CS251: Cryptocurrencies and Blockchain Technologies **Fall 2020**

Programming Project #4

Due: 11:59pm on Wed., **Nov. 11, 2020**
Submit via Gradescope code: **92ZNG8**

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 Zcash shielded spend circuit, and
- generating a proof of validity for a Zcash shielded spend.

1 Setup

To get your environment set up, do the following:

1. Install `nodejs` and `npm`.
2. Install `snarkjs` (`npm install -g snarkjs@0.1.11`).
3. Install our fork of `circom` (`npm install -g alex-ozdemir/circom#cs251`).
4. Install the `mocha test runner` (`npm install -g mocha`).
5. Download and unzip the starter code.
6. Run `npm install` within the resulting folder.
7. 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.

2 Learning About `circom`

First, follow the Iden3 `circom` tutorial at https://iden3.io/blog/circom-and-snarkjs-tutorial2.html. You can stop after the "Verifying the proof" section.

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

> *Deliverable:* `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.` and the proof in `artifacts/proof_factor.json`.

> *Deliverable:*
> `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 3 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`.

> *Deliverable:* `IfThenElse`

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.

> *Deliverable:* `SelectiveSwitch`

4 A Spend Circuit

We're going to implement shielded spend verification for a simple version of ZCash that handles only whole coins.

In our version of Zcash, users can engage in normal transactions using the bitcoin protocol, but have the additional option to use the *shielded pool* to anonymize their coins.

The shielded pool contains *commitments* to coins, which in our case will be hashes of a *nullifier* and a *nonce*:

$$\mathsf{commitment} = H(\mathsf{nullifier}, \mathsf{nonce})$$

The pool is append-only and is summarized by an append-only Merkle tree whose leaves at the commitments, in order of creation.[1] To convert a normal coin to a shielded coin, one chooses a large, random, (nullifier, nonce) pair, spends the normal coin, and publishes the commitment corresponding the nullifier and nonce. At this point, all network participants update their Merkle trees to include the new commitment. The commitment, and its location in the Merkle tree, are public, but the nullifier and nonce are secret.

The interesting part (the part you'll implement) is the spend. One could spend the shielded coin by revealing the (nullifier, nonce) pair, proving to the network that the corresponding commitment is in the shielded ool, and having the network check that you haven't already spent this coin. However, this way of spending reveals a direction link between the shielded coin's creation and spend—violating privacy.

What you'll do instead is craft an arithmetic circuit for verifying that a (nullifier, nonce) pair corresponds to a valid commitment in the Merkle tree. You'll 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 commitment is in the Merkle tree, in zero-knowledge. The inputs to the circuit are thus:

- the Merkle digest (public),
- the nullifier (public),
- the nonce (private), and
- the Merkle path—a list of direction, hash pairs (private)

[1] The tree has fixed depth and empty leaves are taken to have a value of 0 for the purpose of computing the hash of the tree

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

For the commitment and Merkle hash function, use the `Mimc2` circuit which has been included into the file. You should make use of your `SelectiveSwitch` circuit to handle the `directions` properly.

Deliverable: `Spend`

5 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 for the shielded pool.
- `transactions`: a list of transactions. 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 commitment.
- `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'll find in `src/sparse_merkle_tree.js`.

For the commitment hash-function, use `mimc2`, which has been included into the file.

Deliverable: `src/compute_spend_input.js`

6 Proving a Spend

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`.

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

7 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`.

8 Debugging Tips

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

9 Submission

Please upload a compressed file that contains all of your code files and write-up document to Gradescope. Do not include your npm modules directory for this project.