Proof Systems and Zero-Knowledge Proofs

Noah Stephens-Davidowitz

June 9, 2023

1 Computational Proof Systems

A *huge* amount of progress in computer science has been made by attempting to computationally formalize the notion of a proof. This led to the definition of NP;[1] the definition of the classes IP (interactive proofs, which we will see today), AM, and MA; the celebrated PCP theorem (where PCP stands for "probabilistically checkable proofs"); etc.

1.1 NP review

The canonical example is the complexity class NP. Intuitively, a language $L \subseteq \{0,1\}^*$ is in NP if "for every $x \in L$, there exists a simple proof w showing that $x \in L$." The proof is written using the letter w because it is often referred to as a *witness* for the fact that $x \in L$.

For example, consider the language $L_{\mathsf{2primes}} := \{N \in \mathbb{N} \ : \ N = pq, \text{ for primes } p, q\}$ consisting of natural numbers N that are the product of two primes. (Here, we have jumped from bit strings $\{0,1\}^*$ to natural numbers $\mathbb{N}$. We of course are not bothered by questions like how to represent natural numbers as bit strings.) Without even formally defining what a "simple proof" is, it should be clear that this particular language is in NP. In particular, to prove that $N \in L_{\mathsf{2primes}}$, it suffices to simply provide the factorization of N. In other words, to prove that $N \in L$, it suffices to simply provide primes p, q such that $pq = N$. (Here, we are taking for granted the fact that it is easy to check whether or not p and q are prime. In fact, we have already used this fact in a few places, though it is certainly not obvious.)

This kind of proof has two key properties (shared by any reasonable proof system), which we first describe only informally, called completeness and soundness. *Completeness* means that you can always prove a true statement, e.g., if N *is* a product of two primes, then you can always present me with the two primes. *Soundness* means that you *cannot* prove a false statement, e.g., if N is *not* a product of two primes, then you will not be able to present me with two primes p, q whose product is N. (In this context, completeness and soundness are so trivial that it might seem strange to directly call attention to them at all. Indeed, it is often quite obvious that a proof system is complete and sound, but one must be careful, since obviously "proof systems" that are not complete or not sound are quite a problem.)

To make this idea useful, we must formally define the *verifier* of the proof and place a computational bound on the verifier. For example, if I were computationally unbounded (I am *not*!), then it would be silly to prove to me that N is the product of two primes, since I would already know

[1] Okay, actually the original definition of NP was not proof based. But in hindsight it should have been defined in terms of proofs.

that N is the product of two primes! In this class we therefore of course restrict our attention to polynomial-time verifiers. We will actually briefly restrict our attention to *deterministic* verifiers because this is how NP was historically defined. (The distinction won't be very important for us. The class in which the verifier is allowed to be randomized is known as MA. NP has some additional advantages over MA in that it seems to have *many* natural complete problems, while MA does not seem to have very many.)

The verifier will take as input the *statement* x and a *witness* (or proof) w and output 1 if it "agrees with the proof" and zero otherwise. So, our formal definition is as follows.

Definition 1.1. *A language L is an* NP language *if there exists a deterministic polynomial-time algorithm $\mathcal{V}$ (the* verifier*) and a polynomial $p(n)$ (a bound on the length of the proof) such that the following both hold.*

- **(Completeness.)** *For every $x \in L$, there exists a* witness $w \in \{0,1\}^*$ *with $|w| \leq p(|x|)$ (i.e., a "polynomial-sized proof") such that $\mathcal{V}(x,w) = 1$.*
- **(Soundness.)** *For every $x \notin L$ and every $w \in \{0,1\}^*$ with $|w| \leq p(|x|)$, $\mathcal{V}(x,w) = 0$.*

Notice that we require that the length $|w|$ of w is polynomially bounded in the length $|x|$ of x. Otherwise, the fact that $\mathcal{V}$ is polynomially bounded would be meaningless. (Make sure you see why.)

(You might have seen a different definition of NP before. NP is often defined as the class of languages decidable by a polynomial-time *non-deterministic* Turing machine. If you have seen the other definition, then you should be able to see the equivalence by noting that the non-deterministic choices of the Turing machine correspond to a witness w. Also, I hope you agree that the above definition is much nicer!)

For example, for the L_{2primes} language above, a verifier can take as input an integer N and a witness $w = (p,q)$ consisting of a pair of two integers. The verifier then checks that $N = pq$ and that p and q are both prime (which can be done efficiently).

1.2 Interactive Proofs (IP)

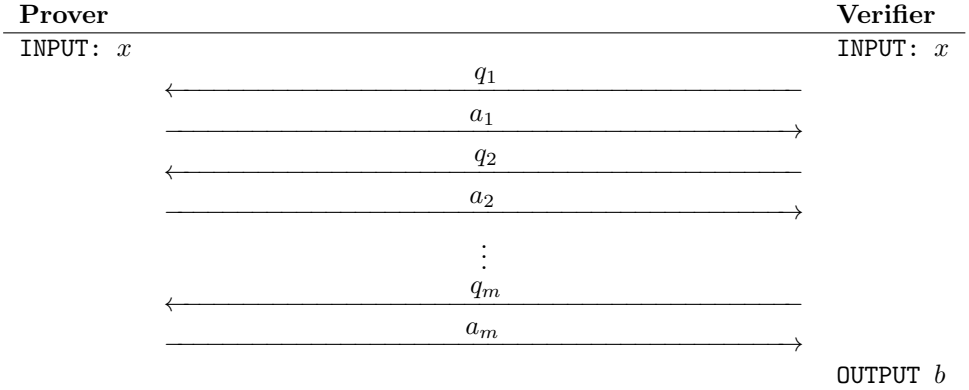
In some sense, the class NP fully captures the idea of a proof as we typically use it in mathematics. E.g., when we prove a statement like $x \in L$ in a paper or in lecture or for a problem set, we more-or-less just write down a string of characters w, which can be checked mechanically by any mathematically-inclined human in order to be fully convinced that the statement is true. (At least, this is an idealized version of what happens. In reality, our proofs are rarely fully formal :), and what counts as a fully formal proof is rather ambiguous and is really more about culture than about rigor. Still, this idealized model is good to keep in mind when writing proofs.)

However, this is clearly not the only way that we can convince each other of certain truths. E.g., this type of formal mathematical proof is inherently *non-interactive*, in the sense that once the proof is written, the verifier is expected to understand it by himself. But, of course, it is often far easier for the verifier to understand something if he can *interact* directly with the *prover*.

This leads us to the notion of an *interactive proof*. An interactive proof augments the notion of proof in three ways. First, we explicitly consider a *prover* $\mathcal{P}$. Our prover is actually a computationally *unbounded* algorithm. In the case of NP, the prover was implicitly defined in terms of the witness w as follows: we can imagine some computationally unbounded prover $\mathcal{P}$ taking as input

x and searching all possible strings to find a witness that she knows would convince $\mathcal{V}$ that $x \in L$. E.g., she might try all possible factors of N and output the factors once she finds them. (Since $\mathcal{P}$ is computationally unbounded, she can iterate through all possible bit strings, and she can certainly check whether $\mathcal{V}(x, w) = 1$.)

The second way in which we change our definition is to make the proof *interactive*. In NP the role of the prover (to the extent that she has any role at all) is simply to write down the string w. In contrast, in our new definition of *interactive proofs*, the prover "stays around to answer questions that the verifier asks." In particular, the prover and the verifier engage in some *interactive* protocol.



Remark (A tedious remark about how to formalize interaction between two algorithms)**.** *One way to pedantically formalize an interactive protocol between two algorithms $\mathcal{V}$ and $\mathcal{P}$ is to view $\mathcal{V}$ and $\mathcal{P}$ as "next-message algorithms." E.g., $\mathcal{V}$ takes as input the "actual input x," a state σ, and the* transcript *so far $(q_1, a_1, \ldots, q_i, a_i)$, and outputs the next message q_{i+1} and an updated state. The protocol proceeds by first calling $\mathcal{V}$ on input x together with the empty state and the empty transcript, receiving as output a new state σ and the first message q_1; then calling $\mathcal{P}$ on input (x, q_1) and the empty state, receiving as output a_1 and the state ρ of $\mathcal{P}$; then calling $\mathcal{V}$ on input (x, σ, q_1, a_1); etc. The protocol ends when $\mathcal{V}$ outputs a special message (say, e.g., the empty message, or the special symbol $\perp$), and the "output of $\mathcal{V}$" is the final state of $\mathcal{V}$. The running time of $\mathcal{V}$ and $\mathcal{P}$ in the above is measured relative to the length of x.*

It is sometimes useful to work at this level of formality, but usually best to just draw pretty diagrams like the one above. One important thing that this formality makes clear, however, is that the prover and the verifier can definitely maintain a state between messages. I.e., they have a state that they update at every step. For example, if in the first step the verifier samples $x \sim \{0,1\}^n$ and then sends its first message $q_1 := f(x)$ for some one-way function f, the verifier can still "remember" x later. This is formally captured by the idea that x can be included as part of the state σ.

The third way in which we change the definition is to allow the verifier to be randomized. Of course, this is quite natural for us because in this course we almost always allow our algorithms to be randomized. Here, randomness serves a particularly important purpose (just like it does in *many* of our applications), and we will have more to say about this below.

For now, let's fix some notation to allow us to reason about protocols more easily. We write $\langle \mathcal{P}(x), \mathcal{V}(y) \rangle$ to represent the interaction of $\mathcal{P}$ and $\mathcal{V}$ in which $\mathcal{P}$ takes as input x and $\mathcal{V}$ takes as input y. (In the above example, $y = x$, but this of course does not have to be true in general.) We write $\mathsf{out}_{\mathcal{V}}\langle \mathcal{P}(x), \mathcal{V}(y) \rangle$ to represent the output of $\mathcal{V}$ in this interaction. Notice that $\mathsf{out}_{\mathcal{V}}\langle \mathcal{P}(x), \mathcal{V}(y) \rangle$ is a random variable, which depends on the random coins of $\mathcal{V}$ and $\mathcal{P}$. We also sometimes use the notation $\Pi = (\mathcal{P}, \mathcal{V})$ to represent the pair of algorithms $\mathcal{P}$ and $\mathcal{V}$, which we think of as a representation of the protocol that $\mathcal{P}$ and $\mathcal{V}$ engage in.

We can now write our formal definition.

Definition 1.2. *A language $L \subseteq \{0,1\}^*$ is* in IP *if there exists a probabilistic polynomial time* verifier $\mathcal{V}$ *and a (possibly unbounded)* $\mathcal{P}$ prover *such that the following hold.*

- **(Completeness.)** *For every $x \in L$,*

$$\Pr[\mathsf{out}_{\mathcal{V}}\langle \mathcal{P}(x), \mathcal{V}(x)\rangle = 1] = 1 \ .$$

- **(Soundness.)** *For every $x \notin L$ and every (possibly unbounded) algorithm (called a "cheating prover" or "malicious prover") $\mathcal{P}^*$,*

$$\Pr[\mathsf{out}_{\mathcal{V}}\langle \mathcal{P}^*(x), \mathcal{V}(x)\rangle = 1] \leq 1/2$$

Intuitively, the above definition says that "the verifier can always be convinced of a true statement," and "no prover can convince the verifier of a false statement with probability better than $1/2$." This value of $1/2$ is called the *soundness error*, and the choice of $1/2$ is arbitrary. In particular, by repeating ℓ times a protocol with soundness error s, we get a protocol with soundness error s^ℓ. Therefore, the definition of IP remains unchanged if we instead require the soundness error to be, e.g., $2/3$ or $1 - 1/n$ or 2^{-n} or $2^{-n^{100}}$. (It is always nice when our definitions are robust to such changes.)

It might seem strange that we require that the prover *always* convinces the verifier when $x \in L$. This property is called *perfect completeness*. We could instead introduce a *completeness parameter* c, and replace the completeness condition above with the condition that

$$\Pr[\mathsf{out}_{\mathcal{V}}\langle \mathcal{P}(x), \mathcal{V}(x)\rangle = 1] \geq c \ .$$

However, it turns out that we may always assume that $c = 1$. In other words, if you give me a protocol satisfying the above definition with, e.g., $c = 2/3$, I can convert it into a different protocol that has $c = 1$. (This is *not* obvious. To convert a protocol with imperfect completeness $c < 1$ into one with perfect completeness, one must show that the statement "a prover can cause the verifier to accept with probability larger than $2/3$" can be proven with perfect completeness.)

Finally, we notice that our definition of soundness is quite strong in that we allow for arbitrary malicious provers $\mathcal{P}^*$. E.g., we *do not* restrict our attention to PPT provers. This is similar to the definition of NP, where we say that when $x \notin L$, there simply does not exist a witness.

1.3 Why the verifier *must* be randomized

We could imagine a different definition in which the verifier $\mathcal{V}$ is required to be deterministic. However, this definition turns out to be uninteresting because it turns out to be equivalent to NP. In other words, "interaction is useless without randomness." To see this, notice that if $\mathcal{V}$ were deterministic, then an unbounded prover $\mathcal{P}$ could *predict* the messages $q_1, q_2, \ldots, q_m$ sent by $\mathcal{V}$. So, from the prover's perspective, the transcript $(q_1, a_1, \ldots, q_m, a_m)$ would be *fixed* given the input x if $\mathcal{V}$ were deterministic. Therefore, instead of interacting with a deterministic verifier $\mathcal{V}$, the prover could simply send the entire transcript $w = (q_1, a_1, \ldots, q_m, a_m)$ as an NP witness for x.

So, any language with an interactive proof with a deterministic verifier has a non-interactive proof with a deterministic verifier—i.e., is in NP. People sometimes explain this succinctly by saying that "interaction is useless without randomness."

1.4 Example: quadratic *non*-residuosity

Consider the group $\mathbb{Z}_N^* := \{z \in \mathbb{Z}_N \ : \ \gcd(N, z) = 1\}$ where the group operation is multiplication modulo N, and consider the subgroup $\mathsf{QR}_N := \{x^2 \ : \ x \in \mathbb{Z}_N^*\}$ of squares modulo N. This is called the group of *quadratic residues* modulo N. Notice that it is in fact a group—i.e., the product of quadratic residues is also a quadratic residue, and the inverse a quadratic residue is also a quadratic residue.

And, not all elements in $\mathbb{Z}_N^*$ are quadratic residues (unless $N = 2$). E.g., 3 is not a quadratic residue modulo 10. (It can be helpful to work out small examples like this, so that these ideas feel less abstract.)

We can then define the language

$$L_{\mathsf{QR}} := \{(N, y) \ : \ y \in \mathsf{QR}_N\} = \{(N, y) \ : \ y = x^2 \bmod N\} \ .$$

This language is called the language of *quadratic residuosity* (which I find to be very difficult to pronounce, and slightly difficult to spell). It is also believed to be hard to efficiently decide whether $y \in \mathsf{QR}_N$, as we saw when we saw Goldwasser-Micali encryption, i.e., we think that quadratic residuosity is not in P (or even BPP). However, quadratic residuosity is clearly in NP, since a square root of y modulo N serves as a witness that $y \in \mathsf{QR}_N$.

But, what about the language

$$L_{\overline{\mathsf{QR}}} := \{(N, y) \ : \ y \notin \mathsf{QR}_N\} \ ?$$

This is called the language of *quadratic non-residuosity* (which is also very difficult for me to say!). Now, it is not so obvious what a witness for this language would look like. (Well, actually we have seen that the factorization of N can be used as a witness for $L_{\overline{\mathsf{QR}}}$, since with this we can check ourselves whether an element is a quadratic residue. But, for now we'll pretend that we don't know this. It is annoyingly difficult to find a simple example of a language that is not known to be in NP but that has a simple interactive proof. So, we often settle for languages that have very simple interactive proofs but more complicated non-interactive proofs.) Fortunately, there is a beautiful interactive proof system $\Pi(\mathcal{V}, \mathcal{P})$ for $L_{\overline{\mathsf{QR}}}$. It goes as follows.

Prover		**Verifier**
`INPUT:` (N, y)		`INPUT:` (N, y) $x \sim \mathbb{Z}_N^*$, $b \sim \{0, 1\}$
	$\xleftarrow{\qquad z := x^2 y^b \qquad}$	
`IF` $z \in \mathsf{QR}_N$, $b' = 0$ `ELSE`, $b' = 1$		
	$\xrightarrow{\qquad b' \qquad}$	
		`IF` b = b', `OUTPUT` 1 `ELSE`, `OUTPUT` 0

To see that this protocol works, we only need to observe two things. First, if $y \notin \mathsf{QR}_N$, then $x^2 y \notin \mathsf{QR}_N$ while of course $x^2 \in \mathsf{QR}_N$. Completeness follows immediately from this.

Second, if $y \in \mathsf{QR}_N$, then $z := x^2 y \in \mathsf{QR}_N$ is a uniformly random element in QR_N. (This follows from the fact that QR_N is a group.) From this, we immediately derive soundness, by noting that when $(N, y) \notin L$ (i.e., when $y \in \mathsf{QR}_N$), the distribution of the element z is independent of the bit b. Therefore, no matter how a malicious prover behaves, it cannot guess the bit b with probability better than $1/2$.

Therefore, $L_{\overline{\mathsf{QR}}} \in \mathsf{IP}$.

I personally think that this proof system is quite beautiful! I highly recommend taking a moment to just sort of take in how elegant this protocol is. Notice that it is closely related to the Goldwasser-Micali encryption scheme. E.g., it can be used to prove that the public key for a Goldwasser-Micali encryption scheme was generated properly—or to prove that a ciphertext is an encryption of 1. Below, in Section 1.6, I say more about why I find this to be so beautiful.

1.5 Example: graph non-isomorphism

Recall that a *graph* with n vertices is defined by a set $G \subseteq [n]^2$ of edges. Each edge $e = (u, v)$ is a pair of vertices $u, v \in [n]$. (Formally, we have defined a directed graph here, because we have written our edges as *ordered* pairs. Graphs with edges $\{u, v\}$ that are unordered are *undirected* graphs. For the purposes of this lecture, the distinction is not important.)

Two graphs with edge sets G_1, G_2 are *isomorphic* if there exists a bijection (often called a permutation in this context) $\pi : [n] \to [n]$ such that $\pi(G_1) := \{(\pi(u), \pi(v)) \ : \ (u, v) \in G_1\} = G_2$. In other words, two graphs are isomorphic if there is a way to *rename* the vertices $[n]$ in a way that converts G_1 into G_2. We write $G_1 \cong G_2$ if G_1 is isomorphic to G_2.

We can then define the language $L_{\mathsf{iso}} := \{(G_1, G_2) \ : \ G_1 \cong G_2\}$ to be the language of *graph isomorphism*. Clearly, graph isomorphism is in NP, with the witness given by an isomorphism π. (It is actually widely believed that graph isomorphism is in P, but this is not proven. The best that we know is that there exists

an algorithm for graph isomorphism that runs in time $2^{\text{poly}\log(n)}$ [Bab16], while to place it in P, we would need a running time of $\text{poly}(n) = 2^{O(\log n)}$. The fact that graph isomorphism is widely believed to be in P makes this a less interesting example :-/.)

However, the complement language $L_{\overline{\mathsf{iso}}} := \{(G_1, G_2) \ : \ G_1 \ncong G_2\}$, *graph non-isomorphism*, is not obviously in NP. (Indeed, it is not known to be in NP, though certainly if graph isomorphism were in P, then graph non-isomorphism would also be in $\mathsf{P} \subset \mathsf{NP}$. Formally P is closed under taking the complement of a language, while NP is thought not to be closed under complement.)

There is, however, a very simple and beautiful *interactive* protocol for graph non-isomorphism, which we present below. For the purposes of this protocol, we write $S_n := \{\pi : [n] \to [n] \ : \ \pi \text{ is a bijection}\}$ for the set of all bijections from $[n]$ to $[n]$. (S_n is commonly referred to as the *symmetric group*, and it plays a fundamental role in both group theory and graph theory.)

Prover		**Verifier**
`INPUT:` (G_0, G_1)		`INPUT:` (G_0, G_1) $\pi^* \sim S_n$, $b \sim \{0,1\}$
	$\xleftarrow{\qquad G^* := \pi^*(G_b) \qquad}$	
`IF` $G^* \cong G_0$, $b' = 0$ `ELSE`, $b' = 1$		
	$\xrightarrow{\qquad b' \qquad}$	
		`IF` b = b', `OUTPUT` 1 `ELSE`, `OUTPUT` 0

The analysis of this protocol is essentially identical to the analysis of the protocol for quadratic non-residuosity. To see that the protocol is complete, notice that we always have $G^* \cong G_b$, and if $G_0 \ncong G_1$, then $G^* \ncong G_{1-b}$. So, when $G_0 \ncong G_1$, we always have $b = b'$.

To see that the protocol is sound, notice that if $G_0 \cong G_1$, then no matter what value b takes, G^* is a uniformly random permutation of G_0 *and* a uniformly random permutation of G_1, since all permutations of G_0 are also permutations of G_1 and vice versa. (Formally, this follows from the fact that S_n is a group.) So, no matter how a cheating prover $\mathcal{P}^*$ behaves, the bit b is uniformly random and independent of G^* from his perspective, and therefore his probability of success is at most $1/2$.

1.6 But the verifier "didn't learn anything that he didn't already know"!

Notice that the two protocols described above have a very curious property. We saw that the protocols are sound. So, if the prover manages to guess the bit b (or perhaps guesses many bits $b_1, \ldots, b_\ell$ obtained by running this protocol many times in a row, to reduce the soundness error), the verifier has in some sense "learned" that, e.g., $G_0 \ncong G_1$. Certainly, the verifier has been convinced of this fact.

On the other hand, the verifier seems to have "learned nothing that he did not already know." In particular, all the prover did was send the bit $b' = b$ to the verifier. But, the verifier picked b himself! So, he already knew b! It seems that "the only information that the verifier gained is the fact that the prover can tell him what he already knew."

There are many other things that the verifier has *not* learned. He has not learned "why the graphs are not isomorphic." (E.g., perhaps G_0 contains some kind of subgraph that G_1 does not.) He has not even learned how to convince someone else that the graphs are not isomorphic! So, this really is quite a different kind of proof than the one that we're used to. In particular, we usually think of proofs as transferable—if I prove something to you and you understand the proof, then you can prove the same statement yourself. (I guess that even if you do *not* understand the proof, you can still copy it verbatim.) But this strange kind of proof is not at all transferable. (Notice that this uses the fact that the verifier is randomized. If I have proven to you that y is not a quadratic residue by successfully guessing your bit b given $z = y^b x^2$, this does not seem to help you to guess the bit b' from $z' = y^{b'}(x')^2$.)

This weird property in which $\mathcal{V}$ seems to have "learned nothing at all from a proof" is called the *zero-knowledge* property, which we define formally below.

2 Zero-knowledge proofs—I might as well be talking to myself

The definition of a zero-knowledge proof is perhaps my favorite in all of cryptography. (I also like the protocols themselves. I'm kind of a huge fan of zero-knowledge proofs, though I don't actually work in the area :).) We want to somehow say that a proof is zero knowledge if "the verifier learns nothing from the proof except for the fact that the statement is true." For example, you might want to convince me that the Riemann Hypothesis is true without revealing your proof. Or, you might wish to convince me that some bit string y is the output of some one-way function f without revealing the preimage to me. Or, maybe you want to prove to me that your public key N is really the product of two primes without revealing the two primes. Or, maybe for some reason you want to convince me that two graphs aren't isomorphic without revealing anything else. (Like, you know, at a party or whatever.)

So, we want to prove things "in zero knowledge." It seems like in order to come up with a good definition of "zero knowledge," we might have to define what "knowledge" is or what "learning" is or at least come close to defining some of these tricky concepts. E.g., how do I argue that "this proof shows that $y = f(x)$ for some x without letting the verifier learn anything at all about x" without somehow defining what it *means* to "learn" something? This is scary the definition of "learning" is the kind of thing that philosophers have spent millenia debating—not the kind of thing that computer scientists solve in a week.

It turns out that you can mostly avoid this quagmire, by just showing a sufficient condition to imply that the verifier "gains no knowledge" from an interaction, rather than trying to define what it means to "gain knowledge." The brilliant observation of Goldwasser, Micali, and Rackoff [GMR85] is that "you don't learn anything from a boring conversation." (This is my own interpretation; not theirs.) What's a boring conversation? Well, Veronica (the verifier) is bored by a conversation with Peter (the prover) if Peter doesn't contribute anything. In particular, if Veronica "could have had the same conversation by talking to herself," then she is going to be bored and she won't learn anything. (This kind of reminds me of the expression "like talking to a brick wall" to describe what it's like to have a boring conversation.)

Before making this formal, maybe you can see how it applies to our examples of quadratic non-residuosity and graph non-isomorphism. In both cases, the only thing that Peter did to convince Veronica of something was to tell her what she already knew (in particular, her bit b). So, Veronica could have had the whole conversation without Peter! The "only thing that Veronica learned from the conversation was that Peter *could* successfully guess her bit b, which is exactly equivalent to the statement that Peter was trying to prove." (You'll notice that I use scare quotes *a lot* when I'm talking about "knowledge" because "knowledge" is such a thorny concept.)

To make this definition formal, we introduce the (very clever!) notion of a *polynomial-time simulator* S. Intuitively, we say that $\Pi = (\mathcal{P}, \mathcal{V})$ is zero knowledge if there is some polynomial-time simulator S that can replicate "what $\mathcal{V}$ sees in the protocol," without access to $\mathcal{P}$. E.g., in the two examples that we gave above, the simulator S is quite simple. For example for graph non-isomorphism, the simulator S samples $b \sim \{0,1\}$ and $\pi^* \sim S_n$, and produces the two-message transcript $(G^* := \pi^*(G_b), b' = b)$. Clearly, the distribution of (b, π^*, G^*, b') is identical to the distribution that $\mathcal{V}$ sees in an honest run of the protocol.

To make this formal more generally, we introduce the notation $\mathsf{view}_{\mathcal{V}}\langle \mathcal{P}(x), \mathcal{V}(y)\rangle$ for the *view* of $\mathcal{V}$ in the interaction $\langle \mathcal{P}(x), \mathcal{V}(y)\rangle$. Formally, the view is a random variable consisting of $\mathcal{V}$'s random coins together with all messages that $\mathcal{V}$ sends and receives (i.e., the *transcript* of the protocol). In practice, it is often much easier to think of the view as just "the full list of variables that $\mathcal{V}$ sees in the protocol." E.g., for the graph non-isomoprism protocol, the view is (b, π^*, G^*, b'). Notice, e.g., that the view includes the bit b, which is in some sense part of the internal state of $\mathcal{V}$. (In all of our examples, what counts as part of the view of $\mathcal{V}$ will be more-or-less as clear as it is in this case.)

Our simulator will reproduce this random variable. Or, more accurately, our simulator's output will be distributed identically to this random variable. (We will later relax the requirement that the distributions must be identical.)

2.1 Honest verifiers

We are now ready for our definition. Our first version of zero knowledge will only consider *honest verifiers*, that is, we will only guarantee that the protocol is zero knowledge if the verifier behaves as it is supposed to. (Sometimes such verifiers are called "honest but curious," since they behave honestly but are still curious enough to still try to learn what they can from an interaction with the prover. "Honest but curious" is a great name. Such verifiers are also sometimes called "semi-honest," but this is a terrible name.)

Definition 2.1. *We say that a proof system $\Pi = (\mathcal{V}, \mathcal{P})$ for a language $L \subseteq \{0,1\}^*$ (the fact that Π is a protocol for L already implies that $\mathcal{V}$ is PPT, and the protocol is complete and sound) is* honest-verifier perfectly zero knowledge *if there exists a PPT simulator S such that for all $x \in L$, $S(x)$ is distributed identically to $\mathsf{view}_{\mathcal{V}}\langle \mathcal{P}(x), \mathcal{V}(x)\rangle$.*

(Notice that we only ask for the above to hold when $x \in L$. One reason for this is that the behavior of the honest prover $\mathcal{P}$ is simply undefined when $x \notin L$. Honest provers don't try to convince us that x is in the language when it is not :)! You can also think of this as a very clever way of capturing the notion that "$\mathcal{V}$ learns nothing *except* the fact that $x \in L$.")

The two protocols that we saw in the previous section are both honest-verifier perfectly zero-knowledge protocols. Our argument above should make this clear. Indeed, in both cases the simulator just behaves identically to the verifier and sets $b' = b$.

Again, this definition is extremely beautiful and extremely clever. It is worth taking a moment to just enjoy the definition :).

2.2 Malicious verifiers

But, there is a problem with the above definition in that it implicitly assumes that the verifier behaves honestly. But, what if the verifier does not behave honestly? We often refer to arbitrary verifiers as *malicious* verifiers (or sometimes "possibly malicious" verifiers, which I suppose is a bit more polite).

In fact, both of the protocols that we saw in the previous section are horribly insecure if the verifier is malicious. I.e., if the verifier deviates from the protocol, then he can learn information that he should not learn. For example, in the quadratic non-residuosity protocol, suppose that instead of setting $z := x^2 y^b$, the verifier chooses z in some other way. Then, the prover will dutifully tell the verifier whether $z \in \mathsf{QR}_N$. In other words, the prover $\mathcal{P}$ is effectively an *oracle* for the quadratic residuosity problem. (Or, if you like, an oracle that decrypts Goldwasser-Micali ciphertexts!) Interacting with this oracle is only zero knowledge if the verifier happens to behave honestly and only calls the oracle on values of z for which it already knows the answer. This is a bit silly, since the prover probably does not trust the verifier to behave exactly as he's supposed to!

So, we need a stronger definition that considers arbitrary *malicious* PPT verifiers $\mathcal{V}^*$. Of course, we can't hope to create a single simulator S that simulates the view of an arbitrary $\mathcal{V}^*$, since, e.g., different malicious verifiers $\mathcal{V}^*$ could send different first messages. Instead, we build a simulator $S^{\mathcal{V}^*}$ that has *oracle access* to $\mathcal{V}^*$.

Definition 2.2. *We say that a proof system $\Pi = (\mathcal{V}, \mathcal{P})$ for a language $L \subseteq \{0,1\}^*$ (i.e., $\mathcal{V}$ is PPT, and the protocol is complete and sound) is* perfectly zero knowledge *if there exists a PPT simulator $S^{\mathcal{V}^*}$ such that for all $x \in L$ and all PPT (possibly malicious) verifiers $\mathcal{V}^*$, $S^{\mathcal{V}^*}(x)$ is distributed identically to $\mathsf{view}_{\mathcal{V}^*}\langle \mathcal{P}(x), \mathcal{V}^*(x)\rangle$.*

When we prove security against malicious verifiers, we typically describe the simulator as interacting with the malicious verifier $\mathcal{V}^*$. We then argue that the interaction with S is identical from the perspective $\mathcal{V}^*$ to the interaction with $\mathcal{P}$. This is equivalent to the above definition. (We will see this below.)

2.3 A protocol for quadratic residuosity

We now present two protocols that are secure against malicious verifiers.[2]

Our protocols are for graph isomorphism and quadratic residuosity. Of course, we already saw that these problems are in NP, so they have very simple (even non-interactive!) proofs. But that does not make it immediately obvious that they have *zero-knowledge* proofs. (In the next lecture, we will see that every language in NP has a zero-knowledge proof, after weakening the notion of zero knowledge slightly. In fact, every language in IP = PSPACE has a zero-knowledge proof, though we will not see that in this class.)

Here is a protocol for quadratic residuosity. Let x be a square root of y modulo N. (Notice that we give the prover x as input. This isn't strictly necessary for our formal definition, since the prover can be computationally unbounded and therefore is perfectly capable of computing a square root x of y itself. However, it's nice to notice that, if the prover is given x as input, the prover below actually can be implemented efficiently. We won't make a big deal out of this in this course, but it is of course much better to have a protocol in which the prover can be implemented efficiently. It turns out that this is possible if and only if we're trying to prove a language that is in NP—ignoring the minor distinction between NP and MA.)

Prover		**Verifier**
`INPUT:` (N, y, x)		`INPUT:` (N, y)
$r \sim \mathbb{Z}_N^*$		
	$\xrightarrow{\quad z := r^2 \quad}$	
		$b \sim \{0, 1\}$
	$\xleftarrow{\quad b \quad}$	
	$\xrightarrow{\quad c := x^b r \quad}$	
		`IF` $c^2 = zy^b$, `OUTPUT` 1 `ELSE`, `OUTPUT` 0

Intuitively, in this protocol, $\mathcal{P}$ "commits" to a uniformly random element $z \in \mathsf{QR}_N$ by sending it to $\mathcal{V}$. $\mathcal{V}$ then requests either the square root of z or the square root of yz. (This format of "commit, challenge, response" is very common in these protocols. Protocols with this form are called "Sigma protocols" because apparently the three-message format looks kind of like the Greek letter Σ. I don't really see how it looks like a Σ. I would have called them "E protocols," or if I wanted to be fancy "Ξ protocols," where Ξ is the Greek letter Xi, pronounced "ksi.")

Since QR_N is a group, if $y \in \mathsf{QR}_N$, then both elements have a square root, and $\mathcal{P}$ can easily respond to either request. (I.e., the protocol is complete.) On the other hand, if $y \notin \mathsf{QR}_N$, then z and yz cannot both be in QR_N. So, with probability at least $1/2$, a malicious prover will not be able to provide the requested square root. (I.e., the protocol is sound. Beautiful, right?! I *really* love these protocols.)

So, the above protocol is complete and sound. We now prove that it is zero knowledge. The intuition is that no matter what a malicious verifier $\mathcal{V}^*$ does, it only learns the square root of a random element in QR_N, because both z and yz are uniformly random. So, a simulator should be able to sample the square root c first uniformly at random, then set z such that $c^2 = zy^b$ to generate the same distribution. Making this formal takes some work. (It is a nice exercise to simply prove that the above protocol is perfectly *honest-verifier* zero knowledge, before reading the more difficult proof below that works even for malicious verifiers.)

Theorem 2.3. *The above protocol for quadratic residuosity is perfectly zero knowledge.*

[2] In both cases, our simulators will only run in *expected* polynomial time. We have been deliberately vague throughout this course about the precise definition of PPT, and in particular whether a PPT algorithm must *always* run in polynomial time or whether it's sufficient for the *expected* running time to be polynomial. Anyway, in the next lecture, we will weaken the definition of *perfect* zero knowledge that we used above. It is then easy to see how to convert these simulators to polynomial-time simulators at the expense of moving from perfect zero knowledge to *statistical* zero knowledge.

Proof. Our simulator S behaves as follows on input (N, y) for $y \in \mathsf{QR}_N$. (Remember that we only need to prove the zero-knowledge property when the input string is in the language. This is important.) The simulator first samples a uniformly random bit $b' \sim \{0,1\}$ and uniformly random $c \sim \mathbb{Z}_N^*$. (This trick of sampling things in a different order is very common.) It then sets $z := c^2/y^{b'}$ and sends this to the (possibly malicious) verifier $\mathcal{V}^*$, receiving in response some bit $b \in \{0,1\}$. If $b \neq b'$, the simulator simply starts the whole process over. Otherwise, the simulator responds with c and outputs the resulting view of $\mathcal{V}^*$ in this interaction. (This ability to "start over" is a very important trick. Intuitively, this is the main advantage of the simulator—it can simply start over if $\mathcal{V}^*$ does not output what it was hoping for. Cool, right?!)

First, we observe that our simulator runs in expected polynomial time. To see this, it suffices to notice that $\Pr[b = b'] = 1/2$, regardless of the behavior of $\mathcal{V}^*$. This then implies that the expected number of times that the simulator needs to start over is simply $1/2 + 1/4 + 1/8 + 1/16 + \cdots = 1$ (which is certainly polynomially bounded), so that its expected running time is polynomially bounded. Indeed, since $y \in \mathsf{QR}_N$, z is a uniformly random element in QR_N, regardless of b'. Therefore, the bit b must be independent of the bit b', and we have $\Pr[b = b'] = 1/2$, as needed.

It remains to argue that the view produced by the simulator is indistinguishable from the view produced by $\mathcal{V}^*$ interacting with $\mathcal{P}$. To do this, we again notice that z is a uniformly random element in QR_N in both the real protocol and the simulated protocol. Since z is independent of b' in the simulated protocol (as we argued above), it follows that whether or not S starts over is independent of z. Therefore, even conditioned on reaching the end of the protocol, z is uniformly random in QR_N.

The bit b is then chosen by $\mathcal{V}^*$, and its distribution is clearly the same in both cases since z has the same distribution in both cases (even conditioned on $b = b'$). (Notice that this previous statement is slightly delicate because b could depend in some complicated way on z. There are many ways to get this proof wrong :), and in general one should be very careful when working with random variables conditioned on some event.) Finally, the last message c is fixed given z and b in both cases, so its distribution is also identical in both cases, and the result follows. □

2.4 A protocol for graph isomorphism

Finally, we give a protocol for graph isomorphism that is zero knowledge against malicious verifiers. It is quite similar to the protocol for quadratic residuosity, and we therefore do not include the analysis. I'm mostly just including it because it's another pretty protocol :).

Let π be a bijection such that $\pi(G_1) = G_0$. We write $\pi \circ \pi'$ for the map obtained by composing π and π', and we define $\pi^1 := \pi$ and π^0 to be the identity.

Prover		**Verifier**
`INPUT:` (G_0, G_1, π)		`INPUT:` (G_0, G_1)
$\pi^* \sim S_n$		
	$G^* := \pi^*(G_0) \longrightarrow$	
		$b \sim \{0,1\}$
	$\longleftarrow b$	
	$\pi' := \pi^* \circ \pi^b \longrightarrow$	
		`IF` $\pi'(G_b) = G^*$, `OUTPUT` 1 `ELSE`, `OUTPUT` 0

References

[Bab16] László Babai. Graph isomorphism in quasipolynomial time. In *STOC*, 2016. 6

[GMR85] S Goldwasser, S Micali, and C Rackoff. The knowledge complexity of interactive proof-systems. In *STOC*, 1985. 7