COS 487: Theory of Computation **Fall 2000**

Handout 5: December 6

Sanjeev Arora

5.1 Some hard number theoretic problems

1. Let p be a prime, and g be an integer that is a generator for p, which means that for every number y that is not divisible by p, there is a $t \in \{1, \ldots, p-1\}$ such that $y \equiv g^t (\mathrm{mod} p)$. Then t is called the *discrete logarithm* of y with respect to g. Finding this for a randomly chosen y seems hard.

2. Let $n = pq$ where p, q are primes and $p, q \equiv 3(\mathrm{mod} 4)$.

   (a) For every integer a, the number a^2 has four "square roots" modulo n. Two are $a, n-a$ and the other two are obtained by the use of chinese remainder theorem. If somebody randomly picks a and gives us $a^2(\mathrm{mod} n)$ then finding a square root is as hard as factoring n. Reason: If we have an algorithm then we can factor n by giving it $a^2(\mathrm{mod} n)$ for a random a. The algorithm does not know what a is; it just finds some square root r. With probability at least $1/2$ this square root is neither a nor $n-a$. In that case $r^2 \equiv a^2(\mathrm{mod} n)$, i.e., $(r+a)(r-a) \equiv 0(\mathrm{mod} n)$, and neither $r + a \equiv 0(\mathrm{mod} n)$ nor $r - a \equiv 0(\mathrm{mod} n)$. Hence exactly one of p, q must divide $r + a$ and so we can extract it by computing the greatest common divisor of $r + a$ and n.

   (b) If e is a number that is coprime to $(p-1)(q-1)$ then for a random chosen x it seems hard to find x given $x^e(\mathrm{mod} n)$. This is the famous RSA encryption function. We can easily find an integer n whose factorization we know, and take a random e and publish e it as our *public key*. Then since we know the factorization of n we know $(p-1)(q-1)$ and so we can find an integer d such that $de \equiv 1(\mathrm{mod}(p-1)(q-1))$. This will be our *private* key. Anybody who wishes to send us an encrypted message can break up the message into chunks of $\lfloor \log n \rfloor$ bits —namely, numbers mod n— and send each raised to power e. Eavesdroppers who do not know the factorization of n will have a hard time decrypting. We can decrypt by raising the received message to the power d. This is a valid decryption since $x^{de} \equiv x(\mathrm{mod} n)$, as is checked by noticing that $x^{(p-1)(q-1)} \equiv 1$.

5.2 Zero Knowledge proofs

Graph Nonisomorphism Input: Pair of graphs (G_1, G_2).

The verifier randomly picks one of the two graphs, and randomly permutes its vertices to get a graph H. It asks the prover to identify which of the two graphs is isomorphic to H. If the prover identifies correctly, the verifier accepts.

The prover is all-powerful, but does not know what random bits were used by the verifier to produce H. All he sees is a random-looking graph. However, if G_1, G_2 are indeed nonisomorphic, H is isomorphic to only one of them and the prover can detect which one (using his huge computational power). Thus he can make the verifier accept. Suppose on the other hand that G_1, G_2 are isomorphic. The prover has only a $1/2$ chance of guessing which one the verifier had in mind, and hence has only a $1/2$ chance of making the verifier accept.

Why is this zero knowledge? Intuitively, the verifier does not learn anything new during the protocol, except some confidence that the graphs are indeed nonisomorphic. We can formalize this by showing that if the graphs are indeed nonisomorphic, the transcript of the protocol can be generated by a probabilistic polynomial time machine. (This shows that any "knowledge" acquired in the protocol is of no use since a probabilistic polynomial time computation would also provide this knowledge.) Note that the machine can produce the transcript in any order it likes, in particular, outputting the last step first. Think about it!

Square roots modulo $n = pq$ In the description below all numbers are $\bmod n$. The input is a^2. The prover wishes to prove to the verifier in a zero-knowledge way that he knows a. The prover picks a random number $r \bmod n$ and sends the verifier r^2. The verifier picks a random bit b and sends it to the prover. If b is 0 the prover has to provide r; the verifier checks that its square is the number r^2 provided earlier and if so it accepts. If b is 1 the prover has to provide ra; the verifier squares this and checks that the square is r^2a^2 which it knows from before.

Again, a simulation argument shows that the protocol is zeroknowledge and the verifier does not learn anything. We show that a probabilistic polynomial time simulator generates "transcripts" of the protocol. First pick a random number x. Then pick the verifier's random bit b. If b is 0, then use x as r in the protocol. If $b = 1$ then use x as ra. In other words, put x^2/a^2 as r^2 in the protocol and then fills in the rest of the transcript.

This protocol is different from the graph nonisomorphism protocol in that the prover does not need to be all-powerful; it can run in polynomial time if it knows a. Thus this protocol can be used in password authentication for rlogins. Each user is provided with a random a as his password. He can identify himself by revealing a^2 and then proving in zero knowledge that he holds a. A polynomial-time eavesdropper will not learn *anything* about a. Note that this is a stronger guarantee than standard encryption, where the eavesdropper could learn something about a. This and related protocols are widely used these days.