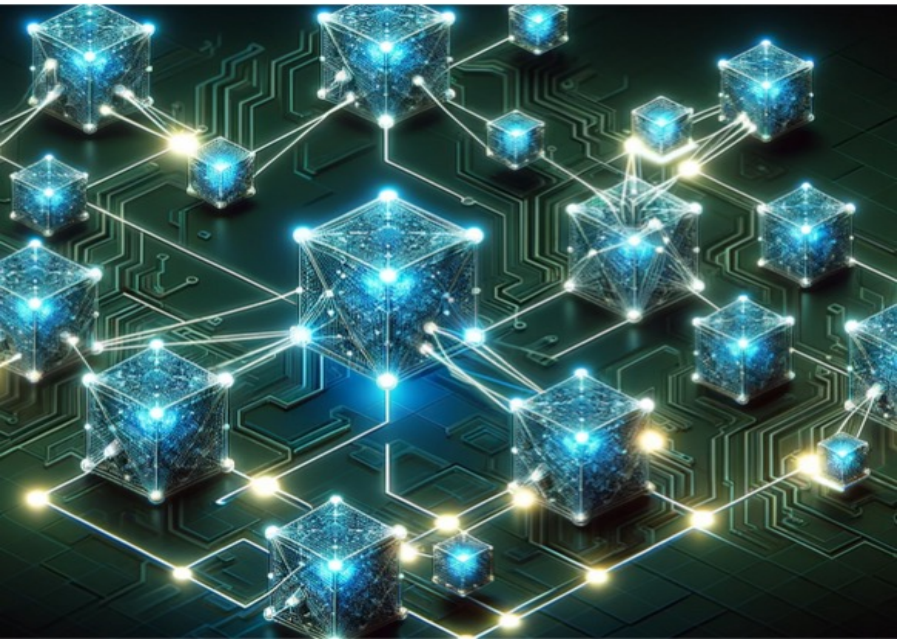
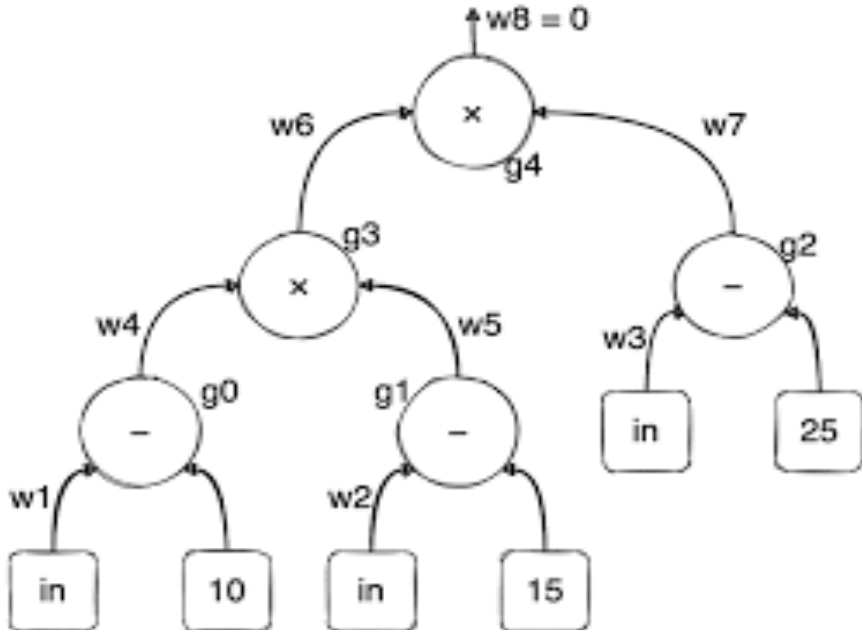
Compute Offchain, Verify Onchain: How to Build zk-DApps with Circom and ZoKrates





Alvaro Alonso, Johannes Sedlmeir, Jonathan Heiss

Tutorial at ICBC 2024

Who are we?

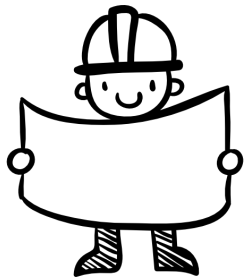
Alvaro Alonso

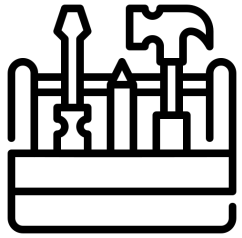
Dr. Johannes Sedlmeier

Dr. Jonathan Heiss

Agenda

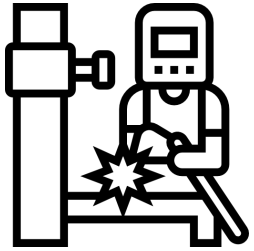
Motivation & Concept





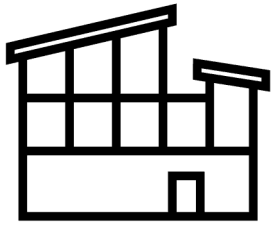
Why are zk-SNARKs relevant for blockchains?

Hands-on



How to build and deploy dApps with zk-DSLs?

Applications



What can be built with zk-SNARKs?

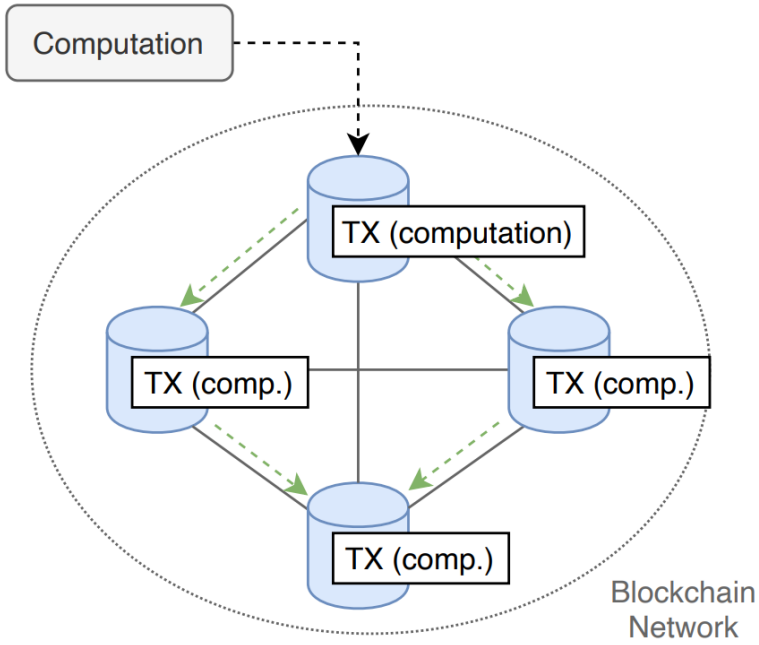
Blockchain-based decentralized applications often involve the processing of large amounts of or sensitive data

#	Pattern	Example use cases	References	Types of sensitive information
1	Payment	Bitcoin, central bank digital currencies	Nakamoto (2008), Dashkevich et al. (2020)	Individuals' and businesses' revenues, expenses, balances, turnover and business partners
2	Tamper-proof documentation	Notarization, Cardossier	EC (2021), Zavolokina et al. (2020)	Content and validity status of documents, information that could be sold on a market
3	Cross-organizational workflow management	Tradelens, MediLedger	Jensen et al. (2019), Mattke et al. (2019)	Frequency and type of processes, relationships between organizations involved
4	Ubiquitous services	Oracles (Chainlink), DeFi (Uniswap)	Al-Breiki et al. (2020), Wang et al. (2019), Werner et al. (2021)	Risk exposure associated with financial investments
5	Digital identities	Namecoin, German asylum case	Kalodner et al. (2015), Amend et al. (2021)	Individuals' names, addresses, health information, permissions and achievements
6	Tokenization	Ticketing (GUTs), investments and fractional ownership	Regner et al. (2019), Sunyaev et al. (2021), Whitaker and Kräussl (2020)	Individuals' and organizations' investment decisions and voting behaviour
7	Machine economy	Micropayments, economically autonomous robots	Jöhnk et al. (2021), Schweizer et al. (2020)	All of the above; machines are typically associated with organizations or individuals

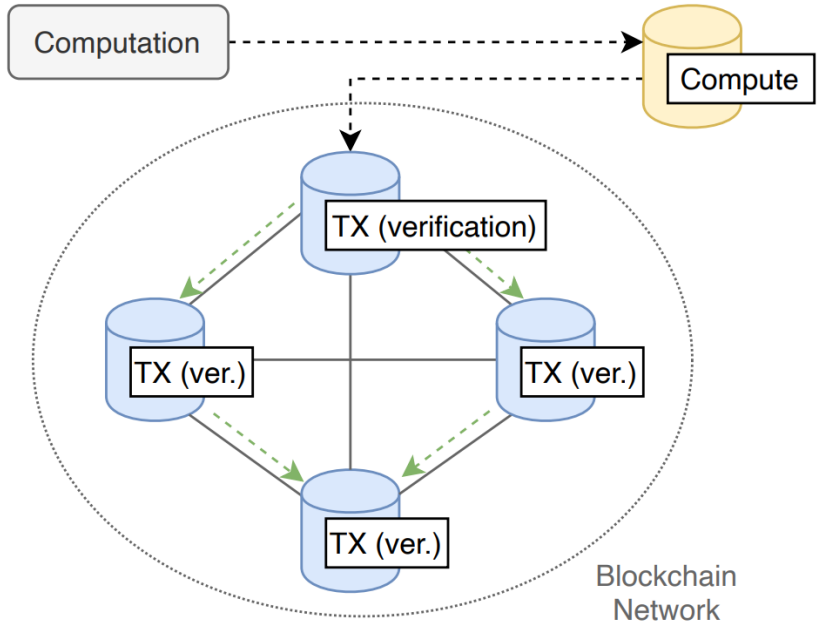
Sedlmeir, J., Lautenschlager, J., Fridgen, G., & Urbach, N. (2022). The transparency challenge of blockchain in organizations. *Electronic Markets*, *32*(3), 1779-1794.

Motivation

Default on Blockchains:
Replicated Computation



Verifiable Off-chain Computation (VOC):
Replicated Verification

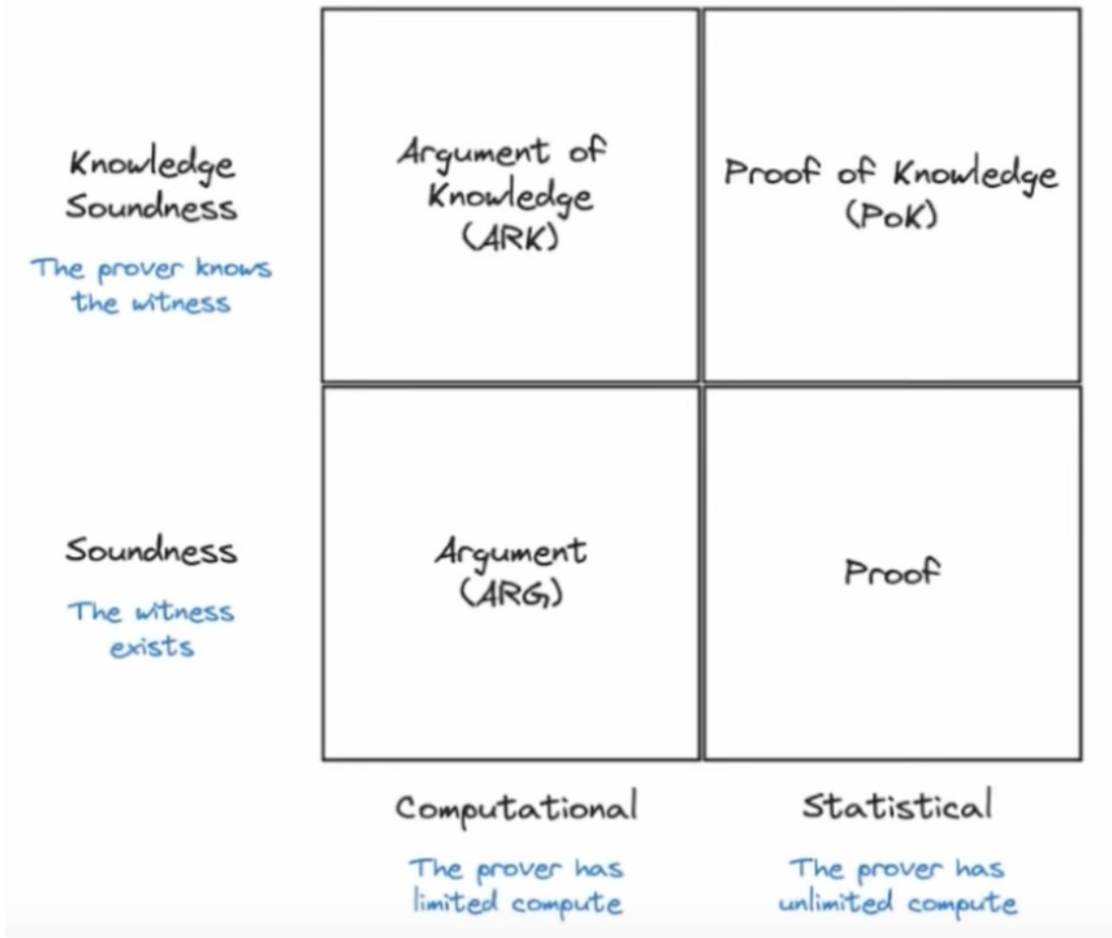


→ *Limitations: Privacy, Scalability*

Why are SNARKs important?

1. Arguments of Knowledge (ARK) → convince another party not only of the accuracy of a statement but also that the prover knows why it is true

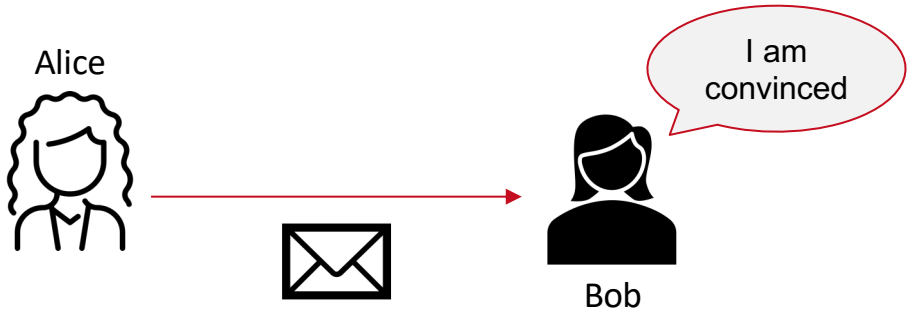
This is important in situations where conflict of interests arise and where replay is a potential issue.



[1] Nico Mohnblatt. (2024, Jan 18). ZK HACK IV - ZK Jargon Decoder [Video]. YouTube. URL https://www.youtube.com/watch?v=68AXfOR74cA&t=1465s

Why are SNARKs important?

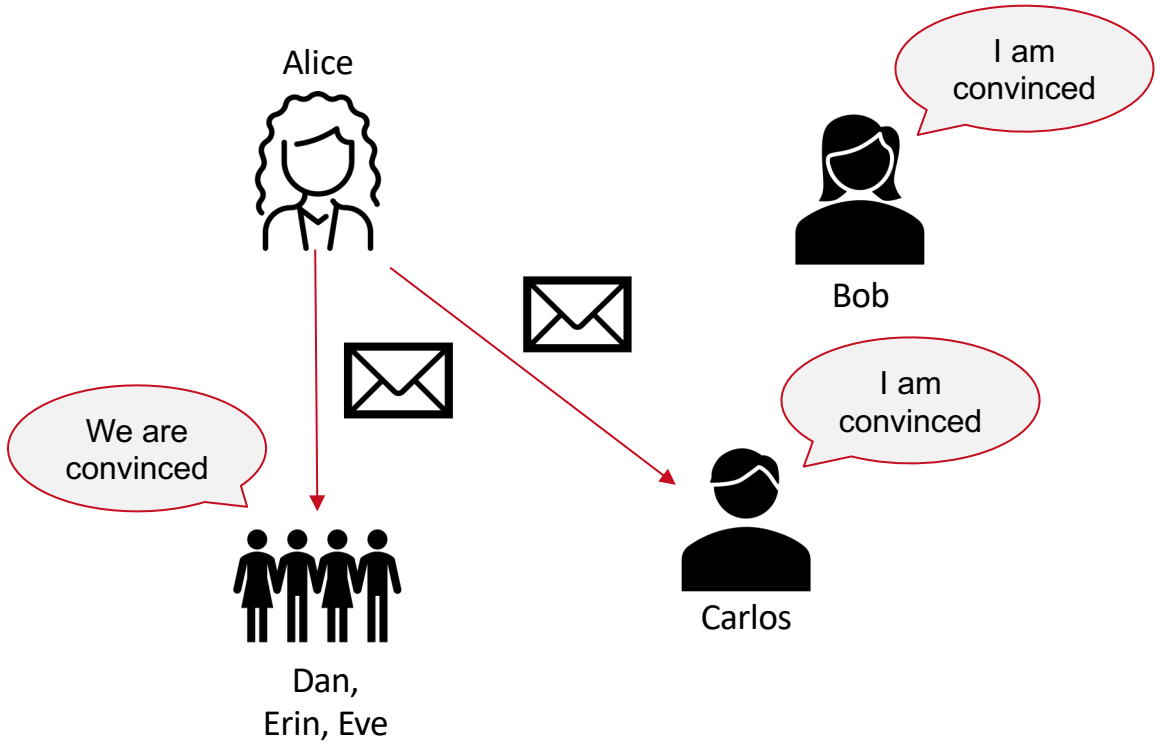
1. Arguments of Knowledge (ARK)
2. Non-Interactive



Note: In some scenarios, it may be desirable to have an interactive process because then only the designated verifier is convinced (and cannot sell verifiable data). For the blockchain case, however, non-interactiveness is critical to simultaneously convince all nodes (deterministically).

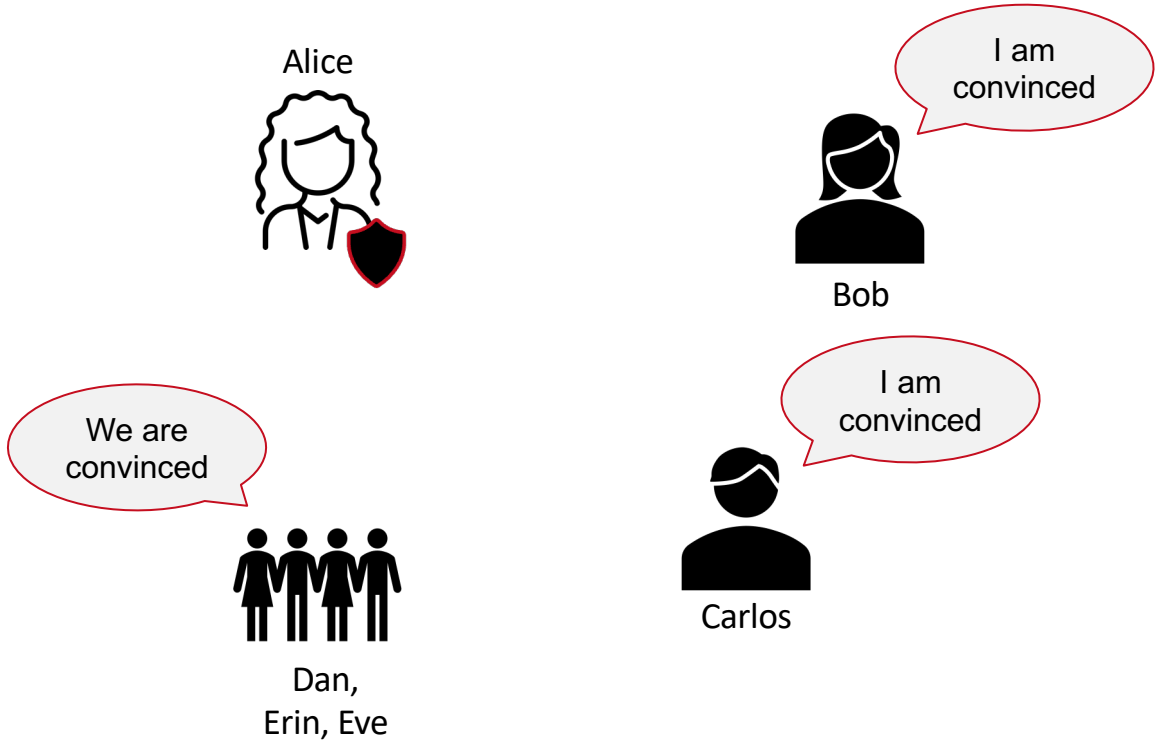
Why are SNARKs important?

1. Arguments of Knowledge (ARK)
2. Non-Interactive
3. Succinct – its verification is much shorter than the verification of an actual proof



Why are zk-SNARKs important?

1. Arguments of Knowledge (ARK)
2. Non-Interactive
3. Succinct
4. Zero Knowledge – the verifier(s) will learn nothing other than the validity of the statement



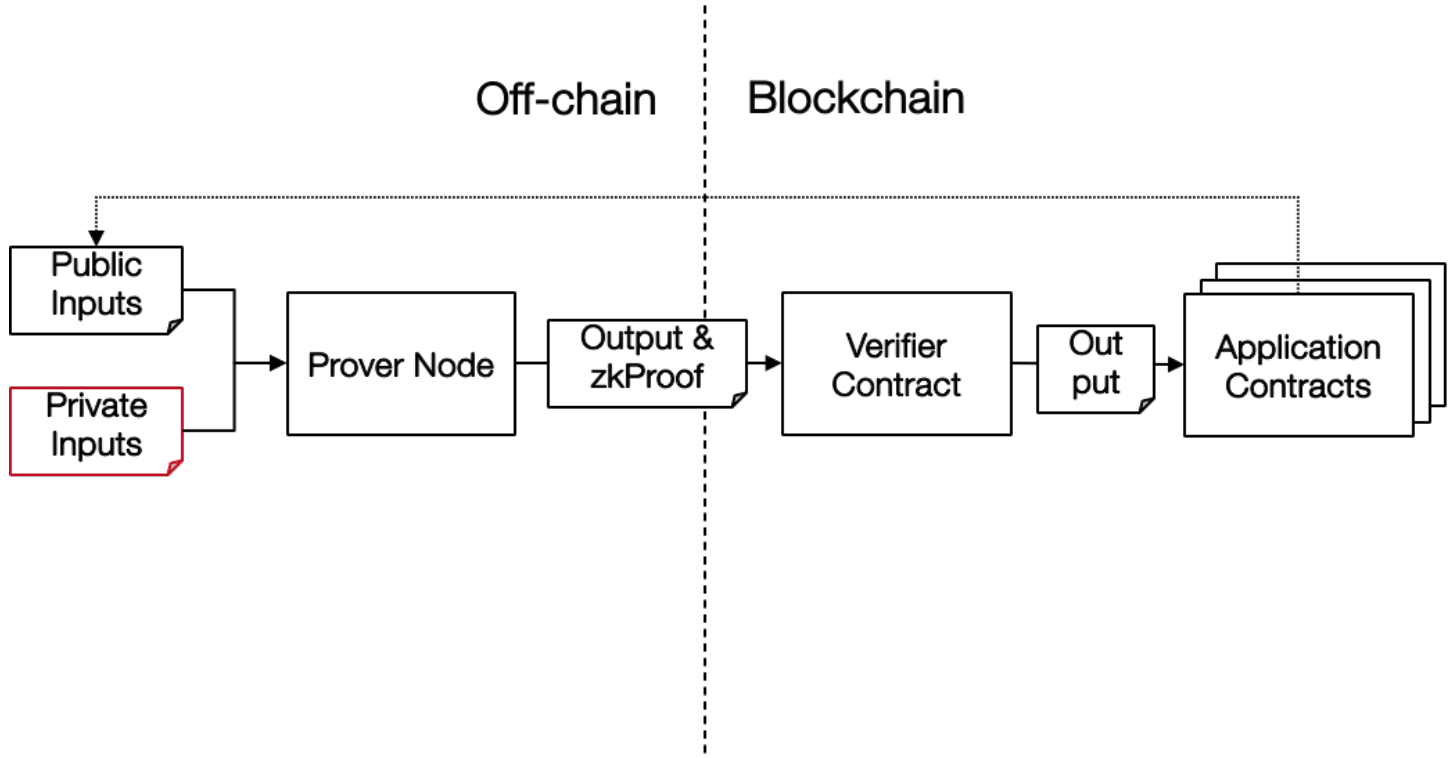
Zk-DApps

Build decentralized applications (dApp) where parts of the logic are executed off-chain.

Computational correctness can be verified on-chain through ZKPs like zkSNARKs.

→Privacy, Scalability

Examples: zk-Rollups, zk-Exchanges, zk-(Decentralized) Federated Learning, …



Why a Domain Specific Language (DSL) toolkit?

- Current programming languages do not satisfy this paradigm
- Zk-SNARK "circuits" are very large
- Requires expert knowledge
- Requires of low-level optimizations to run efficiently
- The process stages are well-defined

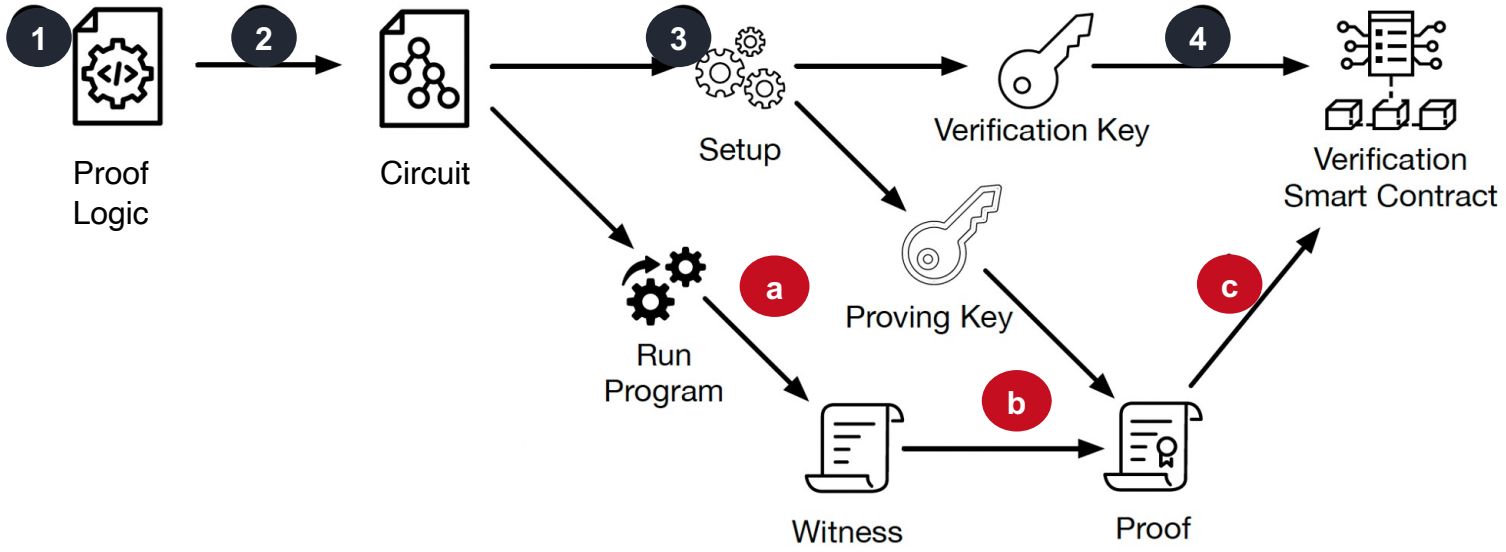
zk-SNARKs Development Workflow

One time operations

1. Define Program Logic
2. Compile into Algebraic Circuit
3. Execute Setup (get Key Pair)
4. Export Verifier Contract

Recurring Operations

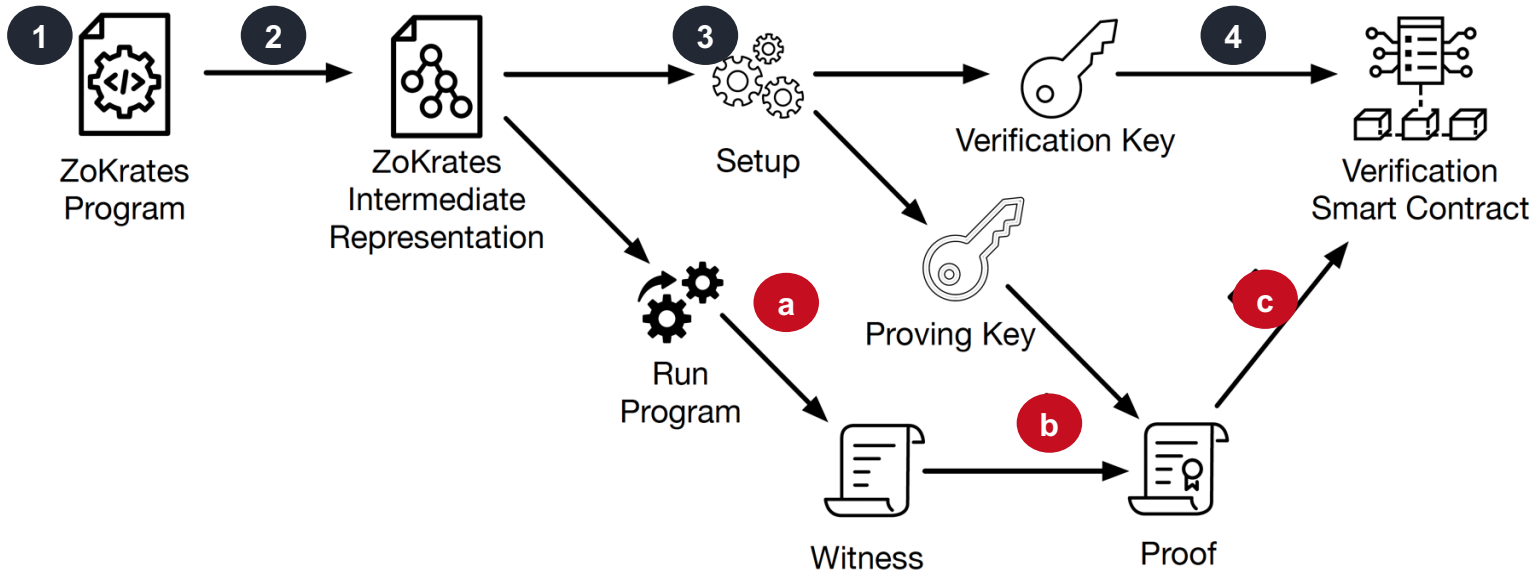
a. Compute Witness
b. Generate Proof
c. Verify Proof On-chain



[1] Jakob Eberdhardt. (2021). Scalable and Privacy-preserving Off-chain Computations

ZoKrates

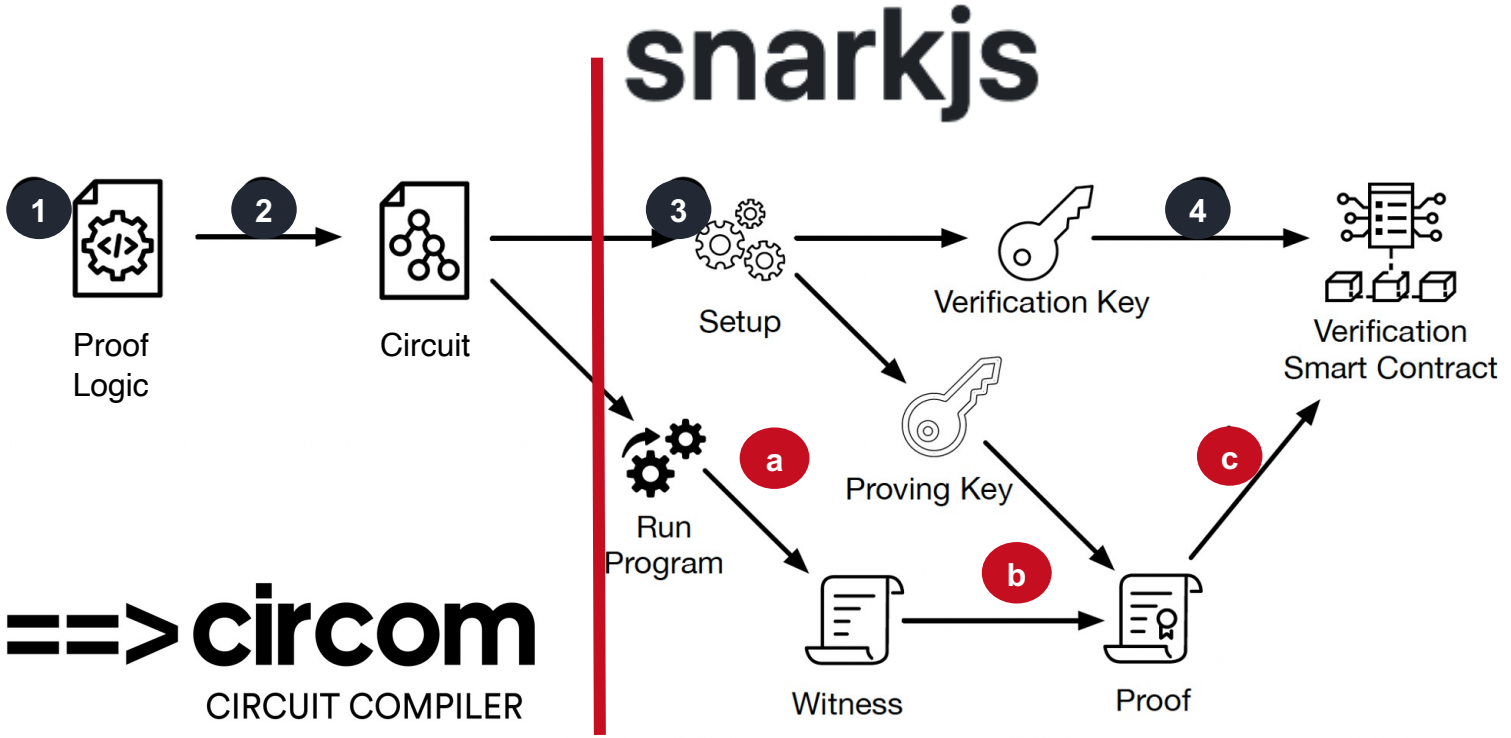
- **One-stop solution** for Verifiable Offchain Computations (VOC)
  - Supports all steps 1-c
- Python-/Rust-like syntax
- Opinionated Compiler



[1] Jakob Eberdhardt. (2021). Scalable and Privacy-preserving Off-chain Computations

Circom / SnarkJS

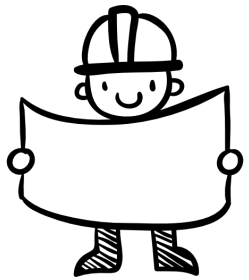
- Modular tool for CIRcuit COMpilation
- HDL

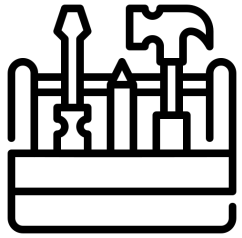


[1] Jakob Eberdhardt. (2021). Scalable and Privacy-preserving Off-chain Computations

Agenda

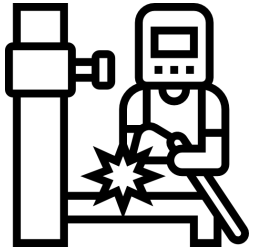
Motivation & Concept





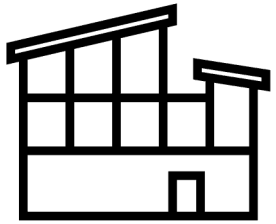
What are Zero-Knowledge Proofs (ZKPs) and why are they relevant for blockchains?

Hands-on



How to use zk-DSLs?

Applications

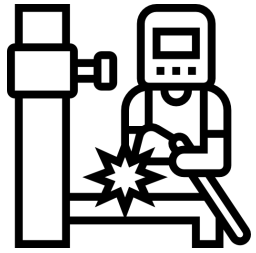


What can be built with ZKPs?

Learning Objectives

- Understand constraint systems
- Start an interactive environment
- Write our first program in ZoKrates & Circom
- Generate & verify proofs

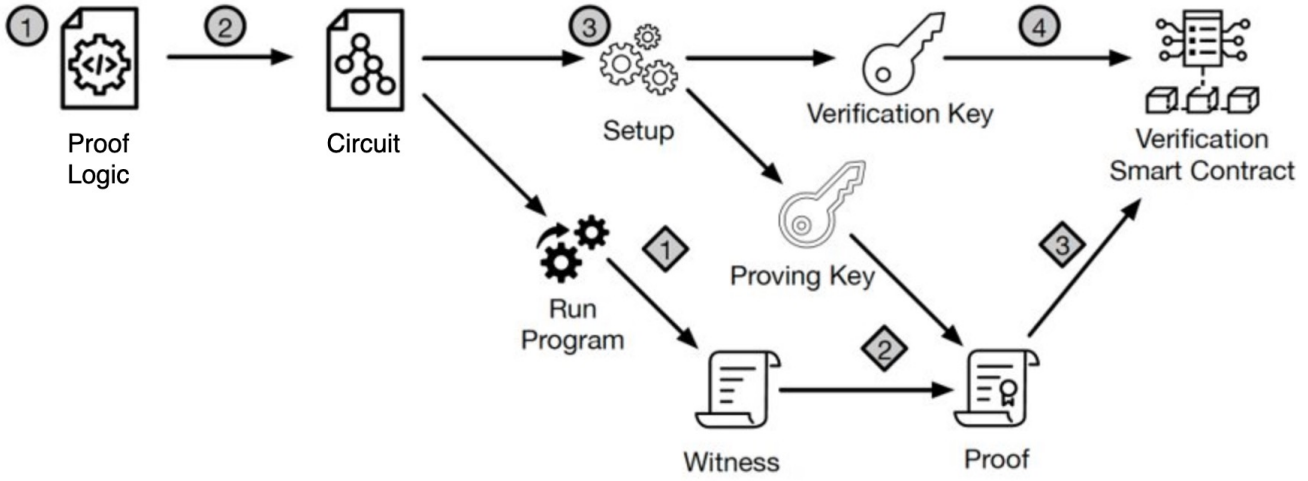
Hands-on



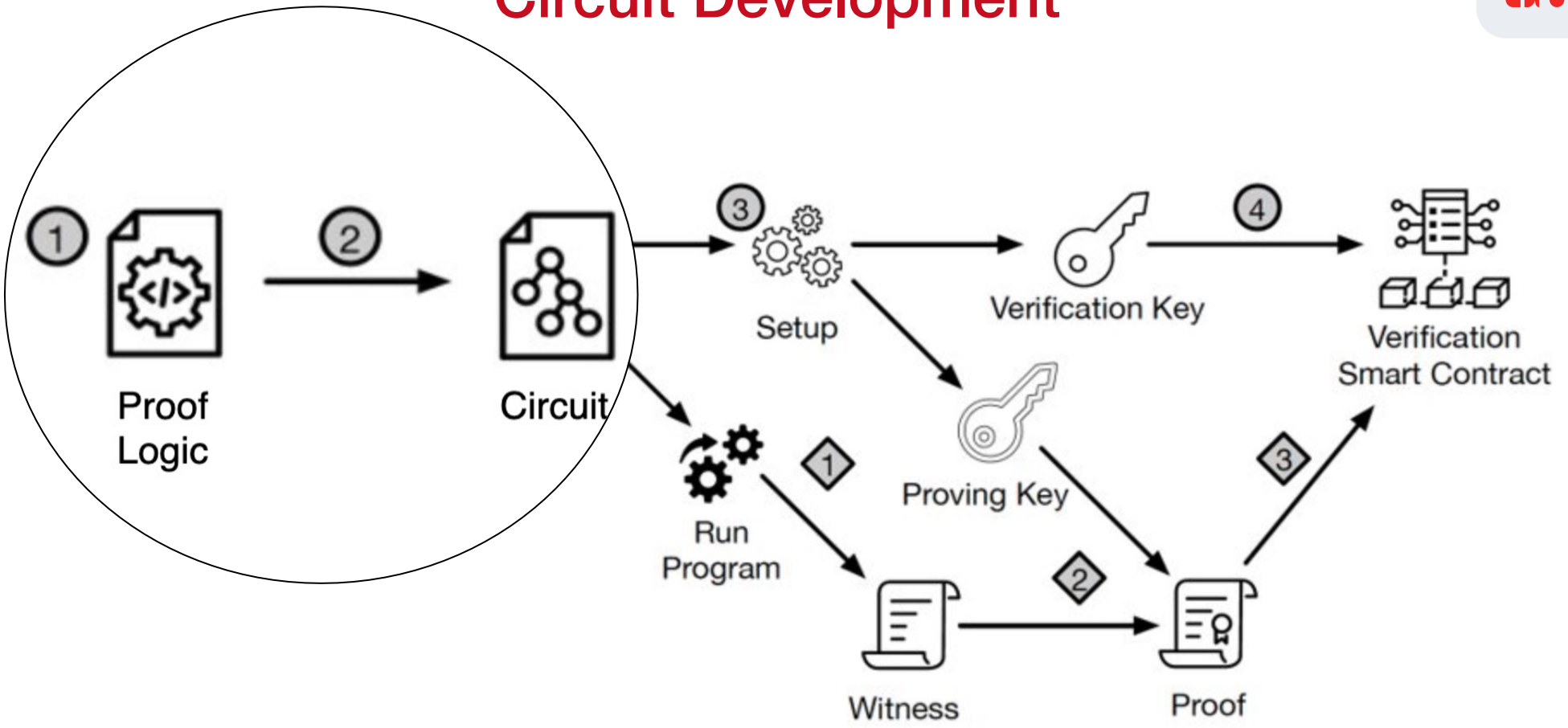
How to use zk-DSLs?

Learning Objectives

- Understand constrain systems (1&2)
- Write our first program in ZoKrates & Circom (1&2)
- Proof & Verify the proofs created



Circuit Development



How do you formally verify a program ?

Arithmetization is the process of turning a generic statement or question into a set of equation to be verified or solved[1]

In zk-SNARKs, we need arithmetization to for several reasons:

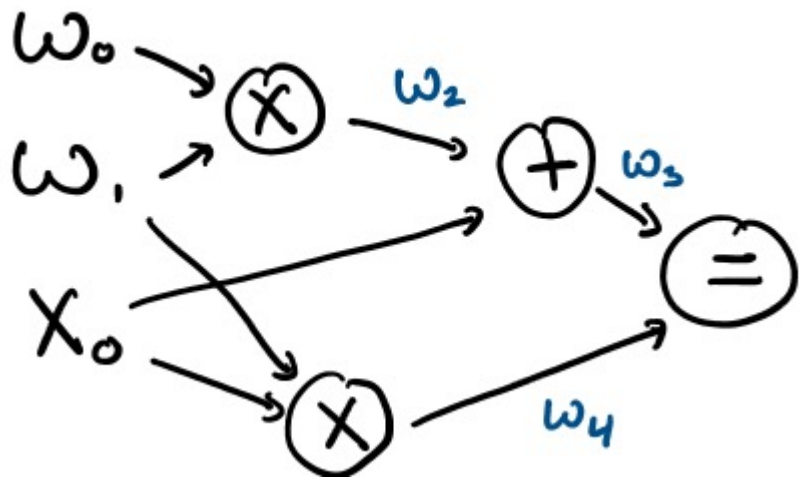
1. Guarantee the correctness of the computation
2. We need the set of equations to follow the same structure
3. Succinctness - The verification of the computation can be done efficiently
4. It allows for Zero-Knowledge

To satisfy all the above points we use algebraic constrain systems. An specially for this tutorial we will use the **Rank 1 Constrain System (R1CS)**.

[1] Nico Mohnblatt. (2024, Jan 18). ZK Jargon Decoder [Online]. URL https://nmohnblatt.github.io/zk-jargon-decoder/definitions/arithmetization.html

R1CS: a common Arithmetic Circuit format

- Circuit form: Directed Cyclic Graph (DAG) of "wires" and "gates"
  - Gates: nodes
  - Wires: edges



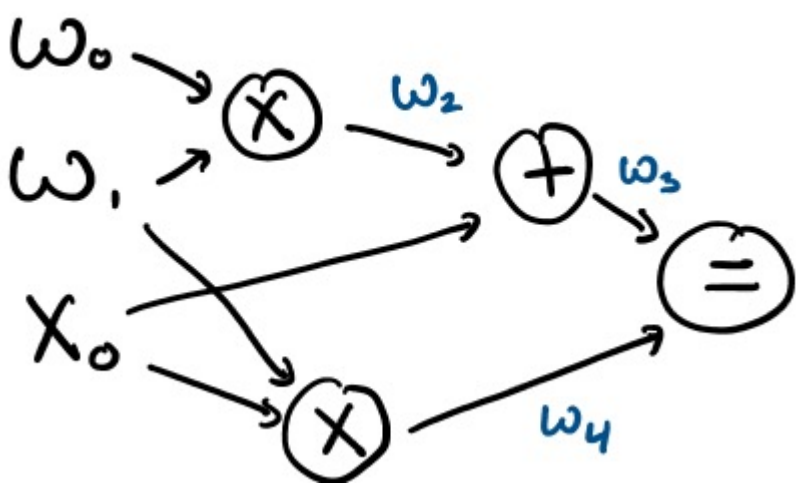
<u>The number of gates determine the memory requirements and proving time</u>

[1] Pratyush Mishra and Alex Ozdemir (2023). Zero Knowledge Proofs MOOC: Lecture 3: Programming ZKPs [Online] URL https://zk-learning.org/assets/lecture3-2023.pdf

R1CS: a common Arithmetic Circuit format

- Circuit form: Directed Cyclic Graph (DAG) of "wires" and "gates"
  - Gates: nodes
  - Wires: edges
- Matrix form: n equations of form: $\alpha \times \beta = \gamma$
  - where α, β, γ are affine (linear) combinations of variables

<u>The number of gates determine the memory requirements and proving time</u>



- $w_0 \times w_1 = w_2$
- $w_3 = w_2 + x_0$
- $w_1 \times x_0 = w_4$
- $w_3 = w_4$

[1] Pratyush Mishra and Alex Ozdemir (2023). Zero Knowledge Proofs MOOC: Lecture 3: Programming ZKPs [Online] URL https://zk-learning.org/assets/lecture3-2023.pdf

Circom

- Wires -> Signals
- Gates -> Gates

```
// Primitive Types
signal aSignal = 42;
signal arraySignal[254];

// Composite Types


// functions
Component myComponent(){
    signal input in1;
    signal input in2;
    …
    signal output out;
}

// for loops
signal arraySignal[254];
```

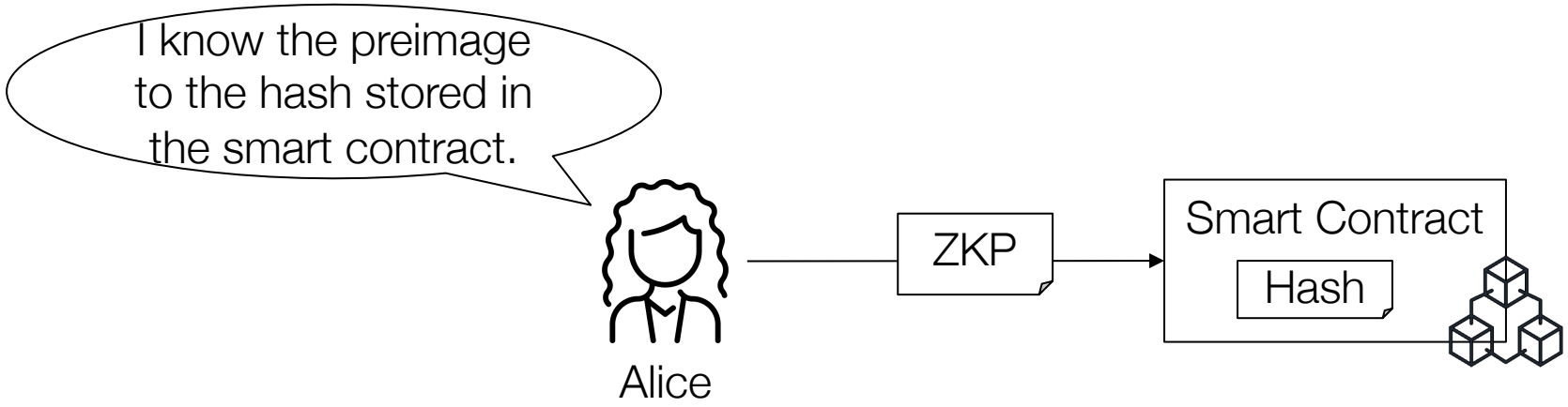
```
// Imports from Standard Library
include "circomlib/poseidon.circom";
include "https://github.com/0xPARC/circom-
secp256k1/blob/master/circuits/bigint.circom"
```

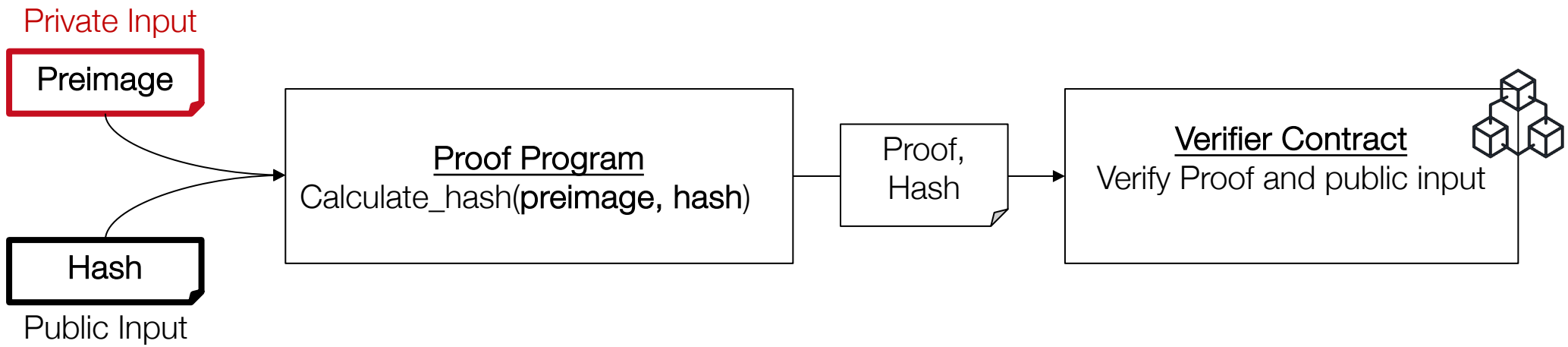
```
let dc = 0;
for (dc=0; cd<256; dc++) {
   arraySignal[dc] <== 2;
}

// conditional expressions
signal input condition1;
signal input condition2;
component conditionCheck = IsEqual();
IsEqual.in[0] <== condition1;
IsEqual.in[1] <== condition2;

signal output out
out <== conditionCheck.out * 42 + (1 -
conditionCheck.out) * 7;
```

ZoKrates Language

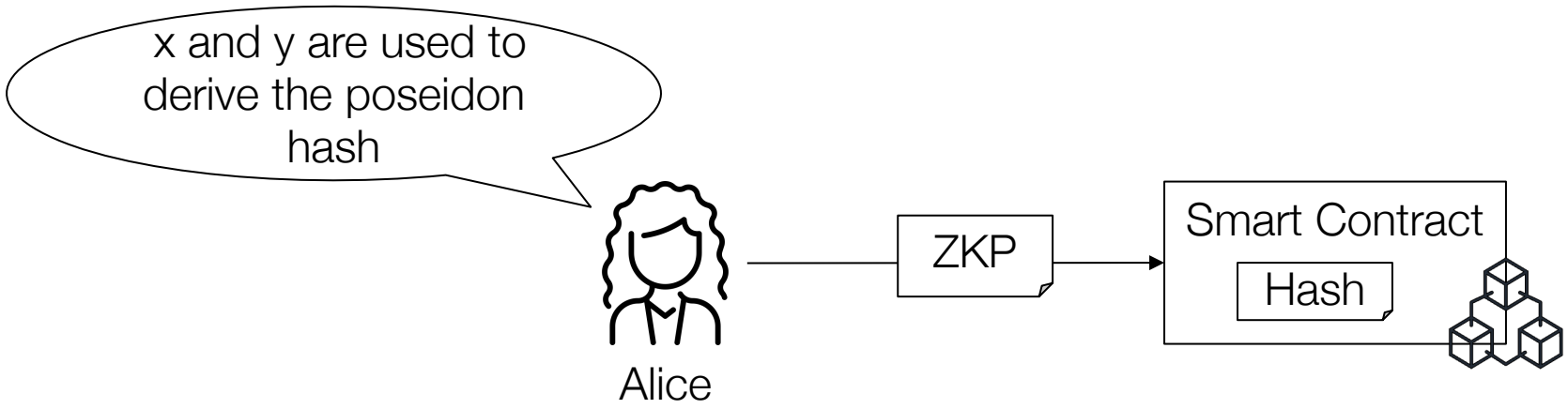
```
// Imports from Standard Library
from "ecc/babyjubjubParams" import
BabyJubJubParams
import "hashes/sha256/512bit" as sha256;

// Primitive Types
field a = 42;
bool b;
u32 = 255;

// Composite Types
field[256] c = [0; 256]

// functions
def incr(private field a, field b) -> field {
   return a + b;
}

// conditions & assertions
assert(2 + 2 == 4);

// for loops
for const u32 i in 0..4 {
   res = res + i;
}

// conditional expressions
if x == 1 {
 cheap(x)
} else {
 expensive(x) // both branches executed
}
```

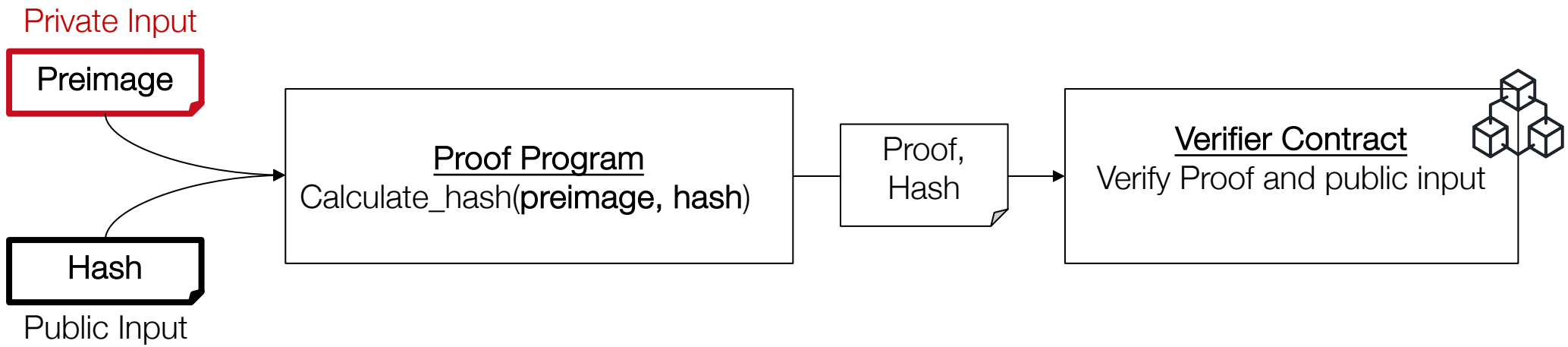
Proving Knowledge of a Hash's Preimage





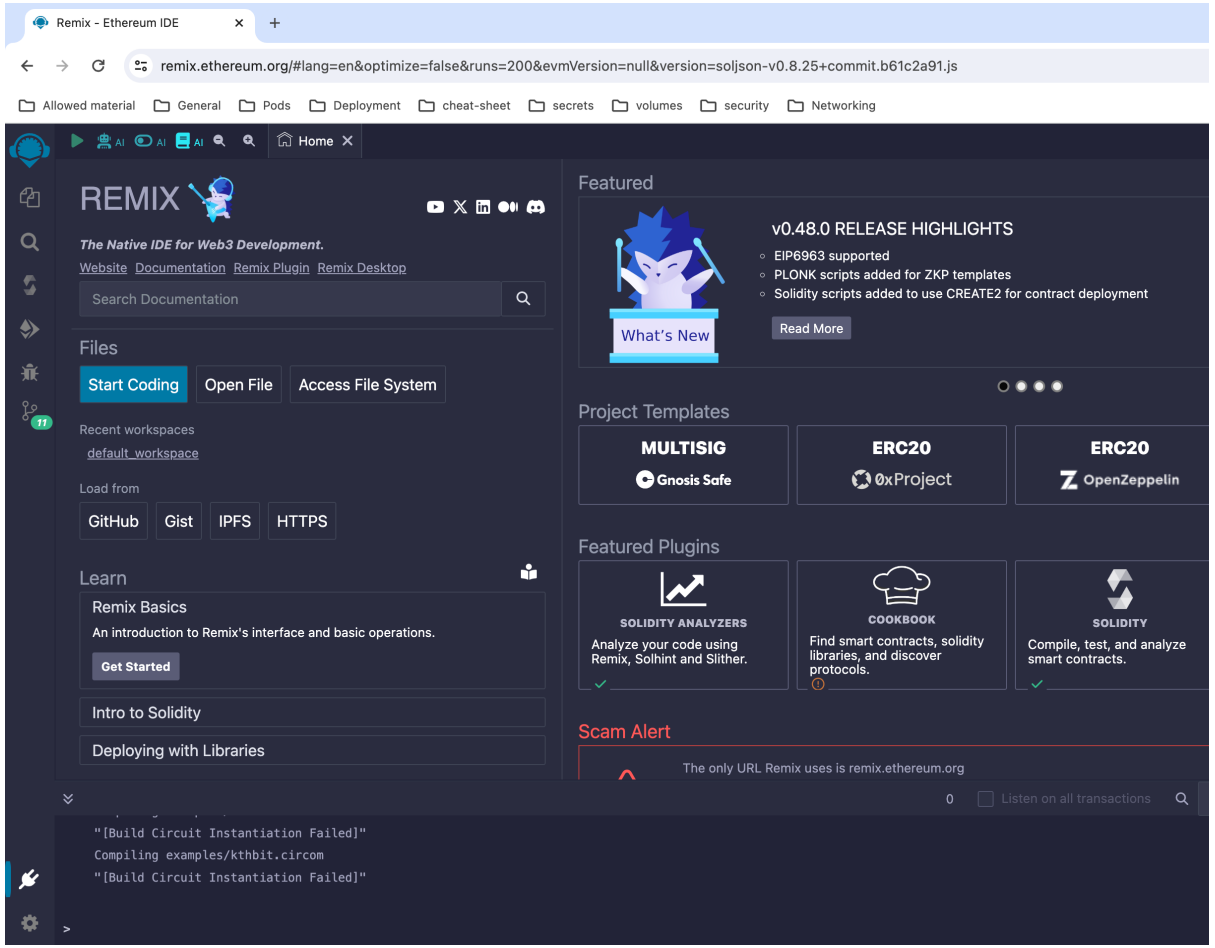
Ex.: Proving Knowledge of a Poseidon Hash





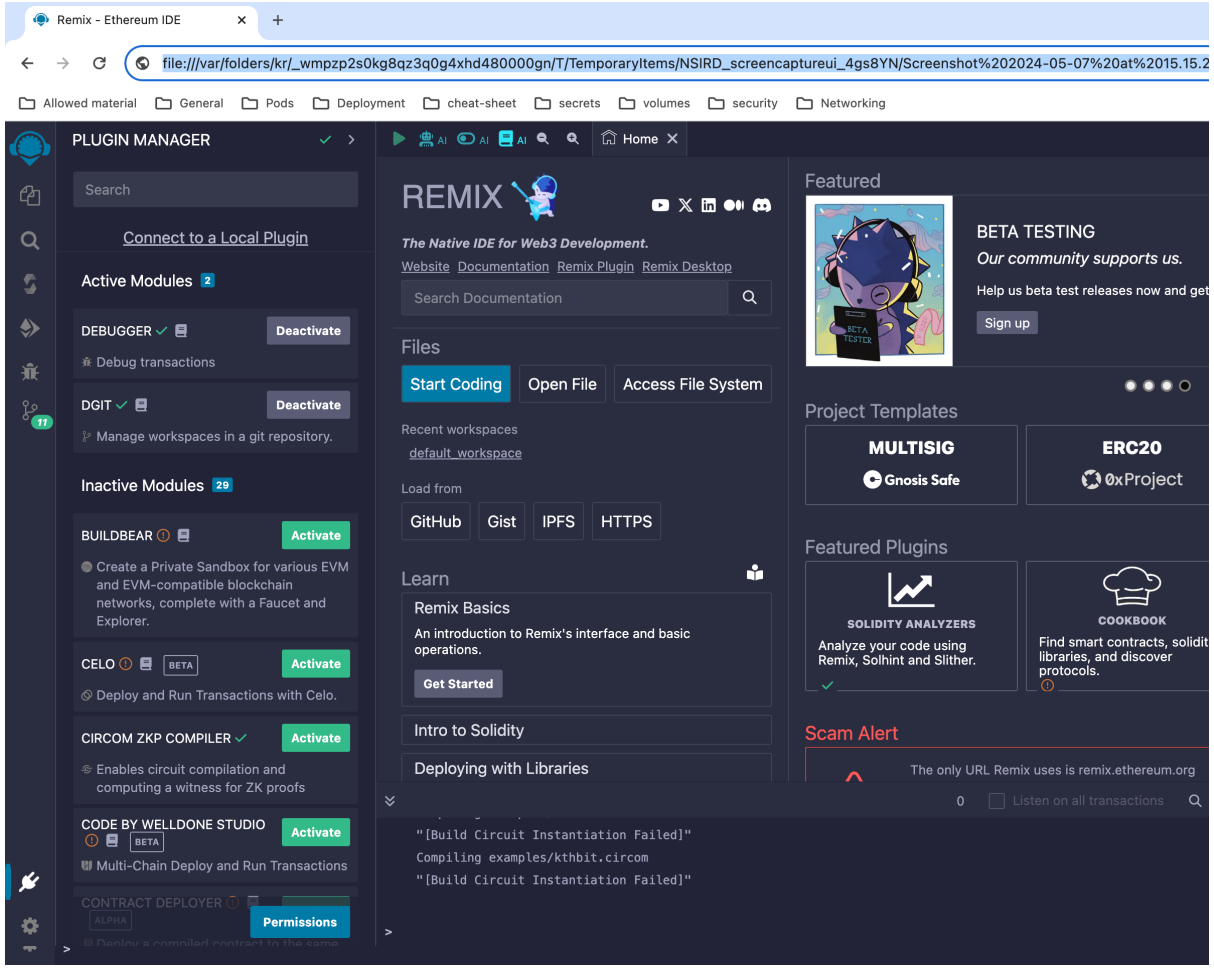
Setup

1. Go to remix.ethereum.org



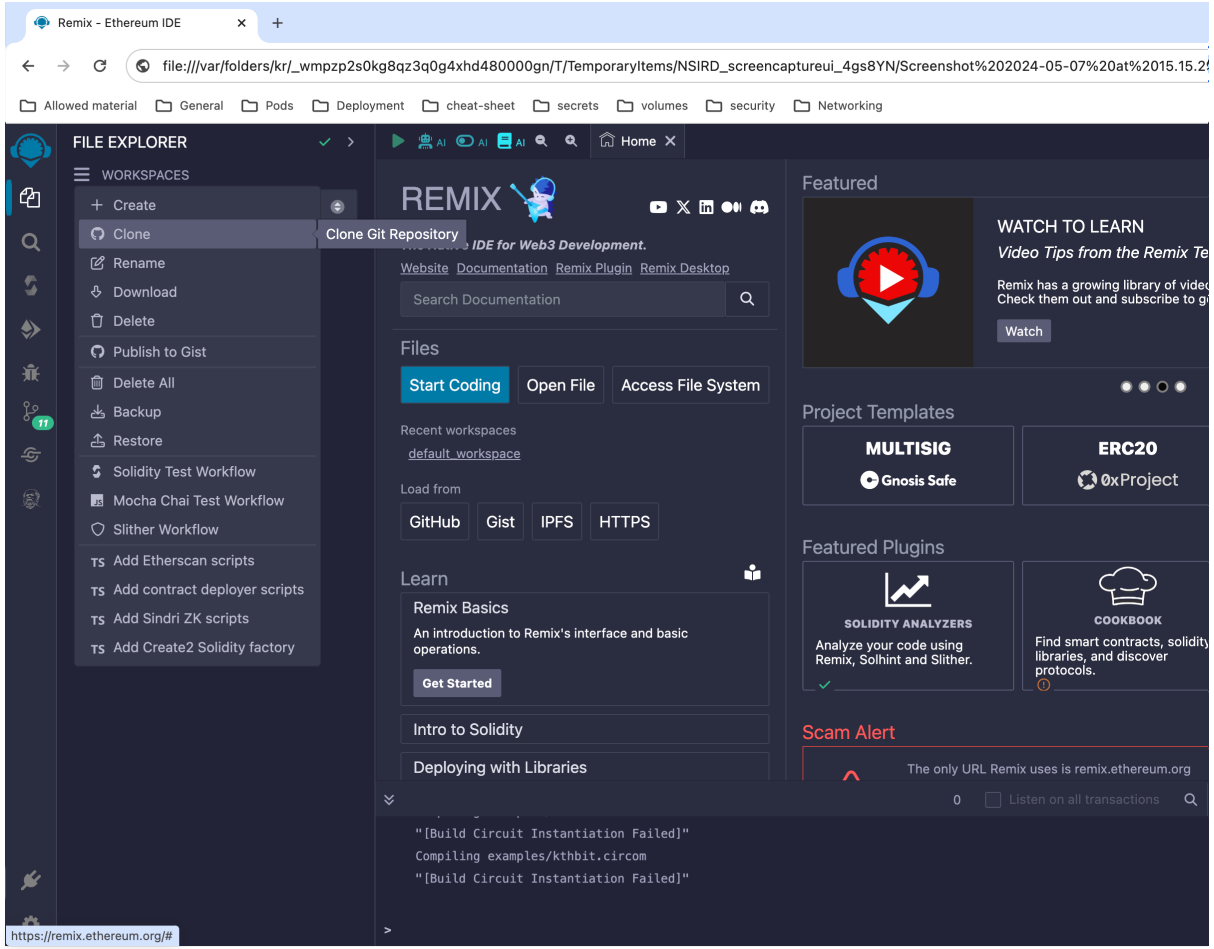
Setup

1. Go to remix.ethereum.org
2. Install the ZoKrates and Circom plug-in



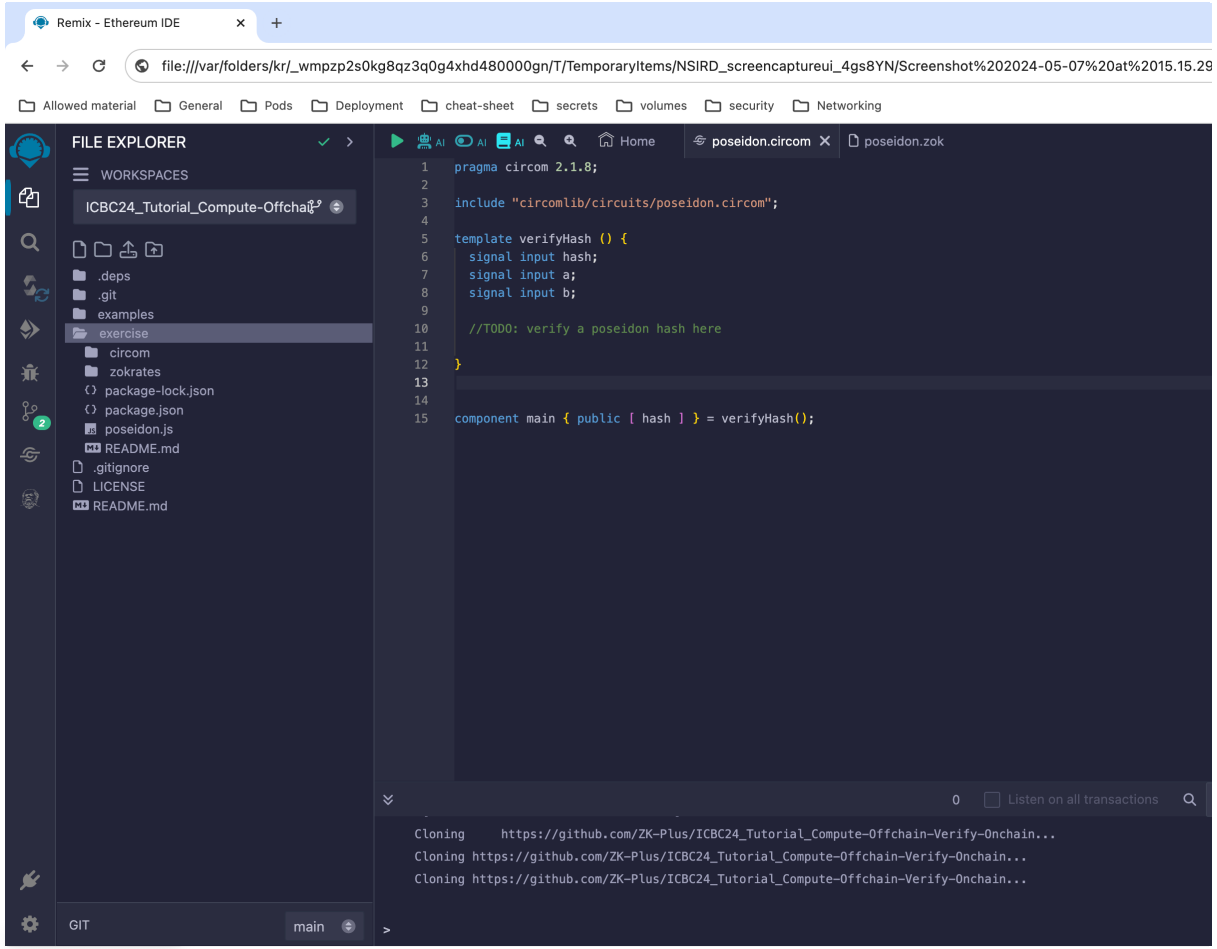
Setup

1. Go to remix.ethereum.org
2. Install the ZoKrates and Circom plug-in
3. Import following repo to remix: https://github.com/ZK-Plus/ICBC24_Tutorial_Compute-Offchain-Verify-Onchain



Setup

1. Go to remix.ethereum.org
2. Install the ZoKrates and Circom plug-in
3. Import following repo to remix: https://github.com/ZK-Plus/ICBC24_Tutorial_Compute-Offchain-Verify-Onchain
4. Program templates are in the exercise folder



Resources

- Standard library (stdlib):
  - **Circom**: https://github.com/iden3/circomlib
  - **ZoKrates**: https://github.com/Zokrates/ZoKrates > zokrates_stdlib
- Documentation:
  - **Circom**: https://docs.circom.io/circom-language/signals/
  - **ZoKrates**: https://zokrates.github.io/
- Getting Help:
  - Circom's Telegram group: https://t.me/iden3io
  - ZoKrates Gitter group: https://app.gitter.im/#/room/#ZoKrates_Lobby:gitter.im
- Test example:
  - Preimage: 1, 2
  - Hash: 7853200120776062878684798364095072458815029376092732009249414926327459813530

Computing a Poseidon hash

Circom

```
pragma circom 2.1.6;

include "circomlib/poseidon.circom";

template verifyHash () {
        signal input hash;
        signal input a;
        signal input b;

  component hasher = Poseidon(2);
  hasher.inputs[0] <== a;
  hasher.inputs[1] <== b;

  hasher.out === hash;
  log("hash", hasher.out);
}

component main { public [ hash ] } = verifyHash();
```

ZoKrates

```
import "hashes/poseidon/poseidon" as poseidon;

def main(field hash, private field[2] preimage) {

  field digest = poseidon(preimage);
  assert(digest == hash);
}
```

Non-deterministic advice

Some operations are complex to do in a circuit but their correctness can easily be checked.

In this case, it makes sense to supply the result of this computation to the circuit as external information that needs to be validated.

```
1   pragma circom 2.1.8;
2
3
4   template squareRoot () {
5     signal input a;
6     signal input b;
7
8     a * a === b;
9   }
10
11
12  component main { public [ b ] } = squareRoot();
13
```

Number of iterations must be known at compile time.

```
15
16  def merkleTreeProof<DEPTH>(u32[8] root, MerkleTreeProofStruct<DEPTH> proof) -> bool {
17      // Start from the leaf
18      u32[8] mut digest = proof.leaf;
19
20      // Loop up the tree
21      for u32 i in 0..DEPTH {
22          (u32[8], u32[8]) s = select(proof.directionSelector[i], digest, proof.path[i]);
23          digest = hash(s.0, s.1);
24      }
25
26      return digest == root;
27  }
28
29  const u32 TREE_DEPTH = 2;
30
31  def main(u32[8] tree_root, MerkleTreeProofStruct<TREE_DEPTH> merkle_proof) {
32      assert(merkleTreeProof(tree_root, merkle_proof));
33  }      You, 13 minutes ago • merkle proof example
```

α	β	γ
x_1	y_1	z_1
x_2	y_2	z_2
…	…	…
x_n	y_n	z_n

Branching

1. Both branches are always executed.

```
def main(field x) -> field {
    return if x == 1 {
        cheap(x)
    } else {
        expensive(x) // both branches executed
    };
}
```

[] ZoKrates Documentation [Online] URL https://zokrates.github.io/language/control_flow.html

Branching

1. Both branches are always executed.

2. An unsatisfied constraint inside any branch will make the whole execution fail, even if this branch is not logically executed.

```
def main(field x) -> field {
    return if x == 0 {
        0
    } else {
        1 / x // executed even for x := 0, which leads to the execution failing
    };
}
```

[] ZoKrates Documentation [Online] URL https://zokrates.github.io/language/control_flow.html

Branching

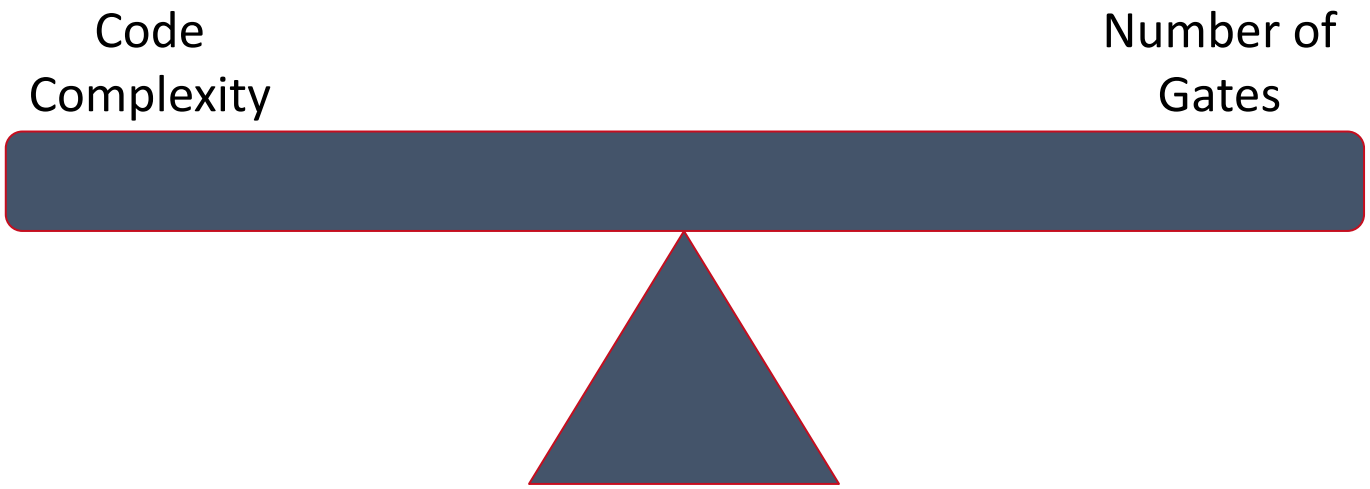
1. Both branches are always executed.

2. An unsatisfied constraint inside any branch will make the whole execution fail, even if this branch is not logically executed.

3. No break / continue. If we need to break a loop and return a specific value, we can

```
pragma circom 2.1.8;

include "circomlib/comparators.circom";

template ReturnIfCondition () {
    signal input inputs[5];
    signal input index;

    component checkers[5];

    signal runningReturnSum[6];
    runningReturnSum[0] <== 0;

    signal output value;

    for (var i=0; i<5; i++){
        checkers[i] = IsEqual();
        checkers[i].in[0] <== i;
        checkers[i].in[1] <== index;

        runningReturnSum[i+1] <== runningReturnSum[i] + checkers[i].out * inputs[i];
        log("Running return sum at index: ", runningReturnSum[i+1]);
    }

    value <== runningReturnSum[5];

}

component main = ReturnIfCondition();

/* INPUT = {
    "inputs": ["1", "2", "3", "4", "5"],
    "index": "2"
} */
```

Circom Documentation [Online] URL https://docs.circom.io/circom-language/signals/


Code
Complexity
Number of
Gates

Support of Cryptographic Primitives

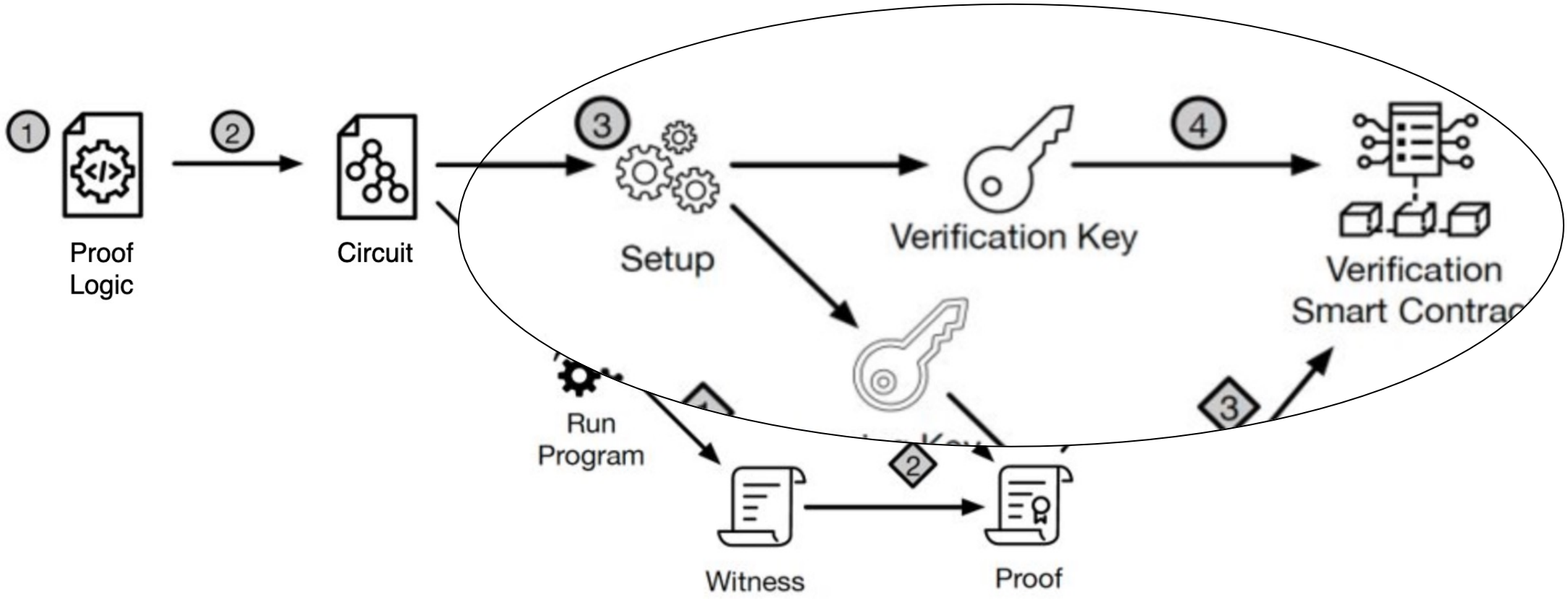
Support of Cryptograp

Standard library

3rd party library

	Circom	ZoKrates
Hashes		
SHA256	✓	✓
SHA3	✓	✓
Poseidon	✓	✓
Pedersen	✓	✓
MimC	✓	✓
Poseidon2	✓	✗
Keccak	✓	✗
Signatures	✓	
EdDSA	✓	✓
ECDSA	✓	✗
Encryption Algoritms		
ElGamal	✓	✗
RSA	✓	✗
AES	✓	✗

Setup & Deployment



Setup

The trusted setup is done in two steps:

1. "Phase 1", does not depend on the program and is called Powers of Tau.

2. "phase 2" is circuit-specific, so it should be done separately for each different program.

Check Circom's and ZoKrates' documentation for information on the process.

snarkjs

```
# export proving key
$ snarkjs groth16 setup circuit.r1cs pot12_final.ptau
proving_key.zkey

# export verification key
$ snarkjs zkey export verificationkey proving_key.zkey
verification_key.json
```

ZoKrates

```
# export keys from final parameters (proving and
verification key)
$ zokrates mpc export -i final.params
```

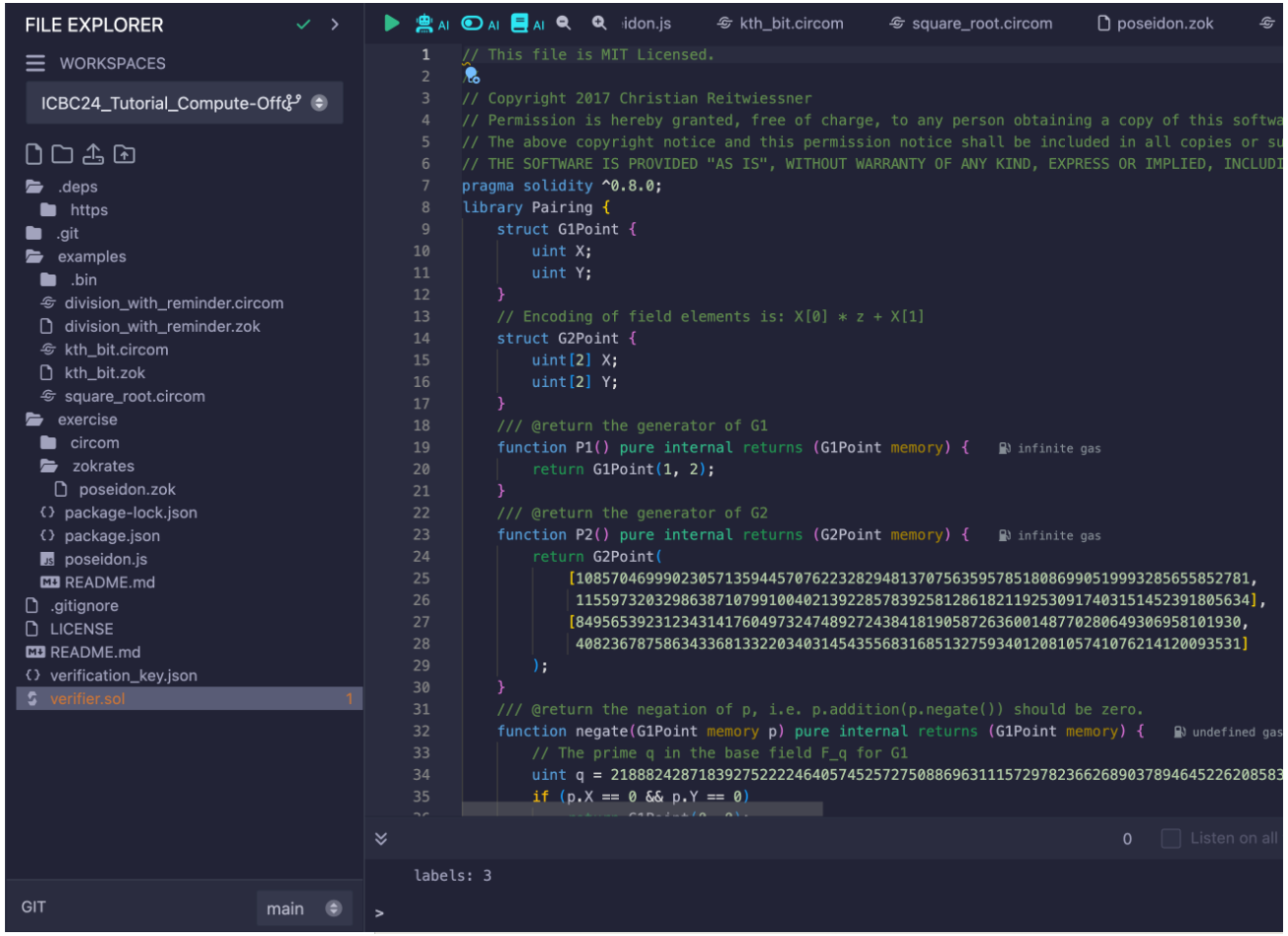
Export verifier contract

Snarkjs:

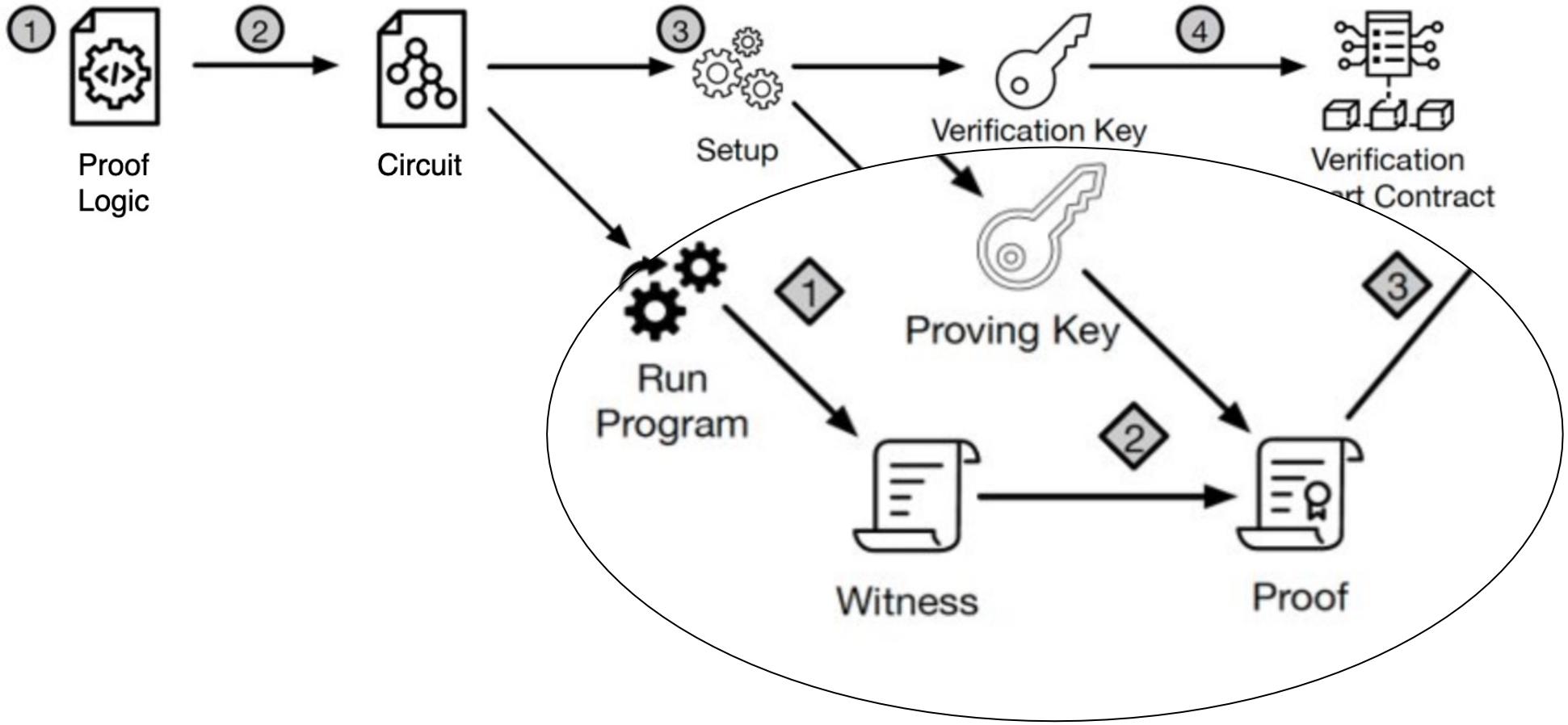
```
$ snarkjs zkey export
solidityverifier proving_key.zkey
verifier.sol
```

ZoKrates:

```
$ zokrates export-verifier -i
verification_key.key -o
verifier.sol
```



Proving


1
Proof
Logic
2
Circuit
3
Setup
Verification Key
4
Verification
Contract
Run
Program
1
Proving Key
3
2
Witness
Proof

Generate the input arguments

Through popular cryptography libraries:

- Circomlib (js)
- ZnaKes (python)
- Hashlib (python)
- ...

```
3
4   const circomlibjs = require("circomlibjs");
5
6   (async function () {
7       const preimage = [1, 2]
8       const poseidon = await circomlibjs.buildPoseidon();
9       const hash = poseidon.F.toString(poseidon(preimage));
10      console.log(`poseidon hash of preimage(${preimage}) is: ${hash}`);
11  })()
12
```

Proving

1. Automatically in JS

```
const snarkjs = require("snarkjs");
const fs = require("fs");

async function run() {
    const { proof, publicSignals } = await snarkjs.groth16.fullProve(
        {a: 10, b: 21},
        "circuit.wasm",
        "proving.zkey"
    );
}
```

```
import { initialize } from "zokrates-js";

initialize().then((zokratesProvider) => {

  // witness computation
  const { witness, output } = zokratesProvider.computeWitness(circuit, ["2"]);

  // generate proof
  const proof = zokratesProvider.generateProof(
    circuit.program,
    witness,
    keypair.pk
  );
}
```

Proving

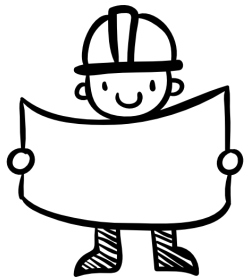
1. Automatically in JS
2. Manually with the cli

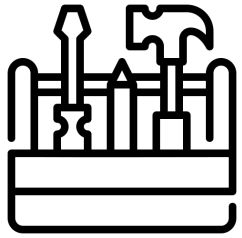
```
$ node generate_witness.js multiplier2.wasm input.json
witness.wtns
$ snarkjs groth16 prove proving_key.zkey witness.wtns
proof.json public.json
```

```
$ zokrates compute-witness -i circuit -a 1 2 …
$ zokrates generate-proof -i circuit -p proving_key -w
witness
```

Agenda

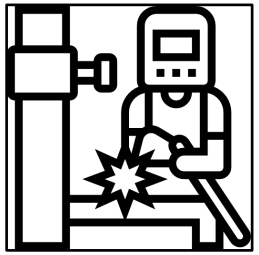
Motivation & Concept





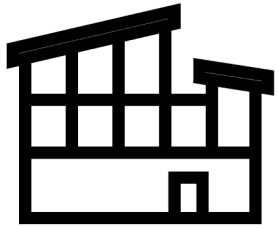
What are Zero-Knowledge Proofs (ZKPs) and why are they relevant for blockchains?

Hands-on



How to use zk-DSLs?

Applications



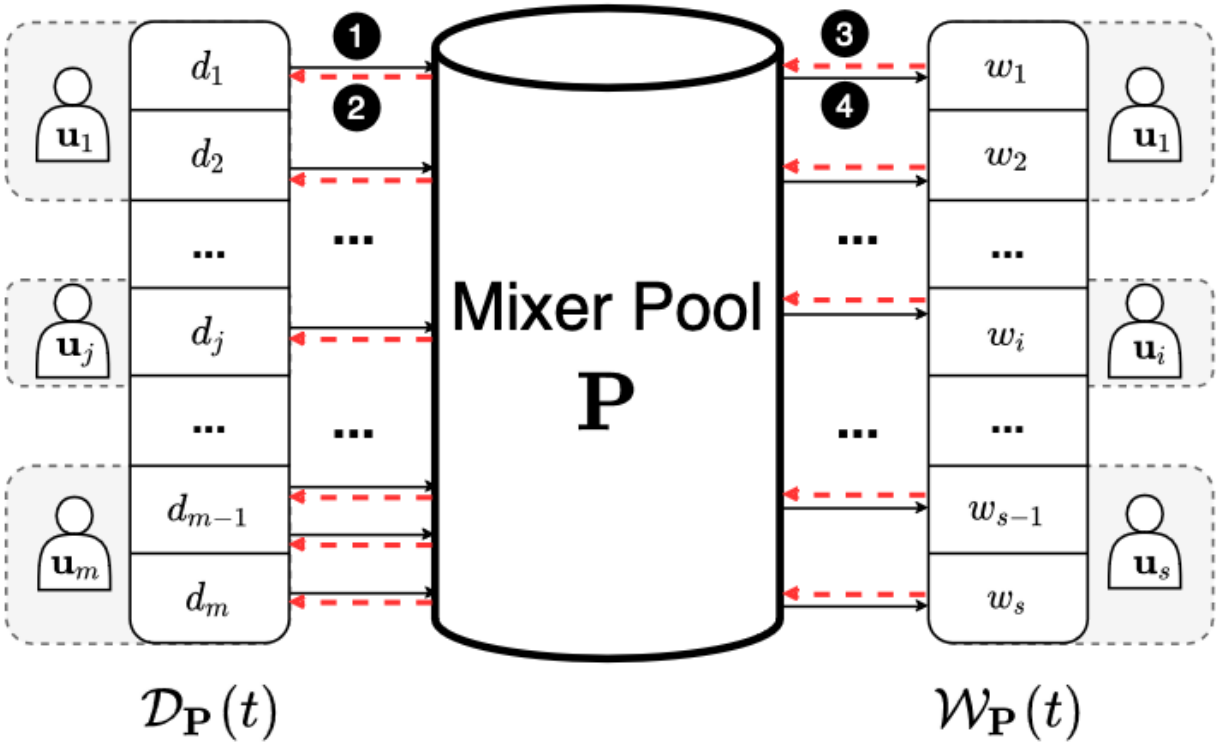
What can be built with ZKPs?

Application 1: Transaction Mixers

Mixers (Tornado Cash)

Achieve higher account privacy by pooling together funds from multiple users.

1. Deposit tokens in mixer pool
2. Receive deposit key
3. Prove that you are the holder of the deposit key with zk (from another address)
4. Receive withdraw



Application 2: ZK-Rollups

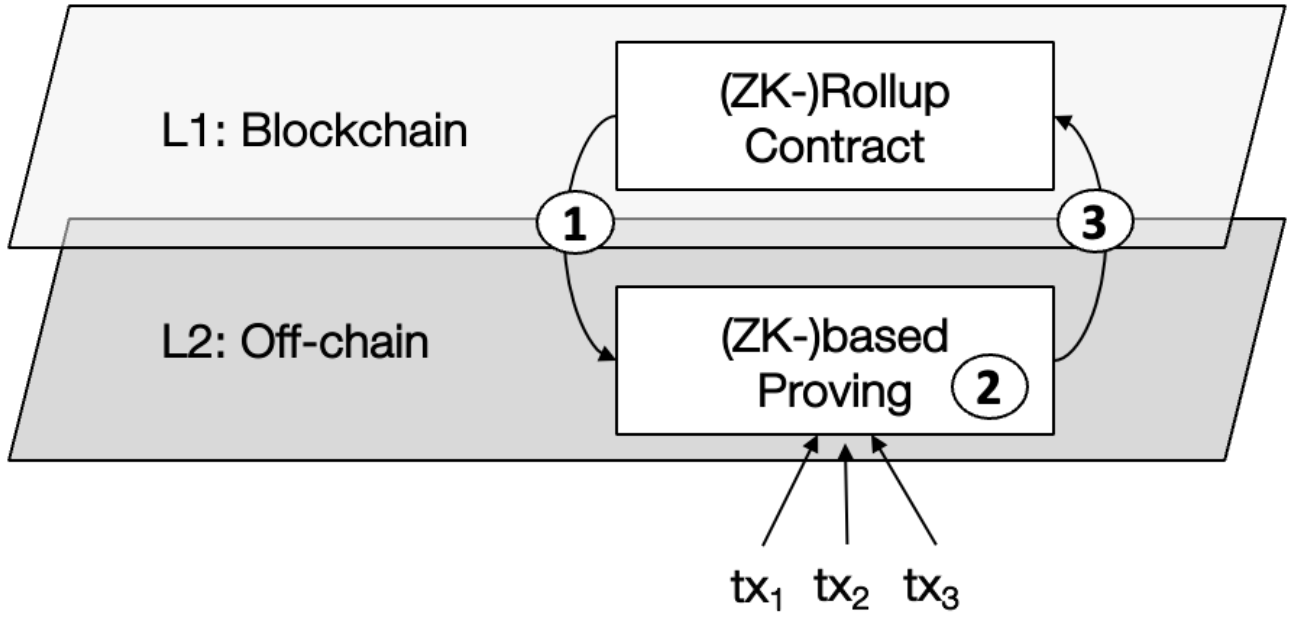
(ZK-)Rollups Overview

Objective: Reduce transaction costs through off-chain transaction aggregation

- Storage (compression),
- Computation (cheap verification)

Proof Correctness with

- Fraud Proofs (Mechanism Design), e.g., Arbitrum, Optimism
- Validity proofs (ZKP) , e.g., zkSync, Polygon, Starknet



1. Get state (accounts, balances) from L1.
2. Aggregate transactions and create proof of computational correctness on L2.
3. Verify proof and apply state to L1.

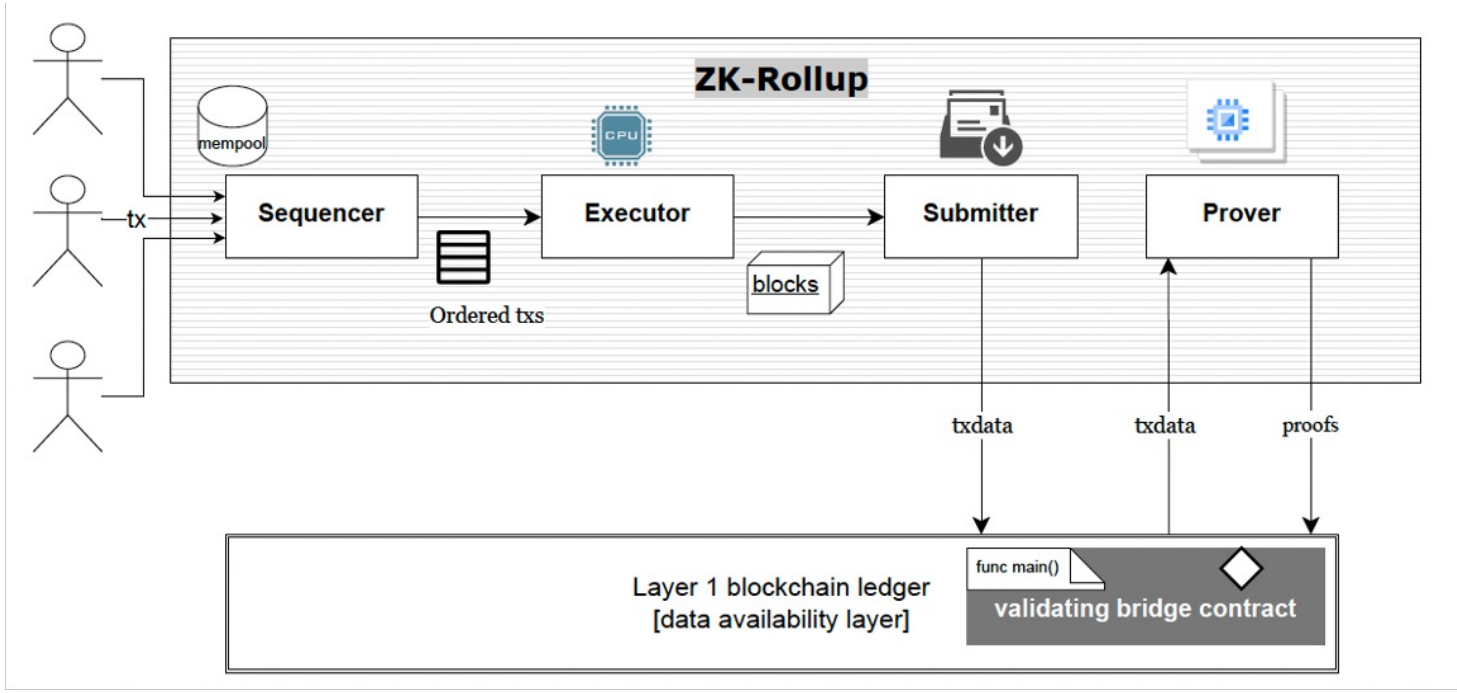
Example Architecture [1]

Sequencer: order transactions in preparation for execution.

Executors: construct transaction blocks from sequenced transactions.

Submitters: compute the hash of the Merkle tree's root and forward the state differences to L1.

Provers: actively monitor L1 checkpoint events, producing "validity proofs" with ZKPs.



[1] Motepalli, Shashank, Luciano Freitas, and Benjamin Livshits. "Sok: Decentralized sequencers for rollups." *arXiv preprint arXiv:2310.03616* (2023).

Example Architecture [1]

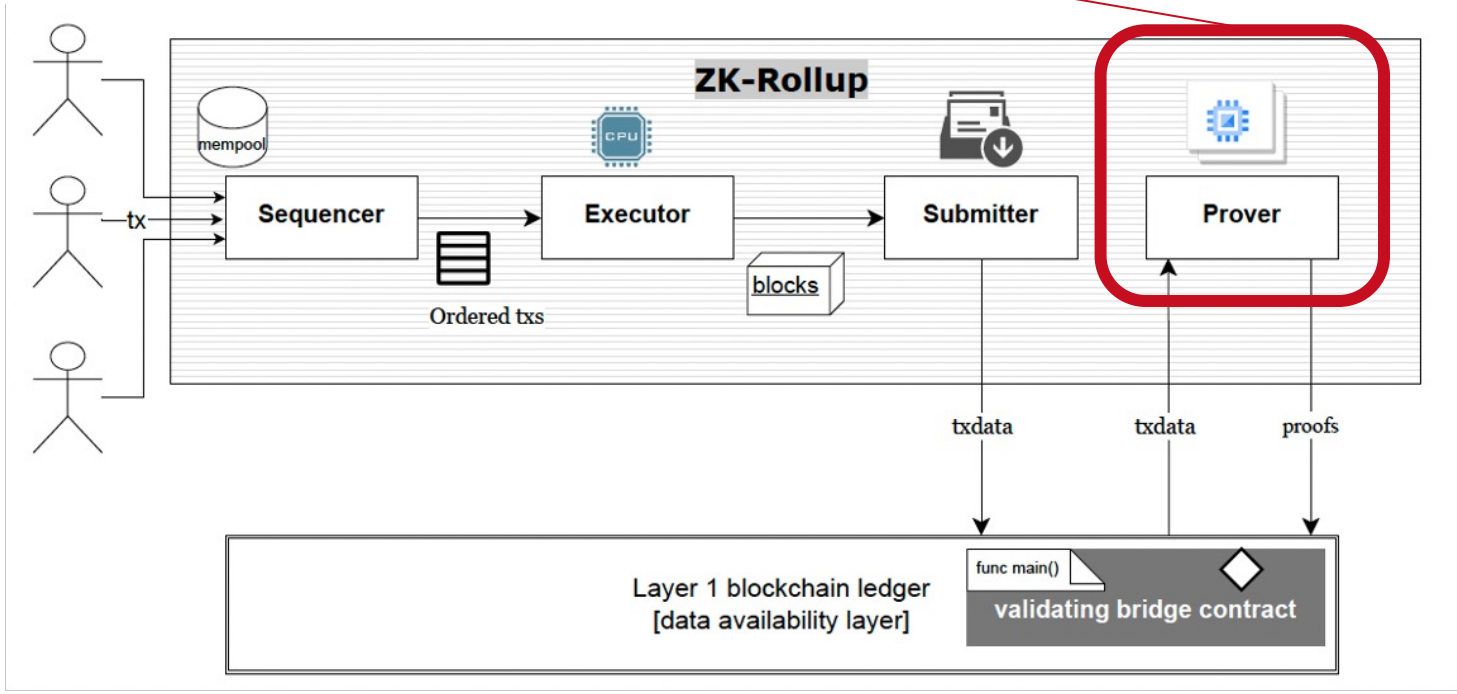
Sequencer: order transactions in preparation for execution.

Executors: construct transaction blocks from sequenced transactions.

Submitters: compute the hash of the Merkle tree's root and forward the state differences to L1.

Provers: actively monitor L1 checkpoint events, producing "validity proofs" with ZKPs.

- Proofs heavily rely on *Merkle Tree* operations.
- Optimization through *Recursive Proofs*, e.g., Nova, and *Efficient Hashing*, e.g., Poseidon.



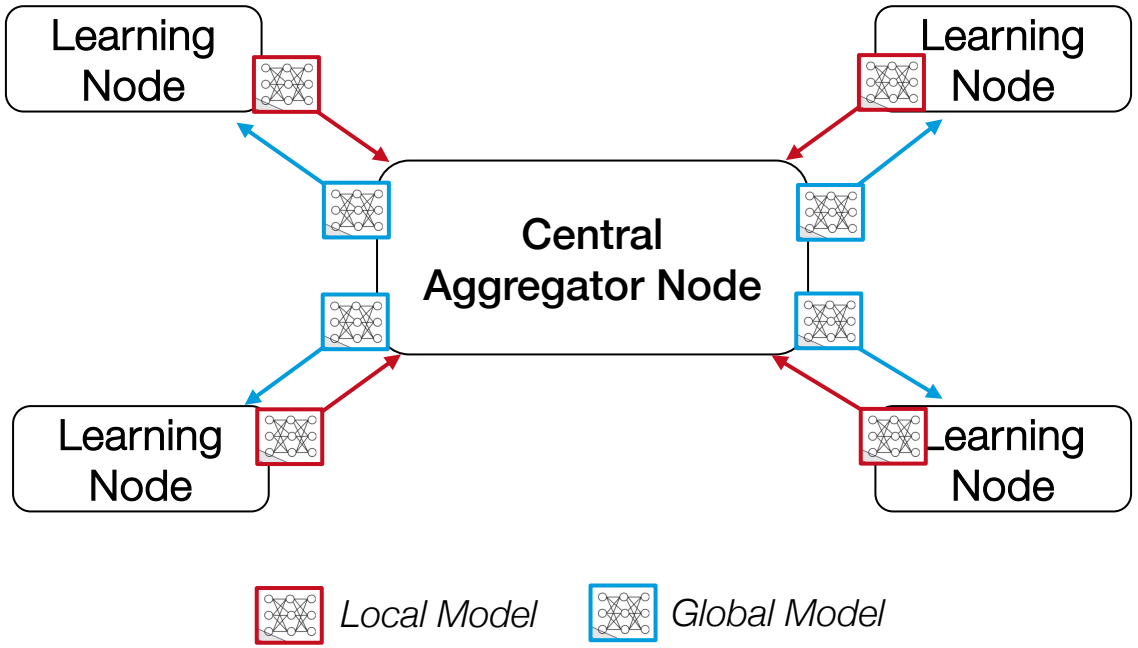
[1] Motepalli, Shashank, Luciano Freitas, and Benjamin Livshits. "Sok: Decentralized sequencers for rollups." *arXiv preprint arXiv:2310.03616* (2023).

Application 3: **Verifiable Decentralized Federated Learning**

Federated Learning

Learning nodes compute and submit *local model*.

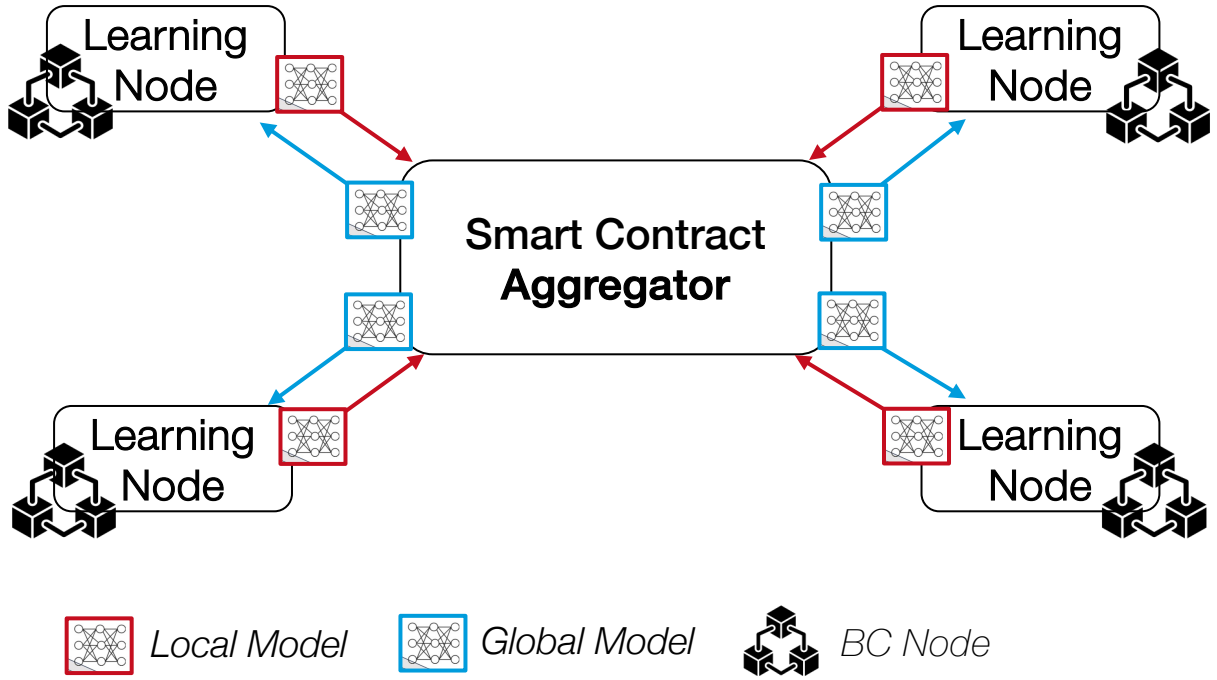
Aggregator node creates and returns *global model.*



Decentralized Federated Learning (+ Blockchain)

The aggregator is implemented as a smart contract.

Learning node submit local models through a blockchain client.

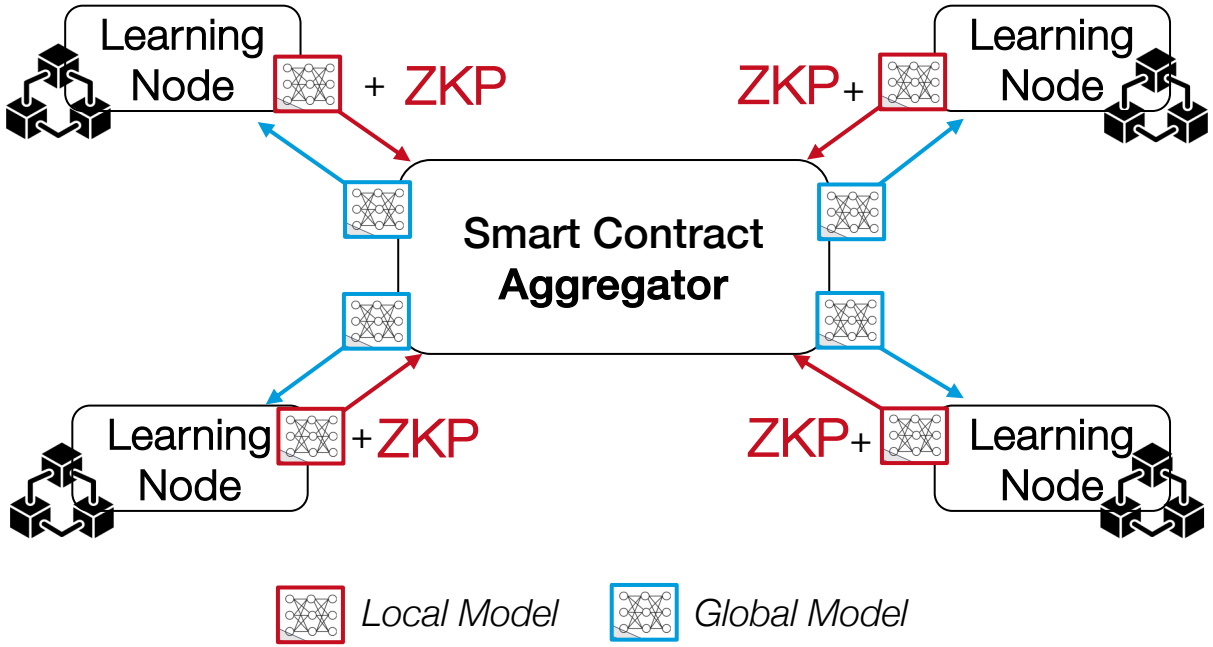


Verifiable Decentralized Federated Learning (+ ZKPs)

ZK-based local learning returns local model and computational proof.

Aggregator verifies ZKP before creating and returning global model.

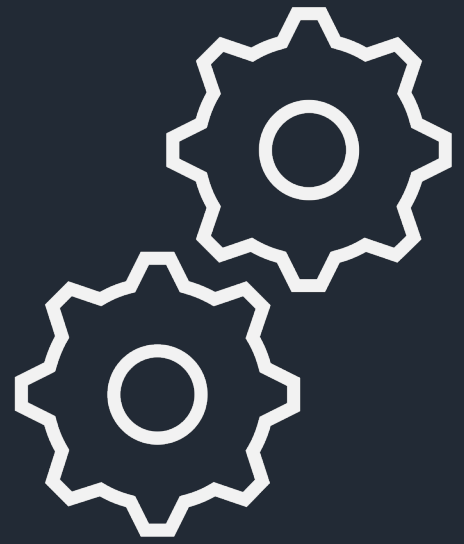
*...prevents **model poisoning** but adds high computational overhead.*



[1] J. Heiss, E. Grunewald, S. Tai, N. Haimerl, S. Schulte. *Advancing Blockchain-based Federated Learning through Verifiable Off-chain Computations*, IEEE Blockchain 2022

Other Application Contexts

- Netting in Energy Grids [3]
- Sensor Data Pre-processing in IoT scenarios [4]
- Blockchain Interoperability (zkBridges)
- Access Policy Evaluation
- ...



[3] J. Eberhardt, M. Peise., D.H. Kim, S. Tai. *Privacy-Preserving Netting in Local Energy Grids*, ICBC 2020
[4] J. Heiss, A. Busse, and S. Tai. *Trustworthy Pre-Processing of Sensor Data in Data On-chaining Workflows for Blockchain-based IoT Applications*, ICSOC 2021

Thanks for your attention!

We are looking forward to hear from you ☺

Please get in touch with us.

Mail: aa@ise.tu-berlin.de, johannes.sedlmeir@uni.lu, jh@ise.tu-berlin.de
Research Groups: https://www.tu.berlin/ise, https://www.uni.lu/snt-en/research-groups/finatrax/