Computer Science 2426F | Fall, 2020
St. George Campus | University of Toronto

Notes #6

Where do pseudo-random generators come from?

Later we will define One-way Functions: functions that are easy to compute, but hard on the average to invert. We will see that:
one-way functions exist $\Leftrightarrow$
pseudo-random number generators exist $\Leftrightarrow$
pseudo-random function generators exist $\Leftrightarrow$
strongly pseudo-random permutation generators exist.

All of the $\Leftarrow$ implications are trivial to construct and prove. We have proved the second $\Rightarrow$ and we have asserted the third $\Rightarrow$. The first $\Rightarrow$ is the hardest to prove and involves the most inefficient constructions in general, but for special cases of one-way functions it is not too difficult.

Where do these things come from, in practice? We can directly construct (hopefully) one-way functions, and then use them to construct the other things, or we can directly construct pseudo-random number generators, or we can directly construct pseudo-random function generators, or we can directly construct pseudo-random permutation generators. In practice, we usually do the last.

Techniques for *Directly* Constructing One-Way Functions

Although in practice we construct one-way functions by using (directly constructed) permutation generators that we hope are pseudo-random – such as DES or AES, there are a number of conjectures that lead to direct constructions of one-way functions. These are mostly conjectures about the computational complexity of certain number theory problems such as discrete-log or integer-factorization. These conjectures are most often used in the context of "public-key" cryptography, and we will discuss them later.

Techniques for *Directly* Constructing Pseudo-Random Number Generators

Although in practice we construct pseudo-random number generators by using (directly constructed) permutation generators that we hope are pseudo-random – such as DES or AES, there are a number of conjectures that lead to direct constructions of pseudo-random number generators. In particular, certain complexity assumptions about simple combinatorial constructions can be used to directly construct pseudo-random number generators. A good example involves "subset-sum".

Let $l(n) > n$ be a fixed function.
Let $A = a_1, a_2, \ldots, a_n$ be a sequence of n integers, of $l(n)$ bits each;
let $x = x_1, x_2, \ldots, x_n$ be a sequence of n bits;
define $G(A, x) = [A, (x_1a_1 + x_2a_2 + \ldots + x_na_n) \bmod 2^{l(n)}]$. (Note that we are using *integer addition* here.

Note that G expands from length $nl(n) + n$ to length $nl(n) + l(n)$. It can be shown that if G is one-way, then G is pseudo-random. Most (or at least many) people think that G *is* one-way, at least if $l(n)$ is "small enough". In fact, once $l(n)$ gets to be of size about n^2, then it can be proven *not* to be one-way (using difficult "lattice reduction" techniques).

If we consider the problem of inverting G in the *worst case* (instead of merely in the average case) then we get an NP-hard problem, and it is tempting to think that this fact is relevant. It is not clearly relevant, however: when $l(n) = n^2$, it is still NP-hard to invert G in the worst case, but easy on the average.

Techniques for *Directly* Constructing Pseudo-Random Function Generators

It appears that most techniques to directly construct pseudo-random function generators actually construct pseudo-random permutation generators.

Techniques for *Directly* Constructing Pseudo-Random Permutation Generators

Now we come to the type of constructions used in DES and AES. We hope that these ideas yield *strongly* pseudo-random function generators as defined in Notes #4. We will see in a moment why the ideas in these constructions naturally yield permutation generators. The easy-to-invert (if you know the key) property, together with the pseudo-randomness being strong, is necessary in some applications (as we have seen), but not in all applications; this was considered much more important in the seventies than it is today.

Here is the general idea of DES, AES, and similar constructions.
We want to construct a pseudo-random permutation generator F.
We start with a special permutation generator F' which is not at all pseudo-random, but which is very easy to compute; we then compose F' with itself, for r rounds, using different keys each time. The hope is that pseudo-randomness emerges from this, as composition adds complexity, while the one-one/onto property ensures that our range isn't collapsing. Of course, it is crucial that F' have some nice "mixing" and "nonlinearity" properties, but no one has ever been able to articulate how to measure such things sufficiently well to make the choice of F' a science. In practice, a standard such as AES is chosen based on efficiency (which is easy to measure), on how well experts in the field have been able to break it so far, and on the spiritual state of mind that the code induces in those who read it.

Let's fix a block size n, a key size m for F, and a key size m' for F', and let r be the number of rounds. A key for F will be an m-bit string K, and the input will be an n-bit string x. The computation $F_K(x)$ begins by using K to generate r "round-keys" $k_1, k_2, \ldots, k_r$ of m' bits each; this is done using a "key scheduling" function $KS : \{0,1\}^m \to \{0,1\}^{m'r}$. KS must be a very easy to compute function, and is chosen in a particular way for various mystical reasons that I cannot understand. We can now define:
$F_K(x) = [F'_{k_1} \circ F'_{k_2} \circ \cdots \circ F'_{k_r}](x)$.

Further Details About DES
(with only a few insignificant lies)

DES became a standard in 1976. Because its key size is only 56 bits, and because it can be broken even faster than the brute force time of 2^{56} using clever methods, it was replaced by AES in 2002. (There was also a concern that the block size of 64 bits was too small.)

If the generator F is DES, then the block size is $n = 64$, the key size is $m = 56$, and the number of rounds is $r = 16$. I will not discuss the key scheduling function KS here.

The permutation generator F' works as follows. The block size is $n = 64$ and the key size is $m' = 48$. F' works as follows. Let the input to F'_k be (L, R) where $|L| = |R| = 32$. Then
$F'_k(L, R) = (R, L \oplus f(k, R))$, where $f : \{0,1\}^{48+32} \to \{0,1\}^{32}$ is a special function discussed below. The important point to realize is that no matter what f is, F' will be a permutation generator: to invert F'_k on (α, β) (where $|\alpha| = |\beta| = 32$), just compute (L, R) where $R = \alpha$ and $L = \beta \oplus f(k, \alpha)$. This very generic way of getting a permutation generator is called a Feistal construction.

The function f works as follows: $f(k, R) = g(k \oplus R')$ where R' is a 48-bit extension of R (obtained by repeating certain bits), and where g is a special function mapping 48-bit strings to 32-bit strings. The function g is specified by 8 "S-boxes", each being a table for a function to map a particular 6 bit piece of the input to a 4-bit piece of the output.

Because the key length of DES is so short, often one used TDES (for *triple DES*) instead. TDES has a $3 \cdot 56$ bit key and 64 bit block size, and is defined:
$TDES_{K_1 K_2 K_3} = DES_{K_1} \circ DES_{K_2} \circ DES_{K_3}$.

The Feistal construction can be used in a very nice way to construct a strongly pseudo-random permutation generator from a pseudo-random function generator.[1] The idea is to do four rounds of the construction, where instead of a simple f, we use a pseudo-random function generator on four independent keys.

Theorem: If there exists a pseudo-random function generator, then there exists a strongly pseudo-random permutation generator.

Construction: Let F be a pseudo-random function generator with key-length=block-size=n. Define the following permutation generator G. On block size $2n$, G will have keys of length $4n$. If $k_1, k_2, k_3, k_4 \in \{0,1\}^n$, $L_0, R_0 \in \{0,1\}^n$, then
$G_{k_1 k_2 k_3 k_4}(L_0, R_0) = (L_4, R_4)$ where
$(L_1, R_1) = (R_0, L_0 \oplus F_{k_1}(R_0))$,
$(L_2, R_2) = (R_1, L_1 \oplus F_{k_2}(R_1))$,
$(L_3, R_3) = (R_2, L_2 \oplus F_{k_3}(R_2))$,
$(L_4, R_4) = (R_3, L_3 \oplus F_{k_4}(R_3))$.

We will not prove here that this construction works.
As discussed above, G can be fixed up so that the key length equals the block size. It is harder to see that the construction can be modified so that G is defined on all block sizes (and not just even lengths) and is still a strongly pseudo-random permutation generator.

Also, if we are only interested in G being pseudo-random, then it suffices to do 3 rounds of the above construction, instead of 4. $\square$

Further Details About AES

(with only a few insignificant lies)

If the generator F is AES, then the block size is $n = 128$, the key size is $m = 128$, and the number of rounds is $r = 10$. (There are actually two other version, both having block size 128. One has key size 192 and 12 rounds, and the other has key size 256 and 14 rounds.) I will not discuss the key scheduling function KS here.

The permutation generator F' works as follows. The block size is $n = 128$ and the key size is $m' = 128$. F' is defined by
$F'_k(x) = g(x) \oplus k$, where g is a special one-one, onto, easily invertible function. It is easy to see that F'_k is easy to invert if one knows k. Note that unlike the generic Feistal method used in DES, the invertibility of F'_k comes from the invertibility of g, which itself happens because of the very ad-hoc, algebraic construction of g.

One inaccuracy in the above description is the following. In AES, before any application of F', the input is first exclusive-or'd with an extra round-key, k_0.

[1]Don't let this theorem confuse you into thinking we can convert a one-way function into a one-way permutation; in fact, we don't know how to do this.

One-Way Functions

We will now see that "one-way functions" exist if and only if pseudo-random generators exist. A one-way function is a function that is easy to compute but hard, on the average, to invert. The following is a formal definition in the "uniform adversary" setting. (The nonuniform setting would be similar, except that the adversary would be a polynomial size family $\{D_n\}$ of circuits where D_n has $\ell(n)$ input bits and n output bits.)

Definition: ("Uniform adversary" setting)

Let $f : \{0,1\}^* \rightarrow \{0,1\}^*$ be a function; for convenience, we assume that the length of $f(x)$ is determined by the length of x; say $|f(x)| = \ell(|x|)$. (It is also convenient to assume that the length of x is determined by the length of $f(x)$, that is, ℓ is one-one.) We say f is *one-way* if:

- f is computable in (deterministic) polynomial time, and
- The following holds for every A:

  Let A be a probabilistic, polynomial time algorithm. We assume that for every n, if A is given a string of length $\ell(n)$, then A outputs a string of length n.
  For each n, define
  $q_A(n)$ = the probability that if x is randomly chosen from $\{0,1\}^n$ and A is run on $f(x)$ and A outputs β, then $f(\beta) = f(x)$.
  THEN for every c and sufficiently large n, $q_A(n) \leq \frac{1}{n^c}$.

Remark: It is not too hard to see that if a one-way function f exists, we can convert it to a one-way function f' such that $|f'(x)| = f(x)$.

Theorem: One-way functions exist $\Longleftrightarrow$ pseudo-random generators exits.

Proof of $\Longrightarrow$: (Håstad, Impagliazzo, Levin, Luby) We will do part of this proof later.

Proof of $\Longleftarrow$: ("Uniform adversary setting.")

Let G be a pseudo-random number generator with length function $\ell(n) = 2n$. We claim G is one-way.

To see this, assume to the contrary that A is an algorithm breaking the one-wayness of G; let $\{q_A(n)\}$ be the inversion probabilities for A, as in the definition of one-way function. Define the following distinguisher D for G:
On input α where $|\alpha| = 2n$, compute $\beta = A(\alpha)$, and see if $G(\beta) = \alpha$; if so, accept, and if not, reject.

Fix n. It is easy to see that the probability that D accepts when α is chosen to be $G(s)$ for a random n bit string s, is $q_A(n)$. Since there are at most 2^n possible values for $G(s)$, A can invert at most 2^n values of α. So if a random $2n$ bit α is chosen, then the probability D accepts α is $\leq \frac{2^n}{2^{2n}} = \frac{1}{2^n}$.

Because A breaks the one-wayness of G, $q_A(n) > \frac{1}{n^c}$ for some c and infinitely many n, so D breaks the pseudo-randomness of G. $\square$

Both pseudo-random generators and one-way functions have many applications in cryptography. The above theorem can be used in (at least) two different ways. One way is $\Longrightarrow$: one starts with a computational complexity assumption about (for example) number theory, which can be used to get a one-way function, and this in turn is used to construct pseudo-random generators. The other way, which is much more common, is $\Longleftarrow$: one constructs from scratch a generator that is (hopefully) pseudo-random (such as DES or AES), and uses it to construct a one-way function.

An example of the latter approach is the way DES was used to get a (hopefully) one-way function in the old UNIX password scheme. The scheme, essentially, worked like this:
A user U chooses a random 56 bit DES seed s as his password; the pair $(U, DES_s(\bar{0}))$ is then stored in a *publicly viewable* password file. When someone later tries to login in with name U and password x, the system checks that $(U, DES_x(\bar{0}))$ is in the password file. Clearly, therefore, we want the function f that maps s to $DES_s(\bar{0})$ to be one-way. We want to be able to *prove* that the one-wayness of f follows from the pseudo-randomness of DES.

The above proof does *not* show this, however. Assuming DES is a pseudo-random function generator, we know that the function that maps s to $DES_s(\bar{0})DES_s(\bar{1})$ is a pseudo-random number generator that expands its seed by at least a factor of 2. We therefore know that we can get a one-way function by mapping s to at least the first 112 bits of $DES_s(\bar{0})DES_s(\bar{1})$. However, UNIX merely includes in the password file $64 = 56 + 8$ bits of encryption. If we look at the above proof carefully, we see that it leaves open the possibility that we cannot effectively break the pseudo-randomness of DES, but that we *can* easily break the password scheme for about one out of every $2^8 = 256$ keys, a true disaster.

The following theorem asserts the security of an improved construction of a one-way function from a pseudo-random number generator. This theorem justifies the old UNIX password scheme (even though the theorem came about much later). In particular, it is sufficient to start with an arbitrary pseudo-random number generator – the output may exceed the seed length by only one bit, for example. This function is then one-way, even if one truncates the output to be not too much smaller than the seed length.

Theorem: Let G be a pseudo-random number generator with length function $\ell_1(n) > n$. Let $\ell_2(n)$ be a function that is computable in time polynomial in n, such that $\ell_2(n) \leq \ell_1(n)$ and such that for some constant d and sufficiently large n, $\ell_2(n) \geq n - d \log_2 n$. Define the function f on input x of length n by $f(x) =$ the leftmost $\ell_2(n)$ bits of $G(x)$.
Then f is one-way.

Proof: Omitted. This is a hardish exercise.

It turns out that if we allow $\ell_2(n)$ to be much smaller than in the above theorem, then the theorem becomes false. For instance, we leave it as an exercise to show that (assuming pseudo-random generators exist) there exists a pseudo-random number generator G such that if we define $f(x) =$ the first $\lfloor |x|/2 \rfloor$ bits of $G(x)$, then f is not one-way.

Constructing pseudo-random generators from one-way functions

We have stated above the theorem that if there are one-way functions, then there are pseudo-random generators.

Before discussing this proof, we should point out that there is a notion of *weakly* one-way function, whose existence is still sufficient to yield one-way functions. We define the function f to be *weakly one-way* if there is some d such that for every adversary A, if we define $q_A(n)$ as in the definition of one-way, then for sufficiently large n, $q_A(n) \leq 1 - \frac{1}{n^d}$. (Contrast this with the stipulation, in the definition of one-way, that for every A and every c and sufficiently large n, $q_A(n) \leq \frac{1}{n^c}$.)

Lemma: If there is a weakly one-way function, then there is a one-way function.

Construction: Let f be weakly one-way such that for every adversary A, $q_A(n) \leq 1 - \frac{1}{n^d}$. The idea is that if it should be very hard to invert f on many, independently chosen inputs – in particular on n^{d+1} inputs. Define the function f' by, for each n, for $u = n^{d+1}$, for n-bit strings $x_1, x_2, \ldots, x_u$, $f'(x_1 x_2 \ldots x_u) = f(x_1)f(x_2)\ldots f(x_u)$.

We claim f' is one-way. The intuition is that in order to invert f', one must invert f on n^{d+1} independent inputs. If one tried to do all of these inversions independently of each other, the probability of success would be $\leq (1 - \frac{1}{n^d})^{n^{d+1}}$, which is about $(1/e)^n$. The actual proof, however, is surprisingly difficult, and is omitted here. $\square$

The reader will note that the function f' constructed above has not been defined on inputs of every length, but only on *special* lengths of the form n^{d+2}. We can get around this as follows: Define the function f'' on all lengths by $f''(\alpha) = f'(\beta)$ where β is the largest prefix of α of special length. We claim that if f' is one-way, then f'' is one-way (exercise).

The full proof (and even the related construction) that if there exist one-way functions then there exist pseudo-random generators is very complicated, and some of the details are in *Goldreich.*

For now, we will outline the proof of the much weaker theorem that states that if there are one-way "permutations" then there are pseudo-random generators.

Definition: A one-way *permutation* is a one-way function f such that $|f(x)| = |x|$ for every x, and such that for every n, f maps $\{0,1\}^n$ one-one, onto $\{0,1\}^n$.

In order to construct a pseudo-random generator, we will need a one-way permutation that comes with a "hard-core predicate". In general, it makes sense to define a hard-core predicate for any easily computable one-one function whose length depends only on the length of the input.

Definition: ("nonuniform adversary" setting)
Let $f : \{0,1\}^* \to \{0,1\}^*$ be a polynomial time computable, one-one function where for some function $\ell(n)$, $|f(x)| = \ell(|x|)$ for every input x. Let $B : \{0,1\}^* \to \{0,1\}$. We say *$B$ is a hard-core predicate for f* if B is polynomial time computable and if the following holds for every C:

Let $C = \{C_n\}$ be a polynomial size family of circuits where C_n has $\ell(n)$ input bits and one output bit. For each n, define
$q_C(n) =$ the probability that if x is randomly chosen from $\{0,1\}^n$ and C_n is run on $f(x)$, then C_n outputs $B(x)$.
THEN for every c and sufficiently large n, $q_C(n) \leq \frac{1}{2} + \frac{1}{n^c}$.

It is not hard to see (Exercise!) that if f is a permutation with a hard-core predicate, then f must be one-way. The following is the converse of this.

Lemma: If there is a one-way permutation f, then there is a one-way permutation f' and function B' such that B' is a hard-core predicate for f'.

Construction: Let f be a one-way permutation, and fix n. We will describe two different constructions here, but we will not prove that either of them works.

The first construction is due to Yao, although the best proof is due to Levin. The idea is as follows. If f is hard to invert, then one of the *literal* bits of x should be at least slightly hard to compute given $f(x)$. (A *literal* bit of x is either the first bit, or the second bit, etc.) By *slightly* hard, we mean that a polynomial time adversary will not be able to compute that bit (given $f(x)$)

with probability better than $1 - \frac{1}{2n}$ (exercise). Assume that the first bit of x is slightly hard to compute in this way. We would then let f' be the concatenation of the results of evaluating f on n^2 inputs, and let B' of those inputs be the *exclusive-or* of the first bits of those inputs. It is easy to calculate that if an adversary tried to compute B' by independently trying to find each of those bits and then taking the exclusive-or of those guesses, the probability of success would be exponentially (in $1/n$) close to $1/2$. It turns out, but is hard to show, that even if the adversary is not constrained to this way of computing B', he still won't succeed with probability much higher than $1/2$. Since we don't know which of the n literal bits is slightly hard, we have to do the above independently for each bit, and then do a huge exclusive-or. More formally:

f' will have as input a string of length n^4 consisting of n^3 strings of length n; we will write this input as $\{x_{i,j}\}$ where $1 \leq i \leq n$ and $1 \leq j \leq n^2$. $f'(\{x_{i,j}\})$ will be the n^4 bit string consisting of the concatenation of the values of each $f(x_{i,j})$. (It is not hard to fix up this construction so that f' is defined on strings of any length.)

We define $B(\{x_{i,j}\})$ to be the exclusive-or, over all i and j, of the i-th bit of $x_{i,j}$.

The second construction is due to Goldreich and Levin, and is discussed in Theorem 2.5.2 of *Goldreich*. f' will have as input two strings x and r, each of length n. We define $f'(x, r) = [f(x), r]$ (It is not hard to fix up this construction so that f' is defined on strings of any length.) We define $B'(x, r)$ to be the *inner-product* mod 2 of the binary vectors x and r. That is, if the n bits of x are $x^1, x^2, \ldots, x^n$ and the n bits of r are $r^1, r^2, \ldots, r^n$, then $B'(x, r) = (x^1 \cdot r^1) \oplus \ldots \oplus (x^n \cdot r^n)$. This is a very elegant and efficient and useful construction, but we omit the (very interesting) proof that B' is hard-core for f'. $\square$

Theorem:
If there are one-way permutations, then there are pseudo-random generators.

Proof: Using the previous lemma, we can assume we have a one-way permutation f and a hard-core predicate B for f. Define the number generator $G(s) = f(s)B(s)$; clearly G has length function $l(n) = n + 1$.

We claim G is pseudo-random. To show this, we know from earlier notes that it is sufficient to show "unpredictability" (from the left). That is, we want to show that if s is chosen randomly from $\{0, 1\}^n$, then none of the bits of $G(s)$ can be predicted from the previous bits, with probability significantly above chance. Fix n. Since f is a permutation, as s varies randomly amongst all n bit strings, $f(s)$ also varies randomly amongst all n bit strings. Hence, none of the first n bits of $G(s)$ can be predicted. The last bit of $G(s)$ also cannot be predicted, because B is hard-core: any polynomial time adversary predicts $B(s)$ after seeing $f(s)$ (for every c and sufficiently large n) with probability $\leq \frac{1}{2} + \frac{1}{n^c}$. $\square$

We have not presented the most general construction of a pseudo-random generator from an arbitrary one-way function. It is important to realize, however, that often it is not necessary to do the general construction because the (supposedly) one-way function we start with may have special properties that allow us to short-circuit the general construction. For example, we have just seen that if the one-way function f we start with is a permutation, then a very simple construction yields a pseudo-random number generator with length function $l(n) = n + 1$.

If we want to use this pseudo-random number generator to construct a pseudo-random *function* generator at this point, we *could* do it by using the exact construction described earlier. There is, however, a better way. Recall that our number generator G maps (x, r) to $(f(x), r, B(x, r))$ where $B(x, r)$ is the inner product mod 2 of x and r. We can now use an earlier construction to create a new number generator G' where for n bit strings x_0 and r, we define $G'(r, x_0) = [r, x_n, B(x_0, r), B(x_1, r), \cdots, B(x_{n-1}, r)]$, where we define $x_{i+1} = f(x_i)$. G' maps a $2n$ bit seed to a $3n$ bit string. Superficially, G' isn't strong enough to use in our tree-like construction of a pseudo-random function generator, since the output length is not double the input length. However, we note that r appears (unchanged) in both the input and output of G'. If we choose a random r and then consider the mapping G'_r which takes x to $G'(r, x)$, we have a mapping of n bits to $2n$ bits which looks pseudo-random even to adversaries who know r. We can therefore construct the function generator F: for a key consisting of an n bit string r and an n bit string s, consider (although it is too big to actually construct) the binary tree of height n with s at the root so that the children of every node are determined by the function G'_r; for an n bit string σ we then define $F_{r,s}(\sigma)$ to be the string at the node obtained by following the path σ. We can prove that the pseudo-randomness of G' implies the pseudo-randomness of F.