How to Construct Pseudorandom Functions
Extracts from Foundations of Cryptography[1]

Oded Goldreich
Department of Computer Science
Weizmann Institute of Science
Rehovot, ISRAEL.

February 8, 2001

[1]This is Section 3.6 of the book *Foundations of Cryptography*. For further details on the book, see webpage `http://www.wisdom.weizmann.ac.il/~oded/foc.html`.

3.6 Pseudorandom Functions

In this section we present definitions and constructions for pseudorandom functions (using any pseudorandom generator as a building block). Pseudorandom functions will be instrumental to some construction to be presented in Chapters 5 and 6.

Motivation: Recall that 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 for any such application. The latter 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 that 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 this section, is to achieve the above task at the cost of generating only n random bits *and storing only n bits*. The key to the solution is the notion of pseudorandom functions. Intuitively, a pseudorandom function shared by a group of users provides them with a function that appears random to adversaries (outside of the group).

3.6.1 Definitions

Loosely speaking, pseudorandom functions are functions that cannot be distinguished from truly random functions by any efficient procedure that 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). Indeed, to formalize the notion of pseudorandom functions we need to consider ensembles of functions. For sake of simplicity, we consider in the sequel ensembles of length preserving functions, and the reader is encouraged to further simplify the discussion by setting $\ell(n) = n$ (below). Generalizations are discussed in Section 3.6.4.

Definition 3.6.1 (function ensembles): *Let $\ell : \mathbb{N} \rightarrow \mathbb{N}$ (e.g., $\ell(n) = n$). An* ℓ-bit 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 $\ell(n)$-bit long strings to $\ell(n)$-bit long strings. The* uniform ℓ-bit function ensemble, *denoted $H = \{H_n\}_{n\in\mathbb{N}}$, has H_n uniformly distributed over the set of all functions mapping $\ell(n)$-bit long strings to $\ell(n)$-bit long strings.*

To formalize the notion of pseudorandom functions we use (probabilistic polynomial-time) *oracle machines* (see Section **??**). We stress that our use of the term oracle machine is almost identical to the standard one. One minor deviation is that the oracle machines we consider have a length preserving function as oracle rather than a Boolean function (as is more standard in complexity theory). Furthermore, we assume that on input 1^n the oracle machine only makes queries of length $\ell(n)$. These conventions are not really essential (they merely simplify the exposition a little). We let M^f denote the execution of the oracle machine M when given access to the oracle f.

Definition 3.6.2 (pseudorandom function ensembles): *An ℓ-bit 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*

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

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

Using techniques similar to those presented in the proof of Proposition **??** (of Subsection **??**), one can demonstrate the existence of pseudorandom function ensembles that are not statistically close to the uniform one. However, to be of practical use, we require that the pseudorandom functions can be efficiently computed. That is, functions in the ensemble should have succinct representation that supports both selecting them and evaluating them. These aspects are captured by the following definition, in which I is an algorithm selecting representations of functions (which are associated to the functions themselves by the mapping ϕ).

Definition 3.6.3 (efficiently computable function ensembles): *An ℓ-bit 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 function assigned to the string i (i.e., $f_i \stackrel{\text{def}}{=} \phi(i)$).

2. (efficient evaluation): *There exists a polynomial time algorithm, V, so that $V(i,x) = f_i(x)$, for every i in the range of $I(1^n)$ and $x \in \{0,1\}^{\ell(n)}$.*

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

Another point worthy of stressing is that efficiently computable pseudorandom functions are efficiently evaluated at given points, *provided that the function description is given as well.* However, if the function (or its description) is *not* known 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 are also given.

Terminology: In the rest of this book we consider only efficiently computable pseudorandom function ensembles. Hence, in the sequel, whenever we talk of pseudorandom functions we actually mean functions chosen at random from an efficiently computable pseudorandom function ensemble.

Observe that, without loss of generality, the sequence of coin tosses used by the indexing algorithm in Definition 3.6.3 can serve as the function's description. Combining this observation with Definition 3.6.2 , we obtain the following alternative definition of efficiently computable pseudorandom functions:

Definition 3.6.4 (efficiently computable pseudorandom function ensembles – alternative formulation): *A* efficiently computable pseudorandom function ensemble (pseudorandom function) *is a set of finite functions*

$$\{f_s : \{0,1\}^{\ell(|s|)} \to \{0,1\}^{\ell(|s|)}\}_{s\in\{0,1\}^*}$$

where $\ell : \mathbb{N}\to\mathbb{N}$ and the following two conditions hold

1. (efficient evaluation): *There exists a polynomial time algorithm that on input* s *and* $x \in \{0,1\}^{\ell(|s|)}$ *returns* $f_s(x)$.
2. (pseudorandomness): *The function ensemble* $F = \{F_n\}_{n\in\mathbb{N}}$, *defined so that* F_n *is uniformly distributed over the multi-set* $\{f_s\}_{s\in\{0,1\}^n}$, *is pseudorandom.*

We comment that more general notions of pseudorandom functions can be defined and constructed analogously; see Section 3.6.4.

3.6.2 Construction

Using any pseudorandom generator, we construct a (efficiently computable) pseudorandom function ensemble (for $\ell(n) = n$).

Construction 3.6.5 *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* $f_s : \{0,1\}^n \to \{0,1\}^n$ *so that for every* $\sigma_1, ..., \sigma_n \in \{0,1\}$

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

That is, on input s *and* $x = \sigma_1\sigma_2\cdots\sigma_n$, *the value* $f_s(x)$ *is computed as follows:*

Let $y = s$. *For* $i = 1$ *to* n *do* $y \leftarrow G_{\sigma_i}(y)$. *Output* y.

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 (see Figure 3.1), the function f_s is defined by n-step walks down a full binary tree of depth n having labels at the vertices. The root of the tree, hereafter referred to as the level 0 vertex of the tree, is labeled by the string s. If an internal vertex is labeled r then its left child is labeled $G_0(r)$ whereas its right child is labeled $G_1(r)$. The value of $f_s(x)$ is the string residing in the leaf reachable from the root by a path corresponding to the string x. The random variable F_n is assigned labeled trees corresponding to all possible 2^n labelings 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.

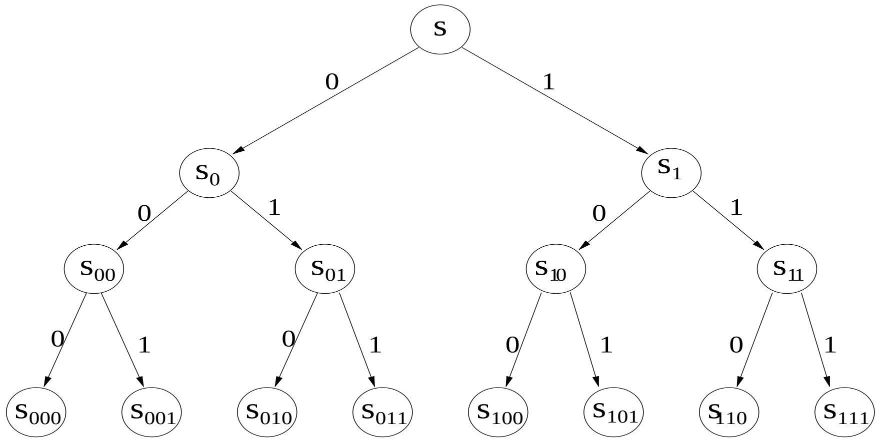
Theorem 3.6.6 *Let* G *and* F *be as in Construction 3.6.5, and suppose that* G *is a pseudorandom generator. Then* F *is an efficiently computable ensemble of pseudorandom functions.*

Combining Theorems **??** and 3.6.6, we immediately get:

Corollary 3.6.7 *If there exist one-way functions then pseudorandom functions exist as well.*

Also, combining Theorem 3.6.6 with the observation that, for $\ell(n) > \log_2 n$, any pseudorandom function (as in Definition 3.6.4) gives rise to a pseudorandom generator (see Exercise **??**), we obtain:

We let $s_\lambda = s$ and $s_{\alpha\sigma} = G_\sigma(s_\alpha)$. The value of $f_s(\sigma_1\sigma_2\cdots\sigma_n) = s_{\sigma_1\sigma_2\cdots\sigma_n}$ is obtained at the leaf reachable from the root (labeled s) by following the path $\sigma_1\sigma_2\cdots\sigma_n$.



For example, $f_s(001) = s_{001} = G_1(s_{00}) = G_1(G_0(s_0)) = G_1(G_0(G_0(s)))$.

Figure 3.1: Construction 3.6.5, for $n = 3$.

Corollary 3.6.8 *Pseudorandom functions (for super-logarithmic ℓ) exist if and only if pseudorandom generators exist.*

Proof of Theorem 3.6.6: 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 a function that results 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 3.6.5. 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 neighboring 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 **??**, we derive a contradiction to the hypothesis (that G is a pseudorandom generator). Details follows.

For every k, $0 \le k \le n$, we define a hybrid distribution H_n^k, assigned as values functions $f : \{0,1\}^n \to \{0,1\}^n$, as follows. For every $s_1, s_2, ..., s_{2^k} \in \{0,1\}^n$, we define a function $f_{s_1,...,s_{2^k}} : \{0,1\}^n \to \{0,1\}^n$ 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 the index of α in the standard lexicographic order of binary strings of length $|\alpha|$. Namely, $f_{s_1,...,s_{2^k}}(x)$ 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 stages (of Construction 3.6.5). The random variable H_n^k is uniformly distributed over the above $(2^n)^{2^k}$ possible functions (corresponding to all possible choices of $s_1, s_2, ..., s_{2^k} \in \{0,1\}^n$). 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 neighboring hybrids. This ability is further transformed 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.

Algorithm D: 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*, 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 $P_0(\alpha)$ (resp., $P_1(\alpha)$) denotes the n-bit prefix of α (resp., 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 (i.e., the input string α_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 that 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 that has been labeled 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* 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 the correspondence between D's actions on checkpoint k and the hybrids k and $k+1$:

- When the inputs are taken from the $t(n)$-product of U_{2n} (and algorithm D chooses k as the checkpoint), the invoked machine M behaves exactly as on the $k+1^{\text{st}}$ hybrid. This is so because D places halves of truly random $2n$-bit long strings at level $k+1$ (which is the same as placing truly random n-bit long strings at level $k+1$).

- On the other hand, 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. Indeed, D does not place the (unknown to it) corresponding seeds (generating these pseudorandom strings) at level k; but putting the two halves of the pseudorandom strings at level $k+1$ has exactly the same effect.

Thus,

Claim 3.6.6.1: 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 an N-bit string that is selected by a pair of interactive processes, that proceed in N iterations. At each iteration the first party chooses a new location (i.e., an unused $i \in \{1,...,N\}$), based on the entire history of the interaction, and the second process sets the value of this bit (i.e., the i^{th} bit) by flipping an unbiased coin. It is intuitively clear that the resulting string is uniformly distributed, still a proof is required (since randomized processes are subtle objects that often lead to mistakes). In our setting the situation is slightly more involved. The process of determining the string is terminated after $T < N$ iterations and statements are made regarding the resulting string that is only partially determined. Consequently, the situation is slightly confusing, and we feel that a detailed argument is required. However, the argument provides no additional insights and may be skipped without significant damage (especially by readers that are more interested in cryptography than in "probabilistic analysis").

Proof of Claim 3.6.6.1: 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 $g:\{0,1\}^m \to \{0,1\}^n$, that is later used to determine a function $f:\{0,1\}^n \to \{0,1\}^n$. The first random process, denoted ρ, is an arbitrary process ("given to us from the outside"), that 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 $g:\{0,1\}^m \to \{0,1\}^n$ is defined by iterating the process ρ with the random process ψ defined below. Process ψ starts with g that 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, process 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 $g:\{0,1\}^m \to \{0,1\}^n$ is totally defined, we define a function $f^g:\{0,1\}^n \to \{0,1\}^n$ by

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

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 $k+1^{\rm st}$ hybrid random variable.

Suppose, on the other hand, that (again 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 $k^{\rm th}$ hybrid random variable. $\Box$

Combining Claim 3.6.6.1 and $\Delta(n) = \mathsf{Pr}[M^{H_n^0}(1^n)=1] - \mathsf{Pr}[M^{H_n^k}(1^n)=1]$, it follows that

$$\begin{aligned}
&\mathsf{Pr}\left[D(G(U_n^{(1)}),...,G(U_n^{(t)}))=1\right] - \mathsf{Pr}\left[D(U_{2n}^{(1)},...,U_{2n}^{(t)})=1\right] \\
&= \left(\frac{1}{n}\sum_{k=0}^{n-1}\mathsf{Pr}\left[M^{H_n^k}(1^n)=1\right]\right) - \left(\frac{1}{n}\sum_{k=0}^{n-1}\mathsf{Pr}\left[M^{H_n^{k+1}}(1^n)=1\right]\right) \\
&= \frac{\Delta(n)}{n}
\end{aligned}$$

which, by the contradiction hypothesis is greater than $\frac{1}{n\cdot p(n)}$, for infinitely many n's. So it follows that D (which is probabilistic polynomial-time) distinguishes polynomially many samples of $G(U_n)$ from polynomially many samples of U_{2n}. Using Theorem **??**, 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 Applications – A general methodology

Sharing a pseudorandom function allows parties to determine random-looking values depending on their current views of the environment (which need not be known a priori). To appreciate the potential of this tool, one should realize that sharing a pseudorandom function is essentially as good as being able to agree, on the fly, on the association of random values to (on-line) given values, where the latter are taken from a huge set of possible values. We stress that this agreement is achieved without communication and synchronization: Whenever some party needs to associate a random value to a given value, $v \in \{0,1\}^n$, it will associate it the same random value $r_v \in \{0,1\}^n$.

As an illustrative example, consider the problem of *identifying friend or foe*, in which members of a club sharing some secret wish to be able to identify one another as belonging to the club. A possible solution is for the club members to share a secret function, defined over a huge domain, and prove their membership in the club by answering a random challenge presented to them, with the value of the secret function evaluated at the challenge. We claim that using a pseudorandom function in the role of the secret function guarantees that it is infeasible for an adversary to pass as a member, even after conducting polynomially-many interactions with members in which the adversary may ask them to reply to challenges of its choice. To prove this claim, consider what happens when the secret function is a truly random one. (We stress that this is merely a mental experiment, since it is infeasible to share such a huge random object.) In such a case, the random function's values at new points (corresponding to a new challenge that the adversary should answer) are uncorrelated to its values at any other point (corresponding to answers the adversary has obtained by challenging legitimate members). Thus, the adversary will fail in such an imaginary situation. It follows that the adversary must also fail in the actual situation (in which the secret function is selected from a pseudorandom ensemble), or else we derive a distinguisher of pseudorandom functions from truly random ones.

In general, the following two-step methodology is useful in many cases:

1. Design your scheme assuming that all legitimate users share a random function, $f:\{0,1\}^n \to \{0,1\}^n$. (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 itself.)[1] 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, the adversaries may at most 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 the legitimate users can share random/secret information not known to the adversary (e.g., as is the case in private-key encryption schemes).[2]

3.6.4 * Generalizations

We present generalizations of the notion of a pseudorandom function, first to the case where the function is not length preserving, and next to the case where the function is defined over the set of all strings. These generalizations offer greater flexibility in using pseudorandom functions in applications.

3.6.4.1 Functions that are not length preserving

Departing from Definition 3.6.4, we present the following generalization of the notion of a pseudorandom function ensemble.

Definition 3.6.9 (pseudorandom function ensembles – generalization): *Let $d, r : \mathbb{N} \to \mathbb{N}$. We say that*

$$\{f_s : \{0,1\}^{d(|s|)} \to \{0,1\}^{r(|s|)}\}_{s \in \{0,1\}^*}$$

is an efficiently computable generalized pseudorandom function ensemble (generalized pseudorandom function) *if the following two conditions hold*

1. (efficient evaluation): *There exists a polynomial time algorithm that on input s and $x \in \{0,1\}^{d(|s|)}$ returns $f_s(x)$.*

[1] This is different from the *Random Oracle Model*, where the adversary has a *direct* access to a random oracle (that is later "implemented" by a function, the description of which is given also to the adversary).

[2] In contrast, the *Random Oracle Methodology* refers to a situation in which the adversary is also given the description of the function, which replaces the random oracle to which it has *direct* access (as discussed in Footnote 1). We warn that, in contrast to the methodology presented in the main text (above), the Random Oracle Methodology is a heuristics. See further discussion in Section **??**.

2. (pseudorandomness): *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 F_n is a random variable uniformly distributed over the multi-set $\{f_s\}_{s\in\{0,1\}^n}$, and H_n is uniformly distributed among all functions mapping $d(n)$-bit long strings to $r(n)$-bit long strings.

Clearly, $r : \mathbb{N}\rightarrow\mathbb{N}$ must be upper bounded by a polynomial. Definition 3.6.4 is obtained as a special case (of Definition 3.6.9) by letting the functions d and r equal the function ℓ. Similarly to Construction 3.6.5, for any $d, r : \mathbb{N}\rightarrow\mathbb{N}$ where $r(n)$ is computable in poly(n)-time from n, we can construct general pseudorandom functions using any pseudorandom generator. Specifically:

Construction 3.6.10 *Let G, G_0 and G_1 be as in Construction 3.6.5. Let $d, r : \mathbb{N}\rightarrow\mathbb{N}$, and let G' be a deterministic algorithm mapping n-bit long inputs into $r(n)$-bit outputs. Then, for every $s \in \{0,1\}^n$, we define a function $f_s\colon\{0,1\}^{d(n)} \rightarrow \{0,1\}^{r(n)}$ so that for every $\sigma_1, ..., \sigma_{d(n)} \in \{0,1\}$*

$$f_s(\sigma_1\sigma_2\cdots\sigma_{d(n)}) \stackrel{\rm def}{=} G'(G_{\sigma_{d(n)}}(\cdots(G_{\sigma_2}(G_{\sigma_1}(s))\cdots))$$

Construction 3.6.5 is regained from Construction 3.6.10 by letting $d(n) = r(n) = n$ and using the identity function in role of G'. By extending a little the proof of Theorem 3.6.6, we obtain:

Theorem 3.6.11 *Let G, G' and the f_s's be as in Construction 3.6.10, and suppose that G is a pseudorandom generator. Further suppose that G' is polynomial-time computable and that the ensemble $\{G'(U_n)\}_{n\in\mathbb{N}}$ is pseudorandom,[3] as defined in Definition* **??**. *Then $\{f_s\}_{s\in\{0,1\}^*}$ is an efficiently computable ensemble of generalized pseudorandom functions.*

Proof: In case G' is the identity transformation (and $r(n) = n$), the proof is almost identical to the proof of Theorem 3.6.6. To deal with the general case, we use a hybrid argument. Specifically, we use a *single* intermediate *hybrid* (i.e., a single hybrid of the function ensemble $\{f_s\}$ and a truly random function): for every n, we consider the (random) function $g : \{0,1\}^{d(n)} \rightarrow \{0,1\}^{r(n)}$ defined by letting $g(x) = G'(h'(x))$, where h' is uniformly selected among all functions mapping $d(n)$-bit long strings to n-bit strings. The theorem follows by showing that this hybrid ensemble is indistinguishable from both the uniform function ensemble and the function ensemble of Construction 3.6.10.

Below, we denote by H_n (resp., H'_n) a random variable uniformly distributed over the set of all functions mapping $d(n)$-bit long strings to $r(n)$-bit (resp., n-bit) long strings. Recall that the hybrid distribution, denoted $G' \circ H'_n$, is obtained by functional composition of the fixed function G' and the random function distribution H'_n. As usual, F_n denotes a random variable uniformly distributed over the multi-set $\{f_s\}_{s\in\{0,1\}^n}$.

Claim 3.6.11.1: For every probabilistic polynomial-time oracle machine M, every polynomial $p(\cdot)$ and all sufficiently large n's

$$|\mathsf{Pr}[M^{G'\circ H'_n}(1^n)=1] - \mathsf{Pr}[M^{H_n}(1^n)=1]| < \frac{1}{p(n)}$$

[3] In case $r(n) > n$ (for all n's), what we require is that G' be a pseudorandom generator. But otherwise, this cannot be required since G' is not expanding. Still the other features of a pseudorandom generator (i.e., efficient computability and pseudorandomness of the output) are always required here.

Proof Sketch: Intuitively, an oracle access to $G' \circ H'_n$ is equivalent to being given multiple independent samples from the distribution $G'(U_n)$, whereas oracle access to H_n is equivalent to being given multiple independent samples from the distribution $U_{r(n)}$. Using the pseudorandomness of $\{G'(U_n)\}_{n\in\mathbb{N}}$, the claim follows.

In the actual proof we transform the oracle machine M into an ordinary machine M' that gets a sequence of samples, and emulates an execution of M while using its input sequence in order to emulate some related oracle for M. Specifically, on input $\alpha_1, ..., \alpha_T$, machine M' invokes M, and answers its i^{th} distinct query with α_i. (Without loss of generality, one may assume that M never issues the same query twice.)

1. Indeed, on input a sequence of samples from distribution $G'(U_n)$, machine M' emulates an execution of $M^{G' \circ H'_n}(1^n)$.

   (The key observation is that the responses of oracle $G' \circ H'_n$ to a sequence $q_1, ..., q_t$ of distinct queries are $G'(s_{q_1}), ..., G'(s_{q_t})$, where the s_{q_i}'s are uniformly and independently distributed in $\{0,1\}^n$.)

2. On the other hand, on input a sequence of samples from distribution $U_{r(n)}$, machine M' emulates an execution of $M^{H_n}(1^n)$.

   (The key observation is that the responses of oracle H_n to a sequence $q_1, ..., q_t$ of distinct queries are uniformly and independently distributed in $\{0,1\}^{r(n)}$.)

Thus, if M violates the statement of the claim then M' violates the pseudorandomness of $\{G'(U_n)\}_{n\in\mathbb{N}}$, in contradiction to the theorem's hypothesis. $\Box$

Claim 3.6.11.2: For every probabilistic polynomial-time oracle machine M, every polynomial $p(\cdot)$ and all sufficiently large n's

$$|\Pr[M^{G' \circ H'_n}(1^n)=1] - \Pr[M^{F_n}(1^n)=1]| < \frac{1}{p(n)}$$

Proof Sketch: Any function f_s (as defined in Construction 3.6.10) can be written as $f_s(x) = G'(f'_s(x))$, where f'_s is defined by

$$f'_s(\sigma_1\sigma_2\cdots\sigma_{d(n)}) \stackrel{\text{def}}{=} G_{\sigma_{d(n)}}(\cdots(G_{\sigma_2}(G_{\sigma_1}(s)))\cdots) \tag{3.1}$$

We have already established above that $\{f'_s\}$ is a generalized pseudorandom function ensemble (i.e., f'_s corresponds to the case where G' is the identity), and so incorporating G' in the distinguisher – the claim follows.

In the actual proof we transform the oracle machine M into an oracle machine M' that emulates M while using its own oracle in order to emulate some related oracle for M. Specifically, when M issues a query q, machine M' forwards q to its own oracle, applies G' to the answer that it has received, and forwards the result to M.

1. Indeed, when given oracle access to h', machine M' emulates an execution of $M^{G' \circ h'}(1^n)$. (The reason being that, in this case, M' responds to query q (made by M) with $G'(h'(q)) = (G' \circ h')(q)$.) Thus, when given oracle access to H'_n, machine M' emulates an execution of $M^{G' \circ H'_n}(1^n)$.

2. On the other hand, when given oracle access to f'_s, machine M' emulates an execution of $M^{f_s}(1^n)$. (The reason being that, in this case, M' responds to query q (made by M) with $G'(f'_s(q)) = f_s(q)$.) Thus, for uniformly selected $s \in \{0,1\}^n$, when given oracle access to f'_s, machine M' emulates an execution of $M^{F_n}(1^n)$.

Thus, if M violates the statement of the claim then M' violates the pseudorandomness of $\{f'_s\}$, which contradicts what we have already established. □

Combining Claims 3.6.11.1 and 3.6.11.2, the theorem follows. ■

Comment: One major component of the proof of Theorem 3.6.11 is proving the following proposition:

> *Let $\{f'_s : \{0,1\}^{d(|s|)} \rightarrow \{0,1\}^{|s|}\}_{s\in\{0,1\}^*}$ be a generalized pseudorandom function ensemble, and G' as in the theorem's hypothesis. Then, the generalized function ensemble $\{f_s : \{0,1\}^{d(|s|)} \rightarrow \{0,1\}^{r(|s|)}\}_{s\in\{0,1\}^*}$, defined by $f_s(x) \stackrel{\rm def}{=} G'(f'_s(x))$, is pseudorandom.*

The proof of Claim 3.6.11.2 actually establishes this proposition, and then applies it to $\{f'_s\}_{s\in\{0,1\}^*}$ as defined in Eq. (3.1).

3.6.4.2 Functions defined on all strings

So far we have only considered function ensembles in which each function is finite (i.e., maps a finite domain to a finite range). Using such functions requires a-priori knowledge of an upper bound on the length of the inputs to which the function is to be applied. (Shorter inputs can always be encoded as inputs of some longer and predetermined length.) However, it is preferable not to require such an a-priori known upper bound (e.g., since such a requirement may rule out some applications). It is thus useful to have a more flexible notion of a pseudorandom function ensemble, allowing to apply individual functions to inputs of varying and a-priori unknown length. Such ensembles are defined and constructed next.

Definition 3.6.12 (pseudorandom function ensembles with unbounded inputs): *Let $r : \mathbb{N}\rightarrow\mathbb{N}$. We say that*

$$\{f_s : \{0,1\}^* \rightarrow \{0,1\}^{r(|s|)}\}_{s\in\{0,1\}^*}$$

is an efficiently computable unbounded-input pseudorandom function ensemble (unbounded-input pseudorandom function) *if the following two conditions hold*

1. (efficient evaluation): *There exists a polynomial time algorithm that on input s and $x \in \{0,1\}^*$ returns $f_s(x)$.*
2. (pseudorandomness): *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 F_n is a random variable uniformly distributed over the multi-set $\{f_s\}_{s\in\{0,1\}^n}$, and H_n is uniformly distributed[4] *among all functions mapping arbitrary long strings to $r(n)$-bit long strings.*

[4] Since the running-time of M is a-priori bounded by some polynomial, it follows that for some polynomial d and all n's it holds that, on input 1^n, machine M makes only queries of length at most $d(n)$. Thus, H_n can be defined as the uniform distribution over all functions mapping strings of length up-to $d(n)$ to $r(n)$-bit long strings. This resolve the technical problem of what is meant by a uniform distribution over an infinite set (i.e., the set of all functions mapping arbitrary long bit strings to $r(n)$-bit long strings).

A few comments regarding Definition 3.6.12 are in place. Firstly note that the fact that the length of the input to f_s is not a-priori known raises no problems in Item 1, since the running-time of the evaluating algorithm may depend (polynomially) on the length of the input to f_s. Regarding Item 2, the fact that M has a-priori bounded (polynomial) running-time, upper bounds the length of the queries made to the oracle. The latter fact resolves a technical problem that arises in the above definition (see Footnote 4). In typical applications, one uses $r(n) = n$ (or $r(n)$ that is polynomially related to n). Another special case of interest is the case where $r \equiv 1$; that is, of pseudorandom Boolean functions.

Similarly to Constructions 3.6.5 and 3.6.10, for any $r : \mathbb{N} \rightarrow \mathbb{N}$ so that $r(n)$ is computable in poly(n)-time from n, we can construct unbounded-input pseudorandom functions using any pseudorandom generator. Specifically:

Construction 3.6.13 *Let G be a deterministic algorithm expanding inputs of length n into strings of length $2n + r(n)$. We denote by $G_0(s)$ the $|s|$-bit long prefix of $G(s)$, by $G_1(s)$ the next $|s|$ bits in $G(s)$, and by $G_2(s)$ the $r(|s|)$-bit long suffix of $G(s)$ (i.e., $G(s) = G_0(s)G_1(s)G_2(s)$). Then, for every $s \in \{0,1\}^n$, we define a function $f_s : \{0,1\}^* \rightarrow \{0,1\}^{r(n)}$ so that for every non-negative integer d and every $\sigma_1, ..., \sigma_d \in \{0,1\}$*

$$f_s(\sigma_1\sigma_2\cdots\sigma_d) \stackrel{\rm def}{=} G_2(G_{\sigma_d}(\cdots(G_{\sigma_2}(G_{\sigma_1}(s))\cdots))$$

Pictorially the function f_s is defined by walks down an infinite ternary tree having labels at the vertices. Internal vertices have $|s|$-bit long labels, and leaves have $r(|s|)$-bit long labels. The root of the tree, hereafter referred to as the level 0 vertex of the tree, is labeled by the string s. If an internal vertex is labeled s' then its leftmost child is labeled $G_0(s')$, its middle child is labeled $G_1(s')$, and its rightmost child is labeled $G_2(s')$. The first two children of each internal vertex are internal vertices, whereas the rightmost child of an internal vertex is a leaf. The value of $f_s(\sigma_1 \cdots \sigma_d)$ is the string residing in the leaf reachable from the root by "following the path $\sigma_1, ..., \sigma_d, 2$", when the root is labeled by s. Again, by extending a little the proof of Theorem 3.6.6, we obtain:

Theorem 3.6.14 *Let G and the f_s's be as in Construction 3.6.13, and suppose that G is a pseudorandom generator. Then $\{f_s\}_{s\in\{0,1\}^*}$ is an efficiently computable ensemble of unbounded-input pseudorandom functions.*

Proof Sketch: We follow the proof method of Theorem 3.6.6. That is, we use the hybrid technique, where the $k^{\rm th}$ hybrid will be assigned a function that results by uniformly selecting labels for the vertices of the highest $k+1$ levels of the tree, and computing the labels of lower levels as in Construction 3.6.13. Specifically, the $k^{\rm th}$ hybrid is defined as equal function $f_{s_1,...,s_{3^k}} : \{0,1\}^* \rightarrow \{0,1\}^{r(n)}$, defined below, where $s_1, ..., s_{2^k} \in \{0,1\}^{2n+r(n)}$ are uniformly and independently distributed.

$$f_{s_1,...,s_{3^k}}(\sigma_1\sigma_2\cdots\sigma_d) \stackrel{\rm def}{=} \begin{cases} P_2(s_{{\rm idx}(2^{k-d}\cdot\sigma_d\cdots\sigma_1)})) & \text{if } d \leq k \\ G_2(G_{\sigma_d}(\cdots(G_{\sigma_{k+2}}(G_{\sigma_{k+1}}(s_{{\rm idx}(\sigma_k\cdots\sigma_1)}))\cdots)) & \text{otherwise} \end{cases}$$

where idx(α) is the index of α in the standard lexicographic order of ternary strings of length $|\alpha|$, and $P_2(\beta)$ is the $r(n)$-bit long suffix of β.

Note that (unlike in the proof of Theorem 3.6.6), for every n, there are infinitely many hybrids, since here k may be any non-negative integer (rather $k \in \{0,1,...,n\}$ as in the proof of Theorem 3.6.6). Still, since we consider an (arbitrary) probabilistic *polynomial-time* distinguisher, denoted M, there exists a polynomial d so that on input 1^n the oracle machine M only makes queries of length at most $d(n)-1$. Thus, giving M oracle access to the $d(n)^{\text{th}}$ hybrid is equivalent to giving M oracle access to the uniform random variable H_n (where H_n is as in Definition 3.6.12), since a uniformly chosen label is assigned to each i-level leaf for $i \leq d(n)$. On the other hand, the 0-hybrid corresponds to the random variable F_n (where F_n is as in Definition 3.6.12), since a uniformly chosen label is assigned to the root. Thus, if M can distinguish $\{F_n\}$ from $\{H_n\}$ then it can distinguish a (random) pair of neighboring hybrids (i.e., the $k-1^{\text{st}}$ and k^{th} hybrids, where k is uniformly selected in $\{1,...,d(n)\}$). As in the proof of Theorem 3.6.6, the latter assertion can be shown to violate the pseudorandomness of G. Specifically, we can distinguish multiple independent samples taken from the distribution $U_{2n+r(n)}$ and multiple independent samples taken from the distribution $G(U_n)$: Given a sequence of $(2n+r(n))$-bit long strings, we use these strings in order to label vertices in the $k+1$ highest levels of the tree (by breaking each string into three parts, and using these parts as labels of the three children of some $i-1^{\text{st}}$ level node, for $i \leq k$). In case the strings are taken from $U_{2n+r(n)}$, we emulate the k^{th} hybrid; whereas, in case the strings are taken from $G(U_n)$, we emulate the $k-1^{\text{st}}$ hybrid. The theorem follows. $\Box$

Comment: Unbounded-input (and generalized) pseudorandom functions can be constructed directly from (ordinary) pseudorandom functions; see Section **??**.