Building with ZKPs: From Concepts to Code

Tarek Galal · Cedarcrypt 2026, Paphos

Cryptography sounds magical

We can share a secret in front of everyone??

Other magical stuff:

- (Shamir) secret sharing
- Threshold signatures & encryption
- **Polynomial commitments**
- Homomorphic encryption
- **Zero-knowledge proofs**
- Group signatures

This Workshop

Building with Zero-Knowledge Proofs: From ~~Concepts to Code~~ Code to Concepts

Goals:

- Go through (most of) the journey from high level description of a computation all the way to the proof generation and verification.
- enough **intuition** that zk-SNARKs stop being a mysterious black box
- enough **excitement** to go explore the magical small pieces and their other applications on your own

> Disclaimer: Intuition over rigor

Suggestion: Take notes and interrupt me

Part I: Background & ZKP Recap

Schnorr Identification: Proving knowledge of private key

Prover proves she knows x with $X = g^x$, revealing nothing about x:

Prover · knows x		Verifier · knows X
commit: pick random y, $Y = g^y$	$\xrightarrow{Y}$	
	$\xleftarrow{c}$	**challenge:** pick random c
response: $r = y + c\,x$	$\xrightarrow{r}$	**check:** $g^r \overset{?}{=} Y \cdot X^c$

- **Completeness**: honest Prover always convinces Verifier.
- **Special soundness**: two accepting transcripts on the same commitment Y extracts x.
- **Honest-verifier ZK**: the transcript is simulatable; Verifier learns nothing.

Sigma Protocols

Same three moves, now abstract from "I know x such that $X = g^x$" onto "I know a witness w such that $R(X, w) = \text{true}$"* for a public instance X , revealing nothing about w:

Prover · knows witness w		Verifier · knows instance X
commit: compute commitment a	$\xrightarrow{a}$	
	$\xleftarrow{c}$	**challenge:** pick random c
response: compute z	$\xrightarrow{z}$	**check:** accept iff $V(X, a, c, z)$

- **Completeness**: if $R(X, w)$ holds, an honest Prover is always accepted.
- **Special soundness**: two accepting transcripts (a, c, z), (a, c', z'), $c \neq c' \Rightarrow$ *extract* w.
- **Honest-verifier ZK**: a simulator given only (X, c) fakes accepting transcripts without knowing w.

Fiat–Shamir: kill the interaction

Replace the verifier's random challenge with a hash of the transcript:

$$c = H(\ \text{public inputs} \ \|\ a\ )$$

The proof becomes a **single message** with no back-and-forth.

Proving *any* computation

What if we want to prove a statement like "I know x such that `sha256(x) = y`"?

Claim: Any computation can be modeled as a **circuit**

In 2016:

> We construct a NIZK argument for arithmetic **circuit** satisfiability where a proof consists of only 3 group elements. In addition to being small, the proof is also easy to verify. (Jens Groth, 2016)

→ AKA zk-SNARK

zk-SNARKs

Succinct · **N**on-Interactive · **A**rgument · of **K**nowledge

- **Succinct:** the proof is *small*, verification is *fast*.
- **Non-Interactive:** one message, no back-and-forth.
- **Argument:** *computational* soundness (vs. a "proof": statistically infeasible to break).
- **of Knowledge:** the prover genuinely *knows* a witness.

Essentially a Non-interactive ZKP (NIZK) with small, easy to verify proofs.

Part II: The Pipeline & Examples

> Real-World Cryptography (book, p.335), Wong, 2021:
>
> Most zk-SNARK schemes build on a proving system, **and a translation of a program to something the proving system can prove**.

The starting point: a high-level DSL

We start with ordinary-looking code (language) for checking: $x^3 + x + 5 = 35$

```
// ZoKrates
def main(private field x) {
    assert(x**3 + x + 5 == 35);
}
```

```
// Noir: parameters private by default
fn main(x: Field) { assert(x * x * x + x + 5 == 35); }
```

```
// circom: x is private
template Cube() {
  signal input x; signal x2; signal x3;
  x2 <== x * x;   x3 <== x2 * x;   x3 + x + 5 === 35;
}
component main = Cube();
```

Our running example for the whole talk: prove you know a secret `x` with `x³ + x + 5 == 35` and largely based on https://vitalik.eth.limo/general/2016/12/10/qap.html

The finish line: Groth16

Groth16 generates **tiny** proofs for special form of arithmetic circuits: Quadratic Arithmetic Programs (QAP)

- the proof is **only 3 group elements**,
- verification is **~3 pairings** which is **constant**, whatever the circuit's size.

What is *exactly* being proven?

Not the program; nothing "runs in zero knowledge," and the program is **fixed, public, and trusted** in advance.

We prove that **one specific execution** of that known program is correct i.e., the claimed output really follows from the inputs.

- some/all inputs stay **hidden** from the verifier;
- verification is **far cheaper** than re-running.

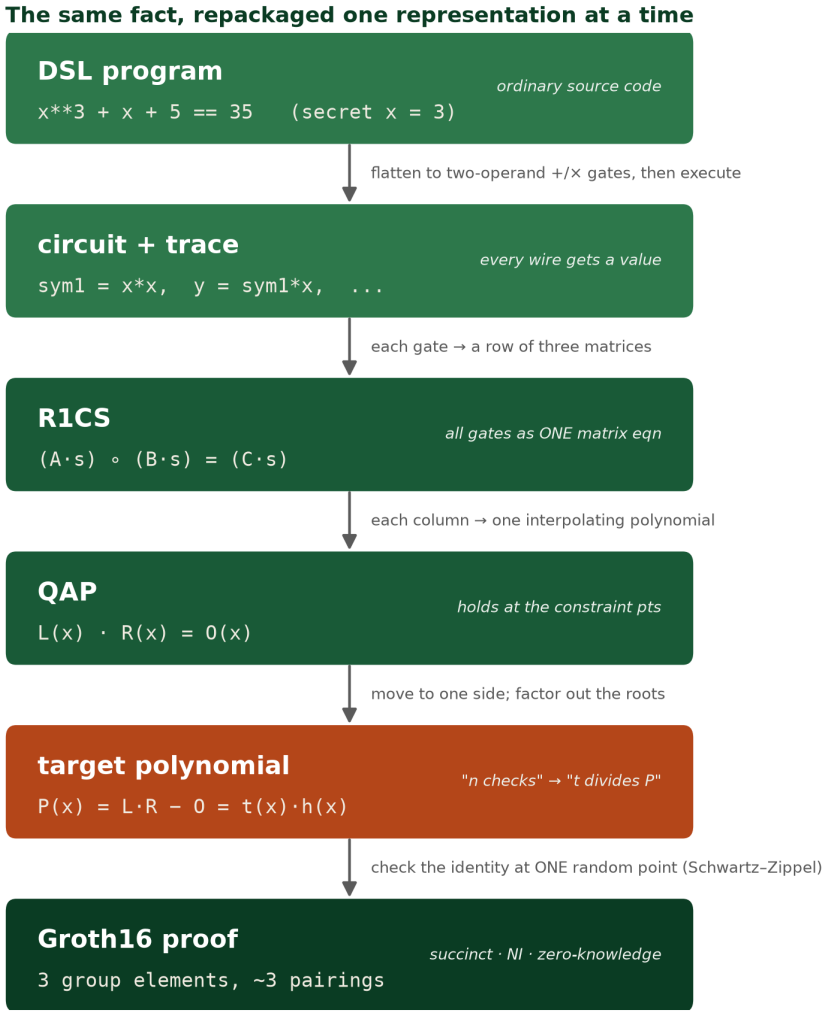
> Think **unit tests**: we're asserting *this function on these inputs produces this output*.

What is *exactly* being proven? (Lookahead teaser)

Prover proves knowledge of $P(x)$ and $h(x)$ such that $P(x) = t(x)h(x)$ for some target polynomial $t(x)$.

> Real-World Cryptography (book, p.335), Wong, 2021:
>
> But that's it! That's what zk-SNARKs proving systems usually provide: something to prove that you know some polynomial. I'm repeating this because the first time I learned about that it made no sense to me. **How can you prove that you know some secret input to a program if all you can prove is that you know a polynomial?**

The roadmap


The same fact, repackaged one representation at a time
DSL program
x**3 + x + 5 == 35 (secret x = 3)
ordinary source code
flatten to two-operand +/× gates, then execute
circuit + trace
sym1 = x*x, y = sym1*x, ...
every wire gets a value
each gate → a row of three matrices
R1CS
(A·s) ∘ (B·s) = (C·s)
all gates as ONE matrix eqn
each column → one interpolating polynomial
QAP
L(x) · R(x) = O(x)
holds at the constraint pts
move to one side; factor out the roots
target polynomial
P(x) = L·R − O = t(x)·h(x)
"n checks" → "t divides P"
check the identity at ONE random point (Schwartz–Zippel)
Groth16 proof
3 group elements, ~3 pairings
succinct · NI · zero-knowledge

First: Demos

https://github.com/tgalal/zokrates-demos

Short Discussion

01-workshop_example

```
$ make
Compiling main.zok

Compiled code written to 'out'
Number of constraints: 3
...
```

```
$ ls -l

1.3K out
 576 out.r1cs
 204 out.wtns
 773 proof.json
2.1K proving.key
1.4K verification.key
 168 witness
```

03-bitcoin_pow

```
$ make
Compiling main.zok

Compiled code written to 'out'
Number of constraints: 86106
...
```

```
$ls -lh

405M out
164M out.r1cs
2.7M out.wtns
1.4K proof.json
 35M proving.key
2.7K verification.key
3.3M witness
```

Part III: Preprocessing

Turning a program to something provable

Part III: Preprocessing

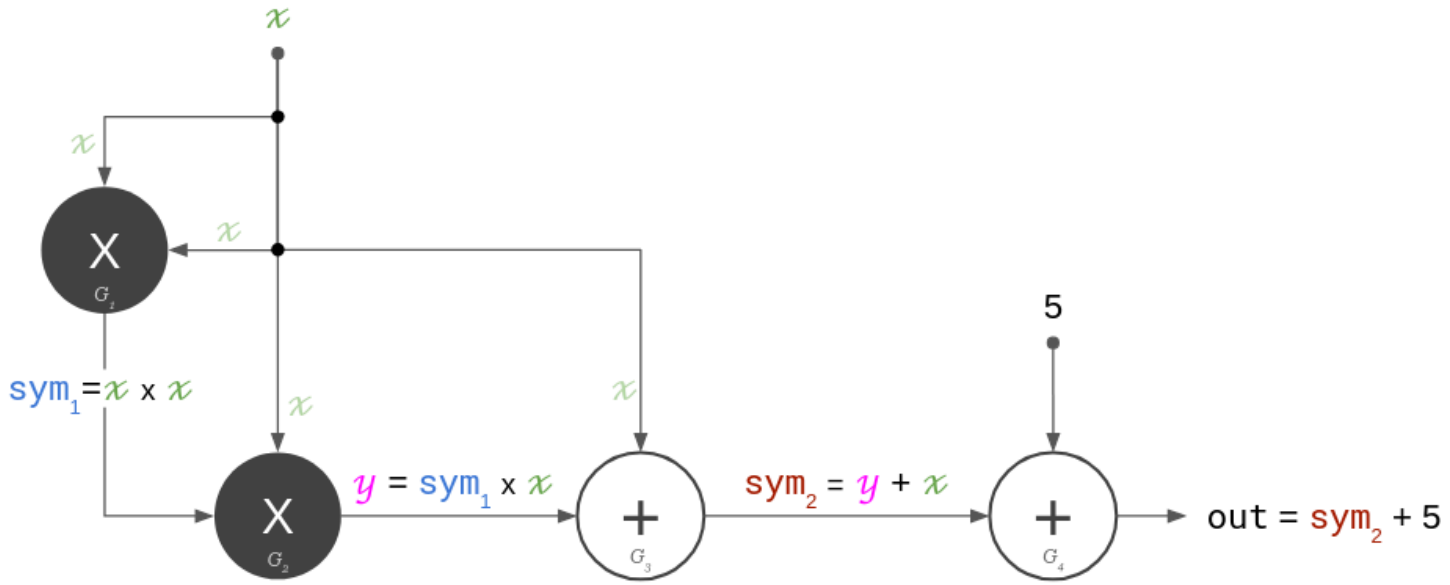
Step 1: Encode the program as math

From source code to an arithmetic circuit

Claim: every program is just + and ×

> Any program can be rewritten as equations using **only additions and multiplications**.

Our example: $x^3 + x + 5 = 35$



$$
\begin{aligned}
G_1 &: \ \mathtt{sym}_1 = x \times x \\
G_2 &: \ y = \mathtt{sym}_1 \times x \\
G_3 &: \ \mathtt{sym}_2 = y + x \\
G_4 &: \ \mathtt{out} = \mathtt{sym}_2 + 5
\end{aligned}
$$

- Lets us prove **every intermediate step** is correct (execution trace)
- Puts the program in the form the next stage (R1CS) needs.

The claim also holds for non-mathy code

- **Loops** get *unrolled*: a fixed-bound `for` becomes its body repeated
- **Subtraction** is addition in disguise: `a - b = a + (−1)·b`
- **Branches** multiply by a branch-activator bit. To encode `out = cond ? a : b`:

$$\texttt{out} = \texttt{cond} \cdot a + (1 - \texttt{cond}) \cdot b$$

`cond = 1` gives `a`; `cond = 0` gives `b` (**both** branches are evaluated).

05-branching demo:

```
def main(field x) -> field {
    return if x == 0 {
        0
    } else {
        1 / x
    };
}
```

```
$ zokrates compute-witness -i out -a 0

Computing witness...
Execution failed: Division by zero
make: *** [../common.mk:43: witness] Error 1
```

Witness Generation

```
# 01-workshop_example
$ zokrates compute-witness --arguments "3"
```

Run the program on the secret input **x = 3** and every wire gets a value. We call the filled-in steps the **execution trace**, and the collected values the **witness**:

wire	value/witness
x	3
$\texttt{sym}_1 = x \cdot x$	9
$y = \texttt{sym}_1 \cdot x$	27
$\texttt{sym}_2 = y + x$	30
$\texttt{out} = \texttt{sym}_2 + 5$	35

Proving the statement means proving that every gate's inputs and output obey that gate's rule.

Part III: Preprocessing

Step 2: Rewrite the Circuit as R1CS

packing every gate into one matrix equation

Goal: Compact the Equations

$$x^3 + x + 5 = 35$$

LHS	RHS	Gate Output	$x = 3$
x	x	$\mathtt{sym}_1$	9
$\mathtt{sym}_1$	x	y	27
$y + x$	1	$\mathtt{sym}_2$	30
$\mathtt{sym}_2 + 5$	1	out	35

Right now we'd validate the trace by recomputing **each** step and comparing. We want **one** check instead:

$$L \cdot R = O$$
$$(A\mathbf{s}) \circ (B\mathbf{s}) = (C\mathbf{s})$$

Read it as: **one linear combination times another equals a third**, applied to the witness $\mathbf{s}$.

- We are going to build the matrices A,B,C row by row.
- Each gate adds one row in each

This is a **Rank-1 Constraint System**: every constraint has the shape (linear) × (linear) = (linear).

Building Matrix Rows

First, stack all variables into one vector. Start with **1**, then the public output, then the private and intermediate wires:

$$\mathbf{s} = \big(\overset{0}{1},\ \overset{1}{\mathtt{out}},\ \overset{2}{x},\ \overset{3}{\mathtt{sym}_1},\ \overset{4}{y},\ \overset{5}{\mathtt{sym}_2}\big)$$

Take $G_1:\ x \times x = \mathtt{sym}_1$. Wire x sits at index **2** in $\mathbf{s}$, $\mathtt{sym}_1$ at index **3**:

$$a_1 = b_1 = \big(\overset{1}{0},\ \overset{\mathtt{out}}{0},\ \overset{x}{1},\ \overset{\mathtt{sym}_1}{0},\ \overset{y}{0},\ \overset{\mathtt{sym}_2}{0}\big) \qquad c_1 = \big(\overset{1}{0},\ \overset{\mathtt{out}}{0},\ \overset{x}{0},\ \overset{\mathtt{sym}_1}{1},\ \overset{y}{0},\ \overset{\mathtt{sym}_2}{0}\big)$$

Check it regenerates the gate:

$$\underbrace{(0,0,1,0,0,0)\cdot\mathbf{s}}_{=\,0\cdot 1+0\cdot\mathtt{out}+1\cdot x+0\cdot\mathtt{sym}_1+0\cdot y+0\cdot\mathtt{sym}_2=x} \times \underbrace{(0,0,1,0,0,0)\cdot\mathbf{s}}_{=\,x} = \underbrace{(0,0,0,1,0,0)\cdot\mathbf{s}}_{=\,\mathtt{sym}_1}$$

Additions

$$\mathbf{s} = (\overset{0}{1},\ \overset{1}{\mathbf{out}},\ \overset{2}{x},\ \overset{3}{\mathbf{sym}_1},\ \overset{4}{y},\ \overset{5}{\mathbf{sym}_2})$$

Take G_4 : $\mathbf{sym}_2 + 5 = \mathbf{out}$, read as $(\mathbf{sym}_2 + 5) \times 1 = \mathbf{out}$:

- $\mathbf{sym}_2$ at index **5**
- the constant 5 on index **0** (constant-one) wire
- $\mathbf{out}$ at index **1**

$$a_4 = (5, 0, 0, 0, 0, 1) \quad b_4 = (1, 0, 0, 0, 0, 0) \quad c_4 = (0, 1, 0, 0, 0, 0)\$$$

$$\underbrace{(5 + \mathbf{sym}_2)}_{a_4 \cdot \mathbf{s}} \times \underbrace{1}_{b_4 \cdot \mathbf{s}} = \underbrace{\mathbf{out}}_{c_4 \cdot \mathbf{s}}$$

$$\begin{aligned} a_4 \cdot \mathbf{s} &= 5 \cdot 1 + 0 \cdot \mathbf{out} + 0 \cdot x + 0 \cdot \mathbf{sym}_1 + 0 \cdot y + 1 \cdot \mathbf{sym}_2 \\ &= 5 + \mathbf{sym}_2 \end{aligned}$$

The Full R1CS

$$\mathbf{s} = (\overset{0}{1},\ \overset{1}{\mathtt{out}},\ \overset{2}{x},\ \overset{3}{\mathtt{sym}_1},\ \overset{4}{y},\ \overset{5}{\mathtt{sym}_2})$$

Repeat for the remaining gates and stack everything to get:

$$A = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 5 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \quad B = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad C = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Rows = gates, columns = wires.

The whole system is now in R1CS form $(A\mathbf{s}) \circ (B\mathbf{s}) = (C\mathbf{s})$.

Lets validate

With $\mathbf{s} = (1, 35, 3, 9, 27, 30)$:

$$A\mathbf{s} = \begin{pmatrix} 3 \\ 9 \\ 30 \\ 35 \end{pmatrix}, \qquad B\mathbf{s} = \begin{pmatrix} 3 \\ 3 \\ 1 \\ 1 \end{pmatrix}, \qquad C\mathbf{s} = \begin{pmatrix} 9 \\ 27 \\ 30 \\ 35 \end{pmatrix}$$

Element-wise:

$$3 \cdot 3 = 9, \quad 9 \cdot 3 = 27, \quad 30 \cdot 1 = 30, \quad 35 \cdot 1 = 35$$

Every row holds: **the witness satisfies the R1CS.** ✓

Part III: Preprocessing

Step 3: Convert the Matrices to Polynomials

Where we are

We took a high-level description:

```
def main(private field x) -> field {
    return x**3 + x + 5;
}
```

converted into into two-operand equations then compacted into R1CS with matrices that enforce **every** constraint

```
$ zokrates compile -i program.zok
```

We also generated witnesses for $x = 3$

```
$ zokrates compute-witness --arguments "3"
```

$$\mathbf{s} = (1, 35, 3, 9, 27, 30)$$

```
$ make decode
$ cat out.wtns.decoded.json
{
  ...
  "witness_vector": [
    {
      "wire": 0,
      "value": 1
    },
    {
      "wire": 1,
      "value": 35
    },
    {
      "wire": 2,
      "value": 3
    },
    {
      "wire": 3,
      "value": 9
    },
    {
      "wire": 4,
      "value": 27
    }
  ]
}
```

Where we're heading

Problem: the final product is still element-wise: each row is its own check. We really still have n **separate constraints**.

Idea: Each variable has n values per matrix...

$$A = \begin{pmatrix} 1 & \texttt{out} & \texttt{x} & \texttt{sym}_1 & \texttt{y} & \texttt{sym}_2 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 5 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \quad B = \begin{pmatrix} 1 & \texttt{out} & \texttt{x} & \texttt{sym}_1 & \texttt{y} & \texttt{sym}_2 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad C = \begin{pmatrix} 1 & \texttt{out} & \texttt{x} & \texttt{sym}_1 & \texttt{y} & \texttt{sym}_2 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

...instead of storing n raw values, store one polynomial that generates those values.

Polynomial Interpolation Overview

Forward: a polynomial *makes* points.

Take $p(x) = x^2 - x + 2$ (degree 2).

Plug in anything:

$p(0) = 2,\ p(1) = 2,\ p(2) = 4,\ p(3) = 8,\ \ldots$

→ **infinitely many** points.

Backward: points *pin down* a polynomial.

How many do we need? Just enough to match the degree:

- 2 points → a unique **line** (deg 1)
- 3 points → a unique **parabola** (deg 2)
- $n + 1$ points → a unique degree-(n) polynomial

Interpolation. Through any $n + 1$ points there is **exactly one** polynomial of maximum degree n. The interpolation is mechanical (e.g., Lagrange Interpolation), takes n values at n places and produces the polynomial.

Teaser: Shamir secret sharing hides a secret as a polynomial's constant term, gives out evaluations as "shares"; any $d+1$ shares rebuild the polynomial and reconstructs the secret, fewer reveal *nothing*. Go read about it.

From Matrices to Interpolated Polynomials

Idea: Each variable has n values per matrix. Instead of storing n raw values, store one polynomial that generates those values at predetermined set of inputs say $0, 1, \ldots, n-1$.

$$A = \begin{pmatrix} 1 & \texttt{out} & \texttt{x} & \texttt{sym}_1 & \texttt{y} & \texttt{sym}_2 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 5 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \quad B = \begin{pmatrix} 1 & \texttt{out} & \texttt{x} & \texttt{sym}_1 & \texttt{y} & \texttt{sym}_2 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad C = \begin{pmatrix} 1 & \texttt{out} & \texttt{x} & \texttt{sym}_1 & \texttt{y} & \texttt{sym}_2 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Lets take $\mathbf{x}$ we define a new polynomial $f_{\mathbf{x}}^{A}$ such that:

- $f_{\mathbf{x}}^{A}(0) = 1$
- $f_{\mathbf{x}}^{A}(1) = 0$
- $f_{\mathbf{x}}^{A}(2) = 1$
- $f_{\mathbf{x}}^{A}(3) = 0$

then interpolate into the degree 3 polynomial $f_{\mathbf{x}}^{A}$

Repeat for all variables all 3 matrices

$A = (f_1^A, f_{\texttt{out}}^A, f_{\texttt{x}}^A, f_{\texttt{sym}_1}^A, f_y^A, f_{\texttt{sym}_2}^A) \quad B = (f_1^B, f_{\texttt{out}}^B, f_{\texttt{x}}^B, f_{\texttt{sym}_1}^B, f_y^B, f_{\texttt{sym}_2}^B) \quad C = (f_1^C, f_{\texttt{out}}^C, f_{\texttt{x}}^C, f_{\texttt{sym}_1}^C, f_y^C, f_{\texttt{sym}_2}^C)$

We just turned $n \times m \times 3$ raw values (here $4 \times 6 \times 3 = 24$) into $6 \times 3 = 18$ polynomials.

Now Combine with **s** to match the R1CS form: $(A\mathbf{s}) \circ (B\mathbf{s}) = (C\mathbf{s})$.

Set $L(x) = 1 \cdot f_1^A(x) + \ldots + \texttt{sym}_2 \cdot f_{\texttt{sym}_2}^A(x)$

Set $R(x) = 1 \cdot f_1^B(x) + \ldots + \texttt{sym}_2 \cdot f_{\texttt{sym}_2}^B(x)$

Set $O(x) = 1 \cdot f_1^C(x) + \ldots + \texttt{sym}_2 \cdot f_{\texttt{sym}_2}^C(x)$

The entires R1CS becomes a statement about **three** polynomials:

$$L(x) \cdot R(x) = O(x) \qquad \text{at every constraint point } x \in \{0, 1, 2, 3\}$$

> **AKA Quadratic Arithmetic Program**

Part IV: Cryptography

Going Succinct

Where we are

We started with a high level program, and encoded it into 3 polynomials in QAP form:

$$L(x) \cdot R(x) = O(x)$$

Right now:

- The prover who has the full witness can compute those 3 polynomials
- The verifier can evaluate them at $x \in \{0, 1, 2, 3\}$ and check equality

> Q: What are the problems here?
>
> - No ZK: nothing is really hidden
> - No Succinctness: n checks (1 per constraint)
> - No Soundness

Coming Up: Compress n checks into 1

Trick 1: Certificate of Divisibility

We have $L(x)R(x) = O(x)$ at $x \in \{0, 1, 2, 3\}$ for a correct witness.

Rewrite it as:

$$L(x)R(x) - O(x) = 0 \qquad \Longrightarrow \qquad P(x) := L(x)R(x) - O(x)$$

P is **zero at every constraint point/slot**, so those points $\{0, 1, 2, 3\}$ are **roots**.

> **Factor Theorem:** For a polynomial $f(x)$, a linear term $(x - c)$ is a factor of $f(x)$ if and only if $f(c) = 0$ (meaning c is a root of the polynomial).

In other words, we can rewrite

$$P(x) := L(x)R(x) - O(x)$$

as

$$P(x) = (x - r_1) \cdots (x - r_n)\, h(x) \;=\; t(x)\, h(x)$$

- t is called **target polynomial**
- All constraints hold means $t \mid P$
- Honest prover $\Longrightarrow h$ exists, otherwise does not exist
- h is called certificate of divisibility

Example Flow

- Publish $t(x) = (x - r_1) \cdot \ldots \cdot (x - r_n)$ In our example: $t(x) = (x - 0)(x - 1)(x - 2)(x - 3)$
- Prover uses the witness to compute
  - $P(x) := L(x)R(x) - O(x)$, and
  - $h(x) = \frac{P(x)}{t(x)}$
- Verifier gets $P(x), h(x)$, checks $P(x) = t(x)h(x)$

Q: ZK and soundness aside, how can we check the equality?

Compare the full polynomials, coefficient by coefficient?
Evaluate at the constraint points ?
Evaluate everywhere?

Next up: How can the verifier check $P(x) = t(x)h(x)$ **without** doing n operations?

Trick 2: Test at exactly one point

Q: How can one check $P(x) = t(x)h(x)$ **without** doing n operations?

> **Evaluate both sides at a single random point and compare the two numbers.** If they match there, the polynomials are equal *almost everywhere*.

Equivalently, we test whether

$$P(x) - t(x)h(x)$$

is the **zero polynomial** using one random input. This is the **Schwartz–Zippel** lemma.

Works bec a non-zero degree d poly has at most d roots (inputs that evaluate to 0). Pick a random point from a large enough space, the probability it lands on a root is negligible.

> **We (almost) have succinctness!** Verification no longer scales with the size of the computation/ number of constraints. **Q: Why almost? Proof size!**

Trick 3: Homomorphic Commitments

In cyclic group g, large prime order r, if $(a, b) \leftarrow_{\$} \mathbb{Z}_r^2$ then g^a, g^b are commitments of a and b

Also:

- $g^a = g^c$ means $a = c$
- $g^a = g^b g^c$ means $a = b + c$
- $g^a = (g^b)^c$ means $a = bc$ (Note: c must be known)

> However given g^a, g^b we cannot check if some $T = g^{ab}$ (DDH) or compute g^{ab} (CDH).

Thus if we do $com(f(x)) = com(t(x))com(h(x))$ we also end up with $com(t(x)h(x))$ and we have a way to hide things in the exponents

> **Teaser:** Prover will evaluate polynomials at a random point, without knowing the random point.

Trick 3: Homomorphic Commitments (Trusted Setup)

We perform a setup with 2 phases.

Phase 1: a third party

- generates a random τ
- publishes $g, g^{\tau}, g^{\tau^2}, g^{\tau^3}, \ldots,$
- destroys τ

> AKA: Perpetual Powers of Tau

Phase 2:

Now if my QAP polynomial is $P(x) = 3x^2 + x + 2$, then to commit blind evaluation $P(\tau)$ (i.e., without knowing τ).

- $(g^{\tau^2})^3 g^{\tau} g^2$ using the published values
- which is $g^{3\tau^2} g^{\tau} g^2 = g^{3\tau^2+\tau+2} = g^{P(\tau)}$

```
# Combines both steps (insecure)
$ zokrates setup

Performing setup...
Verification key written to 'verification.key'
Proving key written to 'proving.key'
Setup completed
```

Applied to our proof

Prover has computed: $P(x) := L(x)R(x) - O(x) \qquad h(x) = \frac{P(x)}{t(x)}$

so knows the coefficients of L, R, O, h.

By blind evaluation at the secret point τ prover also computes:

$$g^{L(\tau)}, \quad g^{R(\tau)}, \quad g^{h(\tau)}, g^{P(\tau)}, \ldots \text{etc}$$

Problem

Recall, verifier has $g^{P(\tau)}, g^{t(\tau)}, g^{h(\tau)}$ and wants to check:

$$P(\tau) = t(\tau)h(\tau)$$

but non of the homomorphic properties gave us that ability.

Trick 4: Bilinear Pairings for Multiplication Check

A pairing, AKA bilinear map, is a function $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ between three groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ of prime order p, with generators $g_1 \in \mathbb{G}_1$ and $g_2 \in \mathbb{G}_2$, $g_T \in \mathbb{G}_T$ respectively.

When $\mathbb{G}_1 = \mathbb{G}_2$ the pairing is called symmetric. Otherwise, it is asymmetric.

Two useful properties for cryptography

1. Bilinearity

For all $u \in \mathbb{G}_1$, $v \in \mathbb{G}_2$ and $a, b \in \mathbb{Z}_p$:

$$e(u^a, v^b) = e(u, v)^{ab}$$

also $e(u^a, v) = e(u, v^a) = e(u, v)^a$

2. Non-degeneracy

$$e(g_1, g_2) \neq \mathbb{1}_{G_T}$$

For our purposes

Given $g^{P(\tau)}, g^{t(\tau)}, g^{h(\tau)}$ to check if $P(\tau) = t(\tau)h(\tau)$:

Check
$e(g^{P(\tau)}, g) = e(g^{t(\tau)}, g^{h(\tau)})$
which is same as
$e(g, g)^{P(\tau)} = e(g, g)^{t(\tau)h(\tau)}$

Definition from https://alinush.github.io/pairings

Result

With that we achieved many things:

- **Unaimable point.** The check happens at τ, which the prover *never learns*.
- **Succinctness.** Committed evaluations are single group elements
- **Witness hidden.** $g^{L(s)}$ exposes nothing about L's coefficients/witness (not ZK yet though)
- **Non-interactive.** No live verifier is needed to supply the challenge

We still need to deal with **soundness** and **ZK**.

Part IV: Cryptography

Soundness and ZK

Restrict the Prover

The prover has too much freedom

Problem: verifier cannot look *inside* a group element to tell a genuine evaluation from a fabricated one.

So the prover's task collapses from *"know a witness"* to *"produce handles satisfying one equation"*:

$$\text{pick any } h^* : \quad H = g^{h^*},\ \ P = \left(g^{t(\tau)}\right)^{h^*} \implies e(P, g) = e\left(g^{t(\tau)}, H\right) \checkmark$$

Passes **by construction**. No L, no R, no circuit, no witness.

> "$t \mid P$" alone is insufficient: a cheater can find a polynomial with the right roots while ignoring the real values.

Restrict the prover: divide the work

Don't let the prover build the whole polynomial.: Prover fills in only its **secret** variables, the verifier builds the **public** part itself:

$$L(x) = \underbrace{\sum_{j \in \text{public}} s_j A_j(x)}_{\textbf{verifier builds}} + \underbrace{\sum_{j \in \text{private}} s_j A_j(x)}_{\textbf{prover builds}}$$

Prover can't claim a different output, can't touch the public slots.

> **Note: Further measures are required to sufficiently restrict the prover and guarantee soundness**

Where zero-knowledge comes from

Hiding is not yet ZK. $g^{L(\tau)}$ hides the witnes, but proof is a *deterministic* function of it: same witness → same proof, every time.

Fix: blind it. The prover mixes fresh randomness into its elements

Why that's ZK: with the randomness, a **simulator** knowing *no witness* can produce a transcript with the *same distribution*. Anything the verifier sees, it could have produced itself

Recap: the tricks that lined up

1. Any computation → **only** `+` **and** `×` (arithmetic circuit).
2. Every gate → a **row** of three matrices (**R1CS**).
3. Every **column** → an interpolating polynomial; fold to $L \cdot R = O$ (**QAP**).
4. "n checks" → "t **divides** P" (the target polynomial).
5. Identity check → **one random point** (Schwartz–Zippel) → **succinct**.
6. Commit + pairings + setup + restricted prover → **ZK & soundness**

> No single step is magic. The magic is how they **compose**.

> Real-World Cryptography (book, p.335), Wong, 2021:
>
> Most zk-SNARK schemes build on a proving system, **and a translation of a program to something the proving system can prove**...

continued:

> ... the first part is not too hard to understand, while the **second part requires a graduate course in the subject**.

.. so you should be proud of yourselves!

Thanks!

Tarek Galal

tgalal.com

Resources & Further Readings

- https://medium.com/@VitalikButerin/quadratic-arithmetic-programs-from-zero-to-hero-f6d558cea649
- https://www.cryptologie.net/posts/the-missing-explanation-of-zk-snarks-part-1/
- https://www.cryptologie.net/posts/the-missing-explanation-of-zk-snarks-part-2/
- https://risencrypto.github.io/zkSnarks/
- https://risencrypto.github.io/R1CSQAP/
- https://risencrypto.github.io/Groth16/
- https://securitylab.github.io/cs251-fall22/lectures/lecture14.pdf
- https://securitylab.github.io/cs251-fall22/lectures/lecture15.pdf
- https://zkdl-camp.github.io/
- https://alinush.github.io/pairings
- https://alinush.github.io/groth16
- https://eprint.iacr.org/2016/260.pdf