Introduction to Zero-Knowledge Proof systems

Balázs Kőműves
(bkomuves@gmail.com)

Budapest, 2022.10.21.

Introduction

The problem of *zero-knowledge proof* can be formalized as follows:

Given a (computable) function

$$f : W \to \mathsf{Bool} = \{\texttt{true}, \texttt{false}\},$$

party $\mathcal{P}$ (the *prover*) wants to convince[1] another party $\mathcal{V}$ (the *verifier*) that they know a value $w \in W$ (usually called the *witness*) such that

$$f(w) = \texttt{true}.$$

[1] note: the word *proof* here is a slight abuse of language: this is *not* a mathematical proof; instead, we are talking about *computationally binding arguments*: we are convinced if it is infeasible in practice to create a false "proof"

But isn't this trivial?!

Problem: Given a function $f : W \to \text{Bool}$, the prover wants to convince the verifier that they know a value $w \in W$ such that $f(w) = \texttt{true}$.

Trivial solution: Send $w \in W$ to the verifier, and let them compute $f(w)$.

However, we want something *better* than this!

Desired properties

Recall the problem: Given a function $f : W \to \text{Bool}$, the prover wants to convince the verifier that they know a value $w \in W$ such that $f(w) = \texttt{true}$.

Some desirable properties of such a system are:

- the verifier learns nothing about w **(zero-knowledge)**
- the proof is very small **(succintness)**
- verifying the proof takes very little time **(scalability)**
- creating the proof doesn't take much more time (asymptotically) than evaluating the function f **(scalability)**
- there is no need of interaction between the prover and the verifier, apart from sending the proof **(non-interactivity)**
- no need for "trusted setup" **(transparency)**

Systems with (some of) these properties are often called SNARKs or STARKs.

Generalizations

Public input: We instead have a function

$$f : W \times X \to \text{Bool}$$

where $x \in X$ is a public input (both $\mathcal{P}$ and $\mathcal{V}$ knows it).

Public output: We have a function

$$f : W \times X \to Y$$

and we want to convince $\mathcal{V}$ that we know a $w \in W$ such that $f(w, x) = y$ for a given $y \in Y$.

It's easy to see how to reduce these to the original problem.

Example: Digital signatures

A well-known example of such a system (satisfying zero-knowledge) are digital signature schemes (RSA, ECDSA, Schnorr, etc...)

A digital signature scheme can be defined as a collection of 3 algorithms:

- **key-generation:** Generates a (random) key-pair consisting of a private (or secret) key sk and a corresponding public key pk;
- **signing:** Given a message $m \in M$ and a private key sk, outputs a *signature* $s \in S$;
- **verifying:** Given a message $m \in M$, a signature $s \in S$ and a public key pk, checks whether the signature was indeed generated by the signing algorithm using the corresponding private key sk (at least with very high probability).

Such signature schemes are used everywhere on the internet, and increasingly often in business, finance, healthcare[2], etc.

[2] for example in the online Covid-19 vaccination certificates!

Digital signatures as a ZKP system

It is easy to see how a digital signature scheme fits into the zero-knowledge setting:

Suppose we want to convince somebody that we know a private key sk corresponding to a given public key pk. The idea is that the verifier generates a *challenge* (a random message), and asks the prover to sign it. Then they can check if the signature is valid.

- the witness w will be the private key sk;
- the public input x will be the pair (pk, m) consisting of the public key and a the challenge message;
- the function f checks if the public key pk really corresponds to the private key sk (the message m is not relevant here, it is only used in the proof);
- and the proof will be the signature s itself.

How is the function f defined?

In the digital signature example, f was a very specific, concrete function. However, we want to solve the ZKP problem for *arbitrary* (computable) functions f!

In practice, we would prefer to represent f as a *computer program*.

However, all known such proof systems are based on some algebraic structures and their properties: polynomials, elliptic curves, pairings, lattices, etc. So we need to *translate* our program f into an algebraic statement of the right form; this process (usually called *arithmetization*) can be very involved in practice, and often encompasses nontrivial engineering challenges.

Popular arithmetization strategies are for example: arithmetic circuits, R1CS, PLONK, AIR (more about these later).

Applications

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

Some example application possibilities 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
- audited binaries (running the compiler inside ZK; not yet practical)

Practicality of ZKP systems

Actual ZKP systems, with many or all desirable properties described above, being able to prove arbitrary computer programs f, exist (in fact, there are many of them, with different tradeoffs; and new ones are popping up basically every week or month).

However, proving is usually many orders of magnitude slower than just evaluating the function f, and the size of f is limited in practice, say to $\approx 10^8$ operations or so.

As an example, one implementation, simulating an existing CPU architecture on which you can define your function, claims a prover speed of about 30k operations per second on a modern laptop, and maybe up to 500k operations per second with GPU acceleration.

However, verification is usually very fast, say in the 10s of millisecond range, and proof sizes are usually between say 300 and 200,000 bytes (depending on the proof system, statement f, and security parameters).

Intuition: How are succint proofs possible?

Most popular ZKP systems, at their core, encode the statement in question into statements about *polynomials* (typically either univariate or multilinear) over *finite fields*.

Why finite fields? Well, that's easy: The computer is discrete and finite.

Why polynomials? The two properties of (univariate) polynomials which make this possible are:

- **interpolation:** A degree d poly can encode any vector $\underline{x} \in \mathbb{F}^{d+1}$
- **roots:** Two distinct polynomials of degree at most d can agree at at most d points (because their difference can have at most d roots)

The latter property essentially means that interpolation polynomials form an *error correcting code* (Reed-Solomon code). In particular, if $|\mathbb{F}| \gg d$, then you can check the equality of two polynomials of degree d by checking their evaluation at a *single random point*: The probability of a false positive[3] is $d/|\mathbb{F}|$, which can be made arbitrarily small.

[3] This is called the Schwartz-Zippel lemma

A blueprint for building ZKP systems

Many of the ZKP systems currently used in practice are built according to the following general plan:

1. translate the statement we want to prove into a statement about polynomials over finite fields (**“arithmetization”**)
2. use a so-called **“Interactive Oracle Proof”** (IOP) to prove it
3. replace the polynomial oracles in the IOP by a **“polynomial commitment scheme”**, making into something practical
4. optionally, add **“blinding factors”** to the polynomials to achieve full zero-knowledge
5. make it non-interactive using the **“Fiat-Shamir heuristic”**

This blueprint is quite modular: Different instances of the first 3 points can be mixed and matched somewhat freely, resulting in many different practical systems with different tradeoffs.

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[4] 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[5]

[4] it is known than IP=PSPACE

Example: Schnorr's identification protocol

An example of an interactive ZK proof is Schnorr's protocol. Given a 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 $h = \mathbf{g}^w \in \mathbb{G}$.

The protocol is the following:

- 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 $\mathbf{g}^r = \mathbf{g}^{u+cw} = a \cdot h^c \in \mathbb{G}$.

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)

Interactive Oracle Proofs

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

IOPs[6] 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.

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

Polynomial commitment schemes (PCS)

The idea is that 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 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.[7]

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 usually achieved in the protocols by adding so called "blinding terms" to the polynomials in question.

[7] 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 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.

An IOP for vanishing on a subset

Suppose we have a polynomial $P(x)$ defined over a finite field $\mathbb{F}_q$, and want to convince the verifier that P vanishes on a given subset $S \subset \mathbb{F}_q$. This is equivalent to $P(x)$ being divisible by $Z_S(x) := \prod_{s\in S}(x - s)$. So the prover can compute the quotient $Q(x) = P(x)/Z_S(x)$, and send over its oracle. The verifier can now choose a random field element $u \in \mathbb{F}_q$, and verify $Q(u)Z_S(u) = P(u)$ by querying the oracles for P and Q.

If[8] P has degree d, Q has degree $d - |S|$, and the relation between the polynomials P, Q, Z_S does not actually hold, the latter equation has the probability of d/q to pass. Typically q is very big, say $q \sim 2^{256}$, while d is related to the size of the statement we want to prove, say $d \sim 2^{24}$, so the probability of a false positive is negligible.

The above test is particularly useful when $S = \langle \mathbf{g} \rangle < \mathbb{F}_p^\times$ is a multiplicative subgroup (of size N). This is because in this case $Z_S(x) = \prod_{i=0}^{N-1}(x - \mathbf{g}^i) = x^N - 1$, which can be computed by the verifier efficiently, in $\log_2(N)$ steps.

[8] note: actually proving the degree bounds is very important for this to work!

An IOP for multiset equality

Given two polynomials P, Q, we want to prove that the *multiset* of values $p_i := P(\mathbf{g}^i)$ is the same as the multiset $q_i := Q(\mathbf{g}^i)$. 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

$$Z(\mathbf{g}^k) := \prod_{i<k}(\gamma - p_i)/(\gamma - q_i)$$

Now we need to show that $Z(\mathbf{g}x)/Z(x) = (\gamma - P(x))/(\gamma - Q(x)) = 0$ for all $x \in \langle \mathbf{g}^i \rangle$ and that $Z(1) = Z(\mathbf{g}^N) = 1$; which we can do using the vanishing IOP, for the polynomials (or just a random linear combination):

$$Z(\mathbf{g}x)(\gamma - Q(x)) - Z(x)(\gamma - P(x)) \qquad \text{and} \qquad \mathcal{L}_0(x)(Z(x) - 1)$$

The random linear combination trick

Suppose we want to prove that $z_i = 0$ for some field elements $z_i \in \mathbb{F}$, $i \in \{0 \dots k\}$. Consider the polynomial

$$f(\alpha) = \sum_{i=0}^{k} \alpha^i z_i$$

If this is not the constant zero polynomial, then it can have at most k roots, so checking the equation $f(\alpha) = 0$ at randomly chosen $\alpha \in \mathbb{F}$ will produce a false positive with only $k/|\mathbb{F}|$ probability.

Similarly, if we want to prove $P_i(x)$ are degree $\leq d$ polynomials, we can check

$$\mathcal{P}(x; \alpha) = \sum_{i=0}^{k} \alpha^i P_i(x)$$

instead, with a random $\alpha \in \mathbb{F}$. The probability of a potential higher-degree term cancelling out is again $k/|\mathbb{F}|$.

This trick is used all the time in ZK proof systems, making them much more efficient.

Arithmetization strategies

As we mentioned before, we have to translate our statements we want to prove into some algebraic statements, for which we have IOP-s, like vanishing on a subgroup etc.

A popular model of computation for this purpose are arithmetic circuits, which are similar to feed-forward electronic circuits, but instead of 0/1 signals we have field elements, and instead of logic gates we have but with addition and multiplication gates. Arithmetic circuits can serve as an "intermediate representation".

Three well-known arithmetization strategies are then:

- rank-1 constraint systems (R1CS)
- algebraic intermediate representation (AIR)
- arithmetic gates + wiring (PLONK)

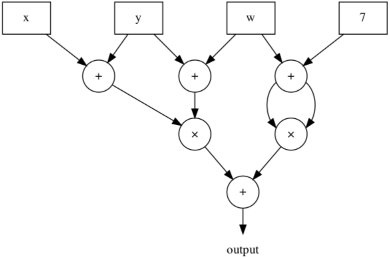
These basic constructions can be then extended with additional features, which are very important to achieve practical performance.

Arithmetic circuits

Consider for example the polynomial function f:

$$f(x, y; w) := (x + y)(y + w) + (w + 7)^2$$

A way of computing this function can be drawn like this:



This is an example of an *arithmetic circuit*. Such circuits are very well-suited for some types of zero-knowledge proofs; that is, for example we want to prove that we know a w such that for some concrete x, y, b we have $f(x, y; w) = b$.

R1CS - Rank-1 Constraint Systems

R1CSs are a popular arithmetization method, well-suited for many ZKP systems. They are essentially a system of quadratic equations.

The idea is very simple: Given a vector z (whose elements correspond to the public inputs, secret witness, intermediate results and final outputs of our computation, all concatenated together), we can consider the generic quadratic equations

$$\langle A_i, z \rangle \cdot \langle B_i, z \rangle - \langle C_i, z \rangle = 0$$

where A_i, B_i, C_i are constant vectors, encoding the computation.

This is somehow simple enough to deal with in actual protocols, and arithmetic circuits are easy to encode as R1CSs (though the naive encoding is not efficient).

AIR - Algebraic Intermediate Representation

In AIR we represent the *execution trace* of a program as an $N \times K$ matrix of field elements. You can think of the columns as “registers”, and the rows as the state of a machine at consecutive time steps.

time	X	Y	Z
	$\vdots$	$\vdots$	$\vdots$
$i-1$	x_{i-1}	y_{i-1}	z_{i-1}
i	x_i	y_i	z_i
$i+1$	x_{i+1}	y_{i+1}	z_{i+1}
	$\vdots$	$\vdots$	$\vdots$

$$
\begin{array}{ll}
x_1 = 17 & \\
y_1 = 0 & y_2 = 0 \\
 & \\
\forall i \in [1, N] : & y_i(1 - y_i) = 0 \\
\forall i \in [1, N] : & 3x_i - y_i \;\; = z_i^2 \\
\forall i \in [1, N-1] : & x_i^2 + 5y_i \;\; = z_{i+1} \\
 & \\
z_N = 1337 &
\end{array}
$$

State transitions are represented as low-degree polynomial constraints, *uniformly repeating for all rows*. Further boundary conditions, translating to equations for particular rows, represent the initial and final states of the machine (alternatively you can just open the corresponding committed values).

AIR example: squared Fibonacci

Consider the following problem: Fix some prime field $\mathbb{F}_p$, and construct the Fibonacci-like sequence:

$$\begin{aligned} a_0 &:= 1 \\ a_1 &:= w \\ a_k &:= a_{k-2}^2 + a_{k-1}^2 \qquad \mod p \end{aligned}$$

Now we want to convince somebody that we know a secret $w \in \mathbb{F}_p$ such that $a_{1000} = y$ for some concrete $y \in \mathbb{F}_p$.

The way we will achieve this is somewhat involved:

1. encode the values $a_0, a_1 \ldots, a_{1000}$ as an interpolation polynomial
2. commit to this polynomial
3. encode the relations between the a_i-s as polynomial constraints
4. rephrase these as statements about the roots of some other polys
5. meaning that some quotient functions are degree-bound polynomials
6. and convince the other party that these are indeed low-degree polys

AIR example: squared Fibonacci (details)

Encode the values a_i in the interpolation polynomial $f(\mathbf{g}^i) = a_i$ where $\mathbf{g}$ generates a subgroup $H < \mathbb{F}_p^\times$ of size $|H| = 2^{10} = 1024$.

The statement $\forall\, k \in \{0 \dots 1021\} : \; a_{k+2} = a_k^2 + a_{k+1}^2$ is equivalent to saying that the polynomial $-f(\mathbf{g}^2 x) + f(x)^2 + f(\mathbf{g}x)^2$ has roots at $\mathbf{g}^k$ for $k \in \{0 \dots 1021\}$. This can be further rewritten as

$$P(x) := \Big(-f(\mathbf{g}^2 x) + f(x)^2 + f(\mathbf{g}x)^2 \Big)(x - \mathbf{g}^{1022})(x - \mathbf{g}^{1023})$$

vanishing on H. But we have already seen an IOP for this!

We also need to prove that $f(\mathbf{g}^0) = 1$ and $f(\mathbf{g}^{1000}) = y$, which can be done either by opening f at those points; or as vanishing on a single point; or alternatively by vanishing on H of

$$\mathcal{L}_0(x)\big(f(x) - 1\big) \qquad \text{and} \qquad \mathcal{L}_{1000}(x)\big(f(x) - y\big)$$

where $\mathcal{L}_k(\mathbf{g}^i) = \delta_{i=k}$ are the Lagrange basis polynomials.

PLONK arithmetization

PLONK is a third popular arithmetization, somewhere inbetween R1CS and AIR. It encodes arithmetic circuits as follows:

First it encodes the individual arithmetic gates as a set of cubic equations; then, separately, it encodes the "wires" between the gates as additional equality constraints between the inputs and outputs of the gates. The latter is achieved using a *permutation argument*.

Encoding of gates: There are $5 + 3 = 8$ column vectors, each row corresponding to a gate:

$$Q_L X + Q_R Y + Q_O Z + Q_M XY + Q_C = 0$$

The Q-s are constant vectors encoding the circuit, and X, Y, Z are the column vectors corresponding to the two inputs and the output of each gate. Finally the vectors are encoded as interpolation polynomials as before.

PLONK wiring

So now each gate is “working” separately, but we also need to ensure the *wiring constraints* connecting the gates. These are simple equality constraints like $Z_3 = X_7$, which means that the output of the 3rd gate is the same as the left input of the 7th gate. All these constrainsts together generate an *equivalence relation* over the $3N$ wires.

The trick is the following: Within each equivalence class, we can fix a cyclic permutation (the elements ordered arbitrarily). Now a vector respecting the equivalence relation is equivalent to it being invariant under the resulting permutation.

So it's enough to prove that the long vector (X, Y, Z) is invariant under this particular permutation $\sigma \in S_{3N}$ (which is fixed as part of the circuit description).

An IOP for permutation check

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

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

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

Now we can use the multiset equality IOP introduced earlier. This results in the interpolation polynomial

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

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

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 $\alpha \in \mathbb{F}$ selected by the verifier, it's enough to prove that $x + \alpha y + \alpha^2 z$ is an element of the set

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

Plookup: An IOP for set membership

Given values p_i (encoded by an interpolation polynomial) and a finite set $T = \{t_j\} \subset \mathbb{F}$, we want to prove that $p_i \in T$ for all i.

The underlying idea is to extend the n values p_i with the m values $t_j \in T$, sort the resulting multiset S, and take the differences $s_{i+1} - s_i$. This difference set (discarding the zeros) should be the same as the difference set of only T (also sorted) if $P \subset T$.

Less naively, instead of taking the differences, we can take *pairs*. Let $S := P \cup T$ (sorted); it is easy to see that $P \subset T$ iff the multiset of pairs (s_i, s_{i+1}) is the same as the pairs (t_i, t_{i+1}) plus the duplicated elements (p_i, p_i). In fact this is true for *any* $S \in \mathbb{F}^{n+m}$! As usual we can fold the pairs into scalars using a random linear combination:

$$s'_i := s_i + \beta s_{i+1} \qquad t'_i := t_i + \beta t_{i+1} \qquad p'_i := (1+\beta) p_i$$

where $\beta \in \mathbb{F}$ is chosen by the verifier. Now it's enough to do a multiset equality check between S' and $P' \cup T'$, which we can do using the grand product argument for

$$\prod_{i=1}^{n+m-1} (s'_i + \gamma) = \prod_{i=1}^{n} (p'_i + \gamma) \prod_{i=1}^{m-1} (t'_i + \gamma)$$

Polynomial commitment schemes

Recall that the problem we want to solve is the following: The prover first *commits* to a degree $\leq d$ polynomial $P(x)$, then later the verifier can ask for evaluations $P(x_i)$ at some points x_i, together *evaluation proofs* ensuring that prover didn't cheat.

Some popular schemes with different tradeoffs are:

- KZG - $O(1)$ proof size and verification; needs trusted setup; based on bilinear pairings
- IPAs (eg. Bulletproofs) - $O(\log(d))$ proof size; $O(d)$, that is, *linear* verification time (but can be aggregated?); transparent (no trusted setup); based on discrete log
- FRI - $O(\log^2(d))$ proof size and verification time; transparent; based on hash functions (Merkle trees) - post-quantum safe
- DARK, Dory, etc...

KZG polynomial commitment

Suppose we are given two copies $\mathbb{G}_1 = \mathbb{G}_2$ of a cyclic group of prime order p, 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. For $u \in \mathbb{F}_p$, 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 $\tau \in \mathbb{F}_p$ is a secret *unknown to both the prover and the verifier*:

$$[1]_1,\ [\tau]_1,\ [\tau^2]_1,\ [\tau^3]_1,\ \dots,\ [\tau^N]_1 \quad \text{and} \quad [1]_2,\ [\tau]_2$$

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

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

KZG evaluation proof

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

First, $P(x_0) = y_i$ 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:

$$\begin{aligned}\mathtt{com}(\text{LHS}) &= \mathtt{com}\big(Q(x)(x - x_0)\big) = (\tau - x_0) \cdot \mathtt{com}(Q) \\ \mathtt{com}(\text{RHS}) &= \mathtt{com}\big(\quad P(x) - y_0 \quad\big) = \mathtt{com}(P) - y_0 \cdot [1]_1\end{aligned}$$

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

$$\big\langle \mathtt{com}(Q)\ ,\ [\tau]_2 - x_0 \cdot [1]_2 \big\rangle = \big\langle \mathtt{com}(P) - y_0 \cdot [1]_1\ ,\ \mathbf{g}_2 \big\rangle$$

This makes sense because by bilinearity, $\langle a\mathbf{g}_1, b\mathbf{g}_2 \rangle = \langle \mathbf{g}_1, \mathbf{g}_2 \rangle^{ab}$.

Interlude: 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[9] H. Assume the length of our vector $\{a_i\}$ is 2^m. First compute $h_i^{(m)} := H(a_i)$. Then define

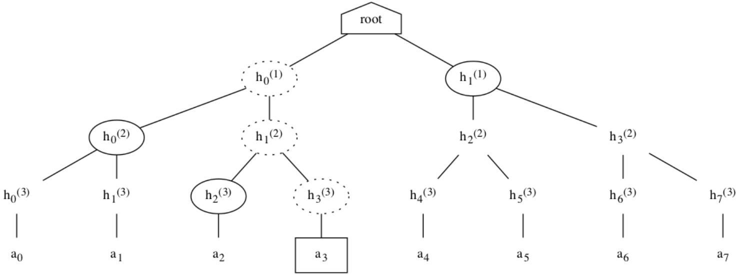
$$h_j^{(k)} := H\left(h_{2j}^{(k+1)} \,\|\, h_{2j+1}^{(k+1)} \right)$$

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!).

[9] these take arbitrary inputs, and produce a random-looking output in a fixed codomain, say $Y = \{0,1\}^{256}$, with the expectation that it is practically impossible to find a pre-image for a random $y \in Y$

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).

FRI approximate low-degree test

Suppose we have a size N vector of committed values a_i (eg. via a Merkle tree), and we want to prove that it is a polynomial of degree $\leq d < N$. This of course impossible to do faster than $O(N)$, because we can just change a single value. What we can do instead, to prove that it is *close to* a degree d polynomial (in Hamming distance). The simple solution for that is to query $d+1$ random points, but d is usually big.

The **FRI step** takes a degree d polynomial f defined on $\langle \mathbf{g} \rangle$, writes it as $f(x) = f_{\mathtt{even}}(x^2) + x f_{\mathtt{odd}}(x^2)$, and returns the random linear combination

$$f'(y) = f_{\mathtt{even}}(y) + \beta f_{\mathtt{odd}}(y)$$

which has degree $\lfloor d/2 \rfloor$ and is defined on the half-size domain $\langle \mathbf{g}^2 \rangle$. We can iterate this to get a sequence $f^{(0)}, f^{(1)}, \ldots, f^{(k)}$ defined on $D_0, D_1, \ldots, D_k$, with $f^{(k)}$ being a constant function.

Consistency check: We can compute

$$f^{(i+1)}(x^2) = \frac{f^{(i)}(x) + f^{(i)}(-x)}{2} + \beta^{(i)} \cdot \frac{f^{(i)}(x) - f^{(i)}(-x)}{2x}$$

FRI low-degree test (IOP of proximity)

Instead of checking the consistency each layer independently, we can optimize it a little bit:

1. the prover commits to $f^{(0)}, f^{(1)}, \ldots, f^{(k)}$, interleaved with the verifier sending $\beta^{(i)}$-s. The last, $f^{(k)}$ is a constant, sent in clear.
2. the verifier asks to open $f^{(0)}$ at a and $-a$ for a random $a \in \langle \mathbf{g} \rangle$;
3. the verifier asks to open $f^{(1)}$ at a^2, and checks the consistency equation;
4. the verifier asks to open $f^{(1)}(-a^2)$ and $f^{(2)}(a^4)$, and checks;
5. the verifier asks to open $f^{(2)}(-a^4)$ and $f^{(3)}(a^8)$, and checks;
6. $\ldots$
7. the verifier asks to open $f^{(k-1)}(-a^{2^{k-1}})$ and checks consistency with $f^{(k)}(x) = c$.

To add ZK, do this on a coset $\{z\mathbf{g}^i : i \in [N]\}$, and add a blinding factor.

FRI polynomial commitment scheme

To commit a polynomial $P(x)$, defined on $H = \langle \mathbf{g}^m \rangle$, first we send the Merkle root of its values *on a larger domain* $D_0 = \langle \mathbf{g} \rangle$, and also do an FRI proving that it is (close to) a low-degree polynomial.

Then for an evaluation proof $P(x_0) = y_0$, there are two cases: If $x_0 \in D_0$, we can just do a Merkle opening. If $x_0 \notin D_0$, we do another FRI low-degree test, this time for the quotient polynomial

$$Q(x) = \frac{P(x) - y_0}{x - x_0}$$

Note that there is no need to commit to Q, because the verifier can simply compute $Q(x)$ from $P(x)$ for any $x \in D_i \subset \mathbb{F}$.

Cost: since we have $O(\log(d))$ Merkle-openings, and each has $O(\log(d))$ cost, the total size of the proof, and also the verification time, is $O(\log(d)^2)$.

Nondeterministic input

An observation, which can be surprising at first, that the prover can include arbitrary extra data into the witness. In fact, when simulating a computation, you could argue that technically all the intermediate results are part of the witness!

This makes the following optimization possible: The prover can compute things *outside* the proof system, and proof system only needs to *verify* the validity of this "god-given" data. This makes sense because running computation inside the proof is orders of magnitude slower than running it outside.

An example of this is sorting a list. Instead of implementing a sorting algorithm inside the proof, the prover can just sort the list itself, and insert the sorted list *and the sorting permutation* into the proof. The proof then only needs to check that: 1) it is a permutation; 2) it permutes the input list into the output list; and 3) the output list is indeed sorted. Note that this can be done in $O(n)$ steps instead of $O(n\log(n))$!

Zero-knowledge Virtual Machines

The idea of zkVM-s is to implement a *virtual machine* inside a ZK proof; that is, the function f you want to check is an interpreter for a programming language (typically something similar to a machine code for a CPU).

This is a very powerful idea! Now you can just write normal programs and run them inside a proof system, instead of manually or algorithmically translating them using the above techniques. It also means that the verification algorithm is exactly the same for arbitrary computation, which is a useful property (especially so in the blockchain context, where the verification needs to run "on-chain").

An example of such a VM is Risc0, which implements a standard-compliant Risc-V microprocessor ISA. Some other examples are Cairo and MidenVM.

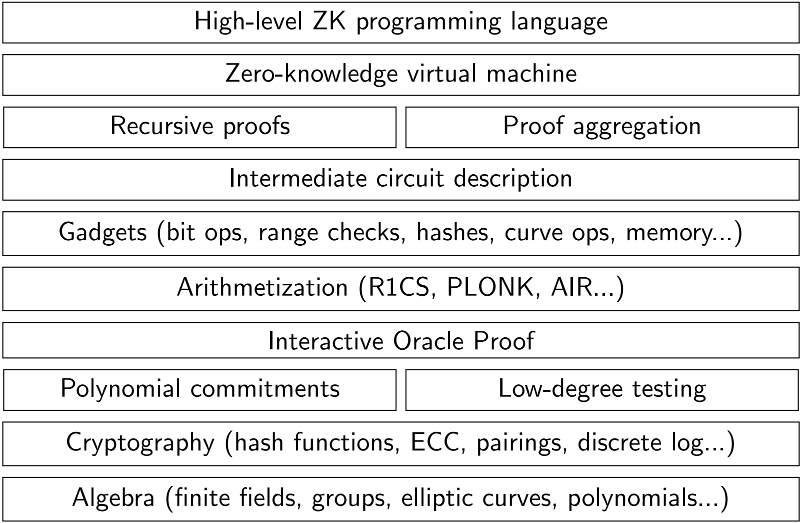
Recursive proofs

The idea of recursive proofs is to put the verification algorithm of a ZK proof system *inside another ZK proof*. So the proof verifies that *another proof is valid*. This is becoming very important, and has several applications:

- making a chain of proofs, each one verifying some statement *and the previous proof*, effectively including a long chain of statement in a single small proof
- making a binary tree of proofs, so the root verifies everything; this enables parallelism in the prover (very important because proving is *slow*!), and again results in a very small proof for a large amount of statements
- make the non-final proofs fast / recursion friendly, and the final proof small or otherwise friendly to a given system, by changing proof systems

Summary

There are a lot of moving component in a ZKP system, so here is a bird's eye view summary:



High-level ZK programming language

Zero-knowledge virtual machine

Recursive proofs | Proof aggregation

Intermediate circuit description

Gadgets (bit ops, range checks, hashes, curve ops, memory...)

Arithmetization (R1CS, PLONK, AIR...)

Interactive Oracle Proof

Polynomial commitments | Low-degree testing

Cryptography (hash functions, ECC, pairings, discrete log...)

Algebra (finite fields, groups, elliptic curves, polynomials...)

Further outlook

There are a lot of important subjects not mentioned above, to keep this talk to a reasonable length (also everything above was overly simplified for the sake of presentation).

Some examples of further subjects:

- proof systems based on multilinear polynomials
- blinding factors to achieve full zero-knowledge
- range checks (extend circuits with inequality checks)
- other “gadgets” or “accelerator chips” (bitwise operations, hashing, elliptic curves, etc)
- how to implement memory and stack in zkVMs
- intricacies of recursive proofs
- ...

This is a very young and very actively researched subject, new research is coming out every month!

References

Some online material, in case you want more:

- Justin Thaler: Proofs, Arguments, and Zero-Knowledge (draft book) `https://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.htm`
- ZK Whiteboard Sessions (youtube talks) `youtube.com/playlist?list=PLj80z0cJm8QErn3akRcqvxUsyXWC81OGq`
- zk Study Club (online talks) `youtube.com/playlist?list=PLj80z0cJm8QHm_9BdZ1BqcGbgE-BEn-3Y`
- Alan Szepieniec: Anatomy of a STARK (online tutorial) `https://aszepieniec.github.io/stark-anatomy/`
- Ulrich Haböck: A summary on the FRI low degree test (review paper) `https://eprint.iacr.org/2022/1216`
- Dankrad Feist: KZG polynomial commitments (blog post) `https://dankradfeist.de/ethereum/2020/06/16/kate-polynomial-commitments.html`
- Goldberg, Papini, Riabzev: Cairo – a Turing-complete STARK-friendly CPU architecture (paper); `https://eprint.iacr.org/2021/1063`