A Mini-Course on the Theory of Cryptography

Charles Rackoff

Department of Computer Science, University of Toronto

Bonn, June 17-21, 2006

(Part 2)

Review

We showed that a number generator is pseudo-random
if and only if
it is pseudo-random against multiple sampling
if and only if
it is unpredictable.

Do pseudo-random number generators exist?

Theorem: If $\mathbf{P} = \mathbf{NP}$, then there is no pseudo-random generator.

Proof: Assume $\mathbf{P} = \mathbf{NP}$. Let G be a pseudo-random number generator with length function $\ell(n) > n$.

Since $\mathbf{P} = \mathbf{NP}$, there is a polynomial time algorithm D which on inputs α, accepts if and only if there is an n-bit string s such that $G(s) = \alpha$.

So $p_D(n)$, the probability D accepts $G(s)$ for random n-bit s, is 1.

Since there are only 2^n strings of length n, we have that $r(n)$, the probability D accepts a random $\ell(n)$ bit string, is $\leq \frac{2^n}{2^{\ell(n)}} \leq \frac{1}{2}$.

So $|p_D(n) - r_D(n)| \geq \frac{1}{2} > \frac{1}{n^1}$.

Remark: It appears that $\mathbf{P} \neq \mathbf{NP}$ is *not* sufficient to prove the existence of pseudo-random generators. To prove the existence of pseudo-random generators, we need a strong kind of *average case* version of $\mathbf{P} \neq \mathbf{NP}$.

Let G be a pseudo-random number generator with expansion function $\ell(n) = e(n) + n$, $e(n) \geq 1$. How can we create a new pseudo-random number generator with *longer* expansion?

Let G be a pseudo-random number generator with expansion function $\ell(n) = e(n) + n$, $e(n) \geq 1$. How can we create a new pseudo-random number generator with *longer* expansion?

Define $G'(s_0) = \alpha_1\alpha_2\alpha_3\ldots$ where for $i \geq 1$, $G(s_{i-1}) = \alpha_i s_i$.

Formally, for this to be a number generator, we have to limit the output to some polynomial $t(n)$ steps: $G'(s_0) = \alpha_1\alpha_2\ldots\alpha_{t(n)}s_{t(n)}$.

Theorem: If G is pseudo-random, then G' is pseudo random.

Proof (outline): Let D' be a distinguisher that breaks the pseudo-randomness of G'. Fix n, and consider the following hybrid experiments for $0 \leq j \leq t(n)$:

Experiment j: $\alpha_1, \alpha_2, \cdots, \alpha_j$ are randomly chosen strings from $\{0,1\}^{e(n)}$, and s_j is randomly chosen from $\{0,1\}^n$, and D' is given as input $[\alpha_1\alpha_2\cdots\alpha_{t(n)}s_{t(n)}]$ where for $i > j$, $G(s_{i-1}) = \alpha_i s_i$.

THEN q_j is the probability that D' accepts.

$q_0 = p_{D'}(n)$ and $q_{t(n)} = r_{D'}(n)$. Say that $q_{j+1} - q_j > \varepsilon$.

To break the pseudo-randomness of G:
Given αs,
form, and give to D', the string $[\alpha_1 \alpha_2 \cdots \alpha_{t(n)} s_{t(n)}]$ where
$\alpha_1 \ldots \alpha_j$ are chosen randomly, $\alpha_{j+1} = \alpha$, $s_{j+1} = s$,
and for $i > j+1$, $G(s_{i-1}) = \alpha_i s_i$.

If αs was randomly generated, D' accepts with probability q_{j+1}.
If αs was pseudo-randomly generated, D' accepts with probability q_j.

Theorem: There is a pseudo-random number generator with length function $\ell(n) = n + 1 \iff$ for every d there is a pseudo-random number generator with length function $\ell(n) = n^d$.

Background: Most of the above work was done by Blum, Micali, and Yao in the early eighties.

In 1985, Goldwasser, Goldreich, and Micali asked: Can we have a pseudo-random generator that, on an n bit seed, outputs *exponentially* many bits?
How can we even define this, given that both good guys and bad guys have to work in polynomial time?

Background: Most of the above work was done by Blum, Micali, and Yao in the early eighties.

In 1985, Goldwasser, Goldreich, and Micali asked: Can we have a pseudo-random generator that, on an n bit seed, outputs *exponentially* many bits?
How can we even define this, given that both good guys and bad guys have to work in polynomial time?

Solution: "random access" to the bits. The good guy must be able to generate any desired bit efficiently, given its index.
The bad guy can see any bits he wishes to (interactively), but must work in time polynomial in the length of the seed.

Typically we view the exponentially long string as a *function.*
We now define *pseudo-random function generator.* Quite arbitrarily, on a seed of length n, the function generated will be a function from n-bit strings to n-bit strings.

Definitions: A *Function Generator* F associates with each $n \in \mathbb{N}$ and each $s \in \{0,1\}^n$ a function $F_s : \{0,1\}^n \rightarrow \{0,1\}^n$, such that there is a polynomial time algorithm which given $s \in \{0,1\}^n$ and $x \in \{0,1\}^n$, computes $F_s(x)$. F is *pseudo-random* if the following holds for every (prob. polytime, or poly size) D:

D is given 1^n and access to an "oracle" for a function $f : \{0,1\}^n \rightarrow \{0,1\}^n$. It makes queries to f and sees answers, and eventually accepts or rejects. Let:

$p_D(n) =$ the probability that if s is randomly chosen from $\{0,1\}^n$ and D is given F_s, then D accepts.
$r_D(n) =$ the probability that if f is randomly chosen from amongst all functions from $\{0,1\}^n$ to $\{0,1\}^n$ and D is given f, then D accepts.

THEN for every c and sufficiently large n,
$|p_D(n) - r_D(n)| \leq 1/(n^c)$.

Given a pseudo-random number generator G, how can we construct a pseudo-random function generator F?

Assume that the length function for G is $\ell(n) = 2n$.
Say that we are given an n-bit seed s for F.
Consider the complete binary tree of depth n.
Identify each node with a binary string α of length $\leq n$;
we want to label each node with an n-bit string.
Label the root λ (the empty string) with the seed $s_\lambda \leftarrow s$.
For each internal node α labeled with string s_α, label the two children of α with $s_{\alpha 0}$ and $s_{\alpha 1}$ where $G(s_\alpha) = s_{\alpha 0} s_{\alpha 1}$.

The leaves of the tree contain the function F_s, that is, for every n-bit α, $F_s(\alpha) = s_\alpha$.

Let D be a distinguisher that breaks the pseudo-randomness of F.
Fix n, and consider the following hybrid experiments for $0 \leq j \leq n$:

Experiment j: Randomly label all the nodes from the root through level j; label the rest as in the construction.
THEN q_j is the probability that D accepts.

$q_0 = p_D(n)$ and $q_n = r_D(n)$. Say that $q_{j+1} - q_j > \varepsilon$.

To break the pseudo-randomness of G with multiple sampling:

Run D, viewing it a querying leaves of subtrees rooted at level j; whenever D queries a new subtree, we put at level 1 of that subtree a new ($2n$-bit) sample.

We accept randomly generated samples with probability q_{j+1};
We accept pseudo-randomly generated samples with probability q_j.

Theorem: There exist pseudo-random number generators $\Longleftrightarrow$ for every d_1, d_2 there exist pseudo-random function generators that, on a seed of length n, map n^{d_1}-bit strings to n^{d_2}-bit strings.

Proof: For $\Leftarrow$, let $G(s) = F_s(\bar{0})F_s(\bar{1})\ldots$.

For $\Rightarrow$ use the previous construction. Either first pseudo-randomly expand the seed and then use G on that expanded seed, or (better) do the construction to a greater depth.

The second construction is preferable, since it doesn't require the given generator G to be run on larger seeds than that of F. In practice, we do not construct pseudo-random function generators from (supposed) pseudo-random number generators or from one-way functions, but rather we construct them from "scratch". And they are not "scalable", but we we only have them for one (or two or three) seed lengths.

Most important example: AES (advanced encryption system) is actually a collection of three generators, with key lengths 128, 192, or 256 bits; each key generates a function : $\{0,1\}^{128} \to \{0,1\}^{128}$.

We hope they are “pseudo-random” in the sense that a “feasible” distinguisher cannot “effectively” tell the difference between a randomly generated function and a pseudo-randomly generated function.

What if we want, for example, to use AES to construct an (instance of) a pseudo-random function generator that maps 256-bit strings to 256-bit strings?

More formally, given a pseudo-random function generator (as defined above) F, we want to construct a new one F' that on an n bit seed, maps $2n$ bits to $2n$ bits; furthermore, F' on an n bit seed should only use F on n bit seeds. Any ideas?

Construction 1: Use the idea of the above tree construction, and view F as generating a tree with 2^n leaves with the input specifying a path to a leaf; then use each leaf to also generate a tree with 2^n leaves to get a big tree with 2^{2n} leaves. That is:
$F'_s(xy) = F_{F_s(x)}(y)$ (where $|x| = |y| = |s|$).

Unfortunately, F'_s maps $2n$ bits to only n bits, but we can fix this by letting:
$F''_{s_1 s_2}(xy) = F'_{s_1}(xy)F'_{s_2}(xy)$.

Unfortunately, F'' uses a $2n$-bit seed, but we can fix this by letting:
$F'''_s(xy) = F''_{G(s)}(xy)$ where G is a length doubling pseudo-random number generator. For example, we can use
$G(s) = F_s(\overline{0})F_s(\overline{1})$.

Note: This and our other constructions generalize to polynomial lengths.

Construction 2: Use "cipher-block chaining". Define:
$F'_s(xy) = F_s(F_s(x) \oplus y)$. This is secure.

Proof: Somewhat messy. Hard part:
Lemma: For an arbitrary function $F : \{0,1\}^n \to \{0,1\}^n$, define $F' : \{0,1\}^{2n} \to \{0,1\}^n$ by $F'(xy) = F(F(x) \oplus y)$.
Then even a computationally unbounded adversary cannot significantly distinguish between a randomly chosen $f : \{0,1\}^{2n} \to \{0,1\}^n$, and F' for a randomly chosen $F : \{0,1\}^n \to \{0,1\}^n$.

More generally, we want to define a function generator with *unbounded inputs*, that is, $F_s : \{0,1\}^* \rightarrow \{0,1\}^n$ where $|s| = n$. We define this kind of function generator, and what it means for it to be pseudo-random, in the obvious way. ($F(s)(x)$ should be computable in time polynomial in $n + |x|$. We require that adversaries run in time polynomial in n, so function queries can only be of size polynomial in n.)

The previous constructions generalize to get a function generator F' such that for $|s| = n$, $F'_s : \{0,1\}^* \rightarrow \{0,1\}^n$ is pseudo-random, **but only with respect to an adversary who guaranties that all his queries will be of the same length**.

So all we have to do is, in effect, use a different seed for every length.

For $x \in \{0,1\}^*$, define $\mathrm{lth}(x)$ to be the n bit string representing the length of x (mod 2^n).

Construction: Define $F''_s(x) = F'_{s'}(x)$ where $s' = F_s(\mathrm{lth}(x))$.

Theorem: This construction yields a pseudo-random function generator with unbounded inputs.

We will see that $F_s(x)$ can be used as a *signature* (or MAC), to someone else knowing s, of x. In case x is very long, the signer might want to begin computing $F_s(x)$ before seeing all of x and before knowing it's length. Here is a different class of constructions that starts with a (normal) pseudo-random function generator F and produces a pseudo-random function generator F' with unbounded inputs.

We will use two n-bit keys s_1 and s_2 (although of course they can be pseudo-randomly generated from one key.)
We will associate with s_1 a "hash" function $H_{s_1} : \{0,1\}^* \rightarrow \{0,1\}^n$. (More about this below.)

Construction: For $s_1 s_2 \in \{0,1\}^n$ and $x \in \{0,1\}^*$, define $F'_{s_1 s_2}(x) = F_{s_2}(H_{s_1}(x))$.

What property do we need from our hash family?

Definition: A *privately collision resistant hash family* H associates with every n-bit key k a function $H_k : \{0,1\}^* \to \{0,1\}^n$ such that

- There is a polynomial-time algorithm that given k and m, computes $H_k(m)$.
- The security property says that no two distinct strings of length polynomial in n have a significant chance of mapping to the same string by a random H_k. More formally:
  for every d and c, for sufficiently large n, for every two distinct strings m_1 and m_2 of length at most n^d, if a random k of length n is chosen, then the probability that $H_k(m_1) = H_k(m_2)$ is $\leq \frac{1}{n^c}$.

Theorem: These things provably exist.

Theorem: The above construction yields a pseudo-random F'.

Construction of a privately collision resistant hash family:
Let $k \in \{0,1\}^n$; view $k \in F = \mathrm{GF}(2^n)$.
Associate with each $m \in \{0,1\}^*$ a unique polynomial $P_m \in F[x]$.
Define $H_k(m) = P_m(k)$.

Note: $H_k(m)$ can be evaluated as the bits of m are coming in.

Remark: This doesn't exactly fit directly into our model, since we have to use randomness to find a way of representing field elements. Namely, we have to find an irreducible degree n polynomial over GF(2). However, we can assume this polynomial is fixed and known, or – to make things really fit into our model – we can use some of the bits of s_1 to find this polynomial.

Of course, there are many other constructions that work as well.
What about the following:?
View k as an integer; view each m as corresponding to a unique integer; define $H_k(m) = m \mod k$.

Permutation Generators

Traditionally, people wanted function generators to be *permutation generators*. This isn't really necessary, but it seemed like a good idea at the time (security was not as well understood), and we seem to get it for free anyway (see below).

Definitions:
Let F be a function generator such that $F_s : \{0,1\}^n \rightarrow \{0,1\}^n$. We say F is a *permutation generator* if for every s, F_s is one-one/onto. Usually we also insist that there is a poly time algorithm which given s and $y \in \{0,1\}^n$ computes $F_s^{-1}(y)$.

Of course, it doesn't hurt to say, in the definition of pseudo-random, that when the adversary is given a random function, it is actually a random permutation. Often we insist that for a *permutation generator* to be pseudo-random, the adversary can not only query the given permutation f, but also f^{-1}.

Theorem: If there exist pseudo-random function generators, then there exists pseudo-random permutation generators.

Proof: the construction uses the idea of “Feistel” permutations. (See below.)

How do we, in practice, construct permutation generators that we hope are pseudo-random?

We could use assumptions from number theory, but this is too inefficient for our tastes.
We could construct them from pseudo-random function generators, or from pseudo-random number generators, or from one-way functions (see below), but we don’t.
We construct them from “scratch”.

How we construct an (instance of) a (hopefully) pseudo-random permutation generator using the engine of composition

1. Start with a very simple basic permutation generator that on key k of length m generates permutation $F_k : \{0,1\}^n \to \{0,1\}^n$. F should do "enough" mixing, but not give us any real security. F_k will usually have the feature that it is easy to invert given k.
2. Define $F'_{k_1 k_2 \ldots k_r}(x) = F_{k_1} \circ F_{k_2} \circ \ldots \circ F_{k_r}$.
   That is, we compose F for r rounds.
3. We don't want our key to be too long, so we use a (very simple)round-key generation function
   $\mathit{keygen} : \{0,1\}^l \to \{0,1\}^{r \cdot m}$;
   for key K of length l, we finally define:
   $F''_K(x) = F'_{\mathit{keygen}(K)} = F'_{k_1 k_2 \ldots k_r}$ where $\mathit{keygen}(K) = k_1 k_2 \ldots k_r$.