[Deivitto/ZKP: Zero Knowledge Proof Deivitto (github.com)](github.com)

Part 1 Theoretical background of zk-SNARKs and zk-STARKS

1. ~~Write down two examples of SNARK proofs.~~~~Write down two examples of SNARK proofs.~~ Edit: Write down two types of SNARK proofs.

   Transparent setup, universal set up, circuit specific set up

   Reference: Comparing General Purpose zk-SNARKs | by Ronald Mannak | Coinmonks | Medium

2. Explain in 2-4 sentences why SNARK requires a trusted setup while STARK doesn't.

   Basically, SNARK is based on using elliptic curves, because of that needs to use keys, but if somebody would keep this keys that would be untrusted, for that reason a scenario where the keys are destroyed after being used is needed. In case of STARK, is based on hash technology, so it doesn't have those sensible keys.

3. Name two more differences between SNARK and STARK proofs.

   STARK will be post quantum secure while SNARK won't
   STARK gas cost is higher than SNARK

Part 2 Getting started with circom and snarkjs

Circom is perhaps the single most important tool we will be using throughout this course. Let's get familiar with it!

1. Follow the instructions on Circom 2 Documentation to install circom (2.0.3 or above) and snarkjs (0.4.16 or above) on your machine (Windows users are recommended to install via WSL). Read through the rest of the documentation to learn about the syntax of the circom language. You might also find this tutorial useful.

   Done.

2. Fork the `Week 1` repo and go into the `Q2` directory. Install all the node dependencies. In the `contracts/circuits` folder, you will find `HelloWorld.circom`. Run the bash script `scripts/compile-HelloWorld.sh` to compile the circuit. Answer the following questions (word answers should go into the PDF file):
   1. What does the circuit in `HelloWorld.circom` do?

      Checks c to be the multiplication of a and b

   2. Lines 7-12 of `compile-HelloWorld.sh` download a file called `powersOfTau28_hez_final_10.ptau` for Phase 1 trusted setup. Read more about how this is generated here. What is a Powers of Tau ceremony? Explain why this is important in the setup of zk-SNARK applications.

      Powers of Tau ceremony is the phase 1 trusted setup. Is a process where a group of players contributes their multiplicative factor to intermediate values of the CRS from previous players. If this is not done correctly this phase in zk-SNARK applications, this could lead into security problems, if the secrets created on the setup phase are not destroyed, the secrets could be utilized to forge transactions by false verifications, giving the holder the ability to perform actions such as creating new tokens out of thin air and using them for transactions.

   3. Line 24 of `compile-HelloWorld.sh` makes a random entropy contribution as a Phase 2 trusted setup. How are Phase 1 and Phase 2 trusted setup ceremonies different from each other?

      The first phase, is based on doing a setup with Powers of Tau, produces generic setup parameters that can be used for all circuits of the scheme, up to a given size. The phase 2 is based on convert the output of the powers of Tau into an NP-relation-specific CRS.

3. In this question, you will learn about an important restriction on circom circuits:
    1. In the empty `scripts/compile-Multiplier3-groth16.sh`, create a script to compile `contracts/circuits/Multiplier3.circom` and create a verifier contract modeling after `compile-HelloWorld.sh`.

```bash
#!/bin/bash

# [assignment] create your own bash script to compile _Multiplier3.circom modeling after compile-HelloWorld.sh below

cd contracts/circuits

mkdir Multiplier3

if [ -f ./powersOfTau28_hez_final_10.ptau ]; then
    echo "powersOfTau28_hez_final_10.ptau already exists. Skipping."
else
    echo 'Downloading powersOfTau28_hez_final_10.ptau'
    wget https://hermez.s3-eu-west-1.amazonaws.com/powersOfTau28_hez_final_10.ptau
fi

echo "Compiling Multiplier3.circom..."

# compile circuit

circom Multiplier3.circom --r1cs --wasm --sym -o Multiplier3
snarkjs r1cs info Multiplier3/Multiplier3.r1cs

# Start a new zkey and make a contribution

snarkjs groth16 setup Multiplier3/Multiplier3.r1cs powersOfTau28_hez_final_10.ptau Multiplier3/circuit_0000.zkey
snarkjs zkey contribute Multiplier3/circuit_0000.zkey Multiplier3/circuit_final.zkey --name="1st Contributor Name" -v -e="random text"
snarkjs zkey export verificationkey Multiplier3/circuit_final.zkey Multiplier3/verification_key.json

# generate solidity contract
snarkjs zkey export solidityverifier Multiplier3/circuit_final.zkey ../Multiplier3Verifier.sol

cd ../..
```

    2. Try to run `compile-Multiplier3-groth16.sh`. You should encounter an error with the circuit as is. Explain what the error means and how it arises.

       The error is that is not valid to use non quadratic constraints

    3. Modify `Multiplier3.circom` to perform a multiplication of three input signals under the restrictions of circom.

```
template Multiplier3 () {

    // Declaration of signals.
    signal input a;
    signal input b;
    signal input c;
    signal output d;
    signal ab <== a * b;
    // Constraints.
    d <== ab * c;
}
```

4. In the empty `scripts/compile-Multiplier3-plonk.sh`, create a script to compile `circuit/Multiplier3.circom` using PLONK in snarkjs. Add a `_plonk` prefix to the build folder and the output contract to distinguish the two sets of output.
    1. You will encounter an error if you just change `snarkjs groth16 setup` to `snarkjs plonk setup`. Resolve this error and answer the following question - How is the process of compiling with PLONK different from compiling with Groth16?

       You don't need to add a second phase into the process

    2. What are the practical differences between Groth16 and PLONK? Hint: compare and contrast the resulted contracts and running time of unit tests (see Q5 below) from the two protocols.

       One is using transparent setup while the other is using global setup. Also groth is faster than plonk in true test but not in false test.

5. So far we have not tested our circuit yet. While you can verify your circuit in the terminal using `snarkjs groth16 fullprove`, you can also do so directly in a Node.js script. We will practice doing so by creating some unit tests to try out our verifier contract(s):
    1. Running `npx hardhat test` will prompt an error. Before we can test our verifier contracts with hardhat, we must modify the solidity version. In `scripts/bump-solidity.js`, we have already written the regular expressions to modify `HelloWorldVerifier.sol`. Add script to `bump-solidity.js` to do the same for your new contract for `Multiplier3`.

```
const fs = require("fs");
const solidityRegex = /pragma solidity \^\d+\.\d+\.\d+/

const verifierRegex = /contract Verifier/
const plonkVerifierRegex = /contract PlonkVerifier/

var content = fs.readFileSync("./contracts/HelloWorldVerifier.sol", { encoding: 'utf-8' });
var bumped = content.replace(solidityRegex, 'pragma solidity ^0.8.0');
bumped = bumped.replace(verifierRegex, 'contract HelloWorldVerifier');

fs.writeFileSync("./contracts/HelloWorldVerifier.sol", bumped);

// [assignment] add your own scripts below to modify the other verifier contracts you will build during the assignment

// Code for Multiplier Verifier
content = fs.readFileSync("./contracts/Multiplier3Verifier.sol", { encoding: 'utf-8' });
bumped = content.replace(solidityRegex, 'pragma solidity ^0.8.0');
bumped = bumped.replace(verifierRegex, 'contract Multiplier3Verifier');

fs.writeFileSync("./contracts/Multiplier3Verifier.sol", bumped);


// Code for Plonk version
content = fs.readFileSync("./contracts/_plonkMultiplier3Verifier.sol", { encoding: 'utf-8' });
bumped = content.replace(solidityRegex, 'pragma solidity ^0.8.0');
bumped = bumped.replace(plonkVerifierRegex, 'contract _plonkMultiplier3Verifier');

fs.writeFileSync("./contracts/_plonkMultiplier3Verifier.sol", bumped);
```

2. You can now perform the unit tests for `HelloWorldVerifier` by running `npm run test`. Add inline comments to explain what each line in the test `Should return true for correct proof` is doing.

```
describe("HelloWorld", function () {
    let Verifier;
    let verifier;

    // before each test, deploy the contract

    beforeEach(async function () {
        Verifier = await ethers.getContractFactory("HelloWorldVerifier");
        verifier = await Verifier.deploy();
        await verifier.deployed();
    });

    it("Should return true for correct proof", async function () {
        //[assignment] Add comments to explain what each line is doing
        // it performs groth16.fullProve with input {"a": "1", "b": "2"},
        // wasm file "./contracts/circuits/HelloWorldVerifier.wasm",
        // and zkey file "./contracts/circuits/HelloWorldVerifier.zkey"
        const { proof, publicSignals } = await groth16.fullProve({"a":"1","b":"2"},
        "contracts/circuits/HelloWorld/HelloWorld_js/HelloWorld.wasm",
        "contracts/circuits/HelloWorld/circuit_final.zkey");

        // print the first value of the publicSignals, is the result of the circuit
        console.log('1x2 =',publicSignals[0]);

        // gets the signals in big int format
        const editedPublicSignals = unstringifyBigInts(publicSignals);
        // gets the proof in big int format
        const editedProof = unstringifyBigInts(proof);
        // gets the calldata string with the signals and the proof
        const calldata = await groth16.exportSolidityCallData(editedProof, editedPublicSignals);

        // gets from the calldata the proof and the public signals in big int format
        const argv = calldata.replace(/["[\]\s]/g, "").split(',').map(x => BigInt(x).toString());

        const a = [argv[0], argv[1]];
        const b = [[argv[2], argv[3]], [argv[4], argv[5]]];
        const c = [argv[6], argv[7]];
        const Input = argv.slice(8);

        // performs the verification with the variables above and expects to be true
        expect(await verifier.verifyProof(a, b, c, Input)).to.be.true;
    });
    it("Should return false for invalid proof", async function () {
        // examples that are gonna fail the verification
        let a = [0, 0];
        let b = [[0, 0], [0, 0]];
        let c = [0, 0];
        let d = [0]
        // expects to be false
        expect(await verifier.verifyProof(a, b, c, d)).to.be.false;
    });
});
```

3. In `test/test.js`, add the unit tests for `Multiplier3` for both the Groth16 and PLONK versions. Include a screenshot of all the tests (for `HelloWorld`, `Multiplier3 with Groth16`, and `Multiplier3 with PLONK`) passing in your PDF file.

```
chaparro_d@DESKTOP-JQCOJ81 MINGW64 ~/Documents/zk/week1/Q2 (master)
$ npm run test

> test
> node scripts/bump-solidity.js && npx hardhat test



  HelloWorld
1x2 = 2
    √ Should return true for correct proof (3799ms)
    √ Should return false for invalid proof (389ms)

  Multiplier3 with Groth16
1x2x3 = 6
    √ Should return true for correct proof (2535ms)
    √ Should return false for invalid proof (496ms)

  Multiplier3 with PLONK
1x2x4 = 8
    √ Should return true for correct proof (3295ms)
    √ Should return false for invalid proof


  6 passing (12s)


chaparro_d@DESKTOP-JQCOJ81 MINGW64 ~/Documents/zk/week1/Q2 (master)
$
```

Part 3 Reading and designing circuits with circom

Though it will be nice if we write entirely innovative circuits for every project we create, we should also utilize existing circuit libraries to help us. In this question, you will be learning about two such libraries that you can import to create more complicated circuits. To start, go into the `Q3` directory in `Week 1` repo and run `npm install` in each project folder to install the dependencies.

1. [circomlib]() is the official library of circuit templates released by iden3, the creator of Circom. One important template included is `comparators.circom`, which implements value comparisons between two numbers. The following questions will cover the use of this template in our own circuits:
    1. `contracts/circuits/LessThan10.circom` implements a circuit that verifies an input is less than 10 using the `LessThan` template. Study how the template is used in this circuit. What does the 32 in Line 9 stand for?

        N is the number of bits the input has

    2. What are the possible outputs for the `LessThan` template and what do they mean respectively? (If you cannot figure this out by reading the code alone, feel free to compile the circuit and test with different input values.)

        0 and 1. It means if it is less than the other number or not

    3. Proving a number is within a range without revealing the actual number could be useful in applications like proving our income when applying for a credit card. In `contracts/circuits/RangeProof.circom`, create a template (not circuit, so don't add `component main = ...`) that
    uses `GreaterEqThan` and `LessEqThan` to perform a range proof.

```
template RangeProof(n) {
    assert(n <= 252);
    signal input in; // this is the number to be proved inside the range
    signal input range[2]; // the two elements should be the range, i.e. [lower bound, upper bound]
    signal output out;

    component low = LessEqThan(n);
    component high = GreaterEqThan(n);

    // [assignment] insert your code here
    low.in[0] <== in[0];
    low.in[1] <== in[1];

    high.in[0] <== in[0];
    high.in[1] <== in[1];

    high.out && low.out ==> out;

}
```

2. [circomlib-matrix]() is a library covering basic matrix operations, modeled after circomlib, and created by our very own mentor Cathie. Matrix operations can be useful in puzzles (e.g. [zkPuzzles](), [zkGames]()), image processing (e.g [zkPhoto]()), and machine learning (e.g. [zk-mnist](), [zk-ml]()). Let's take a look at matrix operations in action in a [Sudoku]() circuit in the [zkPuzzles]() repo.
    1. In `projects/zkPuzzles/circuits`, modify Lines 20-23 of `sudoku.circom` so that it implements the check on the inputs to be between ~~1~~ **Edit: 0** and 9 (inclusive) using your `RangeProof` template from 1.3.

```
//[assignment] hint: you will need to initialize your RangeProof components here
component rangeProof = RangeProof(32);
rangeProof.range = [0,9]; // range of the puzzle
signal sucess;
for (var i=0; i<9; i++) {
    for (var j=0; j<9; j++) {
        //assert that the puzzle is in the range
        rangeProof.in = (puzzle[i][j])
        assert(rangeProof.out);

        //assert that the solution is in the range
        rangeProof.in = (solution[i][j])
        assert(rangeProof.out);

        mul.a[i][j] <== puzzle[i][j];
        mul.b[i][j] <== solution[i][j];
    }
}
```

    2. You can run `npm run test:fullProof` while inside the `zkPuzzles` directory to test your modified circuit. You are expected to encounter an error. Record the error, resolve it by modifying `project/zkPuzzles/scripts/compile-`

`circuits.sh`, and explain why it has occurred and what you did to solve the error.

I didn't make it work so im gonna say the error is the way of getting the output

3. Copy your modified `sudoku.circom` into `contracts/circuits/sudokuModified.circom` for submission, so you don't have to commit the submodule.

   Done

4. Instead of using a brute force method to verify a sudoku puzzle solution, the circuit here uses the sum and sum of squares of each row, each column, and each "box" to prove the solution. What is/are the benefit(s) of this algorithmic implementation over the brute force implementation?

   Brute force doesn't consider that the sum of rows, boxes or columns sums always the same number. This verification accelerates the speed by ignoring cases.