Assignment 1

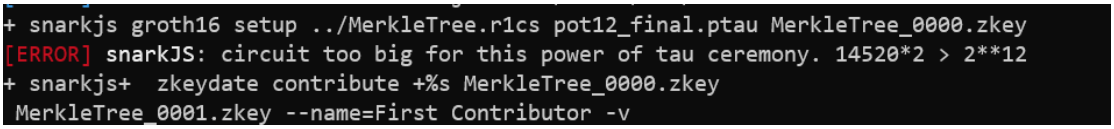
1. Question 1: Intro to circom

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]}

   [Answer] code**:**
   https://github.com/geesimon/zku/blob/main/week1/MerkleTree.circom

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.

   [Answer] Got error when trying to use snarkjs to start a powers of tau ceremony (see below). Apparently, parameter 12 (power of two of the maximum number of constraints that the ceremony can accept) is too small for generating the circuits (with 8 inputs). Set the parameter to 16 fixed this error.



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.

   [Answer] circom implementation for this case seems an overkill and is too expensive. It took 15 minutes on my desktop (i7 CPU) to build the whole manifest.

   If the purpose is to hide input but still be able to verify data integrity. I believe a public smart contract can do the same by leveraging merkle proof technique.

Say we need a way to log all sensitive transactions on chain but don't want to make the transaction data public. We can:

1. Build a merkle tree and store the tree on chain. Note: we only store the hash values of these transactions, thus no one can decode the transactions by looking at these hash values.
2. Once a user submit a transaction and we logged the transaction to the tree, we can give him the root hash and intermediate/proof hashes so that he can verify the transaction is logged (and we can't cheat him). To do so, he first hashes the transaction data locally, then use the hashes for transaction data, root and intermediate to do the verification.

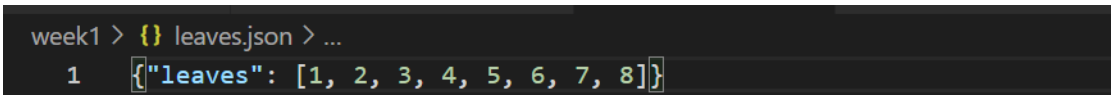
This approach is very efficient and fast for this case. It only needs to compute and transfer a few hashes. It also doesn't require to do ZKP trust setup.

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 to quickly compile circuits.

   [Answer] Wrote a shell script to compile, generate witness and prove files for a circom circuits file using Groth16 zk-SNARK protocol. Please check my script (and comments inline):
   https://github.com/geesimon/zku/blob/main/week1/build.sh

Add screenshots of the execution and the generated public.json file.

- The input file

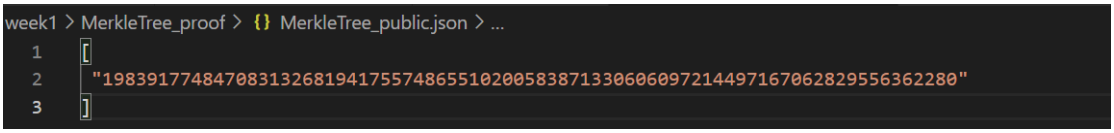


- Execution of shell.sh

```
+ snarkjs groth16 setup ../MerkleTree.r1cs pot16_final.ptau MerkleTree_0000.zkey
[INFO]  snarkJS: Reading r1cs
[INFO]  snarkJS: Reading tauG1
[INFO]  snarkJS: Reading tauG2
[INFO]  snarkJS: Reading alphatauG1
[INFO]  snarkJS: Reading betatauG1
[INFO]  snarkJS: Circuit hash:
                4a2c1dc6 a6eb3c8d edc200f9 18a487c5
                01acb076 1127c16b 46caf030 c07d5721
                51d36f98 f1fbea3d b9df339c 7c6363cc
                df1f4824 ac43c272 5ea0c31e 3ae2b9e0
+ snarkjs zkey contribute MerkleTree_0000.zkey MerkleTree_0001.zkey+  --name=First Contributor -v
date +%s
+ printf 1646570205\n
Enter a random text. (Entropy): 1646570205
[DEBUG] snarkJS: Applying key: L Section: 0/14527
[DEBUG] snarkJS: Applying key: H Section: 0/16384
[INFO]  snarkJS: Circuit Hash:
                4a2c1dc6 a6eb3c8d edc200f9 18a487c5
                01acb076 1127c16b 46caf030 c07d5721
                51d36f98 f1fbea3d b9df339c 7c6363cc
                df1f4824 ac43c272 5ea0c31e 3ae2b9e0
[INFO]  snarkJS: Contribution Hash:
                a732abd1 e5df41ba 9a20d915 979e95ca
                01ce5a8d d1669874 45f2bb6f 48ceaf7e
                0beb0e1e 6b4d04b1 495970be 3018c981
                fa2b49a7 2f263c46 092bae4d 0cae5594
+ snarkjs zkey export verificationkey MerkleTree_0001.zkey MerkleTree_verification_key.json
+ snarkjs groth16 prove MerkleTree_0001.zkey ../MerkleTree_js/witness.wtns MerkleTree_proof.json MerkleTree_public.json
+ snarkjs groth16 verify MerkleTree_verification_key.json MerkleTree_public.json MerkleTree_proof.json
[INFO]  snarkJS: OK!
+ snarkjs zkey export solidityverifier MerkleTree_0001.zkey MerkleTree_verifier.sol

real    15m55.610s
user    96m53.574s
sys     0m7.723s
```

- public.json file



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

In this question, we will ask you to start with a simple contract that mints an NFT, then commit the mint data to a Merkle tree contract.

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]

   [Answer] Code: https://github.com/geesimon/zku/blob/main/week1/ToyNFT.sol

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.

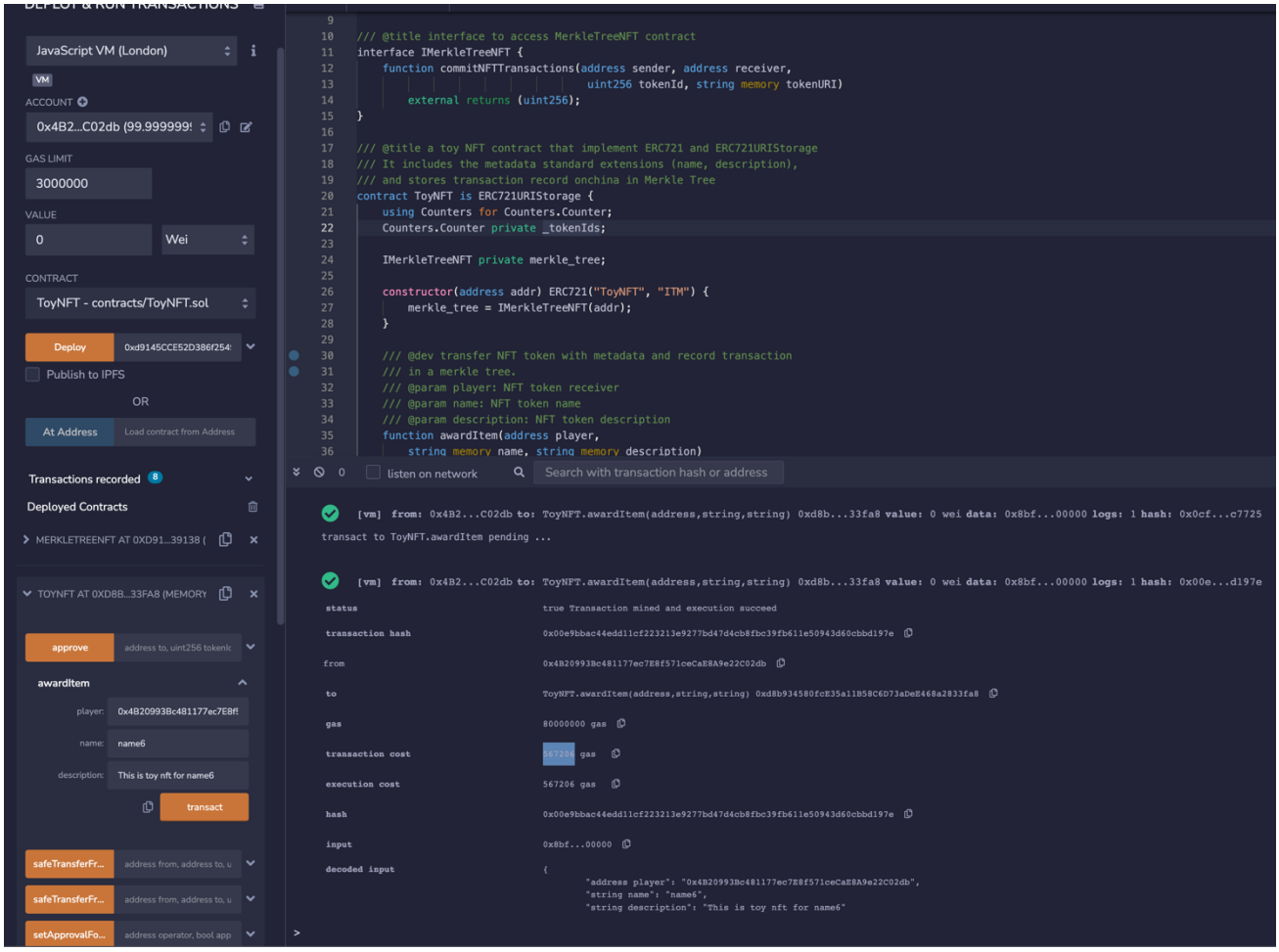
   [Answer] Code: https://github.com/geesimon/zku/blob/main/week1/MerkleTree.sol . Note, when insert a tree leaf, the code is optimized to only update impacted tree nodes instead of

rebuilding tree from scratch.

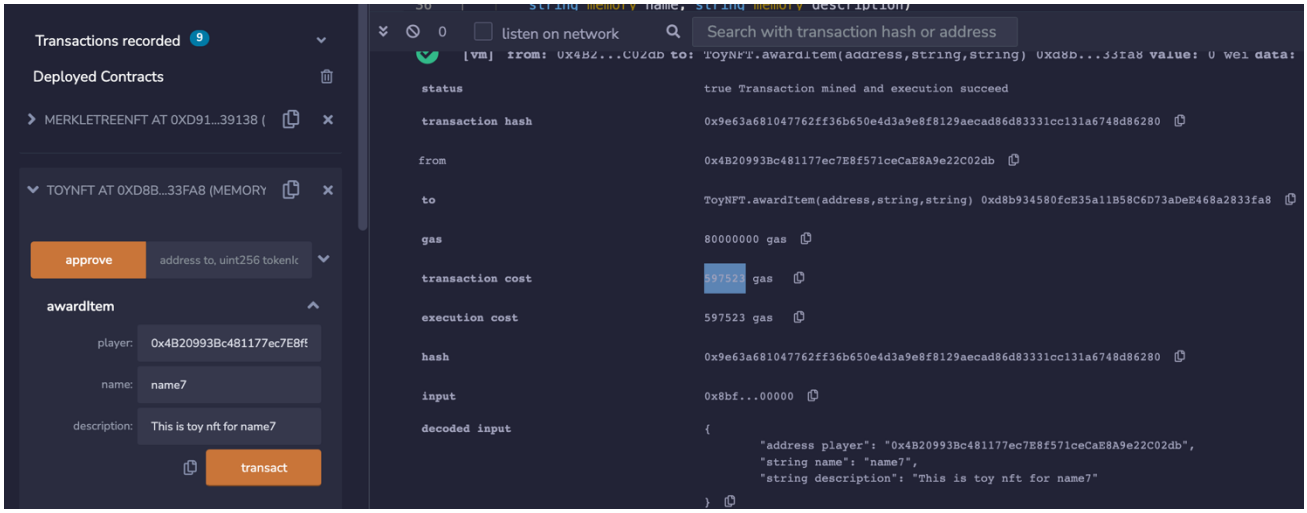
- Use remix to mint a couple of NFTs to the sender address or to other addresses. Include screenshots of the transactions and the amount of gas spent per transaction in your repo.

[Answer]

Screen shot for committing 6th NFT token (gas: 567,206)



Screen shot for 7th NFT token (gas: 597,523)



3. *[Bonus]* Build a minimal frontend application that allows MetaMask to interact with your contract and mint an NFT.

   [Answer]

4. *[Extra bonus if you did Part (4)]* Use your application to fetch the Merkle leaves from the NFT contract, then generate a proof of the root calculation using snarkjs. Submit this proof to the solidity verifier contract generated from Q1.
   [Answer]

3. Question 3: Understanding and generating ideas about ZK technologies

Understanding and generating ideas about ZK technologies

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

   [Answer] SNARKS and STARKS are both zero-knowledge proof technology enables one party to prove to another party that they know something without the prover having to convey the information itself. There key differences are:

   1. Proof size: STARKs have far larger proof sizes than SNARKs, which means that verifying STARKs takes more time than SNARKs and leads to STARKs requiring more gas to verify in EVM. (SNARKs win)

2. Trusted setup: no trusted set-up is required to begin utilizing STARKs in a network while SNARKs need such steps. This means STARKs are more deployment friendly. (STARKs win)
3. Quantum secure: SNARK depends on elliptic curves and is theoretically quantum computer breakable. STARKs relies on pure hash function and is considered more quantum robust. (STARKs win)

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

   [Answer] The major difference is PLONK is universal trusted setup and Groth16 is not. This means every time you change the circuit (for example: the circom file), you need to do a new trusted setup on Groth16.

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

   [Answer] Say people don't want to make the NFT transaction public accessible for privacy considerations. We can use ZK to hide these transactions, while the owner still can prove the ownership.

4. [Bonus] Give a novel idea on how we can apply ZK for Dao Tooling.

   [Answer]
   1. O2O: Decentralized Airbnb and Uber where homeowner, traveler, driver and passenger don't need to store all transaction and review records in a centralized organization (such as in airbnb and uber' data center today) in order to evaluate credibility by each other (which gives up privacy to big org and maybe abused). Instead, they owe their own data and ZKP can be leveraged to calculate such credit.
   2. AI federated training: data (such as medical record) are highly valued property to business. Today, deep learning needs more and more data to get trained and we'd better be able to consolidate data from different orgs without compromise their individual data property. ZKP could be a solution.
   3. Personal/Business Credit System: instead of reporting credit related behaviors to centralized system (like xxx in U.S. today). We could have such kind of records stored in a ZKP systematic and use smart contract to calculate all kinds of credits easily.This provides benefits as: more accessible to orgs, individual and developers, global support and privacy protection.
   4. SaaS on blockchain: I can image this is as decentralized SalesForce where web3, Internet computer and ZKP technology provide computing capability and trusted integration for different component/functionalities offered by various vendors. Users are free to choose their preference and don't need to worry about data privacy and vendor lock-in. Here, ZKP is used to validate the distributed components (offered by different vendors) comply which standards

(functionality/API) and measure credibility (user review and usage) without need to make centralized deployment.