Introduction to Zero-Knowledge Proofs

Balázs Kőműves
Faulhorn Labs & Codex storage
`bkomuves@gmail.com`

Budapest
2025.09.30.

slides available at:
`https://github.com/bkomuves/slides/`

What is a zero-knowledge proof (ZKP)?

I want to convince you about the truth of a statement, *without telling you* everything to be able to actually check it.

Already sounds like magic?! Well, cryptography *is magical*...

Furthermore, the “proof” can be non-interactive and publicly verifiable (so I’m actually convincing *everybody*, not just you).

Even better,

- this “proof” can be very short (down to a few hundred bytes);
- and you can convince yourself very efficiently (usually much faster than checking the original statement; possibly even in *constant time*, which can be only a few milliseconds!)

The main drawback is that I (the prover) have to work a lot more.

What is a statement?

Mathematically speaking, in our context the statement will be a *relation*

$$\mathcal{R} \;:\; X \times W \;\to\; \texttt{Bool}$$

between a public $x \in X$, called the *instance*, and a secret $w \in W$, called the *witness*.

And I want to convince you (or a third party) that for a given public $x \in X$, I'm in the possession of a corresponding $w \in W$ such that the relation holds:

$$\mathcal{R}(x, w) \;=\; \texttt{true}$$

An example would be that I want to convince you that I know a SHA256 pre-image of a given 32 byte hash (without telling you the pre-image); in this case $x \in X$ would be the target hash, and $w \in W$ can be the pre-image.

Programmable cryptography

In practice, we want the relation $\mathcal{R}$ to be realized by a *computer program*. The public input parameter $x \in X$ is useful because we really want to keep this computer program fixed, while proving several instances of the same statement.

If we allow *any* such $\mathcal{R}$, that's an example of 'programmable cryptography': Some of the inputs of our algorithms are *programs*[1].

We could also relax the previous formalism a bit and allow outputs too; so instead of relations, we can have a function

$$\mathcal{F} \;:\; X \times W \;\to\; Y$$

and ask for a proof of existence of w such that $\mathcal{F}(x, w) = y$ for a given input-output pair $(x, y) \in X \times Y$. It's not hard to see how this is equivalent to the previous version.

[1] how exactly those programs are represented is a very central theme

Sudoku

Another very good example for a statement is a Sudoku puzzle:
I want to convince you that I can solve a particular puzzle.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Here $x \in X$ (the puzzle) is on the left; $w \in W$ (the solution) is on the right; and the $\mathcal{R}$ relation (computer program) checks that the solution is valid, and compatible with the given puzzle.

Proofs vs. arguments vs. zero-knowledge

It's important to remember that zero-knowledge proofs are *NOT* mathematical proofs, just *very convincing* arguments. Technically they should be called "ZK arguments", but that ship already sailed...

Also, 'zero-knowledge' has a specific meaning: It means that you learn *nothing else* than the statement is true (and what follows from that); not even a single bit of information should leak. For many applications, this is not at all important (or satisfied); however, people call those ZKPs too, even though they are not really ZK (again, the ship has sailed).

- ARG = argument (the statement is true)
- ARK = argument of knowledge (...and I also know *why* is it true)
- NARG = non-interactive argument
- NARK = non-interactive argument of knowledge
- NIZK = non-interactive zero-knowledge proof
- SNARK = succinct non-interactive argument of knowledge
  (the proof is small, and usually also fast to verify)
- zk-SNARK = a zero-knowledge SNARK (satisfies the ZK property)

Desired properties

Some desirable properties of such a proof system are:

- the verifier learns nothing about w **(zero-knowledge)**
- the proof is very small **(succinctness)**
- verifying the proof takes very little time **(verifier scalability)**
- creating the proof doesn't take much more time (asymptotically) than evaluating the relation $\mathcal{R}$ **(prover scalability)**
- there is no need of interaction between the prover and the verifier, apart from sending the proof **(non-interactivity)**
- no need for a "trusted setup" **(transparency)**
- safe from future quantum computers **(post-quantum)**

Proof systems with some combination of these properties are often called (zk-)SNARKs or (zk-)STARKs.

Applications

There are many potential applications of ZK proof systems. These can be grouped into 3 big areas: **privacy**, **security**, and **scaling**.

Some possible applications are:

- age verification (both online and offline)
- ...more generally: identity management and authentication
- privacy-friendly business/healthcare/law processes
- scaling up blockchains (this is currently the biggest application)
- decentralized exchanges
- ...more generally: privacy on public blockchains
- electronic voting systems
- more trustworthy machine learning (zkML)
- audited binaries (running the compiler inside ZK; not yet practical)

But how is this possible at all?

Intuition: The colorblind friend example (I can convince a colorblind person that red and green exist as different colors).

Some interactive proofs from mathematics:

- quadratic residue test: $x \equiv w^2 \pmod{N}$
- existence of 3-coloring for a graph
- existence of Hamiltonian cycle in a graph

Other existing, more practical examples:

- digital signatures (you don't leak your private key)
- Schnorr's protocol for the knowledge of discrete logarithm

The big difference from these is that the mainstream ZKPs of today can prove *any statement* (in NP).

Interactive proof for quadratic residue

The prover $\mathcal{P}$ wants to convince the verifier $\mathcal{V}$ that for a given $x \in \mathbb{Z}_N^\times$, they know a square root $w \in \mathbb{Z}_N^\times$ for it, that is, $x \equiv w^2 \pmod{N}$.

The protocol goes as follows:

- the prover samples a uniformly random $r \in \mathbb{Z}_N^\times$, and sends it to the verifier its square $s = r^2$.
- the verifier chooses a random bit $b \in \{0, 1\}$
- if $b = 0$, the prover sends $z := r$; if $b = 1$, they send $z := r \cdot w \pmod{N}$; in other words, $z = r \cdot w^b$.
- the verifier checks that $z^2 = s \cdot x^b \pmod{N}$

This is then repeated, let's say 128 times. The verifier's choice keeps the prover honest: If the prover tries to cheat, either with r or with w, they have an $1/2$ chance to get caught. So after 128 rounds, they have only a chance of 2^{-128} to cheat.

Schnorr's protocol for discrete logarithm

Another example of an interactive ZK proof is Schnorr's protocol. Given a cyclic group $\mathbb{G} = \langle \mathbf{g} \rangle$ of size n (for which the discrete log problem is assumed hard), the prover wants to convince the verifier that they know a (secret) discrete logarithm $w \in \mathbb{Z}_n$ of a public element $x = \mathbf{g}^w \in \mathbb{G}$.

The protocol is the following (only 1 round!):

- the prover generates a random $u \in \mathbb{Z}_n$, and sends $a := \mathbf{g}^u$ to the verifier (this will serve as a commitment to a *blinding factor*);
- the verifier generates a *random challenge* $c \in \mathbb{Z}_n$, sending it back;
- the prover computes $r := u + cw \in \mathbb{Z}_n$ and sends it to the verifier;
- the verifier checks whether $\mathbf{g}^r = \mathbf{g}^{u+cw} = a \cdot x^c \in \mathbb{G}$.

Why does it work?

- soundness: the prover has $1/n$ chance to cheat if they don't know w
- zero-knowledge: technically this is only "honest verifier ZK" (eg. choosing $c = n/2$ reveals the lowest bit of w)

A blueprint for general purpose ZKP

The previous examples were for particular statements. We want a system for *arbitrary* (NP) statements. There are *many* such systems these days, using different approaches. The following is one popular “blueprint”:

1. encode the instance-witness pair $(x, w) \in X \times W$ (and other temporary variables) into numbers (finite field elements)
2. translate the statement into *mathematical equations* over these variables, such that they are a solution if and only of the relation $\mathcal{R}(x, w)$ was satisfied
3. pack the variables into polynomials, and compress the (large number of) equations into only a few equations over these polynomials
4. use an IOP (“interactive oracle proof”) to prove that you know a solution of these polynomial equations
5. replace the idealized “oracle” (which doesn't exist in the real world) in the IOP with a polynomial commitment scheme (PCS)
6. remove the interaction using the Fiat-Shamir heuristic, to make it non-interactive

Converting Sudoku into equations, page I.

Let's fix a prime field $\mathbb{F}_p$ (with $p \gg 9$). We can encode a Sudoku solution as $9 \times 9 = 81$ field elements $s_{ij} \in \mathbb{F}$, by the natural mapping $\mathrm{mod}_p : \mathbb{N} \to \mathbb{F}_p$. Similarly, we can encode the puzzle into $x_{ij} \in \mathbb{F}$, simply by mapping the empty cells to 0.

We then want equations for the following rules:

- if $x_{ij} \neq 0$, then $x_{ij} = s_{ij}$ (the solutions solves the given puzzle)
- $s_{ij} \in \{1, 2, \ldots, 9\}$ (all cells contains numbers between 1 and 9)
- all rows s_{i*} contain distinct numbers
- all columns s_{*j} contain distinct numbers
- all little 3×3 squares contain distinct numbers

Such equations are actually quite simple to write down.

Sudoku equations, page II.

Compatibility equations:	$\forall\, 1 \leq i,j \leq 9:$	$x_{ij}(s_{ij} - x_{ij}) = 0$
Range equations:	$\forall\, 1 \leq i,j \leq 9:$	$\prod_{n=1}^{9}(s_{ij} - n) = 0$
Row "equations":	$\forall\, 1 \leq i,j,k \leq 9,\ \ j < k:$	$s_{ij} \neq s_{ik}$

The equations for the columns and the little 3×3 squares are then essentially the same as for the rows.

Oops. *Houston, we have a problem!* Unfortunately $s_{ij} \neq s_{ik}$ is NOT an equation. We need to reformulate it somehow. The first step is a simple rearrangement: $s_{ij} - s_{ik} \neq 0$.

Now we probably want to abstract $a \neq 0$ into a reusable pattern. Let's call it 'forceNonZero(a)'. So how to do that? The trick is to recognize that if $a \in \mathbb{F}$ is not zero, then it has an inverse $b = 1/a$! Then we can simply write down the equation $a \cdot b = 1$. Clearly, whatever is the value of b, if that equation holds, then surely $a \neq 0$ (as we are in a field).

This technique introduces a *new witness variable* $b \in \mathbb{F}$.

Interlude: What exactly is the witness?

So at the beginning, I said that we have this fixed relation $\mathcal{R} : X \times W \to \texttt{Bool}$, and $w \in W$ is the witness.

Which is both true and somewhat untrue. There are two different (but very closely related) meanings of the word "witness" in the ZKP context. Maybe best to use an example:

In the SHA256 pre-image example, I said that $x \in X$ is the target hash, and $w \in W$ is the pre-image. And indeed it's easy to write a program which implements the relation

$$\mathcal{R}(x, w) = \begin{cases} \texttt{true}, & \text{if } \mathsf{SHA256}(w) = x \\ \texttt{false} & \text{otherwise} \end{cases}$$

But in practice, that program will have a lot of temporary variables, and when translating it into equations, we will need those values too. Indeed, they become part of the witness.

Nondeterminstic advice

This leads to an interesting phenomenon, pretty unique to ZKPs: When implementing the relation $\mathcal{R}(x, w)$ as a computer program, we can rely on external, "god-given" inputs - as long as we *check* that they are correct.

And for some problems, checking can be rather more efficient than computing! Of course the prover still needs to compute these "external" inputs, but computing on a CPU is normally orders of magnitudes faster than computing "inside the proof", so this still makes a lot of sense.

Consider the following example: We want to implement a *function* 'isNonZero(x)' which returns 1 if $x \neq 0$, and 0 if $x = 0$. This can be translated to the following equations:

$$\begin{aligned} x \cdot y &= z \\ x \cdot (1 - z) &= 0 \\ \text{isNonZero}(x) &= z \end{aligned}$$

It's easy to check that this works. The external advice is $y = 1/x$, which the prover has to compute *outside the proof*. Note that the equations only contain multiplications, no division!

Another interlude: Equation degrees

The “range equations” in the Sudoku example were kind of an outlier compared to the rest:

$$\prod_{n=1}^{9}(s-n) \;=\; (s-1)(s-2)(s-3)\cdots(s-9) \;=\; 0$$

These equations have degree 9, while all the other equations had degree 2. Imagine if this was a 25×25 hard-core Sudoku, it would have degree 25! And so on...

Now in general, we don’t like high degree equations, because they are expensive (for the prover). Fortunately, it’s easy to convert such a high-degree equation into a set of low-degree equations, simply by introducing more witness variables:

$$\begin{array}{rcl} t_2 & = & (s-1)(s-2) \\ t_3 & = & t_2(s-3) \\ t_4 & = & t_3(s-4) \end{array} \qquad \cdots \qquad \begin{array}{rcl} t_7 & = & t_6(s-7) \\ t_8 & = & t_7(s-8) \\ 0 & = & t_8(s-9) \end{array}$$

But as everything in this space, this is a tradeoff (here the degree against the witness size).

Uniformization

So we translated Sudoku into

$$\underbrace{9^2}_{\text{comp.}} + \underbrace{9^2 \times 8}_{\text{range}} + 3 \times \underbrace{\left[\binom{9}{2} \times 9\right]}_{\text{row}} = 1701$$

quadratic equations over

$$\underbrace{9^2}_{x} + \underbrace{9^2}_{s} + \underbrace{9^2 \times 7}_{t} + 3 \times \underbrace{\left[\binom{9}{2} \times 9\right]}_{a^{-1}} = 1701$$

witness variables.

Now the verifier would need to check all of it (with whatever magic math) - that obviously won't scale! So we need to "uniformize" them somehow, and then combine them into a single (or a few) "big equations", which then the verifier can check.

Arithmetization strategies

The combination of translating the problem into equations and then packing those equation in a few “big” ones is loosely called *arithmetization*.

There are many different approaches to do that, for example:

- arithmetic circuit + indicator functions
- rank-1 constraint systems (R1CS)
- Plonk (and more generally, Plonkish)
- algebraic intermediate representation (AIR)
- lookup tables
- combinations of the above
- Binius / Binius64
- etc

R1CS (rank-1 constraint systems)

Pack all the public input $\underline{x} \in X$, the witness $\underline{w} \in W$, and all the temporary variables $\underline{t}$ (these are often considered part of $\underline{w}$ in practice) into a single big vector z:

$$z := (\, 1 \,|\, \underline{x} \,|\, \underline{w} \,|\, \underline{t} \,)$$

Then R1CS is simply a set of quadratic equations of the form

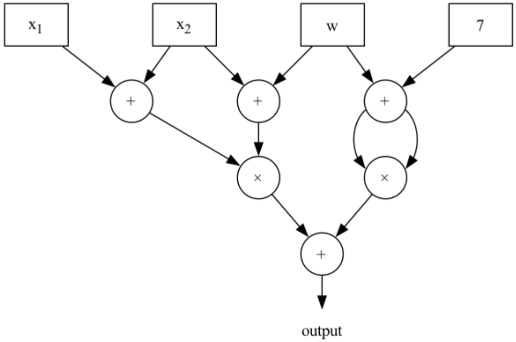
$$\forall i. \qquad \Big(\sum_j A_{ij} z_j \Big) \cdot \Big(\sum_j B_{ij} z_j \Big) \; + \; \sum_j C_{ij} z_j \; = \; 0$$

with A, B, C three fixed, public, sparse coefficient matrices.

This is a useful form, because having only 1 multiplication per equation makes it amenable to some mathematical tricks (for example elliptic curve pairings). This is the output of the popular circuit DSL `circom`, and the input of the popular proof system Groth16. It's very easy to see how to convert the Sudoku equations to this form.

Arithmetic circuits

An 'arithmetic circuit' is made up from "arithmetic gates" (for example: addition and multiplication) and "wires" between them:



This circuit implements the following simple function:

$$f(x_1, x_2\,;\,w) := (x_1 + x_2)(x_2 + w) + (w + 7)^2$$

Our Sudoku example can be converted into an arithmetic circuit too.

Encoding arithmetic circuits

The following is one possible, relatively simple way to encode an arithmetic circuit as equations.

First, pack all the values on the wires into a big vector z:

$$z \ := \ (\,\underline{x}\,|\,\underline{w}\,|\,\underline{t}\,) \ = \ (\,x_1,\, x_2,\, 7 \mid w \mid x_1 + x_2,\, x_2 + w,\, \ldots\,)$$

Define the indicator functions add_{ijk} and mul_{ijk} to be 1 if there is an addition (resp. multiplicaiton) gate with inputs i, j and output k, and 0 otherwise. These are fixed and public, they simply encode the circuit.

Then the circuit is equivalent to the set of equations:

$$\forall k. \quad \sum_{i,j} \mathsf{add}_{ijk}(z_k - z_i - z_j) + \mathsf{mul}_{ijk}(z_k - z_i z_j) = 0$$

While this doesn't seem to be immediately useful, there are further tricks to apply to make this into a single equation which can be checked efficiently. But that would be a too big detour here.

Toy Plonk

This is a different encoding of an arithmetic circuit. Create an $N \times 3$ matrix $(X|Y|Z)$, with the rows corresponding to the N gates of the circuit, and X_i, Y_i being the two inputs, while Z_i being the output of the gate (note: these contain the same values as the (x, w) pair, but with some repetitions).

Define a (public, fixed) *selector vector* S, so that $S_i = 0$ if the i-th gate is an addition gate, and $S_i = 1$ if it's a multiplication gate.

Then we have the so-called *gate equations*:

$$\forall i. \quad (1 - S_i) \cdot (Z_i - X_i - Y_i) + S_i \cdot (Z_i - X_i Y_i) \; = \; 0;$$

but we also have *wiring constraints*: For example if the output of j-th gate is the left input of the k-th gate, then we need to enforce $Z_j = X_k$; and so on for all wires.

This is a bit tricky, but it will be proven later via a so-called 'permutation argument'.

Actual Plonk, and Plonkish arithmetization

In the “real” Plonk, the gate equation is a bit more general:

Instead of having a single selector column containing only 0s and 1s, we have five: Q_C, Q_L, Q_R, Q_M and Q_O, and they can contain arbitrary values in $\mathbb{F}$. The gate equations then have the form:

$$Q_C + Q_L \cdot X + Q_R \cdot Y + Q_M \cdot X \cdot Y + Q_O \cdot Z \;=\; 0$$

It's easy to see how to simulate the previous “toy Plonk” with this more general one.

More generally, in “Plonkish” arithmetization, you can have an arbitrary number of “selector columns” (instead of 1 or 5), arbitrary number of witness columns (instead of 3), and arbitrary gate equations (possibly several different ones), and even extra features like “lookup gates” (these can prove that a value, or tuple of values in the same row, are included in some fixed, public list of possibilities).

AIR (Algebraic Intermediate Representation)

AIR looks very different from the above arithmetizations: Instead of encoding a circuit, we encode a *state machine*.

Consider a state transition function $\texttt{step} : \mathbb{F}^m \to \mathbb{F}^m$. We want to prove that starting from an initial state $S_0 \in \mathbb{F}^m$, executing the state transition $S_{i+1} := \texttt{step}(S_i)$ many times, after N steps we end up in the final state S_N. Usually (some parts of) the initial and final state form the public input x, and the rest is the witness w.

To do this, first we translate the step function into a set of equations over the variables in two neighboring rows S and S' (this can be seen as a "mini-circuit"). We also run the state machine and collect all the states into an $(N+1) \times m$ matrix A, so that the i-th row is the state S_i after i steps.

We now have a set of uniform equations (one for each row), which we will be able to do with similar methods as the gate equations of Plonk. At this point maybe it's less surprising to learn that AIR can be also combined with Plonk(ish) to have even more expressive power.

AIR example: Squared Fibonacci

A simple example for AIR is the squared Fibonacci series:

$$F_0 = A \qquad F_1 = B \qquad F_{k+2} = F_k^2 + F_{k+1}^2$$

Suppose we want to prove that for a public $x = A$ we know a secret $w = B$ such that the 1000th number F_{1000} equals[2] a given value $y = C$.

We can have the state $S_k = (U_k, V_k)$ consist of two consecutive values $S_k := (F_k, F_{k+1})$ (a single column would be enough, but it's a bit more interesting this way). The step function is then:

$$\texttt{step} : (u\,,\,v) \longmapsto (v\,,\,u^2 + v^2)$$

It's easy to write down the transition equations:

$$\begin{aligned} U_{k+1} &= V_k \\ V_{k+1} &= U_k^2 + V_k^2 \end{aligned}$$

Then we can simply apply the general machinery (which will come at some point, I promise!).

[2] well, modulo p, as all this lives in a prime field $\mathbb{F}_p$...

zkVMs (zero-knowledge virtual machines)

Manually creating all these equations or circuits can be very effective, but it's also a huge amount of work and *VERY* error-prone. Worse, if you accidentally miss an equation, you may never notice! Standard software testing approaches do not apply here. And by definition ZK proofs are always security critical! A missing equation means a malicious prover could cheat, and you won't even realize it...

What if we could write our statements in a more usual programming language instead? Here comes a big realization: A CPU (or even a full computer) is just a huge state machine!

So instead of developing a new circuit for each problem, we could just do it once for a virtual CPU, and then have our statement, encoded as machine code, as part of the input $x \in X$. Then use (a generalized version of) the AIR (state machine) approach to prove it.

This solution is called a 'zkVM', and is very popular. The main drawback is significant overhead compared to optimized problem-specific circuits (kind of like a real CPU vs. an ASIC).

Recursive proofs

Another mindbending idea is to let the statement $\mathcal{R}(x, w)$ to prove to be the *verification algorithm of another ZK proof*. This is usually called "recursive proofs" or "proof composition".

It allows both IVC[3] (incrementally verifiable computation) and "infinite" proof compression. For the latter you can organize the proofs into a tree similar to a Merkle tree: The original statements are on the leaves, and then each node proves their children, until there is a single final proof for all the statements[4].

Another typical use-case is to combine a fast proof system generating large proofs with a final compression step, which is slower but generates a smaller proof. As the (verification algorihtm of the) middle proof is typically much smaller than the original problem, this makes sense; you essentially get the best from both.

[3]other techniques like folding can be also used to achieve similar results

[4]it can be still tricky to deal with all the public inputs...

Polynomials

So far we ignored the elephant in the room: How to actually *prove* anything?

The key component is *polynomials*. Univariate[5] polynomials (over finite fields) have the following two useful properties:

- **interpolation**: A polynomial of degree d can encode an arbitrary piece of data $\underline{w} \in \mathbb{F}^{d+1}$
- **roots**: A polynomial of degree d has at most d roots.

The second point has a very interesting consequence: If you want to convince yourself whether a polynomial $p(x)$ is constant zero or not, then assuming that $|\mathbb{F}| \gg d$, it's enough to check that $p(\zeta) = 0$ at a single (or few) random location(s) $\zeta \in \mathbb{F}$.

[5]Quite often, multilinear polynomials are used instead of univariate ones. Looks quite different at the beginning, but works rather similarly at the end of the day.

Schwartz-Zippel lemma

This is formally called the Schwartz-Zippel lemma. Well in fact it's quite a bit more general; and also works for multivariate polynomials:

Lemma. Let $g : \mathbb{F}^m \to \mathbb{F}$ be a (not constant zero!) multivariate polynomial of total degree d. For any subset $S \subset \mathbb{F}$, the probability of g vanishing on $\underline{x} \in S^m \subset \mathbb{F}^m$ (chosen uniformly random) is at most $d/|S|$.

In practice we usually have something like $|\mathbb{F}| \geq 2^{124}$ and $d \leq 2^{24}$ and $S = \mathbb{F}$; so this means that you can check whether a polynomial is constant zero with just *a single evaluation*, with a false positive rate of at most 2^{-100}.

A trivial corollary is that you can also check whether two polynomials are the same: Just subtract them and check for the result being constant zero.

An IOP for vanishing on a subset

Two polynomials being the same is not a very interesting property in itself. However, we can build more interesting things out of this “basic lego block”.

One of the simplest and most useful is claiming that a polynomial $P(x)$ is vanishing on a given *subset* $S \subset \mathbb{F}$, that is, if $s \in S$ then $P(s) = 0$.

The trick is the following: If $P(s) = 0$, then $P(x)$ is divisible by the linear polynomial $(x - s)$. Iterating this, we can form the *quotient polynomial*

$$Q(x) := \frac{P(x)}{Z_S(x)} = \frac{P(x)}{\prod_{s \in S}(x - s)}$$

Multiplying back with the *zero polynomial* $Z_S(x)$, we have the equation

$$P(x) = Q(x) \cdot \prod_{s \in S}(x - s)$$

Now this is more interesting: The prover “provides” P and Q, and the verifier can compute the third term itself. So it’s enough to check for a single (randomly chosen) $\zeta \in \mathbb{F}$ that

$$P(\zeta) = Q(\zeta) \cdot \prod_{s \in S}(\zeta - s)$$

Interlude: Some theory. Interactive proofs

The well-known complexity class NP can be viewed as a simple proof system: Given a problem in NP, the prover computes the solution, and the (deterministic) verifier can check it in polynomial time.

Interactive proof systems generalize this by allowing communication between the prover and the (non-deterministic) verifier. The properties we expect are:

- **Completeness**: If the statement is true, the prover can convince the verifier of this fact;
- **Soundness**: If the statement is false, no prover can convince the verifier that it is true (except with a very small probability).

The class of problems provable by interactive proof systems, IP, is believed to be strictly bigger[6] than NP. The intuitive reason behind this is that the verifier can issue *random challenges* to the prover.

Systems in which the verifier's messages are all sampled randomly from known distributions are called *public coin protocols*; this is an important special class, as these can be turned into non-interactive proofs via the Fiat-Shamir heuristic[7]

[6] it is known than IP=PSPACE

[7] note that there are some *very subtle* caveats here...

Interactive Oracle Proofs

Interactive Oracle Proofs (IOPs) generalize interactive proofs by replacing the prover's messages with *oracles*: idealized black-box computer programs which the verifier can query at some inputs of their choice (in unit time!).

IOPs[8] are even more expressive then IPs, capturing the complexity class NEXPTIME (= solvable by a non-deterministic Turing machine in exponential time), which is believed to be strictly bigger than PSPACE=IP.

An example of such an oracle is a black box which can evaluate a fixed polynomial at different points. Of course the prover could just send the whole polynomial, instead of the oracle, but that has 3 problems:

- the polynomial could be very big;
- the verifier has to spend a lot of time evaluating the polynomial;
- the prover may want to keep the polynomial secret.

[8]and also PCPs, *probabilistically checkable proofs*

Polynomial commitment schemes (PCS)

The idea is to replace polynomial oracles (an idealized mathematical concept) in IOPs by *polynomial commitments* (a cryptographic construction):

Given a degree $d \leq D$ univariate polynomial $P \in \mathbb{F}_q[x]$ over a large finite field $\mathbb{F}_q$ (typically $D \ll q$), the prover first wants to *commit* to $P(x)$ by producing a short *commitment* (or fingerprint) $\mathtt{com}(P)$; then the verifier can ask the prover to evaluate P at some given points x_i and receive the results $y_i = P(x_i)$ together with some *evaluation proofs* π_i, and check that these are consistent with the commitment $\mathtt{com}(P)$; that is, the prover does not cheat (at least with a very high probability).

Examples of such schemes are KZG, Bulletproofs, FRI, DARK, Dory, etc. These all have different tradeoffs like commitment size, proof size, verification speed, cryptographic assumptions...

Polynomial IOP + polynomial commitment = ZK proof

Given an polynomial IOP, we can replace the polynomial oracles by a polynomial commitment scheme, and get something which looks like an IP.[9]

Here is how to do it: Instead of sending over a polynomial oracle for a polynomial $P(x)$, the prover first *commits* to P; then the verifier, instead of querying the oracle, asks for an evaluation proof. The important thing is that the commitments and evaluation proofs are very small (and fast to verify) compared to the size of the polynomial (and the speed of evaluating it).

Note on zero-knowledge: The PCS can be fully ZK or not, but already by virtue of the commitment being very small, it hides a lot of information. Full ZK is often achieved in the protocols by adding so called "blinding terms" to the polynomials in question.

[9] This may look like a contradiction (as IOPs are more powerful than IPs), but polynomial commitment schemes are all based on some computational hardness assumptions, ie. you introduce cryptography into the system.

Non-interactive proofs via Fiat-Shamir

If in an interactive protocol, all the messages from the verifier are sampled randomly from known distributions (which is usually the uniform distribution), that's called a *public coin protocol*.

In that case, there is a well-known method, called the *Fiat-Shamir heuristic*, to transform it into a non-interactive proof (under certain further, *very subtle* assumptions).

The idea is simple: Every time the verifier would choose a random message (usually one or more numbers), the prover simulates it by computing a number it cannot meaningfully bias to its advantage. To do this, a cryptographic hash function can be used, hashing *all previous communication* (and also all choices the prover made), and then mapping the resulting hash to the desired domain.

Note that there are some *extremely subtle* caveats here. This transformation is a very common source of exploitable vulnerabilities!

Proving Fibonacci, part I.

Now we have (almost) enough tools to actually prove the Fibonacci example!

Recall that we had 2 column vectors U and V. Interpolate them into 2 polynomials $\mathcal{U}(x)$ and $\mathcal{V}(x)$, such that

$$\mathcal{U}(\omega^i) = U_i \qquad \text{and } \mathcal{V}(\omega^i) = V_i$$

where $\omega \in \mathbb{F}^\times$ is a generator of a subgroup $H \subset \mathbb{F}^\times$ of size $|H| = 1024 > 1000$:

$$H \;=\; \{\, 1, \omega, \omega^2, \omega^3, \dots, \omega^{1023} \,\}$$

The fact that H is a *subgroup* is important for three reasons:

- interpolation is fast (via FFT);
- it's easy to "shift" to the "next row": If $\mathcal{U}(x) = U_i$ then $\mathcal{U}(\omega x) = U_{i+1}$;
- and finally the zero polynomial $Z_H(x) = x^{1024} - 1$ is easy to compute.

Proving Fibonacci, part II.

Recall that we had two sets of equations: For all $0 \leq k < 1023$

$$U_{k+1} = V_k$$
$$V_{k+1} = U_k^2 + V_k^2$$

It's easy to rewrite these to equations to be about the *polynomials* $\mathcal{U}(x)$ and $\mathcal{V}(x)$: For all $x \in H$ (that is, $x = \omega^i$), we require that

$$\mathcal{U}(\omega \cdot x) = \mathcal{V}(x)$$
$$\mathcal{V}(\omega \cdot x) = \mathcal{U}(x)^2 + \mathcal{V}(x)^2$$

However, there is a small problem here: These equations do NOT hold for $x = \omega^{1023}$. That's because $\omega \cdot x = \omega^{1024} = 1$: Our column vectors became circular!

Proving Fibonacci, part III.

Fortunately, this is not hard to fix: Simply rewrite[10] to $\forall x \in H$

$$0 = \left[1 - \mathcal{L}_{1023}(x)\right] \cdot \left[\mathcal{U}(\omega \cdot x) - \mathcal{V}(x)\right]$$
$$0 = \left[1 - \mathcal{L}_{1023}(x)\right] \cdot \left[\mathcal{V}(\omega \cdot x) - \mathcal{U}(x)^2 - \mathcal{V}(x)^2\right]$$

where $\mathcal{L}_k(x)$ are the Lagrange basis polynomials:

$$\mathcal{L}_k(x) = \begin{cases} 1 & \text{if} \quad x = \omega^k \\ 0 & \text{if} \quad x = \omega^i \text{ but } i \neq k \\ ? & \text{if} \quad x \notin H \end{cases}$$

These are easy to compute:

$$\mathcal{L}_k(x) = \frac{1}{1024} \sum_{i=0}^{1023} \omega^{-ki} x^i = \frac{1}{1024} \prod_{m=0}^{9} \left(1 + (\omega^{-k} x)^{2^m}\right)$$

In particular, the verifier can compute them themselves.

[10] an even simpler version would be $0 = (x - \omega^{1023}) \cdot [\mathcal{U}(\omega \cdot x) - \mathcal{V}(x)]$

Proving Fibonacci, part IV.

We should also have the boundary conditions: $\mathcal{U}(\omega^0) = A$ and $\mathcal{U}(\omega^{1000}) = C$. There are different ways to deal with these, but the simplest is probably to rewrite them in the same form:

$$\begin{aligned} 0 &= \mathcal{L}_0(x) \qquad \cdot [\mathcal{U}(x) - A\,] \\ 0 &= \mathcal{L}_{1000}(x) \cdot [\mathcal{U}(x) - C\,] \end{aligned}$$

So now we have four equations, but they are only valid for $x \in H$, that is, $x = \omega^i$ for some i.

We can now use the earlier "IOP for vanishing on a subset" to rewrite them to something which holds for all $x \in \mathbb{F}$, and then check at a single randomly chosen $\zeta \in \mathbb{F}$. But before that, let's combine them into a single big equation.

This is called the "random linear combination trick", and is very useful. The idea is very simple: Just choose a random $\alpha \in \mathbb{F}^{\times}$ and combine with powers of it.

Proving Fibonacci, part V.

So we will have a single big equation: for all $x \in H$,

$$\begin{aligned}
0 = \mathcal{P}(x) := 1 \ &\times \ \left(x - \omega^{1023}\right) \cdot \left[\mathcal{U}(\omega \cdot x) - \mathcal{V}(x)\right] \\
+ \ \alpha \ &\times \ \left(x - \omega^{1023}\right) \cdot \left[\mathcal{V}(\omega \cdot x) - \mathcal{U}(x)^2 - \mathcal{V}(x)^2\right] \\
+ \alpha^2 \ &\times \ \quad \mathcal{L}_0(x) \cdot \left[\mathcal{U}(x) - A\right] \\
+ \alpha^3 \ &\times \ \mathcal{L}_{1000}(x) \cdot \left[\mathcal{U}(x) - C\right]
\end{aligned}$$

If $\alpha \in \mathbb{F}$ is chosen randomly, and $\mathbb{F}$ is big enough, there is a only a very low chance that this holds but the individual equations don't.

Now we can form the quotient polynomial $\mathcal{Q}(x) := \mathcal{P}(x)/Z_H(x)$, and have a final equation which now holds for all $x \in \mathbb{F}$:

$$\mathcal{P}(x) \ = \ \mathcal{Q}(x) \cdot Z_H(x) = \mathcal{Q}(x) \cdot (x^{1024} - 1)$$

This can be tested at a single random location $x \mapsto \zeta$. By the Schwartz-Zippel lemma, the probability of a false positive is approximately $p \approx (3 + 2048)/|\mathbb{F}|$. With say $|\mathbb{F}| > 2^{128}$ this is negligible.

Proving Fibonacci, part VI.

So the final “Fibonacci protocol” will look something like this:

1. the prover computes the 1024×2 matrix $(U|V)$, and interpolates the corresponding polynomials $\mathcal{U}(x)$ and $\mathcal{V}(x)$
2. the prover then *commits* to these 2 polynomials
3. the verifier choses a random $\alpha \in \mathbb{F}^{\times}$
4. the prover computes the combined polynomial $\mathcal{P}(x)$ and the quotient polynomial $\mathcal{Q}(x)$; and commits to the latter
5. the verifier choses a random $\zeta \in \mathbb{F}$, and asks the prover for the values $\mathcal{U}(\zeta)$, $\mathcal{V}(\zeta)$, $\mathcal{U}(\omega\zeta)$, $\mathcal{V}(\omega\zeta)$ and $\mathcal{Q}(\zeta)$
6. the verifier computes $\mathcal{P}(\zeta)$ (note that they can compute $\mathcal{L}_k(\zeta)$ and also $Z_H(\zeta) = \zeta^{1024} - 1$ themselves)
7. if $\mathcal{P}(\zeta) = \mathcal{Q}(\zeta) \cdot Z_H(\zeta)$, the verifier accepts, otherwise rejects

Commitment schemes

Commitment schemes lets a prover first *commit* to some data, and later reveal some projection of it, together with a *proof* showing consistency with the commitment.

structure	construction	commitment	reveal	proof	assum.
$x \in \mathbb{Z}_n$	Pedersen	$\mathbf{g}^x \in \mathbb{G}$	x	n/a	dlog
$\underline{x} \in \mathbb{Z}_n^k$	Ped. vector	$\prod_i \mathbf{g_i}^{x_i} \in \mathbb{G}$	$\underline{x}$	n/a	dlog
bytes	hash	$H(\texttt{data})$	data	n/a	hash
vector	Merkle tree	Merkle root	a_i	Merkle path	hash
polynom.	KZG	$\texttt{com}(P) \in \mathbb{G}$	$P(x_0)$	$\texttt{com}(Q) \in \mathbb{G}$	pairing
polynom.	FRI	Merkle root	$P(x_0)$	FRI for Q	hash+RS
IPA	Bulletproofs	$\prod_i \mathbf{g_i}^{x_i} \in \mathbb{G}$	$\langle x, y \rangle$	div. & conquer	dlog

A commitment scheme can have the zero-knowledge property or not. ZK is usually easy to achieve by adding *blinding factors*. For example $\mathbf{h}^r \mathbf{g}^x \in \mathbb{G}$ or $H(r \| x)$ where r is random and approx. 256 bits.

Merkle trees

Merkle trees are a very well-known scheme to commit to a vector of some data (eg. strings or numbers) a_i, and then selectively reveal some a_i (together with proofs), without revealing the rest.

The idea is quite simple:

Fix some cryptographic hash function H (eg. SHA256 or Keccak). Assume for brevity that the length of our vector $\{a_i\}$ is 2^m.

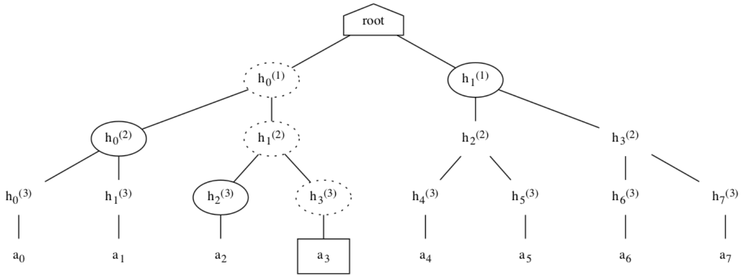
First compute $h_i^{(m)} := H(a_i)$. Then define

$$h_j^{(k)} := H\left(h_{2j}^{(k+1)} \parallel h_{2j+1}^{(k+1)} \right)$$

for $k \in \{0, \ldots, m-1\}$ and $j \in \{0, \ldots, 2^k - 1\}$. The top of this pyramid of hashes, $h_0^{(0)}$, is called the *Merkle root*, and that will be our commitment (note that this has constant size, say 256 bits!).

Merkle proofs

After we sent the Merkle root to the other party, we are committed to the values $a_0, \ldots, a_{2^m-1}$, as changing any of them will change the Merkle root unpredictably.



In the picture, we want to reveal a_3. To achieve this, we send them the values a_3, $h_2^{(3)}$, $h_0^{(2)}$, $h_1^{(1)}$ (the value in question plus m hashes, called the *Merkle path*). From these, they can compute $h_3^{(3)}$, $h_1^{(2)}$, $h_0^{(1)}$ and finally the Merkle root. But we already commited to the root, so they can check if that matches; if yes, we didn't cheat (with high probability).

Polynomial commitment schemes (PCS)

Recall that in a polynomial commitment scheme, the object we commit to is a *polynomial*[11] f (of bounded degree $\deg(f) \le D$), and then later we want reveal *values* $y_i = f(x_i)$ at given locations $x_i \in \mathbb{F}$.

More formally, we can say that a (univariate) PCS is three algorithms:

$$\begin{aligned}
\texttt{commit} &: \mathbb{F}^{\le D}[x] && \to \mathbb{G} \\
\texttt{eval} &: \mathbb{F}^{\le D}[x] \times \mathbb{F} && \to \mathbb{F} \times \Pi \\
\texttt{check} &: \mathbb{G} \times \mathbb{F} \times (\mathbb{F} \times \Pi) && \to \{\texttt{true}, \texttt{false}\}
\end{aligned}$$

which satisfy the following properties:

- $\texttt{eval}(f, x) = \big(f(x)\,,\,\pi\big)$
- $\texttt{check}\big(\texttt{commit}(f), x, \texttt{eval}(f, x)\big) = \texttt{true}$
- it's not feasible to find a different polinomial $f' \in \mathbb{F}[x]$, such that $\texttt{commit}(f) = \texttt{commit}(f')$
- it's not feasible to find a different $y' \neq f(x) \in \mathbb{F}$ and $\pi' \in \Pi$, such that $\texttt{check}(\texttt{com}(f), x, y', \pi') = \texttt{true}$

[11] usually either univariate or multilinear; we concentrate on the first here

KZG polynomial commitment

Suppose we are given two copies $\mathbb{G}_1 \cong \mathbb{G}_2$ of a cyclic group of prime order q (in which the discrete logarithm problem is hard), with generators $\mathbf{g}_i \in \mathbb{G}_i$, and a bilinear map, or "pairing" $\langle -, - \rangle : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_t$ into a third group[12]. For $u \in \mathbb{F}_q$, define by $[u]_i := \hat{u}\mathbf{g}_i \in \mathbb{G}_i$ where $\hat{u} \in \mathbb{Z}_p$ is u but interpreted as an integer.

Suppose furthermore somebody gave us the following data, where $\tau \in \mathbb{F}_q$ is a secret "toxic waste" *unknown to both the prover and the verifier:*

$$[1]_1 = \mathbf{g}_1,\ [\tau]_1,\ [\tau^2]_1,\ [\tau^3]_1,\ \ldots,\ [\tau^N]_1 \quad \text{and} \quad [1]_2 = \mathbf{g}_2,\ [\tau]_2$$

This is called a "trusted setup", which is doable in practice with many participants, even if almost all of them are dishonest.

Now given a polynomial $P(x) = \sum_k A_k x^k$ of degree $d \le N$, the prover can compute the commitment

$$\mathtt{com}(P) := [P(\tau)]_1 = \left[\sum_{k=0}^{d} A_k \tau^k\right]_1 = \sum_{k=0}^{d} A_k [\tau^k]_1 \quad \in \mathbb{G}_1$$

[12] Normally $\mathbb{G}_{1,2}$ are *elliptic curves* and $\mathbb{G}_t$ is a multiplicative subgroup of a field extension of the base field of the curves

KZG evaluation proof

Now the verifier asks for the value $P(x_0) = y_0$ at some point $x_0 \in \mathbb{F}_q$. How can the prover convince it that there is no cheating?

First, $P(x_0) = y_0$ is equivalent to $P(x) - y_0$ having a root at x_0, which is further equivalent to the quotient $Q(x) := (P(x) - y_0)/(x - x_0)$ being a polynomial. The evaluation proof will be $\mathtt{com}(Q) = [Q(\tau)]_1 \in \mathbb{G}_1$.

The verifier wants to check $Q(x)(x - x_0) = P(x) - y_0$, but it doesn't have access to P or Q. However it's almost as good to check $\mathtt{com}(\text{LHS}) = \mathtt{com}(\text{RHS})$ instead. We can easily compute:

$$\mathtt{com}(\text{LHS}) = \mathtt{com}\big(Q(x)(x - x_0)\big) = (\tau - x_0) \cdot \mathtt{com}(Q)$$
$$\mathtt{com}(\text{RHS}) = \mathtt{com}\big(\quad P(x) - y_0 \quad\big) = \mathtt{com}(P) - y_0 \cdot \mathbf{g}_1$$

Now the RHS is fine, but the LHS is not. So instead, it will check that

$$\big\langle \mathtt{com}(Q)\ ,\ [\tau]_2 - x_0\mathbf{g}_2 \big\rangle = \big\langle \mathtt{com}(P) - y_0\mathbf{g}_1\ ,\ \mathbf{g}_2 \big\rangle$$

This makes sense because by bilinearity, $\langle a\mathbf{g}_1, b\mathbf{g}_2\rangle = \langle \mathbf{g}_1, \mathbf{g}_2\rangle^{ab}$. Essentially the pairing allows us to check “one multiplication”.

Proving Plonk

Recall that Plonk had two types of equations: the gate equations and the wiring constraints.

The gate equations can be proven exactly the same way as in the Fibonacci example (there is a single uniform equation, which applies to all rows of the $N \times 3$ witness matrix).

The wiring constraints are more tricky. This is basically a set of equalities between different cells of the matrix. These equations generate an *equivalence relation* on the set of cells. The witness values respecting this (it's constant on the equivalence classes) is equivalent to the matrix being invariant under a permutation which is cyclic on each equivalence class.

For that statement (invariance under a public permutation), we can build an IOP, in two steps.

An IOP for multiset equality

Given two polynomials P, Q, we want to prove that the *multiset* of values $p_i := P(\omega^i)$ is the same as the multiset $q_i := Q(\omega^i)$. We can do so using the so-called *grand product argument*. Define the product polynomials

$$F(\gamma) := \prod(\gamma - p_i) \qquad \text{and} \qquad G(\gamma) := \prod(\gamma - q_i)$$

Clearly, $F \equiv G$ iff the two multisets agree (consider the multisets of roots!); however if they disagree, then as polynomials in $\gamma \in \mathbb{F}$, they must disagree almost everywhere (assuming $|\mathbb{F}| \gg N$)!

Hence, it's enough to check that they are the same for a random $\gamma \in \mathbb{F}$ selected by the verifier. To do so, construct the interpolation polynomial

$$U(\omega^k) \; := \; \prod_{i<k}(\gamma - p_i)/(\gamma - q_i)$$

Now we need to show that $U(\omega x)/U(x) = (\gamma - P(x))/(\gamma - Q(x))$ for all $x = \omega^i$, and also that $U(1) = U(\omega^N) = 1$; which we can do using the vanishing IOP, for the equations (or just a random linear combination):

$$Z(\omega x)(\gamma - Q(x)) = U(x)(\gamma - P(x)) \qquad \text{and} \qquad \mathcal{L}_0(x)(U(x) - 1) = 0$$

An IOP for public permutation check

Given two (secret) polynomials P, Q encoding values on a subgroup $H = \langle \omega \rangle$ as before, and a public permutation $\sigma : H \to H$, we want to prove that $Q(\omega^i) = P(\sigma(\omega^i))$.

This is equivalent to the sets of pairs $\left\{\left(\omega^i, P(\omega^i)\right)\right\}$ and $\left\{\left(\sigma(\omega^j), Q(\omega^j)\right)\right\}$ agreeing. Instead of using pairs, we can take a random linear combination ($\beta \in \mathbb{F}$ chosen by the verifier):

$$a_i := P(\omega^i) + \beta \cdot \omega^i \qquad \text{and} \qquad b_i := Q(\omega^i) + \beta \cdot \sigma(\omega^i)$$

Now we can use the multiset equality IOP just seen. This results in the interpolation polynomial (with random $\beta, \gamma \in \mathbb{F}$)

$$U(\omega^k) = \prod_{i<k} \frac{P'(\omega^i)}{Q'(\omega^i)} := \prod_{i<k} \frac{P(\omega^i) + \beta \cdot \omega^i + \gamma}{Q(\omega^i) + \beta \cdot \sigma(\omega^i) + \gamma}$$

for which we need to check that $U(1) = U(\omega^{|H|}) = 1$ and that $U(\omega x)/U(x) = P'(x)/Q'(x)$ holds for all $x \in H$, which can be done the usual way.

Plonk protocol

Now we can have a protocol for Plonk, similar to the Fibonacci example:

1. the prover executes the circuit; computes the witness columns X, Y, Z; interpolates them into polynomials and commits to those
2. the verifier chooses random $\beta, \gamma \in \mathbb{F}$ (for the permutation arg.)
3. the prover computes the partial product vector, interpolates it into the polynomial U and commits to it (in practice this is done by adding 3 terms in a step, so that U has the same size as X, Y, Z)
4. the verifier chooses random $\alpha \in \mathbb{F}^\times$ combining coefficient
5. the prover computes the combined polynomial $\mathcal{P}(x)$, which is the linear combination of the gate equation, the permutation equations, and public input (with powers of α)
6. the prover computes the quotient polynomial $\mathcal{Q}(x) = \mathcal{P}(x)/Z_H(x)$, and commits to it
7. the verifier chooses random location $\zeta \in \mathbb{F}$
8. the prover sends the values $X(\zeta)$, $Y(\zeta)$, $Z(\zeta)$, $U(\zeta)$, $U(\omega\zeta)$, $\mathcal{Q}(\zeta)$ (optionally also $Q_*(\zeta)$ and $\sigma(\zeta)$, which the verifier knows but slow)
9. the verifier computes $\mathcal{P}(\zeta)$, and checks the equation $\mathcal{P}(\zeta) = \mathcal{Q}(\zeta)Z_H(\zeta)$

Lookup tables

While in theory anything can be implemented with only addition and multiplication (or polynomial constraints), many operations (for example XOR) are extremely inefficient to do so. Hence the idea to extend your proof system with *lookup tables*.

Suppose you want to do a binary operation $\odot$ on 8 bit integers: You can just list all possible combinations in a big $2^{16} \times 3$ table with rows $(x, y, x \odot y)$, and somehow prove that in every row where this operation occurs, the relevant triple (x, y, z) appears in this table.

Observation: using the standard trick of taking a random linear combination, we can roll this into a single *set membership test*: With a random $\lambda \in \mathbb{F}$ selected by the verifier, it's enough to prove that $x + \lambda y + \lambda^2 z$ is an element of the set

$$T = \left\{ \, x + \lambda y + \lambda^2 (x \odot y) \;\middle|\; x \in \{0 \dots 255\}, \; y \in \{0 \dots 255\} \, \right\}$$

There are several different IOPs to prove such a membership test (for example “plookup” or “log-up”). I skip those now.

Summary

In the IOP-based, “commit-and-prove” paradigm, the typical workflow is the following:

1. use an arithmetization strategy to convert the statement into (polynomial) equations
2. commit to the witness, interpolated as polynomial(s)
3. use components from the “IOP toolkit” to prove the equations
4. replace the oracles in the IOPs with polynomial commitment schemes
5. use the Fiat-Shamir heuristic to achieve non-interactivity
6. use blinding factors to achieve zero-knowledge (optional)

There are two parallel worlds, one based on univariate polynomials and one on multilinear polynomials (with bridges between the two). Both have their own “IOP toolkit”.

Links

Some pointers, if you want to learn more:

- Justin Thaler: Proofs, Arguments, and Zero-Knowledge (online book): `https://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.htm`
- ZKDL Camp by Distributed Labs (lecture series) `https://zkdl-camp.github.io/`
- ZK Whiteboard Sessions (youtube lectures): `youtube.com/playlist?list=PLj80z0cJm8QErn3akRcqvxUsyXWC81OGq`
- Berkeley University ZKP MOOC (online course): `https://zk-learning.org/`
- Dankrad Feist: KZG polynomial commitments (blog post): `https://dankradfeist.de/ethereum/2020/06/16/kate-polynomial-commitments.html`
- Alan Szepieniec: Anatomy of a STARK (tutorial): `https://aszepieniec.github.io/stark-anatomy/`
- zkp.science (link collection): `https://zkp.science/`