CS 4501-6501 Topics in Cryptography February 18, 2015

Lecture 11

Lecturer: Mohammad Mahmoody *Scribe: Rainier Rabena*

1 Delegation of computation

Suppose we have some input x and we want to compute some function f on it, where f runs in time T. We want to delegate this computation to another entity. This entity, or the Prover, provides the solution $f(x)$ and a proof $\Pi(x)$ that the solution is correct. As the Verifier, we want to make sure that the Prover is not cheating. Note that the representation of f matters. For the purpose of this lecture, it helps to think of f as a piece of code (like a C or Python program) rather than a description of a circuit.

1.1 Aside: representation of the function f

Generally, there is a relation between the two forms that a function can take. If we have a program and we want to compute it on an input, it is possible to get a circuit which runs that program over that input. This is essentially what we have seen in the NP-completeness theorem of satisfiability: we can get a formula or circuit that simulates an algorithm. By thinking of f as a Turing machine or a program, the description of f becomes small, which means that it is not very costly to send a representation of f over to the Prover. If the code itself is very large and the description of the code is proportional to the time it takes to run it, then by sending this representation over to the Prover, we are already spending time that is proportional to T. Hence, for this lecture, f is described in short form. However, depending on what the protocol is going to be, we may eventually ask the Prover to "open up" the function f to get a circuit from it, perform some operation on that circuit, and send the answer to us.

1.2 Creating a protocol for delegation of computation

From the Verifier's point of view (who is just interested in knowing the answer to $f(x)$), f is going to be short, x is short, $f(x)$ is short, the computation process is delegated to the Prover and takes at least time T, and the Verifier verifies this answer in a time much smaller than T.

Recall from the previous lecture that, by Shamir's Theorem, whatever can be computed in polynomial space can also be proved interactively. If we want to get a short proof that a computation was done correctly ("short" referring to the number of bits in the total communication exchanged between the Prover and the Verifier) and we are willing to use interactive proofs, we can use an older version of the PCP theorem:

$$L \in NP \rightarrow \exists \text{ a randomized Verifier } \overline{Ver} \text{ such that it runs in } polylog(n)$$

$$x \in L \rightarrow \exists \text{ a proof } \Pi \text{ such that } \overline{Ver}^{\Pi} = 1$$

$$x \notin L \rightarrow \forall \text{ proofs } \Pi^*, Pr[\overline{Ver}^{\Pi^*(x)} = 0] \geq 1 - neg(x), n = |x|$$

The Verifier gets "oracle access" to Π in the sense that it can read every bit of Π, and Π is fixed. Π is not an interactive person who can answer the second question based on the first question; it is fixed ahead of time.

1.3 Perfect completeness

The Prover claims that x is in L and it is correct. Furthermore, he claims that there exists a well-written proof Π that the Verifier can access and accept with probability 1. If the Verifier cannot verify the proof, then he will reject with a probability extremely close to 1 no matter what proof the Prover uses, with negligible error. Because the running time of the verification is in $polylog(n)$ (or $log^k n$ for some constant k), the Verifier is not able to read more than $polylog(n)$ number of bits of the proof. The running time is small, and the number of bits that the Verifier can read is also small. If the verification is so efficient that it runs in polylog time, we won't have time to read all of the input x. The input itself should be written in some specific encoded version, not in the original form. The Verifier has random access to the input x; it can read any bit in constant time. Note that the Prover may try to cheat; that is, he may try to prove the wrong $f(x)$ to the Verifier.

What we want is a protocol between the Prover and the Verifier/Delegator that is similar to the following:

- The Verifier sends x and f to the Prover, where the representation of f is short
- The Prover sends back $f(x)$ and a short proof "Π"
- Possibly more steps, if the proof is an interactive proof

1.4 First try: using the PCP theorem

We want to create a protocol for delegation of computation. For our first try, we will use the PCP theorem. First, the Verifier asks the Prover to calculate $f(x)$. The Verifier also runs $\overline{Ver}^{\Pi}$ for "Π", where "Π" is a PCP proof for correctness of $f(x)$ for the language L. L is the language of $(x, f, f(x))$. By the PCP theorem, we can verify the correctness of $f(x)$ in polylog time. Whenever $\overline{Ver}$ wants some part of Π, it would ask it from the Prover.

There are issues with this first try. The Prover might choose Π^* after getting to know the queries of $\overline{Ver}$ to Π. By the property $x \notin L \rightarrow \forall$ proofs $\Pi^* Pr[\overline{Ver}^{\Pi^*(x)} = 0] \geq 1 - neg(x), n = |x|$, we can guarantee that there is no cheating written proof, but here we are not able to enforce that the proof is written because we are asking the Prover to give it to us, and he might not write it down before we ask for it. In other words, the Prover can choose the proof after he sees the queries of the proof. We want to force the Prover to commit something before we give the queries. One possible solution to this is to ask the Prover for some partial information about the proof so that we make it harder for him to change his mind. A hash function is ideal for this scenario.

1.5 Second try

For the second try, the Verifier asks some "hash" value of Π before sending the Prover the PCP queries.

- The inputs are x and the description of the function f.
- The Prover sends over $f(x)$ to the Verifier.
- The Verifier sends over the description of the hash function, or the hash function is universally fixed.
- The Prover sends the hash of Π, or $h(\Pi)$
- The Verifier sends the Prover some queries $\{q_1, q_2, ..., q_k\}$ where $k \leq polylog(n), n = |x|$
- The Prover sends back $\Pi(q_1), \Pi(q_2), ..., \Pi(q_k)$
- The Verifier runs $\overline{Ver}^{\Pi}(x)$ using the answers

Note that the length of the hash function is small: $|h(\Pi)| \leq polylog(|\Pi|) \leq polylog(n)$.

The issue with that second try is that we are not using the hash value in our protocol! We want to make sure that the Prover has a proof Π whose hash is $h(\Pi)$, and if we read some bits of it, it is consistent with the hash. In other words, the Prover must commit to something and cannot change his mind.

The Prover still has some degree of freedom because he is only giving a few bits of information about the proof. After committing to these few bits, there are still exponentially many proofs that are consistent with that hash function; there is no unique proof. (If one asks for 10 bits of information about an object which is 100 bits, there are still 90 bits degree of freedom to reconstruct the object back after we give the hash value.)

Our hope is that, if the hash function is algorithmically hard to break, it is computationally hard for the Prover to come up with a new proof. We cannot ask the Prover to give us all of the proof; we have to ask him to give us the hash. After he gives us the hash, if he is computationally unbounded, he is still able to choose his proof from an exponentially large set of proofs and some of them might be useful for him after he gets to see the queries. What we would like to accomplish is that after he gives us the hash value of the proof, because he is computationally bounded, he is basically committed to one proof and he cannot change his mind. We have to formalize what this means.

1.6 How can we verify the proof?

Our task is to ask the Prover to convince us that he has a proof Π that is consistent with the hash value $h(\Pi) = y$ and that $\Pi(q_1), ..., \Pi(q_k)$ is consistent with the answers sent. For this, we can use a tree structure for the hash function.

Assume we have a hash function h that takes 2α bits and hashes it down to α bits. We want to shrink a long string by a factor much greater than 2. In other words, we have a weak hash function and we want to make it stronger.

If have a string that is 4α bits long, we can split the string into two segments that are both 2α bits in length and apply the hash function on each of those, creating two hash values both of length α. Afterwards, we can concatenate those values together creating a string of length 2α and applying the hash function again. In other words, three applications of the hash function on the string of length 4α would give us α bits. We can apply this idea for many more rounds.

For a hash function to be considered good, it must be computationally

hard for someone to find two different values that hash to the same thing. After they give you the hash value, they can only find one possible string that hashes down. If this is a secure hash function, then this hash tree scheme is going to be secure.

If we have a string of length $\alpha \cdot 2^\ell$, we can have a hash tree that will give a hash of length α in ℓ levels and $2^\ell - 1$ hash applications. We assign this new hash function the symbol H. This is the hash function that we will use in the protocol. Note that $\alpha \approx polylog(n)$. If the string is not of length $\alpha \cdot 2^\ell$, we can just pad it with zeroes.

1.7 One extra step of verification

How can we do one more extra step of verification to make sure that the Prover actually did answer all these values of the proof based on the proof that he hashed it down? The Prover is giving us $\Pi(q_1), ..., \Pi(q_k)$. Suppose q_1 refers to the index of a bit value in the proof. This bit has some role in the hash value y of length α that the Prover gives us. We want to make sure that the Prover has done the hash function consistently. We can look at the path from the bit at q_1 to the root of the tree. The values of the hashes along the path are affected by that bit. We can ask the Prover to give us all the information that is affecting this path, and we can verify that the Prover has computed this path correctly.

To put in more formal words, we must perform this extra verification at the end: For all $i \in [k]$, look at q_i. Let P be the path of nodes in the hash tree from the block containing q_i to y. Let Q be the union of the set of nodes in P and the set of nodes connected to P.

For each of the queries q_i, we would ask the Prover to give us logarithmically many blocks, where each block is α bits long. The total communication remains polylog. If the Prover is honest, he should be able to provide us with all this information consistently. In this case, we would accept. Even though these partial checks only check a small amount of information (we are not checking everything about the proof; we are only checking the path), it is possible to prove that the Prover computationally is bound to some fixed valued ahead of time. How would we formalize something like this? (How can we prove that, if he can answer all of these checks consistently, it is as if he has decided on these values before we give him the queries?) The only thing that he has given us before is the α bits at the root of the tree. He provides everything else after we have given him the queries. This hash value is the only information he has given us about his whole proof Π. When we ask him something about these nodes, he would give us the

answer, and we can check the consistency of each path.

1.8 Collision-resistance of H

It is interesting to know that, if he can cheat and come up with two different proofs that are consistent with all these answers, then he can break the security of the hash function H in the sense that he can find at least two different proofs that hash to the same value. That is also possible to prove that if he can break the security of this bigger hash function H, he can actually find two inputs that hash to the same value for the hash function h.

We start with the assumption that this hash function is "collision-resistant", which means that it is computationally hard to find two different inputs that hash to the same value.

We can prove the security of our protocol as follows: If the Prover can find different answers to $q_1, ..., q_k$ that are consistent with the hash value $y = H(\Pi)$, we can also use this cheating Prover to find $x_1 \neq x_2$ where $h(x_1) = h(x_2)$. So if $h(\cdot)$ is collision-resistant, then the Prover cannot change his mind about $\Pi(q_1), ..., \Pi(q_k)$ after sending over y.

1.9 Final remarks on protocol

Using the collision-resistant hash function $h(\cdot)$, we can get a delegation scheme with polylog complexity for the Verifier/Delegator, but with four messages. This protocol lets us prove anything in NP with polylogarithmic computation on the Verifier. The security in this protocol is purely computational; we are assuming that the hash function is collision-resistant.