Notes #3 (for Lecture 6)

Pseudo-Random Function Generators With Unbounded Inputs

We now want to discuss a more general kind of pseudo-random function generator, where the functions generated are defined on inputs of all lengths.

Definition: (See *Goldreich*, section 3.6.4.2 .)
A function generator F *with unbounded inputs* associates with each n bit key $k \in \{0,1\}^n$ a function $F_k : \{0,1\}^* \to \{0,1\}^n$. We insist that $F_k(x)$ be computable in time polynomial in the lengths of k and x.

By *pseudo-random* for such a generator, we mean the obvious thing: the Distinguisher adversary D is given a function $f : \{0,1\}^* \to \{0,1\}^n$ and can query f on inputs of any length (although since the adversary runs in polynomial time, the queries must be of polynomial length). We define $p_D(n)$ as the probability that D accepts F_k for randomly chosen $k \in \{0,1\}^n$. It is a bit trickier to define $r_D(n)$ because it doesn't make sense to say, "choose a random function $f : \{0,1\}^* \to \{0,1\}^n$" since there are infinitely many such functions. So let us assume that D (for key length n) never makes queries longer than n^e, and let $\{0,1\}^{\leq n^e}$ be the set of strings of length at most n^e. We can now define $r_D(n)$ as the probability that D accepts a randomly chosen $f : \{0,1\}^{\leq n^e} \to \{0,1\}^n$.

We will now discuss a number of ways of constructing a pseudo-random function generator with unbounded inputs. (Yet another way is described in *Goldreich*.) Sometimes it will be more convenient (or more efficient) to construct a function generator F such that for $k \in \{0,1\}^n$, F_k is only defined on inputs of length divisible by n, that is, $F_k : (\{0,1\}^n)^* \to \{0,1\}^n$. Given such a pseudo-random generator F, we can construct one whose functions are defined on all input lengths, as follows. Let $e : \{0,1\}^* \to (\{0,1\}^n)^*$ be a one-one, easy to compute function; define, for $k \in \{0,1\}^n$, $F'_k(x) = F_k(e(x))$ for $x \in \{0,1\}^*$. (For example, we can let $e(x) = 0^m 1x$ where m is the smallest nonnegative integer such that $|0^m 1x|$ is divisible by n.) We leave it as an exercise to prove that F' is pseudo-random.

Construction 1:
Say that F is a "normal" pseudo-random function generator, that is, for $k \in \{0,1\}^n$,
$F_k : \{0,1\}^n \to \{0,1\}^n$. Using the intuition from the previous lecture, define for $k \in \{0,1\}^n$, $F'_k : (\{0,1\}^n)^* \to \{0,1\}^n$ by:

- $F'_k(\lambda) = k$ where λ is the empty string.
- $F'_k(xy) = F_{F'_k(x)}(y)$ if $y \in \{0,1\}^n$ and $x \in (\{0,1\}^n)^*$.

Is F' pseudo-random? As discussed in the previous notes, F' is pseudo-random with respect to adversaries that are *restricted* so that for any given input function f, all of the adversary's queries to f are of the same length. (Exercise: prove this.) What about adversaries that are not so restricted? Clearly F' is not pseudo-random: if $f : (\{0,1\}^n)^* \to \{0,1\}^n$ is pseudo-randomly generated then for any $y \in \{0,1\}^n$ and $x \in (\{0,1\}^n)^*$, $f(xy) = F_{f(x)}(y)$.

We can fix this up by (properly) incorporating the length of the input string into our generator. For $x \in (\{0,1\}^n)^\ell$, let $\overline{\ell}$ be the n-bit representation of ℓ [1] and define $F''_k(x) = F'_k(\overline{\ell}x) = F'_{F_k(\overline{\ell})}(x)$.
Exercise: Prove F'' is pseudo-random (against unrestricted adversaries).
Exercise: Prove that if we had defined $F''_k(x) = F'_k(x\overline{\ell})$ then F'' would *not* be pseudo-random.

In fact, this method of incorporating the length can be used more generally. Say we are given a function generator F' with unbounded inputs, and say that F' is pseudo-random with respect to adversaries that are *restricted* so that for any given input function f, all of the adversary's queries to f are of the same length. We can then define $F''_k(x) = F'_{F'_k(\overline{\ell})}(x)$ where ℓ is the length of x.
Exercise: Prove F'' is pseudo-random (against unrestricted adversaries).

Construction 2: We use cipher-block chaining.
Say that F is a "normal" pseudo-random function generator, that is, for $k \in \{0,1\}^n$, $F_k : \{0,1\}^n \to \{0,1\}^n$. Using the intuition from the previous lecture, define for $k \in \{0,1\}^n$, $CBC_k : (\{0,1\}^n)^* \to \{0,1\}^n$ by:

- $CBC_k(\lambda) = F_k(\overline{0})$ where λ is the empty string.
- $CBC_k(xy) = F_k(y \oplus CBC_k(x))$ if $y \in \{0,1\}^n$ and $x \in (\{0,1\}^n)^*$.

Is CBC pseudo-random? As above, CBC is pseudo-random with respect to adversaries that are *restricted* so that for any given input function f, all of the adversary's queries to f are of the same length. (Difficult Exercise: prove this.) What about adversaries that are not so restricted? Clearly CBC is not pseudo-random: if $f : (\{0,1\}^n)^* \to \{0,1\}^n$ is pseudo-randomly generated then $f(f(\lambda)) = f(\lambda)$.

As above, we can fix this up by defining
$F''_k(x) = CBC_{F_k(\overline{\ell})}(x)$ where $x \in (\{0,1\}^n)^\ell$.
Alternatively, we can define
$F''_k(x) = CBC(\overline{\ell}x)$.
Alternatively, we can change the base case of CBC so that on λ it evaluates to $F_k(\ell)$ where ℓ is the length of the input we are ultimately interested in.
Any of these constructions yield F'' pseudo-random against unrestricted adversaries. The reader should try to prove these statements. The proof of the last two are especially subtle. The reader should note, and try to prove, that if we had defined
$F''_k(x) = CBC(x\overline{\ell})$ then F'' would *not* be pseudo-random against unrestricted adversaries (no matter what function generator F we started with).

Construction 3: A third way of creating a pseudo-random function generator with unbounded inputs, is by "hashing" the input down (or up) to a string of length exactly n, and then applying a normal pseudo-random function generator. By "hashing", we mean applying a function chosen at random from a family that satisfies a particular type of randomness property. For the construction described here, we need the family to be "privately collision resistant". The family will contain functions mapping $\{0,1\}^*$ to $\{0,1\}^n$ and have the property that if a random function is chosen from the family and you don't know what it is, then you will not be able to find two strings that map to the same string.

Definition: A *privately collision resistant hash family* H satisfies the following.

[1] A technicality: in order for this to technically make sense for all values of ℓ, we should let $\overline{\ell}$ be the n-bit representation of $\ell \bmod 2^n$.

- For each security parameter n, for every key k of length n there is a function $H_k : \{0,1\}^* \to \{0,1\}^n$. There is an algorithm that given k and m, runs in time polynomial in n and $|m|$ and computes $H_k(m)$.
  (In practice, we may allow the keys associated with security parameter n to be of length some polynomial in n, rather than of length exactly n.)

- 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 c and e, for sufficiently large n, for every two distinct strings m_1 and m_2 of length at most n^c, if a random k of length n is chosen, then the probability that $H_k(m_1) = H_k(m_2)$ is $\leq \frac{1}{n^e}$.

Theorem: Let F be a (normal) pseudo-random function generator, and let H be a privately collision resistant hash family. Define the function generator F' as follows such that on a key k of length $2n$, F'_k will map $\{0,1\}^*$ to $\{0,1\}^n$. Let k_1 and k_2 be strings of length n; for every $m \in \{0,1\}^*$ define $F'_{k_1k_2}(m) = F_{k_2}(H_{k_1}(m))$.

Then F' is pseudo-random.

Proof: This is not hard, and is left as an exercise.

It is not hard to construct (provably) privately collision resistant hash families. Here is one way. View an n-bit key as an integer α, and for every bit string m, let m' be the integer resulting from adding a 1 on to the left of m. We then define $H_\alpha(m) =$ the n-bit representation of (m' mod α). If an adversary can (without seeing α) construct distinct strings m_1 and m_2 that hash to the same thing, then he can find a non-zero integer m' (namely the absolute value of $m'_1 - m'_2$) such that $m' \bmod \alpha = 0$, that is, α divides m'. But since every integer m' of length polynomial in n is only divisible by an exponentially small fraction of the n bit numbers, it is very unlikely that m' will be divisible by α.

Here is another way (not discussed in class) to construct a (provably) privately collision resistant hash family. First, assume that for each n we have an n bit prime number $p_n > 2^{n-1}$. For security parameter n, a key will be a random $n-1$ bit string α; we wish to view α as an integer less than p_n. We now want to define $H_\alpha : \{0,1\}^* \to \{0,1\}^n$. Let $m = m_0 m_1 \cdots m_{d-1}$ be a d bit string. We define $H_\alpha(m) = (\alpha^d + \sum_{i=0}^{d-1} m_i \alpha^i) \bmod p_n$.

Another way to look at this is that we associate with each string m the distinct formal polynomial $Q_m(u) = m_0 + m_1 u + m_2 u^2 + \cdots + m_{d-1} u^{d-1} + u^d$, where we view this as a polynomial over the ring $\mathbb{Z}_{p_n} = \{0, 1, \cdots, p_n - 1\}$, where arithmetic is performed mod p_n; we then have $H_\alpha(m) = Q_m(\alpha)$.

Say now that m_1 and m_2 are two distinct messages, each of length at most n^c. Then $H_\alpha(m_1) = H_\alpha(m_2)$ if and only if $(Q_{m_1} - Q_{m_2})(\alpha) = 0$ where $Q_{m_1} - Q_{m_2}$ is a nonzero polynomial of degree at most n^c. This means that $Q_{m_1} - Q_{m_2}$ can have at most n^c roots in $\mathbb{Z}_{p_n}$. There are 2^{n-1} possible values for α, so the probability is at most $\frac{n^c}{2^{n-1}}$ that a random α will cause $(Q_{m_1} - Q_{m_2})(\alpha) = 0$.

Note that a more efficient version of this idea is to view m not as a sequence of bits, but rather as a sequence of $n-1$ bit integers, and to treat these integers as the coefficients of a polynomial. We could also use Horner's method to evaluate the polynomial more efficiently:
$H_\alpha(m) = ((\alpha + m_{d-1}) \cdot \alpha + m_{d-2}) \cdot \alpha + m_{d-3} \ldots$.

Note that in this and the previous example of a hash family, $H_\alpha(m)$ can be evaluated from left to right as the blocks of m come in, so it is not necessary to know all of m – or even the length of m – in advance.

This polynomial evaluation example of a hash family doesn't exactly conform to our definition since we didn't say where the primes $\{p_n\}$ come from. If we wanted to make it conform, we could do the following. We "simply" include in the key all the random bits we need to select a prime p_n; these would include all the randomness involved in choosing random numbers and testing them for primality. (Alternatively, we could merely include a seed for a pseudo-random number generator that generates these bits.) Note that it wouldn't hurt if all these random bits, as well as p_n, were made public and fixed for all time. We therefore treat these primes as pre-existing, and not as part of the key.

Message Authentication Codes

One important application of pseudo-random function generators with unbounded inputs is to "shared-private-key signature schemes" where two people share a secret key that allows one of them to sign messages to the other. Rather than define this concept carefully, we will instead define the special case known as "message authentication codes", or MACs.

Formally, a **MAC** is just a function generator F with unbounded inputs. We say a MAC is secure if a polynomially bounded adversary, given oracle access to $f = F_k$ for randomly chosen $k \in \{0,1\}^n$, is unable, except for negligible probability, to predict the value of $f(x)$ for some x for which f has not been queried. (We leave it as an exercise for the reader to give a more formal definition.) It is clear that every pseudo-random function generator is also a secure MAC, although the converse is not true.

A secure MAC can be used as follows. Say that two people have shared a secret random key k. One of them can "sign" a string x to the other by sending (together with x itself) the "signature" $\sigma = F_k(x)$; the other can verify the signature by checking that $\sigma = F_k(x)$. An adversary should not be able to sign any string that he has not already seen signed.

In some settings, the length of string to be MACed will be fixed. Also, in some settings, the good guys will only ever MAC one string; in this case, it is possible to have a secure MAC without any complexity theory assumptions.

Pseudo-Random Permutation Generators

Before discussing encryption and sessions, we will discuss one more kind of pseudo-random generator which is sometimes used in encryption.

Definition: Let F be a function generator where, for $k \in \{0,1\}^n$, $F_k : \{0,1\}^n \to \{0,1\}^n$. We say F is a *permutation generator* if each F_k is one-one (and hence onto) and easy to invert, given k. That is, the function that for $k, \alpha \in \{0,1\}^n$ maps (k, α) to $F_k^{-1}(\alpha)$ is polynomial-time computable.

Our definition of "pseudo-random" for a permutation generator is the same as for any function generator: an adversary shouldn't be able to tell the difference between a randomly generated function and a pseudo-randomly generated function. Note that it wouldn't have mattered if we had replaced the word "function" in the preceding sentence by the word "permutation", since we certainly (in a polynomial in n number of queries) cannot tell the difference between a randomly generated function and a randomly generated permutation from $\{0,1\}^n$ to $\{0,1\}^n$.

However, for many applications, we need a stronger definition of pseudo-randomness for a permutation generator.

Definition: Let F be a permutation generator. A **strong** adversary for F is given a black box for a permutation $f : \{0,1\}^n \to \{0,1\}^n$ *as well as* a black box for f^{-1}; $p_D(n)$ is defined as the probability that D accepts (f, f^{-1}) for a pseudo-randomly generated f, and $r_D(n)$ is defined as the probability

that D accepts (f, f^{-1}) for a randomly chosen permutation f; the adversary must be polynomial size (in the nonuniform setting) or probabilistic polynomial-time (in the uniform setting).

We say F is **strongly pseudo-random** if for every strong adversary D, all c and sufficiently large n, $|p_D(n) - r_D(n)| \leq 1/n^c$.

We will see later how to use an arbitrary pseudo-random function generator to construct a strongly pseudo-random permutation generator. We will also see that DES and AES are defined in such a way that they are automatically permutation generators.

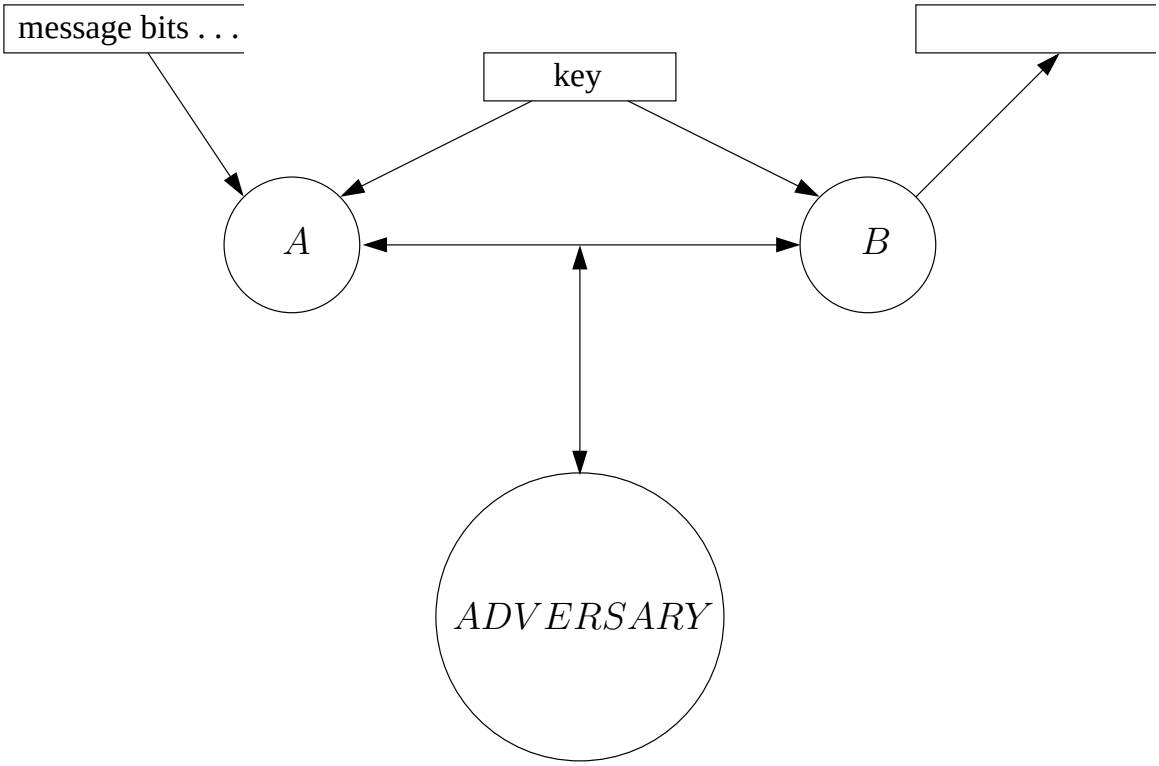
Secure Sessions

We will now talk in detail about one of the main applications of pseudo-random generators: shared-private-key cryptosystems, or sessions.

We have two good guys A and B who have shared a random n bit key k. A wishes to communicate a (long) message to B using an insecure channel. More accurately, a "process" spawned by A is communicating (or trying to) with a "process" spawned by B; we call this a *session*. Keep in mind that there is only one message we are concerned with, but it can be very long and it consists of everything the A-process wishes to say during the seconds or minutes or centuries that the session lasts.

In the simpler setting, the adversary is able to listen in on the channel, but not interfere with it at all. He is trying to break *privacy* by learning something about the message that he "shouldn't know". Who is choosing the message? We will let the adversary choose the message ("chosen plain-text attack") except for one bit that he will try to guess. The adversary will chose a piece of the message, see the encryption, chose the next piece, see the encryption, etc. Thus, the adversary in effect completely determines the environment in which the session is taking place.

In the more complicated and more realistic setting, the adversary is able to completely control the bits on the channel, adding and deleting and changing them at will. In this setting the adversary is trying to do one of two different things, and the system is insecure if he is successful at either one. One thing he may try to do is break privacy, but the definition is now more complicated. The other thing he may try to do is break *integrity* by tricking B into outputting something wrong.



Consider the following example, that we will also talk about later in more detail.
We have a pseudo-random permutation generator F.
The key k is of length n, and the message consists of a sequence of n-bit pieces $m_0, m_1, \ldots$.
A encrypts this by sending $e_0 = F_k(m_0), e_1 = F_k(m_1), \ldots$ and B decrypts in the obvious way.

This system certainly doesn't satisfy integrity, since the adversary can replace $e_0, e_1, \ldots$ with anything he likes, and B will still decrypt without complaining.

But it also doesn't satisfy privacy. It is easy for an adversary to tell which message pieces are equal to which other message pieces.