Proving circuits

After compiling the circuit and running the witness calculator with an appropriate input, we will have a file with extension .wtns that contains all the computed signals and, a file with extension .r1cs that contains the constraints describing the circuit. Both files will be used to create our proof.

Now, we will use thesnarkjs tool to generate and validate a proof for our input. In particular, using the multiplier2,we will prove that we are able to provide the two factors of the number 33 . That is, we will show that we know two integersa andb such that when we multiply them, it results in the number 33.

We are going to use theGroth16 zk-SNARK protocol. To use this protocol, you will need to generate atrusted setup .Groth16 requires a per circuit trusted setup . In more detail, the trusted setup consists of 2 parts:

- The powers of tau, which is independent of the circuit.
- The phase 2, which depends on the circuit.

Next, we provide a very basic ceremony for creating the trusted setup and we also provide the basic commands to create and verifyGroth16 proofs. Review the relatedBackground section and checkthe snarkjs tutorial for further information.

Powers of Tau

You can access the **help** of snarkjs by typing the command:

```
snarkjs --help
```

You can get general **statistics** of the circuit and print the **constraints**. Just run:

```
snarkjs info -c multiplier2.r1cs
snarkjs print -r multiplier2.r1cs -s multiplier2.sym
```

First, we start a new "powers of tau" ceremony:

snarkjs powersoftau new bn128 12 pot12_0000.ptau -v Then, we contribute to the ceremony:

snarkjs powersoftau contribute pot12_0000.ptau pot12_0001.ptau --name="First contribution" -v Now, we have the contributions to the powers of tau in the filepot12_0001.ptau and we can proceed with the Phase 2.

Phase 2

Thephase 2 iscircuit-specific . Execute the following command to start the generation of this phase:

snarkjs powersoftau prepare phase2 pot12_0001.ptau pot12_final.ptau -v Next, we generate a.zkey file that will contain the proving and verification keys together with all phase 2 contributions. Execute the following command to start a new zkey:

snarkjs groth16 setup multiplier2.r1cs pot12_final.ptau multiplier2_0000.zkey Contribute to the phase 2 of the ceremony:

snarkjs zkey contribute multiplier2_0000.zkey multiplier2_0001.zkey --name="1st Contributor Name" -v Verify the latest zkey snarkjs zkey verify 1.r1cs pot12_final.ptau 1_0001.zkey

Apply a random beacon:

```
snarkjs zkey beacon multiplier2_0001.zkey multiplier2_final.zkey 0102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f 10 -n="Final Beacon phase2"
```

Verify the final zkey snarkjs zkey verify 1.r1cs pot12_final.ptau 1_final.zkey

As before, you will be prompted to enter some random text to provide a source of entropy. The output will be a file named multiplier2_final.zkey, which we will use to **export the verification key**.

```
snarkjs zkey export verificationkey multiplier2_final.zkey verification_key.json
```

Now, the verification key from multiplier2_final.zkey is exported into the file verification_key.json.

You can always **verify** that the computations of a .ptau or a .zkey file are correct:

```
snarkjs powersoftau verify pot12_final.ptausnarkjs zkey verify multiplier2.r1cs pot12_final.ptau multiplier2_final.zkey
```

If everything checks out, you should see the following at the top of the output:

```
[INFO]  snarkJS: Powers of Tau file OK![INFO]  snarkJS: ZKey OK!
```

The command snarkjs zkey verify also checks that the .zkey file corresponds to the specific circuit.Export the verification key:

snarkjs zkey export verificationkey multiplier2_0001.zkey verification_key.json

Generating a Proof

Once the witness is computed and the trusted setup is already executed, we cangenerate a zk-proof associated to the circuit and the witness:

snarkjs groth16 prove multiplier2_0001.zkey witness.wtns proof.json public.json This command generates a Groth16 proof and outputs two files:

- proof.json
- : it contains the proof.
- public.json
- : it contains the values of the public inputs and outputs.

Verifying a Proof

Toverify the proof , execute the following command:

snarkjs groth16 verify verification_key.json public.json proof.json The command uses the filesverification_key.json we exported earlier,proof.json andpublic.json to check if the proof is valid. If the proof is valid, the command outputs anOK .

A valid proof not only proves that we know a set of signals that satisfy the circuit, but also that the public inputs and outputs that we use match the ones described in thepublic.json file.

Verifying from a Smart Contract

It is also possible to generate aSolidity verifier that allowsverifying proofs on Ethereum blockchain .

First, we need to generate the Solidity code using the command:

snarkjs zkey export solidityverifier multiplier2_0001.zkey verifier.sol This command takes validation keymultiplier2_0001.zkey and outputs Solidity code in a file namedverifier.sol . You can take the code from this file and cut and paste it in Remix. You will see that the code contains two contracts:Pairing andVerifier . You only need to deploy theVerifier contract.

You may want to use first a testnet like Rinkeby, Kovan or Ropsten. You can also use the JavaScript VM, but in some browsers the verification takes long and the page may freeze.

TheVerifier has aview function calledverifyProof that returnsTRUE if and only if the proof and the inputs are valid. To facilitate the call, you can usesnarkJS to generate the parameters of the call by typing:

snarkjs generatecall Cut and paste the output of the command to the parameters field of theverifyProof method in Remix. If everything works fine, this method should returnTRUE . You can try to change just a single bit of the parameters, and you will see that the result is verifiableFALSE .