1 Lecture 10: Applications of Zero-Knowledge Proofs

1.1 Sigma Protocols

Here, we discuss a specific type of Zero-Knowledge proof called a Sigma Protocol (Σ-Protocol), in which the only thing the verifier does is send random bits. The general format looks something like this:

1. P makes a statement or commitment of some information
2. V sends back a random bit b
3. depending on the value of b, P shows something. The value P shows may or may not reveal important information

The idea is that due to the bit b, V will be able to reject for a dishonest P with probability $1/2$. That is, one value of the bit will reveal information that will allow the verifier to immediately know the prover is lying if it is incorrect. As we will see in a moment, the other value does not need to expose a dishonest prover, but does not have to be completely useless.

1.1.1 Blum's Protocol

Blum's Protocol to prove that a graph is Hamiltonian (i.e., it has a Hamiltonian cycle) is an example of a Σ-Protocol.

An important fact we will use here is that a graph is Hamiltonian i.f.f. any $(n \times n)$ adjacency matrix A has some permutation $\sigma \in S_n$ such $\sigma(i) \neq i$, and all of the entries $A_{1,\sigma(1)}, \ldots, A_{n,\sigma(n)}$ are nonzero.

The protocol is as follows:

1. P generates and commits a permutation π on Hamiltonian graph G.
2. P generates and commits an adjacency matrix of the graph $H = \pi(G)$.
3. V generates and sends back a random bit b
   (a) If $b = 0$, then P opens π and H. This shows that $H \sim G$. (V can theoretically find a hamiltonian cycle when all of H is revealed, but this is as hard as finding a Hamiltonian cycle on G)
   (b) If $b = 1$, then P opens a hamiltonian cycle on the adjacency matrix of H. This shows that H is hamiltonian.

As usual, we immediately have perfect completeness. If P is dishonest, then either $H \not\sim G$, or G is not hamiltonian. In both cases, P has a $1/2$ chance of being caught, so after k trials, we will have soundness that is only broken with probability $1/2^k$.

Then, assuming the commitment is hiding, we may have the simulator do rewind-and-cut, which we should expect to have to do 2 times for each trial. Note that if the verifier is honest, we can simply figure out the bit b before. Then if $b = 1$ we send Hamiltonian $H \not\sim G$ if $b = 1$, and if $b = 0$ we send $H \sim G$ which we do not know to be Hamiltonian. So we should be able to show that the protocol is Zero-Knowledge.

1.1.2 Fiat-Shamir Heuristic

Suppose that we have a zero-knowledge proof which is also a Σ-protocol. That is, all that the verifier does is generate and send random bits, then accept or reject at the end.

Fiat and Shamir showed that we can turn any Σ-protocol into a non-interactive zero-knowledge proof, simply by having the prover compute and provide V's random bits on its own.

In order to guarantee that this prover is playing by the rules, we provide some random hash function that works as a next-message function to generate these random bits.

What we will end up with at the end is a non-interactive proof that demonstrates the claim is true, but which does not reveal any additional information.

We might prove that something like this is zero-knowledge by showing that a **PPT** simulator can simulate one of these (honest) one-party "conversations" without actually knowing any extra details about how the claim is true.

1.2 Non-Malleable Commitments

A malleable commitment is one which is still binding and hiding, but in which we can use $com(x)$ to produce $com(x')$, where x' is related to x in a particular way.

1.2.1 Motivation

Suppose we are running an online auction, and we do not want people to be able to cheat when committing their bids. This might be an issue if we use a malleable commitment scheme.

If A commits their bid x, and B intercepts $com(x)$, consider what might happen. If the commitment is malleable, this might allow B, without figuring out what x is, to send us $com(x')$, where $x' > x$, and win the bidding war every time.

1.2.2 Example

Pedersen's commitment protocol is an example of such a malleable commitment scheme. We work in the unit field of some (large) prime p, and we choose two generators g, h in this field.

Then, we commit $x \in (0, p)$ by generating a random $r \in (0, p)$ and sending $g^x h^r$. We open our commitment by sending x, r.

This has hiding and binding as strong as discrete log. Since g is a generator, there exists an a such that $h = g^a$. Finding a would be solving the discrete log problem given g, h. And, if we find $x, x', r, r' \in (0, p)$ such that $g^x h^r = g^{x'} h^{r'}$, then we have $x + ra \equiv x' + r'a \pmod{p-1}$. Since we know such an a exists if we have done this, we know we can solve $x - x' \equiv (r' - r)a$ for a, which solves the discrete log problem.

However, it is clear that given $com(x)$, we may commit $x + 1$ simply by sending $g \cdot com(x)$. So this protocol is malleable.

1.2.3 Fixing Malleability

ZKP's turn out to be the solution to our malleability problem. What we want when someone commits x is for them to provide a ZKP that they know what x is. This brings us back to "**Proof of Knowledge**", which we have mentioned before.

Formally, a <u>proof of knowledge</u> is an interactive proof that $x \in L$ if an extractor $E \in \mathbf{PPT}$, given the prover P (i.e. the source code) as input, can output a witness W to the fact that $x \in L$. The important thing here is that the prover may be infinitely powerful, but the extractor must be **PPT**.

1.3 Honest-but-Curious vs. Malicious

Generally, it is easy to make a protocol that works if everyone is "honest but curious", but often we run into problems if some player is malicious, and tries to break the rules in a way that doesn't interrupt the protocol but allows them to make the protocol work in their favor, possibly at the expense of others.

Again, since we are still talking about zero-knowledge, it would be a reasonable (and accurate) guess that a solution to this is to have each player provide a zero-knowledge proof that they are following the rules.

1.3.1 Example

One issue, which we have run into before while considering zero-knowledge, is the question of how we know that some player's "random" values are actually random. Our solution to this is something called the "coin flip into the well", where the person generating the randomness essentially "flips a coin" into a "well" , where only the receiver of the randomness can see it. A naive solution to this would be the following:

1. B flips a random bit r_B and commits $c(r_B)$ to A. This sets up the well and should indicate that B is playing honestly.
2. A sends a random bit r_A to B. This is like flipping the coin into the well.
3. B computes $r = r_A \oplus r_B$. This is like B looking at the coin in the bottom of the well.

It is easy to see that if everyone plays by the rules (i.e., everyone is honest but curious), then the protocol works as desired. However, we have an opportunity here for B to be malicious, since there is no way for A to know what r_B is, so B can make r_B, and thus r, whatever it wants. So, we fix it by doing the following:

1. B flips a random bit r_B and commits $c(r_B)$ to A.
2. A sends a random bit r_A to B
3. B computes $r = r_A \oplus r_B$
4. B sends $M = f(T, r)$, where T is the transcript up to this point, and f is the next-message function
5. B provides a zero-knowledge proof that there exists a decommitment d of $c(r_B)$ such that $M = f\left(T, r_A \oplus d \circ c(r_B)\right)$).

As it turns out, the statement being proven in the last step is **NP**, so a ZKP does exist, by what we showed previously using 3-coloring. And now, we can see that this protocol is even secure against a malicious B.

1.4 Identification

One final way that zero-knowledge proofs is for identification. If we must communicate a password or some other form of identification over an insecure connection, we do not want anyone who is eavesdropping to gain any information from our transcript. Of course, zero-knowledge proofs are the obvious choice, since we prove that they are secure by showing a listener who does not know anything could have generated the same transcript on their own!

What we can do is have the prover (the one logging in) and the verifier (the one checking the login) generate some instance of an **NP** problem, like GI (in practice, we do not use GI since we don't actually know what instances are secure, but since we haven't gone over much else, we use it as an illustrative example).

Then, whenever the prover wants to log in, the two of them generate a zero-knowledge proof of the previously decided-on fact. If the prover is able to convince the verifier, then the verifier lets the prover log in.

Note that this is a non-standard application of zero-knowledge, since the verifier actually knows that the statement is true. So we are not using zero-knowledge to hide extra information from the verifier, but to hide the extra information from any potential eavesdroppers who might want to try to pretend to be the prover.

1.5 Probabilistically Checkable Proofs (PCPs)

This is one final, tangentially-related topic. The idea is that we want a verifier to be able to verify or reject a proof by only checking some (constant-size) random sample of bits from the proof. The existence of probabilistically checkable proofs follows from the PCP theorem:

1.5.1 PCP Theorem

For any instance X of 3-SAT, there exists another instance X' such that:

1. X satisfiable $\implies$ X' satisfiable
2. X not satisfiable $\implies$ any assignment to variables of X' violates at least α of the clauses of X', for some constant α (i.e., $\alpha = \frac{3}{4}$).
3. $|X'| = poly(|X|)$

It follows from the PCP Theorem that in order to get a good constant level of soundness for our verifier, we only need to check a constant number of bits, based on our probability α of finding a clause that is not satisfied in X'. It turns out that if we want error probability $1/poly(|X|)$, we still only need to check $O(\log(|X|))$ bits.

Additionally, we do not necessarily want to send a massive proof, so we may apply Merkle Hash trees and simply send a hash of the proof.

1.5.2 Applying ZK

Of course, we don't want the verifier of a PCP gaining information about the proof itself, especially since it might run the proof many times. So, we apply zero-knowledge, together with the hashing mentioned above.

1. V sends a hash h for P to use
2. P commits the entire proof using the hash in Merkle Hash-tree style
3. V requests some bits/blocks from the proof
4. P provides a zero-knowledge proof of the statement: "Had P properly decommitted the requested blocks, V would have accepted the proof".

Of course, the alternative is to simply decommit the blocks, but this would obviously not be zero-knowledge, since the verifier learns what the blocks are and what the hash tree looks like.

As a side note to this, decommitment with Merkle trees involves sending the requested block, its sibling, and the siblings of all blocks along the path from the requested block to the final hash root of the tree.