**snarkjs**

This is a **JavaScript and Pure Web Assembly implementation of zkSNARK and PLONK schemes.** It uses the Groth16 Protocol (3 point only and 3 pairings), PLONK and FFLONK.

This library includes all the tools required to perform trusted setup multi-party ceremonies: including the universal [*powers of tau*](https://medium.com/coinmonks/announcing-the-perpetual-powers-of-tau-ceremony-to-benefit-all-zk-snark-projects-c3da86af8377) ceremony, and the second phase circuit specific ceremonies.

Any zk-snark project can pick a round from the common phase 1 to start their circuit-specific phase 2 ceremony.

The formats used in this library for the multi-party computation are compatible with the ones used in [Semaphore's Perpetual Powers of Tau](https://github.com/weijiekoh/perpetualpowersoftau) and [other implementations](https://github.com/kobigurk/phase2-bn254).

This library uses the compiled circuits generated by the [circom](https://github.com/iden3/circom) compiler.

It works in [node.js](https://github.com/iden3/snarkjs#using-node) as well as directly in the [browser](https://github.com/iden3/snarkjs#in-the-browser).

It's an [ES module](https://hacks.mozilla.org/2018/03/es-modules-a-cartoon-deep-dive/), so it can be directly imported into bigger projects using [Rollup](https://rollupjs.org/guide/en/) or [Webpack](https://webpack.js.org/).

The low-level cryptography is performed directly in wasm, and uses worker threads to parallelize the computations. The result is a high performance library with benchmarks comparable to host implementations.

**Preliminaries**

**Install node**

First off, make sure you have a recent version of Node.js installed. While any version after v12 should work fine, we recommend you install v16 or later.

If you’re not sure which version of Node you have installed, you can run:

node -v

To download the latest version of Node, see [here](https://nodejs.org/en/download/).

**Install snarkjs**

To install snarkjs run:

npm install -g snarkjs@latest

If you're seeing an error, try prefixing both commands with sudo and running them again.

**Understand the help command**

To see a list of all snarkjs commands, as well as descriptions about their inputs and outputs, run:

snarkjs --help

You can also use the --help option with specific commands:

snarkjs groth16 prove --help

Most of the commands have an alternative shorter alias (which you can discover using --help).

For example, the previous command can also be invoked with:

snarkjs g16p --help

**Debugging tip**

If you a feel a command is taking longer than it should, re-run it with a -v or --verbose option to see more details about how it's progressing and where it's getting blocked.

**Install circom**

To install circom, follow the instructions at [installing circom](https://docs.circom.io/getting-started/installation).

**Guide**

**0. Create and move into a new directory**

mkdir snarkjs\_example

cd snarkjs\_example

**1. Start a new powers of tau ceremony**

snarkjs powersoftau new bn128 14 pot14\_0000.ptau -v

The new command is used to start a powers of tau ceremony.

The first parameter after new refers to the type of curve you wish to use. At the moment, we support both bn128 and bls12-381.

The second parameter, in this case 14, is the power of two of the maximum number of constraints that the ceremony can accept: in this case, the number of constraints is 2 ^ 14 = 16,384. The maximum value supported here is 28, which means you can use snarkjs to securely generate zk-snark parameters for circuits with up to 2 ^ 28 (≈268 million) constraints.

**2. Contribute to the ceremony**

snarkjs powersoftau contribute pot14\_0000.ptau pot14\_0001.ptau --name="First contribution" -v

The contribute command creates a ptau file with a new contribution.

You'll be prompted to enter some random text to provide an extra source of entropy.

contribute takes as input the transcript of the protocol so far, in this case pot14\_0000.ptau, and outputs a new transcript, in this case pot14\_0001.ptau, which includes the computation carried out by the new contributor (ptau files contain a history of all the challenges and responses that have taken place so far).

name can be anything you want, and is just included for reference (it will be printed when you verify the file (step 5).

**3. Provide a second contribution**

snarkjs powersoftau contribute pot14\_0001.ptau pot14\_0002.ptau --name="Second contribution" -v -e="some random text"

By letting you write the random text as part of the command, the -e parameter allows contribute to be non-interactive.

**4. Provide a third contribution using third party software**

snarkjs powersoftau export challenge pot14\_0002.ptau challenge\_0003

snarkjs powersoftau challenge contribute bn128 challenge\_0003 response\_0003 -e="some random text"

snarkjs powersoftau import response pot14\_0002.ptau response\_0003 pot14\_0003.ptau -n="Third contribution name"

The challenge and response files are compatible with [this software](https://github.com/kobigurk/phase2-bn254).

This allows you to use different types of software in a single ceremony.

**5. Verify the protocol so far**

snarkjs powersoftau verify pot14\_0003.ptau

The verify command verifies a ptau (powers of tau) file. Which means it checks all the contributions to the multi-party computation (MPC) up to that point. It also prints the hashes of all the intermediate results to the console.

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

[INFO] snarkJS: Powers Of tau file OK!

In sum, whenever a new zk-snark project needs to perform a trusted setup, you can just pick the latest ptau file, and run the verify command to verify the entire chain of challenges and responses so far.

**6. Apply a random beacon**

snarkjs powersoftau beacon pot14\_0003.ptau pot14\_beacon.ptau 0102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f 10 -n="Final Beacon"

The beacon command creates a ptau file with a contribution applied in the form of a random beacon.

We need to apply a random beacon in order to finalise phase 1 of the trusted setup.

To paraphrase Sean Bowe and Ariel Gabizon, a random beacon is a source of public randomness that is not available before a fixed time. The beacon itself can be a delayed hash function (e.g. 2^40 iterations of SHA256) evaluated on some high entropy and publicly available data. Possible sources of data include: the closing value of the stock market on a certain date in the future, the output of a selected set of national lotteries, or the value of a block at a particular height in one or more blockchains. E.g. the hash of the 11 millionth Ethereum block (which as of this writing is some 3 months in the future). See [here](https://eprint.iacr.org/2017/1050.pdf) for more on the importance of a random beacon.

For the purposes of this tutorial, the beacon is essentially a delayed hash function evaluated on 0102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f (in practice this value will be some form of high entropy and publicly available data of your choice). The next input -- in our case 10 -- just tells snarkjs to perform 2 ^ 10 iterations of this hash function.

Note that [security holds](https://eprint.iacr.org/2017/1050) even if an adversary has limited influence on the beacon.

**7. Prepare phase 2**

snarkjs powersoftau prepare phase2 pot14\_beacon.ptau pot14\_final.ptau -v

We're now ready to prepare phase 2 of the setup (the circuit-specific phase).

Under the hood, the prepare phase2 command calculates the encrypted evaluation of the Lagrange polynomials at tau for tau, alpha\*tau and beta\*tau. It takes the beacon ptau file we generated in the previous step, and outputs a final ptau file which will be used to generate the circuit proving and verification keys.

**NOTE**

Ptau files for bn128 with the peraperPhase2 54 contributions and a beacon, can be found here:

| **power** | **maxConstraints** | **file** | **hash** |
| --- | --- | --- | --- |
| 8 | 256 | [powersOfTau28\_hez\_final\_08.ptau](https://hermez.s3-eu-west-1.amazonaws.com/powersOfTau28_hez_final_08.ptau) | d6a8fb3a04feb600096c3b791f936a578c4e664d262e4aa24beed1b7a9a96aa5eb72864d628db247e9293384b74b36ffb52ca8d148d6e1b8b51e279fdf57b583 |
| 9 | 512 | [powersOfTau28\_hez\_final\_09.ptau](https://hermez.s3-eu-west-1.amazonaws.com/powersOfTau28_hez_final_09.ptau) | 94f108a80e81b5d932d8e8c9e8fd7f46cf32457e31462deeeef37af1b71c2c1b3c71fb0d9b59c654ec266b042735f50311f9fd1d4cadce47ab234ad163157cb5 |
| 10 | 1k | [powersOfTau28\_hez\_final\_10.ptau](https://hermez.s3-eu-west-1.amazonaws.com/powersOfTau28_hez_final_10.ptau) | 6cfeb8cda92453099d20120bdd0e8a5c4e7706c2da9a8f09ccc157ed2464d921fd0437fb70db42104769efd7d6f3c1f964bcf448c455eab6f6c7d863e88a5849 |
| 11 | 2k | [powersOfTau28\_hez\_final\_11.ptau](https://hermez.s3-eu-west-1.amazonaws.com/powersOfTau28_hez_final_11.ptau) | 47c282116b892e5ac92ca238578006e31a47e7c7e70f0baa8b687f0a5203e28ea07bbbec765a98dcd654bad618475d4661bfaec3bd9ad2ed12e7abc251d94d33 |
| 12 | 4k | [powersOfTau28\_hez\_final\_12.ptau](https://hermez.s3-eu-west-1.amazonaws.com/powersOfTau28_hez_final_12.ptau) | ded2694169b7b08e898f736d5de95af87c3f1a64594013351b1a796dbee393bd825f88f9468c84505ddd11eb0b1465ac9b43b9064aa8ec97f2b73e04758b8a4a |
| 13 | 8k | [powersOfTau28\_hez\_final\_13.ptau](https://hermez.s3-eu-west-1.amazonaws.com/powersOfTau28_hez_final_13.ptau) | 58efc8bf2834d04768a3d7ffcd8e1e23d461561729beaac4e3e7a47829a1c9066d5320241e124a1a8e8aa6c75be0ba66f65bc8239a0542ed38e11276f6fdb4d9 |
| 14 | 16k | [powersOfTau28\_hez\_final\_14.ptau](https://hermez.s3-eu-west-1.amazonaws.com/powersOfTau28_hez_final_14.ptau) | eeefbcf7c3803b523c94112023c7ff89558f9b8e0cf5d6cdcba3ade60f168af4a181c9c21774b94fbae6c90411995f7d854d02ebd93fb66043dbb06f17a831c1 |
| 15 | 32k | [powersOfTau28\_hez\_final\_15.ptau](https://hermez.s3-eu-west-1.amazonaws.com/powersOfTau28_hez_final_15.ptau) | 982372c867d229c236091f767e703253249a9b432c1710b4f326306bfa2428a17b06240359606cfe4d580b10a5a1f63fbed499527069c18ae17060472969ae6e |
| 16 | 64k | [powersOfTau28\_hez\_final\_16.ptau](https://hermez.s3-eu-west-1.amazonaws.com/powersOfTau28_hez_final_16.ptau) | 6a6277a2f74e1073601b4f9fed6e1e55226917efb0f0db8a07d98ab01df1ccf43eb0e8c3159432acd4960e2f29fe84a4198501fa54c8dad9e43297453efec125 |
| 17 | 128k | [powersOfTau28\_hez\_final\_17.ptau](https://hermez.s3-eu-west-1.amazonaws.com/powersOfTau28_hez_final_17.ptau) | 6247a3433948b35fbfae414fa5a9355bfb45f56efa7ab4929e669264a0258976741dfbe3288bfb49828e5df02c2e633df38d2245e30162ae7e3bcca5b8b49345 |
| 18 | 256k | [powersOfTau28\_hez\_final\_18.ptau](https://hermez.s3-eu-west-1.amazonaws.com/powersOfTau28_hez_final_18.ptau) | 7e6a9c2e5f05179ddfc923f38f917c9e6831d16922a902b0b4758b8e79c2ab8a81bb5f29952e16ee6c5067ed044d7857b5de120a90704c1d3b637fd94b95b13e |
| 19 | 512k | [powersOfTau28\_hez\_final\_19.ptau](https://hermez.s3-eu-west-1.amazonaws.com/powersOfTau28_hez_final_19.ptau) | bca9d8b04242f175189872c42ceaa21e2951e0f0f272a0cc54fc37193ff6648600eaf1c555c70cdedfaf9fb74927de7aa1d33dc1e2a7f1a50619484989da0887 |
| 20 | 1M | [powersOfTau28\_hez\_final\_20.ptau](https://hermez.s3-eu-west-1.amazonaws.com/powersOfTau28_hez_final_20.ptau) | 89a66eb5590a1c94e3f1ee0e72acf49b1669e050bb5f93c73b066b564dca4e0c7556a52b323178269d64af325d8fdddb33da3a27c34409b821de82aa2bf1a27b |
| 21 | 2M | [powersOfTau28\_hez\_final\_21.ptau](https://hermez.s3-eu-west-1.amazonaws.com/powersOfTau28_hez_final_21.ptau) | 9aef0573cef4ded9c4a75f148709056bf989f80dad96876aadeb6f1c6d062391f07a394a9e756d16f7eb233198d5b69407cca44594c763ab4a5b67ae73254678 |
| 22 | 4M | [powersOfTau28\_hez\_final\_22.ptau](https://hermez.s3-eu-west-1.amazonaws.com/powersOfTau28_hez_final_22.ptau) | 0d64f63dba1a6f11139df765cb690da69d9b2f469a1ddd0de5e4aa628abb28f787f04c6a5fb84a235ec5ea7f41d0548746653ecab0559add658a83502d1cb21b |
| 23 | 8M | [powersOfTau28\_hez\_final\_23.ptau](https://hermez.s3-eu-west-1.amazonaws.com/powersOfTau28_hez_final_23.ptau) | 3063a0bd81d68711197c8820a92466d51aeac93e915f5136d74f63c394ee6d88c5e8016231ea6580bec02e25d491f319d92e77f5c7f46a9caa8f3b53c0ea544f |
| 24 | 16M | [powersOfTau28\_hez\_final\_24.ptau](https://hermez.s3-eu-west-1.amazonaws.com/powersOfTau28_hez_final_24.ptau) | fa404d140d5819d39984833ca5ec3632cd4995f81e82db402371a4de7c2eae8687c62bc632a95b0c6aadba3fb02680a94e09174b7233ccd26d78baca2647c733 |
| 25 | 32M | [powersOfTau28\_hez\_final\_25.ptau](https://hermez.s3-eu-west-1.amazonaws.com/powersOfTau28_hez_final_25.ptau) | 0377d860cdb09a8a31ea1b0b8c04335614c8206357181573bf294c25d5ca7dff72387224fbd868897e6769f7805b3dab02854aec6d69d7492883b5e4e5f35eeb |
| 26 | 64M | [powersOfTau28\_hez\_final\_26.ptau](https://hermez.s3-eu-west-1.amazonaws.com/powersOfTau28_hez_final_26.ptau) | 418dee4a74b9592198bd8fd02ad1aea76f9cf3085f206dfd7d594c9e264ae919611b1459a1cc920c2f143417744ba9edd7b8d51e44be9452344a225ff7eead19 |
| 27 | 128M | [powersOfTau28\_hez\_final\_27.ptau](https://hermez.s3-eu-west-1.amazonaws.com/powersOfTau28_hez_final_27.ptau) | 10ffd99837c512ef99752436a54b9810d1ac8878d368fb4b806267bdd664b4abf276c9cd3c4b9039a1fa4315a0c326c0e8e9e8fe0eb588ffd4f9021bf7eae1a1 |
| 28 | 256M | [powersOfTau28\_hez\_final.ptau](https://hermez.s3-eu-west-1.amazonaws.com/powersOfTau28_hez_final.ptau) | 55c77ce8562366c91e7cda394cf7b7c15a06c12d8c905e8b36ba9cf5e13eb37d1a429c589e8eaba4c591bc4b88a0e2828745a53e170eac300236f5c1a326f41a |

There is a file truncated for each power of two.

The complete file is [powersOfTau28\_hez\_final.ptau](https://hermez.s3-eu-west-1.amazonaws.com/powersOfTau28_hez_final.ptau) which includes 2\*\*28 powers.

And it's blake2b hash is:

55c77ce8562366c91e7cda394cf7b7c15a06c12d8c905e8b36ba9cf5e13eb37d1a429c589e8eaba4c591bc4b88a0e2828745a53e170eac300236f5c1a326f41a

You can find more information about the ceremony [here](https://github.com/weijiekoh/perpetualpowersoftau)

The last ptau file was generated using this procedure:

<https://www.reddit.com/r/ethereum/comments/iftos6/powers_of_tau_selection_for_hermez_rollup/>

**8. Verify the final ptau**

snarkjs powersoftau verify pot14\_final.ptau

The verify command verifies a powers of tau file.

Before we go ahead and create the circuit, we perform a final check and verify the final protocol transcript.

Notice there is no longer a warning informing you that the file does not contain phase 2 precalculated values.

**9. Create the circuit**

cat <<EOT > circuit.circom

pragma circom 2.0.0;

template Multiplier(n) {

signal input a;

signal input b;

signal output c;

signal int[n];

int[0] <== a\*a + b;

for (var i=1; i<n; i++) {

int[i] <== int[i-1]\*int[i-1] + b;

}

c <== int[n-1];

}

component main = Multiplier(1000);

EOT

We create a circom file that allows us to easily test the system with a different number of constraints.

In this case, we've chosen 1000, but we can change this to anything we want (as long as the value we choose is below the number we defined in step 1).

**10. Compile the circuit**

circom circuit.circom --r1cs --wasm --sym

The circom command takes one input (the circuit to compile, in our case circuit.circom) and three options:

* r1cs: generates circuit.r1cs (the r1cs constraint system of the circuit in binary format).
* wasm: generates circuit.wasm (the wasm code to generate the witness – more on that later).
* sym: generates circuit.sym (a symbols file required for debugging and printing the constraint system in an annotated mode).

**11. View information about the circuit**

snarkjs r1cs info circuit.r1cs

The info command is used to print circuit stats.

You should see the following output:

[INFO] snarkJS: Curve: bn-128

[INFO] snarkJS: # of Wires: 1003

[INFO] snarkJS: # of Constraints: 1000

[INFO] snarkJS: # of Private Inputs: 2

[INFO] snarkJS: # of Public Inputs: 0

[INFO] snarkJS: # of Outputs: 1

This information fits with our mental map of the circuit we created: we had two private inputs a and b, one output c, and a thousand constraints of the form a \* b = c.

**12. Print the constraints**

snarkjs r1cs print circuit.r1cs circuit.sym

To double check, we print the constraints of the circuit.

You should see a thousand constraints of the form:

[ -main.int[i] ] \* [ main.int[i] ] - [ main.b -main.int[i+1] ] = 0

**13. Export r1cs to json**

snarkjs r1cs export json circuit.r1cs circuit.r1cs.json

cat circuit.r1cs.json

We export r1cs to json format to make it human readable.

**14. Calculate the witness**

First, we create a file with the inputs for our circuit:

cat <<EOT > input.json

{"a": 3, "b": 11}

EOT

Now, we use the Javascript/WASM program created by circom in the directory *circuit\_js* to create the witness (values of all the wires) for our inputs:

circuit\_js$ node generate\_witness.js circuit.wasm ../input.json ../witness.wtns

We can check if the generated witness complies with the r1cs file with the following command:

snarkjs wtns check circuit.r1cs witness.wtns

**15. Setup**

Currently, snarkjs supports 3 proving systems: Groth16, PLONK and FFLONK (Beta version).

Groth16 requires a trusted ceremony for each circuit. PLONK and FFLONK do not require it, it's enough with the powers of tau ceremony which is universal.

**Plonk**

snarkjs plonk setup circuit.r1cs pot14\_final.ptau circuit\_final.zkey

**Fflonk**

snarkjs fflonk setup circuit.r1cs pot14\_final.ptau circuit.zkey

You can jump directly to Section 21 as PLONK or FFLONK does not require a specific trusted ceremony.

**Groth16**

snarkjs groth16 setup circuit.r1cs pot14\_final.ptau circuit\_0000.zkey

This generates the reference zkey without phase 2 contributions

IMPORTANT: Do not use this zkey in production, as it's not safe. It requires at least a contribution,

The zkey new command creates an initial zkey file with zero contributions.

The zkey is a zero-knowledge key that includes both the proving and verification keys as well as phase 2 contributions.

Importantly, one can verify whether a zkey belongs to a specific circuit or not.

Note that circuit\_0000.zkey (the output of the zkey command above) does not include any contributions yet, so it cannot be used in a final circuit.

*The following steps (15-20) are similar to the equivalent phase 1 steps, except we use zkey instead of powersoftau as the main command, and we generate zkey rather that ptau files.*

**16. Contribute to the phase 2 ceremony**

snarkjs zkey contribute circuit\_0000.zkey circuit\_0001.zkey --name="1st Contributor Name" -v

The zkey contribute command creates a zkey file with a new contribution.

As in phase 1, you'll be prompted to enter some random text to provide an extra source of entropy.

**17. Provide a second contribution**

snarkjs zkey contribute circuit\_0001.zkey circuit\_0002.zkey --name="Second contribution Name" -v -e="Another random entropy"

We provide a second contribution.

**18. Provide a third contribution using third party software**

snarkjs zkey export bellman circuit\_0002.zkey challenge\_phase2\_0003

snarkjs zkey bellman contribute bn128 challenge\_phase2\_0003 response\_phase2\_0003 -e="some random text"

snarkjs zkey import bellman circuit\_0002.zkey response\_phase2\_0003 circuit\_0003.zkey -n="Third contribution name"

And a third using [third-party software](https://github.com/kobigurk/phase2-bn254).

**19. Verify the latest zkey**

snarkjs zkey verify circuit.r1cs pot14\_final.ptau circuit\_0003.zkey

The zkey verify command verifies a zkey file. It also prints the hashes of all the intermediary results to the console.

We verify the zkey file we created in the previous step. Which means we check all the contributions to the second phase of the multi-party computation (MPC) up to that point.

This command also checks that the zkey file matches the circuit.

If everything checks out, you should see the following:

[INFO] snarkJS: ZKey Ok!

**20. Apply a random beacon**

snarkjs zkey beacon circuit\_0003.zkey circuit\_final.zkey 0102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f 10 -n="Final Beacon phase2"

The zkey beacon command creates a zkey file with a contribution applied in the form of a random beacon.

We use it to apply a random beacon to the latest zkey after the final contribution has been made (this is necessary in order to generate a final zkey file and finalise phase 2 of the trusted setup).

**21. Verify the final zkey**

snarkjs zkey verify circuit.r1cs pot14\_final.ptau circuit\_final.zkey

Before we go ahead and export the verification key as a json, we perform a final check and verify the final protocol transcript (zkey).

**22. Export the verification key**

snarkjs zkey export verificationkey circuit\_final.zkey verification\_key.json

We export the verification key from circuit\_final.zkey into verification\_key.json.

**23. Create the proof**

**PLONK**

snarkjs plonk prove circuit\_final.zkey witness.wtns proof.json public.json

**FFLONK**

snarkjs fflonk prove circuit.zkey witness.wtns proof.json public.json

**Groth16**

snarkjs groth16 prove circuit\_final.zkey witness.wtns proof.json public.json

We create the proof. this command generates the files proof.json and public.json: proof.json contains the actual proof, whereas public.json contains the values of the public inputs and output.

Note that it's also possible to create the proof and calculate the witness in the same command by running:

snarkjs groth16 fullprove input.json circuit.wasm circuit\_final.zkey proof.json public.json

or

snarkjs plonk fullprove witness.json circuit.wasm circuit\_final.zkey proof.json public.json

or

snarkjs fflonk fullprove witness.json circuit.wasm circuit\_final.zkey proof.json public.json

> ```

**24. Verify the proof**

**PLONK**

snarkjs plonk verify verification\_key.json public.json proof.json

**FFLONK**

snarkjs fflonk verify verification\_key.json public.json proof.json

**Groth16**

snarkjs groth16 verify verification\_key.json public.json proof.json

We use the this command to verify the proof, passing in the verification\_key we exported earlier.

If all is well, you should see that OK has been outputted to your console. This signifies the proof is valid.

**25. Turn the verifier into a smart contract**

snarkjs zkey export solidityverifier circuit\_final.zkey verifier.sol

Finally, we export the verifier as a Solidity smart-contract so that we can publish it on-chain -- using [remix](https://remix.ethereum.org/) for example. For the details on how to do this, refer to section 4 of [this tutorial](https://blog.iden3.io/first-zk-proof.html).

**26. Simulate a verification call**

snarkjs zkey export soliditycalldata public.json proof.json

We use soliditycalldata to simulate a verification call, and cut and paste the result directly in the verifyProof field in the deployed smart contract in the remix environment.

And voila! That's all there is to it :)

**Using Node**

npm init

npm install snarkjs

const snarkjs = require("snarkjs");

const fs = require("fs");

async function run() {

const { proof, publicSignals } = await snarkjs.groth16.fullProve({a: 10, b: 21}, "circuit.wasm", "circuit\_final.zkey");

console.log("Proof: ");

console.log(JSON.stringify(proof, null, 1));

const vKey = JSON.parse(fs.readFileSync("verification\_key.json"));

const res = await snarkjs.groth16.verify(vKey, publicSignals, proof);

if (res === true) {

console.log("Verification OK");

} else {

console.log("Invalid proof");

}

}

run().then(() => {

process.exit(0);

});

**In the browser**

Load snarkjs.min.js and start using it as usual.

cp node\_modules/snarkjs/build/snarkjs.min.js .

<!doctype html>

<html>

<head>

<title>Snarkjs client example</title>

</head>

<body>

<h1>Snarkjs client example</h1>

<button id="bGenProof"> Create proof </button>

<!-- JS-generated output will be added here. -->

<pre class="proof"> Proof: <code id="proof"></code></pre>

<pre class="proof"> Result: <code id="result"></code></pre>

<script src="snarkjs.min.js"> </script>

<!-- This is the bundle generated by rollup.js -->

<script>

const proofComponent = document.getElementById('proof');

const resultComponent = document.getElementById('result');

const bGenProof = document.getElementById("bGenProof");

bGenProof.addEventListener("click", calculateProof);

async function calculateProof() {

const { proof, publicSignals } =

await snarkjs.groth16.fullProve( { a: 3, b: 11}, "circuit.wasm", "circuit\_final.zkey");

proofComponent.innerHTML = JSON.stringify(proof, null, 1);

const vkey = await fetch("verification\_key.json").then( function(res) {

return res.json();

});

const res = await snarkjs.groth16.verify(vkey, publicSignals, proof);

resultComponent.innerHTML = res;

}

</script>

</body>

</html>

**Security**

Avoid using versions before v0.4.16. They contain bugs that may affect the security.

**Further resources**

* [Announcing the Perpetual Powers of Tau Ceremony to benefit all zk-SNARK projects](https://medium.com/coinmonks/announcing-the-perpetual-powers-of-tau-ceremony-to-benefit-all-zk-snark-projects-c3da86af8377)
* [Scalable Multi-party Computation for zk-SNARK Parameters in the Random Beacon Model](https://eprint.iacr.org/2017/1050.pdf)
* [phase2-bn254](https://github.com/kobigurk/phase2-bn254)
* [Perpetual Powers of Tau](https://github.com/weijiekoh/perpetualpowersoftau)
* [Powers of Tau](https://github.com/ebfull/powersoftau)
* [Trusted setup ceremonies explored](https://www.zeroknowledge.fm/133)
* [Simple react projct using snarkjs](https://github.com/LHerskind/snarkjs-react)

**Final note**

We hope you enjoyed this quick walk-through. Please address any questions you may have to our [telegram group](https://t.me/iden3io) (it’s also a great way to join the community and stay up-to-date with the latest circom and snarkjs developments) 💙

**License**

snarkjs is part of the iden3 project copyright 2018 0KIMS association and published with GPL-3 license. Please check the COPYING file for more details.

<https://github.com/iden3/snarkjs>