Q1

circom code

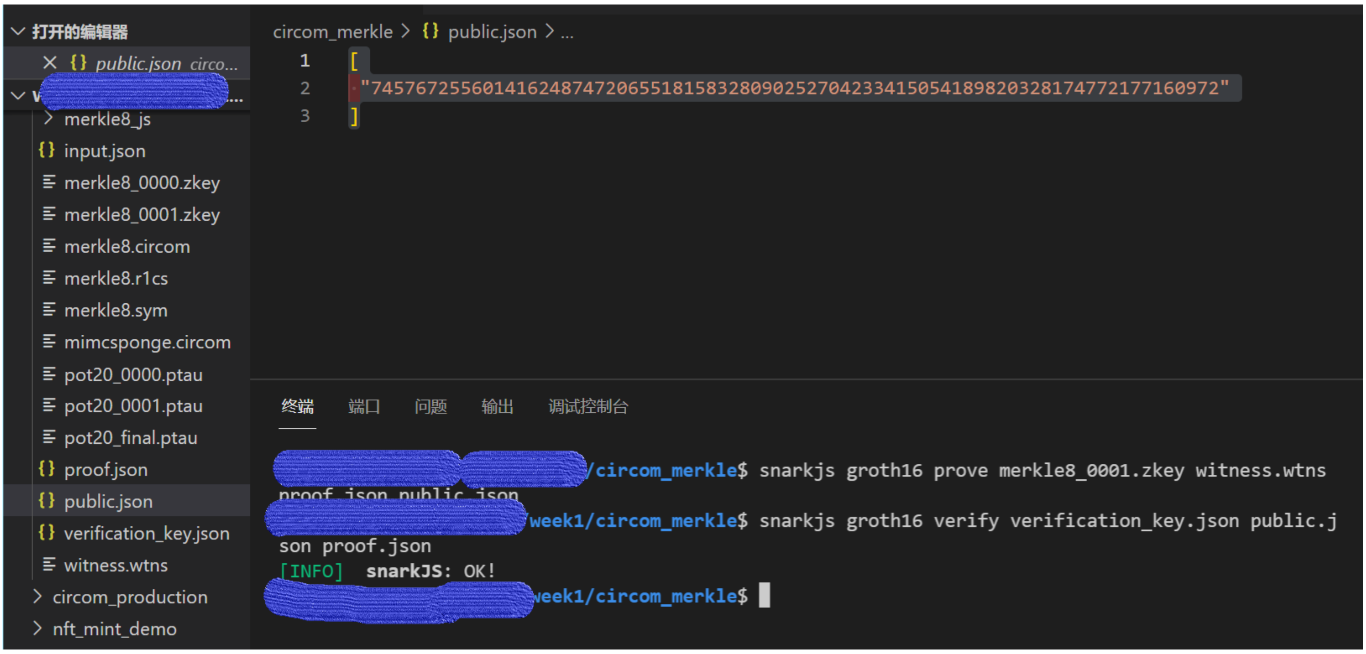
detailed code is here.

```
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.

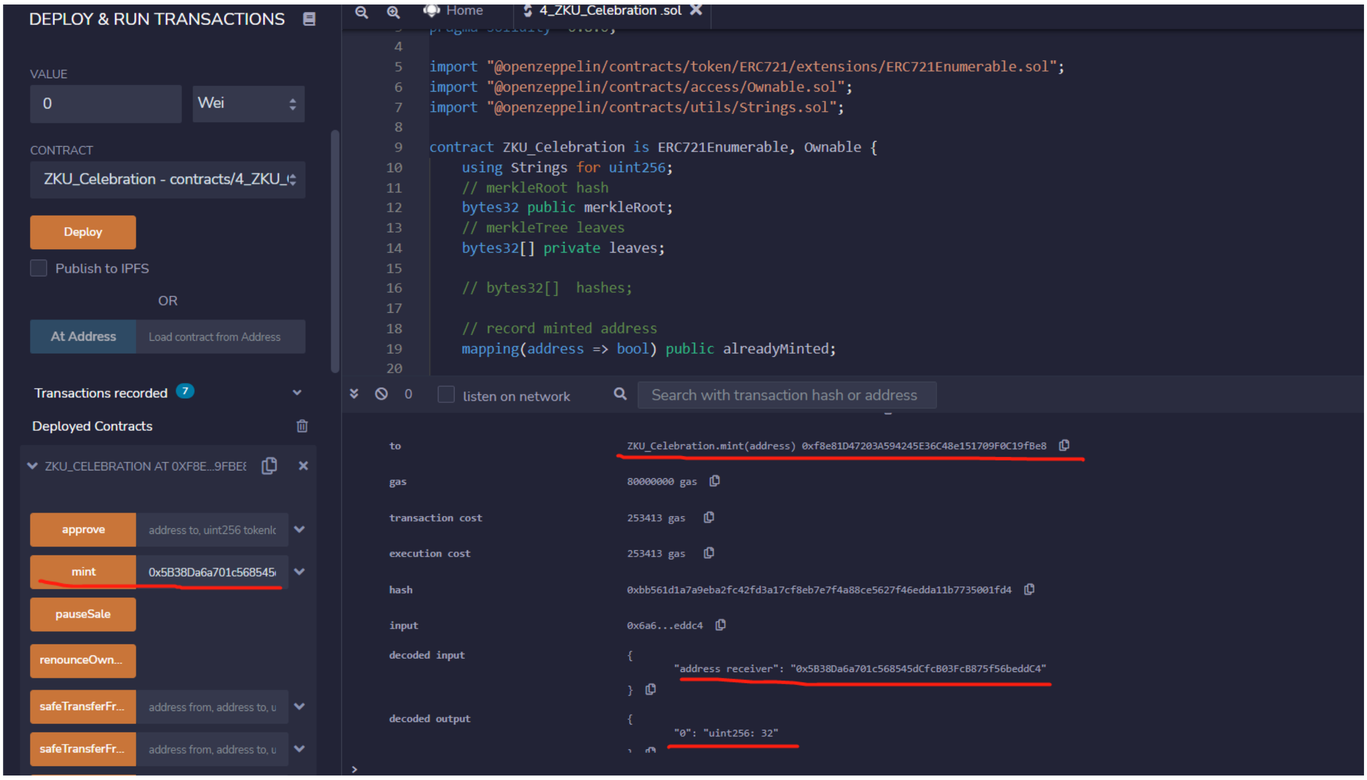
```
// SPDX-License-Identifier: MIT
// pragma solidity >=0.8.0 <0.9.0;
pragma solidity ^0.8.0;

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 merkletree 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 ){
```

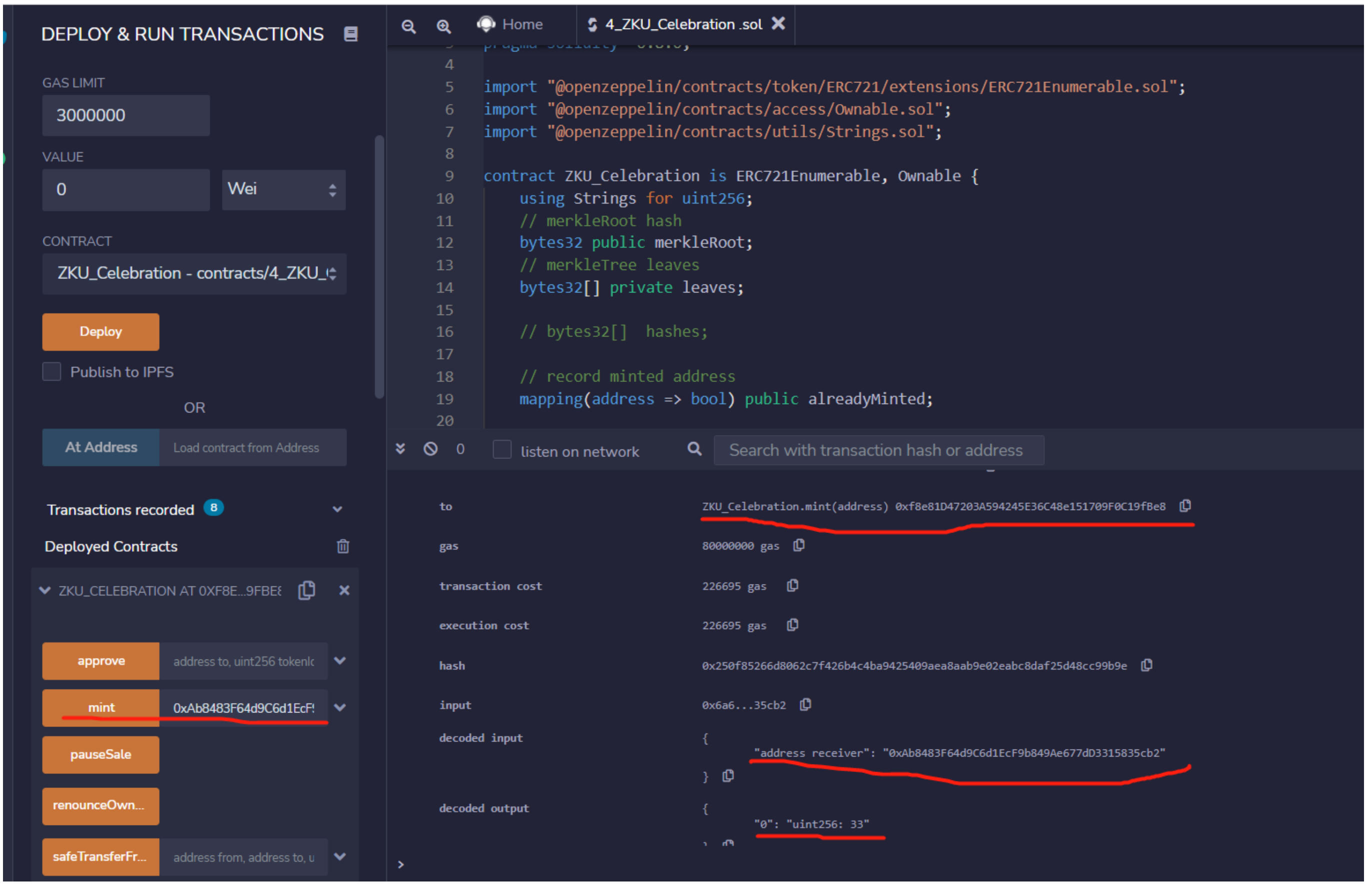
```
                    hashes[j] = keccak256(abi.encodePacked(hashes[i],hashes[i +
1]));
                    j++;
                }
                n = n / 2;
            }else {
                 for(; i <= n-3 ;  i += 2 ){
                    hashes[j] = keccak256(abi.encodePacked(hashes[i],hashes[i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

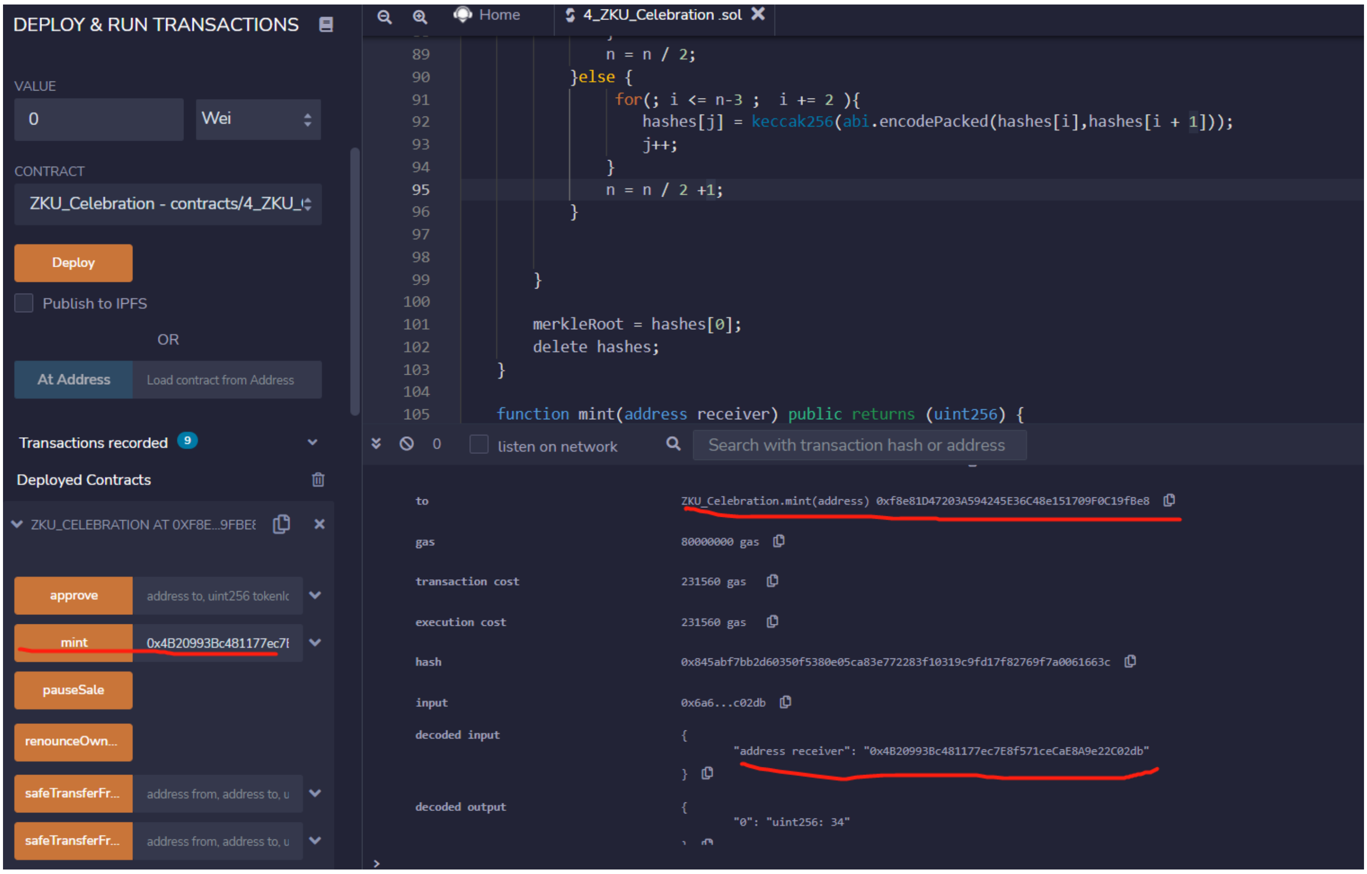
MINT-1



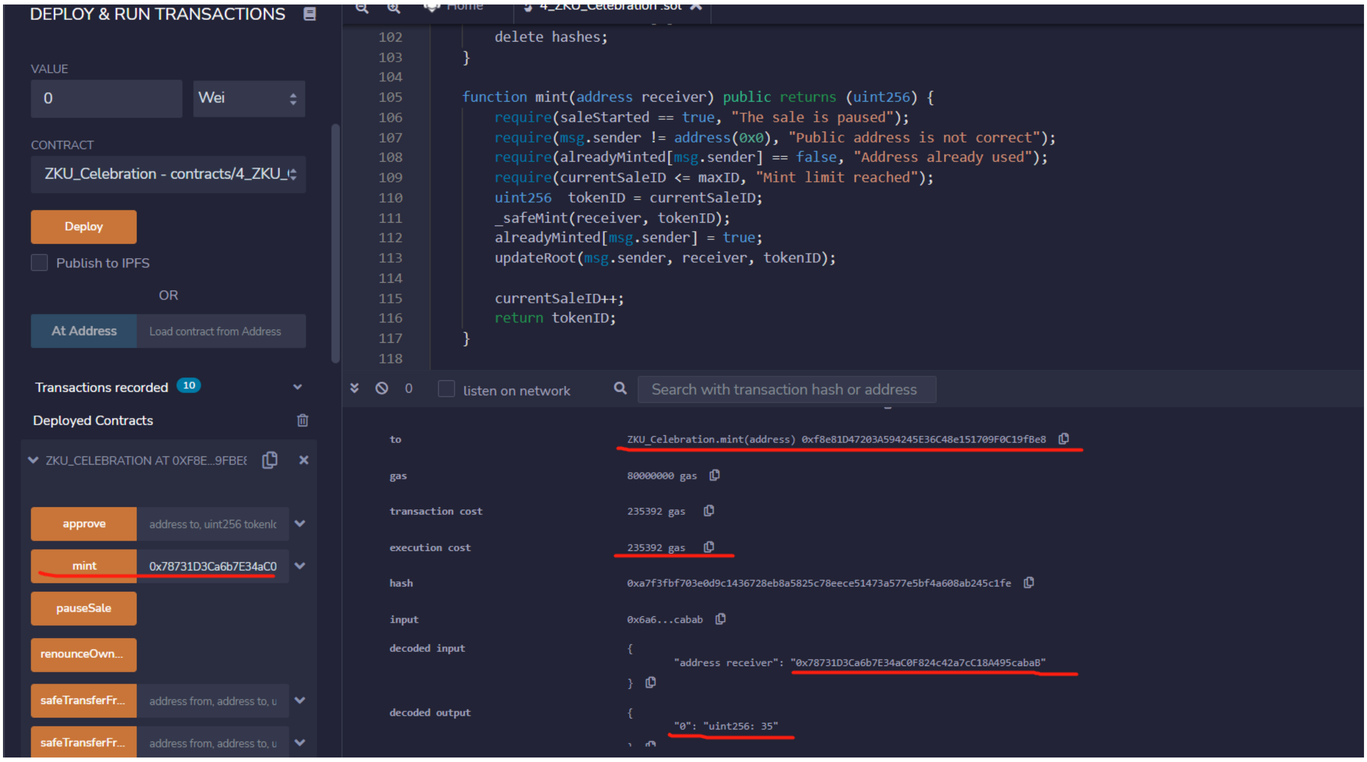
MINT-2



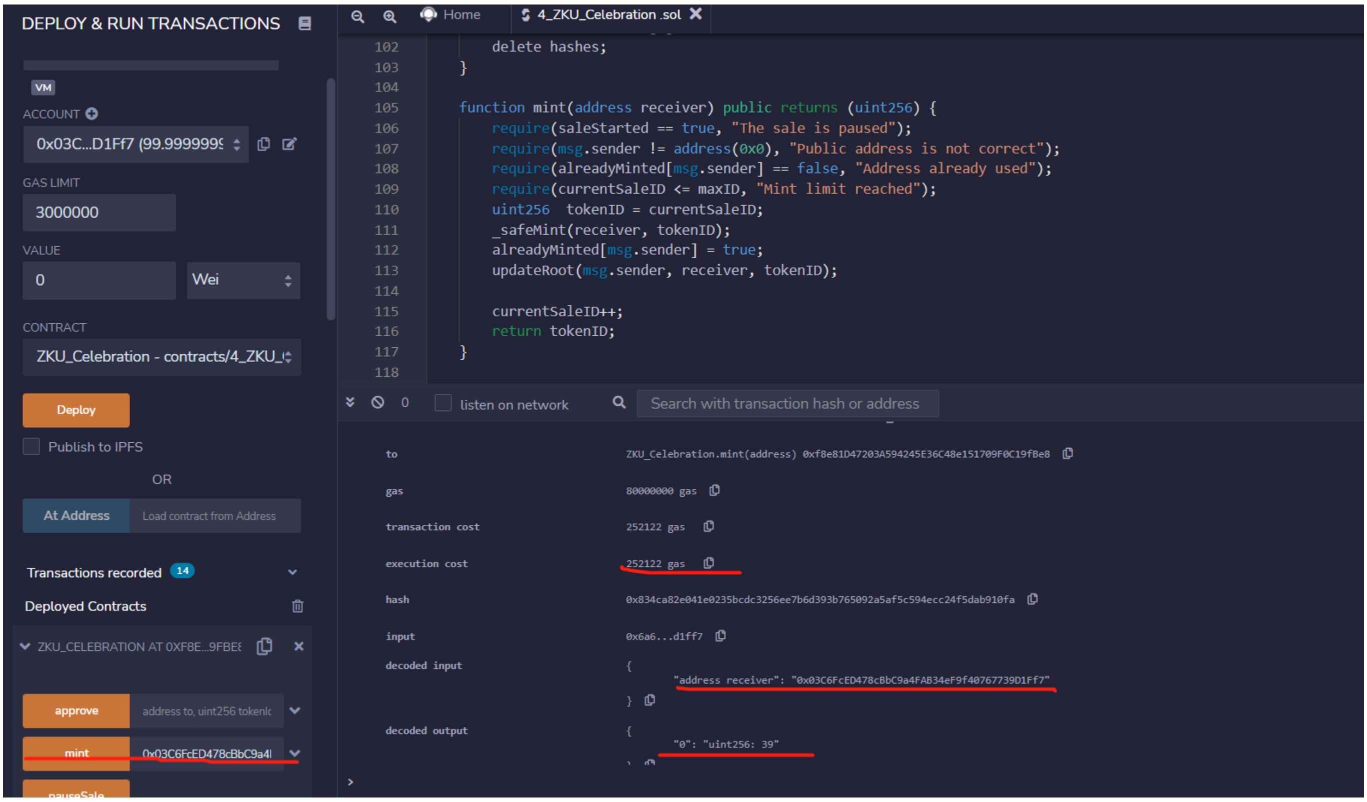
MINT-3



MINT-4



MINT-8



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: frist, you need separate trusted setup for every program, second, the trusted setup can not be updateable, when the program change, you need to repeate 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