A very brief introduction to Zero-Knowledge Proofs

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

Budapest University of Technology
2023.09.30.

Introduction

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

Given a computable function (relation)

$$F : X \times W \to \mathsf{Bool} = \{\texttt{true}, \texttt{false}\},$$

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

$$F(x, w) = \texttt{true}.$$

[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?!

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

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

However, we want something *better* than this!

Desired properties

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 a "trusted setup" **(transparency)**

Note that while (for historical reasons) we use the generic name "zero-knowledge proofs", for some applications the zero-knowledge property is not required.

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

How is the function F defined?

In some applications (eg. digital signatures), F is a very specific, concrete function, and the proof system exploits its structure. But we want to solve the ZKP problem for *arbitrary* (computable) functions F!

In practice, we would prefer to represent $F(x, w)$ 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 month).

However, proving is usually many (4–5–6) orders of magnitude slower than just evaluating the function F, and the size of F is limited in practice, say to $\approx 10^7$–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, fields has a lot of structure, and 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 data $\underline{x} \in \mathbb{F}^{d+1}$
- **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 is $d/|\mathbb{F}|$, which can be made arbitrarily small.

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[2] 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[3]

[2]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[4] 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.

[4] 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.[5]

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.

[5]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[6] 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 $S = H = \langle \mathbf{h} \rangle < \mathbb{F}_p^\times$ is a multiplicative subgroup (of size N). This is because in this case $Z_H(x) = \prod_{i=0}^{N-1}(x - \mathbf{h}^i) = x^N - 1$, which can be computed by the verifier efficiently, in $\log_2(N)$ steps.

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

An IOP toolkit

Other similar IOP-s (Interactive Oracle Proofs) exist for a variety of basic problems, and more complex ones can be assembled from these. Some examples:

- equality of two polynomials (just evaluate at a random point)
- vanishing of a polynomial on a subset (previous slide)
- comparison to a constant (resulting in a boolean vector)
- checking the sum / product of values
- multiset equality check
- public permutation check
- set membership check

Our goal can be summarized as building an IOP for our problem $F(x, w)$; then we can turn it into a non-interactive argument using a polynomial commitment scheme and Fiat-Shamir heuristic.

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 addition and multiplication gates. Arithmetic circuits can serve as an “intermediate representation”.

Three well-known arithmetization strategies are:

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

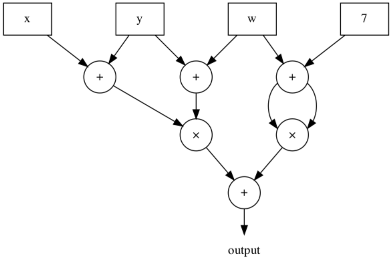
These basic constructions can be then extended with additional features, which are very important to achieve practical performance.

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 $F(x, y; w) = b$.

AIR - Algebraic Intermediate Representation

AIR is an arithmetization strategy different from the arithmetic circuits above.

In AIR we represent the *execution trace* (or intermediate results) of our program $F(x, w)$ as several column vectors, which are then encoded as interpolation polynomials.

In the simplest case these vectors have the same size, forming an $N \times K$ matrix; you can think of the columns as “registers”, and the rows as the state of a machine at consecutive time steps.

State transitions of the program 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 (large) prime field $\mathbb{F}_p$, and construct the Fibonacci-like sequence:

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

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 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.

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 with *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*:

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

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 $d \leq N$, the prover can compute the commitment

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

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, $P(x_0) = y_0$ 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 $\texttt{com}(Q) = [Q(\tau)]_1 \in \mathbb{G}_1$.

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 $\texttt{com}(\text{LHS}) = \texttt{com}(\text{RHS})$ instead. We can easily compute:

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

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

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

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

AIR + KZG + EC pairing + SRS + FS = proof for Fib

Now we have all the ingredients for solving our Fibonacci example. Recall that we want to convince a verifier that we know a $w \in \mathbb{F}_p$ such that for some public $y \in \mathbb{F}_p$ we have $a_{1000} = y$ where

$$a_0 = 1, \qquad a_1 = w, \qquad a_k = a_{k-2}^2 + a_{k-1}^2$$

To do this:

1. compute the sequence $a_0 \dots a_{1023} \in \mathbb{F}_p$
2. interpolate a polynomial $P(x)$ on a subgroup $H < \mathbb{F}_p^\times$ of size $|H| = 1024$ such that $P(\mathbf{h}^i) = a_i$, and commit to it using KZG
3. write down the three constraints (vanishings on H) we need to prove
4. apply the "vanishing on a subgroup" IOP to reduce them to evaluations at a random point
5. compute such a random Fiat-Shamir challenge $\zeta \in \mathbb{F}_p$
6. prove the resulting equation via KZG evaluation proofs
7. the verifier then recomputes the FS challenge ζ, checks the equations, and verifies the KZG evaluation proofs.

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 statements 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 switching proof systems

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/`
- Dankrad Feist: KZG polynomial commitments (blog post) `https://dankradfeist.de/ethereum/2020/06/16/kate-polynomial-commitments.html`
- Craig Costello: Pairings for beginners (online book) `https://www.craigcostello.com.au/s/PairingsForBeginners.pdf`
- Goldberg, Papini, Riabzev: Cairo – a Turing-complete STARK-friendly CPU architecture (paper); `https://eprint.iacr.org/2021/1063`