6.080 / 6.089 Great Ideas in Theoretical Computer Science
Spring 2008

6.080/6.089 GITCS	April 4th, 2008
Lecture 16	
Lecturer: Scott Aaronson	*Scribe: Jason Furtado*

Private-Key Cryptography

1 Recap

1.1 Derandomization

In the last six years, there have been some spectacular discoveries of deterministic algorithms, for problems for which the only similarly-efficient solutions that were known previously required randomness. The two most famous examples are

- the Agrawal-Kayal-Saxena (AKS) algorithm for determining if a number is prime or composite in deterministic polynomial time, and
- the algorithm of Reingold for getting out of a maze (that is, solving the undirected s-t connectivity problem) in deterministic LOGSPACE.

Beyond these specific examples, mounting evidence has convinced almost all theoretical computer scientists of the following
Conjecture: Every randomized algorithm can be simulated by a deterministic algorithm with at most polynomial slowdown. Formally, $P = BPP$.

1.2 Cryptographic Codes

1.2.1 Caesar Cipher

In this method, a plaintext message is converted to a ciphertext by simply adding 3 to each letter, wrapping around to A after you reach Z. This method is breakable by hand.

1.2.2 One-Time Pad

The "one-time pad" uses a random key that must be as long as the message we want to encrypt. The exclusive-or operation is performed on each bit of the message and key ($Msg \oplus Key = EncryptedMsg$) to end up with an encrypted message. The encrypted message can be decrypted by performing the same operation on the encrypted message and the key to retrieve the message ($EncryptedMsg \oplus Key = Msg$). An adversary that intercepts the encrypted message will be unable to decrypt it as long as the key is truly random.

The one-time pad was the first example of a cryptographic code that can *proven* to be secure, even if the adversary has all the computation time in the universe.

The main drawback of this method is that keys can never be reused, and the key must be the same size as the message to encrypt. If you were to use the same key twice, an eavesdropper could compute $(Enc \oplus Msg1) \oplus (Enc \oplus Msg2) = Msg1 \oplus Msg2$. This would leak information about $Msg1$ and $Msg2$.

Example. Suppose $Msg1$ and $Msg2$ were bitmaps and $Msg1$ had sections that were all the same (say, a plain white background). For simplicity, assume $Msg1$ is all zeros at bit positions 251-855. Then $Msg2$ will show through in those bit positions. During the Cold War, spies were actually caught using this sort of technique.

Also, note that the sender and the recipient must agree on the key in advance. Having shared random keys available for every possible message size is often not practical. Can we create encryption methods that are secure with smaller keys, by assuming our adversary doesn't have unlimited computing power (say, is restricted to running polynomial-time algorithms)?

2 Pseudorandom Generators

A pseudorandom generator (PRG) is a function that takes as input a short, truly random string (called the *seed*) and produces as output a long, seemingly random string.

2.1 Seed Generation

A seed is a "truly" random string used as input to a PRG. How do you get truly random numbers? Some seeds used are generated from the system time, typing on a keyboard randomly, the last digits of stock prices, or mouse movements. There are subtle correlations in these sources so they aren't completely random, but there are ways of extracting randomness from weak random sources. For example, according to some powerful recent results, nearly "pure" randomness can often be extracted from two or more weak random sources that are assumed to be uncorrelated with each other.

How do you prove that a sequence of numbers is random? Well, it's much easier to give overwhelming evidence that a sequence is *not* random! In general, one does this by finding a *pattern* in the sequence, i.e. a computable description with fewer bits than the sequence itself. (In other words, by showing that the sequence has less-than-maximal Kolmogorov complexity.)

In this lecture, we'll simply assume that we have a short random seed, and consider the problem of how to expand it into a long "random-looking" sequence.

2.2 How to Expand Random Numbers

2.2.1 Linear-Congruential Generator

In most programming languages, if you ask for random numbers what you get will be something like the following (starting from integers a, b, and N):

$x_1 = ax_0 + b \bmod N$ $x_2 = ax_1 + b \bmod N$

...

$x_n = ax_{n-1} + b \bmod N$

This process is good enough for many non-cryptographic applications, but an adversary could easily distinguish the sequence $x_0, x_1, \ldots$ from random by solving a small system of equations mod N. For cryptography applications, it must not be possible for an adversary to figure out a pattern in the output of the generator in polynomial time. Otherwise, the system is not secure.

2.2.2 Cryptographic Pseudorandom Generator (CPRG)

Definition: (Yao 1982)

A cryptographic pseudorandom generator (CPRG) is a function $f : \{0,1\}^n \to \{0,1\}^{n+1}$ such that:

1. f is computable in polynomial time.
2. For all polynomial-time algorithms A (adversaries),

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

the "advantage", is negligibly small.

In other words, the output of the CPRG must "look random" to any polynomial time algorithm.

In the above definition, "negligibly small" means less than $1/p(n)$ for all polynomials p. This is a minimal requirement, since if the advantage of the adversary were $1/p(n)$, then in polynomial time the adversary could amplify the advantage to a constant (see Lecture 14). Of course it's even better if the adversary's advantage decreases exponentially.

The definition above only requires f to stretch an n-bit seed into a random-looking $(n+1)$-bit string. Could we use such an f to stretch an n-bit seed into, say, a random-looking n^2-bit string? It turns out that the answer is yes; basically we feed f its own output n^2 times. (See Lecture 17 for more details.)

2.2.3 Enhanced One-Time Pad

Using such a CPRG $f : \{0,1\}^n \to \{0,1\}^{p(n)}$, we can make our one-time pad work for messages polynomially larger than the original key s:

$k = f(s)$
$e = x \oplus k$
$x = e \oplus k$

Claim. With this construction, no polynomial-time adversary can recover the plaintext from the ciphertext.

Proof. Assume for simplicity that the plaintext consists of just a single repeated random bit (i.e., is either $00\cdots0$ or $11\cdots1$, both with equal probability). Also, suppose by way of contradiction that a polynomial-time adversary could guess the plaintext given the ciphertext, with probability non-negligibly greater than 1/2. We know that if the key k were truly random, then the adversary would *not* be able to guess the plaintext with probability greater than 1/2. But this means that the adversary must be distinguishing a pseudorandom key from a truly random key with non-negligible bias – thereby violating the assumption that f was a CPRG!

The system above is not yet a secure cryptographic system (we still need to deal with the issue of repeated keys, etc.), but hopefully this gives some idea of how CPRG's can be used to construct computationally-secure cryptosystems.

3 Blum-Blum-Shub CPRG

The Blum-Blum-Shub (BBS) CPRG is proven to breakable if and only if a fast (polynomial-time) algorithm exists for factoring. With this generator, the seed consists of integers x and $N = pq$, where p, q are large primes. The output consists of the last bit of $x^2 \bmod N$, the last bit of $(x^2)^2 \bmod N$, the last bit of $((x^2)^2)^2 \bmod N$, etc.

4 $P \neq NP$-based CPRG

Ideally, we would like to construct a CPRG or cryptosystem whose security was based on an NP-complete problem. Unfortunately, NP-complete problems are always about the worst case. In cryptography, this would translate to a statement like "there *exists* a message that's hard to decode", which is not a good guarantee for a cryptographic system! A message should be hard to decrypt with overwhelming probability. Despite decades of effort, no way has yet been discovered to relate worst case to average case for NP-complete problems. And this is why, if we want computationally-secure cryptosystems, we need to make stronger assumptions than $P \neq NP$.

5 One-Way Functions

The existence of one-way functions (OWF's) is such a stronger assumption.

Definition: A one-way function is a function $f : \{0,1\}^n \to \{0,1\}^{p(n)}$ such that:

1. f is computable in polynomial time.
2. For all polynomial-time algorithmss A,

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

is negligible.

In other words, a polynomial-time algorithm should only be able to invert f with negligible probability. The reason we don't require $A(f(x)) = x$ is to rule out trivial "one-way functions" like $f(x) = 1$.

$CPRG \Rightarrow OWF$?

True. Any CPRG is also an OWF by the following argument: if given the output of a pseudorandom generator we could efficiently find the seed, then we'd be distinguishing the output from true randomness – thereby violating the assumption that we had a CPRG in the first place.

$OWF \Longleftrightarrow CPRG$?

Also true, but this direction took over 20 years to prove! In 1997, Håstad, Impagliazzo, Levin, and Luby showed how to construct a pseudorandom generator from any one-way function, by a complicated reduction with many steps.