledge proofs like this might be useful. Are there any technologies implementing this type of proof? Elaborate in 100 words on how they work.

A 1.3

Given that storage slot and transaction data of major blockchain projects are publicly accessible, it is not desirable to implement Merkle inclusion verification of secret data. If a user submits hash or validation key to contract directly, this information could be exploited by a frontrunner or other illicit actors. TornadoCash is one example of using ZK-SNARK as a validation of deposit. User can deposit ETH with data which can be used when user withdraw deposits. The data have no information of origin transaction and thus, withdrawal has no references to depositor.

Q 1.4

[Bonus] As you may have noticed, compiling circuits and generating the witness is an elaborate process. Explain what each step is doing. Optionally, you may create a bash script and comment on each step in it. This script will be useful later on to quickly compile circuits.

A 1.3

```
#!/bin/bash
if [ $# -ne 2 ]; then
 echo "Usage: $0 param1 param2"
 echo "param1: file name"
 echo "param2: tau ceremony power"
 exit -1
else
 echo "ok"
fi
```

```
filename=$1
power=$2
project=`echo $filename | sed "s/.circom//g"`

# Compile the circuit
circom $filename --r1cs --wasm --sym --c

# Computing the witness with WebAssembly
cp ./input.json "./${project}_js/"
cd "./${project}_js/"
node generate_witness.js "${project}.wasm" input.json witness.wtns
mv ./witness.wtns ../witness.wtns
cd ..

# Start a new "powers of tau" ceremony
snarkjs powersoftau new bn128 $power "pot${power}_0000.ptau" -v

# contribute to the ceremony
snarkjs powersoftau contribute "pot${power}_0000.ptau" "pot${power}_0001.ptau"
--name="First contribution" -v

# start the generation of phase2
snarkjs powersoftau prepare phase2 "pot${power}_0001.ptau" "pot${power}_final.ptau" -v

# Generate a .zkey file that will contain the proving and verification keys together with
all phase 2 contributions
snarkjs groth16 setup "${project}.r1cs" "pot${power}_final.ptau" "${project}_0000.zkey"

# Contribute to the phase 2 of the ceremony
snarkjs zkey contribute "${project}_0000.zkey" "${project}_0001.zkey" --name="1st
Contributor Name" -v

# Export the verification key
snarkjs zkey export verificationkey "${project}_0001.zkey" verification_key.json

# Generate a zk-proof associated to the circuit and the witness
snarkjs groth16 prove "${project}_0001.zkey" witness.wtns proof.json public.json

# Verify the proof
snarkjs groth16 verify verification_key.json public.json proof.json

# Generate the Solidity code
snarkjs zkey export solidityverifier "${project}_0001.zkey" verifier.sol

# Generate the parameters of the call
snarkjs generatecall
```

Automation Script

Question 2: Minting an NFT and committing the mint data to a Merkle Tree

Q 2.1

Create an ERC721 contract that can mint an NFT to any address. The token URI should be on-chain and should include a name field and a description field. [Bonus points for well-commented codebase]

A 2.1

- Used OpeenZeppelin's ERC721 and Ownable, Utils implementations.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.7;

import "@openzeppelin/contracts/token/ERC721/ERC721.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/utils/Strings.sol";
import "@openzeppelin/contracts/utils/Base64.sol";
import "./SparseMerkleTree.sol";
```

Imports

- Minting only available to owner and generate merkle tree

```
    function mint(
        address to,
        uint256 tokenId)
    public onlyOwner {
        _safeMint(to, tokenId);
        string memory uri = tokenURI(tokenId);
        bytes32 leafHash = keccak256(abi.encodePacked(msg.sender, to, uint8(tokenId),
uri));
        _updateLeaf(leafHash, uint8(tokenId));
        batchMerkleUpdate();
    }
```

minting function

- tokenURI is json data which includes name, description and simple svg image.

```
    function getSvg(uint256 tokenId) private pure returns (string memory) {
        string memory svg;
        svg = '<svg height="210" width="500"><polygon points="100,10 40,198 190,78 10,78
160,198" style="fill:lime;stroke:purple;stroke-width:5;fill-rule:nonzero;"/>';
        return string(abi.encodePacked(svg, tokenId, "</svg>"));
    }

    function tokenURI(uint256 tokenId) override(ERC721) public pure returns (string memory)
{
        string memory json = Base64.encode(
            bytes(string(
```

```
                abi.encodePacked(
                    '{"name": zkNFT-"', tokenId , '", ',
                    '"description": "Hello zero knowledge!",',
                    '"image": "', getSvg(tokenId), '"}'
                )
            ))
        );
        return string(abi.encodePacked('data:application/json;base64,', json));
    }
```

tokenURI function

Q 2.2

Commit the msg.sender, receiver address, tokenId, and tokenURI to a Merkle tree using the keccak256 hash function. Update the Merkle tree using a minimal amount of gas.

A 2.2

- I choose to use the Sparse Merkle Tree because this tree is order agnostic and can batch the tree generation with multiple leaves.
- I referenced the code base from [rugpullindexand](#) and [plasma-cash](#).

```
 //SPDX-License-Identifier: Unlicense
pragma solidity ^0.8.7;

contract SparseMerkleTree {
    uint8 constant DEPTH = 8;
    uint256 constant BUFFER_LENGTH = 1;

    mapping(uint8 => mapping(uint8 => bytes32)) public tree;
    uint8[] public pendingIndex;

    function pop(uint8[] storage array) internal returns (uint8){
        uint8 item = array[array.length-1];
        array.pop();
        return item;
    }

    function bitmap(uint256 index) internal pure returns (uint8) {
        uint8 bytePos = (uint8(BUFFER_LENGTH) - 1) - (uint8(index) / 8);
        return bytePos + 1 << (uint8(index) % 8);
    }

    function _updateLeaf(bytes32 leafHash, uint8 index) internal{
        tree[uint8(DEPTH-1)][index] = leafHash;
        pendingIndex.push(index);
    }

    function _batchMerkleUpdate() internal returns(bytes32) {
        for(uint i=pendingIndex.length; i > 0 ; i--) {
            uint8 currentIndex = pop(pendingIndex);
```

```
            for(uint8 j=DEPTH-1; j>0; j--) {
                uint8 siblingIndex = currentIndex % 2 == 0 ? currentIndex + 1 :
currentIndex -1;
                if (siblingIndex > currentIndex) {
                    tree[j-1][currentIndex/2] =
keccak256(abi.encodePacked(tree[j][currentIndex], tree[j][siblingIndex]));
                } else {
                    tree[j-1][currentIndex/2] =
keccak256(abi.encodePacked(tree[j][siblingIndex], tree[j][currentIndex]));
                }
                currentIndex /= 2;
            }
        }
        return tree[0][0];
    }

    function getMerkleProof(uint8 _index) public view returns(bytes memory proof) {
        uint8 proofBits = 0;
        bytes32[] memory proofHash;
        uint8 siblingIndex;
        bytes32 siblingHash;

        for (uint8 level=0; level < DEPTH; level++) {
            siblingIndex = (_index % 2) == 0 ? _index + 1 : _index - 1;
            _index = _index / 2;

            siblingHash = tree[level][siblingIndex];
            if (siblingHash != 0) {
                proofHash[proofHash.length] = siblingHash;
                proofBits += bitmap(level);
            }
        }

        bytes memory encoded = '';
        uint len = proofHash.length;
        for (uint i = 0; i < len; i++) {
            encoded = bytes.concat(
                encoded,
                abi.encodePacked(proofHash[i])
            );
        }

        proof = abi.encodePacked(proofBits, encoded);
        return proof;
    }

    function checkMembership(
        bytes32 leaf,
        bytes32 root,
        uint8 index,
        bytes memory proof) public pure returns (bool)
    {
        bytes32 computedHash = getRoot(leaf, index, proof);
```

```
        return (computedHash == root);
    }

    function getRoot(bytes32 leaf, uint8 index, bytes memory proof) public pure returns
(bytes32) {
        require((proof.length - 8) % 32 == 0 && proof.length <= 2056);
        bytes32 proofElement;
        bytes32 computedHash = leaf;
        uint16 p = 8;
        uint8 proofBits;
        assembly {proofBits := div(mload(add(proof, 32)), exp(256, 24))}

        for (uint d = 0; d < DEPTH; d++ ) {
            if (proofBits % 2 == 0) {
                proofElement = 0;
            } else {
                p += 32;
                require(proof.length >= p);
                assembly { proofElement := mload(add(proof, p)) }
            }
            if (index % 2 == 0) {
                computedHash = keccak256(abi.encodePacked(computedHash, proofElement));
            } else {
                computedHash = keccak256(abi.encodePacked(proofElement, computedHash));
            }
            proofBits = proofBits / 2;
            index = index / 2;
        }
        return computedHash;
    }
}
```

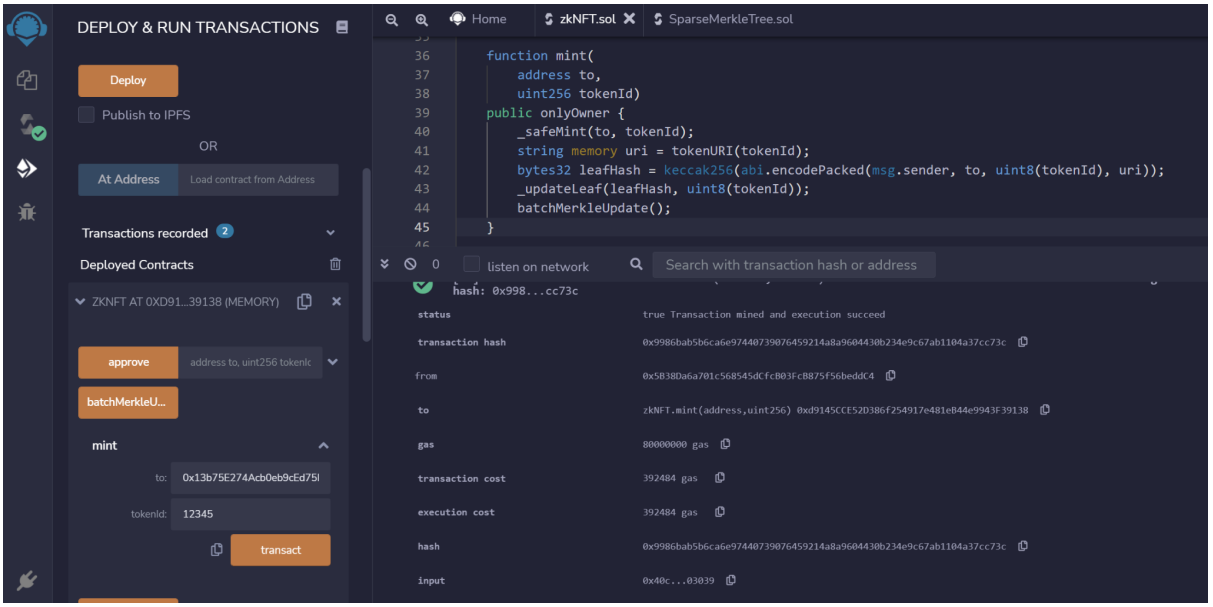
Sparse Merkle Tree

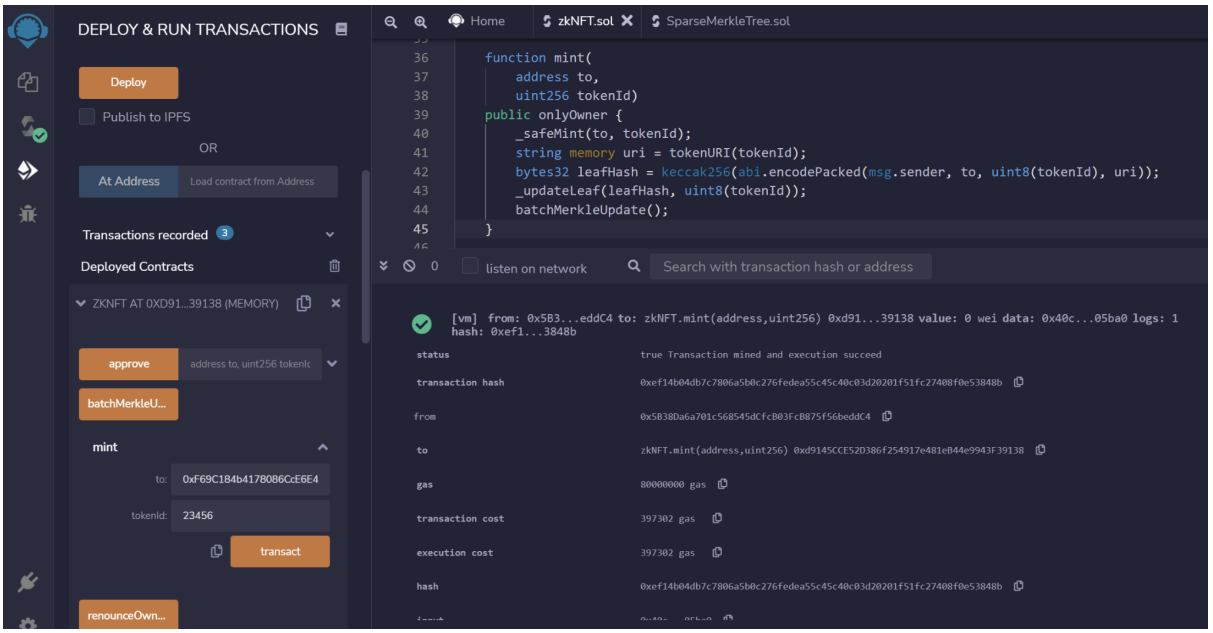
Q 2.3

Use remix to mint a couple of NFTs to the sender address or to other addresses. Include screenshots of the transactions and the amount of gas spent per transaction in your repo.

A 2.3

- mint gas cost is vary between 360000 ~ 400000





Question 3: Understanding and generating ideas about ZK technologies

Q 3.1
Summarize the key differences (in application, not in theory) between SNARKs and STARKs in 100 words.

A 3.1

- SNARKs is based on elliptic curves cryptography, on the contrary, STARKs relies on hash functions. As SNARKs uses ECC, a trusted setup is a required process to create keys that are used for proofs required for private data.
- STARKs is quantum-resistant, SNARKS isn't
- STARKs has no trusted setup, in turn, it needs relatively more proof size than SNARKS, also, more gas fee is required.
- SNARKs has been publicized earlier than STARKs. SNARKs has a more broadened community and knowledge base, adoption cases.

Q 3.2
How is the trusted setup process different between Groth16 and PLONK?

A 3.2

- Groth16 uses rank-1 constraint system(R1CS) format to express circuit-specific trusted setup
- Plonk uses two kinds of gates(Multiplications, Additions) to describe universal processing proving scheme with a preprocessing phase that can be updated, and has a short and constant verification time.

Q 3.3
Give an idea of how we can apply ZK to create unique usage for NFTs.

A 3.3

- Medical records can be tokenized and stored on chain without revealing sensitive private contents. In case the record owner is requested to submit validation of not having certain medical case history from the government or company for certain reasons, the user can facilitate this process without submitting actual records.