

Zero Knowledge University March-April 2022 Cohort

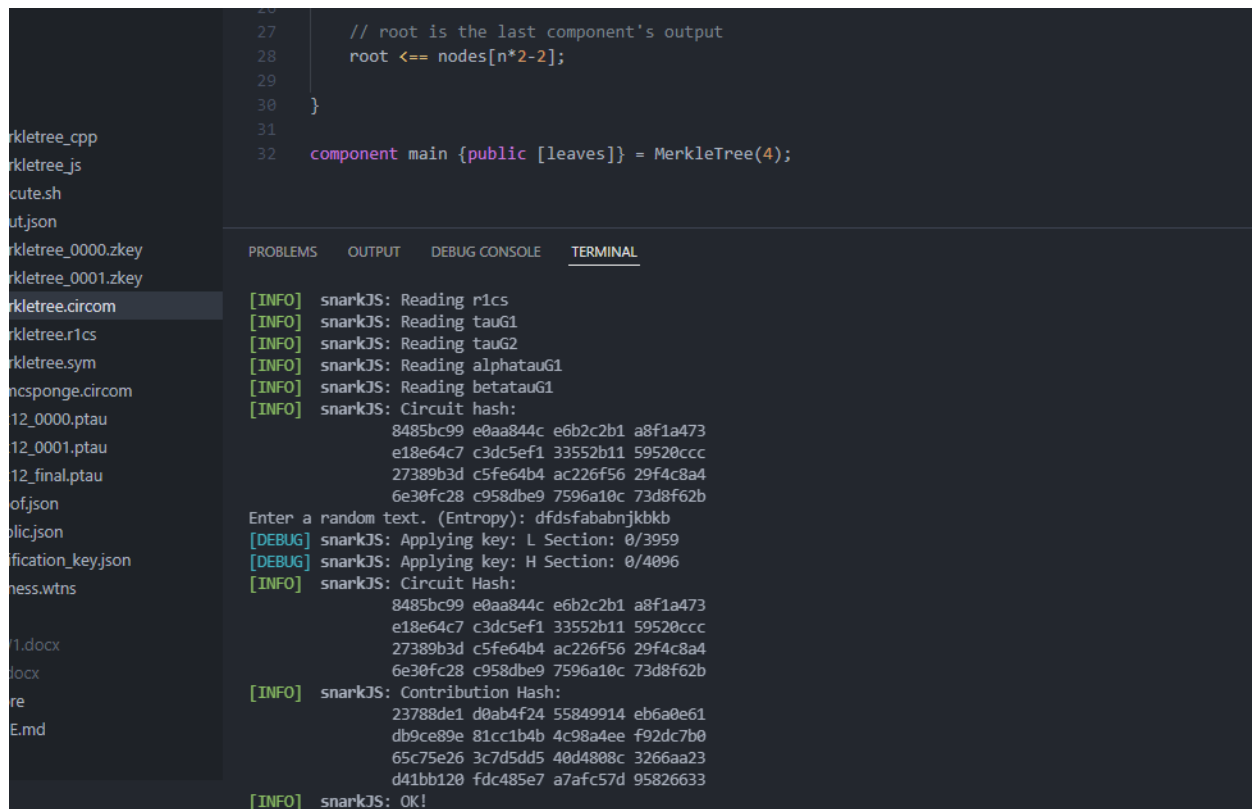
Week 1 Assignment

Course Registration Email: zhang.jeremy.2001@gmail.com

Discord Username: xyz5368#1102

1. Intro to Circom

- a. See https://github.com/jeremyzhang1/zku-submissions/blob/main/week1/hw/q1_1/merkletree.circom. Here is a screenshot of the successful execution.



The screenshot shows a code editor with a file explorer on the left and a terminal window on the right. The file explorer lists several files, with `merkletree.circom` selected. The code editor displays the following Circom code:

```
26
27 // root is the last component's output
28 root <== nodes[n*2-2];
29
30 }
31
32 component main {public [leaves]} = MerkleTree(4);
```

The terminal window shows the output of the circuit execution:

```
[INFO] snarkJS: Reading r1cs
[INFO] snarkJS: Reading tauG1
[INFO] snarkJS: Reading tauG2
[INFO] snarkJS: Reading alphatauG1
[INFO] snarkJS: Reading betatauG1
[INFO] snarkJS: Circuit hash:
      8485bc99 e0aa844c e6b2c2b1 a8f1a473
      e18e64c7 c3dc5ef1 33552b11 59520ccc
      27389b3d c5fe64b4 ac226f56 29f4c8a4
      6e30fc28 c958dbe9 7596a10c 73d8f62b
Enter a random text. (Entropy): dfdsfababnjkbkb
[DEBUG] snarkJS: Applying key: L Section: 0/3959
[DEBUG] snarkJS: Applying key: H Section: 0/4096
[INFO] snarkJS: Circuit Hash:
      8485bc99 e0aa844c e6b2c2b1 a8f1a473
      e18e64c7 c3dc5ef1 33552b11 59520ccc
      27389b3d c5fe64b4 ac226f56 29f4c8a4
      6e30fc28 c958dbe9 7596a10c 73d8f62b
[INFO] snarkJS: Contribution Hash:
      23788de1 d0ab4f24 55849914 eb6a0e61
      db9ce89e 81cc1b4b 4c98a4ee f92dc7b0
      65c75e26 3c7d5dd5 40d4808c 3266aa23
      d41bb120 fdc485e7 a7afc57d 95826633
[INFO] snarkJS: OK!
```

- b. See https://github.com/jeremyzhang1/zku-submissions/blob/main/week1/hw/q1_2/merkletree.circom. While running the code, I ran into an error stating that the circuit was too big for the power of tau ceremony. See picture below:

```

> q_2
> q1_1
> q1_2
  execute.sh
  input.json
  merkleTree.circom
  mimicsponge.circom
  ~$HW1.docx
  HW1.docx
  .gitignore
  README.md
28     root <== nodes[n*2-2];
29
30   }
31
32   component main {public [leaves]} = MerkleTree(8);

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```

[DEBUG] snarkJS: betaTauG1: fft 12 join 11/12 1/2 0/2
[DEBUG] snarkJS: betaTauG1: fft 12 join 11/12 2/2 1/2
[DEBUG] snarkJS: betaTauG1: fft 12 join 11/12 2/2 0/2
[DEBUG] snarkJS: betaTauG1: fft 12 join: 12/12
[DEBUG] snarkJS: betaTauG1: fft 12 join 12/12 1/1 0/4
[DEBUG] snarkJS: betaTauG1: fft 12 join 12/12 1/1 2/4
[DEBUG] snarkJS: betaTauG1: fft 12 join 12/12 1/1 3/4
[DEBUG] snarkJS: betaTauG1: fft 12 join 12/12 1/1 1/4
[ERROR] snarkJS: circuit too big for this power of tau ceremony. 9248*2 > 2**12
[ERROR] snarkJS: Error: merkleTree_0000.zkey: Invalid File format
    at Object.readFile (/usr/local/lib/node_modules/snarkjs/node_modules/@iden3/binfileutils/build/main.cjs:36:35)
    at async phase2contribute (/usr/local/lib/node_modules/snarkjs/build/cli.cjs:4874:45)
    at async cliProcessor (/usr/local/lib/node_modules/snarkjs/build/cli.cjs:300:21)

```

To fix this problem, I increased the powers of tau from 12 to 14, which solved the problem. Here is a screenshot of the successful execution.

```

24     nodes[i+n] = hash[1].outs[0];
25   }
26
27   // root is the last component's output
28   root <== nodes[n*2-2];
29
30 }
31
32 component main {public [leaves]} = MerkleTree(8);

```

tree.cpp
tree.js
e.sh
son
tree_0000.zkey
tree_0001.zkey
tree.circom
tree.r1cs
tree.sym
ponge.circom
0000.ptau
0001.ptau
final.ptau
0000.ptau
0001.ptau
final.ptau
son
son
tion_key.json
s.wtns
ock
k

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```

[INFO] snarkJS: Reading r1cs
[INFO] snarkJS: Reading tauG1
[INFO] snarkJS: Reading tauG2
[INFO] snarkJS: Reading alphaTauG1
[INFO] snarkJS: Reading betaTauG1
[INFO] snarkJS: Circuit hash:
600fd732 44a8ff3c 89f0c31e 9a44f8c6
894bbd16 507668fb 78bdb49f e3d07ae1
6e731c80 a1e329de 6460b02d 01473c96
817c9316 0a28c767 9deaf53a eb3d9d13
Enter a random text. (Entropy): lkjlkj
[DEBUG] snarkJS: Applying key: L Section: 0/9239
[DEBUG] snarkJS: Applying key: H Section: 0/16384
[INFO] snarkJS: Circuit Hash:
600fd732 44a8ff3c 89f0c31e 9a44f8c6
894bbd16 507668fb 78bdb49f e3d07ae1
6e731c80 a1e329de 6460b02d 01473c96
817c9316 0a28c767 9deaf53a eb3d9d13
[INFO] snarkJS: Contribution Hash:
3c4f4fff ab193389 975eb886 d7063268
23e3f0ed f541b577 9677c114 8edbd59a
696bcb68 2e211b21 fbdded427 468cbfe9
e29b52ee 8a404f28 8004a4e7 c6d629dd
[INFO] snarkJS: OK!

```

- c. Zero knowledge proofs are not really necessary for all of this. Creating a Merkle tree on chain like the example provided in the solidity by example documentation could work as well. When the user commits some information, only the hash values would be stored, so it would still be completely anonymous. We would then give the root hash and perhaps a few intermediate hashes to let the user know that their transaction was successful.
- d. See https://github.com/jeremyzhang1/zku-submissions/blob/main/week1/hw/q1_2/execute.sh
2. Minting an NFT and committing the mint data to a Merkle Tree
 - a. See <https://github.com/jeremyzhang1/zku-submissions/blob/main/week1/hw/q2/NFT.sol>
 - b. See <https://github.com/jeremyzhang1/zku-submissions/blob/main/week1/hw/q2/MerkleTree.sol>

- c. The first screenshot is an NFT mint to the contract owner, the second screenshot is an NFT mint to another user.

The screenshot shows a web interface on the left and a transaction details view on the right. The left sidebar includes sections for 'Transactions recorded', 'Deployed Contracts', and 'Low level interactions'. The 'NFT AT 0x0B...33f8 (MEMORY)' section is active, showing a 'mint' transaction. The right pane displays the transaction details for a successful mint operation. The transaction hash is 0x35f7be3e72ad6b6ad734f59839e4db747a32cd525fedc8a74ac89d. The 'from' address is 0x5830da781c568545dcf8b3fc8b75f5b6ddc4. The 'to' address is the contract address 0xb0934588fc35a11858c073ab6468a2833f8. The transaction cost is 16968 gas. The decoded input shows the 'address receiver' as the contract address, 'name' as 'owner', and 'desc' as 'owner_desc'.

```

4 import "@openzeppelin/contracts@4.5.0/token/ERC721/ERC721.sol";
5 import "@openzeppelin/contracts@4.5.0/token/ERC721/extensions/ERC721URIStorage.sol";
6 import "@openzeppelin/contracts@4.5.0/utils/Counters.sol";
7
8 import "./MerkleTree.sol";
9
10 contract NFT is ERC721, ERC721URIStorage, MerkleTree {
11     using Counters for Counters.Counter;
12     Counters.Counter private _tokenId;
13
14     mapping(uint256 => bytes) private _tokenURIs;
15
16     // constructor
17     constructor(string memory name, string memory symbol, uint32 depth)
18     ERC721(name, symbol) MerkleTree(depth) {}
19
20     // keep track of token metadata
21     function _setTokenURI(uint256 tid, bytes memory uri) internal {
22         _tokenURIs[tid] = uri;
23     }
24
25     // mint nft

```

The screenshot shows a web interface on the left and a transaction details view on the right. The left sidebar is similar to the first screenshot, but the 'mint' transaction is now to a different address. The right pane displays the transaction details for a successful mint operation. The transaction hash is 0xdc348bef48b122c31c899db5db53ae3495c34c8aa701edc218e3c3effa8dc3. The 'from' address is 0x5830da781c568545dcf8b3fc8b75f5b6ddc4. The 'to' address is the contract address 0xb0934588fc35a11858c073ab6468a2833f8. The transaction cost is 135464 gas. The decoded input shows the 'address receiver' as the contract address, 'name' as 'user', and 'desc' as 'user_desc'. The logs section at the bottom shows the event 'Transfer' with the 'from' address, 'to' address, and 'tokenId'.

```

13 mapping(uint256 => bytes) private _tokenURIs;
14
15 // constructor
16 constructor(string memory name, string memory symbol, uint32 depth)
17 ERC721(name, symbol) MerkleTree(depth) {}
18
19 // keep track of token metadata
20 function _setTokenURI(uint256 tid, bytes memory uri) internal {
21     _tokenURIs[tid] = uri;
22 }
23
24 // mint nft

```

- d. Minimal frontend bonus not attempted.
- e. Extra bonus not attempted.
3. Understanding and generating ideas about ZK technologies
- a. SNARK relies on elliptic curves, which can be solved with a quantum computer whereas STARK uses hashing algorithms that are more secure. SNARK needs a trusted setup ritual whereas STARK does not. SNARK has a smaller proof size and thus takes less time to verify than STARK.

- b. Groth16 is not a universally trusted setup whereas PLONK is. This essentially means that every time the circom circuit file is modified, you need to go through the trusted setup process again with Groth16.
- c. You can anonymize NFT transactions, so that owners of NFTs would be able to prove that they are the owner without revealing who they actually are. You could also prevent the resale of NFTs by providing some sort of information or original certificate of authenticity. When the NFT is resold, this information is not passed along, and since the new owner does not have that information, they cannot verify into the Merkle tree to prove ownership.
- d. In addition to voting, ZK can be used to verify identity without members of the DAO having to break the anonymity barrier. DAOs would be able to verify the identity or membership status of the member without knowing who that member is.