
PLAYER-1
0
TOP SCORE
35000
PLAYER-2
0
TIME
2:13
snarks @ IPFS camp
unconf

*

*****

dev @ iden3

adria0 @ twitter

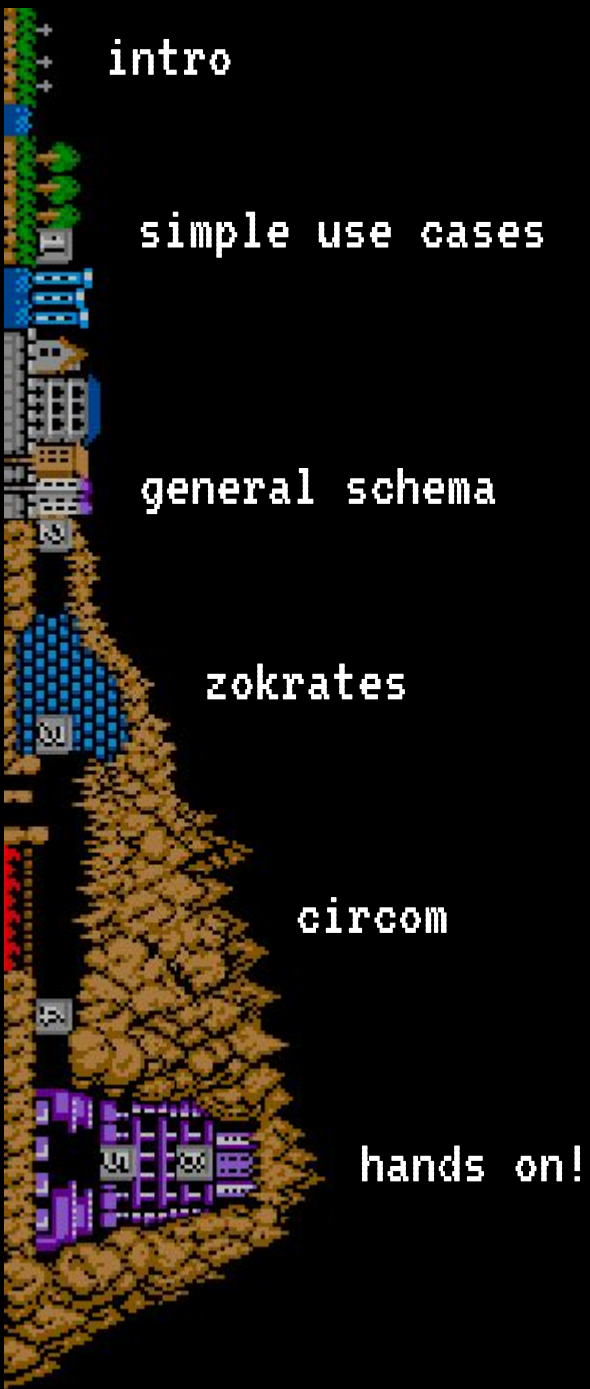
org @ ethdevbcn meetup

auditor 4 maker, aragonOs,...

lecturer secure programming @ UPC

ukelele, hypnosys, analog synths & rust

****


intro
simple use cases
general schema
zokrates
circom
hands on!

zk-snark

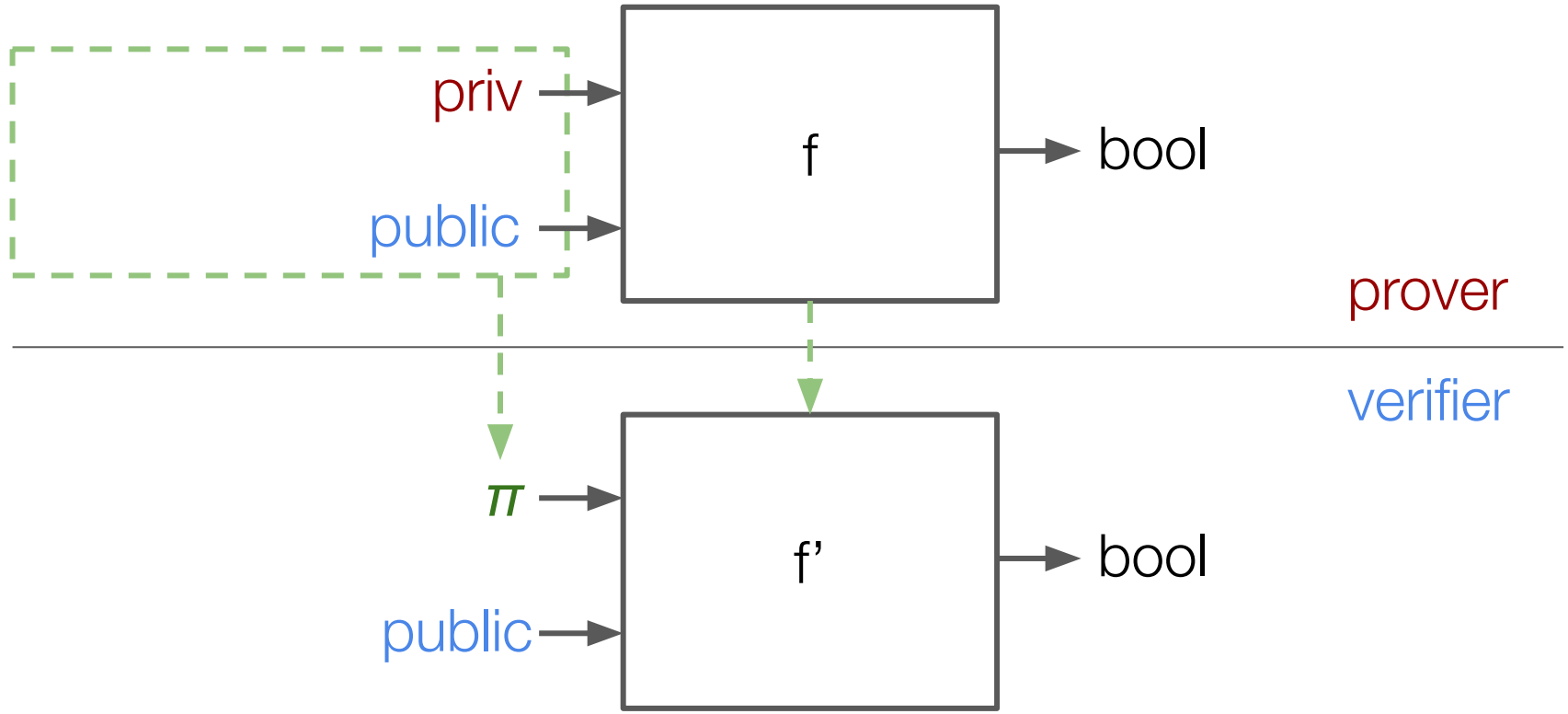
- Zero Knowledge
- Succinct
- Non-interactive
- ARgument of Knowledge

“Zero-knowledge” proofs allow one party (the prover) to prove to another (the verifier) that a statement is true, without revealing any information beyond the validity of the statement itself. For example, given the hash of a random number, the prover could convince the verifier that there indeed exists a number with this hash value, without revealing what it is. (Zcash)

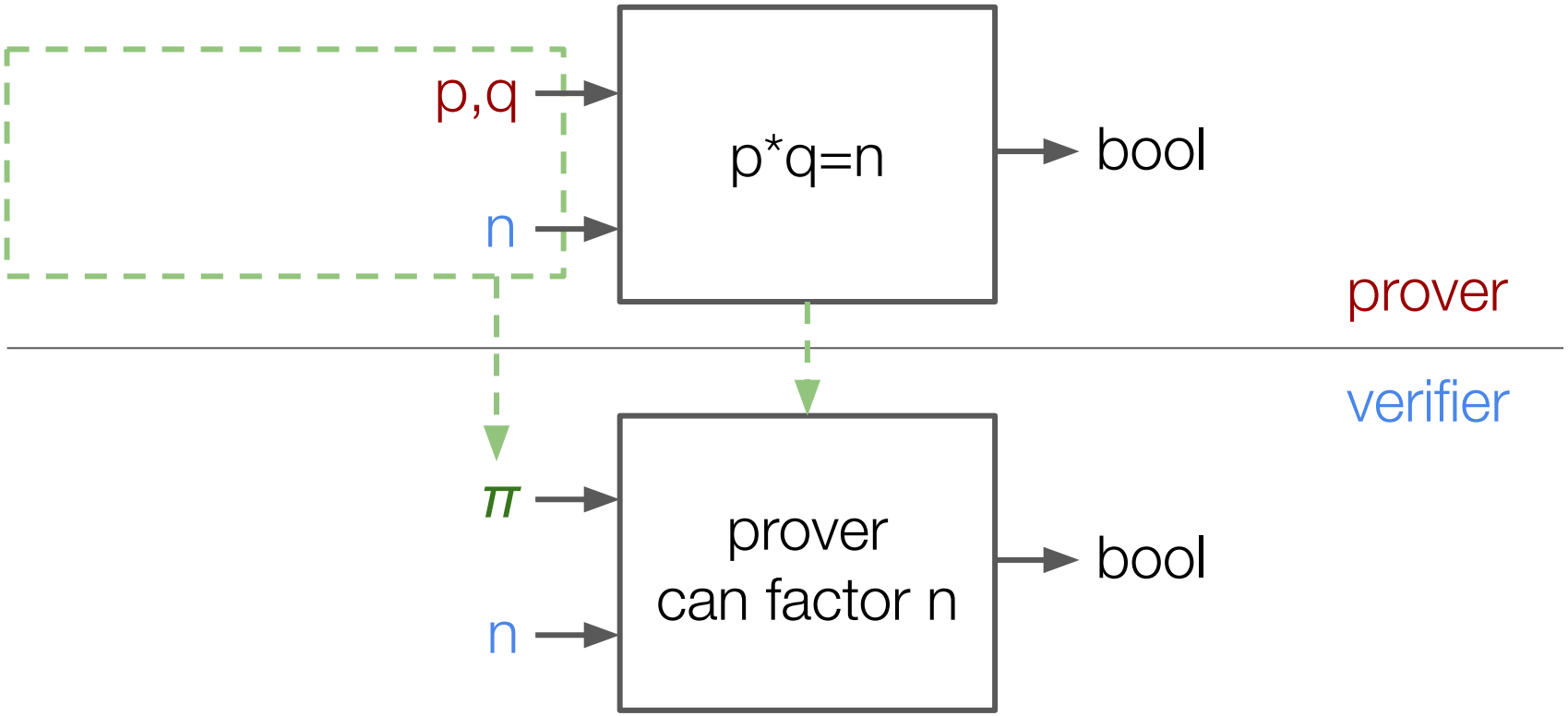
zk-snark

- “Programmable” zero knowledge proofs
- Compatible with ethereum via bn128
- Growing set of tools
- Compression / scalability
- Privacy & Anonymity
- Zcash / Filecoin / Coda / Iden3 / RollUp / ZEXE

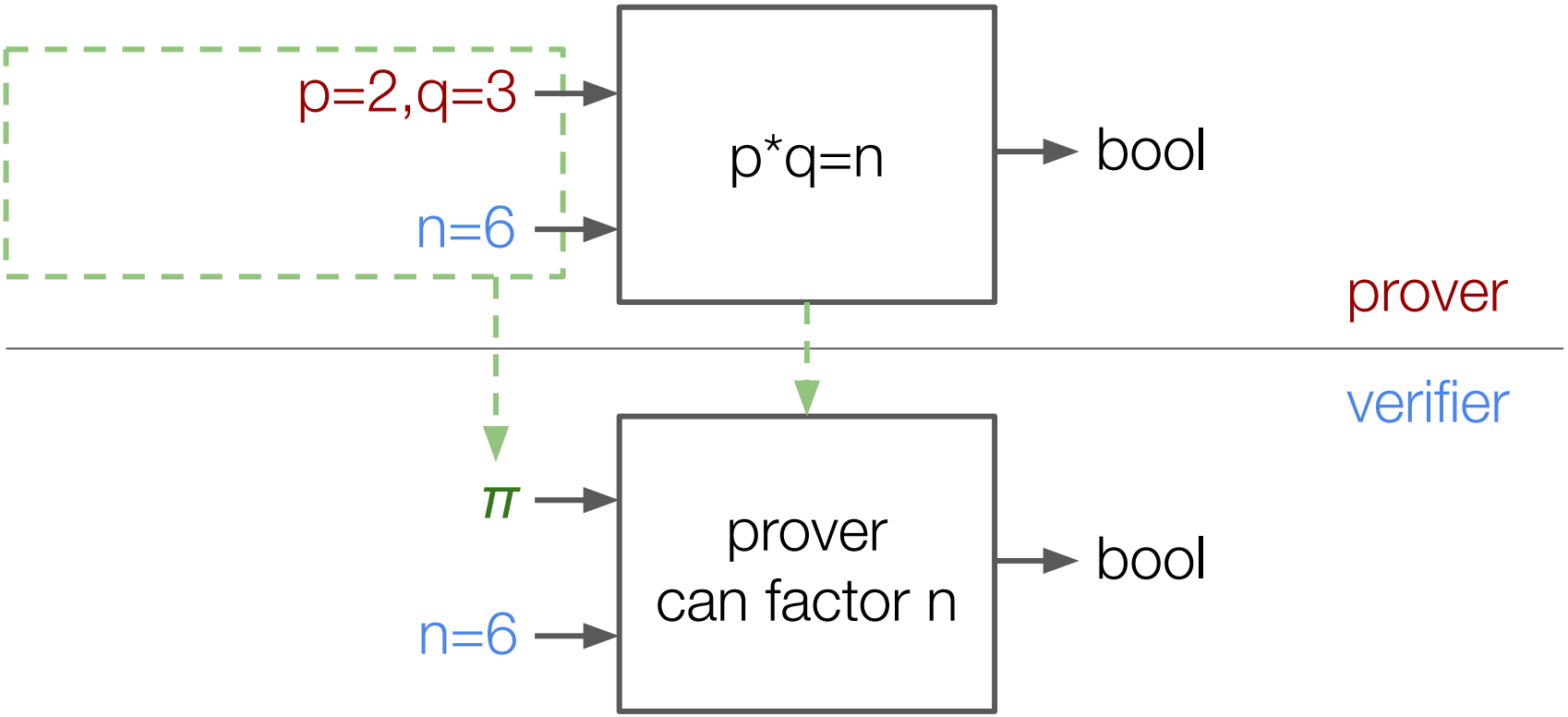
zk-snark



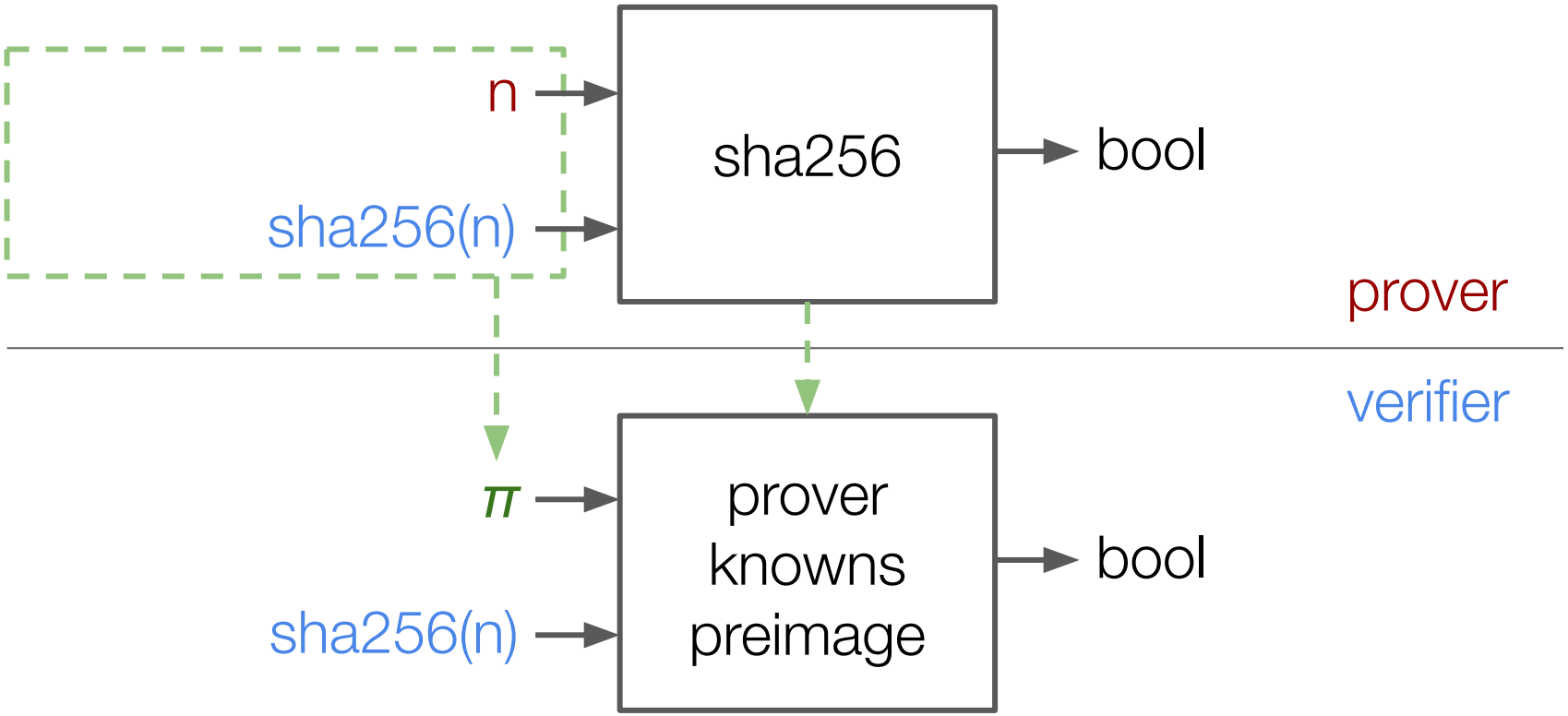
proof-of-factor



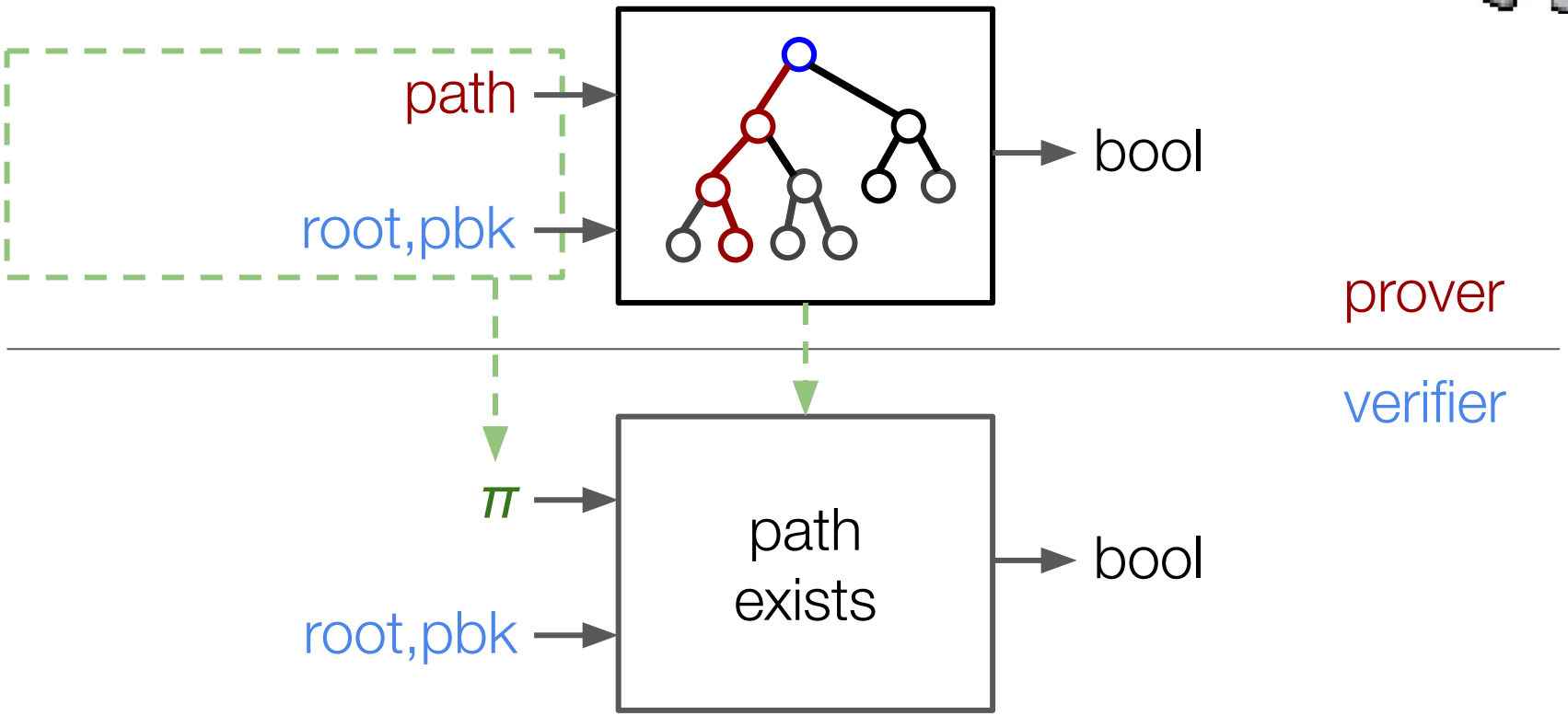
proof-of-factor



knowledge of preimage



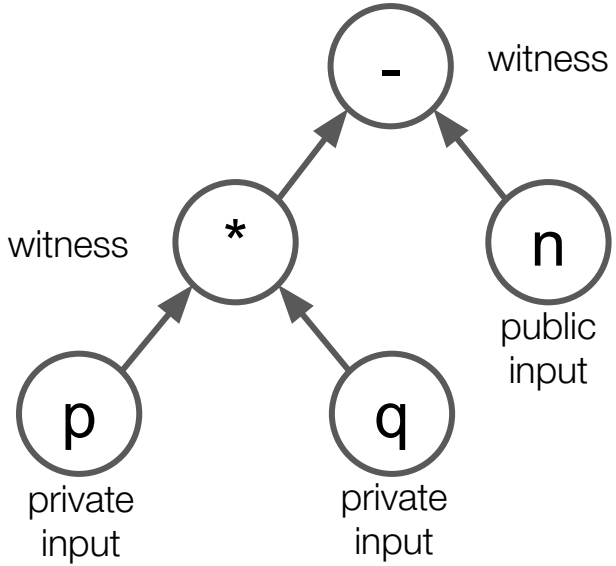
proof-of-existence



check https://github.com/zcash-hackworks/babyzoe/blob/master/talks/2016-07-27-IC3---SNARKs-for-Ethereum.pdf !

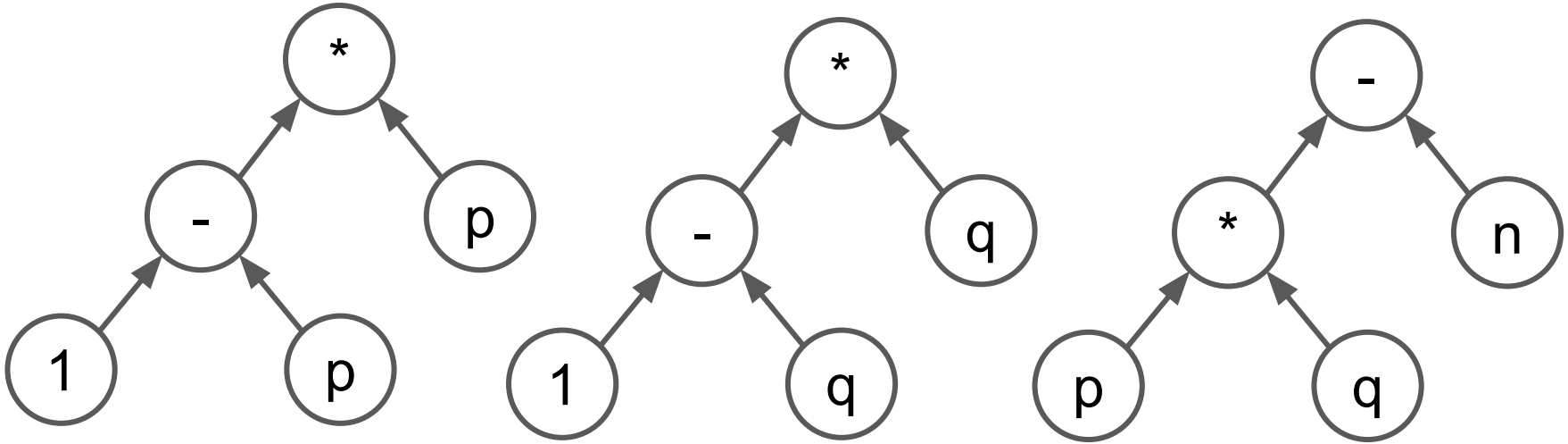
f

- No turing-complete
- Build an arithmetic circuits with +,-,*, that evals to zero
- $[a_1S_1+..+a_nS_n] * [b_1S_1+..+b_nS_n] + [c_1S_1+..+c_nS_n] = 0$
- E.g n=p*q
- $1S_p * 1S_q + -1S_n = 0$



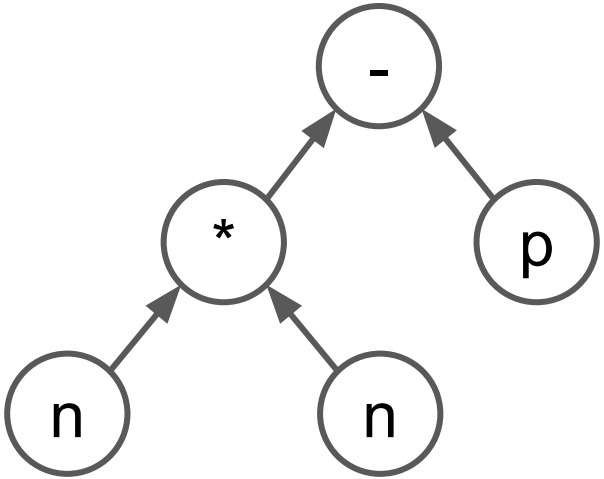
f

- n=p AND q
- $(-1S_p + 1S_{ONE})*(1S_p)+[]$
- $(-1S_q + 1S_{ONE})]*(1S_q)+[]$
- $(1S_p)*(1S_q)+(-1S_n)$



f

- $n=\sqrt{p}$
- $(1S_n)*(1S_n)+(-1S_p)$



about the magic

Generate R1CS constraints

$[a_1S_1+..+a_nS_n] * [b_1S_1+..+b_nS_n] + [c_1S_1+..+c_nS_n]=0$
$[a_1S_1+..+a_nS_n] * [b_1S_1+..+b_nS_n] + [c_1S_1+..+c_nS_n]=0$
$[a_1S_1+..+a_nS_n] * [b_1S_1+..+b_nS_n] + [c_1S_1+..+c_nS_n]=0$
$[a_1S_1+..+a_nS_n] * [b_1S_1+..+b_nS_n] + [c_1S_1+..+c_nS_n]=0$
$[a_1S_1+..+a_nS_n] * [b_1S_1+..+b_nS_n] + [c_1S_1+..+c_nS_n]=0$
$[a_1S_1+..+a_nS_n] * [b_1S_1+..+b_nS_n] + [c_1S_1+..+c_nS_n]=0$
$[a_1S_1+..+a_nS_n] * [b_1S_1+..+b_nS_n] + [c_1S_1+..+c_nS_n]=0$

Rewr. as a matrix

A(w) x B(w)+C(w) = 0

Rewr. as divisibility

(A(w) x B(w) + C(w)) / Z
has no remainder

Elliptic curve pairings

$$e\left(P^a, Q^b\right) = e(P, Q)^{ab}$$

zk-snark proof

Describe the circuit

`p * q * r == n`

`where p,q,r are private`

`p * q + -w1 = 0`

`w1 * r - n = 0`

compile

RAW R1CS

$[a_1S_1+..+a_nS_n] * [b_1S_1+..+b_nS_n] + [c_1S_1+..+c_nS_n]=0$

$[a_1S_1+..+a_nS_n] * [b_1S_1+..+b_nS_n] + [c_1S_1+..+c_nS_n]=0$

$[a_1S_1+..+a_nS_n] * [b_1S_1+..+b_nS_n] + [c_1S_1+..+c_nS_n]=0$

...

OPTIMIZED R1CS

$[a_1S_1+..+a_nS_n] * [b_1S_1+..+b_nS_n] + [c_1S_1+..+c_nS_n]=0$

...

PUBLIC INPUT 30

INPUT 2,3,5,30

trusted setup

WITNESS 2,3,5,30,6

PROV KEY

VERIF KEY

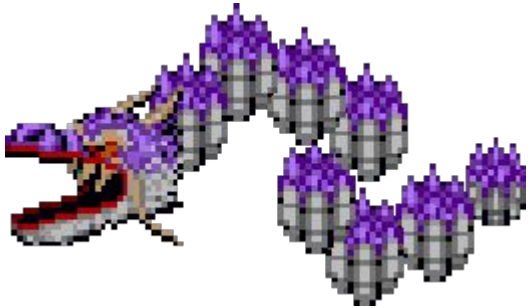
PROVER

π

VERIFIER

bool

compilers



- Zokrates
  - https://github.com/Zokrates/ZoKrates by Thibaut Schaeffer
  - Rust / Bellman
  - Algebra-approach, high level, can handle automatically a*b*c == 0
  - Focused on generating ethereum smartcontracts
- Circom
  - https://github.com/iden3 mainly all zk* made by Jordi
  - Js / wasm
  - Circuit-approach, low level, allows manual optimizations
  - General tooling & flexible framework, smartcontracts & wasm
- There are more! see Harry Roberts' talks

proof-of-factor

- I want a smartcontract that gives 1ETH to whom factors a number
- play(uint256 p, uint256 q) can be frontrunned with higher gas
- I need to proof to the smartcontract that I factored it without revealing p,q

Creator

Create the verifier
Deploy the SC with challenge

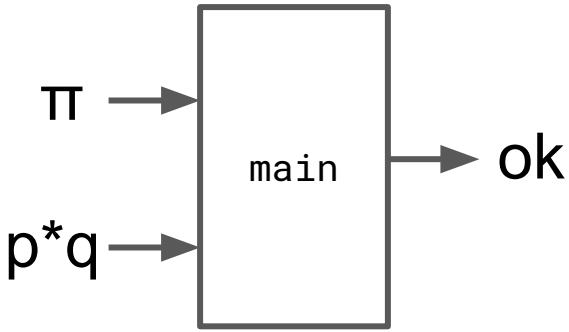
```
uint256 challange;
prove(proof) {
  verify(challange,proof)
  msg.sender.transfer(1 ether);
  selfdestruct();
}
```

User

Read challange from SC
Factorize challange and get p,q
Generate proof that I own p,q
Call SC prove(proof)

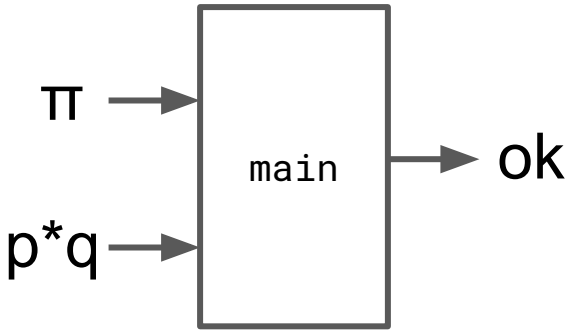
public sybil control by proof-of-factor

```
p * q == r
```



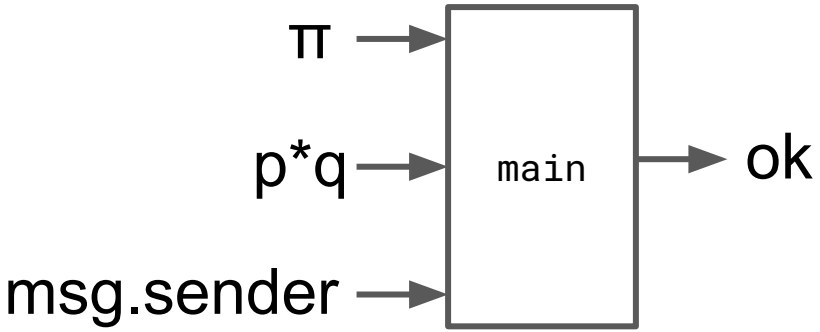
public sybil control by proof-of-factor

```
p * q == r

p != 1

q != 1
```



public sybil control by proof-of-factor

```
p * q == r

p != 1

q != 1

msg.sender != 0
```



zokrates

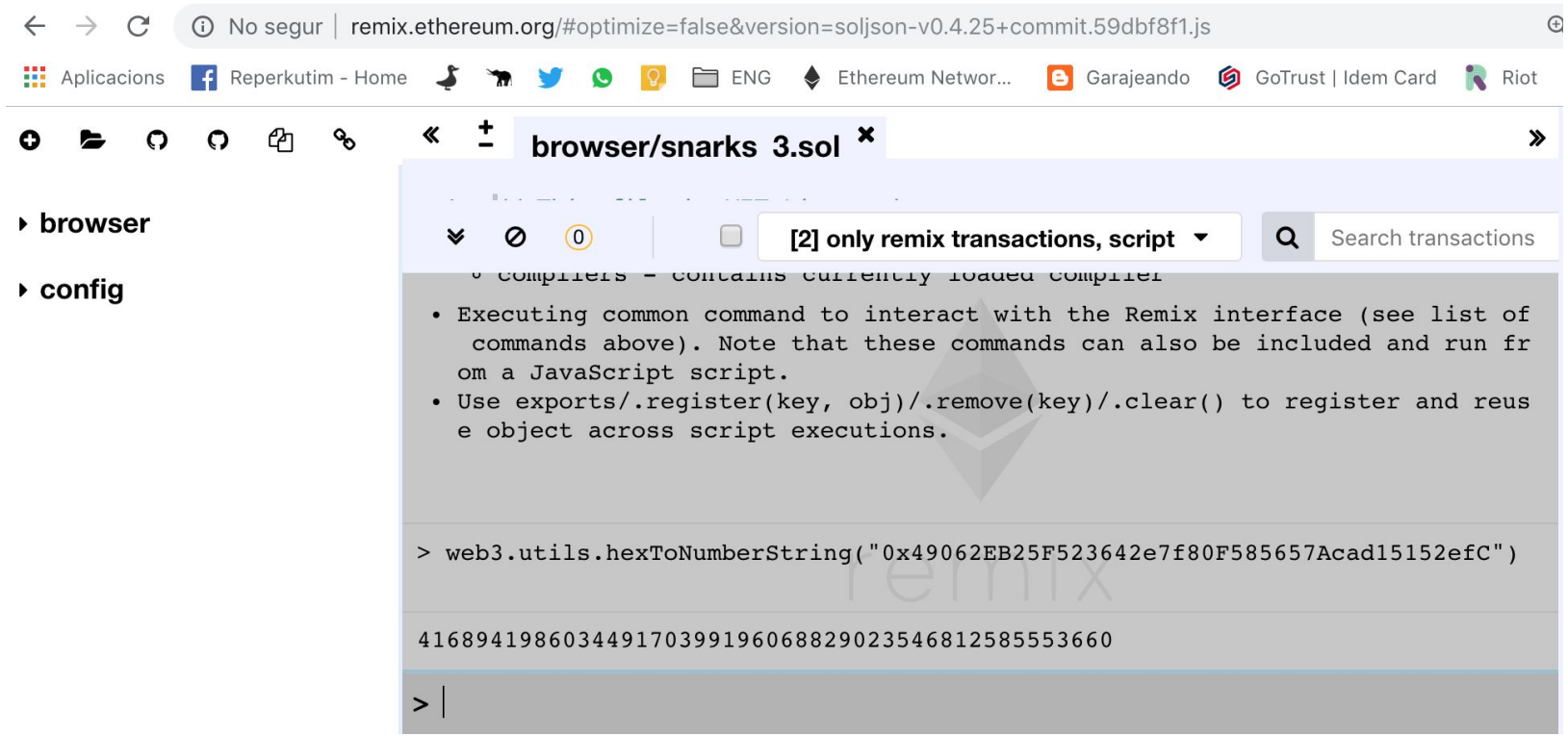
sybil.code

```
def main(private field p, private field q, field id) -> (field):
  0 == if 1 == p then 1 else 0 fi
  0 == if 1 == q then 1 else 0 fi
  0 == if 0 == id then 1 else 0 fi
  return p * q
```

compile, setup, witness, proof and verifier

```
docker run --name zokrates -ti zokrates/zokrates:0.3.3 /bin/bash

  docker exec -u 0 -it zokrates bash

  apt update && apt install vim

vim sybil.code

./zokrates compile -i sybil.code

./zokrates setup

./zokrates export-verifier
```

set-up get msg.sender as decimal number



generate proof that I can factor 6

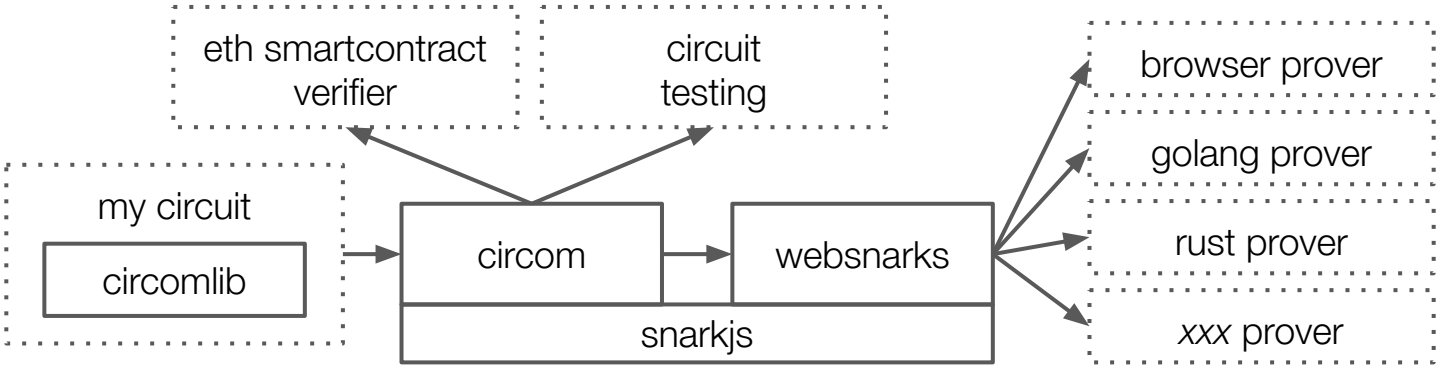
```
./zokrates compute-witness -a 2 3 <address as number>

./zokrates generate-proof
```

```
event CanFactor(bool yes);
function check() public returns(bool) {
    var proof = Verifier.Proof ({  /* put here the output of generate-proof */ });
    uint[] memory inputValues = new uint[](2);
    inputValues[0]=uint256(msg.sender);
    inputValues[1]=6;
    emit CanFactor(verify(inputValues,proof)==0);
}
```

circom

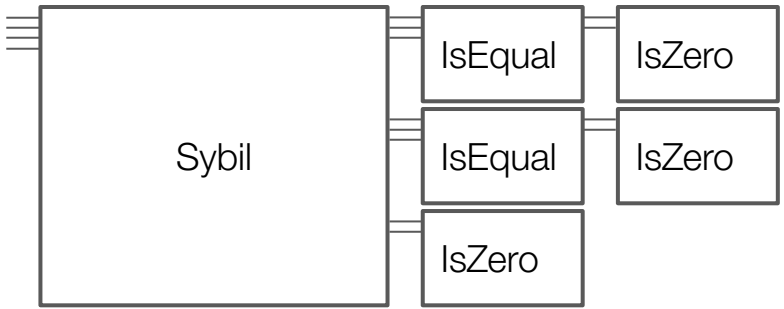
iden3 zk repos

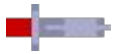
- snarkjs - zk-snarks libraries in javascript
- circom - circuit compiler to R1CS & witness code generation
- circomlib - sha256, pedersen, MiMC, SMTs, EdDSA,... + optimized js libs
- websnarks - webassembly generator for snarks
- rust-circom-experimental - rust port for circom
- current functionality:



circuit.circom

```
include "node_modules/circomlib/circuits/comparators.circom";
template Sybil() {
   signal private input p;
   signal private input q;
   signal input r;
   signal input id;

   component p_eq_1 = IsEqual();
   p_eq_1.in[0] <== p;
   p_eq_1.in[1] <== 1;
   p_eq_1.out === 0;

   component q_eq_1 = IsEqual();
   q_eq_1.in[0] <== q;
   q_eq_1.in[1] <== 1;
   q_eq_1.out === 0;

   component id_eq_0 = IsZero();
   id_eq_0.in <== id;
   id_eq_0.out === 0;
   r <== p * q;
}

component main = Sybil();
```

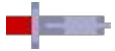




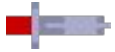
compile, setup, witness, proof and verifier

```
npm i -g circom snarkjs
npm i circomlib

circom
snarkjs setup --protocol=groth

vi input.json
{ “p”: “2”, “q”:”3”, “r”:”6”, “id”: <id> }

snarkjs calculatewitness
snarkjs proof
snarkjs generateverifier
```

generate proof in webassembly



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

node websnark/tools/buildwitness.js -i witness.json -o witness.bin
node websnark/tools/buildpkey.js -i proving_key.json -o proving_key.bin

cp websnark/example/index.html websnark/example/websnark.js
npm i -g live-server
live-server
```

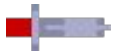


automatic witness generation

```
template Pow(N) {
   signal input in;
   signal output out;

   signal iterm[N-1];
   var i;
   iterm[0] <== in;
   for (i=1;i<N;i++) {
       iterm[i] <== iterm[i-1]*in;
   }
   out <== iterm[N-1];
}

component main = Pow(120);
```

```
const cirDef=await compiler(“pow.circom”)

const circuit=new snarkjs.Circuit(cirDef);

const witness = circuit.calculateWitness({
  "in": "3301"
});

// witness[1] == 3301^120
```

rust-circom-experimental



- some steps behind circom-js
- needs circom-lang formalization
- needs R1CS standardization
- finishing interoperability
- language experiments
- faster than js

twitter adria0

thanks!

level completed!

time for your magic

hands_on!