Building opcode compatible zk EVMs

Guest Lecturer: **Jordi Baylina**

Zero Knowledge Proofs

Instructors: Dan Boneh, Shafi Goldwasser, Dawn Song, Justin Thaler, Yupeng Zhang

Stanford University

Berkeley
UNIVERSITY OF CALIFORNIA

GEORGETOWN UNIVERSITY

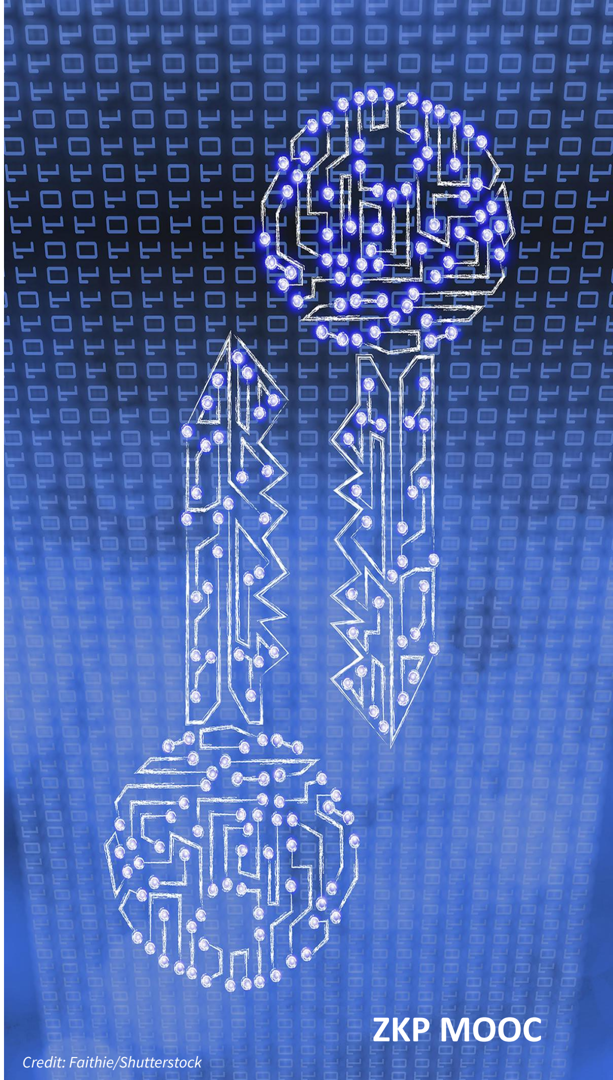
TEXAS A&M UNIVERSITY

Section 1

Introduction to Zero Knowledge Rollups

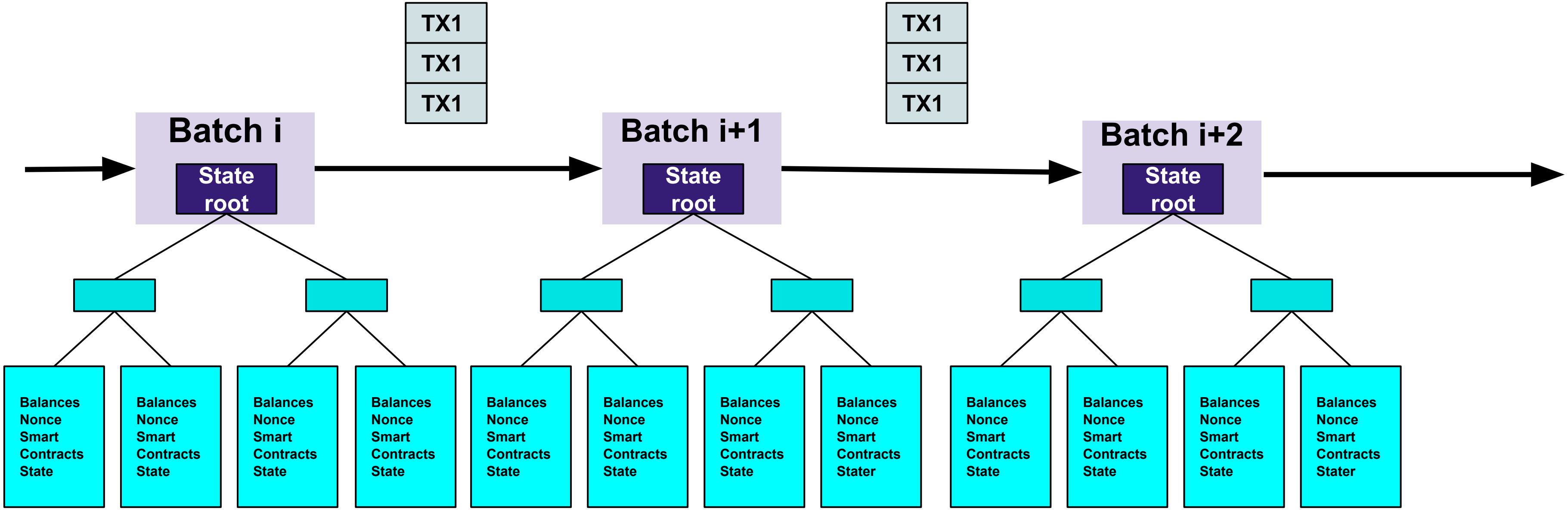
Learning objective:

- understand the concept of rollups
- how they can help achieve scalability
- the components of a zero knowledge rollup

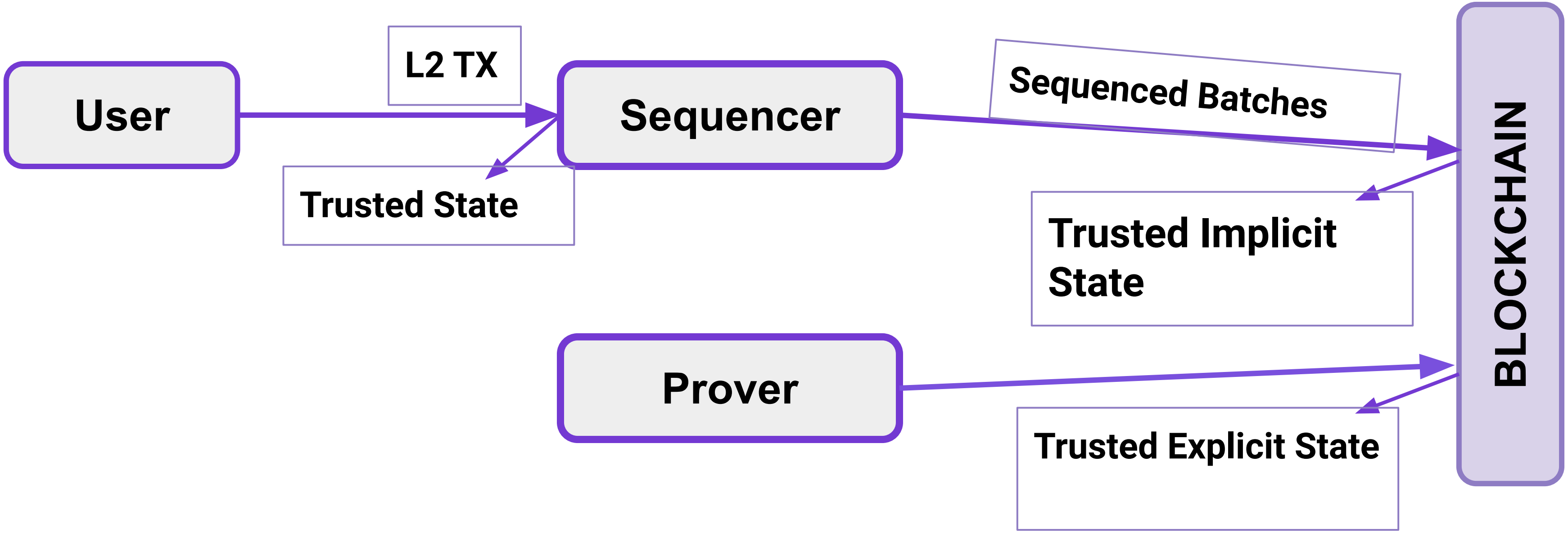


Credit: Faithie/Shutterstock

Rollup Scalability General Idea



Introduction to zkRollup: the Concept



User

L2 TX

Sequencer

Trusted State

Sequenced Batches

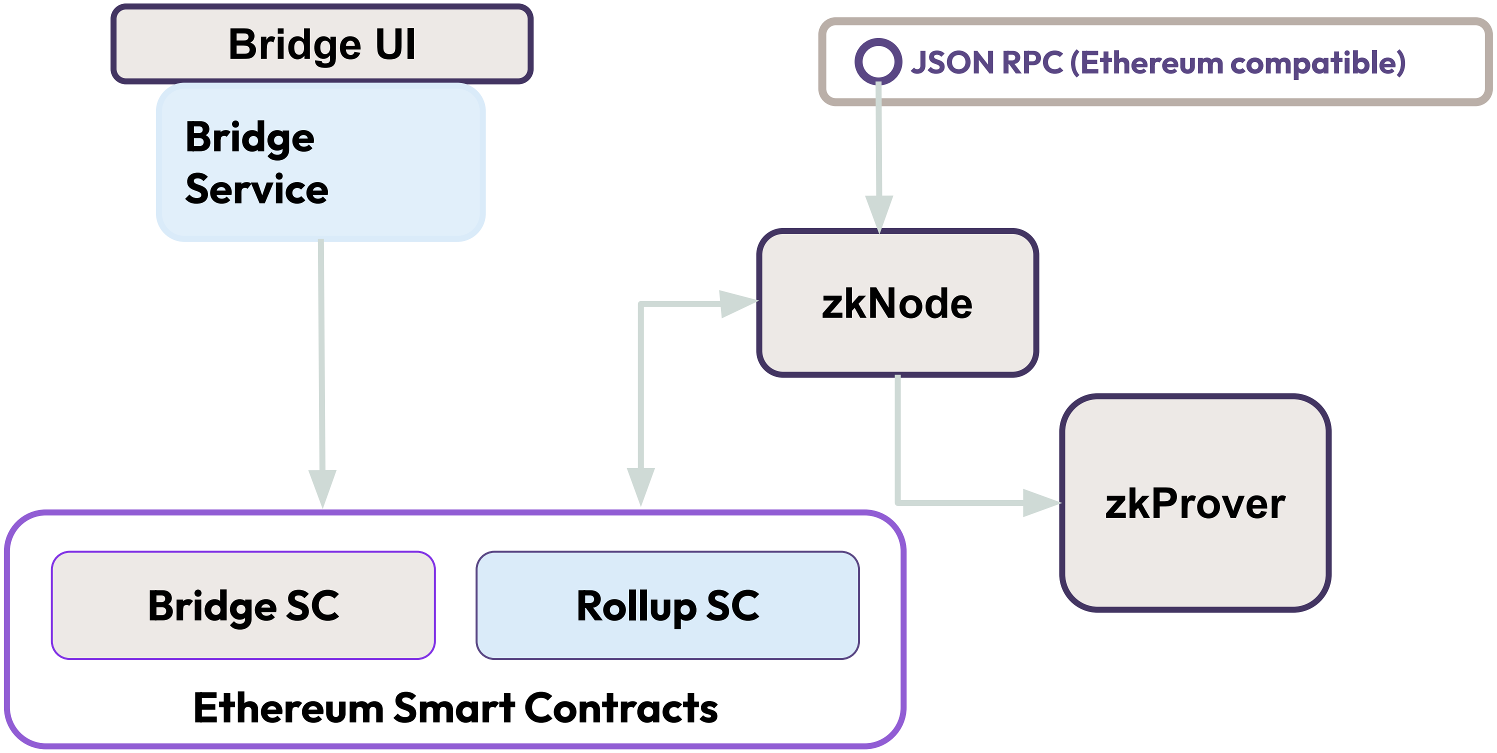
Trusted Implicit State

Prover

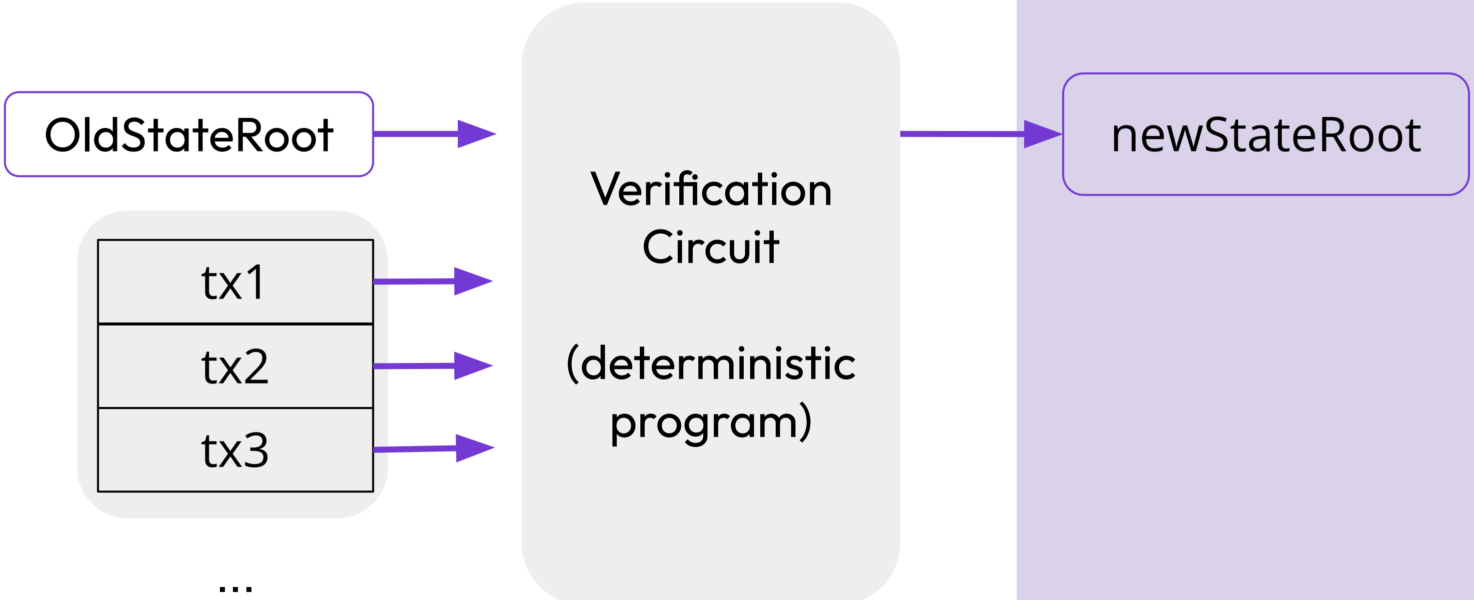
Trusted Explicit State

BLOCKCHAIN

zkRollups Components



zkEVM Prover



OldStateRoot

tx1

tx2

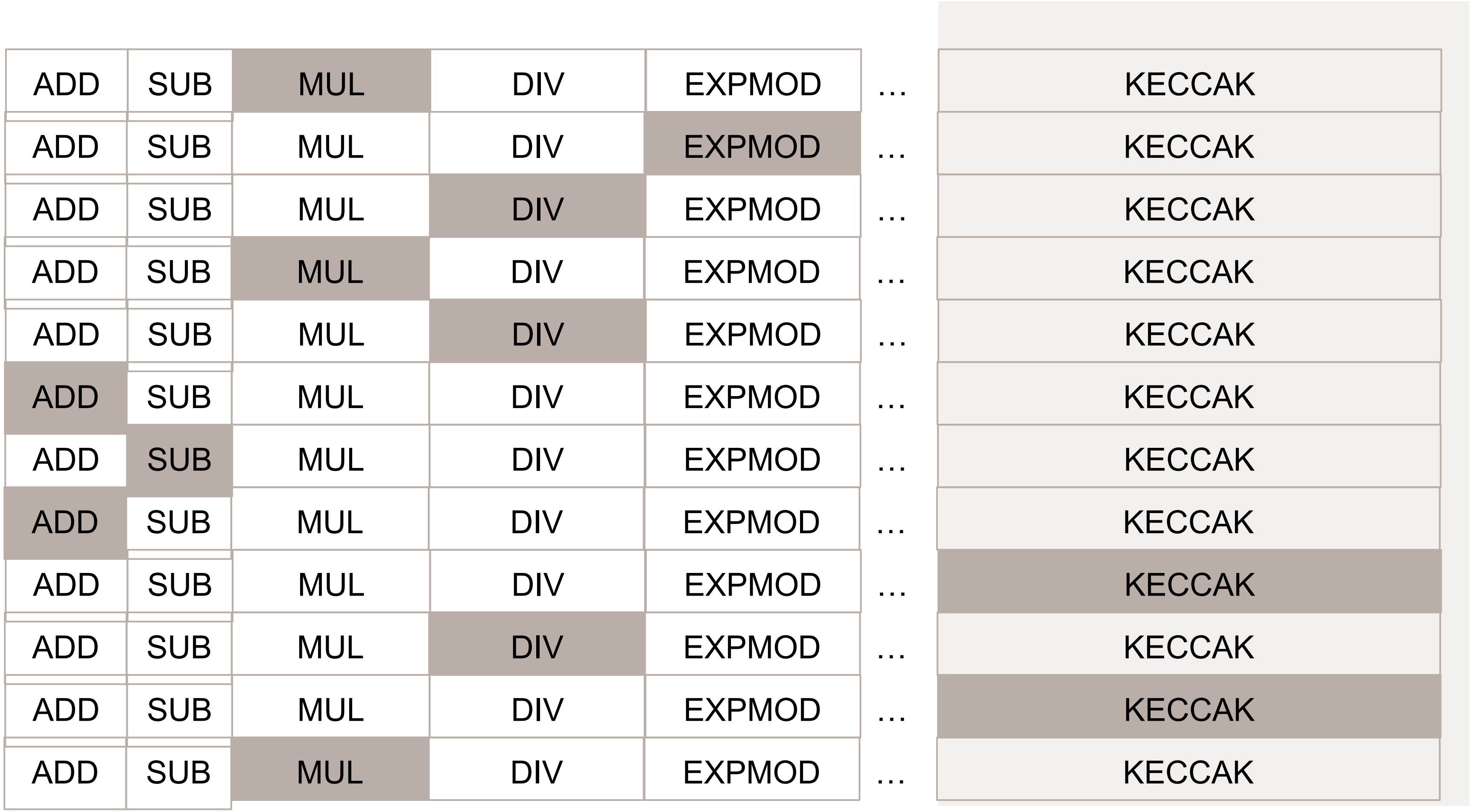
tx3

...

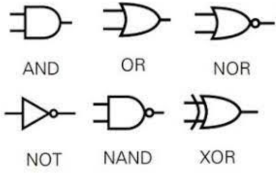
Verification Circuit

(deterministic program)

newStateRoot

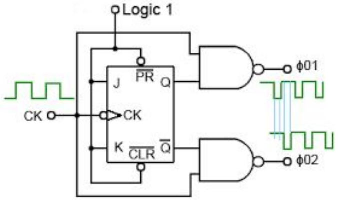


ADD	SUB	MUL	DIV	EXPMOD	…	KECCAK
ADD	SUB	MUL	DIV	EXPMOD	…	KECCAK
ADD	SUB	MUL	DIV	EXPMOD	…	KECCAK
ADD	SUB	MUL	DIV	EXPMOD	…	KECCAK
ADD	SUB	MUL	DIV	EXPMOD	…	KECCAK
ADD	SUB	MUL	DIV	EXPMOD	…	KECCAK
ADD	SUB	MUL	DIV	EXPMOD	…	KECCAK
ADD	SUB	MUL	DIV	EXPMOD	…	KECCAK
ADD	SUB	MUL	DIV	EXPMOD	…	KECCAK
ADD	SUB	MUL	DIV	EXPMOD	…	KECCAK
ADD	SUB	MUL	DIV	EXPMOD	…	KECCAK
ADD	SUB	MUL	DIV	EXPMOD	…	KECCAK



R1CS

circom



Polynomial Identities/ State Machines

PIL

Polynomial
Identity
Language

```
.MODEL SMALL
.STACK 64
.DATA
.CODE

START: MOV AX, @DATA
       MOV DX,AX
       MOV AL,01H
       MOV BL,02H
       ADD AL,BL
       MOV SI,3000H
       MOV [SI], AL
       MOV AH,4CH
       INT 21H
       END START
       .END
```

zkASM

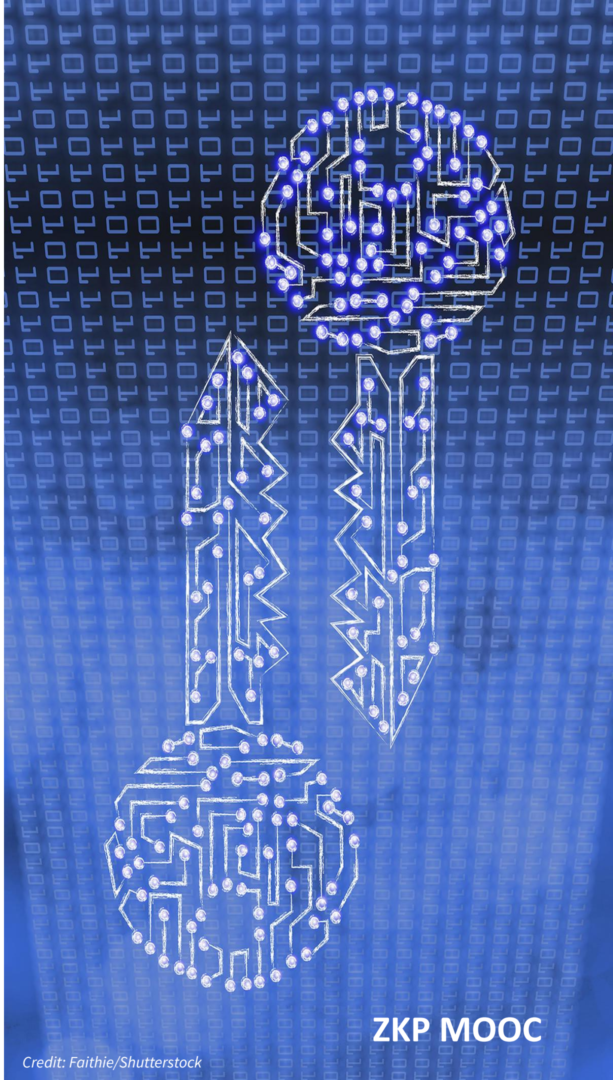
polygon zkEVM

Section 2

Introduction to Polynomial Identity Language (PIL)

Learning objective:

- Overview of PIL and why it is important
- Creating circuits with polynomials
- Introduction to state machines



Credit: Faithie/Shutterstock

Hello World: Fibonacci Series

$$\mathbb{F}_{\texttt{0xffffffff00000001}}$$

a_0	a_1	a_2	a_3	a_4		**a_{1024}**
1	2	3	5	8	……	**11696381471667068125**

$a_{i+1} = a_{i-1} + a_i$

fibonacci.circom

```
pragma circom 2.0.6;

template Fibonacci(n) {
   signal input a0;
   signal input a1;
   signal output out;

   signal im[n-1];

   for (var i=0; i<n-1; i++) {
       if (i==0) {
           im[i] <== a0 + a1;
       } else if (i==1) {
           im[i] <== a1 + im[0];
       } else {
           im[i] <== im[i-2] + im[i-1];
       }
   }
   out <== im[n-2];
}
component main = Fibonacci(1024);
```

Polynomial Identities State Machine

x	ISLAST(x)	aBeforeLast(x)	aLast(x)
1	0	1	2
ω	0	2	3
ω^2	0	3	5
ω^3	0	5	8
	…	…	…
ω^{1022}	0	16804231586740408223	1333889395434124223
ω^{1023}	1	13338893954341244223	11696381471667068125

$\text{aBeforeLast}(\omega x) = \text{aLast}(x)$
$\text{aLast}(\omega x) = \text{aBeforeLast}(x) + \text{aLast}(x)$

hello world

fibonacci.pil

```
constant %N = 1024;

namespace Fibonacci(%N);
   pol constant ISLAST;             // 0,0,0,0,.....,1
   pol commit aBeforeLast, aLast;

   (1-ISLAST) * (aBeforeLast' - aLast) = 0;
   (1-ISLAST) * (aLast' - (aBeforeLast + aLast)) = 0;

   public out = aLast(%N-1);
   ISLAST *( aLast- :out) = 0;
```

fibonacci.js

```
const { FGL } = require("pil-stark");

module.exports.buildConstants = async function
(pols) {

   const N = pols.ISLAST.length;

   for ( let i=0; i<N; i++) {
       pols.ISLAST[i] = (i == N-1) ? 1n : 0n;
   }
}

module.exports.execute = async function (pols,
input) {

   const N = pols.aLast.length;

   pols.aBeforeLast[0] = BigInt(input[0]);
   pols.aLast[0] = BigInt(input[1]);

   for (let i=1; i<N; i++) {
       pols.aBeforeLast[i] =pols.aLast[i-1];
       pols.aLast[i] = FGL.add(
            pols.aBeforeLast[i-1],
            pols.aLast[i-1]
      );
   }
   return pols.aLast[N-1];
}
```

fibonacci.test.js

```
const assert = require("assert");
const path = require("path");
const { FGL, starkSetup, starkGen, starkVerify } =
       require("pil-stark");
const { newConstantPolsArray, newCommitPolsArray,
       compile, verifyPil } = require("pilcom");
const smFibonacci = require("../src/fibonacci.js");

describe("test fibonacci sm", async function () {
   this.timeout(10000000);
   let constPols, cmPols, pil;

   it("It should create the pols main", async () => {
       pil = await compile(
          FGL, path.join(__dirname, "../src/fibonacci.pil"));

       constPols =  newConstantPolsArray(pil);

       await smFibonacci.buildConstants(
          constPols.Fibonacci);

       cmPols = newCommitPolsArray(pil);

       const result = await smFibonacci.execute(
          cmPols.Fibonacci, [1,2]);

       console.log("Result: " + result);

       const res = await verifyPil(
             FGL, pil, cmPols, constPols);
       assert(res.length == 0);
   });
```

```
   it("It should generate and verify the stark", async () => {
       const starkStruct = {
           nBits: 10,
           nBitsExt: 14,
           nQueries: 32,
           verificationHashType : "GL",
           steps: [
               {nBits: 14},
               {nBits: 9},
               {nBits: 4}
           ]
       };
       const setup = await starkSetup(
             constPols,
             pil,
             starkStruct
       );
       const resP = await starkGen(
                    cmPols,
                    constPols,
                    setup.constTree,
                    setup.starkInfo
       );
       const resV = await starkVerify(
                    resP.proof,
                    resP.publics,
                    setup.constRoot,
                    setup.starkInfo
       );
       assert(resV==true);
   });
});
```

Permutation Checks

x	a(x)	b(x)
1	3	1
ω	2	2
ω^2	6	3
ω^3	5	4
ω^4	4	5
ω^5	8	6
ω^6	7	7
ω^7	1	8

```
namespace PermutationExample(%N);
   pol commit a, b;

   a is b;
```

Higher Complexity Permutation Checks

x	selA(x)	a1(x)	a2(x)	SELB(x)	B1(x)	B2(x)
1	1	3	333	1	1	111
ω	1	2	222	1	2	222
ω^2	0			1	3	333
ω^3	0			1	4	444
ω^4	1	4	444	0		
ω^5	0			0		
ω^6	0			0		
ω^7	1	1	111	0		

```
namespace PermutationExample(%N);
   pol constant SELB, B1, B2;
   pol commit selA, a1, a2;

   selA { a1, a2 } is SELB { B1, B2 };
```

Plookup

x	a(x)	b(x)
1	3	1
ω	3	2
ω^2	6	3
ω^3	5	4
ω^4	6	5
ω^5	6	6
ω^6	1	7
ω^7	1	8

```
namespace PlookupExample(%N);
   pol commit a, b;

   a in b;
```

Higher Complexity Plookup

x	selA(x)	a1(x)	a2(x)	SELB(x)	B1(x)	B2(x)
1	1	3	333	1	1	111
ω	1	3	333	1	2	222
ω^2	0			1	3	333
ω^3	0			1	4	444
ω^4	1	4	444	0		
ω^5	0			0		
ω^6	0			0		
ω^7	1	1	111	0		

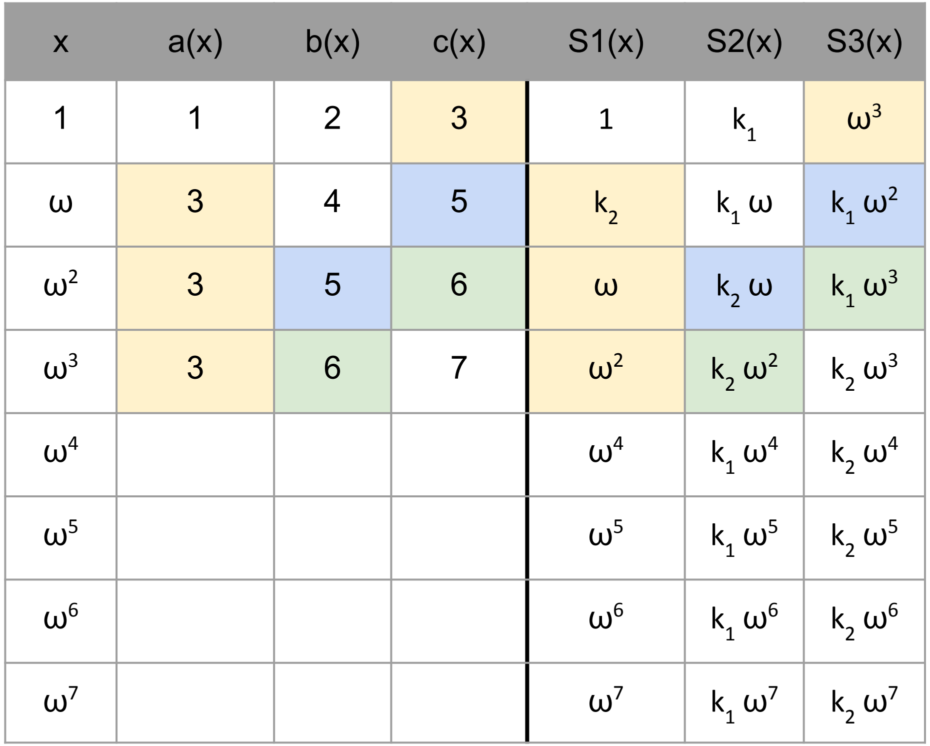
```
namespace PlookupExample(%N);
   pol constant SELB, B1, B2;
   pol commit selA, a1, a2;

   selA { a1, a2 } in SELB { B1, B2 };
```

Connection Checks

x	a(x)	S(x)
1	3	ω^5
ω	66	ω^6
ω^2	1833	ω^7
ω^3	3	1
ω^4	3	ω^3
ω^5	3	ω^4
ω^6	66	ω
ω^7	1833	ω^2

```
namespace ConnectionExample(%N);
   pol constant S;
   pol commit a;

   a connect S;
```

Higher Complexity Connection Checks



x	a(x)	b(x)	c(x)	S1(x)	S2(x)	S3(x)
1	1	2	3	1	k_1	ω^3
ω	3	4	5	k_2	$k_1 \omega$	$k_1 \omega^2$
ω^2	3	5	6	ω	$k_2 \omega$	$k_1 \omega^3$
ω^3	3	6	7	ω^2	$k_2 \omega^2$	$k_2 \omega^3$
ω^4				ω^4	$k_1 \omega^4$	$k_2 \omega^4$
ω^5				ω^5	$k_1 \omega^5$	$k_2 \omega^5$
ω^6				ω^6	$k_1 \omega^6$	$k_2 \omega^6$
ω^7				ω^7	$k_1 \omega^7$	$k_2 \omega^7$

```
namespace PermutationExample(%N);
   pol constant SELB, B1, B2;
   pol commit selA, a1, a2;

   { a1, a2, a3 } connect { S1, S2, S3 };
```

Plonk Example

```
// Plonk circuit

namespace main;

    pol committed a, b, c
    pol constant Sa, Sb, Sc;
    pol constant Ql, Qr, Qm, Qo, Qc;
    pol constant L1;                       // 1, 0, 0, …

    public publicInput = a(0);

    {a, b, c} connect {Sa, Sb, Sc};

    pol ab = a*b;
    Ql*a + Qr*b + Qo*c + Qm*ab + Qc = 0;

    L1 * (a - :publicInput) = 0;
```

Custom Gates in CIRCOM

```
pragma circom 2.0.6;
pragma custom_templates;

template custom CMul() {
   signal input ina[3];
   signal input inb[3];
   signal output out[3];

   var A = (ina[0] + ina[1])  * (inb[0] + inb[1]);
   var B = (ina[0] + ina[2])  * (inb[0] + inb[2]);
   var C = (ina[1] + ina[2])  * (inb[1] + inb[2]);
   var D = ina[0]*inb[0];
   var E = ina[1]*inb[1];
   var F = ina[2]*inb[2];
   var G = D-E;

   out[0] <-- C+G-F;
   out[1] <-- A+C-E-E-D;
   out[2] <-- B-G;
}
```

Custom Gates in CIRCOM

- Can be used as normal templates.
- Witness calculator is generated by CIRCOM like any other template.
- No constraints are allowed.
- All the custom gates are exported to the .r1cs file.
- This allows to do a circuit in circom and proof/verify it with a STARK!
- Supported primes in circom: BN128, BLS-12381, Goldilocks

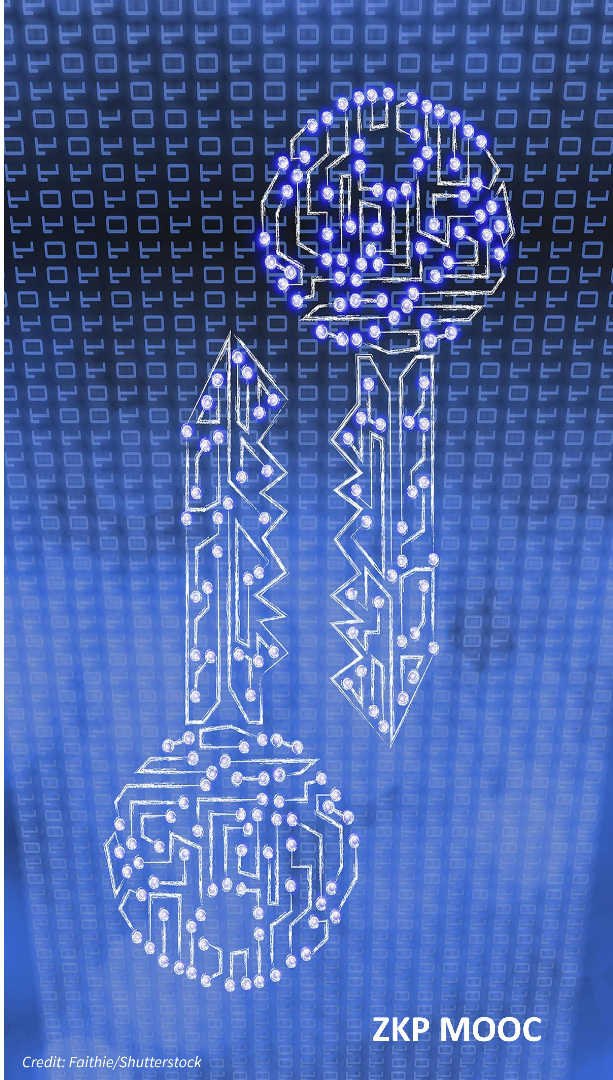
Plonk Example

```
// Plonk circuit

namespace main;

    pol committed a, b, c
    pol constant Sa, Sb, Sc;
    pol constant Ql, Qr, Qm, Qo, Qc;
    pol constant L1;                       // 1, 0, 0, …

    public publicInput = a(0);

    {a, b, c} connect {Sa, Sb, Sc};

    pol ab = a*b;
    Ql*a + Qr*b + Qo*c + Qm*ab + Qc = 0;

    L1 * (a - :publicInput) = 0;
```

Section 3

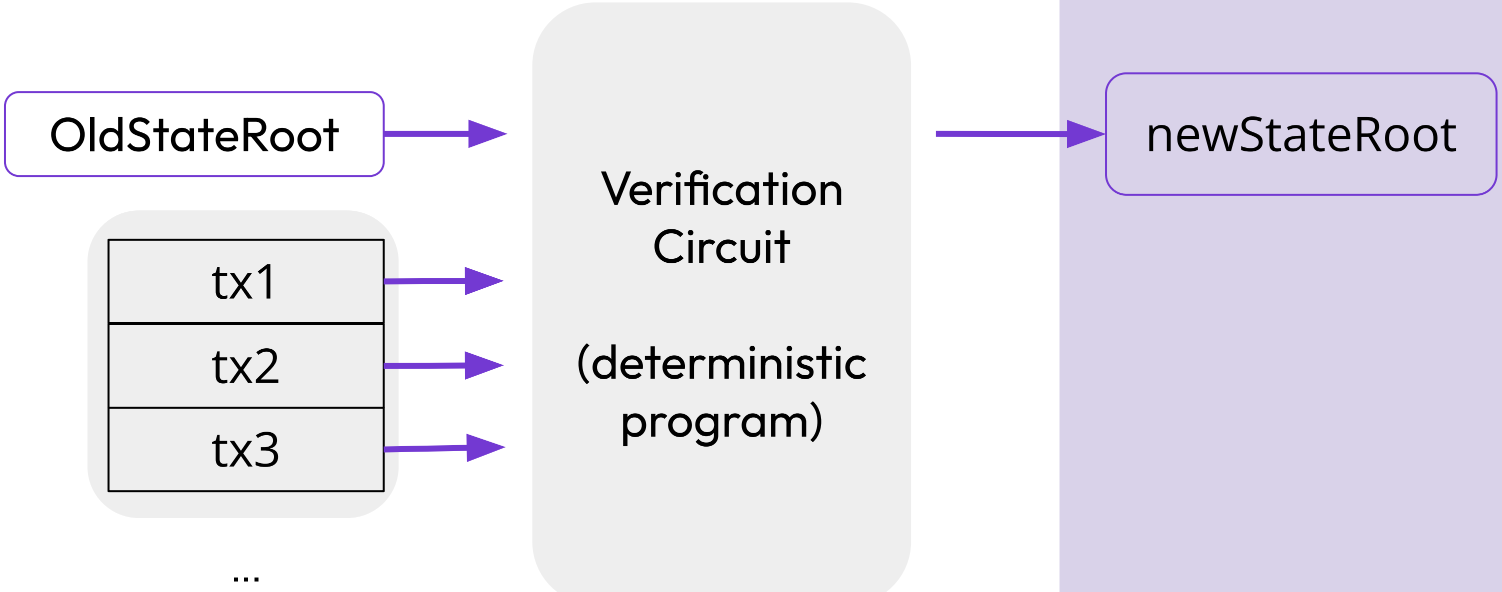
Prover Architecture

Learning objective: how to use PIL to build a processor



Credit: Faithie/Shutterstock

zkEVM Prover



OldStateRoot

tx1

tx2

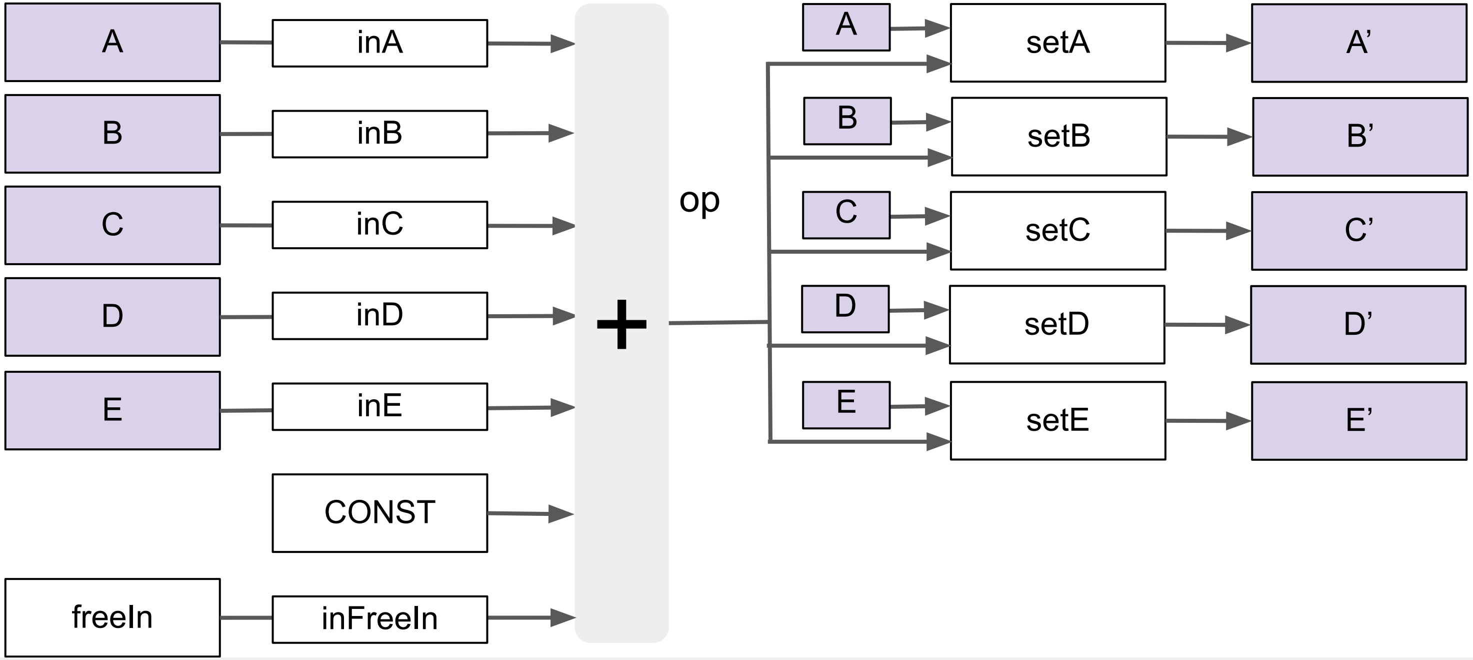
tx3

…

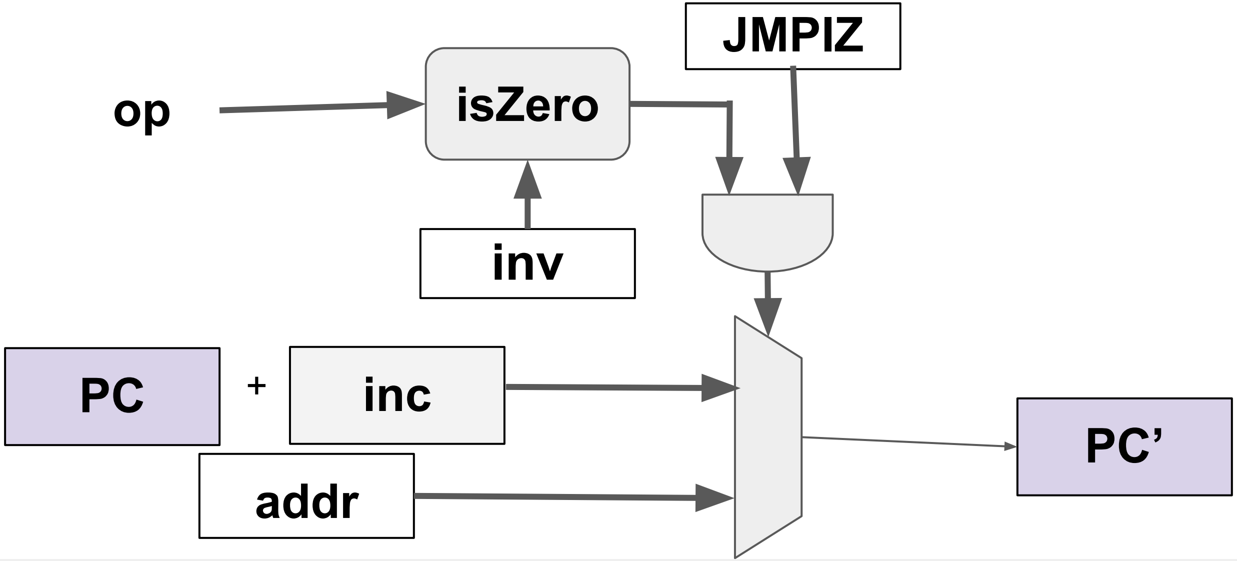
Verification Circuit

(deterministic program)

newStateRoot



```
pol op = A*inA + B*inB + C*inC + D*inD + E*inE + freeIn*inFreeIn + CONST;
A’ = (op - A) * setA + A;
B’ = (op - B) * setB + B;
C’ = (op - C) * setC + C;
D’ = (op - D) * setD + D;
E’ = (op - E) * setE + E;
```



```
pol commit inv;
pol isZero = 1 -op * inv;
isZero * op = 0;
pol jmp = JMPIZ*isZero;

PC’ = jmp * (addr - (PC+INC)) + (PC+INC);
```

Execution Trace

PROGRAM COUNTER REGISTER	INSTRUCTION
0	ADD
1	JMP 5
5	MUL
6	JMP 5
5	MUL
6	JMP 5

$\subset$

ROM

PROGRAM LINE	INSTRUCTION
0	ADD
1	JMP 5
2	ADD
3	ADD
5	MUL
6	JMP 5

Execution Trace

COUNT	INSTRUCTION	ADDR	VALUE
0			
1	WR	5	2
2	WR	3	8
3	RD	5	2
4			
5	RD	5	2
6	RD	3	8
7	WR	5	34
8			
9	RD	5	34

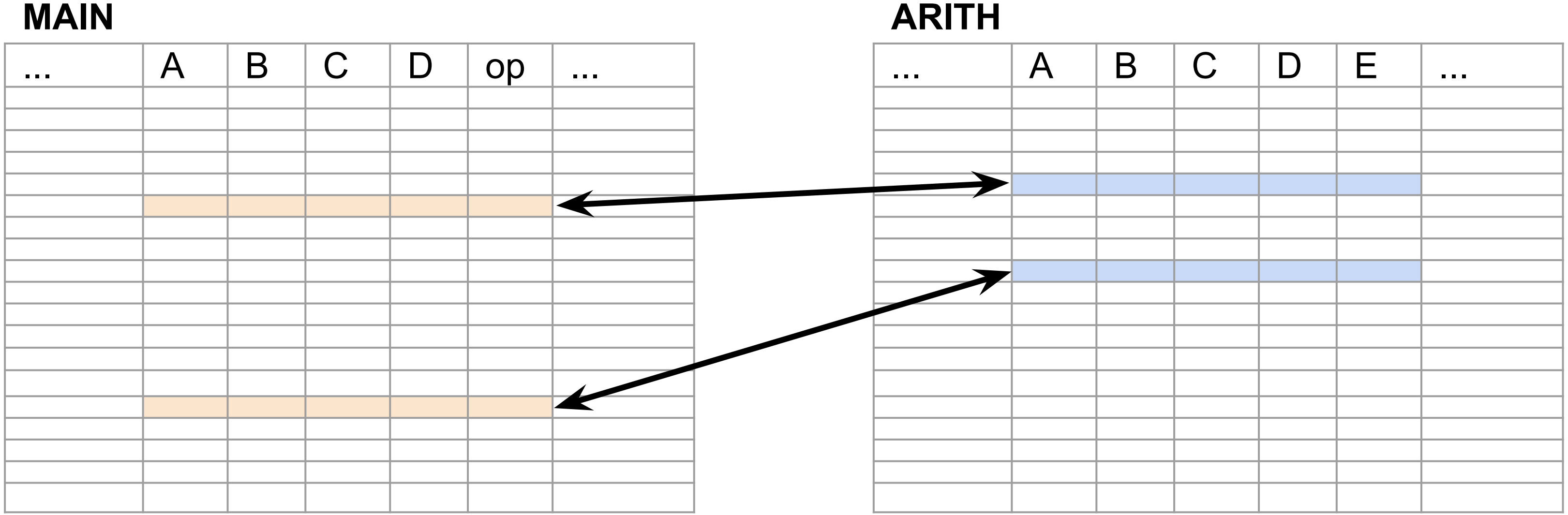
$\subset$

Memory

ADDR	COUNT	INSTRUCTION	VALUE
3	2	WR	2
3	6	RD	2
5	1	WR	8
5	3	RD	8
5	5	RD	8
5	7	WR	34
5	9	RD	34

Connecting two state machines with Plookup

```
MAIN.arith {MAIN.A , MAIN.B , MAIN.C , MAIN.D, MAIN.op} in
ARITH.latch {ARITH.A , ARITH.B , ARITH.C , ARITH.D , ARITH.E};
```



EVM Processor

RAM

- Multiple R/W
- 1 Access per CLOCK
- Paged for handling Ethereum CALL contexts
- 32 byte alignment sub stat machine.

ROM

- The Code that always execute the prover
- It cannot be modified.

STORAGE

- Sparse Merkle Tree
- Goldilocks Poseidon hash function
- Single tree for the system
- Hashes of the smart contract codes are in the tree.

EVM Processor

RAM

ROM

STORAGE

BINARY

- Operations done byte to byte with a carry from a plookup table
    - ADD
    - SUB
    - LT & SLT
    - EQ
    - AND
    - OR
    - XOR

ARITHMETIC

- 256 bits arithmetic operations
- A*B + C = D*2^256 + E
- Range check of inputs and outputs
- 32 CLOCKs per operation
- Includes EC addition formulas for ECDSA multiplication.

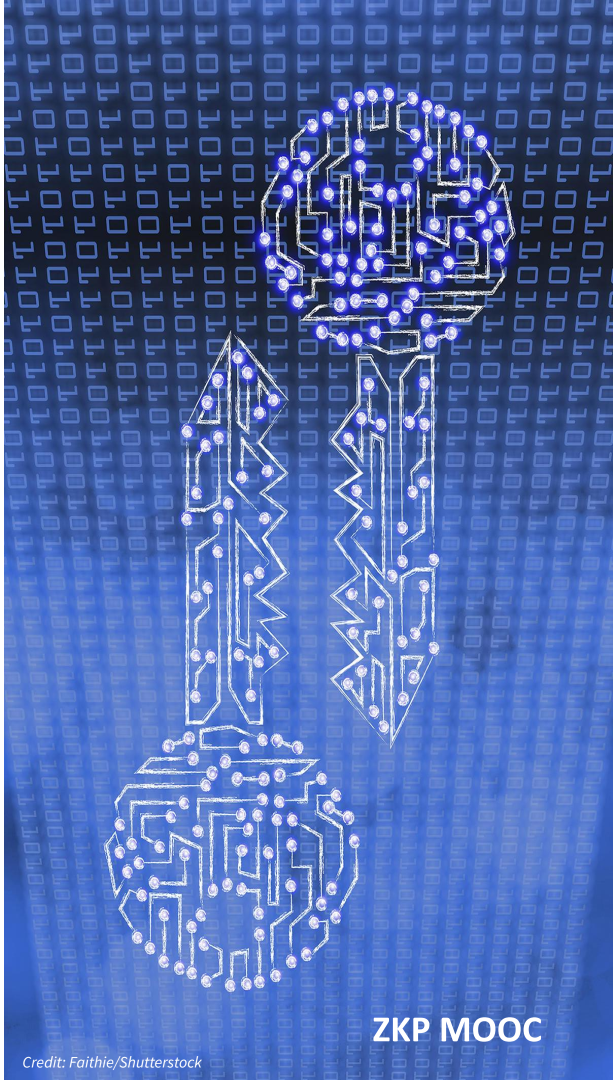
HASH

- Binary circuit of ANDn and XOR
- We use a plookup to do various circuits in parallel
- We currently can do 468 keccakf's in the current circuit. (N = 2^23).

Section 4

zkROM enabling EVM Emulation

Learning objective: how to run a program on top of the processor



Credit: Faithie/Shutterstock

ZKASM-ROM

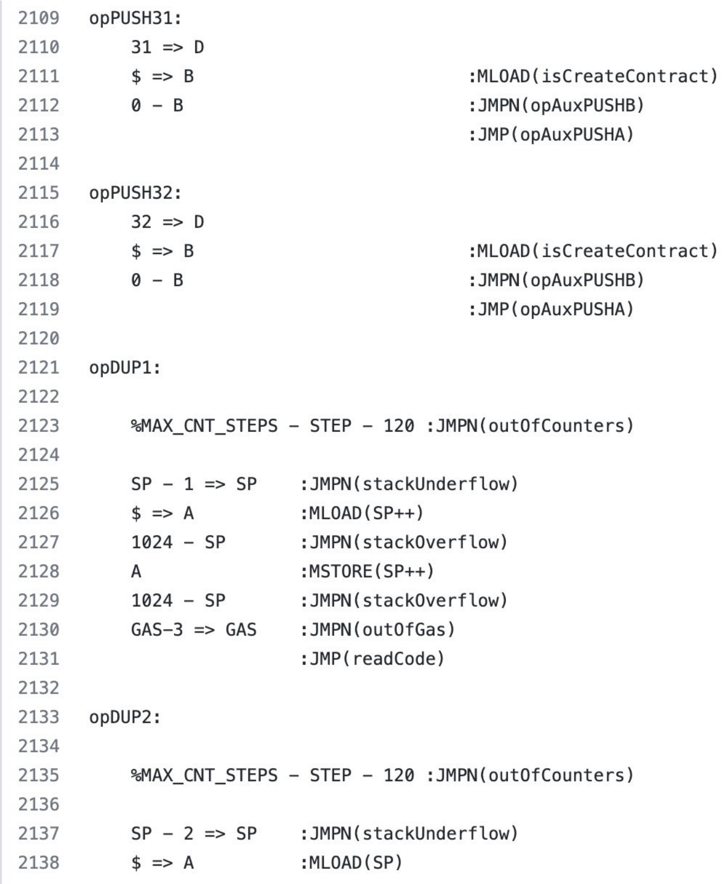
Ethereum Transaction processor

FREE Input the Transactions and the hash must match.

zkCounters to prevent the proof to fail (DoS).

Some examples:

- Opcodes
- RLP Processing

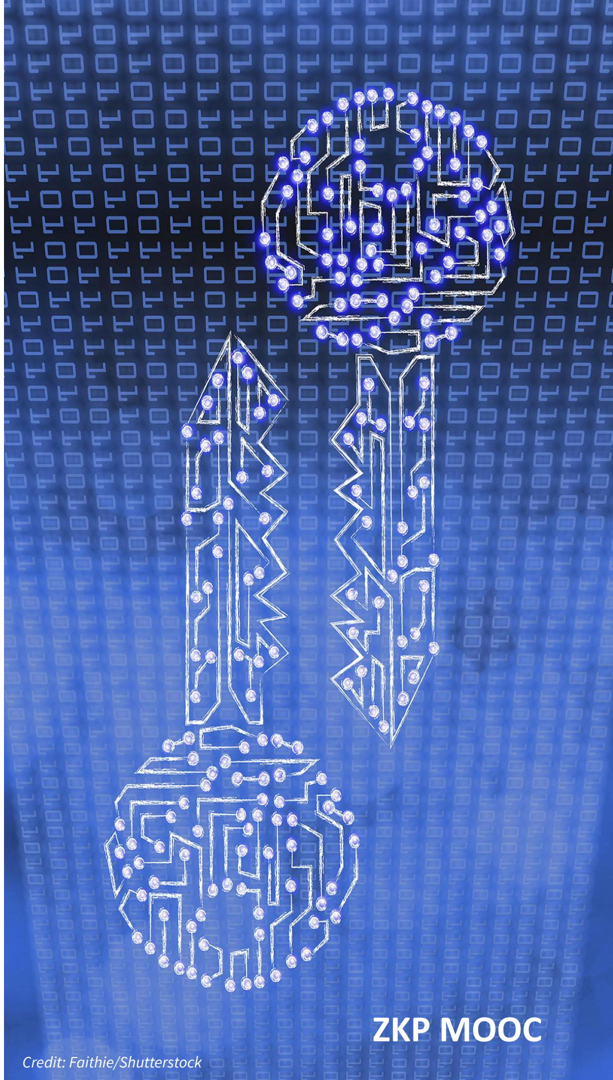


```
opPUSH31:
    31 => D
    $ => B                          :MLOAD(isCreateContract)
    0 - B                           :JMPN(opAuxPUSHB)
                                    :JMP(opAuxPUSHA)

opPUSH32:
    32 => D
    $ => B                          :MLOAD(isCreateContract)
    0 - B                           :JMPN(opAuxPUSHB)
                                    :JMP(opAuxPUSHA)

opDUP1:

    %MAX_CNT_STEPS - STEP - 120 :JMPN(outOfCounters)

    SP - 1 => SP    :JMPN(stackUnderflow)
    $ => A          :MLOAD(SP++)
    1024 - SP       :JMPN(stackOverflow)
    A               :MSTORE(SP++)
    1024 - SP       :JMPN(stackOverflow)
    GAS-3 => GAS    :JMPN(outOfGas)
                    :JMP(readCode)

opDUP2:

    %MAX_CNT_STEPS - STEP - 120 :JMPN(outOfCounters)

    SP - 2 => SP    :JMPN(stackUnderflow)
    $ => A          :MLOAD(SP)
```

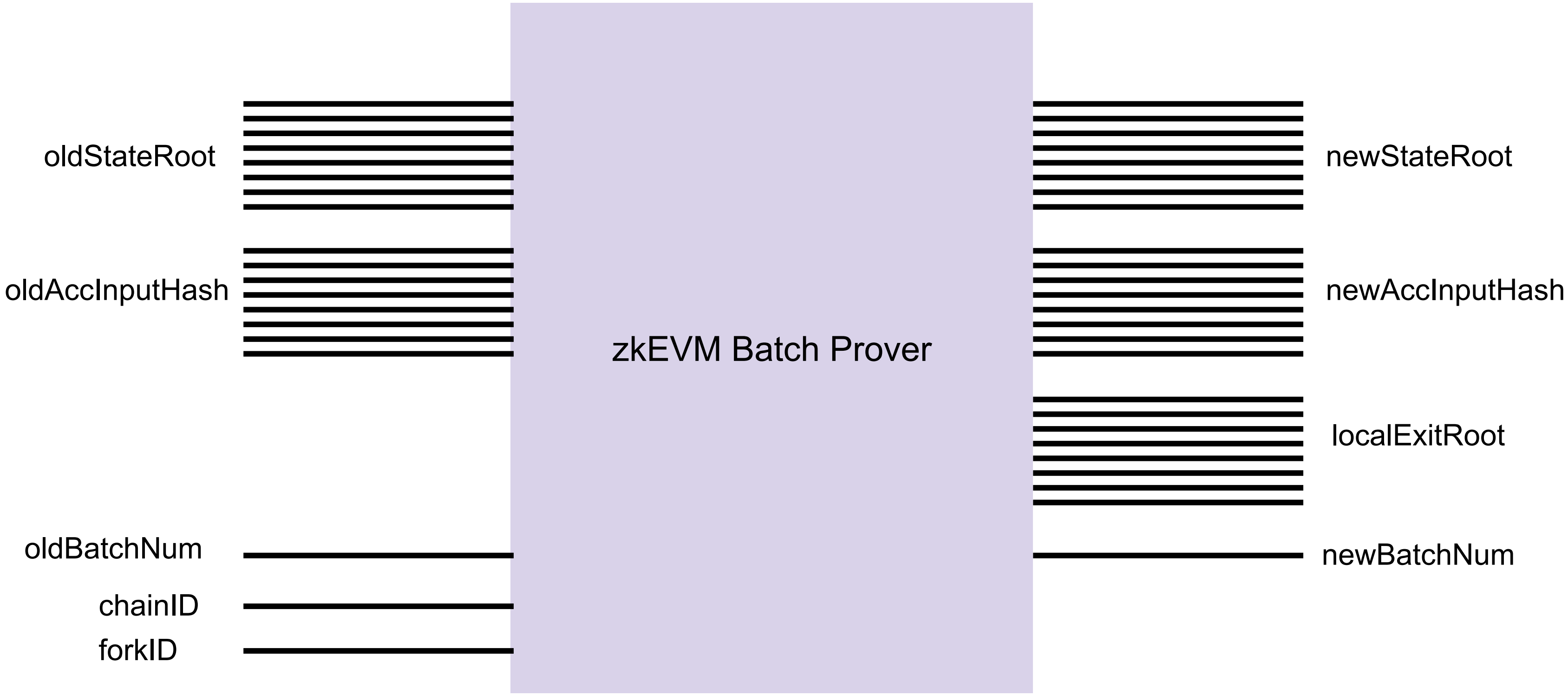
Section 5

Recursion and final proof

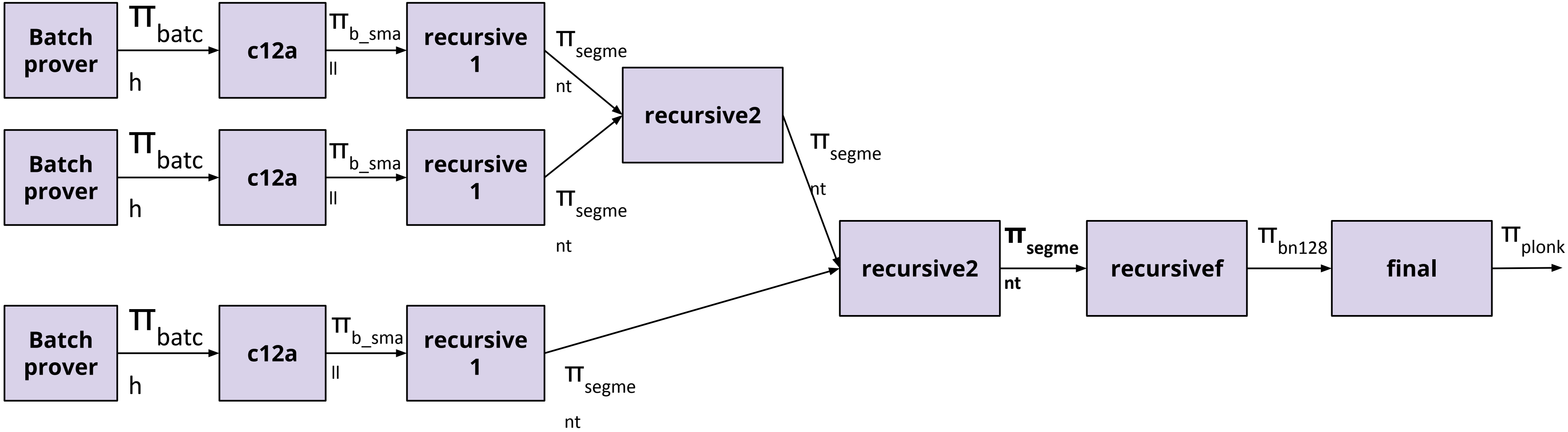
Learning objective: Understand how to compress and aggregate the proof and verify them on-chain

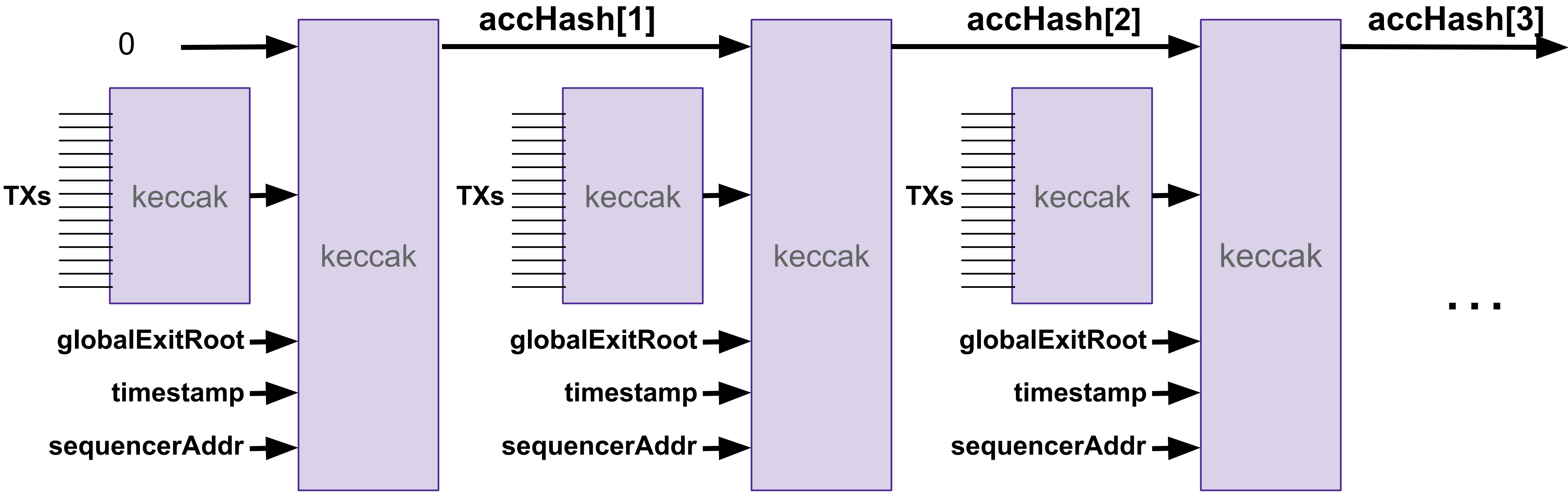


Credit: Faithie/Shutterstock


oldStateRoot
oldAccInputHash
oldBatchNum
chainID
forkID
zkEVM Batch Prover
newStateRoot
newAccInputHash
localExitRoot
newBatchNum

Recursion and on-chain verification




accHash[1]
accHash[2]
accHash[3]
0
TXs
keccak
keccak
globalExitRoot
timestamp
sequencerAddr
TXs
keccak
keccak
globalExitRoot
timestamp
sequencerAddr
TXs
keccak
keccak
globalExitRoot
timestamp
sequencerAddr
...

Statistics of the zkEVM circuit

Number of Commited Polynomials: **669**

Number of permutation checks: **18**

Number of plookups: **29**

Number of connection checks (copy constraints): **2**

Total number of columns: **1184**

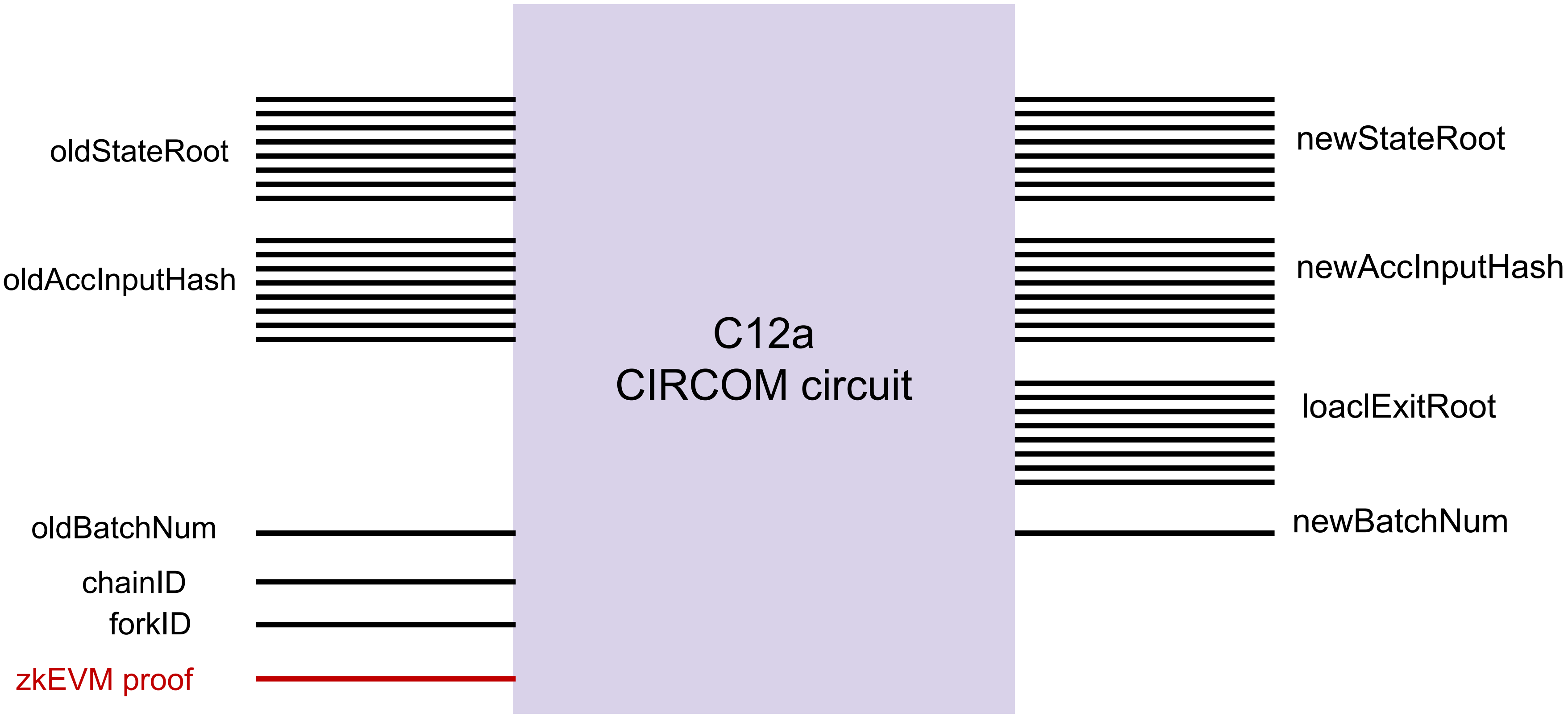
Degree of the polynomials (rows) n= $\mathbf{2^{23}}$

Max degree of the constraint polynomial: **3n**

Blowup factor: **2**

Proof computation time: **129s**

Size of the proof: **1.9M**


oldStateRoot
oldAccInputHash
oldBatchNum
chainID
forkID
zkEVM proof
C12a
CIRCOM circuit
newStateRoot
newAccInputHash
loaclExitRoot
newBatchNum

From Circom to PIL

Circom works in goldilocks

Signals in 12 columns

Gates:

- Standard PLONK gates (4 per row)
- Poseidon STEP (12 inputs 12 outputs). We use 2 rows for a step. 31 rows for a full poseidon hash.
- FFT4 FFT of 4 elements in extension 3 (12 inputs - 12 outputs). Basic block to build bigger FFTs.
- D = A*B + C in extension 3 (1 row)
- Polynomial evaluation custom gate in extension3. newA = (((oldA * x + C3)*x + C2)*x +C1)*x + C0 . This uses 2 rows and can be used to compute evaluations of pols of bigger size.

Given a circom circuit it can be converted to a PIL, a constants polynomial and a witness computation program.

Statistics of the c12a circuit

Number of Commited Polynomials: **12**

Number of permutation checks: **0**

Number of plookups: **0**

Number of connection checks (copy constraints): **1**

Total number of columns: **65**

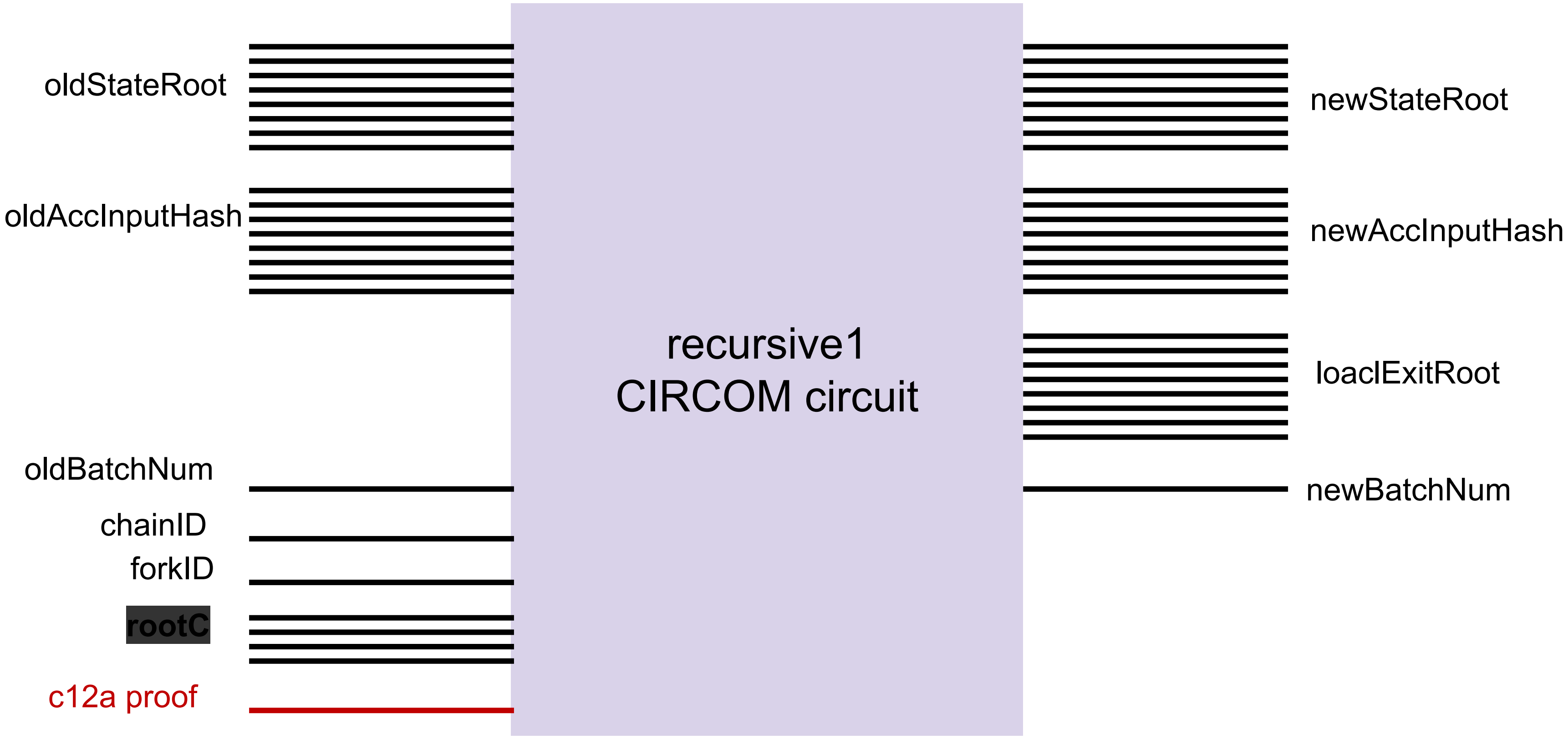
Degree of the polynomials (rows) n= $\mathbf{2^{22}}$

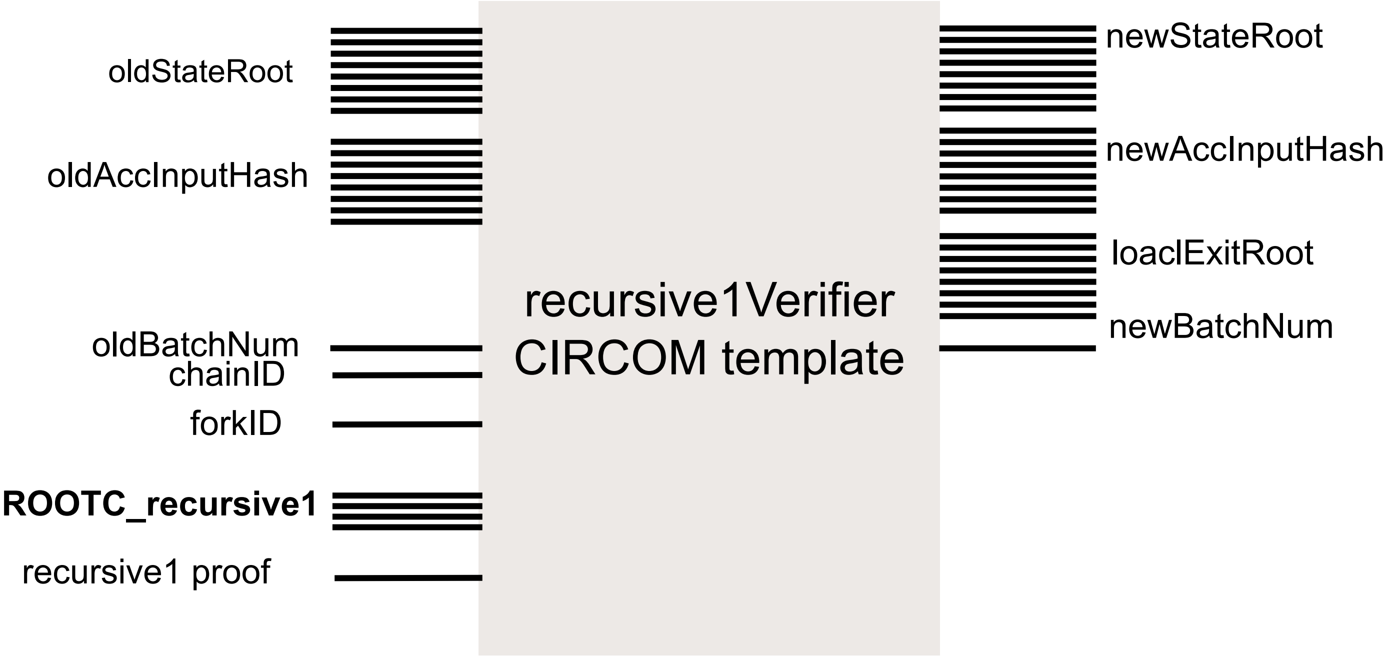
Max degree of the constraint polynomial: **5n**

Blowup factor: **4**

Proof computation time: **14s**

Size of the proof: **494K**

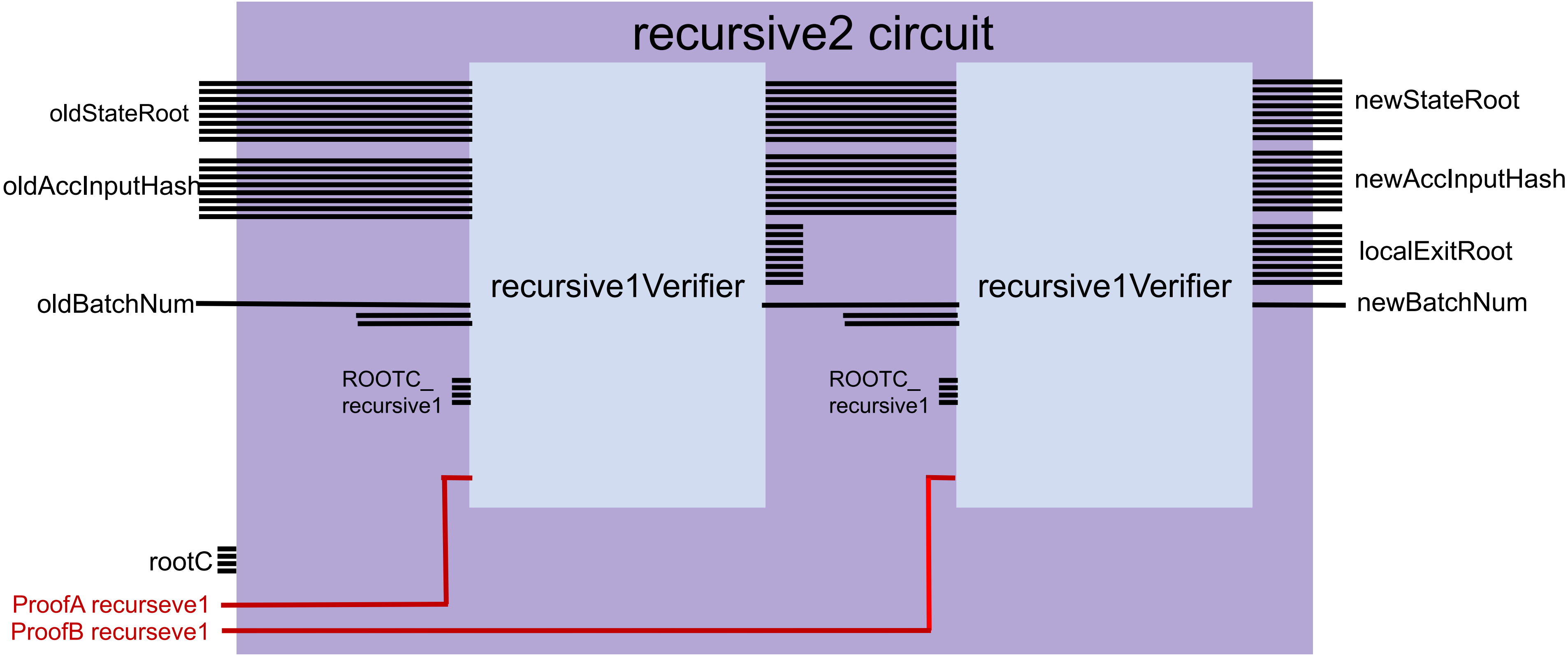

oldStateRoot
oldAccInputHash
oldBatchNum
chainID
forkID
rootC
c12a proof
recursive1
CIRCOM circuit
newStateRoot
newAccInputHash
loaclExitRoot
newBatchNum

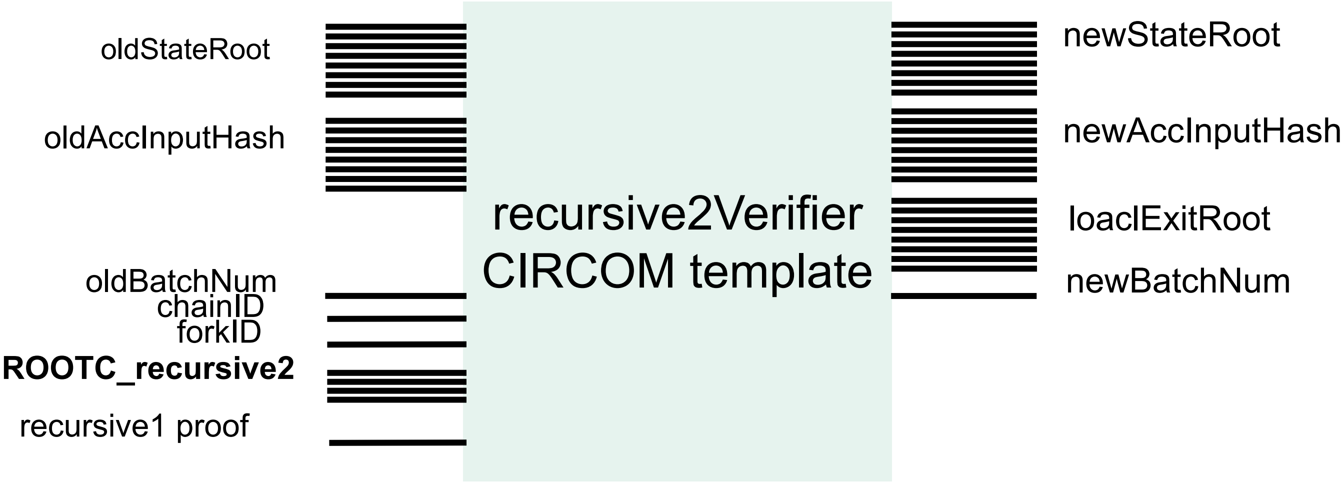


recursive1Verifier
CIRCOM template

oldStateRoot
oldAccInputHash
oldBatchNum
chainID
forkID
ROOTC_recursive1
recursive1 proof

newStateRoot
newAccInputHash
loaclExitRoot
newBatchNum


recursive2 circuit
oldStateRoot
oldAccInputHash
oldBatchNum
ROOTC_
recursive1
rootC
ProofA recurseve1
ProofB recurseve1
recursive1Verifier
ROOTC_
recursive1
recursive1Verifier
newStateRoot
newAccInputHash
localExitRoot
newBatchNum



oldStateRoot

oldAccInputHash

oldBatchNum

chainID

forkID

ROOTC_recursive2

recursive1 proof

recursive2Verifier
CIRCOM template

newStateRoot

newAccInputHash

loaclExitRoot

newBatchNum

recursive1Verifier CIRCOM template

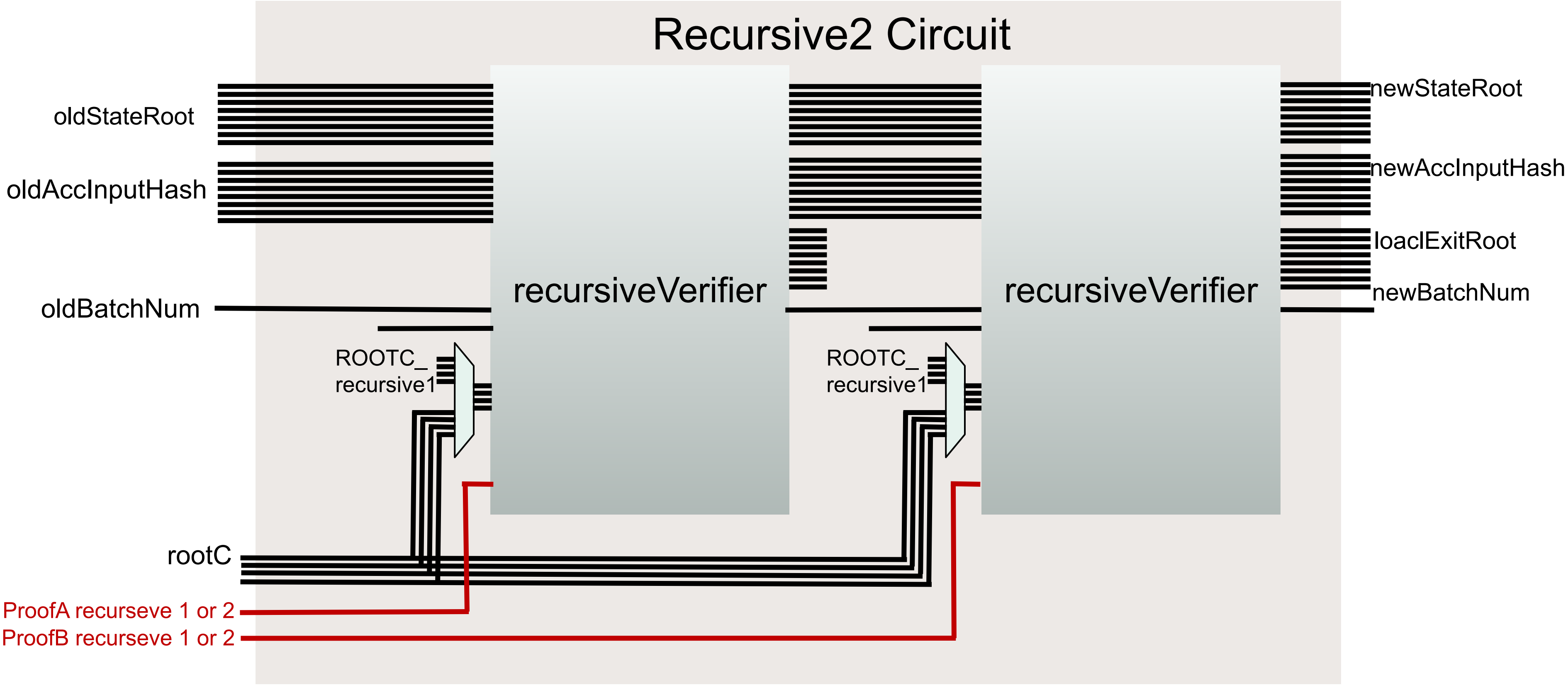
=

recursive2Verifier CIRCOM template

≡

recursiveVerifier CIRCOM template

Recursive2 Circuit



Statistics of the Recursive1 and Recursive2 circuit

Number of Commited Polynomials: **12**

Number of permutation checks: **0**

Number of plookups: **0**

Number of connection checks (copy constraints): **1**

Total number of columns: **45**

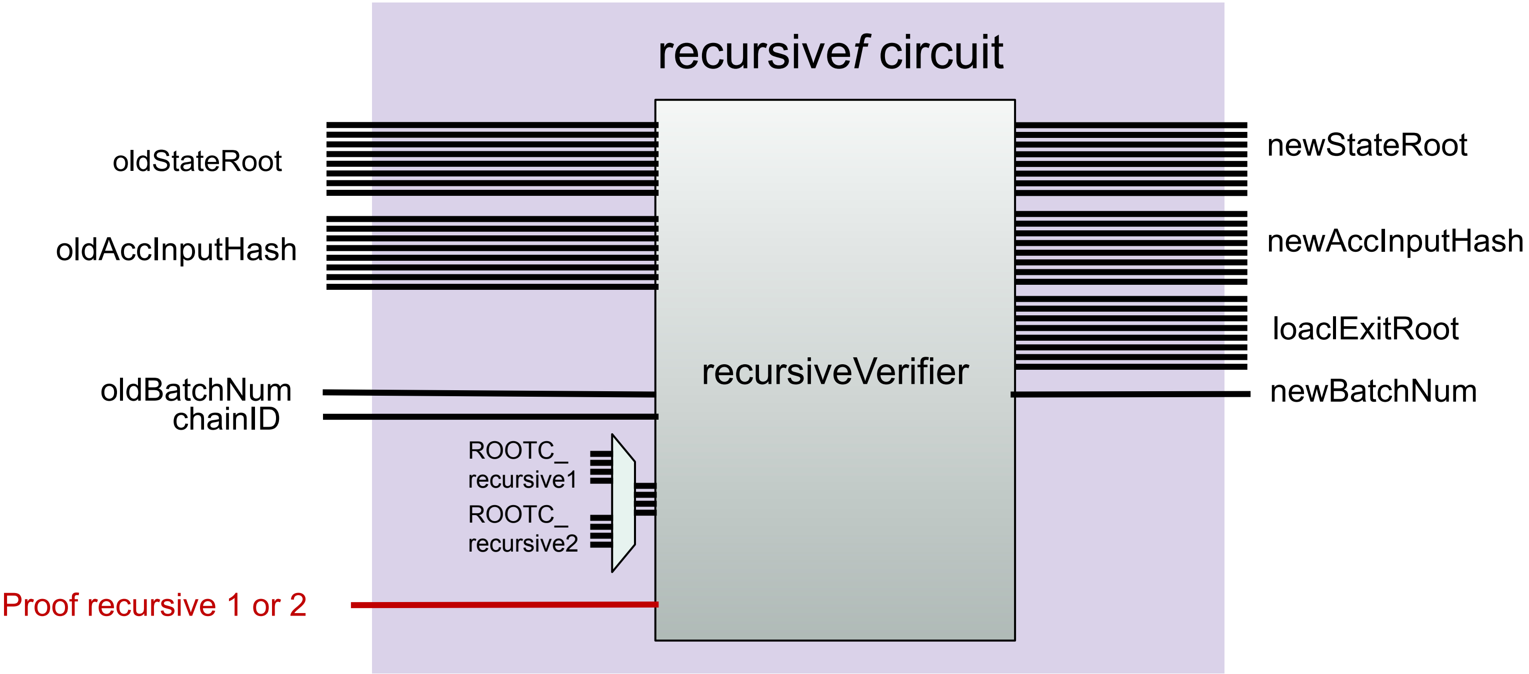
Degree of the polynomials (rows) n= $\mathbf{2^{20}}$

Max degree of the constraint polynomial: **9n**

Blowup factor: **16**

Proof computation time: **10s**

Size of the proof: **~260K**



- Forces the ROOTC_recursive2 in the circuit
- STARK is generated with BN128 poseidon

Statistics of the Recursive*f* circuit

Number of Commited Polynomials: **12**

Number of permutation checks: **0**

Number of plookups: **0**

Number of connection checks (copy constraints): **1**

Total number of columns: **45**

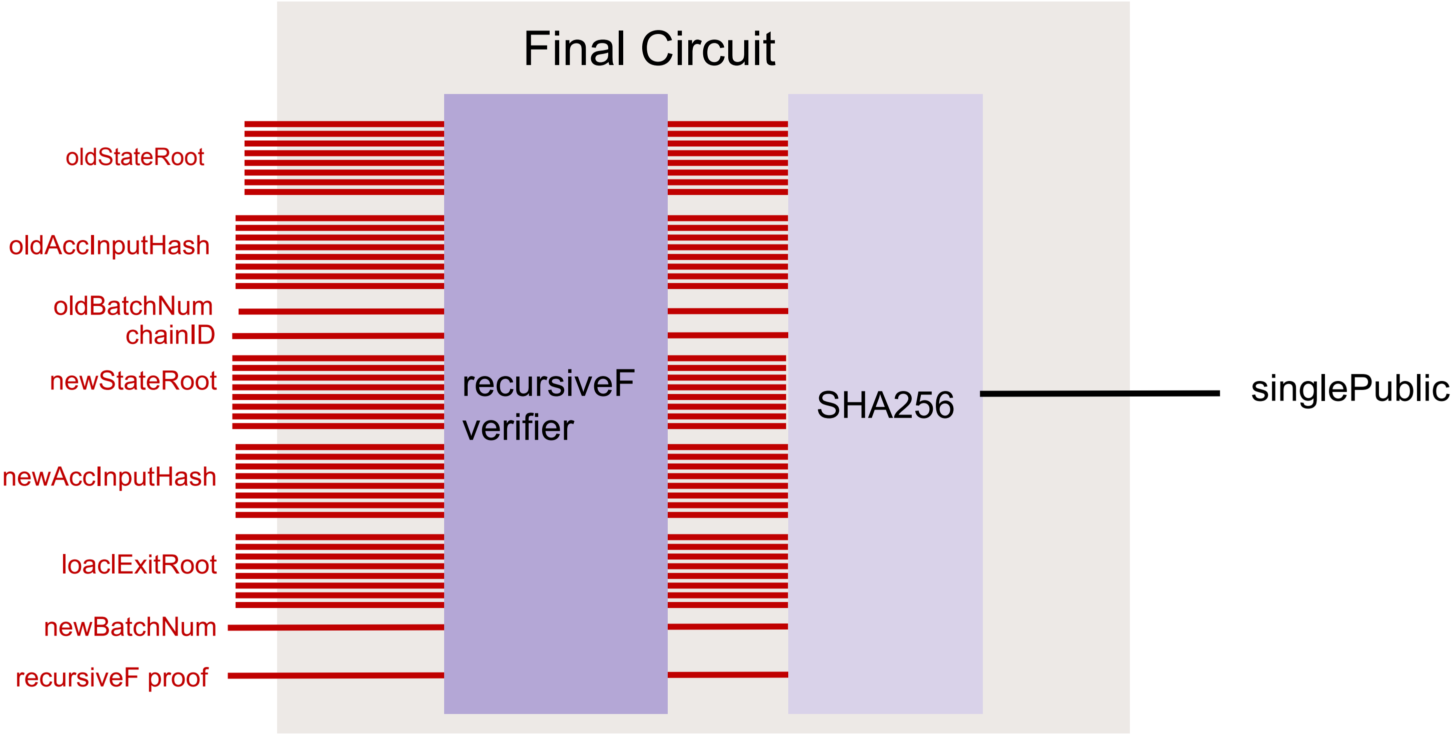
Degree of the polynomials (rows) n= $\mathbf{2^{19}}$

Max degree of the constraint polynomial: **9n**

Blowup factor: **16**

Proof computation time: **17s**

Size of the proof: **~505K**


Final Circuit
oldStateRoot
oldAccInputHash
oldBatchNum
chainID
newStateRoot
newAccInputHash
loaclExitRoot
newBatchNum
recursiveF proof
recursiveF verifier
SHA256
singlePublic

Statistics of the Final circuit

Number of constraints: **16M**

Proof computation time: **120s**

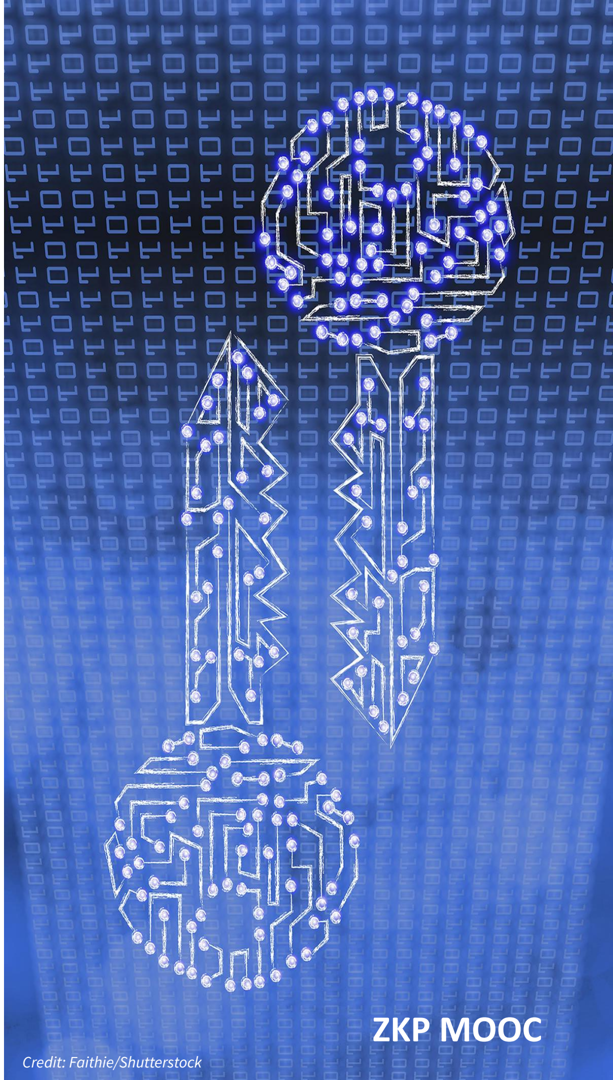
Size of the proof: **<1k**

Gas cost: **362K** (Full proving TX)

Section 6

Looking ahead

Learning objective: Understand the sustained evolution of the zkEVM (Performance, Security and Decentralization



Credit: Faithie/Shutterstock

ZKP MOOC

zkEVM Evolution

The next work in the pipeline to evolve zkEVM:

Transition from a Type 3 to Type 2 zkEVM according to Vitalik's ZK Rollups categorization

Data compression optimizations

EIP 4844

Variable size Prover

Security

Decentralized Sequencer

Open Source zkevm https://github.com/0xPolygonHermez

zkEVM Documentation https://wiki.polygon.technology/docs/zkEVM/introduction/

Core repos

- zkevm-proverjs
- zkevm-rom
- zkevm-prover
- zkevm-node
- zkevm-contracts
- zkevm-bridge-service
- zkevm-bridge-ui
- zkevm-techdocs

zkEVM specific tools and libraries

- zkevm-commonjs
- zkasmcom
- zkevm-testvectors
- zkevm-storagerom

Generic tools and libraries

- pilcom
- pil-stark

polygon zkEVM