

Q1

circom code

detailed code is [here](#).

```
pragma circom 2.0.0;

include "mimcsponge.circom";

template Merkle8 () {

    signal input ins[8];
    signal output out;

    component ins_L1[4];

    for(var i = 0; i < 4; i++) {
        ins_L1[i] = MiMCSponge(2, 220, 1);
        ins_L1[i].ins[0] <-- ins[2*i];
        ins_L1[i].ins[1] <-- ins[2*i+1];
        ins_L1[i].k <== 0;
    }
    component ins_L2[2];
    for(var i = 0; i < 2; i++) {
        ins_L2[i] = MiMCSponge(2, 220, 1);
        ins_L2[i].ins[0] <-- ins_L1[2*i].outs[0];
        ins_L2[i].ins[1] <-- ins_L1[2*i+1].outs[0];
        ins_L2[i].k <== 0;
    }

    component top = MiMCSponge(2, 220, 1);
    top.ins[0] <-- ins_L2[0].outs[0];
    top.ins[1] <-- ins_L2[1].outs[0];
    top.k <== 0;

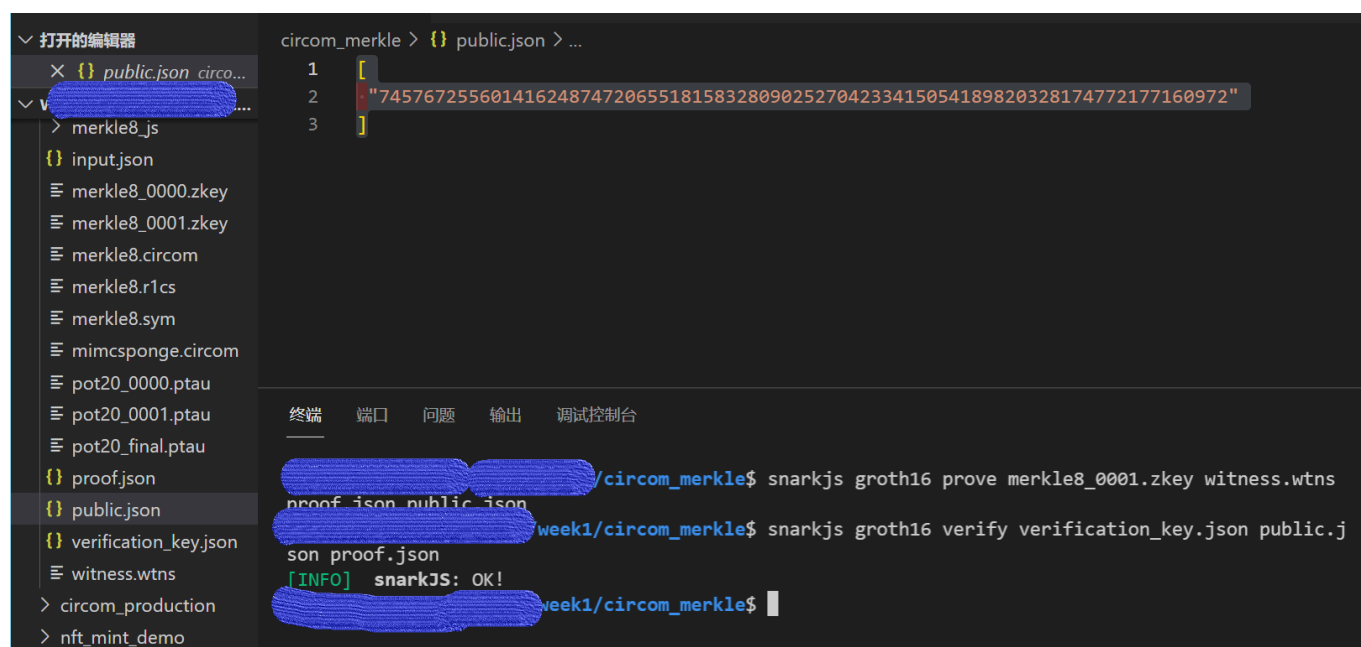
    out <== top.outs[0];
    // note: if circom support loop in loop, the above code can be Refactored to
    be general
}

// component: Instantiate a template.
component main = Merkle8();
```

public.json

```
[
  "7457672556014162487472065518158328090252704233415054189820328174772177160972"
]
```

the proof and verification of public.json is as follow:



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.

Answer: We do not really need zero-knowledge proof for this. Yes, a publicly verifiable smart contract that computes Merkle root achieve the same.

The scenario for Zero-Knowledge proofs: vote in Dao. Using Zero-Knowledge proofs to prove that someone voted, but don't know who voted.

The technologies implementing this type of proof is as follow:

- zksnarks
- zkSTARKs
- Ring signature

Q2

code

detailed code is [here](#).

```
// SPDX-License-Identifier: MIT
// pragma solidity >=0.8.0 <0.9.0;
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/token/ERC721/extensions/ERC721Enumerable.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/utils/Strings.sol";

contract ZKU_Celebration is ERC721Enumerable, Ownable {
    using Strings for uint256;
    // merkleRoot hash
    bytes32 public merkleRoot;
    // merkleTree leaves
    bytes32[] private leaves;

    // bytes32[] hashes;

    // record minted address
    mapping(address => bool) public alreadyMinted;

    mapping(uint256 => string) public tokenNames;

    mapping(uint256 => string) public tokenDescs;

    uint256 private reserveID;
    uint256 private currentSaleID;
    uint256 public constant maxID = 128;

    string private baseURI = "zkuc_";
    bool private saleStarted = true;

    constructor() ERC721("ZKU Celebration", "ZKUC") {
        reserveID = 1; // item 1-127
    }
}
```

```

        currentSaleID = 32; // item 128-1024
    }

    // override tokenURI for adding name and desc to tokenURI
    function tokenURI(uint256 tokenId) public view override returns (string
memory) {
        require(
            _exists(tokenId),
            "ERC721Metadata: URI query for nonexistent token"
        );

        return
            bytes(baseURI).length > 0
                ? string(
                    abi.encodePacked(
                        baseURI,
                        tokenId.toString(),
                        getTokenName(tokenId),
                        getTokenDesc(tokenId)
                    )
                )
                : "";
    }

    // Commit the msg.sender, receiver address, tokenId, and tokenURI to a Merkle
tree using the keccak256 hash function
    function updateRoot(
        address sender,
        address receiver,
        uint256 tokenId
    ) private {
        bytes32[] memory hashes = new bytes32[](128);

        // get tokenURI by tokenId
        string memory _tokenURI = tokenURI(tokenId);
        // compute leaveHash
        bytes32 leaveHash = keccak256(
            abi.encodePacked(sender, receiver, tokenId, _tokenURI)
        );
        // push leaveHash to merkle tree leaves
        leaves.push(leaveHash);

        for (uint256 i = 0; i < leaves.length; i++) {
            hashes[i] = leaves[i];
        }
        // update merkle tree
        uint256 n = leaves.length;
        while(n > 1) {
            uint256 i = 0;
            uint256 j = 0;
            if (n % 2 == 0) {
                for(; i <= n-2 ; i += 2 ){

```

```

        hashes[j] = keccak256(abi.encodePacked(hashes[i],hashes[i +
1]));
        j++;
    }
    n = n / 2;
} else {
    for(; i <= n-3 ; i += 2 ){
        hashes[j] = keccak256(abi.encodePacked(hashes[i],hashes[i +
1]));
        j++;
    }
    n = n / 2 +1;
}

}

merkleRoot = hashes[0];
delete hashes;
}

function mint(address receiver) public returns (uint256) {
    require(saleStarted == true, "The sale is paused");
    require(msg.sender != address(0x0), "Public address is not correct");
    require(alreadyMinted[msg.sender] == false, "Address already used");
    require(currentSaleID <= maxID, "Mint limit reached");
    uint256 tokenID = currentSaleID;
    _safeMint(receiver, tokenID);
    alreadyMinted[msg.sender] = true;
    updateRoot(msg.sender, receiver, tokenID);

    currentSaleID++;
    return tokenID;
}

function getTokenName(uint256 tokenID) public view returns (string memory) {
    return tokenNames[tokenID];
}

function setTokenName(uint256 tokenID, string memory name) public onlyOwner {

    tokenNames[tokenID] = name;
}

function getTokenDesc(uint256 tokenID) public view returns (string memory) {
    return tokenDescs[tokenID];
}

function setTokenDesc(uint256 tokenID, string memory desc) public onlyOwner {
    tokenDescs[tokenID] = desc;
}

function startSale() public onlyOwner {
    saleStarted = true;

```

```

    }

    function pauseSale() public onlyOwner {
        saleStarted = false;
    }
}

```

MINT-1

The screenshot displays the Remix IDE interface. On the left, the 'DEPLOY & RUN TRANSACTIONS' panel shows the 'mint' function selected under the 'ZKU_Celebration' contract. The main editor shows the Solidity code for the 'ZKU_Celebration' contract, which inherits from 'ERC721Enumerable' and 'Ownable'. The contract includes variables for 'merkleRoot', 'merkleTree', and 'leaves', and a 'mapping' for 'alreadyMinted'. The right sidebar shows the transaction details for the 'mint' function call, including the gas cost, execution cost, hash, input, decoded input, and decoded output.

MINT-2

DEPLOY & RUN TRANSACTIONS

GAS LIMIT: 3000000

VALUE: 0 Wei

CONTRACT: ZKU_Celebration - contracts/4_ZKU_

Deploy

☐ Publish to IPFS

OR

At Address Load contract from Address

Transactions recorded 8

Deployed Contracts

ZKU_CELEBRATION AT 0xF8E...9FBE

approve address to, uint256 tokenk

mint 0xAb8483F64d9C6d1EcP

pauseSale

renounceOwn...

safeTransferFr... address from, address to, u

```

4
5 import "@openzeppelin/contracts/token/ERC721/extensions/ERC721Enumerable.sol";
6 import "@openzeppelin/contracts/access/Ownable.sol";
7 import "@openzeppelin/contracts/utils/Strings.sol";
8
9 contract ZKU_Celebration is ERC721Enumerable, Ownable {
10     using Strings for uint256;
11     // merkleRoot hash
12     bytes32 public merkleRoot;
13     // merkleTree leaves
14     bytes32[] private leaves;
15
16     // bytes32[] hashes;
17
18     // record minted address
19     mapping(address => bool) public alreadyMinted;
20

```

to ZKU_Celebration.mint(address) 0xf8e81d47203a594245e36c48e151709f0c19f8e8

gas 80000000 gas

transaction cost 226695 gas

execution cost 226695 gas

hash 0x250f8526d8062c7f426b4c4ba9425409aea8aab9e02eabc8daf25d48cc99b9e

input 0x6a6...35cb2

decoded input { "address receiver": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2" }

decoded output { "0": "uint256: 33" }

MINT-3

DEPLOY & RUN TRANSACTIONS

VALUE: 0 Wei

CONTRACT: ZKU_Celebration - contracts/4_ZKU_

Deploy

☐ Publish to IPFS

OR

At Address Load contract from Address

Transactions recorded 9

Deployed Contracts

ZKU_CELEBRATION AT 0xF8E...9FBE

approve address to, uint256 tokenk

mint 0x4B20993Bc481177ec7

pauseSale

renounceOwn...

safeTransferFr... address from, address to, u

safeTransferFr... address from, address to, u

```

89     n = n / 2;
90 }else {
91     for(; i <= n-3 ; i += 2 ){
92         hashes[j] = keccak256(abi.encodePacked(hashes[i],hashes[i + 1]));
93         j++;
94     }
95     n = n / 2 + 1;
96 }
97
98 }
99
100
101 merkleRoot = hashes[0];
102 delete hashes;
103 }
104
105 function mint(address receiver) public returns (uint256) {

```

to ZKU_Celebration.mint(address) 0xf8e81d47203a594245e36c48e151709f0c19f8e8

gas 80000000 gas

transaction cost 231560 gas

execution cost 231560 gas

hash 0x845abf7bb2d60350f5380e05ca83e772283f10319c9fd17f82769f7a0061663c

input 0x6a6...c02db

decoded input { "address receiver": "0x4B20993Bc481177ec7E8F571ceCaE8A9e22C02db" }

decoded output { "0": "uint256: 34" }

MINT-4

The screenshot shows the Remix IDE interface. On the left, the 'DEPLOY & RUN TRANSACTIONS' panel is active. The 'VALUE' is set to 0 Wei. The 'CONTRACT' is 'ZKU_Celebration - contracts/4_ZKU_C'. The 'Deploy' button is highlighted. Below it, the 'Transactions recorded' section shows 10 transactions. The 'Deployed Contracts' section shows a list of contracts, including 'ZKU_CELEBRATION AT 0xF8E...9FBE'. The 'mint' button is highlighted, and the transaction details are shown below it.

The transaction details for the 'mint' operation are as follows:

- to: ZKU_Celebration.mint(address) 0xf8e81D47203A594245E36C48e151709F0C19f8e8
- gas: 8000000 gas
- transaction cost: 235392 gas
- execution cost: 235392 gas
- hash: 0xa7f3bf703e0d9c1436720eb8a5825c78eccc51473a577e5bf4a608ab245c1fe
- input: 0x6a6...cabab
- decoded input: { "address receiver": "0x78731D3Ca6b7E34aC0F824c42a7cC18A495cabab" }
- decoded output: { "0": "uint256: 35" }

The main editor shows the Solidity code for the 'mint' function:

```

102     delete hashes;
103 }
104
105 function mint(address receiver) public returns (uint256) {
106     require(saleStarted == true, "The sale is paused");
107     require(msg.sender != address(0x0), "Public address is not correct");
108     require(alreadyMinted[msg.sender] == false, "Address already used");
109     require(currentSaleID <= maxID, "Mint limit reached");
110     uint256 tokenId = currentSaleID;
111     _safeMint(receiver, tokenId);
112     alreadyMinted[msg.sender] = true;
113     updateRoot(msg.sender, receiver, tokenId);
114
115     currentSaleID++;
116     return tokenId;
117 }
118

```

MINT-8

The screenshot shows the Remix IDE interface. On the left, the 'DEPLOY & RUN TRANSACTIONS' panel is active. The 'ACCOUNT' is '0x03C...D1Ff7 (99.999999%)'. The 'GAS LIMIT' is set to 3000000. The 'VALUE' is set to 0 Wei. The 'CONTRACT' is 'ZKU_Celebration - contracts/4_ZKU_C'. The 'Deploy' button is highlighted. Below it, the 'Transactions recorded' section shows 14 transactions. The 'Deployed Contracts' section shows a list of contracts, including 'ZKU_CELEBRATION AT 0xF8E...9FBE'. The 'mint' button is highlighted, and the transaction details are shown below it.

The transaction details for the 'mint' operation are as follows:

- to: ZKU_Celebration.mint(address) 0xf8e81D47203A594245E36C48e151709F0C19f8e8
- gas: 8000000 gas
- transaction cost: 252122 gas
- execution cost: 252122 gas
- hash: 0x834ca82e041e0235bcd3256ee7b6d393b765092a5af5c594ecc24f5dab910fa
- input: 0x6a6...d1ff7
- decoded input: { "address receiver": "0x03C6FcED478cBbC9a4FAB34eF9f40767739D1Ff7" }
- decoded output: { "0": "uint256: 39" }

The main editor shows the Solidity code for the 'mint' function:

```

102     delete hashes;
103 }
104
105 function mint(address receiver) public returns (uint256) {
106     require(saleStarted == true, "The sale is paused");
107     require(msg.sender != address(0x0), "Public address is not correct");
108     require(alreadyMinted[msg.sender] == false, "Address already used");
109     require(currentSaleID <= maxID, "Mint limit reached");
110     uint256 tokenId = currentSaleID;
111     _safeMint(receiver, tokenId);
112     alreadyMinted[msg.sender] = true;
113     updateRoot(msg.sender, receiver, tokenId);
114
115     currentSaleID++;
116     return tokenId;
117 }
118

```

Q3

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

Answer:

the key differences between SNARKs and STARKs :

- SNARKs require a trusted setup, but STARKs do not.
- STARKs is more easy to scale than SNARKs in application
- SNARKs are not post-quantum secure, but STARKs are post-quantum secure

2. How is the trusted setup process different between Groth16 and PLONK?

Answer:

PLONK's trusted setup is universal setup. This means two things: first, instead of there being one separate trusted setup for every program you want to prove things about, there is one single trusted setup for the whole scheme after which you can use the scheme with any program. This means two things: first, instead of there being one separate trusted setup for every program you want to prove things about, there is one single trusted setup for the whole scheme after which you can use the scheme with any program

Groth16's trusted setup is non-universal setup. This means two things: first, you need separate trusted setup for every program, second, the trusted setup can not be updateable, when the program change, you need to repeat the trusted setup process.

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

Answer:

apply ZK for NFTs: auction and pricing (making NFTs's price more reasonable and improving liquidity for NFTs).

4. Give a novel idea on how we can apply ZK for Dao Tooling. (Yes, we know voting is a very popular one, but what else can ZK do?)

Answer: apply ZK for Dao Governance like [immutable x](#) to reduce the gas price and improve use experience.

reference

[On-boarding 10,000 Developers for ZK Products](#)

[Assignment 1](#)