Notes #2

Expanding the Length of a Pseudo-Random Number Generator

We now want to show how a pseudo-random number generator that only does a little bit of expansion, can be used to construct a pseudo-random generator that does a lot of expansion. The idea is that we view $G(s)$ as consisting of "stuff" that we can spit out, together with a new seed that we feed back into the generator, etc., a polynomial number of times. This construction is often used in practice to form a generator that is continually spitting out stuff (such as floating point numbers) "forever"; in our definition of security, the adversary will only see the first n^e bits that are spit out, for some e of his choosing.

Let G be a number generator with length function $l(n)$, where $l(n) = e(n)+n$. For every natural number i and every bit string s, $|s| = n$, define
$G_0(s) = \lambda$ =the empty string;
$G_{i+1}(s) = \alpha G_i(\beta)$ where $G(s) = \alpha\beta$ and $|\alpha| = e(n)$ and $|\beta| = n$.
Let $t(n)$ be a function computable in time polynomial in n, such that is $t(n)$ is polynomial in the value of n. Define
$G'(s) = G_{t(n)}(s)$; note that G' is a number generator with length function $l'(n) = e(n)t(n)$. (We will assume $l'(n) > n$.)

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

Proof: (nonuniform setting)
Assume G' is not pseudo-random. Let $D' = \{D'_n\}$ be a polynomial size circuit family for distinguishing G'; for each n, let $p_{D'}(n)$ and $r_{D'}(n)$ be the standard probabilities associated with D'_n and G'. Fix n and assume (without loss of generality) that $p_{D'}(n) - r_{D'}(n) > 1/n^c$.

We now describe a sequence of experiments that are "hybrids" between the experiment that gives rise to $p_{D'}(n)$ and the experiment that gives rise to $r_{D'}(n)$. For $0 \leq i \leq t(n)$ let q_i be the probability: IF $\alpha_1, \alpha_2, \cdots, \alpha_{t(n)-i}$ are $t(n) - i$ randomly chosen strings from $\{0,1\}^{e(n)}$, and s is a randomly chosen string from $\{0,1\}^n$, and D'_n is given as input $[\alpha_1\alpha_2\cdots\alpha_{t(n)-i}G_i(s)]$, THEN D'_n accepts. Clearly $q_{t(n)} = p_{D'}(n)$ and $q_0 = r_{D'}(n)$. So $q_{t(n)} - q_0 > 1/n^c$. So there exists an i, $0 \leq i < t(n)$, such that $q_{i+1} - q_i > 1/(n^c t(n))$; fix such an i.

Let D_n be the following (probabilistic) distinguisher for G. Given a string β of length $l(n)$, $\beta = \alpha\gamma$ where $|\alpha| = e(n)$ and $|\gamma| = n$, choose random $\alpha_1, \alpha_2, \cdots, \alpha_{t(n)-i-1}$ and run D'_n on $[\alpha_1\alpha_2\cdots\alpha_{t(n)-i-1}\alpha G_i(\gamma)]$.

Letting $p_D(n)$ be the probability that D_n accepts $G(s)$ for random s chosen from $\{0,1\}^n$, we see that $p_D(n) = q_{i+1}$. Letting $r_D(n)$ be the probability that D_n accepts a random β from $\{0,1\}^{l(n)}$, we see that $r_D(n) = q_i$. So $p_D(n) - r_D(n) > 1/(n^c t(n))$, so $\{D_n\}$ is an adversary that breaks the pseudo-randomness of G. □

Here is an iterative (as opposed to recursive) way of viewing the above construction. The idea is that we start with an n-bit seed s_0; we then compute $G(s_0) = \alpha_1 s_1$ where $|\alpha_1| = e(n)$ and $|s_1| = n$. We "spit out" α_1 as "pseudo-random" bits, set the new seed to be s_1, and continue as long as we

like. (Of course the adversary is only allowed to run this for a polynomial-in-n amount of time, for a polynomial of his own choosing.)

Pseudo-Random Function Generators

We now wish to define an even stronger notion of pseudo-random generation. A "function generator" will take an n bit seed s, and produce a *function* F_s mapping $\{0,1\}^n$ into $\{0,1\}^n$. Of course F_s is a exponentially large object; rather than actually outputting it, we merely require that the generator can efficiently evaluate F_s on any input. Our definition of "pseudo-random" will be that no efficient algorithm can tell the difference between being given a black-box for a random function, and being given a black box for F_s for random s.

Definition:
A *Function Generator* F associates with each n and each $s \in \{0,1\}^n$ a function
$F_s : \{0,1\}^n \to \{0,1\}^n$, such that there is a polynomial time algorithm that given $s \in \{0,1\}^n$ and $x \in \{0,1\}^n$, computes $F_s(x)$.

Definition: ("uniform adversary setting")
Let F be a function generator; we will define what it means for F to be *pseudo-random*. By an *adversary* D we mean a probabilistic algorithm that has two inputs: a string 1^n of n ones indicating the security parameter n, and an "oracle" for a function f mapping $\{0,1\}^n$ into $\{0,1\}^n$. D must run in time polynomial in n. Formally, D may be thought of as a Turing machine with a special "query" tape on which an n bit string x may be printed; whenever the machine enters a designated query state, the value of $f(x)$ appears (in one step) on a special "answer" tape. D is probabilistic and must halt in time polynomial in n, outputting a single bit indicating whether or not the function f is accepted.

We say F is pseudo-random if for every such adversary D:
For each n, define
$p_D(n)$ = the probability that if s is randomly chosen from $\{0,1\}^n$ and D is given 1^n and an oracle for F_s, then D accepts, and define
$r_D(n)$ = the probability that if D is given 1^n and a randomly chosen function mapping $\{0,1\}^n$ to $\{0,1\}^n$, then D accepts.
Then $|p_D(n) - r_D(n)| \leq \frac{1}{n^c}$ for every c and sufficiently large n.

Note that D is allowed to base the queries it makes to the oracle on the answers it receives to previous queries, and so in this way is permitted to be highly adaptive. The definition of pseudo-random in the "nonuniform adversary" setting is very similar to the one above. In this case, D would be a family $\{D_n\}$ of deterministic, polynomial size circuits, but of a special type. We wish to think of the input to D_n as a function f mapping $\{0,1\}^n$ to $\{0,1\}^n$; formally, the only inputs to D_n will be the constants 0 and 1; however, in addition to the normal boolean gates, D_n will have special "function" gates with n input lines and n output lines. In the experiments defining $r_D(n)$ or $p_D(n)$, D_n will be run with either its "function" gates interpreted as f where f is a randomly chosen function (and the same f is used for all function gates), or interpreted as F_s for a randomly chosen s (where the same f is used for all the function gates).

It turns out that pseudo-random function generators exist if and only if pseudo-random number generators exist. The harder part of this theorem is showing how to use a pseudo-random number generator to construct a pseudo-random function generator. The idea is as follows. Let G be a pseudo-random number generator with length function $l(n) = 2n$. For an n bit seed (or key) s,

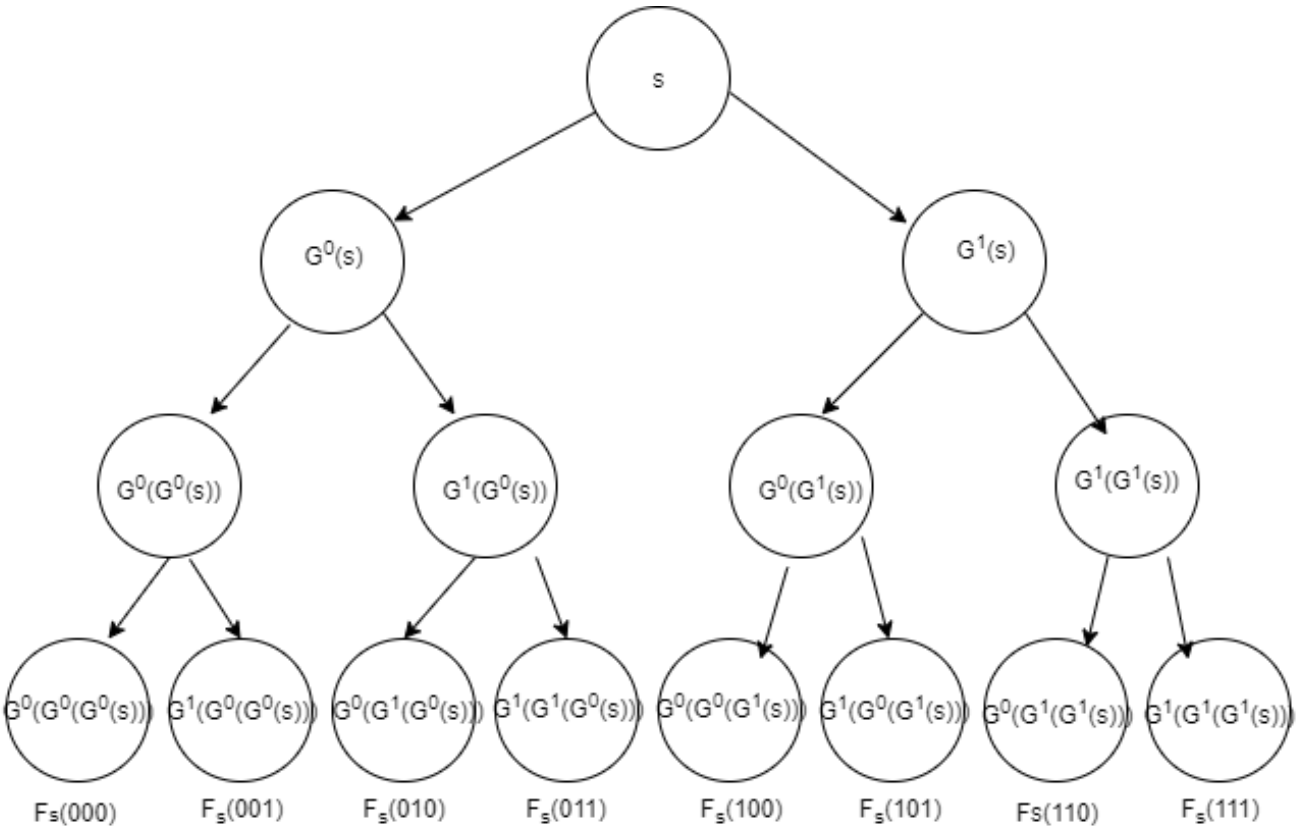

Figure 1: Example where $n = 3$.

consider the following tree: the root is labelled with s, and if a node has label t, then the the left and right children are labelled with (respectively) the left and right halves of $G(t)$. We define $F_s(x)$ to be the label of the leaf corresponding to the path x through the tree. This is usually referred to as the GGM construction, after Goldreich, Goldwasser, and Micali.

Construction:
Let G be a number generator with $l(n) = 2n$. Define the function generator F as follows. Let s and σ be n bit strings, $\sigma = \sigma_1\sigma_2 \cdots \sigma_n$; define $F_s(\sigma) = G^{\sigma_n}(G^{\sigma_{n-1}}(\cdots(G^{\sigma_2}(G^{\sigma_1}(s)))\cdots))$ where $G^0(t)$ is the left half of $G(t)$ and $G^1(t)$ is the right half of $G(t)$.

Theorem: There exists a pseudo-random number generator $\iff$ there exists a pseudo-random function generator.

Proof of $\Longleftarrow$:
Let F be a pseudo-random function generator. Define the number generator G by
$G(s) = F_s(\bar{0})F_s(\bar{1})$, where if $|s| = n$, then $\bar{0}$ is the n bit representation of 0 (that is, the string of all 0's), and $\bar{1}$ is the n bit representation of 1. It is easy to prove that G is a pseudo-random number generator.

Proof of $\Longrightarrow$:
Assume there is a pseudo-random number generator. Then by a previous theorem, there is a pseudo-random number generator with length function $l(n) = 2n$. Let F be the function generator from

the construction above. It is not too hard (nor is it too easy) to prove that F is pseudo-random.

Here is the basic idea. Assume we can break the pseudo-randomness of F. We want to describe "hybrids" between the tree construction (using G) of depth n starting from a random string and yielding 2^n leaves, and the construction where we choose 2^n random strings independently. We let the ith hybrid be where we chose the ith level of the tree randomly and then, using the above construction, let each be the root of a tree of depth $n - i$ with 2^{n-i} leaves.

Assume an adversary can distinguish between the case of examining the leaves of an i-th hybrid, versus the case of examining the leaves of an $i + 1$-st hybrid. That is, an adversary can tell the difference between examining the leaves of trees whose roots at level i have been chosen randomly, versus trees where the two children of each such root have been chosen randomly. This adversary allows us to break G using multiple sampling:
Whenever, for a node x at level i, a leaf of the tree rooted at x is queried for the first time (amongst all leaves in this tree), then the next input string $\alpha\beta$ is used to label the children of x, And then G is used as above to answer all the queries to the leaves of this tree. If all the $\alpha\beta$ were chosen randomly, then this corresponds to the $i + 1$-st hybrid; if all the $\alpha\beta$ were chosen pseudo-randomly, then this corresponds to the i-th hybrid.$\square$

Although we have specified that a function generator on a seed of length n must map n-bit strings to n-bit strings, this is rather arbitrary. In general, we can permit $F_s : \{0,1\}^{l_1(n)} \to \{0,1\}^{l_2(n)}$ where l_1 and l_2 are polynomially bounded functions of $n = |s|$ and are easy to compute.[1] The definition of "pseudo-random" can be modified in the obvious way to apply to such generators.

It is easy to see that given a pseudo-random function generator F as originally defined, we can construct a new pseudo-random function generator F' that on a seed of length n maps n^c-bit strings to n^d-bit strings. Assume $c \geq d$; we leave the case $d > c$ as an exercise. We can define $F'_s(x) =$ the first n^d bits of $F_{G(s)}(x)$ where G is a pseudo-random number generator with length function n^c, and we leave it as an exercise to prove that F' is pseudo-random. We will give some alternative constructions below. From now on, when we say "if there exists a pseudo random generator", we mean the existence of a pseudo-random number or function generator.

[1] Technically, we want to assume that $2^{l_1(n)} \cdot l_2(n) > n$, so that the generator will be length expanding.

DES and *AES*

The most famous examples of function generators assumed to be pseudo-random are *DES* and *AES*. Technically speaking it is not right to call these function generators, since they are only defined on a small number of key lengths and block size. Nonetheless, by calling them function generators we hopefully make it clear what security property we want from them: it should be hard for an adversary to distinguish between a randomly chosen function and a function that comes from a randomly chosen key.

DES (standing for *data encryption standard*) was developed in the United States in the seventies. For a 56-bit key k, $\mathrm{DES}_k : \{0,1\}^{64} \to \{0,1\}^{64}$. Our notion of security means that an adversary that must run in "reasonable" time, will not be able to distinguish between DES_k for random k, and a truly randomly chosen function, except with "very small" probability. In practice, the adversarial attacks that people use on DES are of the following form: given a black box for DES_k, try to find k – it is easy to tell when and if one has succeeded (by seeing if DES_s gives the same answer as the black box on a few arbitrary inputs), and you will hardly ever succeed in this way on a randomly chosen black box. A "brute force" attack would look at a few values of the black box, and then search through all possible DES keys, essentially taking about the time necessary to do 2^{55} DES evaluations. Much more clever algorithms take time about 2^{45}, but require about 2^{45} evaluations of the black box. All this is not of immediate practical relevance, since the clever attacks are not that easy to do and the brute force attack is not that hard to do. And 64 is too small. Do you see why?

However the insights gained as a result of these attacks were used to fine-tune the successor to DES, chosen around 2000 by NIST (the United States National Institute of Standards and Technology) after a three year competition. This successor is called AES (for *advanced encryption standard*), and is actually a collection of three standards comprising three different key lengths and block size 128. For our purposes, we will view AES as having 128-bit keys; for each key k, $\mathrm{AES}_k : \{0,1\}^{128} \to \{0,1\}^{128}$. AES is described at

`http://csrc.nist.gov/groups/ST/toolkit/block_ciphers.html`

and we will discuss both DES and AES in more detail later.

Expanding the Block Length of a Pseudo-Random Function Generator

Let's say we have a function generator F which (as with AES), we only wish to apply with n-bit keys for a specific n (and with n-bit block size). Now we want to create a pseudo-random function generator F' with, say, an n-bit key generating functions that map $2n$-bit strings to n-bit strings. One way we can do this is by using the idea of the tree-like construction above. We can view F as generating a tree with 2^n leaves, and the input specifying a path to a leaf. If we then use each leaf to also generate a tree with 2^n leaves, we get a big tree with 2^{2n} leaves. We can express this construction as $F'_k(xy) = F_{F_k(x)}(y)$ (where $|x| = |y| = |k|$). We leave it as an exercise to prove that F' is pseudo-random; the proof involves the construction of an intermediate hybrid. (It would be convenient to first prove that a pseudo-random function generator is also pseudo-random against multiple sampling.)

An alternative construction uses what is known as "cipher-block chaining". It is:
$F'_k(xy) = F_k(F_k(x) \oplus y)$, where $\oplus$ means bit-wise exclusive-or. We leave it as an exercise to prove that F' is pseudo-random; the proof involves the construction of only one intermediate hybrid (instead of F_k, use a randomly chosen function from n bits to n bits), but it is reasonably difficult and subtle.

All of the above ideas can be generalized and combined, in *provably* secure ways. For example, we can generalize each of the above constructions to obtain pseudo-random F' that on n-bit keys generates functions from n^c bits to n bits. If we now want the output of our functions to consist of, say, n^{d+1} bits, we can create a new generator F'' which uses keys consisting of n^d subkeys of length n as follows: $F''_{k_1 k_2 \ldots k_{n^d}}(\alpha) = F'_{k_1}(\alpha)F'_{k_2}(\alpha)\ldots F'_{k_{n^d}}(\alpha)$. If we would rather the key length were n instead of n^{d+1}, we can define $F'''_k(\alpha) = F''_{G(k)}(\alpha)$ where G is a pseudo-random number generator mapping n bits to n^{d+1} bits. In fact, we could use $G(k) = F_k(\overline{1})F_k(\overline{2})\ldots F_k(\overline{n^d})$, where $\overline{i}$ is the n-bit representation of integer i.[2] All of these constructions can be proven secure using hybrid arguments.

We now give an example of a construction that does *not* work. If F is a function generator (that on keys of length n maps n bits to n bits), define the function generator F' by $F'_k(x) = F_k(k \oplus x)$ (where $|x| = |k|$, and $\oplus$ means bit-wise exclusive-or). A person might prefer to use F' instead of F, in the hopes that it would be somehow better. However, we can show that this construction does not preserve pseudo-randomness, and hence F' might not be pseudo-random, even if F is.

Actually, we cannot really prove that there is a pseudo-random F such that F' is not pseudo-random, since it is possible that there are *no* pseudo-random generators to start with. However, if we assume that pseudo-random generators exists, then we can prove the existence of a pseudo-random F such that F' is not pseudo-random. More specifically, we will show how to take an arbitrary pseudo-random function generator H, and "pervert" it to form a pseudo-random function generator F such that F' is not pseudo-random.

So let H be an arbitrary pseudo-random function generator that on keys of length n maps n bits to n bits. Define the function generator F as follows: for every key k and input x of length n,
$F_k(x) = H_k(x)$ if $x \neq k$;
$F_k(x) = \overline{0}$ if $x = k$.
Since $F_k(k) = \overline{0}$, we have $F'_k(\overline{0}) = \overline{0}$, and so it is easy to see that F' is not pseudo-random.

It remains to show that F *is* pseudo-random. Assume otherwise. Let $\{D_n\}$ be a polynomial size family of circuits that breaks the pseudo-randomness of F; define the probabilities $p_D(n)$ and $r_D(n)$ in the usual way for $\{D_n\}$, and say $|p_D(n) - r_D(n)| > 1/(n^c)$ for infinitely many n. We will describe a family $\{D'_n\}$ for breaking the pseudo-randomness of H. Fix n such that $|p_D(n) - r_D(n)| = \epsilon > 1/(n^c)$.

The idea is that if D_n is "likely" to evaluate F_k on k, then we can use D_n to find the key of H_k, allowing us to break H. If, on the other hand, D_n is "unlikely" to evaluate F_k on k, then D_n behaves essentially the same way on F_k as on H_k, and we can use this fact to break H. It turns out that we can use, for example, "$\geq \epsilon/2$" for "likely".

Let $q(n)$ be the probability that, if a random $k \in \{0,1\}^n$ is chosen and D_n is given F_k (that is, the function gates are interpreted as F_k), then at some point the query k is made.

We first consider the case that $q(n) \geq \epsilon/2$. Then define D'_n as follows on function f. D'_n simulates D_n on f; afterwards, for each query x that was made and for some query y that was not made, D'_n evaluates $f(y)$ and computes $H_x(y)$, and accepts if these two strings are equal; if no such equality is found, then D'_n rejects. We leave it as an exercise to show that D'_n breaks the pseudo-randomness of H.

We lastly consider the case that $q(n) \leq \epsilon/2$. In this case we just let D'_n be D_n. We leave it as an exercise to show that D'_n breaks the pseudo-randomness of H.

We will see later that pseudo-random generators exist if and only if "one-way functions" exist, if

[2]Technically, we should say that $\overline{i}$ is the n-bit representation of integer i mod 2^n. For practical values of n and d however, we will always have $n^d < 2^n$.

only if secure shared-private-key cryptosystems exist, if and only if secure signature schemes exist, and we have constructions for each direction of each of these theorems. In practice, a presumed pseudo-random function generator – let's say AES – is constructed directly, and then AES is used to construct one-way functions, secure shared-private-key cryptosystems, "message authentication codes", pseudo-random number generators, and other kinds (see the previous paragraph) of pseudo-random function generators.

In practice, signature schemes are constructed using ideas related to "public-key" cryptography, rather than by using AES.