6.080 / 6.089 Great Ideas in Theoretical Computer Science
Spring 2008

6.080/6.089 GITCS Apr 15, 2008

Lecture 17

Lecturer: Scott Aaronson *Scribe: Adam Rogal*

1 Recap

1.1 Pseudorandom Generators

We will begin with a recap of pseudorandom generators (PRGs). As we discussed before a pseudorandom generator is a function that takes as input a short truly random input string and produces an output of a seemingly random string. Formally, a PRG is a polytime-computable function $f : \{0,1\}^n \to \{0,1\}^{n+1}$ such that for all deterministic polynomial-time algorithms A,

$$\left| \Pr_{y \in \{0,1\}^{n+1}} [A(y) \text{ accepts}] - \Pr_{x \in \{0,1\}^n} [A(f(x)) \text{ accepts}] \right|$$

is negligible.

Given a PRG that stretches n bits to $n+1$ bits, we can create a PRG that stretches n bits to $p(n)$ bits for any polynomial p. To do so, we repeatedly break off a single bit of the PRG's output, and feeding the remaining n bits back into the PRG to get another $n+1$ pseudorandom bits. This process is shown in figure 1. To prove that it works, one needs to show that, could we distinguish the $p(n)$-bit output from random, we could *also* distinguish the original $(n+1)$-bit output from random, thereby violating the assumption that we started with a PRG. Formalizing this intuition is somewhat tricky and will not be done here.

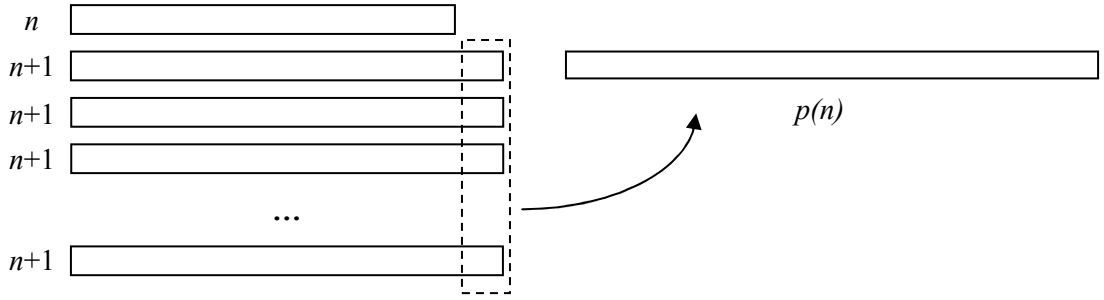


Figure 1: A seemingly random string of size $p(n)$ is generated from an n-bit seed using the feed and repeat method.

1.2 Cryptographic Codes

Using pseudorandom generators, it's possible to create secure cryptographic codes with small key sizes. The details of this are complicated if you want to protect against realistic

attacks (for example, so-called *chosen-message attacks*). But at the simplest level, the intuition is the following: we should be able to simulate a one-time pad (which is provably unbreakable when used correctly) by (1) taking a small random key, (2) stretching it to a longer key using a PRG, and then (3) treating that longer key as the one-time pad. If a polynomial-time adversary could break such a system, that would mean that the adversary was distinguishing the PRG's output from a truly random string, contrary to assumption.

1.3 One-Way Functions

In addition to PRGs, we'll be interested in a closely-related class of objects called OWFs, or *one-way functions*. An OWF is a polytime-computable function $f : \{0,1\}^n \rightarrow \{0,1\}^{p(n)}$ such that for all deterministic polynomial-time algorithms A,

$$\Pr_{x \in \{0,1\}^n}[f(A(f(x))) = f(x)]$$

is negligible.

Or in plainer language, an OWF is a function that's easy to compute but hard to invert.

1.4 Yao's Minimax Principle

As a side note, you might wonder why we assumed the adversary A was determinisic rather than probabilistic. The answer is that it makes no difference! If you're playing rock-paper-scissors, and you *know* the probability distribution over your opponent's move, then there's always some *fixed* move you can make that does as well as any randomized strategy. Similarly, one you fix the probability distribution over inputs – as we do with PRGs and OWFs – there's always a deterministic algorithm whose success probability is as large as any randomized algorithm's. This is (the easy part of) *Yao's Minimax Principle*, one of the most useful facts in theoretical computer science.

1.5 Relation Between PRGs and OWFs

Claim: Every PRG is also an OWF. Why? Because if we could invert a PRG, then it wouldn't be pseudorandom! We'd learn that there was *some* seed that generated the output string, which would be true for a random string with probability at most 1/2.

In 1997, Håstad et al. proved the opposite direction: if OWFs exist then so do PRGs. This direction was much, *much* harder (note that transformations of the OWF are necessary, since it's easy to give examples of OWFs that are not PRGs). Because of this result, we now know that the possibility of private-key encryption with small keys is essentially equivalent to the existence of OWFs.

2 Public-Key Cryptography

2.1 Abstract Problem

Suppose Alice is trying to send Bob a package, so that no third party can open it *en route*. We'll assume that boxes can be "locked," in such a way that you can only open a box if you have the right key.

If Alice and Bob share duplicates of the same key, then this problem is trivial: Alice locks the box with her key and sends it to Bob, who then opens it with his key. But what

if Alice and Bob *don't* share a key? Obviously, we don't want Alice to send the package in a locked box, and the key that opens the lock in an unlocked box! We seem to be faced with an infinite regress.

Fortunately, there's a simple solution. As shown in Figure 2, first Alice puts the package in a box, locks it, and sends it to Bob. Then Bob puts a *second* lock on the box and sends it back to Alice. Then Alice removes her lock and sends the box back to Bob. Finally Bob removes his lock and opens the box.

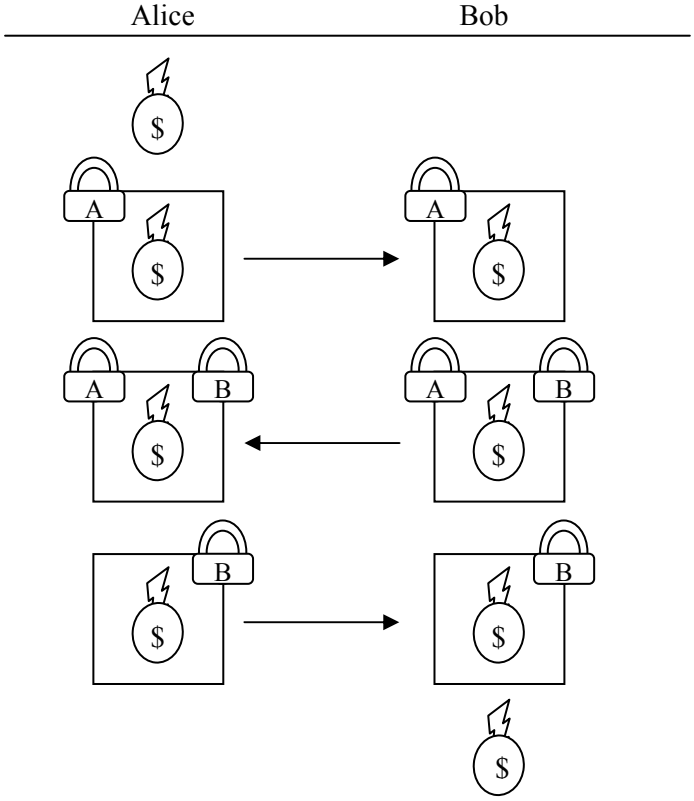
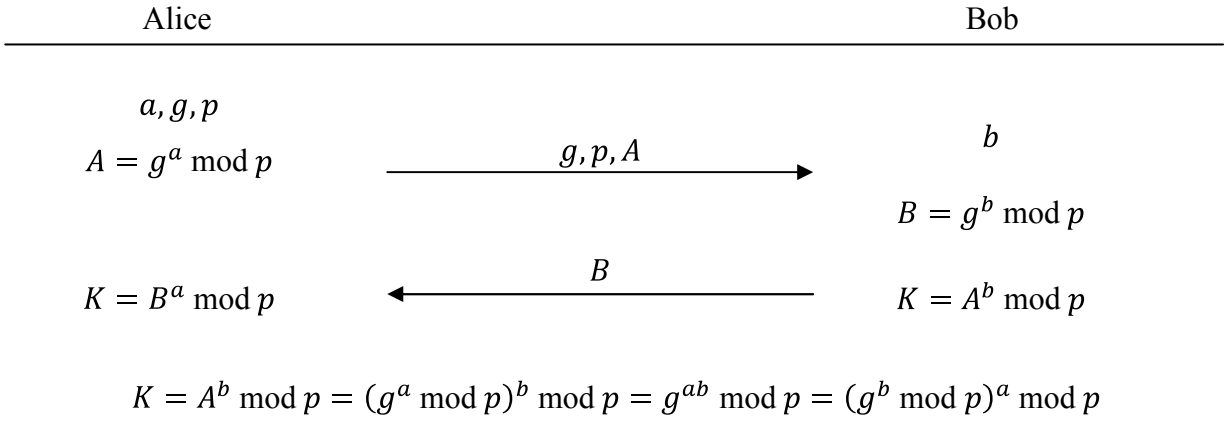


Figure 2: The smarter approach has Alice and Bob passing the package with at least one form of protection at all times. This ensures that only Alice and Bob will be able to open the package.

2.2 Diffie-Hellman

How could we simulate the above protocol, in the situation where Alice and Bob are sending bits of information rather than physical boxes? The first serious proposal in the open literature for how to do this was given by Diffie and Hellman in 1976.



$$K = A^b \bmod p = (g^a \bmod p)^b \bmod p = g^{ab} \bmod p = (g^b \bmod p)^a \bmod p$$

Figure 3: The Diffie-Hellman protocol for creating a shared secret key K between Alice and Bob.

The process, shown in figure 3, begins by Alice choosing a large prime number, p, a base, g, and a secret integer, a. Alice will calculate a public number $A = g^a \bmod p$. She will then send p, g, and A to Bob. Bob will then pick his own secret b, and send $B = g^b \bmod p$ back to Alice. Finally, Alice calculates the secret key K as $K = B^a \bmod p$, and Bob calculates it as $K = A^b \bmod p$. They both now have the same key with which to encode messages to each other.

We've seen that Diffie-Hellman is a simple way to exchange a key; yet, but it's a bit cumbersome in practice. What we'd really like is a public-key protocol that involves fewer messages back and forth—and in which only one person, not two, needs to create public and private keys.

3 RSA

RSA (together with its variants) is probably the most widely-used cryptographic protocol in modern electronic commerce. Much like Diffie-Hellman, it is built on modular arithmetic.

3.1 How It Works

As shown in Figure 4, the process is more direct than with Diffie-Hellman. Let's suppose you want to send your credit card number to Amazon.com. Then in the simplest variant, Amazon picks two large prime numbers, p and q, with the condition that neither $p-1$ nor $q-1$ is divisible by 3. It then multiplies them together to get $N = pq$ and sends N to you. On retrieving N, you calculate $y = x^3 \bmod N$, where x is your credit card number, and send y back to Amazon.

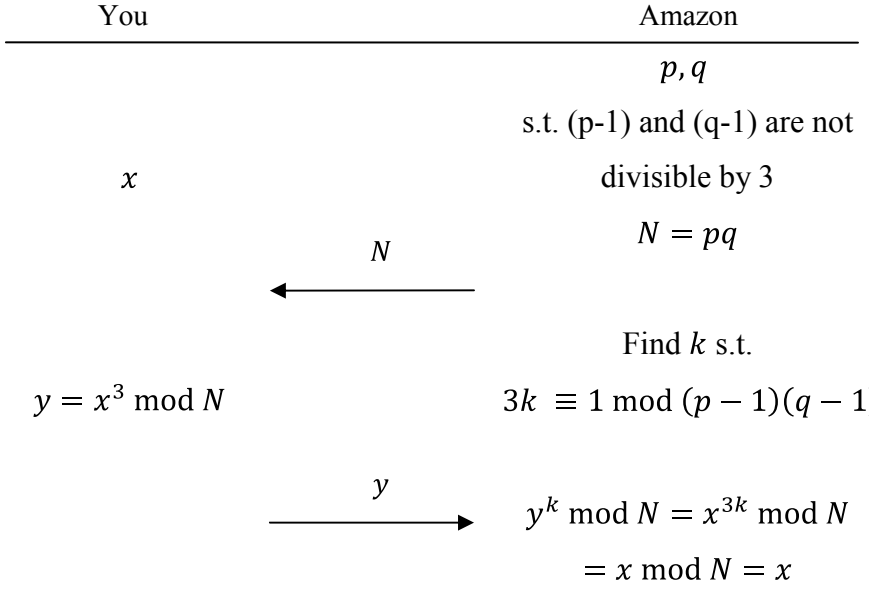


Figure 4: RSA uses modular arithmetic to retrieve x efficiently from an encoded message. An eavesdropper will only see N and $x^3 \bmod N$.

Amazon then faces the problem of how to recover x given y. In other words, how does it take a *cube root* modulo N? Fortunately, it can do that given using its knowledge of the prime factors p and q, together with the following formula discovered by the mathematician Leonhard Euler in the 1700's:

$$x^{(p-1)(q-1)} = 1 \bmod N$$

(Why is this formula true? Basically, because $(p-1)(q-1)$ is the order of the *multiplicative group* mod N, consisting of all numbers from 1 to N that are relatively prime to N. We won't give a more detailed proof here.)

Euler's formula implies that, if Amazon can only find an integer k such that $3k = 1 \bmod (p-1)(q-1)$, then

$$y^k = x^{3k} = x^{c(p-1)(q-1)+1} = x \bmod N,$$

where c is some integer. But the fact that neither $p-1$ nor $q-1$ is divisible by 3 implies that such an integer k must exist – and furthermore k can be found in polynomial time given p and q, for example by using Euclid's algorithm. And once Amazon has k, it can also compute $y^k \bmod N = x$ in polynomial time using repeated squaring. It can thereby recover your credit card number x, as desired.

The obvious question is, how secure is this system? Well, any adversary who could factor N into pq could obviously decrypt the message x, by using the same algorithm that Amazon itself uses. Hence this whole system is predicated on the presumed intractability of factoring large integers (an assumption that would be violated if, for example, we built large-scale quantum computers). And of course, any proof that factoring is hard would also prove $P \neq NP$.

In the other direction, you might wonder: *assuming* the factoring problem is hard, is RSA secure? Alas, that's been an open problem for 30 years! Yet despite its uncertain theoretical foundations, the RSA system has withstood all attacks thus far (unlike many other proposed cryptosystems), and today millions of people rely on it.