CS174 Spr 99 Lecture 26 Summary John Canny

Zero-Knowledge Proofs

The basic idea of zero-knowledge proofs (ZKP) is to prove that you know something without giving away the knowledge itself. This may sound paradoxical but it isnt. Basically, any NP-complete problem has a zero-knowledge proof, and can be verified in polynomial time. For example, if you found a 3-coloring of a large graph (an NP-hard problem), you could prove to a friend that you had the coloring without giving them any idea of what the coloring was. And the verification would take only polynomially-many steps.

A very surprising consequence is that mathematical proofs have zero-knowledge proofs. Pierre de Fermat claimed that he had a marvelous proof of his famous last theorem but that his notebook margin was too small to contain. That claim frustrated mathematicians for centuries as they tried unsuccessfully to find the proof. These days, Fermat could have set up a public web server that other mathematicians could have probed with certain queries. Those queries would have proved to the mathematicians that he did have the proof, even though they would have had no idea what it was.

Graph Coloring

(Goldreich, Micali, Wigderson) A zero-knowledge proof that a graph G with n vertices is 3-colorable (vertices are properly 3-colorable). It is assumed that Prover and Verifier are agreed on the graph G and that the prover claims to know a 3-coloring of G.

Protocol:

Do kn^2 times:

1. **Prover** randomly permutes his 3-coloring of G to get G'. (there are $3! = 6$ permutations of R, W, B).
   Prover paints nodes with gray (lottery-type) paint. (Really done with RSA or a private-key system.)
2. **Verifier** selects 2 adjacent nodes **at random**.
3. **Prover** checks that requested nodes are adjacent. (If not, prover halts); then prover scrapes off the gray paint from the 2 selected nodes.
4. **Verifier** checks that the 2 nodes are colored differently & that both colors are from the set $\{R, W, B\}$.
   If **not**, Verifier **rejects** proof. Theorem might be true, but proof is bad.

Verifier accepts.

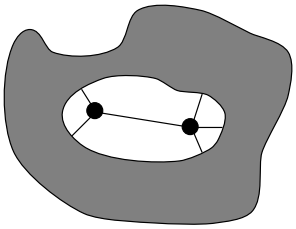
Theorem
The above protocol is a *probabilistic interactive proof* that G is 3-colorable (This is a guarantee to the **Verifier**).

Proof

1. If G is 3-colorable and prover and verifier follow protocol, then verifier **accepts**. Why?
2. If G is **not** 3-colorable & verifier follows protocol (no matter what **Prover** does) then verifier **probably** rejects.
3. Probability of accepting bad proof is $\left(1 - \frac{2}{n^2}\right)^{n^2 k} \leq \frac{1}{e^{2k}}$. Why? If G is not 3-colorable then In any "round", verifier discovers that 2 adjacent nodes are colored the same (ie. there is cheating) with probability $1/\binom{n}{2} \geq \frac{2}{n^2}$. The probability that verifier does **not** discover any cheating in one round is $\leq \left(1 - \frac{2}{n^2}\right)$; therefore in kn^2 rounds, it is $\leq \left(1 - \frac{2}{n^2}\right)^{n^2 k}$. QED

Theorem
Above protocol is 0-knowledge (this is a guarantee to the **Prover**).

Proof idea:
Verifier learns only that adjacent nodes are colored differently. What verifier gets from prover is a **probability distribution** on pairs of adjacent nodes.



If the prover showed this graph, all 3! colorings from $\{R, W, B\}$ are equally likely.

Verifier could create such a probability distribution for himself (called a **simulation**). Thus the verifier gets **no information** from prover that he could not get for himself without help from the prover. QED

A Simple Discrete-log Zero-Knowledge System

Suppose you want to prove your identity to someone, in order to cash a check or pick up a package. Most forms of ID can be copied or forged, but there is a zero-knowledge method that cannot. At least it cant assuming discrete logs are hard to compute. Let p be a large prime, and suppose you choose an x at random which will be your secret ID number. Now choose a generator A and compute $B = A^x \pmod{p}$. You can safely publish A, B, and p because an eavesdropper cannot compute x from that data if discrete log is hard. Your entry might be in a phone book as

"Will smith, Discrete-log key: (A, B, p)"

Now if you show up at the post office to collect a package, you could produce x and anyone could verify that $B = A^x \pmod{p}$. But then any eavesdropper could catch x and impersonate you later. Its better to keep x secret and only answer certain questions about it. Specifically:

1. Prover (you) chooses a random number $r < p$ and sends the verifier $h = A^r \pmod{p}$.
2. Verifier sends back a random bit b.
3. Prover sends $s = (r + bx) \pmod{(p-1)}$ to verifier.
4. Verifier computes $A^s \pmod{p}$ which should equal $hB^b \pmod{p}$.

The basic idea here is that if $b = 1$, the prover gives a number to the verifier that looks random ($s = r + x \pmod{(p-1)}$). But the verifier already knows $h = A^r$ and $B = A^x$ and can multiply these and compare them to A^s.

We should be careful what is proved by that. What the verifier actually sees are h and s, and so what they verify is that $s = dlog(h) + x \pmod{(p-1)}$, where $dlog(h)$ is the discrete log of h relative to A. The verifier knows s and so do you. Now if you also know dlog(h), then its clear that you know x. So it remains for you to convince the verifier that you know dlog(h).

That's where the random bit comes in. If $b = 0$, you the prover just send $s = r$ back to the verifier. The verifier then checks that $h = A^r \pmod{p}$, i.e. that r is the discrete log of h.

So depending on the random bit, the verifier gets either s or r but never both (because their difference is x). Thus the verifier gets no information about x.

On the other hand, if you the prover cheat by sending verifier a h whose discrete log you dont know, the verifier will discover you with 50% probability. So after k trials, the expected number of bits that were 0 is $k/2$ and if the verifier found that $h = A^r$ on all of these, verifier would know that the probability of you cheating on a given round is less than $2^{-k/2}$.

The probability of you cheating on the rounds where $b = 1$ is the same as the rounds where $b = 0$, because you have no control over the random bit. On the first round where $b = 1$, the verifier confirms that $s = dlog(h) + x$. Since the verifier almost certainly knows dlog(h), he almost certainly knows x. We can make that probability arbitrarily high by increasing k.

Discrete-log Signature System

RSA cryptography fails if efficient methods are found for either factoring or the discrete log problem. There are other signature schemes that depend only on the hardness of the discrete log problem. Let's review one such scheme. We assume that the full text x has already been shortened to an MD5 hash value m.

1. Let a be a secret key known only to you, the signer. Let p be a large prime, and g be a generator of $\mathbb{Z}_p^*$. You can publish $(g, p, g^a (\mathrm{mod}\ p))$ as your public key to identify who you are to the world.

2. In order to sign m, choose a random w and compute c as the hash of
   $c = h(m^a (\mathrm{mod}\ p), m^w (\mathrm{mod}\ p), g^w (\mathrm{mod}\ p))$

3. Let $r = ca + w$, you publish the digital signature which is m together with
   $(r, m^a (\mathrm{mod}\ p), m^w (\mathrm{mod}\ p), g^w (\mathrm{mod}\ p))$

4. To check the signature, a verifier first computes c as the hash of the values
   $m^a (\mathrm{mod}\ p), m^w (\mathrm{mod}\ p), g^w (\mathrm{mod}\ p)$
   which were published with the signature. Then the verifier checks that
   $g^r (\mathrm{mod}\ p) = (g^a)^c \times (g^w) (\mathrm{mod}\ p)$ and
   $m^r (\mathrm{mod}\ p) = (m^a)^c \times (m^w) (\mathrm{mod}\ p)$

Your goal is to convince the verifier that you know what a is. This protocol is a form of "zero-knowledge proof". You must convince the verifier that you know a without giving it away. Instead you give away an r which depends on a but which doesnt help the verifier learn a because you have added a random value w to it, which will make its distribution be completely random.

You can safely tell the verifier $g^a (\mathrm{mod}\ p)$ and $g^w (\mathrm{mod}\ p)$ because discrete log is hard, and so those values dont help the verifier discover a or w. The value c is really a "challenge" to you to prove that you know a. It is computed from a "random" hash function and is out of your control. If you didnt know a then when you were challenged with a c, the value $(g^a)^c \times g^w (\mathrm{mod}\ p)$ could be any element of $\mathbb{Z}_p^*$. So trying to find r satisfying

$$g^r (\mathrm{mod}\ p) = (g^a)^c \times g^w (\mathrm{mod}\ p)$$

is a general instance of the discrete log problem which is very hard. So the tests on g establish that you know a and that you are who you claim to be. By similar reasoning, only the person who knows a could construct the powers of m that are published as the signature. Thus those tests establish that you deliberately signed that document.