

Lesson 11 - Risc Zero / Axiom / Circom

Today's topics

- Risc Zero
- Powdr
- Axiom
- Circom

Risc Zero

Introduction


The RISC Zero zkVM is an open-source, zero-knowledge virtual machine designed for constructing trustless, verifiable software applications.



Risc Zero's goal is to integrate existing programming languages and developer tools into the zero-knowledge realm. This is accomplished through a high-performance ZKP prover, which provides the performance allowance necessary to create a zero-knowledge virtual machine (zkVM) implementing a standard RISC-V instruction set.

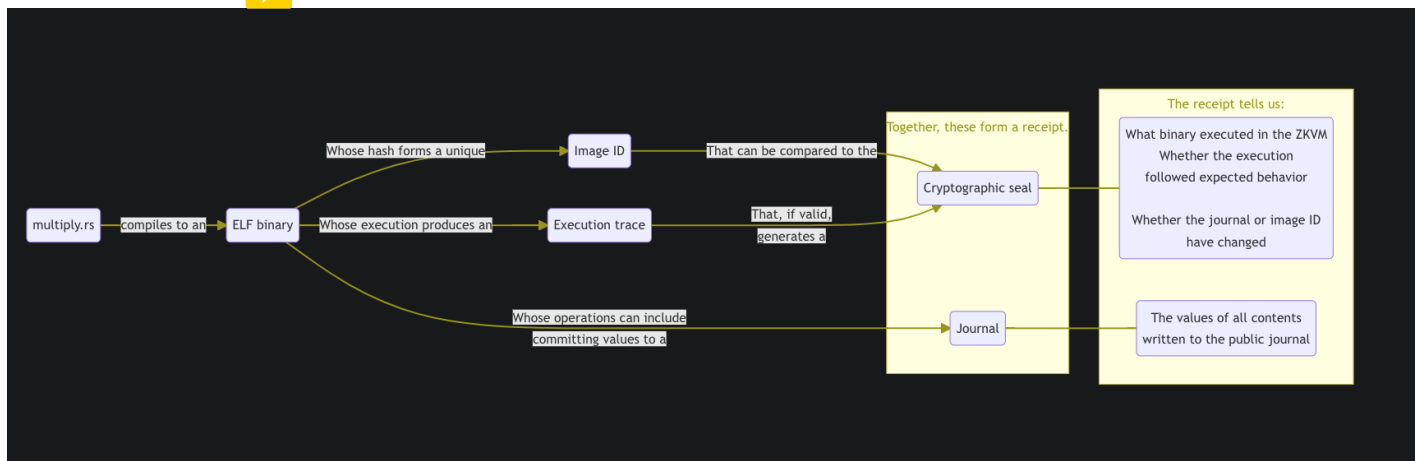
In practical terms, this allows for smooth integration between "host" application code, written in high-level languages running natively on host processors (e.g., Rust on arm64 Mac), and "guest" code in the same language executing within the zkVM.

A zero-knowledge virtual machine is a virtual machine that runs trusted code and generates proofs that authenticate the zkVM output.

RISC Zero's zkVM implementation, based on the RISC-V architecture, executes code and produces a computational receipt. 

Risc-V is an open standard instruction set architecture (ISA) based on established reduced instruction set computer (RISC) principles

In a RISC Zero zkVM program, guest code written for the zkVM is compiled to an ELF binary and executed by the `prover`, which returns a **computational receipt** to the **host program**. Anyone possessing a copy of this receipt can verify the program's execution and access its publicly shared outputs.



Before being executed on the zkVM, guest source code is converted into a RISC-V ELF binary. The binary file is hashed to create a `image ID` that uniquely identifies the binary being executed. The binary may include code instructions to publicly commit a value to the `journal`. Later, the journal contents can be read by anyone with the receipt.

After the binary is executed, an execution trace contains a complete record of zkVM operation. The trace is inspected and the ELF file's instructions are compared to the operations that were actually performed.

A valid trace means that the ELF file was faithfully executed according to the rules of the RISC-V instruction set architecture.

The execution trace and the journal are then used to generate a seal, a blob of cryptographic data that shows the receipt is valid.

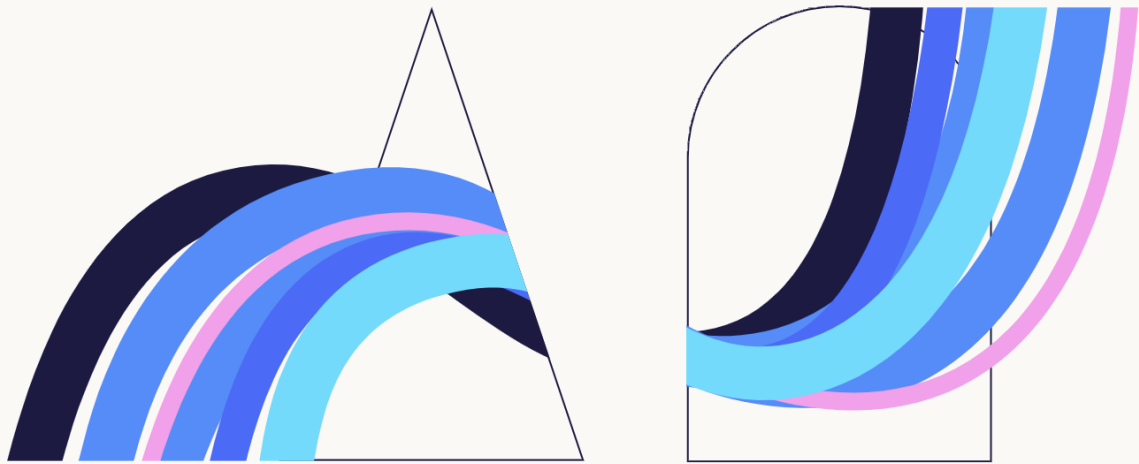


The seal has properties that reveal whether itself or the journal have been altered.

When the receipt is verified, the seal will be checked to confirm the validity of the receipt.

To check whether the correct binary was executed, the seal can be compared to the image ID of the expected ELF file.

powdr

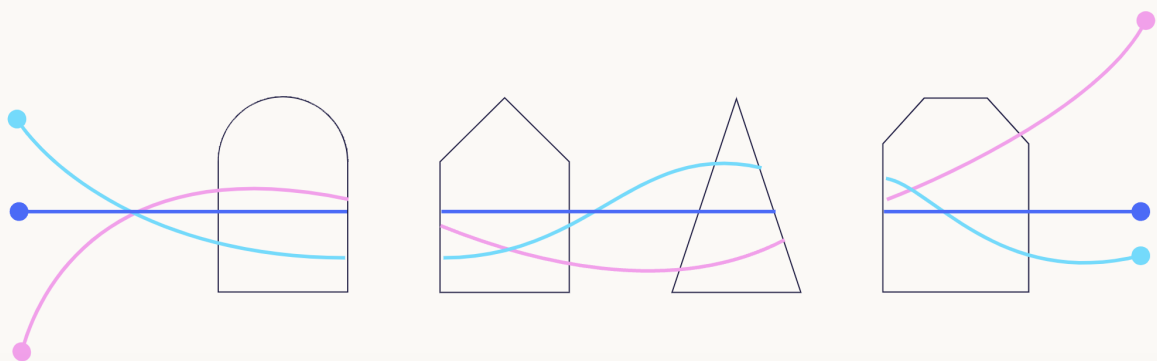


Powdr brings modularity, flexibility, security and excellent developer experience to zkVMs.

How it works




Design a new zkVM in hours, through a user-defined ISA, which powdr compiles into a zkVM.

Generate proofs using eSTARK, Halo2, Nova, and whatever comes next.



See [Docs](#)

powdr is a modular compiler stack to build zkVMs. It is ideal for implementing existing VMs and experimenting with new designs with minimal boilerplate.

- Domain specific languages are used to specify the VM and its underlying constraints, not low level Rust code
- Automated witness generation 
- Support for multiple provers as well as aggregation schemes 
- Support for hand-optimized co-processors when performance is critical
- Built in Rust 

See [video](#) from Christian Reitwiessner

Installation

Rust is a prerequisite

Instructions are [here](#)

Front Ends

A Risc V frontend is available and others are under development.

Backends

 [Halo 2](#) and [eSTARK](#) are supported



See [Demo](#)

Axiom is a ZK coprocessor designed for Ethereum. It allows smart contracts to access on-chain data in a trustless manner and perform various computations on that data. Developers can submit queries to Axiom and utilise the ZK-verified results directly in their smart contracts.

Axiom operates in three steps:

1. Read: Axiom employs ZK proofs to securely retrieve data from block headers, states, transactions, and receipts in past Ethereum blocks. Since all on-chain data is stored in one of these forms, Axiom can access any information available to archive nodes.
2. Compute: Once the data is obtained, Axiom applies verified compute operations on top of it. These operations range from basic analytics like sum, count, max, and min, to cryptographic tasks such as signature verification and key aggregation, as well as machine learning algorithms like decision trees, linear regression, and neural network inference. Each compute operation's validity is confirmed through a ZK proof.
3. Verify: Axiom provides a ZK validity proof with the result of each query, ensuring two things:
 - (1) the input data was accurately fetched from the chain, and
 - (2) the compute operations were correctly executed. This ZK proof is then verified on-chain within the Axiom smart contract, making the final result securely available for use by other smart contracts downstream.



CIRCOM & SNARKJS

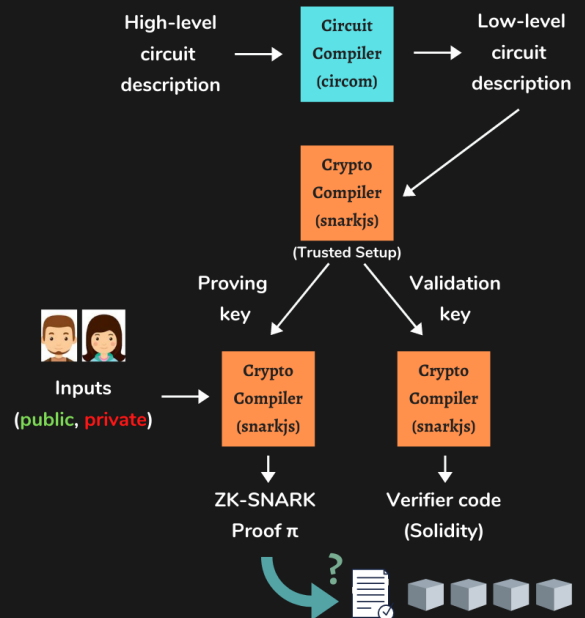
- 1 Design your arithmetic circuit and write your circuit using **circom**
 - Use your own code
 - Use our safe templates
- 2 Compile the circuit to get a low-level representation (R1CS)

```
$ circom circuit.circom --r1cs --wasm --sym
```
- 3 Use **snarkjs** to compute your witness

```
$ snarkjs calculatewitness --wasm circuit.wasm  
--input input.json --witness witness.json
```
- 4 Generate a trusted setup and get your zk-SNARK proof

```
$ snarkjs setup  
$ snarkjs proof
```
- 5 Validate your proof or have a smart-contract validate it!

```
$ snarkjs validate  
$ snarkjs generateverifier
```



Installation

Dependencies

- Rust - use rustup

```
curl --proto '=https' --tlsv1.2 https://sh.rustup.rs -sSf | sh
```

- Circom

```
git clone https://github.com/iden3/circom.git
```

```
cd circom
```

```
cargo build --release
```

```
cargo install --path circom
```

- Snarkjs

```
npm install -g snarkjs
```

Circom Lib

See [Repo](#)

CircomLib

Description

- This repository contains a library of circuit templates.
- All files are copyrighted under 2018 OKIMS association and part of the free software [circom](#) (Zero Knowledge Circuit Compiler).
- You can read more about the circom language in [the circom documentation webpage](#).

Organisation

This repository contains 5 folders:

- `circuits` : it contains the implementation of different cryptographic primitives in circom language.
- `calcpedersenbases` : set of functions in JavaScript used to find a set of points in [Baby Jubjub](#) elliptic curve that serve as basis for the [Pedersen Hash](#).
- `doc` : it contains some circuit schemes in ASCII (must be opened with Monodraw, an ASCII art editor for Mac).
- `src` : it contains similar implementation of circuits in JavaScript.
- `test` : tests.

A description of the specific circuit templates for the `circuit` folder will be soon updated.

Circom coding

Writing circuits

Taken from the [documentation](#)

Circom allows programmers to define the [constraints](#) that define the arithmetic circuit. All constraints must be of the form $A * B + C = 0$, where A , B and C are linear combinations of signals.

You can define constraints in this way

```
pragma circom 2.0.0;

/*This circuit template checks that c is the multiplication of a and b.*/

template Multiplier2 () {

    // Declaration of signals.
    signal input a;
    signal input b;
    signal output c;

    // Constraints.
    c <== a * b;
}
```

Signals

The arithmetic circuits built using circom operate on signals

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.

Signals are by default private.

The programmer can distinguish between public and private signals only when defining the main component, by providing the list of public input signals.

```
pragma circom 2.0.0;

template Multiplier2(){
    //Declaration of signals
    signal input in1;
    signal input in2;
    signal output out;
    out<== in1 * in2;
```



```
}
```

```
component main {public [in1,in2]} = Multiplier2();
```

Circom data types

- Field Element
Integers mod the max field value, these are the default type.
- Arrays
These hold items of the same type

```
var x[3] = [2,8,4];  
var z[n+1]; // where n is a parameter of a template  
var dbl[16][2] = base;  
var y[5] = someFunction(n);
```

Templates and components

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.

```
template tempid ( param_1, ... , param_n ) {  
  signal input a;  
  signal output b;  
  
  .....  
  
}
```

The instantiation of a template is made using the keyword component and by providing the necessary parameters.

```
component c = tempid(v1,...,vn);
```

The values of the parameters should be known constants at compile time.

Components

A component defines an arithmetic circuit has input signals, output signals and intermediate signals, and can have a set of constraints.

Components are immutable once instantiated.

```
template A(N){
    signal input in;
    signal output out;
    out <== in;
}

template C(N){
    signal output out;
    out <== N;
}

template B(N){
    signal output out;
    component a;
    if(N > 0){
        a = A(N);
    }
    else{
        a = A(0);
    }
}

component main = B(1);
```

We can create arrays of components.

```
template MultiAND(n) {
    signal input in[n];
    signal output out;
    component and;
    component ands[2];
    var i;
    if (n==1) {
        out <== in[0];
    } else if (n==2) {
        and = AND();
        and.a <== in[0];
        and.b <== in[1];
        out <== and.out;
    }
}
```

```
} else {  
    and = AND();  
    var n1 = n\2;  
    var n2 = n-n\2;  
    ands[0] = MultiAND(n1);  
    ands[1] = MultiAND(n2);  
    for (i=0; i<n1; i++) ands[0].in[i] <= in[i];  
    for (i=0; i<n2; i++) ands[1].in[i] <= in[n1+i];  
    and.a <= ands[0].out;  
    and.b <= ands[1].out;  
    out <= and.out;  
}  
}
```

The main component

In order to start the execution, an initial component has to be given. By default, the name of this component is “main”, and hence the component main needs to be instantiated with some template.

This is a special initial component needed to create a circuit and it defines the global input and output signals of a circuit. For this reason, compared to the other components, it has a special attribute: the list of public input signals. The syntax of the creation of the main component is:

```
component main {public [signal_list]} = tempid(v1,...,vn);
```

```
pragma circom 2.0.0;
```

```
template A(){  
    signal input in1;  
    signal input in2;  
    signal output out;  
    out <== in1 * in2;  
}
```

```
component main {public [in1]}= A();
```

Useful Tool

zkRepl. from 0xPARC



REPL for circom

main.circom x + Add File

```
1 pragma circom 2.1.2;
2
3 include "circomlib/poseidon.circom";
4 // include "https://github.com/0xPARC/circom-secp256k1/blob/master/circuits/bigint.circom";
5
6 template Example () {
7   signal input a;
8   // The value assigned to 'unused' here is never read.
9   // View Problem (⌘F8) No quick fixes available
10  var unused = 4;
11  c <== a * b;
12  assert(a > 2);
13
14  component hash = Poseidon(2);
15  hash.inputs[0] <== a;
16  hash.inputs[1] <== b;
17
18  log("hash", hash.out);
19 }
20
21
22 component main { public [ a ] } = Example();
23
24 /* INPUT = {
25   "a": "5",
26   "b": "77"
27 } */
```

SHIFT-ENTER TO RUN
CMD-S TO SAVE AS GITHUB GIST

STDOUT:

template instances: 69
non-linear constraints: 241
linear constraints: 0
public inputs: 1
public outputs: 1
private inputs: 1
private outputs: 0
wires: 244
labels: 1111
Written successfully: ./main.r1cs
Written successfully: ./main.sym
Written successfully: ./main.js/main.wasm
Everything went okay, circom safe
Compiled in 3.17s

LOG:

hash 6008246173323011098915936938805752727781568490
715388424063708882447636047656

OUTPUT:

c = 385

ARTIFACTS:

Finished in 3.45s

- main.wasm (1027.06KB)
- main.js (9.18KB)
- main.wtns (7.88KB)
- main.r1cs (110.08KB)
- main.sym (37.71KB)

PLONK KEYS:

- main.plonk.zkey (6996.84KB)
- main.plonk.vkey.json (2.14KB)
- main.plonk.sol (25.63KB)
- main.plonk.html (11111.18KB)

KEYS + SOLIDITY + HTML:

Groth16 PLONK Verify

Circom -> Cairo

See [repo](#) and take note of the caveats

Allows circom projects to be verified on Ethereum by exporting to Cairo

First write and compile a circuit and compute the witness through circom, then generate a validation key through snarkjs (this process is properly explained at <https://docs.circom.io/getting-started/installation/>), this will yield a .zkey, which we can use to generate a solidity verifier through the command:

```
snarkjs zkey export solidityverifier [name of your key].zkey [nme of the verifier produced]
```

