Engineering Blockchain and Web3 Apps
Programming Assignment #4

In this lab you'll learn about

- `circom`, a tool for describing arithmetic circuits; and
- `snarkjs`, a tool for generating and verifying zk-SNARKs of circuit satisfaction.

You'll use this knowledge to explore the implementation of *private transactions* by:

- crafting a simple version of the Tornado spend circuit; and
- generating a proof of validity for a Tornado withdraw.

1 Setup

To get your environment set up, do the following:

1. Install `nodejs` and `npm`.
2. Download and unzip the starter code.
3. Run `npm install` within the resulting folder.
4. Run `npm test` and verify that most of the tests fail, but not because of missing dependencies.

Note if tests fail because of dependency issues, you might be using an incompatible version of `snarkjs`. Please use version 0.1.11 as specified in the `package.json` file. We also use our fork of `circom` from `alex-ozdemir/circom#cs251`. Run `npm list` and verify that you have the right versions of `snarkjs` and `circom` installed.

2 Learning about `circom`

First, follow the Iden3 `circom` tutorial at `https://docs.iden3.io/circom-snarkjs/`. You can stop after the "Verifying a proof" section.

Then, read our example circuits in `circuits/example.circom` and answer the questions in `artifacts/writeup.md`.

> *Deliverables:* `artifacts/writeup.md`

Then, demonstrate your knowledge of `circom` and `snarkjs`by creating a proof that $7 \times 17 \times 19 = 2261$ (using the `SmallOddFactors` circuit). Store the verifier key in `artifacts/verifier_key_factor.json` and the proof in `artifacts/proof_factor.json`.

> *Deliverables:* `artifacts/verifier_key_factor.json`, `artifacts/proof_factor.json`

3 A Switching Circuit

3.1 IfThenElse

The `IfThenElse` circuit (located in `circuits/spend.circom`) verifies the correct evaluation of a conditional expression. It has a single output, `out`, and three inputs:

- `condition`, which should be 0 or 1;
- `true_value`, which `out` will be equal to if `condition` is 1; and
- `false_value`, which `out` will be equal to if condition is 0.

`IfThenElse` additionally enforces that condition is indeed 0 or 1.
Implement `IfThenElse`.

> *Deliverables:* `IfThenElse` in file `circuits/spend.circom`

3.2 SelectiveSwitch

The `SelectiveSwitch` takes two inputs and produces two outputs, flipping the order if a third input is 1.
Implement `SelectiveSwitch`, making use of your `IfThenElse` circuit.

> *Deliverables:* `SelectiveSwitch` in file `circuits/spend.circom`

3.3 A Spend Circuit

We are going to implement a simplified Tornado withdrawal as discussed in our lecture. The user has a pair ($\texttt{nullifier}, \texttt{nonce}$) and this pair defines a coin as

$$\texttt{coin} = H(\texttt{nullifier}, \texttt{nonce})$$

where H is a hash function. Each such coin occupies one leaf of a Merkle tree. The first coin is placed in the left-most leaf of the Merkle tree, and every new coin is placed in the leaf immediately to the right of the previous coin.

When withdrawing a coin, the coin owner uses its ($\texttt{nullifier}, \texttt{nonce}$) pair to prove to the Tornado contract that the corresponding $\texttt{coin} = H(\texttt{nullifier}, \texttt{nonce})$ is one of the leaves of the Merkle tree, without revealing which one.

What you will do in this assignment is craft an arithmetic circuit for verifying that a ($\texttt{nullifier}, \texttt{nonce}$) pair corresponds to a coin in the Merkle tree. You will then reveal the `nullifier` publicly (allowing everyone to verify that this `nullifier` hasn't been spent already), and use a SNARK to prove the existence of a `nonce` such that the corresponding `coin` is in the Merkle tree, in zero-knowledge. The inputs to the circuit are thus:

- `digest`: the Merkle tree root digest (public);
- `nullifier`: the nullifier (public);
- `nonce`: the nonce (private); and
- Merkle path: a list of (direction, hash) pairs (private).

The circuit should verify that $H($`nullifier`$,$`nonce`$)$ is a leaf in a Merkle tree whose root hash is the provided `digest`. Specifically, the circuit should verify that the provided (private) Merkle path is a valid Merkle proof for the coin $H($`nullifier`$,$`nonce`$)$.

You should implement the verification circuit, `Spend`, in `circuits/spend.circom`.

For the hash function H used to define the coin and the Merkle tree, use the hash function `Mimc2` whose circuit has been included into the file. You should make use of your `SelectiveSwitch` circuit to handle the directions properly.

Deliverables: `Spend` in file `circuits/spend.circom`

4 Computing the Spend Circuit Input

The only task that remains is writing a program that computes the Merkle path for a given nullifier/coin.

Implement this by implementing the `computeInput` function in `src/compute_spend_input.js`.
This function takes the following inputs:

- `depth`: the depth of the Merkle tree.
- `coins`: a list of coins. Some are created by you, and are an array of two elements (the `nullifier` and `nonce`). Others were not created by you and are an array of a single value—the coin.
- `nullifier`: the nullifier to compute the circuit inputs for.

The function should return a JSON object suitable as input to the `Spend` circuit.

To assist you, we've provided a `SparseMerkleTree` class, which you will find in `src/sparse_merkle_tree.js`. For the commitment hash-function, use `mimc2`, which has been included in the file.

Deliverables: `src/compute_spend_input.js`

5 Proving a Withdrawal

Finally, use your input computer, `circom`, and `snarkjs` to create a SNARK proving the presence of the `nullifier` "10137284576094" in Merkle tree of depth 10 corresponding to the transcript `test/compute_spend_input/transcript3.txt`. Use a depth of 10 (you'll find a depth-10 instantiation of your `Spend` circuit in `test/circuits/spend10.circom`), and place your verifier key in `artifacts/verifier_key_spend.json` and your proof in `artifacts/proof_spend.json`.

Deliverables: `artifacts/verifier_key_spend.json`, `artifacts/proof_spend.json`

6 Testing

You can, of course, check your proofs using `snarkjs`.

We've also provided a few unit tests for the various components of your system, which can be run using `npm test`

7 Debugging Tips

Your version of `circom` supports the `log` (1 argument) function, which prints its argument.