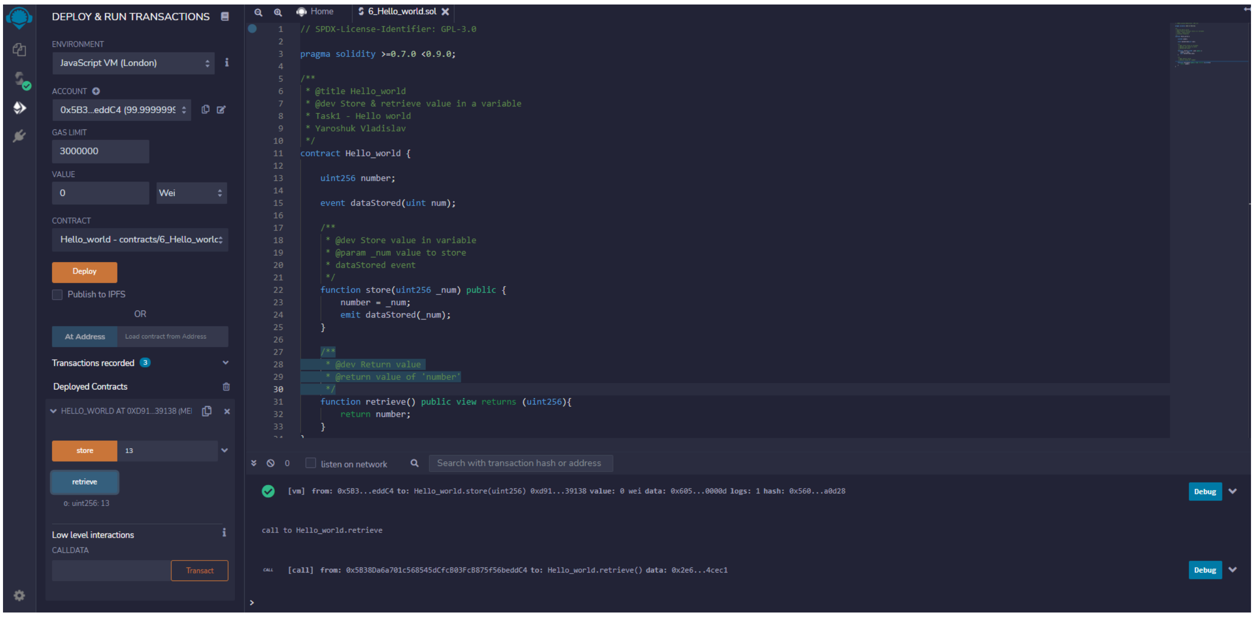
Question 1:

1. Program a super simple “Hello World” smart contract: store an unsigned integer and then retrieve it. Please clearly comment your code. Once completed, deploy the smart contract on Remix. Include the .sol file and a screenshot of the Remix UI once deployed in your final submission pdf (more info about submission formatting below).

Code:

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.7.0 <0.9.0;

/**
 * @title Hello_world
 * @dev Store & retrieve value in a variable
 * Task1 - Hello world
 * Yaroshuk Vladislav
 */
contract Hello_world {

    uint256 number;

    event dataStored(uint num);

    /**
     * @dev Store value in variable
     * @param _num value to store
     * dataStored event
     */
    function store(uint256 _num) public {
        number = _num;
        emit dataStored(_num);
    }

    /**
     * @dev Return value
     * @return value of 'number'
     */
    function retrieve() public view returns (uint256){
        return number;
    }
}
```



2. On the documentation page, the "Ballot" contract demonstrates a lot of features on Solidity. Read through the script and try to understand what each line of code is doing, then implement the Possible Improvements by **reducing the number of transactions in the "giveRightToVote" function while maintaining the same functionality of the program**.

Code:

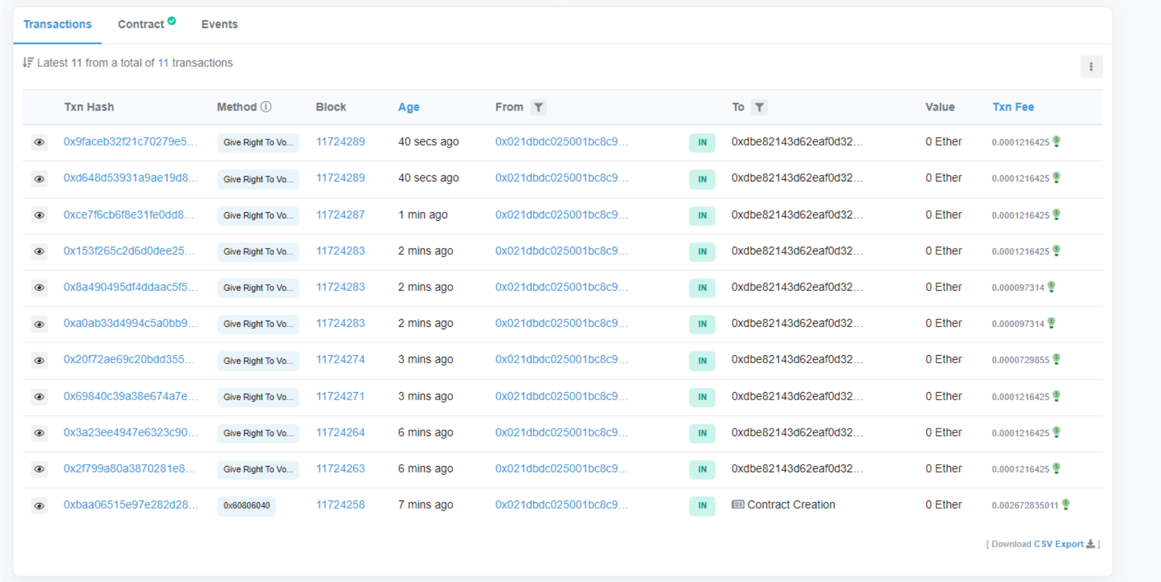
```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
/// @title Voting with delegation.
// Task2 - Ballot upgrade
// Yaroshuk Vladislav
contract Ballot {
    // This declares a new complex type which will
    // be used for variables later.
    // It will represent a single voter.
    struct Voter {
        uint weight; // weight is accumulated by delegation
        bool voted;  // if true, that person already voted
        address delegate; // person delegated to
        uint vote;   // index of the voted proposal
    }

    // This is a type for a single proposal.
    struct Proposal {
        bytes32 name;   // short name (up to 32 bytes)
        uint voteCount; // number of accumulated votes
    }

    address public chairperson;

    // This declares a state variable that
    // stores a `Voter` struct for each possible address.
    mapping(address => Voter) public voters;
```

```
    // A dynamically-sized array of `Proposal` structs.
    Proposal[] public proposals;

    /// Create a new ballot to choose one of `proposalNames`.
    constructor(bytes32[] memory proposalNames) {
        chairperson = msg.sender;
        voters[chairperson].weight = 1;

        // For each of the provided proposal names,
        // create a new proposal object and add it
        // to the end of the array.
        for (uint i = 0; i < proposalNames.length; i++) {
            // `Proposal({...})` creates a temporary
            // Proposal object and `proposals.push(...)`
            // appends it to the end of `proposals`.
            proposals.push(Proposal({
                name: proposalNames[i],
                voteCount: 0
            }));
        }
    }
    // Instead of taking a single address parameter in the giveRightToVote
function, take an array of addresses.
    // May only be called by `chairperson`.
    function giveRightToVote(address[] calldata _voters) external {
        // If the first argument of `require` evaluates
        // to `false`, execution terminates and all
        // changes to the state and to Ether balances
        // are reverted.
        // This used to consume all gas in old EVM versions, but
        // not anymore.
        // It is often a good idea to use `require` to check if
        // functions are called correctly.
        // As a second argument, you can also provide an
        // explanation about what went wrong.
        require(
            msg.sender == chairperson,
            "Only chairperson can give right to vote."
        );

        // Loop through the addresses and execute the current function logic on
each address.
        for (uint i = 0; i < _voters.length; i++) {
            // This require cheks every voter if it has already voted
            require(
                !voters[_voters[i]].voted,
                "Voter already voted"
            );
            // This require cheks every voter if it has an allowance to vote
            require(voters[_voters[i]].weight == 0);
```

```
        voters[_voters[i]].weight = 1;
    }
}

/// Delegate your vote to the voter `to`.
function delegate(address to) external {
    // assigns reference
    Voter storage sender = voters[msg.sender];
    require(!sender.voted, "You already voted.");

    require(to != msg.sender, "Self-delegation is disallowed.");

    // Forward the delegation as long as
    // `to` also delegated.
    // In general, such loops are very dangerous,
    // because if they run too long, they might
    // need more gas than is available in a block.
    // In this case, the delegation will not be executed,
    // but in other situations, such loops might
    // cause a contract to get "stuck" completely.
    while (voters[to].delegate != address(0)) {
        to = voters[to].delegate;

        // We found a loop in the delegation, not allowed.
        require(to != msg.sender, "Found loop in delegation.");
    }

    // Since `sender` is a reference, this
    // modifies `voters[msg.sender].voted`
    sender.voted = true;
    sender.delegate = to;
    Voter storage delegate_ = voters[to];
    if (delegate_.voted) {
        // If the delegate already voted,
        // directly add to the number of votes
        proposals[delegate_.vote].voteCount += sender.weight;
    } else {
        // If the delegate did not vote yet,
        // add to her weight.
        delegate_.weight += sender.weight;
    }
}

/// Give your vote (including votes delegated to you)
/// to proposal `proposals[proposal].name`.
function vote(uint proposal) external {
    Voter storage sender = voters[msg.sender];
    require(sender.weight != 0, "Has no right to vote");
    require(!sender.voted, "Already voted.");
    sender.voted = true;
    sender.vote = proposal;
```

```
        // If `proposal` is out of the range of the array,
        // this will throw automatically and revert all
        // changes.
        proposals[proposal].voteCount += sender.weight;
    }

    /// @dev Computes the winning proposal taking all
    /// previous votes into account.
    function winningProposal() public view
            returns (uint winningProposal_)
    {
        uint winningVoteCount = 0;
        for (uint p = 0; p < proposals.length; p++) {
            if (proposals[p].voteCount > winningVoteCount) {
                winningVoteCount = proposals[p].voteCount;
                winningProposal_ = p;
            }
        }
    }

    // Calls winningProposal() function to get the index
    // of the winner contained in the proposals array and then
    // returns the name of the winner
    function winnerName() external view
            returns (bytes32 winnerName_)
    {
        winnerName_ = proposals[winningProposal()].name;
    }
}
```

3. Deploy your script on Remix and compare the difference in gas fees between the original script and the improved script when giving 10 voters the right to vote. Once completed, submit (1) your improved version of the contract as an .sol file with comments describing the changes you made, and (2) screenshots (before and after) of the gas fees for the transaction(s) to give 10 voters the right to vote.
Giving 10 voters right to vote:



Transactions | Contract | Events

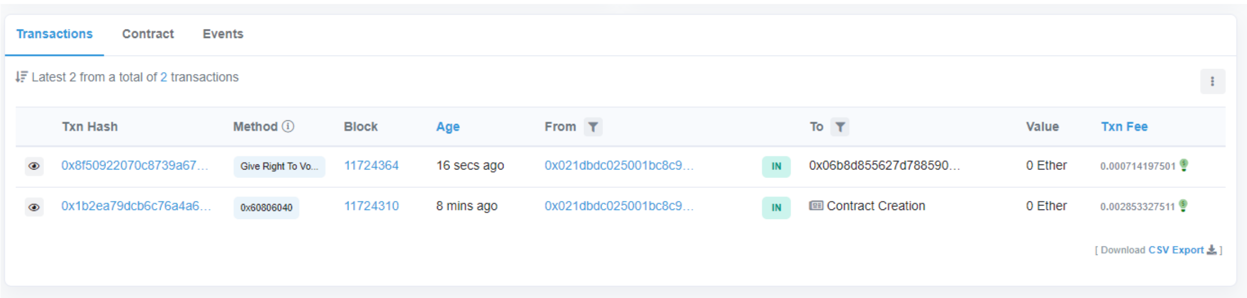
Latest 11 from a total of 11 transactions

Txn Hash	Method	Block	Age	From		To	Value	Txn Fee
0x9faceb32f21c70279e5...	Give Right To Vo...	11724289	40 secs ago	0x021dbdc025001bc8c9...	IN	0xdbe82143d62eaf0d32...	0 Ether	0.0001216425
0xd648d53931a9ae19d8...	Give Right To Vo...	11724289	40 secs ago	0x021dbdc025001bc8c9...	IN	0xdbe82143d62eaf0d32...	0 Ether	0.0001216425
0xce7f6cb6f8e31fe0dd8...	Give Right To Vo...	11724287	1 min ago	0x021dbdc025001bc8c9...	IN	0xdbe82143d62eaf0d32...	0 Ether	0.0001216425
0x153f265c2d6d0dee25...	Give Right To Vo...	11724283	2 mins ago	0x021dbdc025001bc8c9...	IN	0xdbe82143d62eaf0d32...	0 Ether	0.0001216425
0x8a490495df4ddaac5f5...	Give Right To Vo...	11724283	2 mins ago	0x021dbdc025001bc8c9...	IN	0xdbe82143d62eaf0d32...	0 Ether	0.000097314
0xa0ab33d4994c5a0bb9...	Give Right To Vo...	11724283	2 mins ago	0x021dbdc025001bc8c9...	IN	0xdbe82143d62eaf0d32...	0 Ether	0.000097314
0x20f72ae69c20bdd355...	Give Right To Vo...	11724274	3 mins ago	0x021dbdc025001bc8c9...	IN	0xdbe82143d62eaf0d32...	0 Ether	0.0000729855
0x69840c39a38e674a7e...	Give Right To Vo...	11724271	3 mins ago	0x021dbdc025001bc8c9...	IN	0xdbe82143d62eaf0d32...	0 Ether	0.0001216425
0x3a23ee4947e6323c90...	Give Right To Vo...	11724264	6 mins ago	0x021dbdc025001bc8c9...	IN	0xdbe82143d62eaf0d32...	0 Ether	0.0001216425
0x2f799a80a3870281e8...	Give Right To Vo...	11724263	6 mins ago	0x021dbdc025001bc8c9...	IN	0xdbe82143d62eaf0d32...	0 Ether	0.0001216425
0xbaa06515e97e282d28...	0x60806040	11724258	7 mins ago	0x021dbdc025001bc8c9...	IN	Contract Creation	0 Ether	0.002672835011

[Download CSV Export]

Fees in Ropsten:

Updated contract:



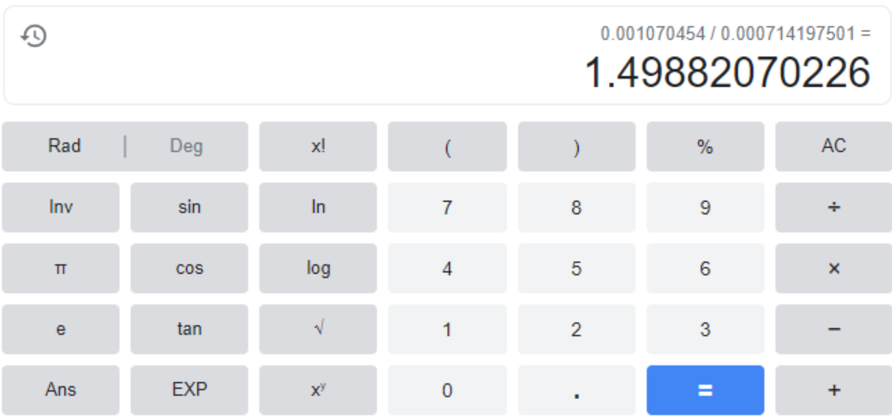
Transactions | Contract | Events

Latest 2 from a total of 2 transactions

Txn Hash	Method	Block	Age	From		To	Value	Txn Fee
0x8f50922070c8739a67...	Give Right To Vo...	11724364	16 secs ago	0x021dbdc025001bc8c9...	IN	0x06b8d855627d788590...	0 Ether	0.000714197501
0x1b2ea79dcb6c76a4a6...	0x60806040	11724310	8 mins ago	0x021dbdc025001bc8c9...	IN	Contract Creation	0 Ether	0.002853327511

[Download CSV Export]

Difference



Q2:

1. Briefly describe signals, templates, components, constraints, and why we use Circom for ZK.

The arithmetic circuits built using circom operate on **signals**, which contain field elements in Z/pZ. **Signals** can be named with an identifier or can be stored in arrays and declared using the keyword signal. **Signals** can be defined as input or output, and are considered intermediate signals otherwise.

The mechanism to create generic circuits in Circom is the so-called **templates**. They are normally parametric on some values that must be instantiated when the **template** is used. The instantiation of a **template** is a new circuit object, which can be used to compose other circuits, so as part of larger circuits.

Circom allows programmers to define the **constraints** that define the arithmetic circuit. The equations that describe the circuit are called **constraints**.

With circom, you design your own circuits with your own constraints, and the compiler outputs the R1CS representation that you will need for your zero-knowledge proof. The nice thing about circuits, is that although most **zero-knowledge protocols have an inherent complexity** that can be overwhelming for many developers, the **design of arithmetic circuits is clear and neat**.

2. For the following truth table, what is the constraint system? Hint: your answer should be a simple equation relating x1 and x2 to y.

x1	x2	y
0	0	0
0	1	0
1	0	0
1	1	1

Answer: $y = x_1 \cdot x_2$

3. Using open source resources like circomlib, **define, compile, and prove** a circuit that checks if a **number is less than 10.**

1) Submit a screenshot of the terminal outputs (to demonstrate your code effectively runs) along with (a well-commented) .circom file. (You will just copy and paste the contents of this file into your final pdf.)
2) Concisely describe what each command in the compilation and proving steps are doing under the hood. If you submit a screenshot of terminal outputs, you could write this in a separate file. If you write a bash file you can add comments above each line and then submit the bash file.
3)

Writing circuits

```
pragma circom 2.0.0;
include "./circim/circomlib/circuits/comparators.circom";
// We will be using LessThan function from circomlib

/* Logic:
   It's convinient to take a = 15, it can be represented as [1 1 1 1].
   The LessThan circuit attempts to build the bit representation of the minimum
integer that requires
   one (1) additional bit, which is 16 (represented with 5 bits: 1 0 0 0 0)
   The LessThan circuit will then add the result of (a - 10) to the 5-bit number.
   If a - 10 < 0, then the most significant bit of 5-bit number will become 0.
   If a - 10 > 0, then the most significant bit of 5-bit number will remain 1.
   Signal output = 1 will give either 0 (meaning a >= 10) or 1 (meaning a < 10).
*/

// N is the number of bits the input  have.
template ex2(n) {
   // Declaration of signals.
   signal input a;
   signal output out;
   var x = 10;

   component lessThan = LessThan(n);
   // Constraints.
   lessThan.in[0] <-- a;
   lessThan.in[1] <-- x;

   out <== lessThan.out;
}

// I will assign n to be 32-bit, it should be enough.
component main = ex2(32);
```

Compiling our circuit

After we write our arithmetic circuit using **circom**, we should save it in a file with the **.circom** extension.

We create a file called **comparison.circom.** Now is time to compile the circuit to get a system of arithmetic equations representing it. As a result of the compilation we will also obtain programs to compute the witness. We can compile the circuit with the following command:

$ circom comparison.circom --r1cs --wasm --sym --c

```
ubuntu@ip-172-31-9-53:~$ circom comparison.circom --r1cs --wasm --sym --c
template instances: 3
non-linear constraints: 33
linear constraints: 0
public inputs: 0
public outputs: 1
private inputs: 1
private outputs: 0
wires: 36
labels: 40
Written successfully: ./comparison.r1cs
Written successfully: ./comparison.sym
Written successfully: ./comparison_cpp/comparison.cpp and ./comparison_cpp/comparison.dat
Written successfully: ./comparison_cpp/main.cpp, circom.hpp, calcwit.hpp, calcwit.cpp, fr.hpp, fr.cpp, fr.asm and Makefile
Written successfully: ./comparison_js/comparison.wasm
Everything went okay, circom safe
ubuntu@ip-172-31-9-53:~$
```

Computing our witness

Before creating the proof, we need to calculate all the signals of the circuit that match all the constraints of the circuit. For that, we will use the **Wasm** module generated by **circom** that helps to do this job.

We need to create a file named **input.json** containing the inputs written in the standard **json** format.

```
ubuntu@ip-172-31-9-53:~$ sudo nano input.json
ubuntu@ip-172-31-9-53:~$
```

```
  GNU nano 4.8
{"a" : 15}
```

Computing the witness with WebAssembly

Enter in the directory **comparison_js**, add the input in a file **input.json** and execute:

$ node generate_witness.js comparison.wasm input.json witness.wtns

```
ubuntu@ip-172-31-9-53:~/comparison_js$ node generate_witness.js comparison.wasm input.json witness.wtns
ubuntu@ip-172-31-9-53:~/comparison_js$
```

It will generate the same **witness.wtns** file. This file is encoded in a binary format compatible with **snarkjs**, which is the tool that we use to create the actual proofs.

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.

```
ubuntu@ip-172-31-9-53:~/comparison_js$ ls
comparison.wasm  generate_witness.js  input.json  witness.wtns  witness_calculator.js
ubuntu@ip-172-31-9-53:~/comparison_js$
```

Now, we will use the **snarkjs** tool to generate and validate a proof for our input.

We are going to use the **Groth16 zk-SNARK protocol**. To use this protocol, you will need to generate a trusted 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 verify **Groth16** proofs..

Powers of Tau

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

$ snarkjs powersoftau new bn128 12 pot12_0000.ptau -v

```
ubuntu@ip-172-31-9-53:~/comparison_js$ snarkjs powersoftau new bn128 12 pot12_0000.ptau -v
[DEBUG] snarkJS: Calculating First Challenge Hash
[DEBUG] snarkJS: Calculate Initial Hash: tauG1
[DEBUG] snarkJS: Calculate Initial Hash: tauG2
[DEBUG] snarkJS: Calculate Initial Hash: alphaTauG1
[DEBUG] snarkJS: Calculate Initial Hash: betaTauG1
[DEBUG] snarkJS: Blank Contribution Hash:
                786a02f7 42015903 c6c6fd85 2552d272
                912f4740 e1584761 8a86e217 f71f5419
                d25e1031 afee5853 13896444 934eb04b
                903a685b 1448b755 d56f701a fe9be2ce
[INFO]  snarkJS: First Contribution Hash:
                9e63a5f6 2b96538d aaed2372 481920d1
                a40b9195 9ea38ef9 f5f6a303 3b886516
                0710d067 c09d0961 5f928ea5 17bcdf49
                ad75abd2 c8340b40 0e3b18e9 68b4ffef
```

Then, we contribute to the ceremony:

$ snarkjs powersoftau contribute pot12_0000.ptau pot12_0001.ptau --name="First contribution" -v

```
ubuntu@ip-172-31-9-53:~$ snarkjs powersoftau contribute pot12_0000.ptau pot12_0001.ptau --name="First contribution" -v
Enter a random text. (Entropy): comparison
[DEBUG] snarkJS: Calculating First Challenge Hash
[DEBUG] snarkJS: Calculate Initial Hash: tauG1
[DEBUG] snarkJS: Calculate Initial Hash: tauG2
[DEBUG] snarkJS: Calculate Initial Hash: alphaTauG1
[DEBUG] snarkJS: Calculate Initial Hash: betaTauG1
[DEBUG] snarkJS: processing: tauG1: 0/8191
[DEBUG] snarkJS: processing: tauG2: 0/4096
[DEBUG] snarkJS: processing: alphaTauG1: 0/4096
[DEBUG] snarkJS: processing: betaTauG1: 0/4096
[DEBUG] snarkJS: processing: betaTauG2: 0/1
[INFO]  snarkJS: Contribution Response Hash imported:
                4178d4b5 e4c15947 48310da1 cc572fff
                9576c272 5a09658f b4518c5d b8a119ce
                b3e964d8 00b85f70 3edf18f2 ebd55521
                95bed1ff abde3816 cb322d13 3e7540d5
[INFO]  snarkJS: Next Challenge Hash:
                a926c1d4 953ba168 99014d32 213be99f
                c5523cf5 5ed46537 0e53a7d0 360e3f7b
                6f5c817d ef193980 448e243b d85ecc51
                ecd2a598 02e0dc90 becc7b55 644bd871
```

Now, we have the contributions to the powers of tau in the file pot12_0001.ptau and we can proceed with the Phase 2.

Phase 2

The phase 2 is circuit-specific. Execute the following command to start the generation of this phase:

snarkjs powersoftau prepare phase2 pot12_0001.ptau pot12_final.ptau -v

```
ubuntu@ip-172-31-9-53:~$ snarkjs powersoftau prepare phase2 pot12_0001.ptau pot12_final.ptau -v
[DEBUG] snarkJS: Starting section: tauG1
[DEBUG] snarkJS: tauG1: fft 0 mix start: 0/1
[DEBUG] snarkJS: tauG1: fft 0 mix end: 0/1
[DEBUG] snarkJS: tauG1: fft 1 mix start: 0/1
[DEBUG] snarkJS: tauG1: fft 1 mix end: 0/1
[DEBUG] snarkJS: tauG1: fft 2 mix start: 0/1
[DEBUG] snarkJS: tauG1: fft 2 mix end: 0/1
[DEBUG] snarkJS: tauG1: fft 3 mix start: 0/1
[DEBUG] snarkJS: tauG1: fft 3 mix end: 0/1
[DEBUG] snarkJS: tauG1: fft 4 mix start: 0/1
[DEBUG] snarkJS: tauG1: fft 4 mix end: 0/1
[DEBUG] snarkJS: tauG1: fft 5 mix start: 0/1
[DEBUG] snarkJS: tauG1: fft 5 mix end: 0/1
[DEBUG] snarkJS: tauG1: fft 6 mix start: 0/1
[DEBUG] snarkJS: tauG1: fft 6 mix end: 0/1
[DEBUG] snarkJS: tauG1: fft 7 mix start: 0/1
[DEBUG] snarkJS: tauG1: fft 7 mix end: 0/1
[DEBUG] snarkJS: tauG1: fft 8 mix start: 0/1
[DEBUG] snarkJS: tauG1: fft 8 mix end: 0/1
[DEBUG] snarkJS: tauG1: fft 9 mix start: 0/1
[DEBUG] snarkJS: tauG1: fft 9 mix end: 0/1
[DEBUG] snarkJS: tauG1: fft 10 mix start: 0/1
[DEBUG] snarkJS: tauG1: fft 10 mix end: 0/1
[DEBUG] snarkJS: tauG1: fft 11 mix start: 0/1
[DEBUG] snarkJS: tauG1: fft 11 mix end: 0/1
[DEBUG] snarkJS: tauG1: fft 12 mix start: 0/1
[DEBUG] snarkJS: tauG1: fft 12 mix end: 0/1
```

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 comparison.r1cs pot12_final.ptau comparison_0000.zkey

```
ubuntu@ip-172-31-9-53:~$ snarkjs groth16 setup comparison.r1cs pot12_final.ptau comparison_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:
                fe31ebfa c5245ca2 ae8c98af d7c316dc
                9540d121 ac47c396 908dcd03 477dd83a
                e2439513 74d1fdfa 5020b8c0 5fd5d232
                60fb79d8 2e10e91b a3f62b7b b81d3e8d
ubuntu@ip-172-31-9-53:~$
```

Contribute to the phase 2 of the ceremony:

$ snarkjs zkey contribute comparison_0000.zkey comparison_0001.zkey --name="1st Contributor Name" -v

```
ubuntu@ip-172-31-9-53:~$ snarkjs zkey contribute comparison_0000.zkey comparison_0001.zkey --name="1st Contributor Name" -v
Enter a random text. (Entropy): dasdasd
[DEBUG] snarkJS: Applying key: L Section: 0/34
[DEBUG] snarkJS: Applying key: H Section: 0/64
[INFO]  snarkJS: Circuit Hash:
                fe31ebfa c5245ca2 ae8c98af d7c316dc
                9540d121 ac47c396 908dcd03 477dd83a
                e2439513 74d1fdfa 5020b8c0 5fd5d232
                60fb79d8 2e10e91b a3f62b7b b81d3e8d
[INFO]  snarkJS: Contribution Hash:
                30555a62 0ebb8be3 160a21c1 2e4fc4f9
                e2b2e112 767d300a 89cb659f 63eabd9e
                54142701 0992a5bf ac8db49f 263a070c
                18e118d8 be910cbe 696bc6e9 831e4447
ubuntu@ip-172-31-9-53:~$
```

Export the verification key:

$ snarkjs zkey export verificationkey comparison_0001.zkey verification_key.json

```
ubuntu@ip-172-31-9-53:~$ snarkjs zkey export verificationkey comparison_0001.zkey verification_key.json
ubuntu@ip-172-31-9-53:~$
```

Generating a Proof

$ snarkjs groth16 prove comparison_0001.zkey witness.wtns proof.json public.json

```
ubuntu@ip-172-31-9-53:~$ snarkjs groth16 prove comparison_0001.zkey comparison_js/witness.wtns proof.json public.json
ubuntu@ip-172-31-9-53:~$ ls
circom            comparison.r1cs  comparison_0000.zkey  comparison_cpp  input.json      pot12_0001.ptau   proof.json   verification_key.json
comparison.circom comparison.sym   comparison_0001.zkey  comparison_js   pot12_0000.ptau pot12_final.ptau  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

To verify the proof, execute the following command:

$ snarkjs groth16 verify verification_key.json public.json proof.json

The command uses the files **verification_key.json** we exported earlier, **proof.json** and **public.json** to check if the proof is valid. If the proof is valid, the command outputs an OK.

```
ubuntu@ip-172-31-9-53:~$ snarkjs groth16 verify verification_key.json public.json proof.json
[INFO]  snarkJS: OK!
ubuntu@ip-172-31-9-53:~$
```

We were checking if 15 < 10, result is **False**.

$ nano public.json

```
  GNU nano 4.8
[
 "0"
]
```

Verifying from a Smart Contract

It is also possible to generate a Solidity verifier that allows verifying proofs on Ethereum blockchain.

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

$ snarkjs zkey export solidityverifier comparison_0001.zkey verifier.sol

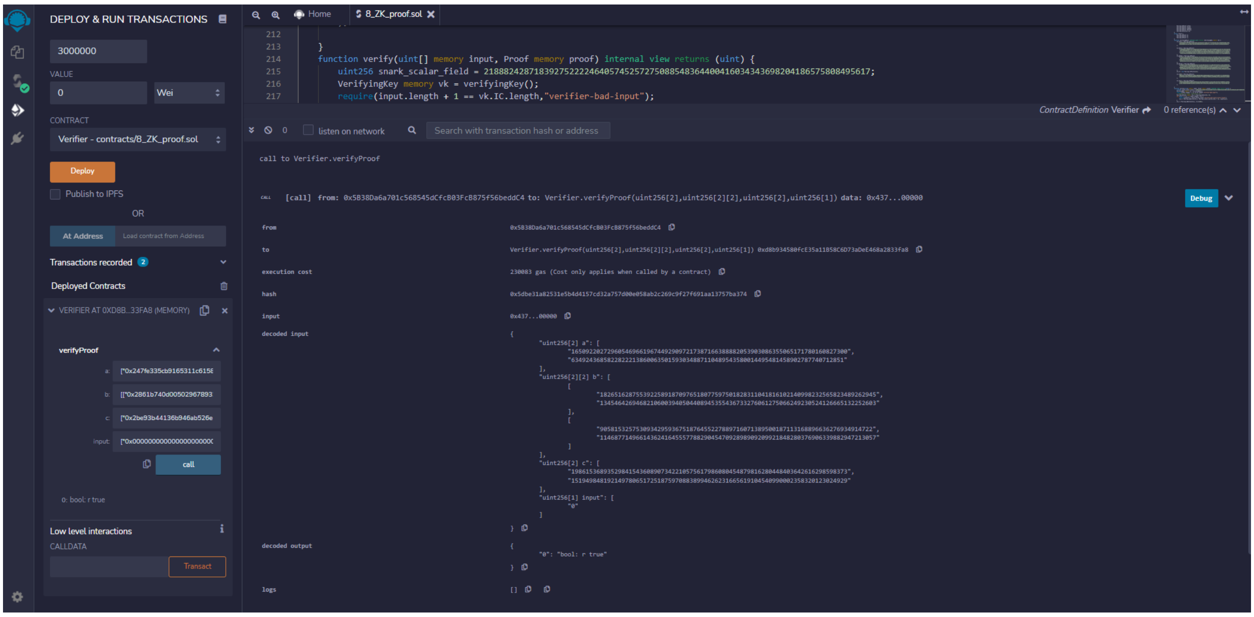
```
ubuntu@ip-172-31-9-53:~$ snarkjs zkey export solidityverifier comparison_0001.zkey verifier.sol
ubuntu@ip-172-31-9-53:~$ ls
circom             comparison.r1cs  comparison_0000.zkey  comparison_cpp  input.json       pot12_0001.ptau   proof.json   verification_key.json
comparison.circom  comparison.sym   comparison_0001.zkey  comparison_js   pot12_0000.ptau  pot12_final.ptau  public.json  verifier.sol
ubuntu@ip-172-31-9-53:~$ nano verifier.sol
ubuntu@ip-172-31-9-53:~$ snarkjs generatecall
["0x247fe335cb9165311c6158374c8d56e932a92e05c1a9f92ea0fb23e4de49afa4", "0x0e098b69511178f32b43ef04851ef41be09a643db144232e5fa8e22299fcbf93"],[["0x2861b740d005029678932be7158563a5eede0fd9ddef
ece4742536b295514561", "0x1dbf0ed262589aa148c67e9a43e446527828881352c4cd673e5f3a64b3fc9dbb"],["0x1406bba9c45c1b4c6067f4847db86f050b060e76167e5b0c0fb4d4b194af32a2", "0x195b1839c2bd9cdb271fc86
ec212c93774898ddd6e501b58a38144bfe2b67701"]],["0x2be93b44136b946ab526e4c6475e175bdeccfcaa66593e00d46c32e1deafefe5", "0x21980e7f82a5c4ae707663cda31ae9dca0f2adc8df1546c0dd64a278bf6b1621"],["0x
0000000000000000000000000000000000000000000000000000000000000000"]
ubuntu@ip-172-31-9-53:~$
```

Output:
["0x247fe335cb9165311c6158374c8d56e932a92e05c1a9f92ea0fb23e4de49afa4", "0x0e098b69511178f32b43ef04851ef41be09a643db144232e5fa8e22299fcbf93"],[["0x2861b740d005029678932be7158563a5eede0fd9ddefece4742536b295514561"
,
"0x1dbf0ed262589aa148c67e9a43e446527828881352c4cd673e5f3a64b3fc9dbb"],["0x1406bba9c45c1b4c6067f4847db86f050b060e76167e5b0c0fb4d4b194af32a2", "0x195b1839c2bd9cdb271fc86ec212c93774898ddd6e501b58a38144bfe2b67701"]],["0x2be93b44136b946ab526e4c6475e175bdeccfcaa66593e00d46c32e1deafefe5"
,
"0x21980e7f82a5c4ae707663cda31ae9dca0f2adc8df1546c0dd64a278bf6b1621"],[

"0x0000000000000000000000000000000000000000000000000000000000000000"]

Cut and paste the output of the command to the parameters field of the **verifyProof** method in Remix. If everything works fine, this method should return TRUE. You can see contract code in my github.

https://github.com/grGred/Zero-Knowledge-University



Question 3: Thinking with ZK

1. Describe an application you believe would benefit from ZK. Explain why ZK can benefit the application. Then formulate the use of ZK in terms of a statement and witness. Be as precise as possible in your use of variables to represent which data should be public and private, as well as the exact relations between them.

It would great to see zk-rollups in Uniswap for example. Right now there are millions of arbitrage bots and frontrunners that can easily spoil your mood. In order to prevent such types of attacks, we can have public function (swap), private input (tx calldata), public output (balance change).
When we making swap, we will receive private and public input, when smart contract will want to know our balance (and to make further transaction), we will give him a prove (private input) that we really own this funds, before that swap everyone will now that you possibly own some amount of this token.

2. List the key difference between an interactive and a non-interactive ZK proof. What type has more relevance to crypto?

SNARKs are short for *succinct non-interactive arguments of knowledge*. In this general setting of so-called interactive protocols, there is a *prover* and a *verifier* and the prover wants to convince the verifier about a statement (e.g. that f(x) = y) by exchanging messages. The generally desired properties are that no prover can convince the verifier about a wrong statement (*soundness*) and there is a certain strategy for the prover to convince the verifier about any true statement (*completeness*).

Non-interactive: there is no or only little interaction. For zkSNARKs, there is usually a setup phase and after that a single message from the prover to the verifier. Furthermore, SNARKs often have the so-called "public verifier" property meaning that anyone can verify without interacting anew, which is important for blockchains.

Non-interactive ZK proof is more secure and decentralized, less vulnerable to the economic attacks.

3. Beyond privacy, ZK helps with scalability. In a paragraph or less, describe what ZK-rollups are, how they work, and why people are excited about them.

Rollups perform transaction execution outside layer 1 and then the data is posted to layer 1 where consensus is reached. As transaction data is included in layer 1 blocks, this allows rollups to be secured by native Ethereum security. The ZK-rollup smart contract maintains the state of all transfers on layer 2, and this state can only be updated with a validity proof. This means that ZK-rollups only need the validity proof instead of all transaction data. With a ZK-rollup, validating a block is quicker and cheaper because less data is included. With a ZK-rollup, there are no delays when moving funds from layer 2 to layer 1 because a validity proof accepted by the ZK-rollup contract has already verified the funds. Being on layer 2, ZK-rollups can be optimised to reduce transaction size further. For instance, an account is represented by an index rather than an address, which reduces a transaction from 32 bytes to just 4 bytes. Transactions are also written to Ethereum as *calldata*, reducing gas.