3.6 Pseudorandom Functions

Pseudorandom generators enable to generate, exchange and share a large number of pseudorandom values at the cost of a much smaller number of random bits. Specifically, poly(n) pseudorandom bits can be generated, exchanged and shared at the cost of n (uniformly chosen bits). Since any efficient application uses only a polynomial number of random values, providing access to polynomially many pseudorandom entries seems sufficient. However, the above conclusion is too hasty, since it assumes implicitly that these entries (i.e., the addresses to be accessed) are fixed beforehand. In some natural applications, one may need to access addresses which are determined "dynamically" by the application. For example, one may want to assign random values to (poly(n) many) n-bit long strings, produced throughout the application, so that these values can be retrieved at latter time. Using pseudorandom generators the above task can be achieved at the cost of generating n random bits and storing poly(n) many values. The challenge, met in the sequel, is to achieve the above task at the cost of generating and storying only n random bits. The key to the solution is the notion of pseudorandom functions. In this section we define pseudorandom functions and show how to efficiently implement them. The implementation uses as a building block any pseudorandom generator.

3.6.1 Definitions

Loosely speaking, pseudorandom functions are functions which cannot be distinguished from truly random functions by any efficient procedure which can get the value of the function at arguments of its choice. Hence, the distinguishing procedure may query the function being examined at various points, depending possibly on previous answers obtained, and yet can not tell whether the answers were supplied by a function taken from the pseudorandom ensemble (of functions) or from the uniform ensemble (of function). Hence, to formalize the

notion of pseudorandom functions we need to consider ensembles of functions. For sake of concreteness we consider in the sequel ensembles of length preserving functions. Extensions are discussed in Exercise 23.

Definition 3.6.1 (function ensembles): *A* function ensemble *is a sequence $F = \{F_n\}_{n\in\mathbb{N}}$ of random variables, so that the random variable F_n assumes values in the set of functions mapping n-bit long strings to n-bit long strings. The* uniform function ensemble*, denoted $H = \{H_n\}_{n\in\mathbb{N}}$, has H_n uniformly distributed over the set of functions mapping n-bit long strings to n-bit long strings.*

To formalize the notion of pseudorandom functions we use (probabilistic polynomial-time) *oracle machines*. We stress that our use of the term oracle machine is almost identical to the standard one. One deviation is that the oracle machines we consider have a length preserving function as oracle rather than a Boolean function (as is standard in most cases in the literature). Furthermore, we assume that on input 1^n the oracle machine only makes queries of length n. These conventions are not really essential (they merely simplify the exposition a little).

Definition 3.6.2 (pseudorandom function ensembles): *A function ensemble, $F = \{F_n\}_{n\in\mathbb{N}}$, is called* pseudorandom *if for every probabilistic polynomial-time oracle machine M, every polynomial $p(\cdot)$ and all sufficiently large n's*

$$|\mathsf{Pr}(M^{F_n}(1^n)=1) - \mathsf{Pr}(M^{H_n}(1^n)=1)| < \frac{1}{p(n)}$$

where $H = \{H_n\}_{n\in\mathbb{N}}$ is the uniform function ensemble.

Using techniques similar to those presented in the proof of Proposition 3.2.3 (of Subsection 3.2.2), one can demonstrate the existence of pseudorandom function ensembles which are not statistically close to the uniform one. However, to be of practical use, we need require that the pseudorandom functions can be efficiently computed.

Definition 3.6.3 (efficiently computable function ensembles): *A function ensemble, $F = \{F_n\}_{n\in\mathbb{N}}$, is called* efficiently computable *if the following two conditions hold*

1. (efficient indexing): *There exists a probabilistic polynomial time algorithm, I, and a mapping from strings to functions, ϕ, so that $\phi(I(1^n))$ and F_n are identically distributed.*
   We denote by f_i the $\{0,1\}^n \mapsto \{0,1\}^n$ function assigned to i (i.e., $f_i \stackrel{\text{def}}{=} \phi(i)$).

2. (efficient evaluation): *There exists a probabilistic polynomial time algorithm, V, so that $V(i,x) = f_i(x)$.*

In particular, functions in an efficiently computable function ensemble have relatively succinct representation (i.e., of polynomial rather than exponential length). It follows that efficiently computable function ensembles may have only exponentially many functions (out of the double-exponentially many possible functions).

Another point worthy of stressing is that pseudorandom functions may (if being efficiently computable) be efficiently evaluated at given points, *provided that the function description is give as well.* However, if the function (or its description) is *not* known (and it is only known that it is chosen from the pseudorandom ensemble) then the value of the function at a point cannot be approximated (even in a very liberal sense and) even if the values of the function at other points is also given.

In the rest of this book we consider only efficiently computable pseudorandom functions. Hence, in the sequel we sometimes shorthand such ensembles by calling them pseudorandom functions.

3.6.2 Construction

Using any pseudorandom generator, we construct a (efficiently computable) pseudorandom function (ensemble).

Construction 3.6.4 *Let G be a deterministic algorithm expanding inputs of length n into strings of length $2n$. We denote by $G_0(s)$ the $|s|$-bit long prefix of $G(s)$, and by $G_1(s)$ the $|s|$-bit long suffix of $G(s)$ (i.e., $G(s) = G_0(s)G_1(s)$). For every $s \in \{0,1\}^n$, we define a function $f_s : \{0,1\}^n \mapsto \{0,1\}^n$ so that for every $\sigma_1, ..., \sigma_n \in \{0,1\}$*

$$f_s(\sigma_1\sigma_2\cdots\sigma_n) \stackrel{\text{def}}{=} G_{\sigma_n}(\cdots(G_{\sigma_2}(G_{\sigma_1}(s)))\cdots)$$

Let F_n be a random variable defined by uniformly selecting $s \in \{0,1\}^n$ and setting $F_n = f_s$. Finally, let $F = \{F_n\}_{n\in\mathbb{N}}$ be our function ensemble.

Pictorially, the function f_s is defined by n-step walks down a full binary tree of depth n having labels on the vertices. The root of the tree, hereafter referred to as the level 0 vertex of the tree, is labelled by the string s. If an internal node is labelled r then its left child is labelled $G_0(r)$ whereas its right child is labelled $G_1(r)$. The value of $f_s(x)$ is the string residing in the leaf reachable from the root by a path corresponding to string x, when the root is labelled by s. The random variable F_n is assigned labelled trees corresponding to all possible 2^n labellings of the root, with uniform probability distribution.

A function, operating on n-bit strings, in the ensemble constructed above can be specified by n bits. Hence, selecting, exchanging and storing such a function can be implemented at the cost of selecting, exchanging and storing a single n-bit string.

Theorem 3.6.5 *Let G and F be as in Construction 3.6.4, and suppose that G is a pseudorandom generator. Then F is an efficiently computable ensemble of pseudorandom functions.*

Proof: Clearly, the ensemble F is efficiently computable. To prove that F is pseudorandom we use the hybrid technique. The k^{th} hybrid will be assigned functions which result by uniformly selecting labels for the vertices of the k^{th} (highest) level of the tree and computing the labels of lower levels as in Construction 3.6.4. The 0-hybrid will correspond to the random variable F_n (since a uniformly chosen label is assigned to the root), whereas the n-hybrid will correspond to the uniform random variable H_n (since a uniformly chosen label is assigned to each leaf). It will be shown that an efficient oracle machine distinguishing neighbouring hybrids can be transformed into an algorithm that distinguishes polynomially many samples of $G(U_n)$ from polynomially many samples of U_{2n}. Using Theorem 3.2.6 (of Subsection 3.2.3), we derive a contradiction to the hypothesis (that G is a pseudorandom generator). Details follows.

For every k, $0 \leq k \leq n$, we define a hybrid distribution H_n^k (assigned as values functions $f : \{0,1\}^n \mapsto \{0,1\}^n$) as follows. For every $s_1, s_2, ..., s_{2^k} \in \{0,1\}^n$, we define a function $f_{s_1,...,s_{2^k}} : \{0,1\}^n \mapsto \{0,1\}^n$ so that

$$f_{s_1,...,s_{2^k}}(\sigma_1\sigma_2\cdots\sigma_n) \stackrel{\text{def}}{=} G_{\sigma_n}(\cdots(G_{\sigma_{k+2}}(G_{\sigma_{k+1}}(s_{\text{idx}(\sigma_k\cdots\sigma_1)}))\cdots)$$

where $\text{idx}(\alpha)$ is index of α in the standard lexicographic order of strings of length $|\alpha|$. (In the sequel we take the liberty of associating the integer $\text{idx}(\alpha)$ with the string α.) Namely, $f_{s_{0^k},...,s_{1^k}}(x)$ is computed by first using the k-bit long prefix of x to determine one of the s_j's, and next using the $(n-k)$-bit long suffix of x to determine which of the functions G_0 and G_1 to apply at each remaining stage. The random variable H_n^k is uniformly distributed over the above $(2^n)^{2^k}$ possible functions. Namely,

$$H_n^k \stackrel{\text{def}}{=} f_{U_n^{(1)},...,U_n^{(2^k)}}$$

where $U_n^{(j)}$'s are independent random variables each uniformly distributed over $\{0,1\}^n$.

At this point it is clear that H_n^0 is identical to F_n, whereas H_n^n is identical to H_n. Again, as usual in the hybrid technique, ability to distinguish the extreme hybrids yields ability to distinguish a pair of neighbouring hybrids. This ability is further transformed (as sketched above) so that contradiction to the pseudorandomness of G is reached. Further details follow.

We assume, in contradiction to the theorem, that the function ensemble F is not pseudorandom. It follows that there exists a probabilistic polynomial-time oracle machine, M, and a polynomial $p(\cdot)$ so that for infinitely many n's

$$\Delta(n) \stackrel{\text{def}}{=} |\mathsf{Pr}(M^{F_n}(1^n)=1) - \mathsf{Pr}(M^{H_n}(1^n)=1)| > \frac{1}{p(n)}$$

Let $t(\cdot)$ be a polynomial bounding the running time of $M(1^n)$ (such a polynomial exists since M is polynomial-time). It follows that, on input 1^n, the oracle machine M makes at most $t(n)$ queries (since the number of queries is clearly bounded by the running time).

Using the machine M, we construct an algorithm D that distinguishes the $t(\cdot)$-product of the ensemble $\{G(U_n)\}_{n\in\mathbb{N}}$ from the $t(\cdot)$-product of the ensemble $\{U_{2n}\}_{n\in\mathbb{N}}$ as follows.

On input $\alpha_1, ..., \alpha_t \in \{0,1\}^{2n}$ (with $t = t(n)$), algorithm D proceeds as follows. First, D selects uniformly $k \in \{0, 1, ..., n-1\}$. This random choice, hereafter called the *checkpoint*, and is the only random choice made by D itself. Next, algorithm D invokes the oracle machine M (on input 1^n) and answers M's queries as follows. The first query of machine M, denoted q_1, is answered by

$$G_{\sigma_n}(\cdots(G_{\sigma_{k+2}}(P_{\sigma_{k+1}}(\alpha_1)))\cdots)$$

where $q_1 = \sigma_1 \cdots \sigma_n$, and $P_0(\alpha)$ denotes the n-bit prefix of α (and $P_1(\alpha)$ denotes the n-bit suffix of α). In addition, algorithm D records this query (i.e., q_1). Subsequent queries are answered by first checking if their k-bit long prefix equals the k-bit long prefix of a previous query. In case the k-bit long prefix of the current query, denoted q_i, is different from the k-bit long prefixes of all previous queries, we associate this prefix a new input string (i.e., α_i). Namely, we answer query q_i by

$$G_{\sigma_n}(\cdots(G_{\sigma_{k+2}}(P_{\sigma_{k+1}}(\alpha_i)))\cdots)$$

where $q_i = \sigma_1 \cdots \sigma_n$. In addition, algorithm D records the current query (i.e., q_i). The other possibility is that the k-bit long prefix of the i^{th} query equals the k-bit long prefix of some previous query. Let j be the smallest integer so that the k-bit long prefix of the i^{th} query equals the k-bit long prefix of the j^{th} query (by hypothesis $j < i$). Then, we record the current query (i.e., q_i) but answer it using the string associated with query q_j. Namely, we answer query q_i by

$$G_{\sigma_n}(\cdots(G_{\sigma_{k+2}}(P_{\sigma_{k+1}}(\alpha_j)))\cdots)$$

where $q_i = \sigma_1 \cdots \sigma_n$. Finally, when machine M halts, algorithm D halts as well and outputs the same output as M.

Pictorially, algorithm D answers the first query by first placing the two halves of α_1 in the corresponding children of the tree-vertex reached by following the path from the root corresponding to $\sigma_1 \cdots \sigma_k$. The labels of all vertices in the subtree corresponding to $\sigma_1 \cdots \sigma_k$ are determined by the labels of these two children (as in the construction of F). Subsequent queries are answered by following the corresponding paths from the root. In case the path does not pass through a $(k+1)$-level vertex which has already a label, we assign this vertex and its sibling a new string (taken from the input). For sake of simplicity, in case the path of the i^{th} query requires a new string we use the i^{th} input string (rather than the first input string not used so far). In case the path of a new query passes through a $(k+1)$-level vertex which has been labelled already, we use this label to compute the labels of subsequent vertices along this path (and in particular the label of the leaf). We stress that the algorithm *does not* necessarily compute the labels of all vertices in a subtree corresponding to $\sigma_1 \cdots \sigma_k$ (although these labels are determined by the label of the vertex corresponding to $\sigma_1 \cdots \sigma_k$), but rather computes only the labels of vertices along the paths corresponding to the queries.

Clearly, algorithm D can be implemented in polynomial-time. It is left to evaluate its performance. The key observation is that when the inputs are taken from the $t(n)$-product of $G(U_n)$ and algorithm D chooses k as the checkpoint then M behaves exactly as on the k^{th} hybrid. Likewise, when the inputs are taken from the $t(n)$-product of U_{2n} and algorithm D chooses k as the checkpoint then M behaves exactly as on the $k+1^{\text{st}}$ hybrid. Namely,

Claim 3.6.5.1: Let n be an integer and $t \stackrel{\text{def}}{=} t(n)$. Let K be a random variable describing the random choice of checkpoint by algorithm D (on input a t-long sequence of $2n$-bit long strings). Then for every $k \in \{0,1,...,n-1\}$

$$\begin{aligned} \mathsf{Pr}\left(D(G(U_n^{(1)}),...,G(U_n^{(t)}))=1 \mid K=k\right) &= \mathsf{Pr}\left(M^{H_n^k}(1^n)=1\right) \\ \mathsf{Pr}\left(D(U_{2n}^{(1)},...,U_{2n}^{(t)})=1 \mid K=k\right) &= \mathsf{Pr}\left(M^{H_n^{k+1}}(1^n)=1\right) \end{aligned}$$

where the $U_n^{(i)}$'s and $U_{2n}^{(j)}$'s are independent random variables uniformly distributed over $\{0,1\}^n$ and $\{0,1\}^{2n}$, respectively.

The above claim is quite obvious, yet a rigorous proof is more complex than one realizes at first glance. The reason being that M's queries may depend on previous answers it gets, and hence the correspondence between the inputs of D and possible values assigned to the hybrids is less obvious than it seems. To illustrate the difficulty consider a n-bit string which is selected by a pair of interactive processes, which proceed in n iterations. At each iteration the first party chooses a new location, based on the entire history of the interaction, and the second process sets the value of this bit by flipping an unbiased coin. It is intuitively clear that the resulting string is uniformly distributed, and the same holds if the second party sets the value of the chosen locations using the outcome of a coin flipped beforehand. In our setting the situation is slightly more involved. The process of determining the string is terminated after $k < n$ iterations and statements are made of the partially determined string. Consequently, the situation is slightly confusing and we feel that a detailed argument is required.

Proof of Claim 3.6.5.1: We start by sketching a proof of the claim for the extremely simple case in which M's queries are the first t strings (of length n) in lexicographic order. Let us further assume, for simplicity, that on input $\alpha_1,...,\alpha_t$, algorithm D happens to choose checkpoint k so that $t = 2^{k+1}$. In this case the oracle machine M is invoked on input 1^n and access to the function $f_{s_1,...,s_{2^{k+1}}}$, where $s_{2j-1+\sigma} = P_\sigma(\alpha_j)$ for every $j \leq 2^k$ and $\sigma \in \{0,1\}$. Thus, if the inputs to D are uniformly selected in $\{0,1\}^{2n}$ then M is invoked with access to the $k+1^{\text{st}}$ hybrid random variable (since in this case the s_j's are independent and uniformly distributed in $\{0,1\}^n$). On the other hand, if the inputs to D are distributed as $G(U_n)$ then M is invoked with access to the k^{th} hybrid random variable (since in this case $f_{s_1,...,s_{2^{k+1}}} = f_{r_1,...,r_{2^k}}$ where the r_j's are seeds corresponding to the α_j's).

For the general case we consider an alternative way of defining the random variable H_n^m, for every $0 \leq m \leq n$. This alternative way is somewhat similar to the way in which

D answers the queries of the oracle machine M. (We use the symbol m instead of k since m *does not* necessarily equal the checkpoint, denoted k, chosen by algorithm D.) This way of defining H_n^m consists of the interleaving of two random processes, which together first select at random a function $g:\{0,1\}^m \mapsto \{0,1\}^n$, that is later used to determine a function $f:\{0,1\}^n \mapsto \{0,1\}^n$. The first random process, denoted ρ, is an arbitrary process ("given to us from the outside"), which specifies points in the domain of g. (The process ρ corresponds to the queries of M, whereas the second process corresponds to the way A answers these queries.) The second process, denoted ψ, assigns uniformly selected n-bit long strings to every new point specified by ρ, thus defining the value of g on this point. We stress that in case ρ specifies an old point (i.e., a point for which g is already defined) then the second process does nothing (i.e., the value of g at this point is left unchanged). The process ρ may depend on the history of the two processes, and in particular on the values chosen for the previous points. When ρ terminates the second process (i.e., ψ) selects random values for the remaining undefined points (in case such exist). We stress that the second process (i.e., ψ) is fixed for all possible choices of a ("first") process ρ. The rest of this paragraph gives a detailed description of the interleaving of the two random processes (*and may be skipped*). We consider a randomized process ρ mapping sequences of n-bit strings (representing the history) to single m-bit strings. We stress that ρ is *not* necessarily memoryless (and hence may "remember" its previous random choices). Namely, for every fixed sequence $v_1, ..., v_i \in \{0,1\}^n$, the random variable $\rho(v_1, ..., v_i)$ is (arbitrarily) distributed over $\{0,1\}^m \cup \{\perp\}$ where $\perp$ is a special symbol denoting termination. A "random" function $g:\{0,1\}^m \mapsto \{0,1\}^n$ is defined by iterating the process ρ with the random process ψ defined below. Process ψ starts with g which is undefined on every point in its domain. At the i^{th} iteration ψ lets $p_i \stackrel{\text{def}}{=} \rho(v_1, ..., v_{i-1})$ and, assuming $p_i \neq \perp$, sets $v_i \stackrel{\text{def}}{=} v_j$ if $p_i = p_j$ for some $j < i$ and lets v_i be uniformly distributed in $\{0,1\}^n$ otherwise. In the latter case (i.e., p_i is new and hence g is not yet defined on p_i), ψ sets $g(p_i) \stackrel{\text{def}}{=} v_i$ (in fact $g(p_i) = g(p_j) = v_j = v_i$ also in case $p_i = p_j$ for some $j < i$). When ρ terminates, i.e., $\rho(v_1, ..., v_T) = \perp$ for some T, ψ completes the function g (if necessary) by choosing independently and uniformly in $\{0,1\}^n$ values for the points at which g is undefined yet. (Alternatively, we may augment the process ρ so that it terminates only after specifying all possible m-bit strings.)

Once a function g is totally defined, we define a function $f^g:\{0,1\}^n \mapsto \{0,1\}^n$ by

$$f^g(\sigma_1\sigma_2\cdots\sigma_n) \stackrel{\text{def}}{=} G_{\sigma_n}(\cdots(G_{\sigma_{k+2}}(G_{\sigma_{k+1}}(g(\sigma_k\cdots\sigma_1)))\cdots)$$

The reader can easily verify that f^g equals $f_{g(0^m),...,g(1^m)}$ (as defined in the hybrid construction above). Also, one can easily verify that the above random process (i.e., the interleaving of ψ with any ρ) yields a function g that is uniformly distributed over the set of all possible functions mapping m-bit strings to n-bit strings. It follows that the above described random process yields a result (i.e., a function) that is distributed identically to the random variable H_n^m.

Suppose now that the checkpoint chosen by D equals k and that D's inputs are independently and uniformly selected in $\{0,1\}^{2n}$. In this case the way in which D answers the

M's queries can be viewed as placing independently and uniformly selected n-bit strings as the labels of the $(k+1)$-level vertices. It follows that the way in which D answers M's queries corresponds to the above described process with $m = k+1$ (with M playing the role of ρ and A playing the role of ψ). Hence, in this case M is invoked with access to the $k+1^{\text{st}}$ hybrid random variable.

Suppose, on the other hand, that the checkpoint chosen by D equals k and that D's inputs are independently selected so that each is distributed identically to $G(U_n)$. In this case the way in which D answers the M's queries can be viewed as placing independently and uniformly selected n-bit strings as the labels of the k-level vertices. It follows that the way in which D answers the M's queries corresponds to the above described process with $m = k$. Hence, in this case M is invoked with access to the k^{th} hybrid random variable. □

Using Claim 3.6.5.1, it follows that

$$|\mathsf{Pr}\left(D(G(U_n^{(1)}),...,G(U_n^{(t)}))=1\right) - \mathsf{Pr}\left(D(U_{2n}^{(1)},...,U_{2n}^{(t)})=1\right)| = \frac{\Delta(n)}{n}$$

which, by the contradiction hypothesis is greater than $\frac{1}{n \cdot p(n)}$, for infinitely many n's. Using Theorem 3.2.6, we derive a contradiction to the hypothesis (of the current theorem) that G is a pseudorandom generator, and the current theorem follows. ■

3.6.3 A general methodology

Author's Note: *Ellaborate on the following.*

The following two-step methodology is useful in many cases:

1. Design your scheme assuming that all legitimate users share a random function, $f: \{0,1\}^n \mapsto \{0,1\}^n$. (The adversaries may be able to obtain, from the legitimate users, the values of f on arguments of their choice, but do not have direct access to f.) This step culminates in proving the security of the scheme assuming that f is indeed uniformly chosen among all possible such functions, while ignoring the question of how such an f can be selected and handled.

2. Construct a real scheme by replacing the random function by a pseudorandom function. Namely, the legitimate users will share a random/secret seed specifying such a pseudorandom function, whereas the adversaries do not know the seed. As before, at most the adversaries may obtain (from the legitimate users) the value of the function at arguments of their choice. Finally, conclude that the real scheme (as presented above) is secure (since otherwise one could distinguish a pseudorandom function from a truly random one).

We stress that the above methodology may be applied only if legitimate users can share random/secret information not known to the adversary (i.e., as is the case in private-key encryption schemes).