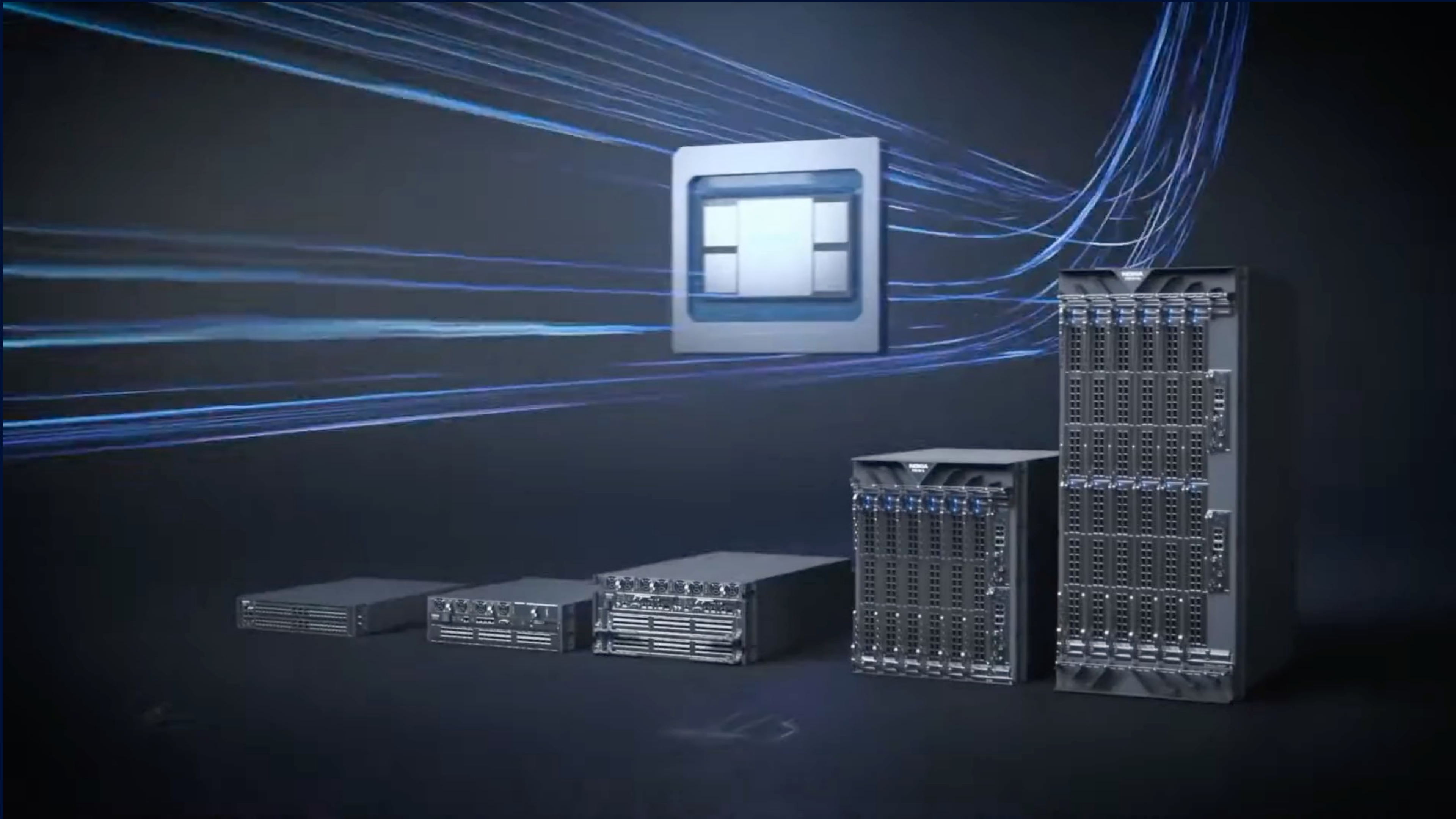
Zero-Knowledge Proofs for Verifiable Computation on Data Streams

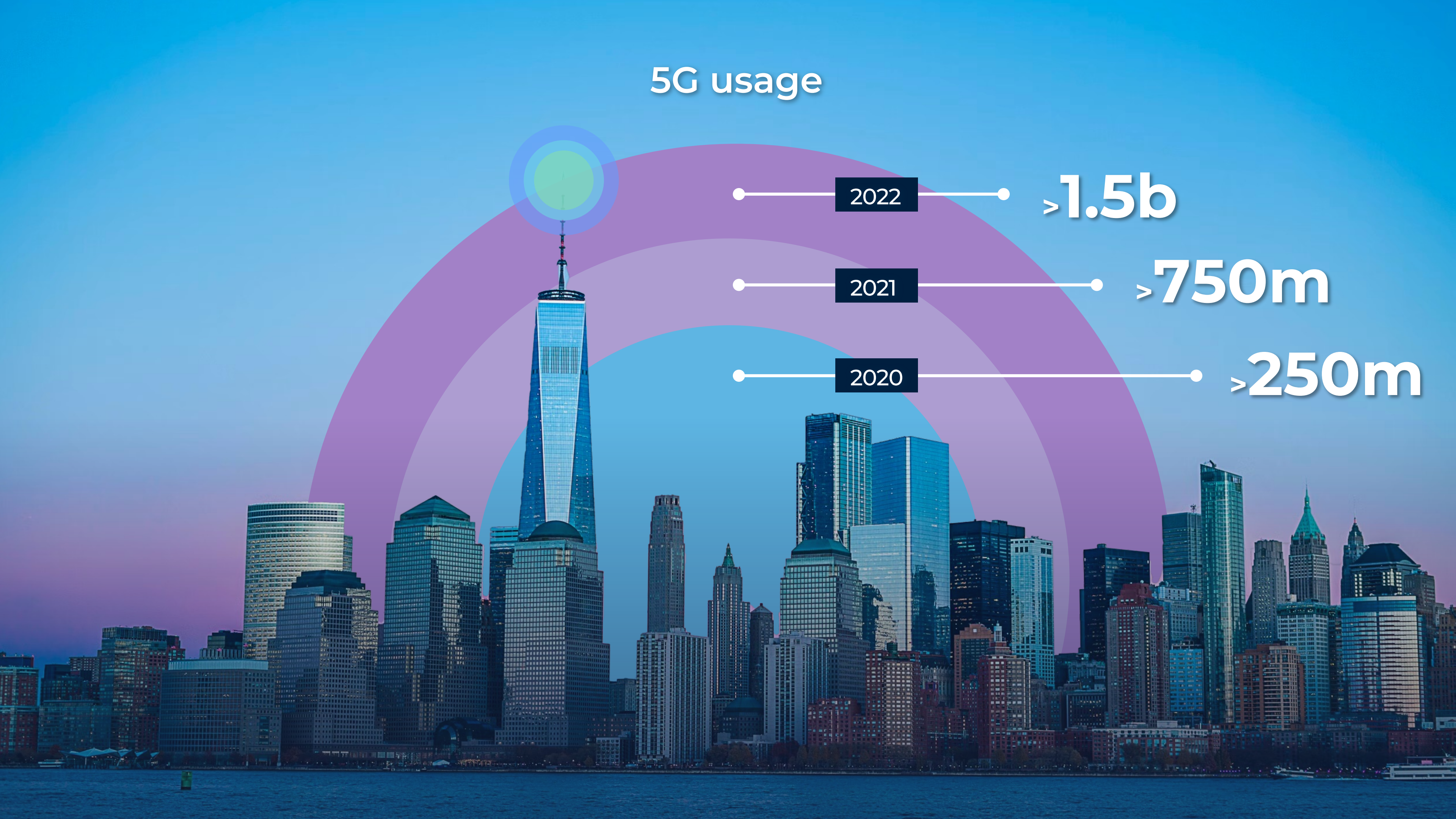
Lode Hoste
Janwillem Swalens

NOKIA BELL LABS

FOCUS > IDEAS

requires quality data





5G usage
2022
>1.5b
2021
>750m
2020
>250m

An unrivalled track record of innovation led by Nokia Bell Labs

9 Nobel Prizes | 5 Turing Awards | 3 Emmys | 2 Grammys | 1 Oscar

Foundations of …

- The entire electronics industry
- The internet, networking and optics
- Mobile and fixed communications


Transistors


Satellite comms

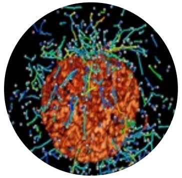

Laser/fiber optics


Unix/C/C++


Solar cells


Coherent optics


Charge-coupled devices


Super-resolution microscopy

Zero-Knowledge Proofs for Verifiable Computation on Data Streams

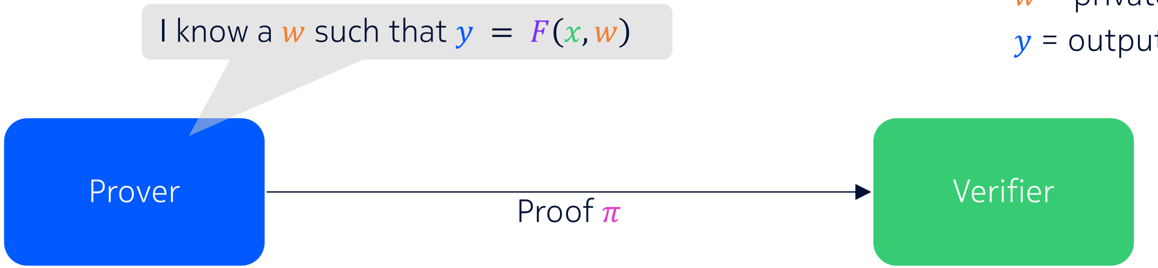
Lode Hoste
Janwillem Swalens

NOKIA BELL LABS

Zero-Knowledge Proofs

A prover can convince a verifier that a statement is true, without revealing anything besides the fact that the statement is true.

F = pre-agreed program
x = public input
w = private input
y = output



- **Completeness**: if the statement is true, an honest prover can convince an honest verifier of this fact.
- **Soundness**: if the statement is false, a cheating prover cannot convince an honest verifier that it is true (except with some small probability).
- **Zero-knowledge**: the verifier learns nothing other than the fact that the statement is true.

Goldwasser, Micali, and Rackoff (1985). "The knowledge complexity of interactive proof-systems." *Proceedings of the 17th Annual ACM Symposium on Theory of Computing*.

Example: the green and red ball and the colorblind friend

Note that:

- Not a mathematical proof, but a **probabilistic “proof”**.
  After n steps, the probability of soundness error is $1/2^n$.

  ⇒ “**argument of knowledge**”
- This example requires **interaction** between prover and verifier.
- I don’t give away which ball is which = **zero-knowledge**.

Since then…

1985: introduction of zero-knowledge proofs
Goldwasser, Micali, Rackoff (1985). "The knowledge complexity of interactive proof-systems."
Proceedings of the 17th Annual ACM Symposium on Theory of Computing.

1986: everything in NP has ZKP

1988: non-interactive ZKPs
Blum, Feldman, Micali (1988). "Non-Interactive Zero-Knowledge and Its Applications."
Proceedings of the 20th Annual ACM Symposium on Theory of Computing.

1992: succinctness
Kilian (1992). "A note on efficient zero-knowledge proofs and arguments."
Proceedings of the 24th Annual ACM Symposium on Theory of Computing.

1995: succinct & non-interactive
Micali (1995). "Computationally-Sound Proofs."
Logic Colloquium.

Non-interactive protocols do not require interaction between prover and verifier.

(Strong) **succinctness**:

- Proof is **short**: $|\pi| = O_\lambda(\log(|C|))$
  where $|C|$ = length of computation,
  λ = security parameter
- Proof is **fast to verify**:
  $\text{time}(V) = O_\lambda(|x|, \log(|C|))$
  here $|x|$ = size of input

Since then...

SNARK = succinct non-interactive argument of knowledge

1985: introduction of zero-knowledge proofs
Goldwasser, Micali, Rackoff (1985). “The knowledge complexity of interactive proof-systems.” *Proceedings of the 17th Annual ACM Symposium on Theory of Computing.*

1986: everything in NP has ZKP

1988: non-interactive ZKPs
Blum, Feldman, Micali (1988). “Non-Interactive Zero-Knowledge and Its Applications.” *Proceedings of the 20th Annual ACM Symposium on Theory of Computing.*

1992: succinctness
Kilian (1992). “A note on efficient zero-knowledge proofs and arguments.” *Proceedings of the 24th Annual ACM Symposium on Theory of Computing.*

1995: succinct & non-interactive
Micali (1995). “Computationally-Sound Proofs.” *Logic Colloquium.*

2010: without the PCP theorem
Groth (2010). “Short Pairing-Based Non-interactive Zero-Knowledge Arguments.” *Asiacrypt.*

2012: “SNARK”s exist for generic computations
Bitansky, Canetti, Chiesa, Tromer (2012). “From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again.” *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference.*

2013: quasi-linear proving time
Gennaro, Gentry, Parno, Raykova (2013). “Quadratic Span Programs and Succinct NIZKs without PCPs.” *Eurocrypt 2013.*

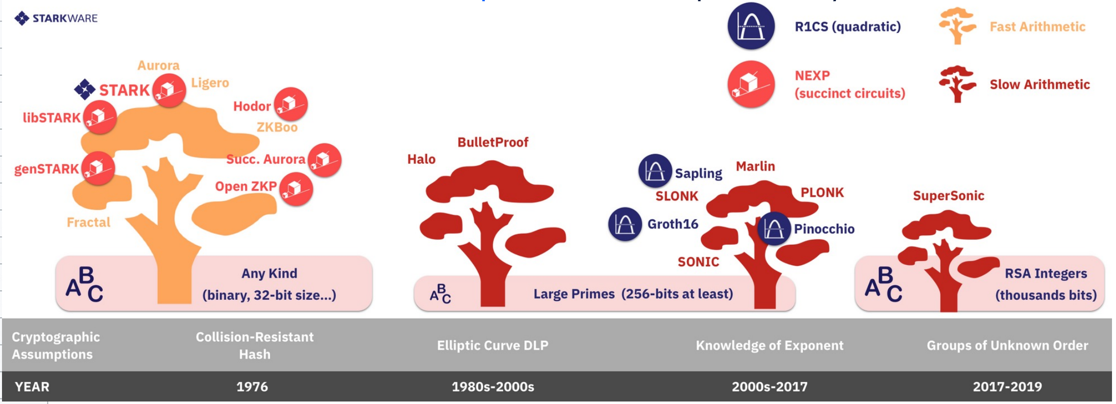
2016: launch of Zcash

…to now

Zero-knowledge proof (ZKP) systems

ZKP System	Publication year	Protocol	Transparent	Universal	Plausibly Post-Quantum Secure	Programming Paradigm
Pinocchio[32]	2013	zk-SNARK	No	No	No	Procedural
Geppetto[33]	2015	zk-SNARK	No	No	No	Procedural
TinyRAM[34]	2013	zk-SNARK	No	No	No	Procedural
Buffet[35]	2015	zk-SNARK	No	No	No	Procedural
ZoKrates[36]	2018	zk-SNARK	No	No	No	Procedural
xJsnark[37]	2018	zk-SNARK	No	No	No	Procedural
vRAM[38]	2018	zk-SNARG	No	Yes	No	Assembly
vnTinyRAM[39]	2014	zk-SNARK	No	Yes	No	Procedural
MIRAGE[40]	2020	zk-SNARK	No	Yes	No	Arithmetic Circuits
Sonic[41]	2019	zk-SNARK	No	Yes	No	Arithmetic Circuits
Marlin[42]	2020	zk-SNARK	No	Yes	No	Arithmetic Circuits
PLONK[43]	2019	zk-SNARK	No	Yes	No	Arithmetic Circuits
SuperSonic[44]	2020	zk-SNARK	Yes	Yes	No	Arithmetic Circuits
Bulletproofs[45]	2018	Bulletproofs	Yes	Yes	No	Arithmetic Circuits
Hyrax[46]	2018	zk-SNARK	Yes	Yes	No	Arithmetic Circuits
Halo[47]	2019	zk-SNARK	Yes	Yes	No	Arithmetic Circuits
Virgo[48]	2020	zk-SNARK	Yes	Yes	Yes	Arithmetic Circuits
Ligero[49]	2017	zk-SNARK	Yes	Yes	Yes	Arithmetic Circuits
Aurora[50]	2019	zk-SNARK	Yes	Yes	Yes	Arithmetic Circuits
zk-STARK[51]	2019	zk-STARK	Yes	Yes	Yes	Assembly
Zilch[31][52]	2021	zk-STARK	Yes	Yes	Yes	Object-Oriented

“Cambrian explosion” of proof systems



tornado Aleo MINA ESPRESSO SYSTEMS

Privacy on blockchains

StarkNet zkSync Aztec Polygon Hermez

Scaling blockchains

Self-Sovereign Identity

Anonymous Credentials

Smart contracts

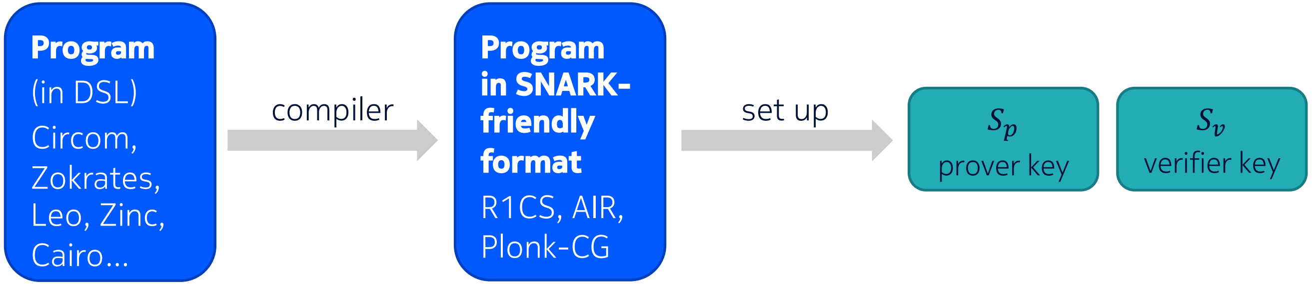
Zero-Knowledge Proofs as a fundamental building block for Web3

NOKIA BELL LABS

Introduction to zk-SNARKs

System overview

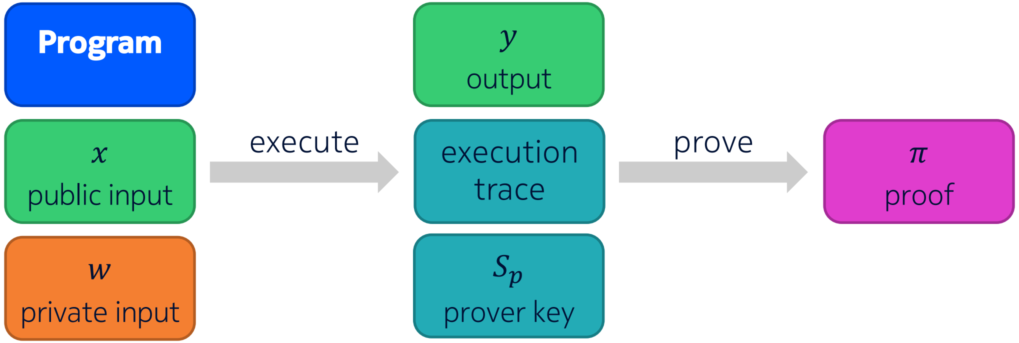
Set-up



Note: some variations depending on proof system

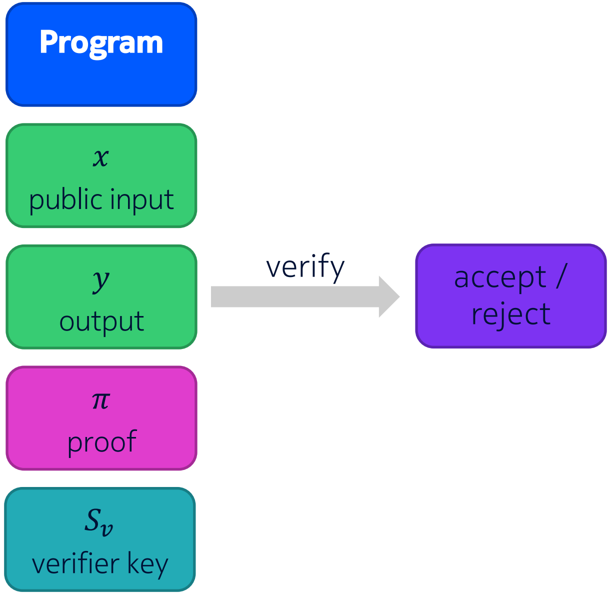
System overview

Prover

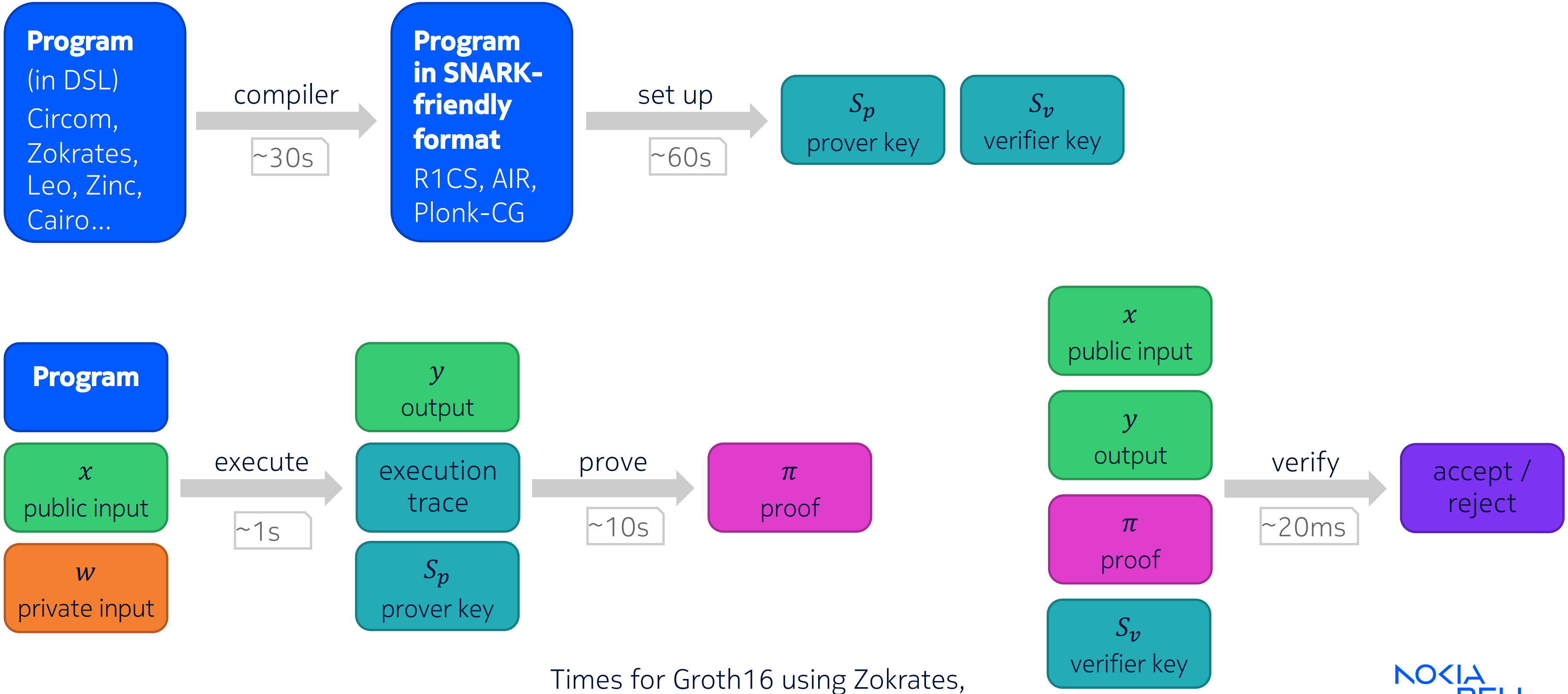


System overview

Verification



System overview



Times for Groth16 using Zokrates, other systems make other trade-offs.

Running example

Compute and prove correct execution of:

```python
def func(w1, w2, w3):
    return w1 * w2 * w3
```

$f\ :\ \mathbb{F}_{11}\times\mathbb{F}_{11}\times\mathbb{F}_{11} \rightarrow \mathbb{F}_{11}$

$f\ : (w_1, w_2, w_3) \rightarrow (w_1 * w_2) * w_3$

All operations are on integers in a field $\mathbb{F}_p$, with p a prime number.

All operations are using modulo arithmetic.

For the example, $p = 11$.

In Circom, $p = 21888242871839275222246405745257275088548364400416034343698204186575808495617$ (a prime slightly smaller than 2^{256}).

This system only supports integers and modulo arithmetic!

⚠️ Watch out for overflows!

Running example

Flattening to constraints

```
def func(w1, w2, w3):
    return w1 * w2 * w3
```

Flatten

```
def func(w1, w2, w3):
    w4 = w1 * w2
    w5 = w4 * w3
    return w5
```

SHA256 ≈ 40K constraints

ECDSA sig. verification ≈ 90K

The compiler flattens the program to a list of constraints,
e.g. Rank-1 Constraint System (R1CS).

Note: different compilers can give very different representations,
so opportunity for compiler optimization.

E.g. in this example, you could do the multiplications in the opposite order.

Running example

Convert to arithmetic circuit

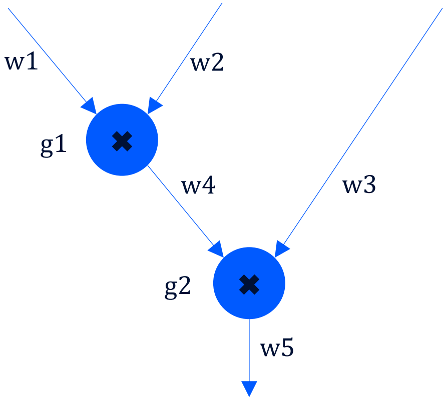
```python
def func(w1, w2, w3):
    return w1 * w2 * w3
```

Flatten

```python
def func(w1, w2, w3):
    w4 = w1 * w2
    w5 = w4 * w3
    return w5
```

SHA256 ≈ 40K constraints
ECDSA sig. verification ≈ 90K

Represented as an **arithmetic circuit** (DAG)



$g1, g2$ = **gates** (multiplication & addition in $\mathbb{F}_{11}$)
$w1, \ldots, w5$ = **wire** labels or wire values

Running example

An execution is an assignment

```
def func(w1, w2, w3):
    return w1 * w2 * w3
```

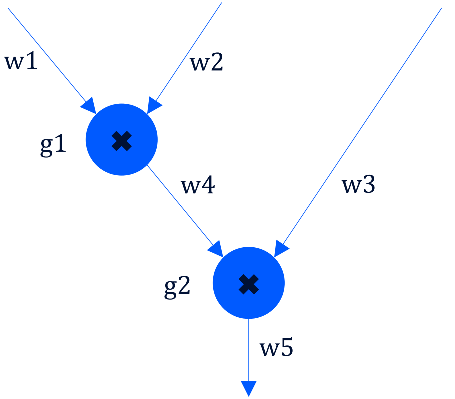
Flatten

```
def func(w1, w2, w3):
    w4 = w1 * w2
    w5 = w4 * w3
    return w5
```

SHA256 ≈ 40K constraints

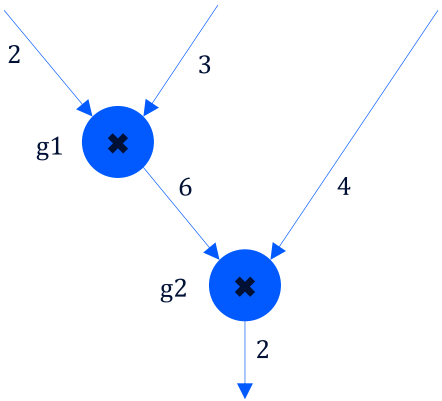
ECDSA sig. verification ≈ 90K

Represented as an **arithmetic circuit** (DAG)



$g1, g2$ = **gates** (multiplication & addition)

$w1, \ldots, w5$ = **wire** labels or wire values

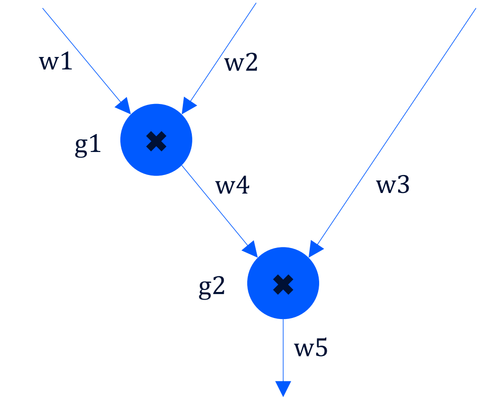


Assignment (witness & public inputs)

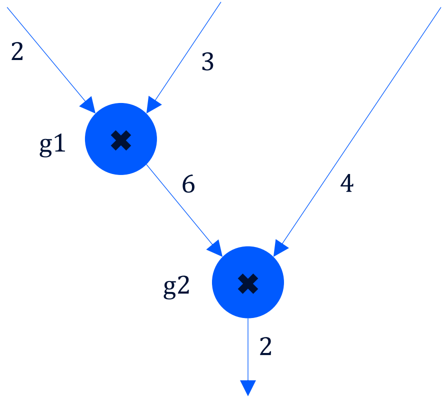
$$W = \{w1, w2, w3, w4, w5\} = \{2,3,4,6,2\}$$

(All computations performed in $\mathbb{F}_{11}$, i.e., mod 11)

Valid assignments



g1, g2 = **gates** (multiplication & addition)
w1, … , w5 = **wire** labels or wire values



Assignment (witness & public inputs)

$W = \{w1, w2, w3, w4, w5\} = \{2,3,4,6,2\}$

(All computations performed in $\mathbb{F}_{11}$, i.e., mod 11)

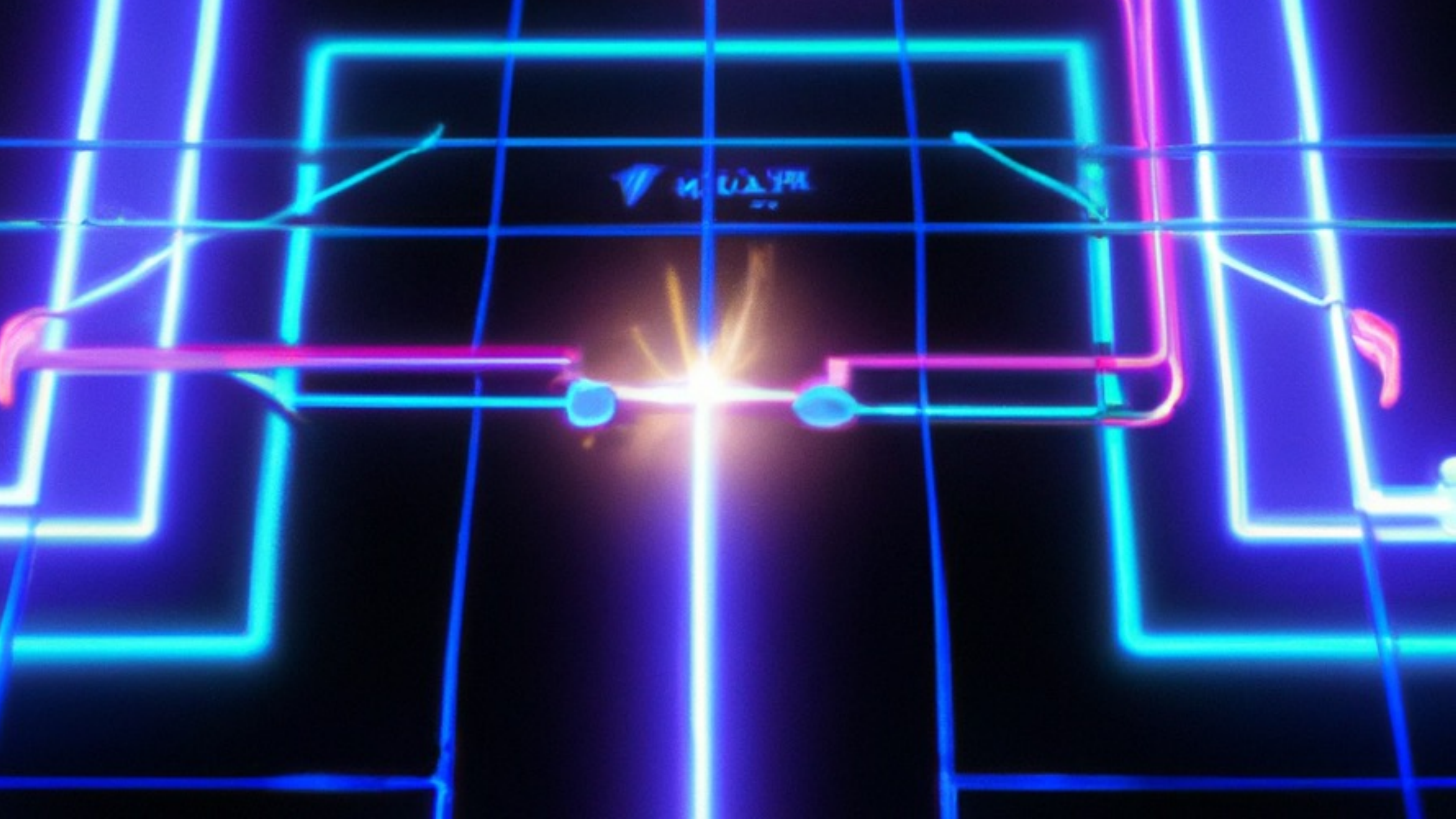
An assignment is **valid** if it was produced by actual circuit execution, i.e. satisfies the constraints imposed by the gates

⇒ a valid assignment is a proof of correct circuit execution.

But it is not succinct nor fast to verify.

Goal: create a **verifiable computation protocol**: a protocol to succinctly transfer an assignment to a verifier & allow it to verify the validity succinctly.

Demo



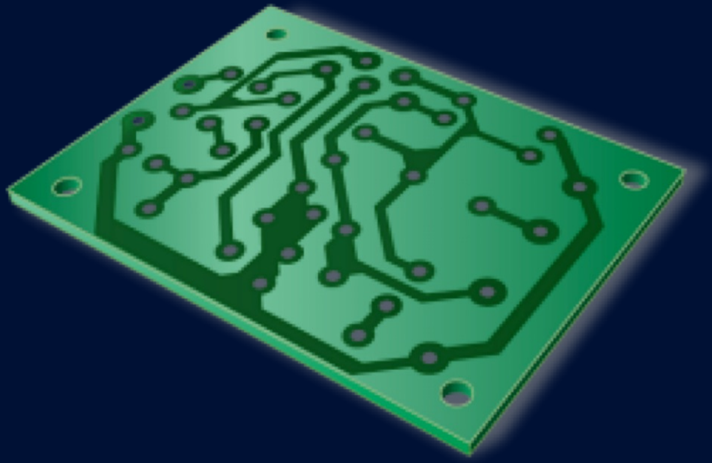

GATE

Circuit Design

```
template MultiAND(n) {
    signal input in[n];
    signal output out;

    var sum = 0;
    for (var i=0; i<n; i++) {
      sum = sum + in[i];
    }

    component isz = IsZero();
    sum - n --> isz.in;
    isz.in === sum - n;

    isz.out --> out;
    out === isz.out;
}

component main = MultiAND(1000);
```

Introducing Circom 2.0 by Iden3 https://www.youtube.com/watch?v=6XxVeBFmIFs

Execution



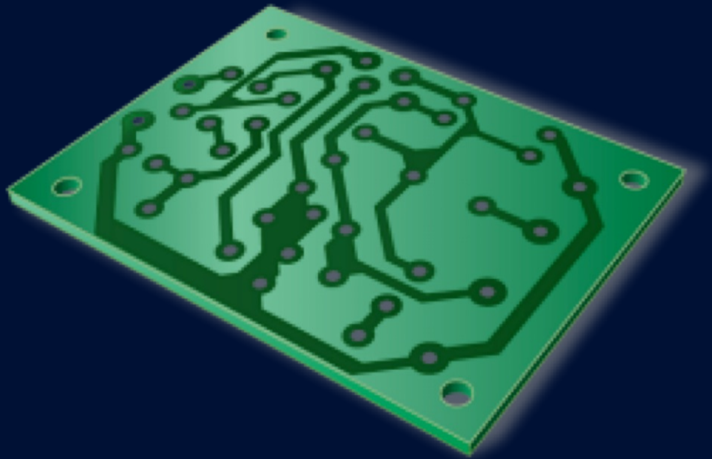
https://github.com/lhoste-bell/snarkjs_multiand

Circuit Design

```
template MultiAND(n) {
    signal input in[n];
    signal output out;

    var sum = 0;
    for (var i=0; i<n; i++) {
      sum = sum + in[i];
    }

    component isz = IsZero();
    sum - n --> isz.in;
    isz.in === sum - n;

    isz.out --> out;
    out === isz.out;
}

component main = MultiAND(1000);
```

Introducing Circom 2.0 by Iden3 https://www.youtube.com/watch?v=6XxVeBFmIFs

Execution



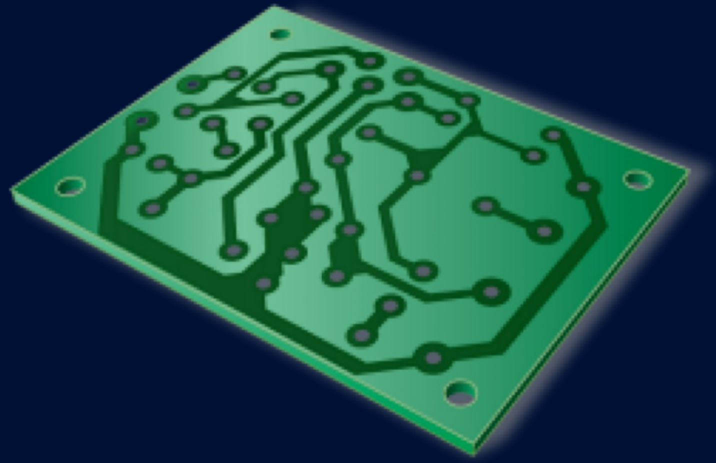
https://github.com/lhoste-bell/snarkjs_multiand

Circuit Design

```
template MultiAND(n) {
    signal input in[n];
    signal output out;

    var sum = 0;
    for (var i=0; i<n; i++) {
      sum = sum + in[i];
    }

    component isz = IsZero();
    sum - n --> isz.in;
    isz.in === sum - n;

    isz.out --> out;
    out === isz.out;
}

component main = MultiAND(1000);
```

Introducing Circom 2.0 by Iden3 https://www.youtube.com/watch?v=6XxVeBFmlFs

Execution



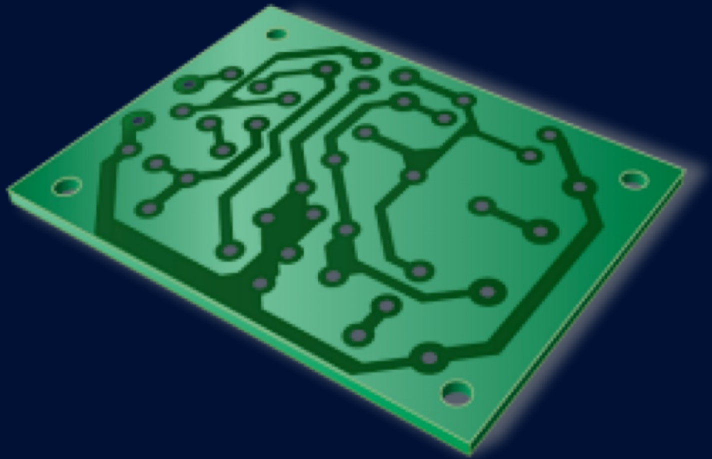
https://github.com/lhoste-bell/snarkjs_multiand

Circuit Design

```
template MultiAND(n) {
    signal input in[n];
    signal output out;

    var sum = 0;
    for (var i=0; i<n; i++) {
      sum = sum + in[i];
    }

    component isz = IsZero();
    sum - n --> isz.in;
    isz.in === sum - n;

    isz.out --> out;
    out === isz.out;
}

component main = MultiAND(1000);
```

Introducing Circom 2.0 by Iden3 https://www.youtube.com/watch?v=6XxVeBFmlFs

Execution



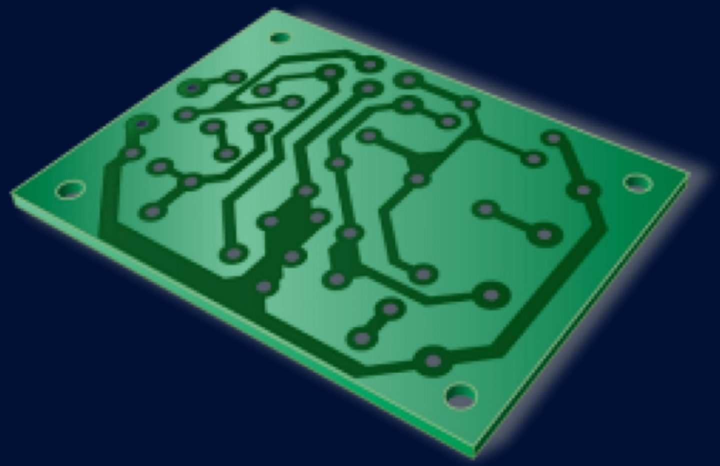
https://github.com/lhoste-bell/snarkjs_multiand

Circuit Design

```
template MultiAND(n) {
    signal input in[n];
    signal output out;

    var sum = 0;
    for (var i=0; i<n; i++) {
      sum = sum + in[i];
    }

    component isz = IsZero();
    sum - n --> isz.in;
    isz.in === sum - n;

    isz.out --> out;
    out === isz.out;
}

component main = MultiAND(1000);
```

Introducing Circom 2.0 by Iden3 https://www.youtube.com/watch?v=6XxVeBFmIFs

Execution



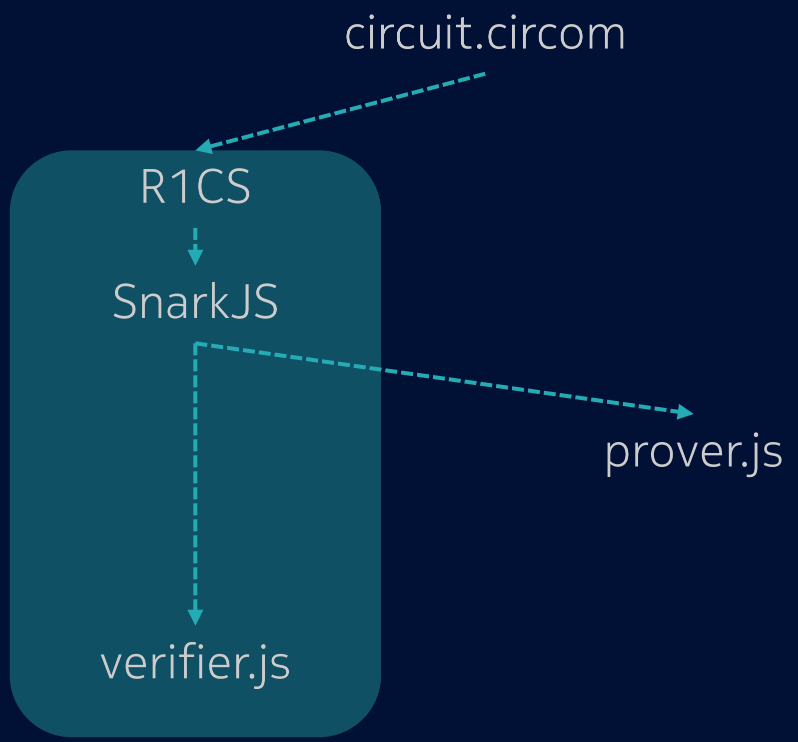
https://github.com/lhoste-bell/snarkjs_multiand

Circuit Design

```
template MultiAND(n) {
    signal input in[n];
    signal output out;

    var sum = 0;
    for (var i=0; i<n; i++) {
        sum = sum + in[i];
    }

    component isz = IsZero();
    sum - n --> isz.in;
    isz.in === sum - n;

    isz.out --> out;
    out === isz.out;
}

component main = MultiAND(1000);
```

Introducing Circom 2.0 by Iden3 https://www.youtube.com/watch?v=6XxVeBFmlFs

Steps



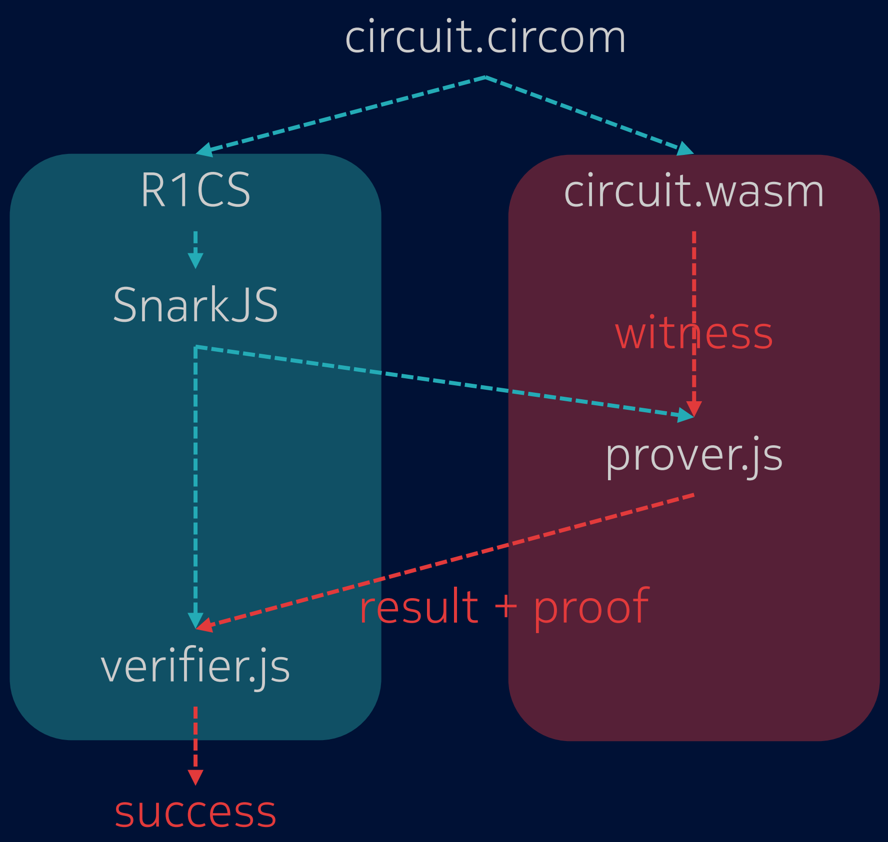
https://github.com/lhoste-bell/snarkjs_multiand

Circuit Design

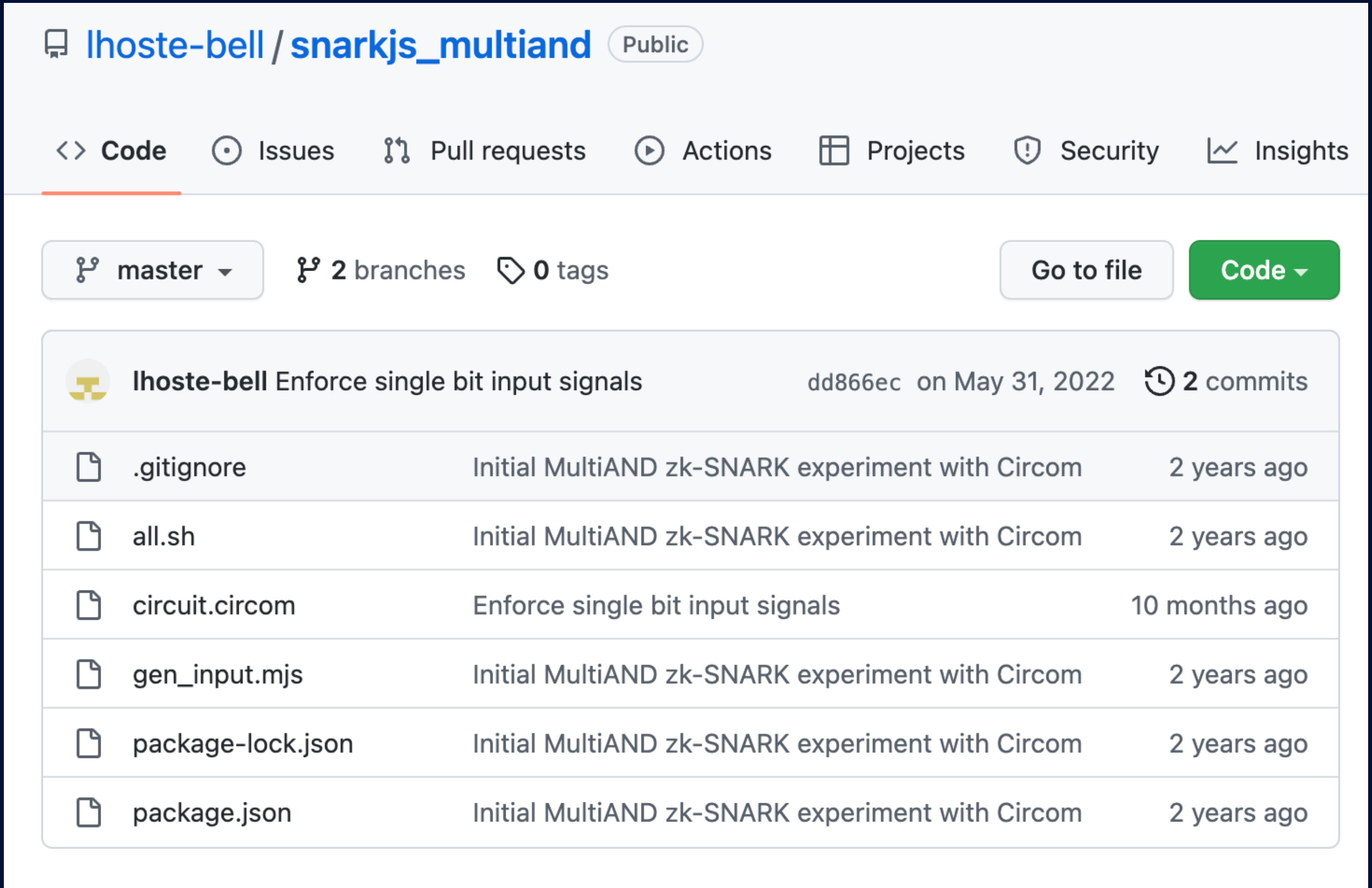
```
template MultiAND(n) {
    signal input in[n];
    signal output out;

    var sum = 0;
    for (var i=0; i<n; i++) {
      sum = sum + in[i];
    }

    component isz = IsZero();
    sum - n --> isz.in;
    isz.in === sum - n;

    isz.out --> out;
    out === isz.out;
}

component main = MultiAND(1000);
```

Introducing Circom 2.0 by Iden3 https://www.youtube.com/watch?v=6XxVeBFmlFs

Steps



https://github.com/lhoste-bell/snarkjs_multiand



https://github.com/lhoste-bell/snarkjs_multiand

The mathematics behind zk-SNARKs

NOKIA BELL LABS

Goal

- To give you some intuition of the math behind ZKPs
- Using a simple end-to-end example
- But there are many different systems out there and they're constantly evolving…

Following slides: Pinocchio / Groth16, one of many systems

- Computation
- Algebraic Circuit
- R1CS
- QAP
- Linear PCP
- Linear Interactive Proof
- zkSNARK

Parno, Howell, Gentry, Raykova (2013). "Pinocchio: nearly practical verifiable computation". *Proceedings of the 2013 IEEE Symposium on Security and Privacy.*

Groth (2016). "On the size of pairing-based non-interactive arguments". *Eurocrypt 2016.*

Trick 1: Succinctly proving knowledge of a polynomial

How to prove something succinctly?

Simple case: Verifier has a polynomial $P(x)$ of degree d. Prover claims to know $P(x)$, i.e., knows the coefficients:

- Verifier sends a random value $\boldsymbol{s}$ and asks the prover to return $P(\boldsymbol{s})$
- Verifier computes $P(\boldsymbol{s})$ on his own and compares results

This trick allows us to create a succinct proof:
evaluating at a single point is sufficient to reveal the identity of the polynomial

NOKIA BELL LABS

Schwarz-Zippel lemma

Take $P(x)$ and $Q(x)$ polynomials of degree d.

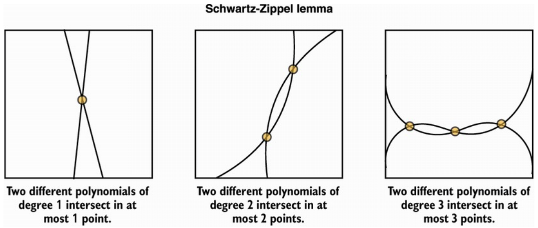
If $P(x) = Q(x)$:

$$P(x) - Q(x) = 0 \quad \forall x$$

If you take a random x,
$P(x) - Q(x)$ will always be 0.

If $P(x) \neq Q(x)$:

$P(x) - Q(x) = 0$ for at most d values of x



If you take a random x,
$P(x) - Q(x)$ is extremely likely to not be 0.

In our case, $d \approx 10^7$ (number of constraints), range of $x \approx 2^{256} \approx 10^{78}$ (field size)

$\Rightarrow$ Prob(randomly chosen point x is one of the d common points) $= \frac{10^7}{10^{78}} \approx 0$

Trick: evaluating P and Q at a random point x will tell us with high probability whether they're equal.

Trick 2: Blind evaluation of a polynomial

How to hide the actual values from the verifier?

Verifier sends encrypted powers of $\boldsymbol{s}$ (e.g. $E[\boldsymbol{s}^2]$, $E[\boldsymbol{s}^1]$, $E[\boldsymbol{s}^0]$) to the prover (instead of $\boldsymbol{s}$)

Suppose: $E[x] = g^x \; mod \; n,\quad E[x] * E[y] = E[x+y],\quad E[x]^y = E[x * y]$

n = large prime, g = generator of a group with a hard to compute discrete log, e.g., elliptic curves

Prover computes $E[P(\boldsymbol{s})]$:

$$E[P(\boldsymbol{s})] = E[w_2\boldsymbol{s}^2 + w_1\boldsymbol{s}^1 + w_0\boldsymbol{s}^0] = E[w_2\boldsymbol{s}^2] * E[w_1\boldsymbol{s}^1] * E[w_0\boldsymbol{s}^0] = \prod_{k=0}^{2} E[w_k\boldsymbol{s}^k] = \prod_{k=0}^{2} \boxed{E[\boldsymbol{s}^k]}^{w_k}$$

e.g. $P(x) = w_2x^2 + w_1x^1 + w_0x^0$

→ with (encrypted) values from verifier, this can be computed by prover

⇒ the verifier does not need to send $\boldsymbol{s}$, everything can happen on encrypted values

NOKIA BELL LABS

Proving correct program execution using polynomials

We encode "proving correct program execution" as "proving knowledge of a (specifically crafted) polynomial"

We encode the program (= constraints imposed by gates on wires) into a set of polynomials $\{p(x)\}$ = **Quadratic Arithmetic Program** (QAP)

One polynomial per input & output

and a **target polynomial** $T(x) = (x - g_1)(x - g_2) \dots (x - g_d)$ where g_k = random int (chosen by verifier), d = number of gates

The prover evaluates the program and generates an **assignment** $W = \{w_1, w_2, w_3, w_4, w_5\}$.

Using the assignment and QAP, the prover derives a single polynomial, $P(x) = \sum_{k \in W} w_k p_k(x)$.

We will create the QAP such that:

If and only if $P(x)$ is derived from a **valid** assignment, then $P(x)$ is expected to be 0 for $x \in \{g_1, g_2, \dots, g_d\}$,

$\Rightarrow P(x)$ will be divisible by $T(x)$ $\Rightarrow$ $P(x) = T(x)H(x)$ (where $H(x) = \frac{P(x)}{T(x)}$)

Hence, a proof of correct execution consists of convincing a verifier that the prover knows $P(x)$ and $H(x)$ that satisfy these equations.

The equations can be verified succinctly (at a single point, trick 1) and without sharing the assignment ($P(x)$ is not shared, hence zero-knowledge).

Verifiable Computation Protocol – High Level

- Trusted setup (once per circuit):
  - Encode program into **polynomials**: $T(x),\ \{p(x)\}$
  - Generate random point $\boldsymbol{s}$ and compute $T(\boldsymbol{s})$ [1]
- Prover
  - **Evaluate program** and generate an **assignment**, $W = \{w_1, w_2, w_3, w_4, w_5\}$
  - Using assignment to derive $P(x) = \sum_{k \in W} w_k p_k(x)$. Then compute $H(x) = \frac{P(x)}{T(x)}$
  - Generate **proof of computation**: $P(\boldsymbol{s}), H(\boldsymbol{s})$ [2]
- Verifier
  - To verify the proof, check: $P(\boldsymbol{s}) = T(\boldsymbol{s}) * H(\boldsymbol{s})$ [3] [4]

[1] Homomorphically encrypted powers of $\boldsymbol{s}$ ($E[s^n], E[s^{n-1}], \dots, E[s^1], E[s^0]$) and $E[T(\boldsymbol{s})]$ are generated, and then $\boldsymbol{s}$ is destroyed.

[2] Prover returns $E[P(\boldsymbol{s})], E[H(\boldsymbol{s})]$, since it only has access to encrypted powers of $\boldsymbol{s}$

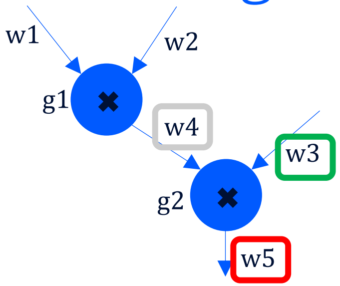
[3] This check is performed in encrypted domain using cryptographic pairing friendly Elliptic curves.

[4] A check that forces the prover to only use the encrypted power of $\boldsymbol{s}$ is also performed. This requires additional randomness from trusted setup.

Succinct proof of execution & quick verification is possible with specially constructed polynomials $T(x)$ & $\{p(x)\}$ such that when $P(x)$ is derived from valid assignments, then $P(x) = T(x)H(x)$

Encoding Program Structure Into Polynomials

Aim: create special $T(x)$, $\{p(x)\}$ encoding 'program structure' such that $P(x) = T(x)H(x)$ for valid assignments



Program: Arithmetic Circuit

↓

Constraints: Rank 1 Constraint System (R1CS)
Encode circuit structure as constraints

For each gate produce 3 vectors, l, r, o, that encode if a particular wire is a left input, right input, or an output of a gate (length of each vector = number of wires)

Associate **left inputs** of gates to wires

$$\begin{matrix} g1 \\ g2 \end{matrix} \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$
$$\quad w1 \quad w2 \quad w3 \quad w4 \quad w5$$

Associate **right inputs** of gates to wires

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Associate **outputs** of gates to wires

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

l, r, o satisfy the constraint: $l \circ W \quad * \quad r \circ W \quad - \quad o \circ W = 0$

W = assignment = $\{w_1, w_2, w_3, w_4, w_5\}$, $\circ$ = dot product

$$\begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \circ \begin{bmatrix} w1 \\ w2 \\ w3 \\ w4 \\ w5 \end{bmatrix} * \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \circ \begin{bmatrix} w1 \\ w2 \\ w3 \\ w4 \\ w5 \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \circ \begin{bmatrix} w1 \\ w2 \\ w3 \\ w4 \\ w5 \end{bmatrix} = 0 \quad \equiv \quad w3 * w4 - w5 = 0$$

If these constraints are satisfied for all gates ⇔ assignment is valid ⇔ program executed correctly

By encoding constraints as polynomials, large number of constraints can be checked all at once

Encoding Program Structure Into Polynomials

Aim: create special $T(x)$, $\{p(x)\}$ encoding 'program structure' such that $P(x) = T(x)H(x)$ for valid assignments

Polynomials: Quadratic Arithmetic Program (QAP) – Encode constraints as polynomials

- Assign arbitrary distinct integers to gates, e.g., $g1 = 5, g2 = 7$
- For each wire W_k, construct 3 polynomials, $L_{w_k}(x)$, $R_{w_k}(x)$, $O_{w_k}(x)$ such that

$$\begin{array}{c|ccccc} & w1 & w2 & w3 & w4 & w5 \\ \hline g1 = 5 & 1 & 0 & 0 & 0 & 0 \\ g2 = 7 & 0 & 0 & 0 & 1 & 0 \end{array}$$

Matrix encoding wires acting as left inputs

$$\left.\begin{array}{l} L_{w1}(x = g1) = 1 \\ L_{w1}(x = g2) = 0 \end{array}\right\} \Rightarrow \boldsymbol{L_{w1}(x) = 5x + 9}$$

(Single polynomial encodes constraints from all gates on w1 acting as left input)

$$\left.\begin{array}{l} L_{w4}(x = g1) = 0 \\ L_{w4}(x = g2) = 1 \end{array}\right\} \Rightarrow \boldsymbol{L_{w4}(x) = 6x + 3}$$

$\{p(x)\} =$

$L_{w1}(x)$: 5x+9	$L_{w2}(x)$: 0	$L_{w3}(x)$: 0	$L_{w4}(x)$: 6x+3	$L_{w5}(x)$: 0
$R_{w1}(x)$: 0	$R_{w2}(x)$: 5x+9	$R_{w3}(x)$: 6x+3	$R_{w4}(x)$: 0	$R_{w5}(x)$: 0
$O_{w1}(x)$: 0	$O_{w2}(x)$: 0	$O_{w3}(x)$: 0	$O_{w4}(x)$: 5x+9	$O_{w5}(x)$: 6x+3

(Trusted setup phase)

Set $\mathbf{T(x) = (x - g1)(x - g2)}$

(Proving phase)

(Encodes constraints from all gates on all wires acting as left input)

assignment $W = \{w_1, w_2, w_3, w_4, w_5\}$,

(Single polynomial encodes constraints from all gates on all wires) → and $P(x) = L(x) * R(x) - O(x)$

When derived from a valid assignment, $P(x) = 0$ for $x \in \{g1, g2\} \Rightarrow P(x) = T(x)H(x)$

All R1CS constraints are compressed into a single polynomial equation that can be verified at a single point

Verifiable Computation Protocol

- Trusted setup (once per circuit):
  - Encode program structure into **polynomials**: $T(x),\ \{p(x)\} = \{L_{w_k}(x),\ R_{w_k}(x),\ O_{w_k}(x)\}$
  - Generate random point $\boldsymbol{s}$ and compute $T(\boldsymbol{s})$ [1]
- Prover
  - **Evaluate program** and generate an **assignment**, $W = \{w_1, w_2, w_3, w_4, w_5\}$
  - Using assignment to derive $P(x) = \left(\sum_{k\in W} w_k L_{w_k}(x)\right) * \left(\sum_{k\in W} w_k R_{w_k}(x)\right) - \left(\sum_{k\in W} w_k O_{w_k}(x)\right)$. Then compute $H(x) = \frac{P(x)}{T(x)}$ — Compute heavy. Uses FFT.
  - Generate **proof of computation**: $L(\boldsymbol{s}), R(\boldsymbol{s}), O(\boldsymbol{s}), H(\boldsymbol{s})$ [2] — Compute heavy. Requires many elliptic curve ops.
- Verifier
  - To verify the proof, check: $L(\boldsymbol{s}) * R(\boldsymbol{s}) - O(\boldsymbol{s}) = H(\boldsymbol{s}) * T(\boldsymbol{s})$ [3] [4]

[1] Homomorphically encrypted powers of $\boldsymbol{s}$ ($E[s^n], E[s^{n-1}], \ldots, E[s^1], E[s^0]$) & $E[T(\boldsymbol{s})]$ are generated, and then $\boldsymbol{s}$ is destroyed.

[2] Prover produces $E[L(\boldsymbol{s})], E[R(\boldsymbol{s})], E[O(\boldsymbol{s})]$, and $E[H(\boldsymbol{s})]$, since it only has access to encrypted power of $\boldsymbol{s}$.

[3] This check is performed in encrypted domain using cryptographic pairing friendly Elliptic curves.

[4] A check that forces the prover to only use the encrypted power of $\boldsymbol{s}$ is also performed. This requires additional randomness from trusted setup.

Universal and transparent SNARKs

Previous approach requires a trusted set-up for each circuit, to:

- Encode program structure into **polynomials**: $T(x)$, $\{p(x)\} = \{L_{w_k}(x),\ R_{w_k}(x),\ O_{w_k}(x)\}$
- Generate random point $\boldsymbol{s}$ and compute $T(\boldsymbol{s})$

A **universal** protocol does not require a trusted set-up *for each circuit*.

A **transparent** protocol does not require any trusted set-up at all, instead uses public randomness.

Generalizing to other types of zk-SNARKs

In general, you need two ingredients:

1. A **polynomial commitment scheme**: a way for the prover to commit to the polynomial once.
2. An **interactive oracle proof**: interactive protocol between prover and verifier.

- Computation
- Algebraic Circuit
- R1CS
- QAP
- Linear PCP
- Linear Interactive Proof
- zkSNARK

1. Polynomial commitment

We need to “commit” to a polynomial. This has two properties:

- **Binding**: once the polynomial has been committed, you cannot change it.
- **Concealing**: it does not reveal the polynomial.

General procedure:

- Prover binds itself to a polynomial P by sending a short string $Com(P)$.
- Verifier chooses an x and asks P to evaluate $P(x)$.
- P sends $y = P(x)$, and a proof π that shows that y is consistent with $Com(P)$ and x.

In Pinocchio/Groth16, this is part of the trusted set-up (secret point $\boldsymbol{s}$ and corresponding $T(\boldsymbol{s})$).

In universal protocols, this happens later.

There are many polynomial commitments in literature: Kate/KZG, Bulletproofs, Hyrax, Dory, FRI, Ligero, Brakedown, Orion…

2. Interactive Oracle Proof (IOP)

Protocol in which prover and verifier interact to convince that a statement is true.

In our case, that we know a polynomial that satisfies a property (divisible by a certain polynomial).

In Pinocchio/Groth16:

- Verifier/trusted party: generates random point $\boldsymbol{s}$ and compute $T(\boldsymbol{s})$
- Prover: generates proof of computation: $L(\boldsymbol{s}), R(\boldsymbol{s}), O(\boldsymbol{s}), H(\boldsymbol{s})$ (= QAP evaluated in $\boldsymbol{s}$)
- Verifier: checks $L(\boldsymbol{s}) * R(\boldsymbol{s}) - O(\boldsymbol{s}) = H(\boldsymbol{s}) * T(\boldsymbol{s})$

There are many IOPs in literature: Marlin, Plonk…

You can **mix and match** different polynomial commitment schemes with different IOPs
to get different trade-offs between proving time, verification time, proof size, need for set-up, etc.

There's a trick to convert an interactive proof protocol into a non-interactive one: the **Fiat-Shamir transformation**.

Recursive proofs

A **recursive** proof builds on top of a previous proof, to prove an incremental computation.

E.g. $y = F^i(x)$

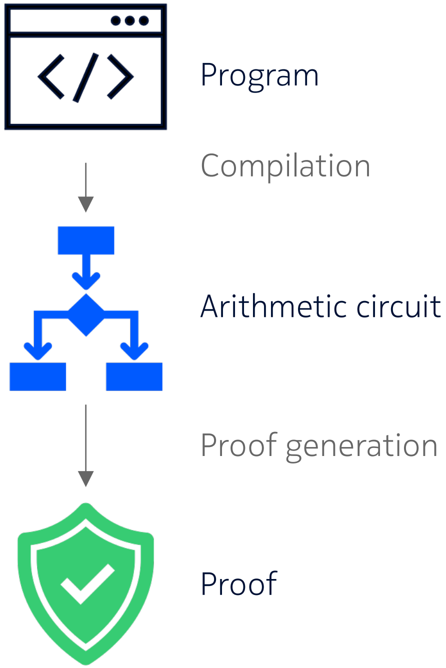
The proof for i will contain a verifier circuit for the previous proof $i - 1$ + the circuit of the current computation.

Kothapalli, Setty, Tzialla, (2022). "Nova: Recursive zero-knowledge arguments from folding schemes." *CRYPTO 2022*.
https://github.com/microsoft/Nova

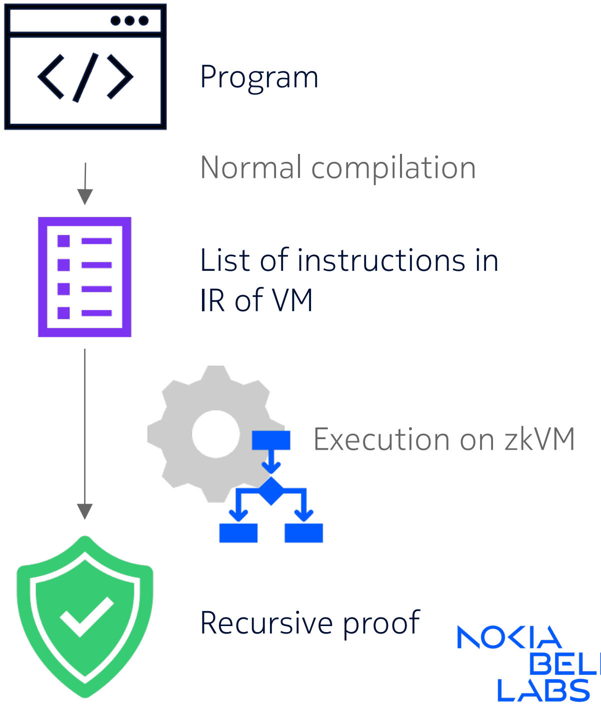
zk-VMs

Imagine: F = processing of one VM instruction

Direct execution using arithmetic circuits



Execution using VM and recursive proofs



NOKIA BELL LABS

MultiAND in RISC-Zero

https://www.risczero.com

```rust
#![no_main]
#![no_std]

use risc0_zkvm::guest::env;

risc0_zkvm::guest::entry!(main);

const N: usize = 10;

pub fn main() {
    let mut sum = 0;
    for _i in 0..N {
        let x: u8 = env::read();
        sum += x;
    }
    assert!(usize::from(sum) == N);
}
```

Compared to Circom:

- Support for strings, floating-point numbers, etc.
- Support for most of Rust (no IO, no random numbers…)
- Larger proofs: $O(\log(|C|))$
  vs. constant size for Circom/Groth16

References

Tutorials & courses:

- https://zkp.science: overview of papers, proof systems, implementations…
- https://zk-learning.org: MOOC by Dan Boneh and others
- "The Mathematics behind zk-SNARKs" (https://www.youtube.com/watch?v=iRQw2RpQAVc): in-depth math of Groth16

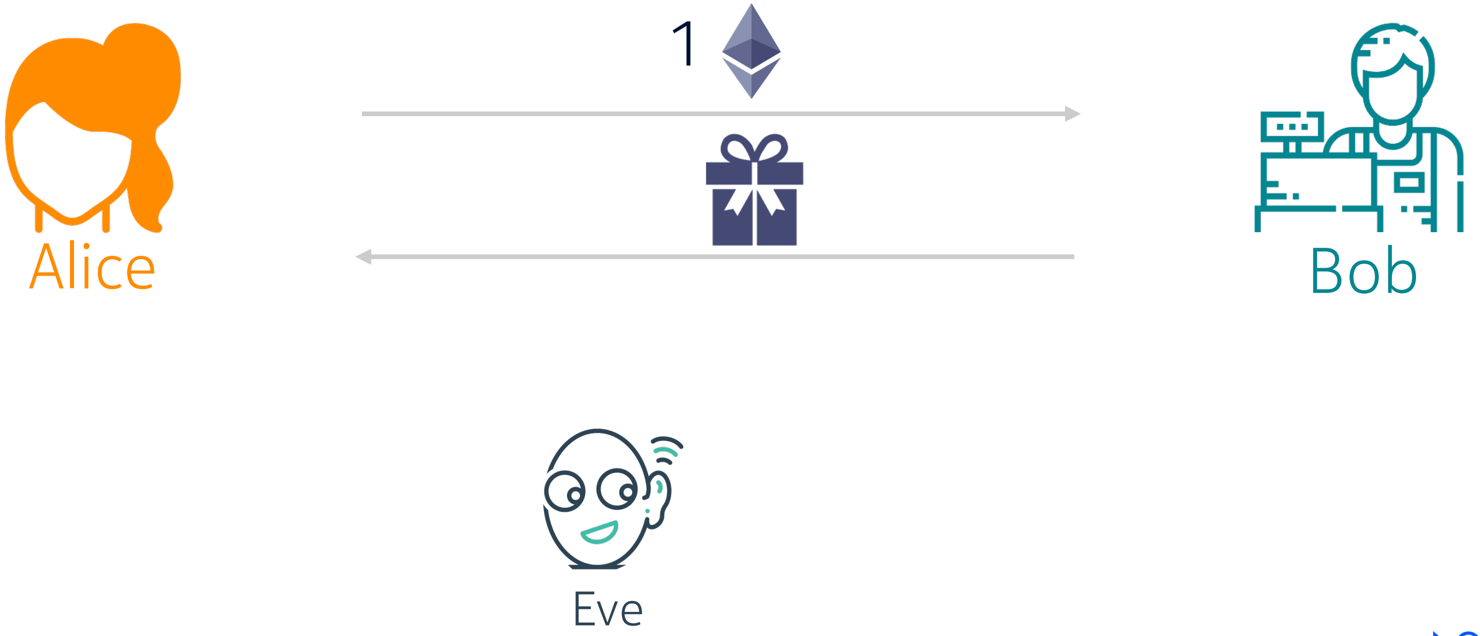
Software & tools:

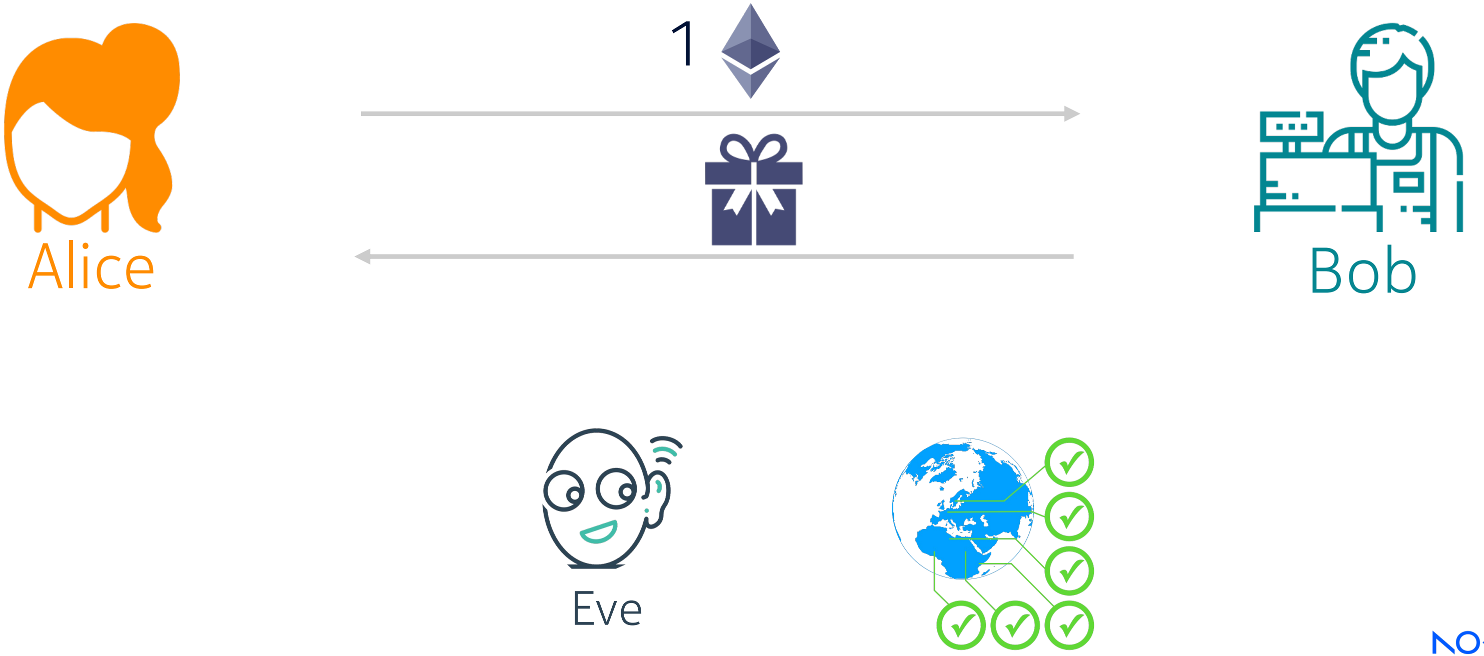
- Circom (https://docs.circom.io): circuit language that compiles to SNARKs
- Zokrates (https://zokrates.github.io): Python-like language that compiles to SNARKs
- RISC Zero (https://www.risczero.com): zero-knowledge VM (based on STARK, not SNARK)

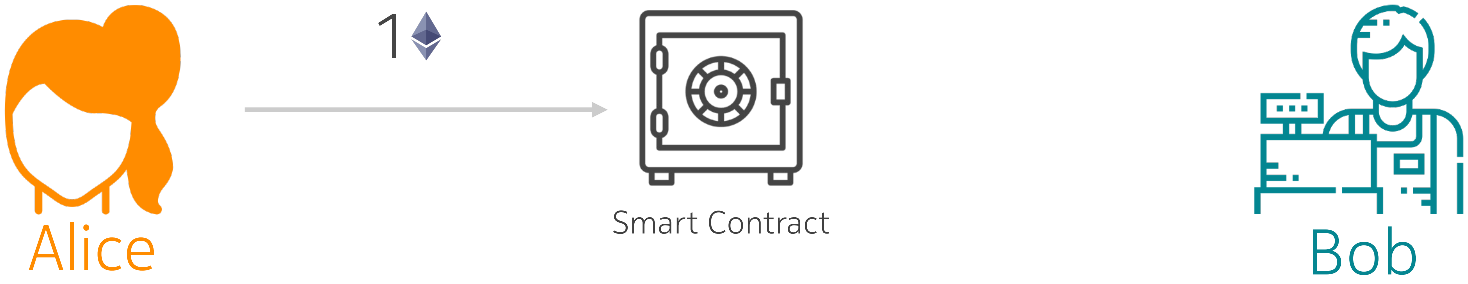
NOKIA BELL LABS

Private TXs with Tornado Cash

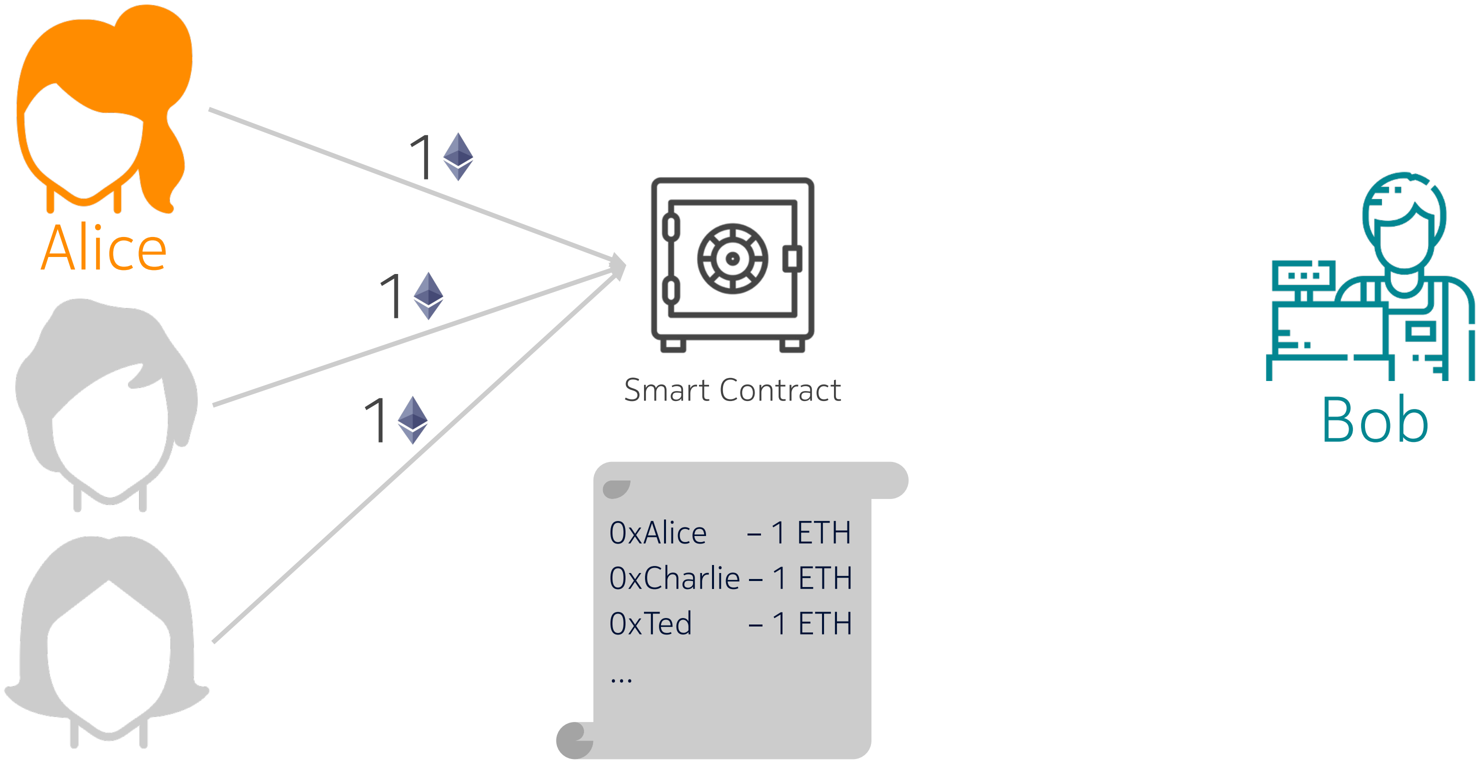
NOKIA BELL LABS


1
Alice
Bob
Eve

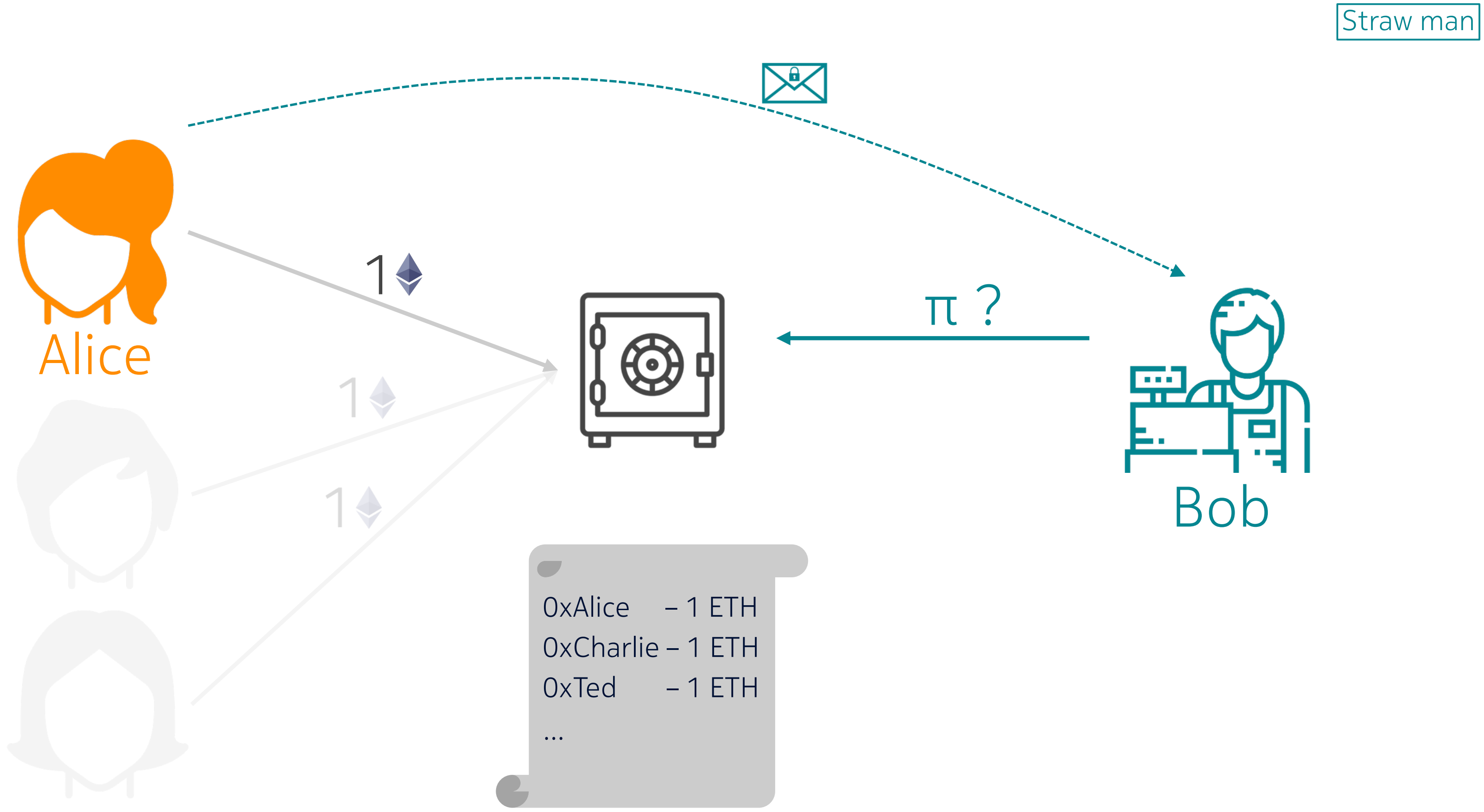

Alice
1
Bob
Eve



  | Smart Contract Programmer, Tornado Cash - How it Works | DeFi + Zero Knowledge Proof https://www.youtube.com/watch?v=z_cRicXX1jI

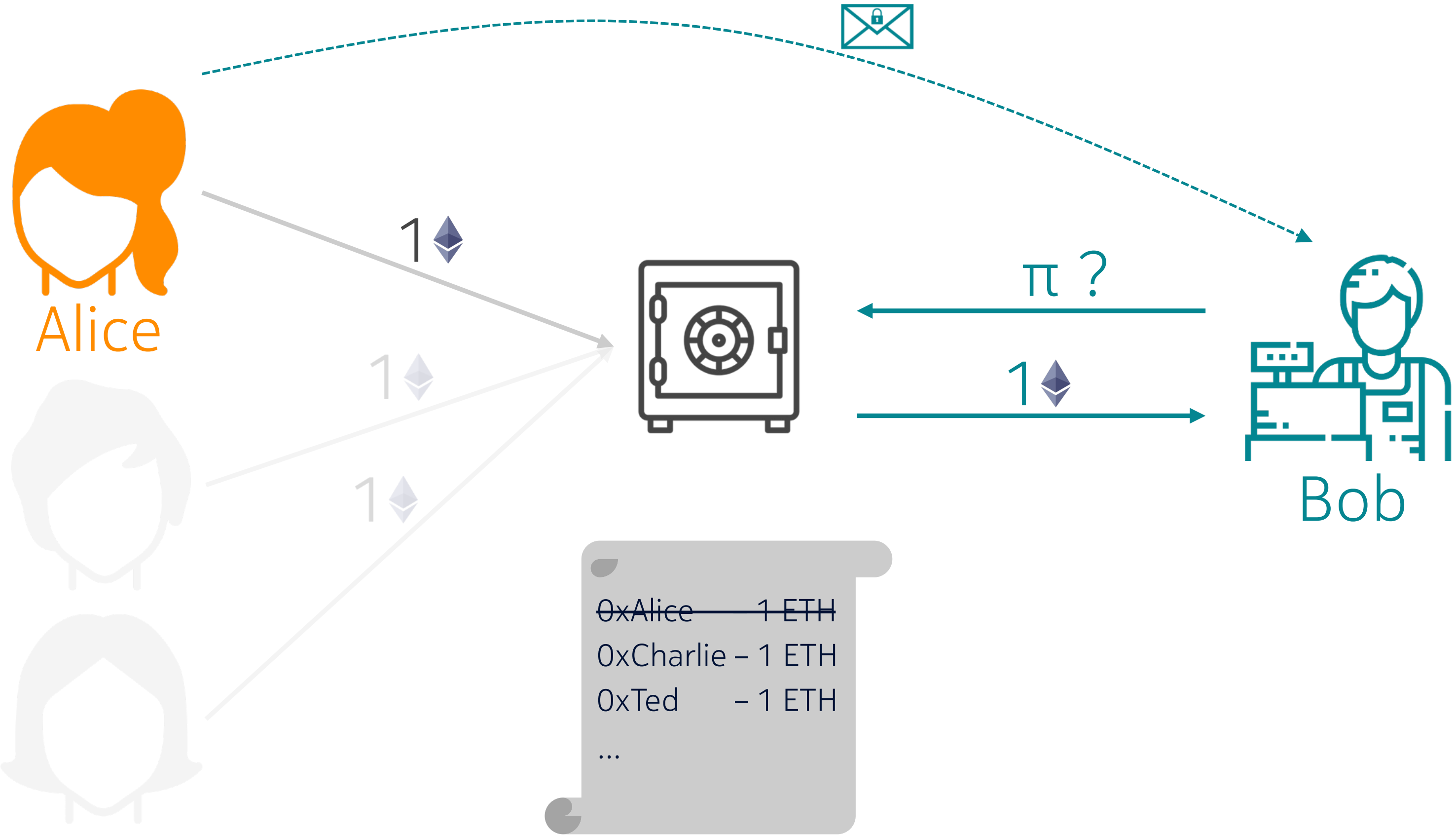


  | Smart Contract Programmer, Tornado Cash - How it Works | DeFi + Zero Knowledge Proof https://www.youtube.com/watch?v=z_cRicXX1jI

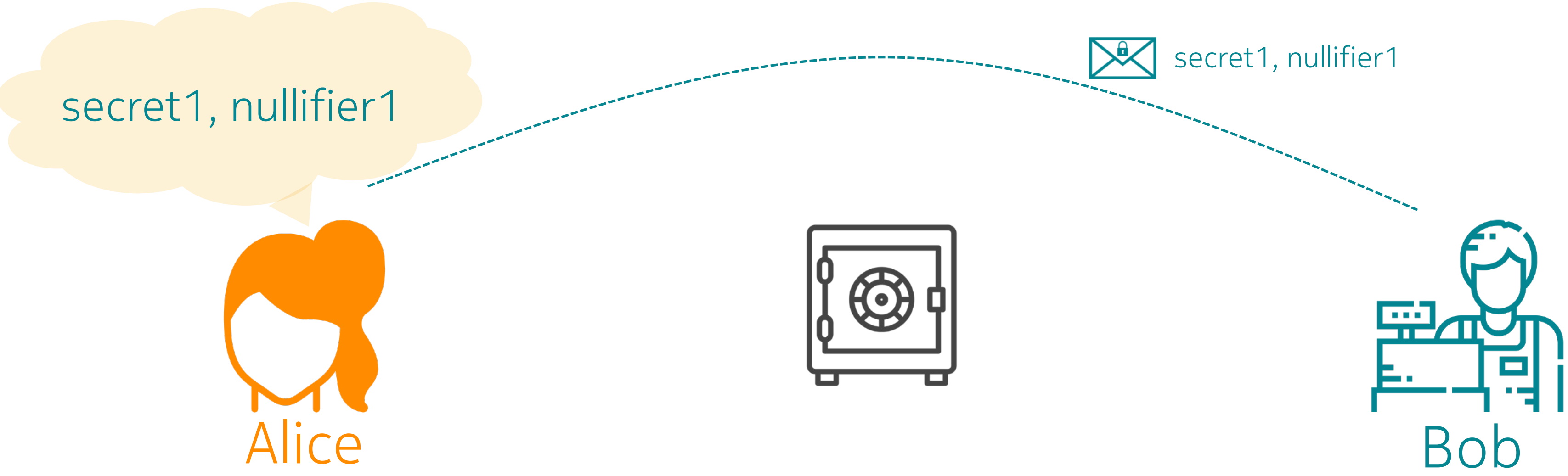


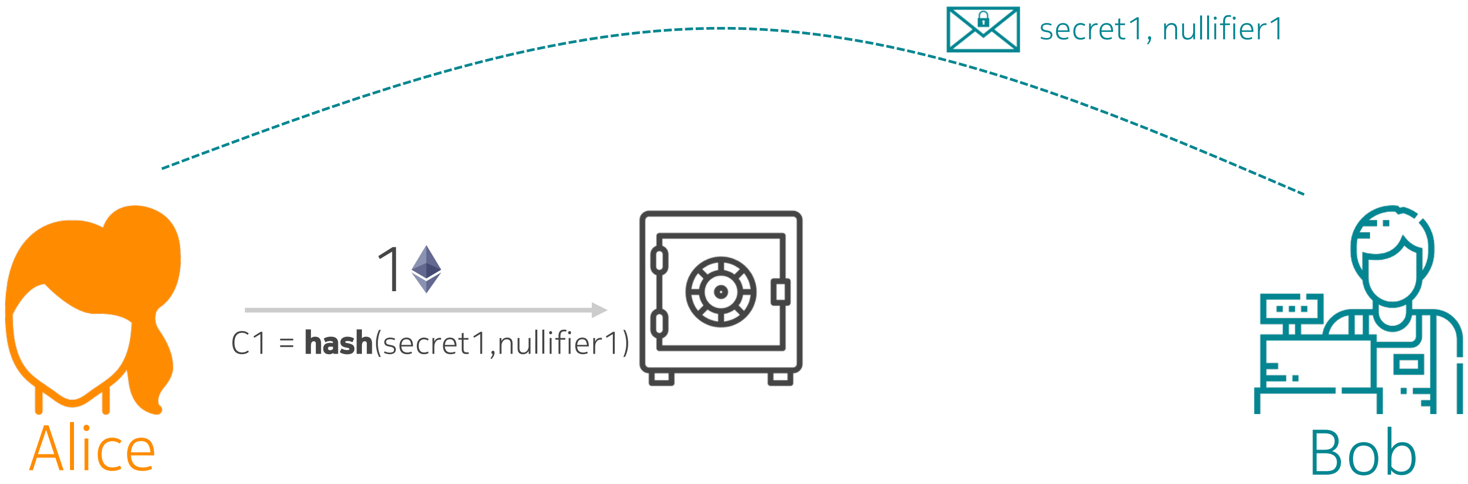
  | Smart Contract Programmer, Tornado Cash - How it Works | DeFi + Zero Knowledge Proof https://www.youtube.com/watch?v=z_cRicXX1jI

Straw man



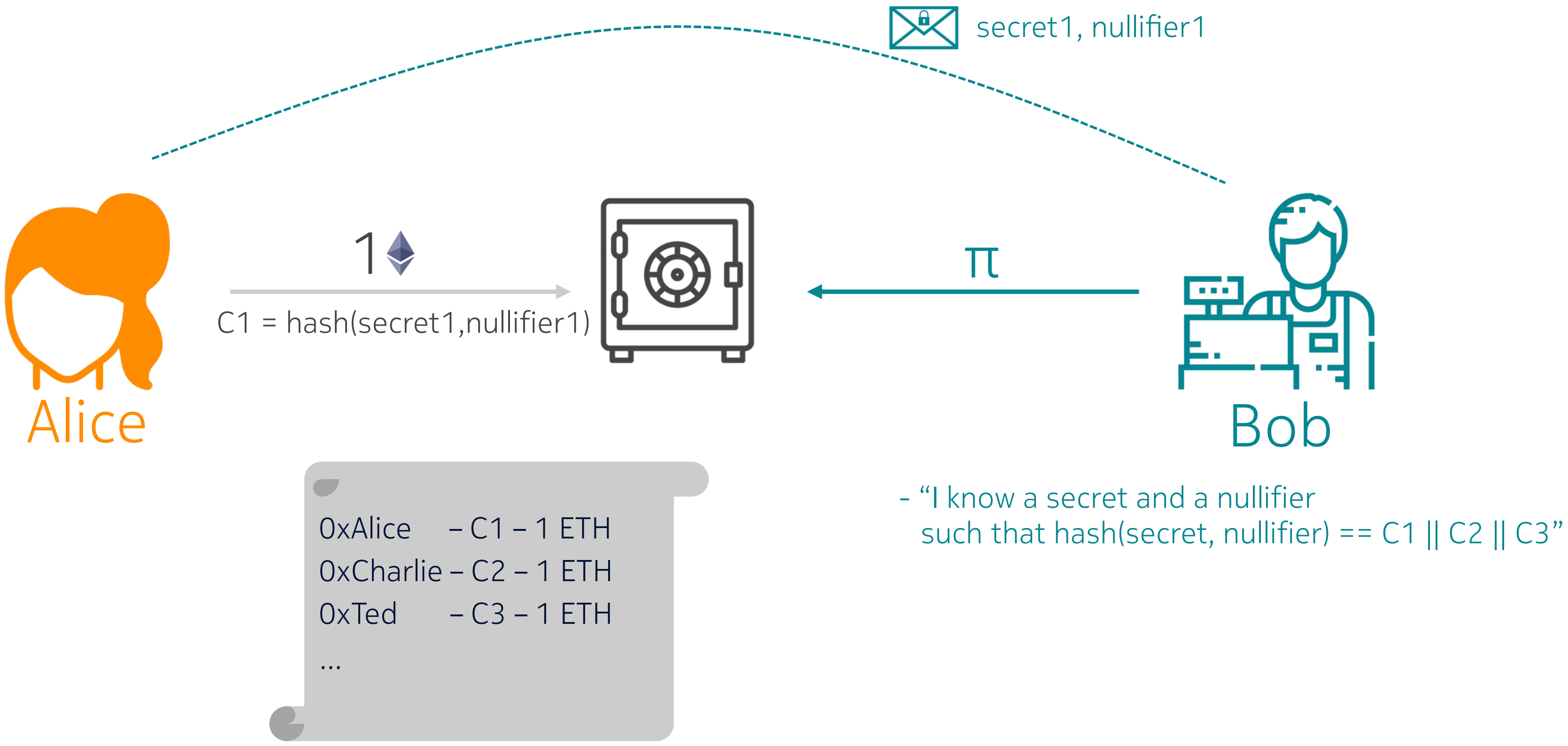
  | Smart Contract Programmer, Tornado Cash - How it Works | DeFi + Zero Knowledge Proof https://www.youtube.com/watch?v=z_cRicXX1jI

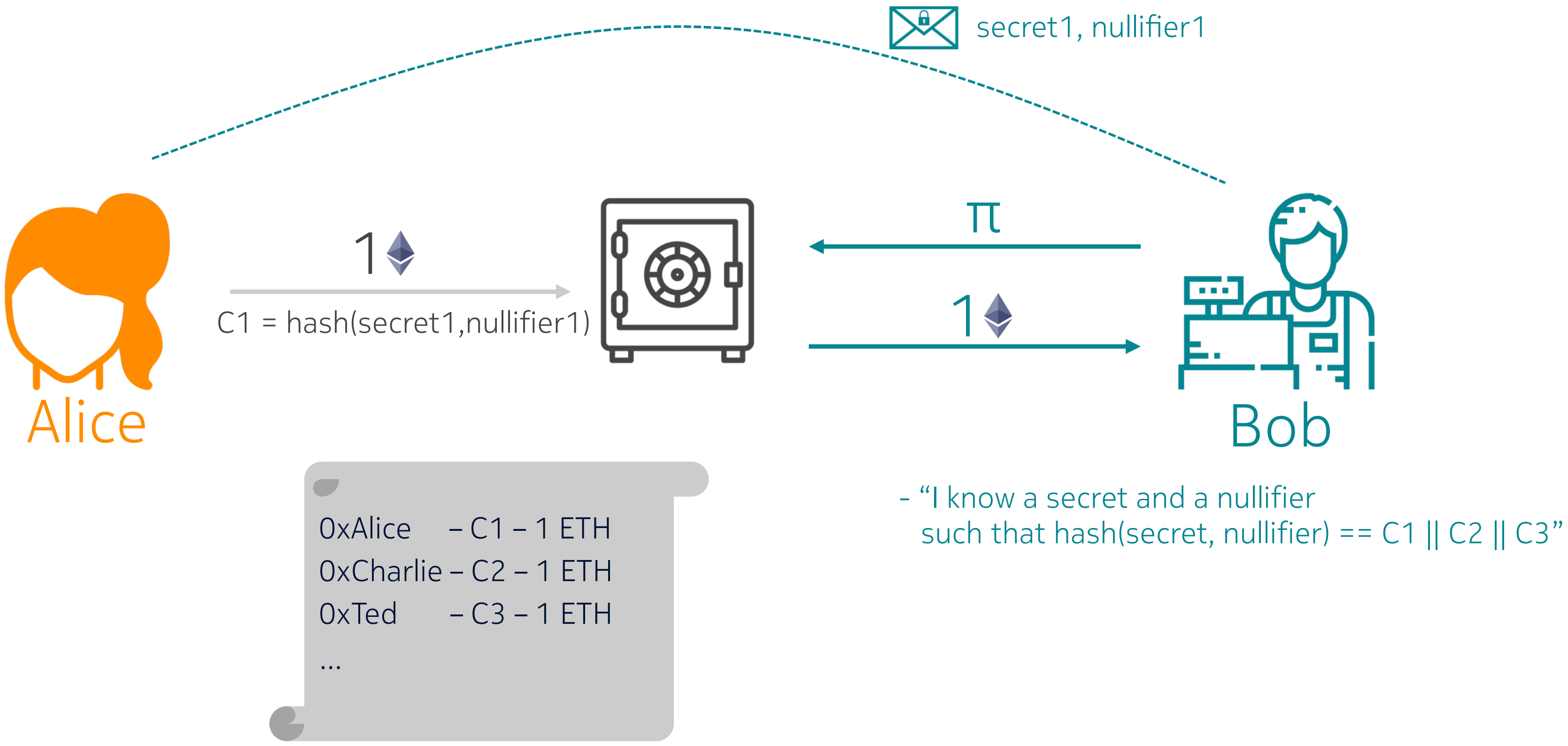

secret1, nullifier1
secret1, nullifier1
Alice
Bob

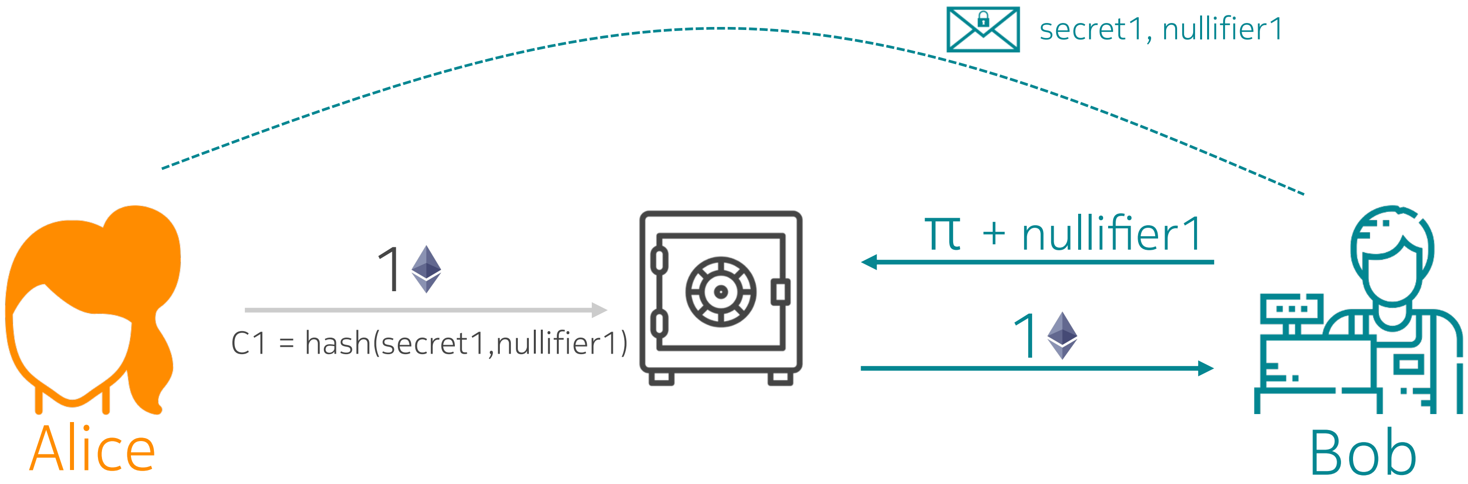


0xAlice – C1 – 1 ETH
0xCharlie – C2 – 1 ETH
0xTed – C3 – 1 ETH
...

NOKIA BELL LABS


secret1, nullifier1
1
C1 = hash(secret1,nullifier1)
π
Alice
Bob
0xAlice – C1 – 1 ETH
0xCharlie – C2 – 1 ETH
0xTed – C3 – 1 ETH
…
- "I know a secret and a nullifier
such that hash(secret, nullifier) == C1 || C2 || C3"

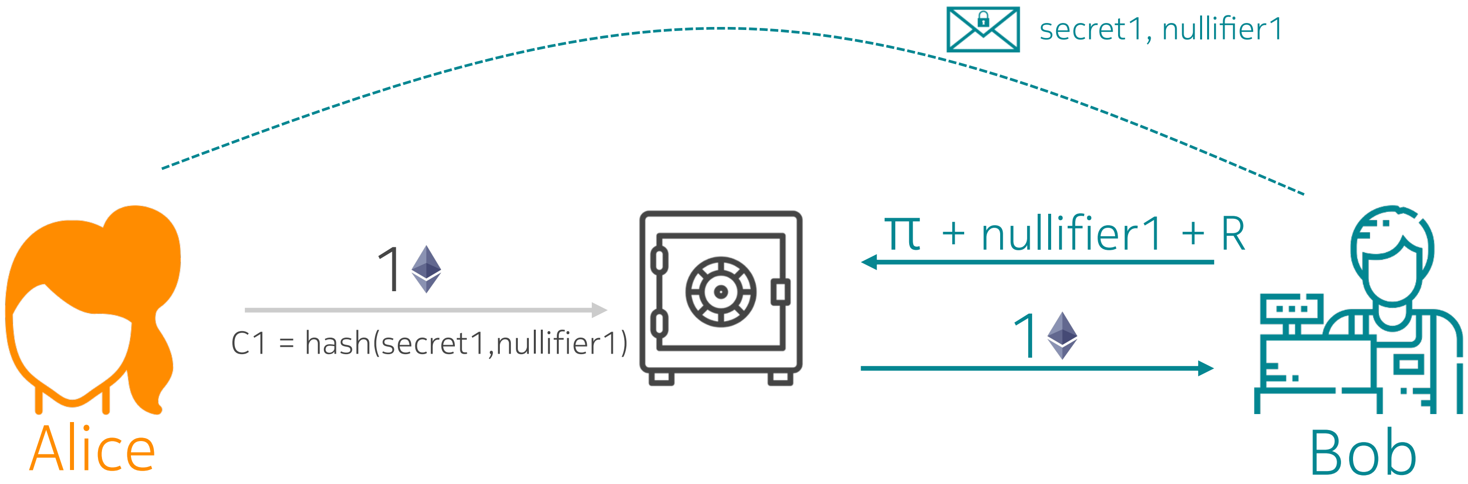

secret1, nullifier1
Alice
1
C1 = hash(secret1,nullifier1)
π
1
Bob
0xAlice – C1 – 1 ETH
0xCharlie – C2 – 1 ETH
0xTed – C3 – 1 ETH
…
- "I know a secret and a nullifier
such that hash(secret, nullifier) == C1 || C2 || C3"

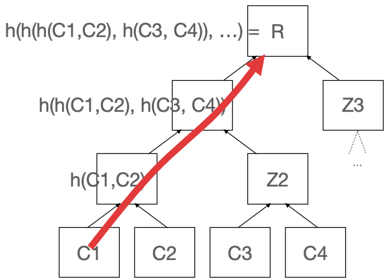


0xAlice – C1 – 1 ETH
0xCharlie – C2 – 1 ETH
0xTed – C3 – 1 ETH

0xBob – nullifier1 – -1 ETH

- "I know a secret and a nullifier such that hash(secret, nullifier) == C1 || C2 || C3"
- And I reveal the nullifier used to compute π





0xAlice – C1 – 1 ETH
0xCharlie – C2 – 1 ETH
0xTed – C3 – 1 ETH

0xBob – nullifier1 – -1 ETH

- "I know a secret and a nullifier such that hash(secret, nullifier) == C1"
- And I reveal the nullifier used to compute π
- And I prove a path from C1 to R


```
/**
  @dev Withdraw a deposit from the contract. `proof` is a zkSNARK proof data, and input is an array of circuit public inputs
  `input` array consists of:
    - merkle root of all deposits in the contract
    - hash of unique deposit nullifier to prevent double spends
    - the recipient of funds
    - optional fee that goes to the transaction sender (usually a relay)
*/
function withdraw(
  bytes calldata _proof,
  bytes32 _root,
  bytes32 _nullifierHash,
  address payable _recipient,
  address payable _relayer,
  uint256 _fee,
  uint256 _refund
) external payable nonReentrant {
  require(_fee <= denomination, "Fee exceeds transfer value");
  require(!nullifierHashes[_nullifierHash], "The note has been already spent");
  require(isKnownRoot(_root), "Cannot find your merkle root"); // Make sure to use a recent one
  require(
    verifier.verifyProof(
      _proof,
      [uint256(_root), uint256(_nullifierHash), uint256(_recipient), uint256(_relayer), _fee, _refund]
    ),
    "Invalid withdraw proof"
  );

  nullifierHashes[_nullifierHash] = true;
  _processWithdraw(_recipient, _relayer, _fee, _refund);
  emit Withdrawal(_recipient, _nullifierHash, _relayer, _fee);
}
```


Tornado Cash Core, Withdraw@Tornado.sol, ~~https://github.com/tornadocash/tornado-core/blob/master/contracts/Tornado.sol#L69~~

```solidity
/**
  @dev Withdraw a deposit from the contract. `proof` is a zkSNARK proof data, and input is an array of circuit public inputs
  `input` array consists of:
    - merkle root of all deposits in the contract
    - hash of unique deposit nullifier to prevent double spends
    - the recipient of funds
    - optional fee that goes to the transaction sender (usually a relay)
*/
function withdraw(
  bytes calldata _proof,
  bytes32 _root,
  bytes32 _nullifierHash,
  address payable _recipient,
  address payable _relayer,
  uint256 _fee,
  uint256 _refund
) external payable nonReentrant {
  require(_fee <= denomination, "Fee exceeds transfer value");
  require(!nullifierHashes[_nullifierHash], "The note has been already spent");
  require(isKnownRoot(_root), "Cannot find your merkle root"); // Make sure to use a recent one
  require(
    verifier.verifyProof(
      _proof,
      [uint256(_root), uint256(_nullifierHash), uint256(_recipient), uint256(_relayer), _fee, _refund]
    ),
    "Invalid withdraw proof"
  );

  nullifierHashes[_nullifierHash] = true;
  _processWithdraw(_recipient, _relayer, _fee, _refund);
  emit Withdrawal(_recipient, _nullifierHash, _relayer, _fee);
}
```

Tornado Cash Core, Withdraw@Tornado.sol, ~~https://github.com/tornadocash/tornado-core/blob/master/contracts/Tornado.sol#L69~~

Other use cases

Proof that you are older than 18

to access stellaartois.com


```
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA256
{"first_name": "Janwillem",
 "last_name": "Swalens",
 "birth_date": "1990-09-08",
 "birth_place": "Jette, Belgium",
 "nationality": "BE",
 "national_registry_number": "90.09.08-123.45",
 "address": "Xyz 12, 1000 Brussel"}
-----BEGIN PGP SIGNATURE-----
iEYEARECAAYFAjdYCQoACgkQJ9S6ULt1dqz6IwCfQ7wP6i/i8
HhbcOSKF4ELyQB1oCoAoOuqpRqEzr4kOkQqHRLE/b8/Rw2k
=y6kj
-----END PGP SIGNATURE-----
```

→

```
f(data, now):
  assert(signature_valid(data))
  json = parse_json(data)
  birth_date = parse_iso8601_date(json["birth_date"])
  delta_t = time_diff(now, birth_date)
  if delta_t > 60*60*24*365*18:
    return true
  else:
    return false
```

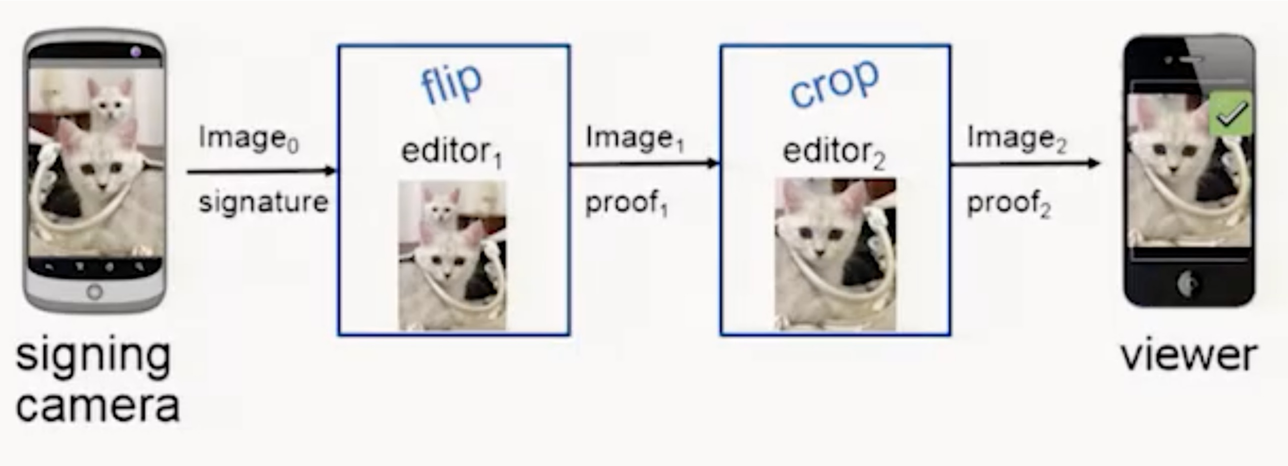
→ true

Government-issued ID
signed by government
but contains private details

Program that verifies signature,
parses data
and checks age

We just return true, and a
proof that the program
was executed correctly.

PhotoProof

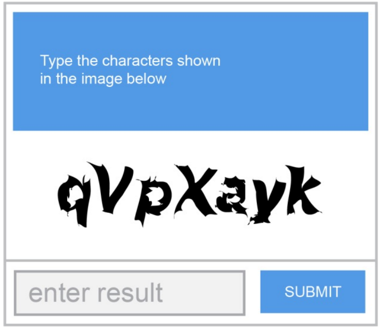


Naveh, Tromer, (2016). “Photoproof: Cryptographic image authentication for any set of permissible transformations.” In *2016 IEEE Symposium on Security and Privacy (SP).*
https://www.youtube.com/watch?v=k6FILzAy4tU

NOKIA BELL LABS

Privacy Pass by Cloudflare

After a single CAPTCHA is solved, 30 tokens are generated, to prevent future CAPTCHAs.



Davidson, Goldberg, Sullivan, Tankersley, Valsorda, (2018). "Privacy Pass: Bypassing Internet Challenges Anonymously". In *Proceedings on Privacy Enhancing Technologies*.
https://privacypass.github.io
https://support.cloudflare.com/hc/en-us/articles/115001992652-Using-Privacy-Pass-with-Cloudflare

Conclusion

Zero-Knowledge Proofs are useful

on the blockchain & beyond!

ZKPs allow you to prove that a computation was executed correctly, while hiding inputs.

This is useful for:

- Smart contracts
- Privacy on blockchains
- Compute on privacy-sensitive data
- Proving identity
- Scaling blockchains (roll-up)
- Compute on commercially sensitive data

Exciting area with many new developments:

- hard-core mathematics: new proving systems, new polynomial commitment schemes, new IOPs
- tooling: new frameworks, libraries, languages
- use cases: as tools get faster, more opportunities open up

  | janwillem.swalens@nokia-bell-labs.com lode.hoste@nokia-bell-labs.com

NOKIA BELL LABS

Copyright and confidentiality