Part I

Basic Tools

Chapter 2

Computational Difficulty

In this chapter we present several variants of the definition of one-way functions. In particular, we define strong and weak one-way functions. We prove that the existence of weak one-way functions imply the existence of strong ones. The proof provides a simple example of a case where a computational statement is much harder to prove than its "information theoretic analogue". Next, we define hard-core predicates, and prove that every one-way function "has" a hard-core predicate.

Organizaton: In Section 2.1 we motivate the definition of one-way functions by arguing imformally that it is implict in various natural cryptographic primitives. The basic definitions are given in Section 2.2 and in Section 2.3 we show that weak one-way functions can be used to construct strong ones. A more efficient construction, for certain cases, is postponed to Section 2.6. In Section 2.5 we define hard-core predicates and show how to construct them from one-way functions.

2.1 One-Way Functions: Motivation

As stated in the introduction chapter, modern cryptography is based on a gap between efficient algorithms guaranteed for the legitimate user versus the computational infeasibility of retrieving protected information for an adversary. To illustrate this, we concentrate on the cryptographic task of secure data communication, namely encryption schemes.

In secure encryption schemes, the legitimate user should be able to easily decipher the messages using some private information available to him, yet an adversary (not having this private information) should not be able to decrypt the ciphertext efficiently (i.e., in probabilistic polynomial-time). On the other hand, a non-deterministic machine can quickly decrypt the ciphertext (e.g., by guessing the private information). Hence, the existence of secure encryption schemes implies that there are tasks (e.g., "breaking" encryption schemes) that can be performed by non-deterministic polynomial-time machines, yet cannot be performed by deterministic (or even randomized) polynomial-time machines. In other words, a necessary condition for the existence of secure encryption schemes is that $\mathcal{NP}$ is not contained in $\mathcal{BPP}$ (and thus $\mathcal{P} \neq \mathcal{NP}$).

Although $\mathcal{P} \neq \mathcal{NP}$ is a necessary condition it is not a sufficient one. $\mathcal{P} \neq \mathcal{NP}$ implies that the encryption scheme is hard to break in the worst case. It does not rule-out the possibility that the encryption scheme is easy to break almost always. Indeed, one can construct "encryption schemes" for which the breaking problem is NP-complete, and yet there exist an efficient breaking algorithm that succeeds 99% of the time. Hence, worst-case hardness is a poor measure of security. Security requires hardness on most cases or at least "average-case hardness". A necessary condition for the existence of secure encryption schemes is thus the existence of languages in $\mathcal{NP}$ which are hard on the average. It is not known whether $\mathcal{P} \neq \mathcal{NP}$ implies the existence of languages in $\mathcal{NP}$ which are hard on the average.

The mere existence of problems (in NP) which are hard on the average does not suffice either. In order to be able to use such hard-on-the-average problems we must be able to generate hard instances together with auxiliary information which enable to solve these instances fast. Otherwise, these hard instances will be hard also for the legitimate users, and consequently the legitimate users gain no computational advantage over the adversary. Hence, the existence of secure encryption schemes implies the existence of an efficient way (i.e. probabilistic polynomial-time algorithm) of generating instances with corresponding auxiliary input so that

1. it is easy to solve these instances given the auxiliary input; and
2. it is hard on the average to solve these instances (when not given the auxiliary input).

The above requirement is captured by the definition of one-way functions presented in the next subsection. For further details see Exercise 1.

2.2 One-Way Functions: Definitions

In this section, we present several definitions of one-way functions. The first version, hereafter referred to as strong one-way function (or just one-way function), is the most popular one. We also present weak one-way functions, non-uniformly one-way functions, and plausible candidates for such functions.

2.2.1 Strong One-Way Functions

Loosely speaking, a one-way function is a function which is easy to compute but hard to invert. The first condition is quite clear: saying that a function f is easy to compute means that there exists a polynomial-time algorithm that on input x outputs $f(x)$. The second condition requires more elaboration. Saying that a function f is hard to invert means that every probabilistic polynomial-time algorithm trying, on input y to find an inverse of y under f, may succeed only with negligible (in $|y|$) probability. A sequence $\{s_n\}_{n \in \mathbb{N}}$ is called negligible in n if for every polynomial $p(\cdot)$ and all sufficiently large n's it holds that $s_n < \frac{1}{p(n)}$. Further discussion proceeds the definition.

Definition 2.2.1 (strong one-way functions): *A function* $f : \{0,1\}^* \mapsto \{0,1\}^*$ *is called* (strongly) one-way *if the following two conditions hold*

1. easy to compute: *There exists a (deterministic) polynomial-time algorithm, A, so that on input x algorithm A outputs $f(x)$ (i.e., $A(x) = f(x)$).*

2. hard to invert: *For every probabilistic polynomial-time algorithm, A', every polynomial $p(\cdot)$, and all sufficiently large n's*

$$\mathsf{Pr}\left(A'(f(U_n), 1^n) \in f^{-1} f(U_n)\right) < \frac{1}{p(n)}$$

Recall that U_n denotes a random variable uniformly distributed over $\{0,1\}^n$. Hence, the probability in the second condition is taken over all the possible values assigned to U_n and all possible internal coin tosses of A', with uniform probability distribution. In addition to an input in the range of f, the inverting algorithm is also given the desired length of the output (in unary notation). The main reason for this convention is to rule out the possibility that a function is consider one-way merely because the inverting algorithm does not have enough time to print the output. Consider for example the function $f_{\rm len}$ defined by $f_{\rm len}(x) = y$ where y is the binary representation of the length of x (i.e., $f_{\rm len}(x) = |x|$). Since $|f_{\rm len}(x)| = \log_2 |x|$ no algorithm can invert $f_{\rm len}(x)$ in time polynomial in $|f_{\rm len}(x)|$, yet there exists an obvious algorithm which inverts $f_{\rm len}(x)$ in time polynomial in $|x|$. In general, the auxiliary input $1^{|x|}$, provided in conjunction to the input $f(x)$, allows the inverting algorithm to run in time polynomial in the total length of the input and the desired output. Note that in the special case of length preserving functions f (i.e., $|f(x)| = |x|$ for all x's), the auxiliary input is redundant.

Hardness to invert is interpreted as an upper bound on the success probability of efficient inverting algorithms. The probability is measured with respect to both the random choices of the inverting algorithm and the distribution of the (main) input to this algorithm (i.e., $f(x)$). The input distribution to the inverting algorithm is obtained by applying f to a uniformly selected $x \in \{0,1\}^n$. If f induces a permutation on $\{0,1\}^n$ then the input to the inverting algorithm is uniformly distributed over $\{0,1\}^n$. However, in the general case where f is not necessarily a one-to-one function, the input distribution to the inverting algorithm may differ substantially from the uniform one. In any case, it is required that the success probability, defined over the above probability space, is negligible (as a function of the length of x), where negligible means being bounded above by all functions of the form $\frac{1}{\mathrm{poly}(n)}$. To further clarify the condition made on the success probability, we consider the following examples.

Consider, an algorithm A_1 that on input $(y, 1^n)$ randomly selects and outputs a string of length n. In case f is a 1-1 function, we have

$$\mathsf{Pr}\left(A_1(f(U_n), 1^n) \in f^{-1}f(U_n)\right) = \frac{1}{2^n}$$

since for every x the probability that $A_1(f(x))$ equals x is exactly 2^{-n}. Hence, the success probability of A_1 on any 1-1 function A_1 is negligible. On the other hand, for every function f, the success probability of A_1 on input $f(U_n)$ is never zero (specifically it is at least 2^{-n}). In case f is constant over strings of the same length (e.g., $f(x) = 0^{|x|}$), we have

$$\mathsf{Pr}\left(A_1(f(U_n), 1^n) \in f^{-1}f(U_n)\right) = 1$$

since every $x \in \{0,1\}^n$ is a preimage of 0^n under f. It follows that a one-way function cannot be constant on strings of the same length. Another trivial algorithm, denoted A_2, is one that computes a function which is constant on all inputs of the same length (e.g., $A_2(y, 1^n) = 1^n$). For every function f we have

$$\mathsf{Pr}\left(A_2(f(U_n), 1^n) \in f^{-1}f(U_n)\right) \geq \frac{1}{2^n}$$

(with equality in case $f(1^n)$ has a single preimage under f). Hence, the success probability of A_2 on any 1-1 function is negligible. On the other hand, if $\mathsf{Pr}(f(U_n) = f(1^n))$ is non-negligible then so is the success probability of algorithm A_2.

A few words, concerning the notion of negligible probability, are in place. The above definition and discussion considers the success probability of an algorithm to be *negligible* if, as a function of the input length, the success probability is bounded above by every polynomial fraction. It follows that repeating the algorithm polynomially (in the input length) many times yields a new algorithm that also has a negligible success probability. In other words, events which occur with negligible (in n) probability remain negligible even if the experiment is repeated for polynomially (in n) many times. Hence, defining negligible success as "occurring with probability smaller than any polynomial fraction" is naturally coupled with defining feasible as "computed within polynomial time".

A "strong negation" of the notion of a negligible fraction/probability is the notion of a non-negligible fraction/probability. We say that a function ν is *non-negligible* if there exists a polynomial $p(\cdot)$ such that for all sufficiently large n's it holds that $\mu(n) > \frac{1}{p(n)}$. Note that functions may be neither negligible nor non-negligible.

2.2.2 Weak One-Way Functions

One-way functions as defined above, are one-way in a very strong sense. Namely, any efficient inverting algorithm has negligible success in inverting them. A much weaker definition, presented below, only requires that all efficient inverting algorithm fails with some non-negligible probability.

Definition 2.2.2 (weak one-way functions): *A function* $f:\{0,1\}^* \mapsto \{0,1\}^*$ *is called* weakly one-way *if the following two conditions hold*

1. easy to compute: *as in the definition of strong one-way function.*
2. slightly-hard to invert: *There exists a polynomial* $p(\cdot)$ *such that for every probabilistic polynomial-time algorithm,* A'*, and all sufficiently large* n*'s*

$$\mathsf{Pr}\left(A'(f(U_n), 1^n) \notin f^{-1} f(U_n)\right) > \frac{1}{p(n)}$$

2.2.3 Two Useful Length Conventions

In the sequel it will be convenient to use the following two conventions regarding the *length* of the of the preimages and images of a one-way function. In the current subsection we justify the used of these conventions.

2.2.3.1 One-way functions defined only for some lengths

In many cases it is more convenient to consider one-way functions with domain partial to the set of all strings. In particular, this facilitates the introduction of some structure in the domain of the function. A particularly important case, used throughout the rest of this section, is that of functions with domain $\cup_{n\in\mathbb{N}}\{0,1\}^{p(n)}$, where $p(\cdot)$ is some polynomial. Let $I \subseteq \mathbb{N}$, and denote by $\mathrm{s}_I(n)$ the successor of n with respect to I; namely, $\mathrm{s}_I(n)$ is the smallest integer that is both greater than n and in the set I (i.e., $\mathrm{s}_I(n) \stackrel{\text{def}}{=} \min\{i \in I : i > n\}$). A set $I \subseteq \mathbb{N}$ is called *polynomial-time enumerable* if there exists an algorithm that on input n, halts within $\mathrm{poly}(n)$ steps and outputs $1^{\mathrm{s}_I(n)}$. (The unary output forces $\mathrm{s}_I(n) = \mathrm{poly}(n)$.) Let I be a polynomial-time enumerable set and f be a function with domain $\cup_{n\in I}\{0,1\}^n$. We call f strongly (resp. weakly) *one-way on lengths in* I if f is polynomial-time computable and is hard to invert over n's in I. Such one-way functions can be easily modified into function with the set of all strings as domain, while preserving one-wayness and some other properties of the original function. In particular, for any function f with domain $\cup_{n\in I}\{0,1\}^n$, we can construct a function $g: \{0,1\}^* \mapsto \{0,1\}^*$ by letting

$$g(x) \stackrel{\text{def}}{=} f(x')$$

where x' is the longest prefix of x with length in I. (In case the function f is length preserving, see definition below, we can preserve this property by modifying the construction so that $g(x) \stackrel{\text{def}}{=} f(x')x''$ where $x = x'x''$, and x' is the longest prefix of x with length in I. The following proposition remains valid also in this case, with a minor modification in the proof.)

Proposition 2.2.3 : *Let* I *be a polynomial-time enumerable set, and* f *be strongly (resp. weakly) one-way on lengths in* I*. Then* g *(constructed above) is strongly (resp. weakly) one-way (in the ordinary sense).*

Although the validity of the above proposition is very appealing, we urge the reader not to skip the following proof. The proof, which is indeed quite simple, uses for the first time in this book an argument that is used extensively in the sequel. The argument used to prove the "hardness to invert" property of the function g proceeds by assuming, to the contradiction, that g can be efficiently inverted with unallowable success probability. Contradiction is derived by deducing that f can be efficiently inverted with unallowable success probability. In other words, inverting f is "reduced" to inverting g. The term "reduction" is used here in a non-standard sense, which preserves the success probability of the algorithms. This kind of an argument is called a *reducibility argument*.

Proof: We first prove that g can be computed in polynomial-time. To this end we use the fact that I is a polynomial-time enumerable set. It follows that on input x one can find in polynomial-time the largest $m \leq |x|$ that satisfies $m \in I$. Computing $g(x)$ amounts to finding this m, and applying the function f to the m-bit prefix of x.

We next prove that g maintains the "hardness to invert" property of f. For sake of concreteness we present here only the proof for the case that f is strongly one-way. The proof for the case that f is weakly one-way is analogous.

The prove proceeds by contradiction. We assume, on contrary to the claim (of the proposition), that there exists an efficient algorithm that inverts g with success probability that is not negligible. We use this inverting algorithm (for g) to construct an efficient algorithm that inverts f with success probability that is not negligible, hence deriving a contradiction (to the hypothesis of the proposition). In other words, we show that inverting f (with unallowable success probability) is efficiently reducible to inverting g (with unallowable success probability), and hence conclude that the latter is not feasible. The reduction is based on the observation that inverting g on images of arbitrary length yields inverting g also on images of length in I, and that on such lengths g collides with f. Details follow.

Given an algorithm, B', for inverting g we construct an algorithm, A', for inverting f so that A' has complexity and success probability related to that of B'. Algorithm A' uses algorithm B' as a subroutine and proceeds as follows. On input y and 1^n (supposedly y is in the range of $f(U_n)$ and $n \in I$) algorithm A' first computes $\mathrm{s}_I(n)$ and sets $k \stackrel{\mathrm{def}}{=} \mathrm{s}_I(n) - n - 1$. For every $0 \leq i \leq k$, algorithm A' initiates algorithm B', on input $(y, 1^{n+i})$, obtaining $z_i \leftarrow B'(y, 1^{n+i})$, and checks if $g(z_i) = y$. In case one of the z_i's satisfies the above condition, algorithm A' outputs the n-bit long prefix of z_i. This prefix is in the preimage of y under f (since $g(x'x'') = f(x')$ for all $x' \in \{0,1\}^n$ and $|x''| \leq k$). Clearly, if B' is a probabilistic polynomial-time algorithm then so is A'. We now analyze the success probability of A' (showing that if B' inverts g with unallowable success probability then A' inverts f with unallowable success probability).

Suppose now, on the contrary to our claim, that g is not strongly one-way, and let B' be an algorithm demonstrating this contradiction hypothesis. Namely, there exists a polynomial $p(\cdot)$ so that for infinitely many m's the probability that B' inverts g on $g(U_m)$ is at least $\frac{1}{p(m)}$. Let us denote the set of these m's by M. Define a function $\ell_I : \mathbb{N} \mapsto I$ so that $\ell_I(m)$ is the largest lower bound of m in I (i.e., $\ell_I(m) \stackrel{\text{def}}{=} \max\{i \in I : i \leq m\}$). Clearly, $m \leq s_I(\ell_I(m)) - 1$ for every m. The following two claims relate the success probability of algorithm A' with that of algorithm B'.

Claim 2.2.3.1: Let m be an integer and $n = \ell_I(m)$. Then

$$\Pr\left(A'(f(U_n), 1^n) \in f^{-1}f(U_n)\right) \geq \Pr\left(B'(g(U_m), 1^m) \in g^{-1}g(U_m)\right)$$

(Namely, the success probability of algorithm A' on $f(U_{\ell_I(m)})$ is bounded below by the success probability of algorithm B' on $g(U_m)$.)

Proof: By construction of A', on input $(f(x'), 1^n)$, where $x' \in \{0,1\}^n$, algorithm A' obtains the value $B'(f(x'), 1^t)$, for every $t \leq s_I(n) - 1$. In particular, since $m \leq s_I(\ell_I(m)) - 1 = s_I(n) - 1$, it follows that algorithm A' obtains the value $B'(f(x'), 1^m)$. By definition of g, for all $x'' \in \{0,1\}^{m-n}$, it holds that $f(x') = g(x'x'')$. The claim follows. □

Claim 2.2.3.2: There exists a polynomial $q(\cdot)$ such that $m < q(\ell_I(m))$, for all m's. Hence, the set $S \stackrel{\text{def}}{=} \{\ell_I(m) : m \in M\}$ is infinite.

Proof: Using the polynomial-time enumerability of I, we get $s_I(n) < \text{poly}(n)$, for every n. Therefore, for every m, we have $m < s_I(\ell_I(m)) < \text{poly}(\ell_I(m))$. Furthermore, S must be infinite, otherwise for n upper-bounding S we get $m < q(n)$ for every $m \in M$.□

Using Claims 2.2.3.1 and 2.2.3.2, it follows that, for every $n = \ell_I(m) \in S$, the probability that A' inverts f on $f(U_n)$ is at least $\frac{1}{p(m)} > \frac{1}{p(q(n))} = \frac{1}{\text{poly}(n)}$. It follows that f is not strongly one-way, in contradiction to the proposition's hypothesis. ■

2.2.3.2 Length-regular and length-preserving one-way functions

A second useful convention is to assume that the function, f, we consider is *length regular* in the sense that, for every $x, y \in \{0,1\}^*$, if $|x| = |y|$ then $|f(x)| = |f(y)|$. We point out that the transformation presented above preserves length regularity. A special case of length regularity, preserved by a the modified transformation presented above, is of *length preserving* functions.

Definition 2.2.4 (length preserving functions): *A function f is* length preserving *if for every $x \in \{0,1\}^*$ it holds that $|f(x)| = |x|$.*

Given a strongly (resp. weakly) one-way function f, we can construct a strongly (resp. weakly) one-way function h which is length preserving, as follows. Let p be a polynomial bounding the length expansion of f (i.e., $|f(x)| \leq p(|x|)$). Such a polynomial must exist since f is polynomial-time computable. We first construct a length regular function g by defining

$$g(x) \stackrel{\text{def}}{=} f(x)10^{p(|x|)-|f(x)|}$$

(We use a padding of the form 10^* in order to facilitate the parsing of $g(x)$ into $f(x)$ and the "leftover" padding.) Next, we define h only on strings of length $p(n)+1$, for $n \in \mathbb{N}$, by letting

$$h(x'x'') \stackrel{\text{def}}{=} g(x')\ ,\ \text{where } |x'x''| = p(|x'|) + 1$$

Clearly, h is length preserving.

Proposition 2.2.5 : *If f is a strongly (resp. weakly) one-way function then so are g and h (constructed above).*

Proof Sketch: It is quite easy to see that both g and h are polynomial-time computable. Using "reducibility arguments" analogous to the one used in the previous proof, we can establish the hardness-to-invert of both g and h. For example, given an algorithm B' for inverting g, we construct an algorithm A' for inverting f as follows. On input y and 1^n (supposedly y is in the range of $f(U_n)$), algorithm A' halts with output $B'(y10^{p(n)-|y|}, 1^{p(n)+1})$. ■

The reader can easily verify that if f is length preserving then it is redundant to provide the inverting algorithm with the auxiliary input $1^{|x|}$ (in addition to $f(x)$). The same holds if f is length regular and does not shrink its input by more than a polynomial factor (i.e., there exists a polynomial $p(\cdot)$ such that $p(|f(x)|) \geq |x|$ for all x). In the sequel, we will only deal with one-way functions that are length regular and does not shrink their its input by more that a polynomial factor. Furthermore, we will mostly deal with length preserving functions. Hence, in these cases, *we assume, without loss of generality, that the inverting algorithm is only given $f(x)$ as input.*

Functions which are length preserving are not necessarily 1-1. Furthermore, the assumption that 1-1 one-way functions exist seems stronger than the assumption that arbitrary (and hence length preserving) one-way functions exist. For further discussion see Section 2.4.

2.2.4 Candidates for One-Way Functions

Following are several candidates for one-way functions. Clearly, it is not known whether these functions are indeed one-way. This is only a conjecture supported by extensive research which has so far failed to produce an efficient inverting algorithm (having non-negligible success probability).

2.2.4.1 Integer factorization

In spite of the extensive research directed towards the construction of efficient (integer) factoring algorithms, the best algorithms known for factoring an integer N, run in time $L(P) \stackrel{\text{def}}{=} 2^{O(\sqrt{\log P \log\log P})}$, where P is the second biggest prime factor of N. Hence it is reasonable to believe that the function f_{mult}, which partitions its input string into two parts and returns the (binary representation of the) integer resulting by multiplying (the integers represented by) these parts, is one-way. Namely, let

$$f_{\text{mult}}(x, y) = x \cdot y$$

where $|x| = |y|$ and $x \cdot y$ denotes (the string representing) the integer resulting by multiplying the integers (represented by the strings) x and y. Clearly, $f_{\rm mult}$ can be computed in polynomial-time. Assuming the intractability of factoring and using the "density of primes" theorem (which guarantees that at least $\frac{N}{\log_2 N}$ of the integers smaller than N are primes) it follows that $f_{\rm mult}$ is at least weakly one-way. Using a more sophisticated argument, one can show that $f_{\rm mult}$ is strongly one-way. Other popular functions (e.g. the RSA) related to integer factorization are discussed in Subsection 2.4.3.

2.2.4.2 Decoding of random linear codes

One of the most outstanding open problems in the area of error correcting codes is that of presenting efficient decoding algorithms for random linear codes. Of particular interest are random linear codes with constant information rate which can correct a constant fraction of errors. An (n,k,d)-*linear-code* is a k-by-n binary matrix in which the vector sum (mod 2) of any non-empty subset of rows results in a vector with at least d one-entries. (A k-bit long message is encoded by multiplying it with the k-by-n matrix, and the resulting n-bit long vector has a unique preimage even when flipping up to $\frac{d}{2}$ of its entries.) The Gilbert-Varshanov Bound for linear codes guarantees the existence of such a code, provided that $\frac{k}{n} < 1 - H_2(\frac{d}{n})$, where $H_2(p) \stackrel{\rm def}{=} -p\log_2 p - (1-p)\log_2(1-p)$ if $p < \frac{1}{2}$ and $H_2(p) \stackrel{\rm def}{=} 1$ otherwise (i.e., $H_2(\cdot)$ is a modification of the binary entropy function). Similarly, if for some $\epsilon > 0$ it holds that $\frac{k}{n} < 1 - H_2(\frac{(1+\epsilon)d}{n})$ then almost all k-by-n binary matrices constitute (n,k,d)-linear-codes. Consider three constants $\kappa, \delta, \epsilon > 0$ satisfying $\kappa < 1 - H_2((1+\epsilon)\delta)$. The function $f_{\rm code}$, hereafter defined, seems a plausible candidate for a one-way function.

$$f_{\rm code}(C, x, i) \stackrel{\rm def}{=} (C, xC + e(i))$$

where C is an κn-by-n binary matrix, x is a κn-dimensional binary vector, i is the index of an n-dimensional binary vector having at most $\frac{\delta n}{2}$ one-entries (the string itself is denoted $e(i)$), and the arithmetic is in the n-dimensional binary vector space. Clearly, $f_{\rm code}$ is polynomial-time computable. An efficient algorithm for inverting $f_{\rm code}$ would yield an efficient algorithm for inverting a non-negligible fraction of the linear codes (an earthshaking result in coding theory).

2.2.4.3 The subset sum problem

Consider the function $f_{\rm ss}$ defines as follows.

$$f_{\rm ss}(x_1, ..., x_n, I) = (x_1, ..., x_n, \sum_{i \in I} x_i)$$

where $|x_1| = \cdots = |x_n| = n$, and $I \subseteq \{1, 2, ..., n\}$. Clearly, $f_{\rm ss}$ is polynomial-time computable. The fact that the subset-sum problem is NP-complete cannot serve as evidence to the one-wayness of $f_{\rm ss}$. On the other hand, the fact that the subset-sum problem is easy for special cases (such as having "hidden structure" and/or "low density") can not serve as evidence for the weakness of this proposal. The conjecture that $f_{\rm ss}$ is one-way is based on the failure of known algorithm to handle random "high density" instances (i.e., instances in which the length of the elements is not greater than their number). Yet, one has to admit that the evidence in favour of this candidate is much weaker than the evidence in favour of the two previous ones.

2.2.5 Non-Uniformly One-Way Functions

In the above two definitions of one-way functions the inverting algorithm is probabilistic polynomial-time. Stronger versions of both definitions require that the functions cannot be inverted even by non-uniform families of polynomial-size circuits. We stress that the "easy to compute" condition is still stated in terms of uniform algorithms. For example, following is a non-uniform version of the definition of strong (length-preserving) one-way functions.

Definition 2.2.6 (non-uniformly strong one-way functions): *A function* $f : \{0,1\}^* \mapsto \{0,1\}^*$ *is called* non-uniformly one-way *if the following two conditions hold*

1. easy to compute: *There exists a (deterministic) polynomial-time algorithm, A, so that on input x algorithm A outputs $f(x)$ (i.e., $A(x) = f(x)$).*

2. hard to invert: *For every (even non-uniform) family of polynomial-size circuits, $\{C_n\}_{n\in\mathbb{N}}$, every polynomial $p(\cdot)$, and all sufficiently large n's*

$$\mathsf{Pr}\left(C_n(f(U_n))\in f^{-1}f(U_n)\right) < \frac{1}{p(n)}$$

The probability in the second condition is taken only over all the possible values of U_n.

It can be shown that if f is non-uniformly one-way then it is one-way (i.e., in the uniform sense). The proof follows by converting any (uniform) probabilistic polynomial-time inverting algorithm into a non-uniform family of polynomial-size circuits, without decreasing the success probability. Details follow. Let A' be a probabilistic polynomial-time (inverting) algorithm. Let r_n denote a sequence of coin tosses for A' maximizing the success probability of A'. Namely, r_n satisfies $\mathsf{Pr}(A'_{r_n}(f(U_n)\in f^{-1}f(U_n)) \geq \mathsf{Pr}(A(f(U_n)\in f^{-1}f(U_n))$, where the first probability is taken only over all possible values of U_n and the second probability is also over all possible coin tosses for A'. (Recall that $A'_r(y)$ denotes the output of algorithm A' on input y and internal coin tosses r.) The desired circuit C_n incorporates the code of algorithm A' and the sequence r_n (which is of length polynomial in n).

It is possible that one-way functions exist (in the uniform sense) and yet there are no non-uniformly one-way functions. However, such a possibility is considered not very plausible.

2.3 Weak One-Way Functions Imply Strong Ones

We first remark that not every weak one-way function is necessarily a strong one. Consider for example a one-way function f (which without loss of generality is length preserving). Modify f into a function g so that $g(x,p) = (f(x),p)$ if p starts with $\log_2|x|$ zeros and $g(x,p) = (x,p)$ otherwise, where (in both cases) $|x| = |p|$. We claim that g is a weak one-way function but not a strong one. Clearly, g can not be a strong one-way function (since for all but a $\frac{1}{n}$ fraction of the strings of length $2n$ the function g coincides with the identity function). To prove that g is weakly one-way we use a "reducibility argument". Details follow.

Proposition 2.3.1 *Let f be a one-way function (even in the weak sense). Then g, constructed above, is a weakly one-way function.*

Proof: Intuitively, inverting g on inputs on which it does not effect the identity transformation is related to inverting f. If g is inverted, on inputs of length $2n$, with probabiliy which is noticeablly greater than $1-\frac{1}{n}$ then it must be inverted on inputs as above with noticeable probability. Thus, if g is not weakly one-way then so is f. The full, straightforward and tedious proof follows.

Given a probabilistic polynomial-time algorithm, B', for inverting g, we construct a probabilistic polynomial-time algorithm A' which inverts f with "related" success probability. Following is the description of algorithm A'. On input y, algorithm A' sets $n \stackrel{\mathrm{def}}{=} |y|$ and $l \stackrel{\mathrm{def}}{=} \log_2 n$, selects p' uniformly in $\{0,1\}^{n-l}$, computes $z \stackrel{\mathrm{def}}{=} B'(y, 0^l p')$, and halts with output the n-bit prefix of z. Let S_{2n} denote the sets of all $2n$-bit long strings which start with $\log_2 n$ zeros (i.e., $s_{2n} \stackrel{\mathrm{def}}{=} \{0^{\log_2 n}\alpha : \alpha \in \{0,1\}^{2n-\log_2 n}\}$). Then, by construction of A' and g, we have

$$\begin{aligned}
\mathsf{Pr}\left(A'(f(U_n)) \in f^{-1}f(U_n)\right) &\geq \mathsf{Pr}\left(B'(f(U_n), 0^l U_{n-l}) \in (f^{-1}f(U_n), 0^l U_{n-l})\right) \\
&= \mathsf{Pr}\left(B'(g(U_{2n})) \in g^{-1}g(U_{2n}) \mid U_{2n} \in S_{2n}\right) \\
&\geq \frac{\mathsf{Pr}\left(B'(g(U_{2n})) \in g^{-1}g(U_{2n})\right) - \mathsf{Pr}\left(U_{2n} \notin S_{2n}\right)}{\mathsf{Pr}\left(U_{2n} \in S_{2n}\right)} \\
&= \frac{1}{n} \cdot \left(\mathsf{Pr}\left(B'(g(U_{2n})) \in g^{-1}g(U_{2n})\right) - \left(1 - \frac{1}{n}\right)\right) \\
&= 1 - n \cdot \left(1 - \mathsf{Pr}\left(B'(g(U_{2n})) \in g^{-1}g(U_{2n})\right)\right)
\end{aligned}$$

(For the second inequality, we used $\mathsf{Pr}(A|B) = \frac{\mathsf{Pr}(A \cap B)}{\mathsf{Pr}(B)}$ and $\mathsf{Pr}(A \cap B) \geq \mathsf{Pr}(A) - \mathsf{Pr}(\overline{B})$.) It should not come as a surprise that the above expression is meaningful only in case $\mathsf{Pr}(B'(g(U_{2n})) \in g^{-1}g(U_{2n})) > 1 - \frac{1}{n}$.

It follows that, for every polynomial $p(\cdot)$ and every integer n, if B' inverts g on $g(U_{2n})$ with probability greater than $1 - \frac{1}{p(2n)}$ then A' inverts f on $f(U_n)$ with probability greater than $1 - \frac{n}{p(2n)}$. Hence, if g is not weakly one-way (i.e., for every polynomial $p(\cdot)$ there exist infinitely many m's such that g can be inverted on $g(U_m)$ with probability $\geq 1 - 1/p(m)$) then also f is not weakly one-way (i.e., for every polynomial $q(\cdot)$ there exist infinitely many n's such that f can be inverted on $f(U_n)$ with probability $\geq 1 - 1/q(n)$). This contradicts our hypothesis (that f is one-way). ■

We have just shown that, unless no one-way functions exist, there exist weak one-way functions which are not strong ones. Fortunately, we can rule out the possibility that all one-way functions are only weak ones. In particular, the existence of weak one-way functions implies the existence of strong ones.

Theorem 2.3.2 : *Weak one-way functions exist if and only if strong one-way functions exist.*

We strongly recommend to the reader not to skip the following proof, since we believe that the proof is very instructive to the rest of the book. In particular, the proof demonstrates that amplification of computational difficulty is much more involved than amplification of an analogous probabilistic event.

Proof: Let f be a weak one-way function, and let p be the polynomial guaranteed by the definition of a weak one-way function. Namely, every probabilistic polynomial-time algorithm fails to invert f on $f(U_n)$ with probability at least $\frac{1}{p(n)}$. We assume, for simplicity, that f is length preserving (i.e. $|f(x)| = |x|$ for all x's). This assumption, which is not really essential, is justified by Proposition 2.2.5. We define a function g as follows

$$g(x_1, ..., x_{t(n)}) \stackrel{\text{def}}{=} f(x_1), ..., f(x_{t(n)})$$

where $|x_1| = |x_{t(n)}| = n$ and $t(n) \stackrel{\text{def}}{=} n \cdot p(n)$. Namely, the $n^2p(n)$-bit long input of g is partitioned into $t(n)$ blocks each of length n, and f is applied to each block.

Clearly, g can be computed in polynomial-time (by an algorithm which breaks the input into blocks and applies f to each block). Furthermore, it is easy to see that inverting g on $g(x_1, ..., x_{t(n)})$ requires finding a preimage to each $f(x_i)$. One may be tempted to deduce that it is also clear that g is a strongly one-way function. An naive argument, assumes implicitly (with no justification) that the inverting algorithm works separately on each $f(x_i)$. If this were indeed the case then the probability that an inverting algorithm successfully inverts all $f(x_i)$'s is at most $(1 - \frac{1}{p(n)})^{n \cdot p(n)} < 2^{-n}$ (which is negligible also as a function of $n^2p(n)$). However, the assumption that an algorithm trying to invert g works independently on each $f(x_i)$ cannot be justified. Hence, a more complex argument is required.

Following is an outline of our proof. The proof that g is strongly one-way proceeds by a contradiction argument. We assume on the contrary that g is not strongly one-way; namely, we assume that there exists a polynomial-time algorithm that inverts g with probability which is not negligible. We derive a contradiction by presenting a polynomial-time algorithm which, for infinitely many n's, inverts f on $f(U_n)$ with probability greater than $1 - \frac{1}{p(n)}$ (in contradiction to our hypothesis). The inverting algorithm for f uses the inverting algorithm for g as a subroutine (without assuming anything about the manner in which the latter algorithm operates). Details follow.

Suppose that g is not strongly one-way. By definition, it follows that there exists a probabilistic polynomial-time algorithm B' and a polynomial $q(\cdot)$ so that for infinitely many m's

$$\mathsf{Pr}\left(B'(g(U_m)) \in g^{-1}g(U_m)\right) > \frac{1}{q(m)}$$

Let us denote by M', the infinite set of integers for which the above holds. Let N' denote the infinite set of n's for which $n^2 \cdot p(n) \in M'$ (note that all m's considered are of the form $n^2 \cdot p(n)$, for some integer n).

We now present a probabilistic polynomial-time algorithm, A', for inverting f. On input y (supposedly in the range f) algorithm A' proceeds by applying the following probabilistic procedure, denoted I, on input y for $a(|y|)$ times, where $a(\cdot)$ is a polynomial depends on the polynomials p and q (specifically, we set $a(n) \stackrel{\text{def}}{=} 2n^2 \cdot p(n) \cdot q(n^2 p(n))$).

Procedure I

Input: y (denote $n \stackrel{\text{def}}{=} |y|$).

`For` $i = 1$ `to` $t(n)$ `do begin`

1. Select uniformly and independently a sequence of strings $x_1, ..., x_{t(n)} \in \{0,1\}^n$.
2. Compute

$$(z_1, ..., z_{t(n)}) \leftarrow B'(f(x_1), ..., f(x_{i-1}), y, f(x_{i+1}), ..., f(x_{t(n)}))$$

   (Note that y is placed in the i^{th} position instead of $f(x_i)$.)

3. If $f(z_i) = y$ then halt and output y.
   (This is considered a *success*).

`end`

We now present a lower bound on the success probability of algorithm A'. To this end we define a set S_n, which contains all n-bit strings on which the procedure I succeeds with non-negligible probability (specifically greater than $\frac{n}{a(n)}$). (The probability is taken only over the coin tosses of algorithm A'). Namely,

$$S_n \stackrel{\text{def}}{=} \left\{x : \mathsf{Pr}\left(I(f(x)) \in f^{-1}f(x)\right) > \frac{n}{a(n)}\right\}$$

In the next two claims we shall show that S_n contains all but a $\frac{1}{2p(n)}$ fraction of the strings of length $n \in N'$, and that for each string $x \in S_n$ the algorithm A' inverts f on $f(x)$ with probability exponentially close to 1. It will follow that A' inverts f on $f(U_n)$, for $n \in N'$, with probability greater than $1 - \frac{1}{p(n)}$, in contradiction to our hypothesis.

Claim 2.3.2.1: For every $x \in S_n$

$$\mathsf{Pr}\left(A'(f(x) \in f^{-1}f(x)\right) > 1 - \frac{1}{2^n}$$

Proof: By definition of the set S_n, the procedure I inverts $f(x)$ with probability at least $\frac{n}{a(n)}$. Algorithm A' merely repeats I for $a(n)$ times, and hence

$$\mathsf{Pr}\left(A'(f(x) \notin f^{-1}f(x)\right) < \left(1 - \frac{n}{a(n)}\right)^{a(n)} < \frac{1}{2^n}$$

The claim follows. □

Claim 2.3.2.2: For every $n \in N'$,

$$|S_n| > \left(1 - \frac{1}{2p(n)}\right) \cdot 2^n$$

Proof: We assume, to the contrary, that $|S_n| \leq (1-\frac{1}{2p(n)})\cdot 2^n$. We shall reach a contradiction to our hypothesis concerning the success probability of B'. Recall that by this hypothesis

$$s(n) \stackrel{\text{def}}{=} \mathsf{Pr}\left(B'(g(U_{n^2p(n)})) \in g^{-1}g(U_{n^2p(n)})\right) > \frac{1}{q(n^2p(n))}$$

Let $U_n^{(1)}, ..., U_n^{(n \cdot p(n))}$ denote the n-bit long blocks in the random variable $U_{n^2p(n)}$ (i.e., these $U_n^{(i)}$'s are independent random variables each uniformly distributed in $\{0,1\}^n$). Clearly, $s(n)$ is the sum of $s_1(n)$ and $s_2(n)$ defined by

$$s_1(n) \stackrel{\text{def}}{=} \mathsf{Pr}\left(B'(g(U_{n^2p(n)})) \in g^{-1}g(U_{n^2p(n)}) \wedge \left(\exists i \text{ s.t. } U_n^{(i)} \notin S_n\right)\right)$$

and

$$s_2(n) \stackrel{\text{def}}{=} \mathsf{Pr}\left(B'(g(U_{n^2p(n)})) \in g^{-1}g(U_{n^2p(n)}) \wedge \left(\forall i : U_n^{(i)} \in S_n\right)\right)$$

(Use $\mathsf{Pr}(A) = \mathsf{Pr}(A \wedge B) + \mathsf{Pr}(A \wedge \neg B)$.) We derive a contradiction to the lower bound on $s(n)$ by presenting upper bounds for both $s_1(n)$ and $s_2(n)$ (which sum up to less).

First, we present an upper bound on $s_1(n)$. By the construction of algorithm I it follows that, for every $x \in \{0,1\}^n$ and every $1 \leq i \leq n \cdot p(n)$, the probability that I inverts f on $f(x)$ in the i^{th} iteration equals the probability that B' inverts g on $g(U_{n^2p(n)})$ when $U_n^{(i)} = x$. It follows that, for every $x \in \{0,1\}^n$ and every $1 \leq i \leq n \cdot p(n)$,

$$\mathsf{Pr}\left(I(f(x)) \in f^{-1}f(x)\right) \geq \mathsf{Pr}\left(B'(g(U_{n^2p(n)})) \in g^{-1}g(U_{n^2p(n)}) \mid U_n^{(i)} = x\right)$$

Using trivial probabilistic inequalities (such as $\mathsf{Pr}(\exists i\ A_i) \leq \sum_i \mathsf{Pr}(A_i)$ and $\mathsf{Pr}(A \wedge B) \leq \mathsf{Pr}(A \mid B)$), it follows that

$$\begin{aligned}
s_1(n) &\leq \sum_{i=1}^{n\cdot p(n)} \mathsf{Pr}\left(B'(g(U_{n^2p(n)})) \in g^{-1}g(U_{n^2p(n)}) \wedge U_n^{(i)} \notin S_n\right) \\
&\leq \sum_{i=1}^{n\cdot p(n)} \mathsf{Pr}\left(B'(g(U_{n^2p(n)})) \in g^{-1}g(U_{n^2p(n)}) \mid U_n^{(i)} \notin S_n\right) \\
&\leq \sum_{i=1}^{n\cdot p(n)} \mathsf{Pr}\left(I(f(U_n)) \in f^{-1}f(U_n) \mid U_n \notin S_n\right) \\
&\leq n \cdot p(n) \cdot \frac{n}{a(n)}
\end{aligned}$$

(The last inequality uses the definition of S_n.)

We now present an upper bound on $s_2(n)$. Recall that by the contradiction hypothesis, $|S_n| \leq (1 - \frac{1}{2p(n)}) \cdot 2^n$. It follows that

$$\begin{aligned}
s_2(n) &\leq \mathsf{Pr}\left(\forall i : U_n^{(i)} \in S_n\right) \\
&\leq \left(1 - \frac{1}{2p(n)}\right)^{n\cdot p(n)} \\
&< \frac{1}{2^{\frac{n}{2}}}
\end{aligned}$$

Hence, on one hand $s_1(n) + s_2(n) < \frac{2n^2 \cdot p(n)}{a(n)} = \frac{1}{q(n^2p(n))}$ (equality by definition of $a(n)$). Yet, on the other hand $s_1(n) + s_2(n) = s(n) > \frac{1}{q(n^2p(n))}$. Contradiction is reached and the claim follows. □

Combining Claims 2.3.2.1 and 2.3.2.2, It follows that the probabilistic polynomial-time algorithm, A', inverts f on $f(U_n)$, for $n \in N'$, with probability greater than $1 - \frac{1}{p(n)}$, in contradiction to our hypothesis (that f cannot be efficiently inverted with such success probability). The theorem follows. ■

Let us summarize the structure of the proof of Theorem 2.3.2. Given a weak one-way function f, we first constructed a polynomial-time computable function g. This was done with the intention of later proving that g is strongly one-way. To prove that g is strongly one-way we used a "reducibility argument". The argument transforms efficient algorithms which supposedly contradict the strong one-wayness of g into efficient algorithms which contradict the hypothesis that f is weakly one-way. Hence g must be strongly one-way. We stress that our algorithmic transformation, which is in fact a randomized Cook reduction, makes no implicit or explicit assumptions about the structure of the prospective algorithms for inverting g. Such assumptions, as the "natural" assumption that the inverter of g

works independently on each block, cannot be justified (at least not at the current state of understanding of the nature of efficient computations).

Theorem 2.3.2 has a natural information theoretic (or "probabilistic") analogue which asserts that repeating an experiment, which has a non-negligible success probability, sufficiently many times yields success with very high probability. The reader is probably convinced at this stage that the proof of Theorem 2.3.2 is much more complex that the proof of the information theoretic analogue. In the information theoretic context the repeated events are independent by definition, whereas in our computational context no such independence can be guaranteed. Another indication to the difference between the two settings follows. In the information theoretic setting the probability that none of the events occur decreases exponentially in the number of repetitions. However, in the computational setting we can only reach a negligible bounds on the inverting probabilities of polynomial-time algorithms. Furthermore, it may be the case that g constructed in the proof of Theorem 2.3.2 can be efficiently inverted on $g(U_{n^2p(n)})$ with success probability which is subexponentially decreasing (e.g., with probability $2^{-\log_2^3 m}$), whereas the analogous information theoretic experiment fails with probability at most 2^{-n}.

By Theorem 2.3.2, whenever assuming the existence of one-way functions, there is no need to specify whether we refer to weak or strong ones. Thus, as far as the mere existence of one-way function goes, the notions of weak and strong one-way functions are equivalent. However, as far as efficiency considerations are concerned the two notions are not really equivalent, since the above transformation of weak one-way functions into strong ones is not practical. An alternative transformation which is much more efficient does exist for the case of one-way permutations and other specific classes of one-way functions. Further details are presented in Section 2.6.

2.4 One-Way Functions: Variations

In this section, we discuss several issues concerning one-way functions. In the first subsection, we present a function that is (strongly) one-way, provided that one-way functions exist. The construction of this function is of strict abstract interest. In contrast, the issues discussed in the other subsections are of practical importance. First, we present a formulation which is better suited for describing many natural candidates for one-way functions, and use it in order to describe popular candidates for one-way functions. Next, we use this formulation to present one-way functions with additional properties; specifically, (one-way) trapdoor permutations, and clawfree functions. We remark that these additional properties are used in several constructions (e.g., trapdoor permutations are used in the construction of public-key encryption schemes whereas clawfree permutations are used in the construction of collision-free hashing). We conclude this section with remarks addressing the "art" of proposing candidates for one-way functions.

2.4.1 * Universal One-Way Function

Using the result of the previous section and the notion of a universal machine it is possible to prove the existence of a universal one-way function.

Proposition 2.4.1 *There exists a polynomial-time computable function which is (strongly) one-way if and only if one-way functions exist.*

Proof: A key observation is that there exist one-way functions if and only if there exist one-way functions which can be evaluated by a quadratic time algorithm. (The choice of the specific time bound is immaterial, what is important is that such a specific time bound exists.) This statement is proven using a padding argument. Details follow.

Let f be an arbitrary one-way function, and let $p(\cdot)$ be a polynomial bounding the time complexity of an algorithm for computing f. Define $g(x'x'') \stackrel{\text{def}}{=} f(x')x''$, where $|x'x''| = p(|x'|)$. An algorithm computing g first parses the input into x' and x'' so that $|x'x''| = p(|x'|)$, and then applies f on x'. The parsing and the other overhead operations can be implemented in quadratic time (in $|x'x''|$), whereas computing $f(x')$ is done within time $p(|x'|) = |x'x''|$ (which is linear in the input length). Hence, g can be computed (by a Turing machine) in quadratic time. The reader can verify that g is one-way using a "reducibility argument" analogous to the one used in the proof of Proposition 2.2.5.

We now present a (universal one-way) function, denoted $f_{\rm uni}$.

$$f_{\rm uni}(\mathrm{desc}(M), x) \stackrel{\text{def}}{=} (\mathrm{desc}(M), M(x))$$

where $\mathrm{desc}(M)$ is a description of Turing machine M, and $M(x)$ is defined as the output of M on input x if M runs at most quadratic time on x, and as x otherwise. Clearly, $f_{\rm uni}$ can be computed in polynomial-time by a universal machine which uses a step counter. To show that $f_{\rm uni}$ is one-way we use a "reducibility argument". By the above observation, we know that there exist a one-way function g which is computed in quadratic time. Let M_g be the quadratic time machine computing g. Clearly, an (efficient) algorithm inverting $f_{\rm uni}$ on inputs of the form $f_{\rm uni}(\mathrm{desc}(M_g), U_n)$, with probability $\epsilon(n)$, can be easily modified into an (efficient) algorithm inverting g on inputs of the form $g(U_n)$, with probability $\epsilon(n)$. It follows that an algorithm inverting $f_{\rm uni}$ with probability $\epsilon(n)$, on strings of length $|\mathrm{desc}(M_g)| + n$, yields an algorithm inverting g with probability $\frac{\epsilon(n)}{2^{|\mathrm{desc}(M_g)|}}$ on strings of length n. Hence, if $f_{\rm uni}$ is not weakly one-way then also g cannot be weakly one-way.

Using Theorem 2.3.2, the proposition follows. ■

The observation, that it suffices to consider one-way functions which can be evaluated within a specific time bound, is crucial to the construction of $f_{\rm uni}$. The reason being, that it is not possible to construct a polynomial-time machine which is universal for the class of polynomial-time machines (i.e., a polynomial-time machine that can "simulate" all polynomial-time machines). It is however possible to construct, for every polynomial $p(\cdot)$, a polynomial-time machine that is universal for the class of machines with running-time bounded by $p(\cdot)$.

The impracticality of the suggestion to use $f_{\rm uni}$ as a one-way function stems from the fact that $f_{\rm uni}$ is likely to be hard to invert only on huge input lengths.

2.4.2 One-Way Functions as Collections

The formulation of one-way functions, used in so far, is suitable for an abstract discussion. However, for describing many natural candidates for one-way functions, the following formulation (although being more cumbersome) is more adequate. Instead of viewing one-way functions as functions operating on an infinite domain (i.e., $\{0,1\}^*$), we consider infinite collections of of functions each operating on a finite domain. The functions in the collection share a single evaluating algorithm, that given as input a succint representation of a function and an element in its domain, return the value of the specified function at the given point. The formulation of a collection of functions is also useful for the presentation of trapdoor permutations and clawfree functions (see the next two subsections). We start with the following definition.

Definition 2.4.2 (collection of functions): *A* collection of functions *consists of an infinite set of* indices*, denoted $\overline{I}$, a finite set D_i, for each $i \in \overline{I}$, and a function f_i defined over D_i.*

We will only be interested in collections of functions that can be applied. As hinted above, a necessary condition for applying a collection of functions is the existence of an efficient function-evaluating algorithm (denoted F) that, on input $i \in \overline{I}$ and x, returns $f_i(x)$. Yet, this condition by itself does not suffice. We need to be able to (randomly) select an index, specifying a function over a sufficiently large domain, as well as to be able to (randomly) select an element of the domain (when given the domain's index). The sampling property of the index set is captured by an efficient algorithm (denoted I) that on input an integer n (presented in unary) randomly selects an poly(n)-bit long index, specifying a function and its associated domain. (As usual unary presentation is used to enhence the standard association of efficient algorithms with those running in time polynomial in their length.) The sampling property of the domains is captured by an efficient algorithm (denoted D) that on input an index i randomly selects an element in D_i. The one-way property of the collection is captured by requiring that every efficient algorithm, when given an index of a function and an element in its range, fails to invert the function, except for with negligible probability. The probability is taken over the distribution induced by the sampling algorithms I and D.

Definition 2.4.3 (collection of one-way functions): *A collection of functions, $\{f_i : D_i \mapsto \{0,1\}^*\}_{i\in\overline{I}}$, is called* strongly (resp., weakly) one-way *if there exists three probabilistic polynomial-time algorithms, I, D and F, so that the following two conditions hold*

1. easy to sample and compute: *The output distribution of algorithm I, on input 1^n, is a random variable assigned values in the set $\overline{I} \cap \{0,1\}^n$. The output distribution of algorithm D, on input $i \in \overline{I}$, is a random variable assigned values in D_i. On input $i \in \overline{I}$ and $x \in D_i$, algorithm F always outputs $f_i(x)$.*

2. hard to invert (version for strongly one-way): *For every probabilistic polynomial-time algorithm, A', every polynomial $p(\cdot)$, and all sufficiently large n's*

$$\mathsf{Pr}\left(A'(f_{I_n}(X_n), I_n) \in f_{I_n}^{-1} f_{I_n}(X_n)\right) < \frac{1}{p(n)}$$

   where I_n is a random variable describing the output distribution of algorithm I on input 1^n, and X_n is a random variable describing the output of algorithm D on input (random variable) I_n.
   (The version for weakly one-way collections is analogous.)

We may relate to a collection of one-way functions by indicating the corresponding triplet of algorithms. Hence, we may say that a *triplet of probabilistic polynomial-time algorithms*, (I, D, F), *constitutes a collection of one-way functions* if there exists a collection of functions for which these algorithms satisfy the above two conditions.

We stress that the output of algorithm I, on input 1^n, *is not* necessarily distributed *uniformly* over $\overline{I} \cap \{0,1\}^n$. Furthermore, it is not even required that $I(1^n)$ is not entirely concentrated on one single string. Likewise, the output of algorithm D, on input i, *is not* necessarily distributed *uniformly* over D_i. Yet, the hardness-to-invert condition implies that $D(i)$ cannot be mainly concentrated on polynomially many (in $|i|$) strings. We stress that the collection is hard to invert with respect to the distribution induced by the algorithms I and D (in addition to depending as usual on the mapping induced by the function itself). Clearly, a collection of one-way functions can be represented as a one-way function and vice versa (see Exercise 12), yet each formulation has its advantages. In the sequel we use the formulation of a collection of one-way functions in order to present popular candidates of one-way functions.

To allow less cumbersome presentation of natural candidates of one-way collections (of functions), we relax Definition 2.4.3 in two ways. First, we allow the index sampling algorithm to output, on input 1^n, indices of length $p(n)$, where $p(\cdot)$ is some polynomial. Secondly, we allow all algorithms to fail with negligible probability. Most importantly, we allow the index sampler I to output strings not in $\overline{I}$ as long as the probability that $I(1^n) \notin \overline{I} \cap \{0,1\}^{p(n)}$ is a negligible function in n. (The same relaxations can be made when discussing trapdoor permutations and clawfree functions.)

2.4.3 Examples of One-way Collections (RSA, Factoring, DLP)

In this subsection we present several popular collections of one-way functions, based on computation number theory (e.g., RSA and Discrete Exponentiation). In the exposition which follows, we assume some knowledge of elementary number theory and some familiarity with simple number theoretic algorithms. Further discussion of the relevant number theoretic material is presented in Appendix [missing(app-cnt)]

2.4.3.1 The RSA function

The RSA collection of functions has an index set consisting of pairs (N, e), where N is a product of two $(\frac{1}{2} \cdot \log_2 N)$-bit primes, denoted P and Q, and e is an integer smaller than N and relatively prime to $(P-1)\cdot(Q-1)$. The function of index (N, e), has domain $\{1, ..., N\}$ and maps the domain element x to $x^e \bmod N$. Using the fact that e is relatively prime to $(P-1)\cdot(Q-1)$, it can be shown that the fuction is in fact a permutation over its domain. Hence, the RSA collection is a collection of permutations.

We first substantiate the fact that the RSA collection satisfies the first condition of the definition of a one-way collection (i.e., that it is easy to sample and compute). To this end, we present the triplet of algorithms $(I_{\mathrm{RSA}}, D_{\mathrm{RSA}}, F_{\mathrm{RSA}})$.

On input 1^n, algorithm I_{RSA} selects uniformly two primes, P and Q, such that $2^{n-1} \leq P < Q < 2^n$, and an integer e such that e is relatively prime to $(P-1)\cdot(Q-1)$. Algorithm I_{RSA} terminates with output (N, e), where $N = P \cdot Q$. For an efficient implementation of I_{RSA}, we need a probabilistic polynomial-time algorithms for generating uniformly distributed primes. Such an algorithm does exist. However, it is more efficient to generate two primes by selecting two integers uniformly in the interval $[2^{n-1}, 2^n - 1]$ and checking via a fast randomized primality test whether these are indeed primes (this way we get, with exponentially small probability, an output which is not of the desired form). For more details concerning the uniform generation of primes see Appendix [missing(app-cnt)].

As for algorithm D_{RSA}, on input (N, e), it selects (almost) uniformly an element in the set $D_{N,e} \stackrel{\mathrm{def}}{=} \{1, ..., N\}$. The output of F_{RSA}, on input $((N, e), x)$, is

$$RSA_{N,e}(x) \stackrel{\mathrm{def}}{=} x^e \bmod N$$

It is not known whether factoring N can be reduced to inverting $RSA_{N,e}$, and in fact this is a well-known open problem. We remark that the best algorithms known for inverting $RSA_{N,e}$ proceed by (explicitly or implicitly) factoring N. In any case it is widely believed that the RSA collection is hard to invert.

In the above description $D_{N,e}$ corresponds to the additive group mod N (and hence contain N elements). Alternatively, the domain $D_{N,e}$ can be restricted to the elements of the multiplicative group modulo N (and hence contain $(P-1)\cdot(Q-1) \approx N - 2\sqrt{N} \approx N$ elements). A modified domain sampler may work by selecting an element in $\{1, ..., N\}$ and discarding the unlikely cases in which the selected element is not relatively prime to N. The function $RSA_{N,e}$ defined above indues a permutation on the multiplicative group modulo N. The resulting collection is as hard to invert as the original one. (A proof of this statement is left as an exercise to the reader.) The question which formulation to prefer seems to be a matter of personal taste.

2.4.3.2 The Rabin function

The Rabin collection of functions is defined analogously to the RSA collection, except that the function is squaring modulo N (instead of raising to the power e mod N). Namely,

$$Rabin_N(x) \stackrel{\text{def}}{=} x^2 \bmod N$$

This function, however, does not induces a permutation on the multiplicative group modulo N, but is rather a 4-to-1 mapping on the multiplicative group modulo N.

It can be shown that extracting square roots modulo N is computationally equivalent to factoring N (i.e., the two tasks are reducible to one another via probabilistic polynomial-time reductions). For details see Exercise 15. Hence, squaring modulo a composite is a collection of one-way functions if and only if factoring is intractable. We remind the reader that it is generally believed that integer factorization is intractable.

2.4.3.3 The Factoring Permutations

For a special subclass of the integers, known by the name of *Blum Integers*, the function $Rabin_N(\cdot)$ defined above induces a permutation on the quadratic residues modulo N. We say that r is a *quadratic residue mod* N if there exists an integer x such that $r \equiv x^2 \bmod N$. We denote by Q_N the set of quadratic residues in the multiplicative group mod N. For purposes of this paragraph, we say that N is a Blum Integer if it is the product of two primes, each congruent to 3 mod 4. It can be shown that when N is a Blum integer, each element in Q_N has a unique square root which is also in Q_N, and it follows that in this case the function $Rabin_N(\cdot)$ induces a permutation over Q_N. This leads to the introduction of the following collection, $SQR \stackrel{\text{def}}{=} (I_{BI}, D_{QR}, F_{SQR})$, of permutations. On input 1^n, algorithm I_{BI} selects uniformly two primes, P and Q, such that $2^{n-1} \leq P < Q < 2^n$ and $P \equiv Q \equiv 3 \bmod 4$, and outputs $N = P \cdot Q$. It is assumed that the density of such primes is non-negligible and thus that this step can be efficiently implemented. On input N, algorithm D_{QR}, uniformly selects an element of Q_N, by uniformly selecting an element of the multiplicative group modulo N, and squaring it mod N. Algorithm F_{SQR} is defined exactly as in the Rabin collection. The resulting collection is one-way, provided that factoring is intractable also for the set of Blum integers (defined above).

2.4.3.4 Discrete Logarithms

Another computational number theoretic problem which is widely believed to be intractable is that of extracting discrete logarithms in a finite field (and in particular of prime cardinality). The DLP collection of functions, borrowing its name (and one-wayness) from the *Discrete Logarithm Problem*, is defined by the triplet of algorithms $(I_{\mathrm{DLP}}, D_{\mathrm{DLP}}, F_{\mathrm{DLP}})$.

On input 1^n, algorithm I_{DLP} selects uniformly a prime, P, such that $2^{n-1} \leq P < 2^n$, and a primitive element G in the multiplicative group modulo P (i.e., a generator of this cyclic group), and terminates with output (P, G). There exists a probabilistic polynomial-time algorithm for uniformly generating primes together with the prime factorization of $P-1$, where P is the prime generated (see Appendix `[missing(app-cnt)]`). Alternatively, one may uniformly generate a prime P of the form $2Q+1$, where Q is also a prime. (In the latter case, however, one has to assume the intractability of DLP with respect to such primes. We remark that such primes are commonly believed to be the hardest for DLP.) Using the factorization of $P-1$ one can find a primitive element by selecting an element of the group at random and checking whether it has order $P-1$ (by raising to powers which non-trivially divide $P-1$).

Algorithm D_{DLP}, on input (P,G), selects uniformly a residue modulo $P-1$. Algorithm F_{DLP}, on input $((P,G),x)$, halts outputting

$$DLP_{P,G}(x) \stackrel{\mathrm{def}}{=} G^x \bmod P$$

Hence, inverting $DLP_{P,G}$ amounts to extracting the discrete logarithm (to base G) modulo P. For every (P,G) of the above form, the function $DLP_{P,G}$ induces a 1-1 and onto mapping from the additive group mod $P-1$ to the multiplicative group mod P. Hence, $DLP_{P,G}$ induces a permutation on the the set $\{1,...,P-1\}$.

Exponentiation in other groups is also a reasonable candidate for a one-way function, provided that the discrete logarithm problem for the group is believed to be hard. For example, it is believed that the logarithm problem is hard in the group of points on an Elliptic curve.

Author's Note: *fill-in more details*

2.4.4 Trapdoor one-way permutations

2.4.4.1 The Definition

The formulation of collections of one-way functions is convenient as a starting point to the definition of trapdoor permutations. Loosely speaking, these are collections of one-way permutations, $\{f_i\}$, with the extra property that f_i is efficiently inverted once given as auxiliary input a "trapdoor" for the index i. The trapdoor of index i, denoted by $t(i)$, *can not* be efficiently computed from i, yet one can efficiently generate corresponding pairs $(i,t(i))$.

Definition 2.4.4 (collection of trapdoor permutations): *Let I be a probabilistic algorithm, and let $I_1(1^n)$ (resp. $I_2(1^n)$) denote the first (resp. second) half of the output of $I(1^n)$. A triple of algorithms, (I,D,F), is called a* collection of strong (resp. weak) trapdoor permutations *if the following two conditions hold*

1. the algorithms induce a collection of one-way permutations: *The triple (I_1,D,F) constitutes a collection of one-way permutations.*

2. easy to invert with trapdoor: *There exists a (deterministic) polynomial-time algorithm, denoted F^{-1}, so that for every (i,t) in the range of I and for every $x \in D_i$, it holds that $F^{-1}(t, F(i,x)) = x$.*

A useful relaxation of the above conditions is to require that they are satisfied with overwhelmingly high probability. Namely, the index generating algorithm, I, is allowed to output, with negligible probability, pairs (i,t) for which either f_i is not a permutation or $F^{-1}(t, F(i,x)) = x$ does not hold for all $x \in D_i$. On the other hand, one typically requires that the domain sampling algorithm (i.e., D), produces almost uniform distribution on the corresponding domain. Putting all these modifications together, we obtain the following version. (We also take the oppertunity to present a slightly different formulation.)

Definition 2.4.5 (collection of trapdoor permutations, revisited): *Let $\overline{I} \subseteq \{0,1\}^*$. A* collection of permutations *with* indices *in $\overline{I}$, is a set $\{f_i : D_i \mapsto D_i\}_{i \in \overline{I}}$ so that each f_i is 1-1 on the corresponding D_i. Such a collection is called a* trapdoor permutation *if there exists 4 probabilistic polynomial-time algorithms I, D, F, F^{-1} so that the following five conditions hold.*

1. (index and trapdoor selection): *For every n,*
$$\mathsf{Pr}(I(1^n) \in \overline{I} \times \{0,1\}^*) \;>\; 1 - 2^{-n}$$
2. (selection in domain): *For every $i \in \overline{I}$,*
   (a) $\mathsf{Pr}(D(i) \in D_i) \;>\; 1 - 2^{-n}$*. Thus, without loss of generality, $D_i \subseteq \{0,1\}^{\mathrm{poly}(|i|)}$.*
   (b) *Conditioned on $D(i) \in D_i$, the output is uniformly distributed in D_i. That is, for every $x \in D_i$,*
$$\mathsf{Pr}(D(i) = x \mid D(i) \in D_i) \;=\; \frac{1}{|D_i|}$$
3. (efficient evaluation): *For every $i \in \overline{I}$ and $x \in D_i$,*
$$\mathsf{Pr}(F(i,x) = f_i(x)) \;>\; 1 - 2^{-n}$$
4. (hard to invert): *For every family of polynomial-size circuits, $\{C_n\}_{n \in \mathbb{N}}$, every positive polynomial $p(\cdot)$, and all sufficiently large n's*
$$\mathsf{Pr}\left(C_n(f_{I_n}(X_n), I_n) = X_n\right) \;<\; \frac{1}{p(n)}$$
where I_n is a random variable describing the distribution of the first element in the output of $I(1^n)$, and $X_n \stackrel{\mathrm{def}}{=} D(I_n)$.
5. (inverting with trapdoor): *For every pair (i,t) in the range of I, and every $x \in D_i$,*
$$\mathsf{Pr}(F^{-1}(t, f_i(x)) = x) \;>\; 1 - 2^{-n}$$

2.4.4.2 The RSA (or factoring) Trapdoor

The RSA collection presented above can be easily modified to have the trapdoor property. To this end algorithm I_{RSA} should be modified so that it outputs both the index (N, e) and the trapdoor (N, d), where d is the multiplicative inverse of e modulo $(P-1)\cdot(Q-1)$ (note that e has such inverse since it has been chosen to be relatively prime to $(P-1)\cdot(Q-1)$). The inverting algorithm F_{RSA}^{-1} is identical to the algorithm F_{RSA} (i.e., $F_{\mathrm{RSA}}^{-1}((N,d),y) = y^d \bmod N$). The reader can easily verify that

$$F_{\mathrm{RSA}}\left((N,d), F_{\mathrm{RSA}}\left((N,e),x\right)\right) = x^{ed} \bmod N$$

indeed equals x for every x in the multiplicative group modulo N. In fact, one can show that $x^{ed} \equiv x \pmod{N}$ for every x (even in case x is not relatively prime to N).

We remark that the Rabin collection presented above can be easily modified in an analogous manner, enabling to efficiently compute all 4 square roots of a given quadratic residue (mod N). The square roots mod N can be computed by extracting a square root modulo each of the primes factors of N and combining the result using the Chinese Reminder Theorem. Efficient algorithms for extracting square root modulo a given prime are known. Furthermore, in case the prime, P, is congruent to 3 mod 4, the square roots of x mod P can be computed by raising x to the power $\frac{P+1}{4}$ (while reducing the intermediate results mod P). Furthermore, in case N is a Blum integer, the collection SQR, presented above, forms a collection of trapdoor permutations (provided of course that factoring is hard).

2.4.5 * Clawfree Functions

2.4.5.1 The Definition

Loosely speaking, a clawfree collection consists of a set of pairs of functions which are easy to evaluate, both have the same range, and yet it is infeasible to find a range element together with preimages of it under each of these functions.

Definition 2.4.6 (clawfree collection): *A collection of pairs of functions consists of an infinite set of indices, denoted $\overline{I}$, two finite sets D_i^0 and D_i^1, for each $i \in \overline{I}$, and two functions f_i^0 and f_i^1 defined over D_i^0 and D_i^1, respectively. Such a collection is called* clawfree *if there exists three probabilistic polynomial-time algorithms, I, D and F, so that the following conditions hold*

1. easy to sample and compute: *The random variable $I(1^n)$ is assigned values in the set $\overline{I} \cap \{0,1\}^n$. For each $i \in \overline{I}$ and $\sigma \in \{0,1\}$, the random variable $D(\sigma, i)$ is distributed over D_i^σ and $F(\sigma, i, x) = f_i^\sigma(x)$.*

2. identical range distribution: *For every i in the index set $\overline{I}$, the random variables $f_i^0(D(0,i))$ and $f_i^1(D(1,i))$ are identically distributed.*

3. hard to form claws: *A pair* (x,y) *satisfying* $f_i^0(x) = f_i^1(y)$ *is called a* claw *for index* i*. Let* C_i *denote the set of claws for index* i*. It is required that for every probabilistic polynomial-time algorithm,* A'*, every polynomial* $p(\cdot)$*, and all sufficiently large* n*'s*

$$\mathsf{Pr}\left(A'(I_n) \in C_{I_n}\right) < \frac{1}{p(n)}$$

where I_n *is a random variable describing the output distribution of algorithm* I *on input* 1^n*.*

The first requirement in Definition 2.4.6 is analogous to what appears in Definition 2.4.3. The other two requirements (in Definition 2.4.6) are kind of conflicting. On one hand, it is required that that claws do exist (to say the least), whereas on the other hand it is required that calws cannot be efficiently found. Clearly, a clawfree collection of functions yields a collection of strong one-way functions (see Exercise 16). A special case of interest is when both domains are identical (i.e., $D_i \stackrel{\rm def}{=} D_i^0 = D_i^1$), the random variable $D(\sigma,i)$ is uniformly distributed over D_i, and the functions, f_i^0 and f_i^1, are permutations over D_i. Such a collection is called a collection of (clawfree) permutations.

Again, a useful relaxation of the conditions of Definition 2.4.6 is obtained by allowing the algorithms (i.e., I, D and F) to fail with negligible probability.

An additional property that a (clawfree) collection may (or may not) have is an efficiently recognizable index set (i.e., an probabilistic polynomial-time algorithm for determining whether a give string is $\overline{I}$). This property is useful in some applications of clawfree collections (hence this discussion). Efficient recognition of the index set may be important since the function-evaluating algorithm F may induce functions also in case its second input (which is supposedly an index) is not in $\overline{I}$. In this case it is no longer guaranteed that the induced pair of functions has identical range distribution. In some applications (e.g., see section 4.8), dishonest parties may choose, on purpose, an illegal index and try to capitalize on the induce functions having different range distributions.

2.4.5.2 The DLP Clawfree Collection

We now turn to show that clawfree collections do exists under specific reasonable intractability assumptions. We start by presenting such a collection under the assumption that the Discrete Logarithm Problem (DLP) for fields of prime cardinality is intractable.

Following is the description a collection of clawfree *permutations* (based on the above assumption). The index sets consists of triples, $P,G,Z)$, where P is a prime, G is a primitive element mod P, and Z is an element in the field (of residues mod P). The index sampling algorithm, selects P and G as in the DLP collection presented in Subsection 2.4.3, and Z is selected uniformly among the residues mod P. The domain of both functions with index (P,G,Z) is identical, and equals the set $\{1,...,P-1\}$, and the domain sampling algorithm selects uniformly from this set. As for the functions themselves, we set

$$f^{\sigma}_{P,G,Z}(x) \stackrel{\rm def}{=} Z^{\sigma} \cdot G^{x} \bmod P$$

The reader can easily verify that both functions are permutations over $\{1, ..., P-1\}$. Also, the ability to form a claw for the index (P, G, Z) yields the ability to find the discrete logarithm of Z mod P to base G (since $G^x \equiv Z \cdot G^y \bmod P$ yields $G^{x-y} \equiv Z \bmod P$). Hence, ability to form claws for a non-negligible fraction of the index set translates to a contradiction to the DLP intractability assumption.

The above collection *does not* have the additional property of having an efficiently recognizable index set, since it is not known how to efficiently recognize primitive elements modulo a prime. This can be amended by making a slightly stronger assumption concerning the intractability of DLP. Specifically, we assume that DLP is intractable even if one is given the factorization of the size of the multiplicative group (i.e., the factorization of $P-1$) as additional input. Such an assumption allows to add the factorization of $P-1$ into the description of the index. This makes the index set efficiently recognizable (since one can first test P for primality, as usual, and next test whether G is a primitive element by raising it to powers of the form $(P-1)/Q$ where Q is a prime factor of $P-1$). If DLP is hard also for primes of the form $2Q+1$, where Q is also a prime, life is even easier. To test whether G is a primitive element mod P one just computes $G^2 \pmod P$ and $G^{(P-1)/2} \pmod P$, and checks whether either of them equals 1.

2.4.5.3 The Factoring Clawfree Collection

We now show that a clawfree collection (of functions) does exists under the assumption that integer factorization is infeasible for integers which are the product of two primes each congruent to 3 mod 4. Such composite numbers, hereafter referred to as *Blum integers*, have the property that the Jacobi symbol of -1 (relative to them) is 1 and half of the square roots of each quadratic residue, in the corresponding multiplicative group (modulo this composite), have Jacobi symbol 1 (see Appendix `[missing(app-cnt)]`).

The index set of the collection consists of all Blum integers which are composed of two primes of equal length. The index selecting algorithm, on input 1^n, uniformly select such an integers, by uniformly selecting two (n-bit) primes each congruent to 3 mod 4, and outputting their product, denoted N. Let J_N^{+1} (respectively, J_N^{-1}) denote the set of residues in the multiplicative group modulo N with Jacobi Symbol $+1$ (resp., -1). The functions of index N, denoted f_N^0 and f_N^1, consist both of squaring modulo N, but their corresponding domains are disjoint. The domain of function f_N^σ equals the set $J_N^{(-1)^\sigma}$. The domain sampling algorithm, denoted D, uniformly selects an element of the corresponding domain as follows. Specifically, on input (σ, N) algorithm D uniformly selects polynomially many residues mod N, and outputs the first residue with Jacobi Symbol $(-1)^\sigma$.

The reader can easily verify that both $f_N^0(D(0,N))$ and $f_N^1(D(1,N))$ are uniformly distributed over the set of quadratic residues mod N. The difficult of forming claws follows from the fact that a claw yield two residues, $x \in J_N^{+1}$ and $y \in J_N^{-1}$ such that $x^2 \equiv y^2 \pmod N$. Since $-1 \in J_N^{+1}$, it follows that $x \neq \pm y$ and the gcd of $x \pm y$ and N yields a factorization of N.

The above collection *does not* have the additional property of having an efficiently recognizable index set, since it is not even known how to efficiently distinguish products of two primes from products of more than two primes.

2.4.6 On Proposing Candidates

Although we do believe that one-way functions exist, their *mere* existence does not suffice for practical applications. Typically, an application which is based on one-way functions requires the specification of a concrete (candidate one-way) function. As explained above, the observation concerning the existence of a universal one-way function is of little practical significance. Hence, the problem of proposing reasonable candidates for one-way functions is of great practical importance. Everyone understands that such a reasonable candidate (for a one-way function) should have a very efficient algorithm for evaluating the function. (In case the "function" is presented as a collection of one-way functions, especially the domain sampler and function-evaluation algorithm should be very efficient.) However, people seem less careful in *seriously considering* the difficulty of inverting the candidates that they propose. We stress that the candidate has to be difficult to invert on "the average" and not only on the worst case, and that "the average" is taken with respect to the instance-distribution determined by the candidate function. Furthermore, "hardness on the average" (unlike worst case analysis) is extremely sensitive to the instance-distribution. Hence, one has to be extremely careful in deducing average-case complexity with respect to one distribution from the average-case complexity with respect to another distribution. The short history of the field contains several cases in which this point has been ignored and consequently bad suggestions has been made.

Consider for example the following suggestion to base one-way functions on the conjectured difficulty of the Graph Isomorphism problem. Let $f_{\rm GI}(G, \pi) = (G, \pi G)$, where G is an undirected graph, π is a permutation on its vertex set, and πG denotes the graph resulting by renaming the vertices of G using π (i.e., $(\pi(u), \pi(v))$ is an edge in πG iff (u, v) is an edge in G). Although it is indeed believed that Graph Isomorphism cannot be solved in polynomial-time, it is easy to see that $F_{\rm GI}$ is easy to invert on most instances (e.g., use vertex degree statistics to determine the isomorphism).

2.5 Hard-Core Predicates

Loosely speaking, saying that a function f is one-way means that given y it is infeasible to find a preimage of y under f. This does not mean that it is infeasible to find out partial information about the preimage of y under f. Specifically it may be easy to retrieve half of the bits of the preimage (e.g., given a one-way function f consider the function g defined by $g(x, r) \stackrel{\rm def}{=} (f(x), r)$, for every $|x| = |r|$). The fact that one-way functions do not necessarily hide partial information about their preimage limits their "direct applicability" to tasks as secure encryption. Fortunately, assuming the existence of one-way functions, it is possible to construct one-way functions which hide specific partial information about their preimage (which is easy to compute from the preimage itself). This partial information can be considered as a "hard core" of the difficulty of inverting f.

2.5.1 Definition

A *polynomial-time* predicate b, is called a hard-core of a function f if all efficient algorithm, given $f(x)$, can guess $b(x)$ only with success probability which is negligibly better than half.

Definition 2.5.1 (hard-core predicate): *A polynomial-time computable predicate* $b : \{0,1\}^* \mapsto \{0,1\}$ *is called a* hard-core *of a function* f *if for every probabilistic polynomial-time algorithm* A'*, every polynomial* $p(\cdot)$*, and all sufficiently large* n*'s*

$$\mathsf{Pr}\left(A'(f(U_n))=b(U_n)\right) < \frac{1}{2} + \frac{1}{p(n)}$$

It follows that if b is a hard-core predicate (for any function) then $b(U_n)$ should be almost unbiased (i.e., $|\mathsf{Pr}(b(U_n)=0) - \mathsf{Pr}(b(U_n)=1)|$ must be a negligible function in n). As b itself is polynomial-time computable the failure of efficient algorithms to approximate $b(x)$ from $f(x)$ (with success probability significantly more than half) must be due to either an information loss of f (i.e., f not being one-to-one) or to the difficulty of inverting f. For example, the predicate $b(\sigma\alpha) = \sigma$ is a hard-core of the function $f(\sigma\alpha) \stackrel{\text{def}}{=} 0\alpha$, where $\sigma \in \{0,1\}$ and $\alpha \in \{0,1\}^*$. Hence, in this case the fact that b is a hard-core of the function f is due to the fact that f losses information (specifically the first bit σ). On the other hand, in case f losses no information (i.e., f is one-to-one) hard-cores for f exist only if f is one-way (see Exercise 19). Finally, we note that for every b and f, there exist obvious algorithms which guess $b(U_n)$ from $f(U_n)$ with success probability at least half (e.g., either an algorithm A_1 that regardless of its input answers with a uniformly chosen bit, or, in case b is not biased towards 0, the constant algorithm $A_2(x) \stackrel{\text{def}}{=} 1$).

Simple hard-core predicates are known for the RSA, Rabin, and DLP collections (presented in Subsection 2.4.3), provided that the corresponding collections are one-way. Specifically, the least significant bit is a hard-core for the RSA collection, provided that the RSA collection is one-way. Namely, assuming that the RSA collection is one-way, it is infeasible to guess (with success probability significantly greater than half) the least significant bit of x from $RSA_{N,e}(x) = x^e \bmod N$. Likewise, assuming that the DLP collection is one-way, it is infeasible to guess whether $x < \frac{P}{2}$ when given $DLP_{P,G}(x) = G^x \bmod P$. In the next subsection we present a general result of the kind.

2.5.2 Hard-Core Predicates for any One-Way Function

Actually, the title is inaccurate, as we are going to present hard-core predicates only for (strong) one-way functions of special form. However, every (strong) one-way function can be easily transformed into a function of the required form, with no substantial loss in either "security" or "efficiency".

Theorem 2.5.2 *Let f be an arbitrary strong one-way function, and let g be defined by $g(x,r) \stackrel{\text{def}}{=} (f(x), r)$, where $|x|=|r|$. Let $b(x,r)$ denote the inner-product mod 2 of the binary vectors x and r. Then the predicate b is a hard-core of the function g.*

In other words, the theorem states that if f is strongly one-way then it is infeasible to guess the exclusive-or of a random subset of the bits of x when given $f(x)$ and the subset itself. We stress that the theorem requires that f is strongly one-way and that the conclusion is false if f is only weakly one-way (see Exercise 19). We point out that g maintains properties of f such as being length-preserving and being one-to-one. Furthermore, an analogous statement holds for collections of one-way functions with/without trapdoor etc.

Proof: The proof uses a "reducibility argument". This time inverting the function f is reduced to predicting $b(x,r)$ from $(f(x), r)$. Hence, we assume (for contradiction) the existence of an efficient algorithm predicting the inner-product with advantage which is not negligible, and derive an algorithm that inverts f with related (i.e. not negligible) success probability. This contradicts the hypothesis that f is a one-way function.

Let G be a (probabilistic polynomial-time) algorithm that on input $f(x)$ and r tries to predict the inner-product (mod 2) of x and r. Denote by $\varepsilon_G(n)$ the (overall) advantage of algorithm G in predicting $b(x,r)$ from $f(x)$ and r, where x and r are uniformly chosen in $\{0,1\}^n$. Namely,

$$\varepsilon_G(n) \stackrel{\text{def}}{=} \mathsf{Pr}\left(G(f(X_n), R_n) = b(X_n, R_n)\right) - \frac{1}{2}$$

where here and in the sequel X_n and R_n denote two independent random variables, each uniformly distributed over $\{0,1\}^n$. Assuming, to the contradiction, that b is not a hard-core of g means that exists an efficient algorithm G, a polynomial $p(\cdot)$ and an infinite set N so that for every $n \in N$ it holds that $\varepsilon_G(n) > \frac{1}{p(n)}$. We restrict our attention to this algorithm G and to n's in this set N. In the sequel we shorthand ε_G by ε.

Our first observation is that, on at least an $\frac{\varepsilon(n)}{2}$ fraction of the x's of length n, algorithm G has an $\frac{\varepsilon(n)}{2}$ advantage in predicting $b(x, R_n)$ from $f(x)$ and R_n. Namely,

Claim 2.5.2.1: there exists a set $S_n \subseteq \{0,1\}^n$ of cardinality at least $\frac{\varepsilon(n)}{2} \cdot 2^n$ such that for every $x \in S_n$, it holds that

$$s(x) \stackrel{\text{def}}{=} \mathsf{Pr}(G(f(x), R_n) = b(x, R_n)) \geq \frac{1}{2} + \frac{\varepsilon(n)}{2}$$

This time the probability is taken over all possible values of R_n and all internal coin tosses of algorithm G, whereas x is fixed.
Proof: The observation follows by an averaging argument. Namely, write $\mathsf{E}(s(X_n)) = \frac{1}{2} + \varepsilon(n)$, and apply Markov Inequality.$\Box$

In the sequel we restrict our attention to x's in S_n. We will show an efficient algorithm that on every input y, with $y = f(x)$ and $x \in S_n$, finds x with very high probability. Contradiction to the (strong) one-wayness of f will follow by noting that $\mathsf{Pr}(U_n \in S_n) \geq \frac{\varepsilon(n)}{2}$.

The next three paragraphs consist of a motivating discussion. The inverting algorithm, that uses algorithm G as subroutine, will be formally described and analyzed later.

A motivating discussion

Consider a fixed $x \in S_n$. By definition $s(x) \geq \frac{1}{2}+\frac{\varepsilon(n)}{2} > \frac{1}{2}+\frac{1}{2p(n)}$. Suppose, for a moment, that $s(x) > \frac{3}{4}+\frac{1}{2p(n)}$. Of course there is no reason to believe that this is the case, we are just doing a mental experiment. In this case (i.e., of $s(x) > \frac{3}{4} + \frac{1}{\text{poly}(|x|)}$) retrieving x from $f(x)$ is quite easy. To retrieve the i^{th} bit of x, denoted x_i, we randomly select $r \in \{0,1\}^n$, and compute $G(f(x),r)$ and $G(f(x),r \oplus e^i)$, where e^i is an n-dimensional binary vector with 1 in the i^{th} component and 0 in all the others, and $v \oplus u$ denotes the addition mod 2 of the binary vectors v and u. Clearly, if both $G(f(x),r) = b(x,r)$ and $G(f(x),r \oplus e^i) = b(x,r \oplus e^i)$, then

$$\begin{aligned} G(f(x),r) \oplus G(f(x),r \oplus e^i) &= b(x,r) \oplus b(x,r \oplus e^i) \\ &= b(x,e^i) \\ &= x_i \end{aligned}$$

since $b(x,r) \oplus b(x,s) \equiv \sum_{i=1}^n x_i r_i + \sum_{i=1}^n x_i s_i \equiv \sum_{i=1}^n x_i (r_i + s_i) \equiv b(x, r \oplus s) \bmod 2$. The probability that both equalities hold (i.e., both $G(f(x),r) = b(x,r)$ and $G(f(x),r \oplus e^i) = b(x,r \oplus e^i)$) is at least $1 - 2 \cdot (\frac{1}{4} - \frac{1}{\text{poly}(|x|)}) > 1 - \frac{1}{\text{poly}(|x|)}$. Hence, repeating the above procedure sufficiently many times and ruling by majority we retrieve x_i with very high probability. Similarly, we can retrieve all the bits of x, and hence invert f on $f(x)$. However, the entire analysis was conducted under (the unjustifiable) assumption that $s(x) > \frac{3}{4}+\frac{1}{2p(|x|)}$, whereas we only know that $s(x) > \frac{1}{2}+\frac{1}{2p(|x|)}$.

The problem with the above procedure is that it doubles the original error probability of algorithm G on inputs of form $(f(x), \cdot)$. Under the unrealistic assumption, that the G's error on such inputs is significantly smaller than $\frac{1}{4}$, the "error-doubling" phenomenon raises no problems. However, in general (and even in the special case where G's error is exactly $\frac{1}{4}$) the above procedure is unlikely to invert f. Note that the error probability of G can not be decreased by repeating G several times (e.g., G may always answer correctly on three quarters of the inputs, and always err on the remaining quarter). What is required is an *alternative way of using* the algorithm G, a way which does not double the original error probability of G. The key idea is to generate the r's in a way which requires applying algorithm G only once per each r (and i), instead of twice. Specifically, we used algorithm G to obtain a "guess" for $b(x, r \oplus e^i)$ and obtain $b(x,r)$ in a different way. The good news are that the error probability is no longer doubled, since we only need to use G to get a "guess" of $b(x, r \oplus e^i)$. The bad news are that we still need to know $b(x,r)$, and it is not clear how we can know $b(x,r)$ without applying G. The answer is that we can guess $b(x,r)$ by ourselves. This is fine if we only need to guess $b(x,r)$ for one r (or logarithmically in $|x|$ many r's), but the problem is that we need to know (and hence guess) $b(x,r)$ for polynomially many r's. An obvious way of guessing these $b(x,r)$'s yields an exponentially vanishing success probability. The solution is to generate these polynomially many r's so that, on one hand they are "sufficiently random" whereas on the other hand we can guess all the $b(x,r)$'s with non-negligible success probability. Specifically, generating the r's in a particular *pairwise independent* manner will satisfy both (seemingly contradictory) requirements. We stress that in case we are successful (in our guesses for the $b(x,r)$'s), we can retrieve x with high probability. Hence, we retrieve x with non-negligible probability.

A word about the way in which the pairwise independent r's are generated (and the corresponding $b(x,r)$'s are guessed) is indeed in place. To generate $m = \text{poly}(n)$ many r's, we uniformly (and independently) select $l \stackrel{\text{def}}{=} \log_2(m+1)$ strings in $\{0,1\}^n$. Let us denote these strings by $s^1,...,s^l$. We then guess $b(x,s^1)$ through $b(x,s^l)$. Let use denote these guesses, which are uniformly (and independently) chosen in $\{0,1\}$, by σ^1 through σ^l. Hence, the probability that all our guesses for the $b(x,s^i)$'s are correct is $2^{-l} = \frac{1}{\text{poly}(n)}$. The different r's correspond to the different non-empty subsets of $\{1,2,...,l\}$. We compute $r^J \stackrel{\text{def}}{=} \oplus_{j\in J} s^j$. The reader can easily verify that the r^J's are pairwise independent and each is uniformly distributed in $\{0,1\}^n$. The key observation is that

$$b(x,r^J) = b(x,\oplus_{j\in J}s^j) = \oplus_{j\in J}b(x,s^j)$$

Hence, our guess for the $b(x,r^J)$'s is $\oplus_{j\in J}\sigma^j$, and with non-negligible probability all our guesses are correct.

Back to the formal argument

Following is a formal description of the inverting algorithm, denoted A. We assume, for simplicity that f is length preserving (yet this assumption is not essential). On input y (supposedly in the range of f), algorithm A sets $n \stackrel{\text{def}}{=} |y|$, and $l \stackrel{\text{def}}{=} \lceil \log_2(2n\cdot p(n)^2+1)\rceil$, where $p(\cdot)$ is the polynomial guaranteed above (i.e., $\epsilon(n) > \frac{1}{p(n)}$ for the infinitely many n's in N). Algorithm A uniformly and independently select $s^1,...,s^l \in \{0,1\}^n$, and $\sigma^1,...,\sigma^l \in \{0,1\}$. It then computes, for every non-empty set $J \subseteq \{1,2,...,l\}$, a string $r^J \leftarrow \oplus_{j\in J}s^j$ and a bit $\rho^J \leftarrow \oplus_{j\in J}\sigma^j$. For every $i \in \{1,...,n\}$ and every *non-empty* $J \subseteq \{1,..,l\}$, algorithm A computes $z_i^J \leftarrow \rho^J \oplus G(y, r^J \oplus e^i)$. Finally, algorithm A sets z_i to be the majority of the z_i^J values, and outputs $z = z_1 \cdots z_n$. (Remark: in an alternative implementation of the ideas, the inverting algorithm, denoted A', tries all possible values for $\sigma^1,...,\sigma^l$, and outputs only one of resulting strings z, with an obvious preference to a string z satisfying $f(z) = y$.)

Following is a detailed analysis of the success probability of algorithm A on inputs of the form $f(x)$, for $x \in S_n$, where $n \in N$. We start by showing that, in case the σ^j's are correct, then the with constant probability, $z_i = x_i$ for all $i \in \{1,...,n\}$. This is proven by bounding from below the probability that the majority of the z_i^J's equals x_i.

Claim 2.5.2.2: For every $x \in S_n$ and every $1 \le i \le n$,

$$\mathsf{Pr}\left(|\{J : b(x,r^J)\oplus G(f(x), r^J \oplus e^i) = x_i\}| > \frac{1}{2}\cdot(2^l-1)\right) > 1 - \frac{1}{2n}$$

where $r^J \stackrel{\text{def}}{=} \oplus_{j\in J}s^j$ and the s^j's are independently and uniformly chosen in $\{0,1\}^n$.

Proof: For every J, define a 0-1 random variable ζ^J, so that ζ^J equals 1 if and only if $b(x,r^J)\oplus G(f(x), r^J \oplus e^i) = x_i$. The reader can easily verify that each r^J is uniformly distributed in $\{0,1\}^n$. It follows that each ζ^J equals 1 with probability $s(x)$, which by $x \in S_n$, is at least $\frac{1}{2}+\frac{1}{2p(n)}$. We show that the ζ^J's are pairwise independent by showing that the r^J's are pairwise independent. For every $J \neq K$ we have, without loss of generality, $j \in J$ and $k \in K - J$. Hence, for every $\alpha, \beta \in \{0,1\}^n$, we have

$$\begin{aligned} \mathsf{Pr}\left(r^K = \beta \mid r^J = \alpha\right) &= \mathsf{Pr}\left(s^k = \beta \mid s^j = \alpha\right) \\ &= \mathsf{Pr}\left(s^k = \beta\right) \\ &= \mathsf{Pr}\left(r^K = \beta\right) \end{aligned}$$

and pairwise independence of the r^J's follows. Let $m \stackrel{\text{def}}{=} 2^l - 1$. Using Chebyshev's Inequality, we get

$$\begin{aligned} \mathsf{Pr}\left(\sum_J \zeta^J \leq \frac{1}{2} \cdot m\right) &\leq \mathsf{Pr}\left(|\sum_J \zeta^J - (\frac{1}{2}+\frac{1}{2p(n)}) \cdot m| \geq \frac{1}{2p(n)} \cdot m\right) \\ &< \frac{\mathrm{Var}(\zeta^{\{1\}})}{(\frac{1}{2p(n)})^2 \cdot (2n \cdot p(n)^2)} \\ &< \frac{\frac{1}{4}}{(\frac{1}{2p(n)})^2 \cdot (2n \cdot p(n)^2)} \\ &= \frac{1}{2n} \end{aligned}$$

The claim now follows. □

Recall that if $\sigma^j = b(x, s^j)$, for all j's, then $\rho^J = b(x, r^J)$ for all non-empty J's. In this case z output by algorithm A equals x, with probability at least half. However, the first event happens with probability $2^{-l} = \frac{1}{2n \cdot p(n)^2}$ independently of the events analyzed in Claim 2.5.2.2. Hence, in case $x \in S_n$, algorithm A inverts f on $f(x)$ with probability at least $\frac{1}{4p(|x|)}$ (whereas, the modified algorithm, A', succeeds with probability $\geq \frac{1}{2}$). Recalling that $|S_n| > \frac{1}{2p(n)} \cdot 2^n$, we conclude that, for every $n \in N$, algorithm A inverts f on $f(U_n)$ with probability at least $\frac{1}{8p(n)^2}$. Noting that A is polynomial-time (i.e., it merely invokes G for $2n \cdot p(n)^2 = \mathrm{poly}(n)$ times in addition to making a polynomial amount of other computations), a contradiction, to our hypothesis that f is strongly one-way, follows. ■

2.5.3 * Hard-Core Functions

We have just seen that every one-way function can be easily modified to have a hard-core predicate. In other words, the result establishes one bit of information about the preimage which is hard to approximate from the value of the function. A stronger result may say that several bits of information about the preimage are hard to approximate. For example, we may want to say that a specific pair of bits is hard to approximate, in the sense that it is infeasible to guess this pair with probability significantly larger than $\frac{1}{4}$. In general, a *polynomial-time* function, h, is called a hard-core of a function f if no efficient algorithm can distinguish $(f(x), h(x))$ from $(f(x), r)$, where r is a random string of length $|h(x)|$. For further discussion of the notion of efficient distinguishability the reader is referred to Section 3.2. We assume for simplicity that h is length regular (see below).

Definition 2.5.3 (hard-core function): *Let $h : \{0,1\}^* \mapsto \{0,1\}^*$ be a polynomial-time computable function, satisfying $|h(x)| = |h(y)|$ for all $|x| = |y|$, and let $l(n) \stackrel{\text{def}}{=} |h(1^n)|$. The function $h : \{0,1\}^* \mapsto \{0,1\}^*$ is called a* hard-core *of a function f if for every probabilistic polynomial-time algorithm D', every polynomial $p(\cdot)$, and all sufficiently large n's*

$$|\mathsf{Pr}\left(D'(f(X_n), h(X_n))=1\right) - \mathsf{Pr}\left(D'(f(X_n), R_{l(n)})=1\right)| < \frac{1}{p(n)}$$

where X_n and $R_{l(n)}$ are two independent random variables the first uniformly distributed over $\{0,1\}^n$, and the second uniformly distributed over $\{0,1\}^{l(n)}$,

Theorem 2.5.4 *Let f be an arbitrary strong one-way function, and let g_2 be defined by $g_2(x,s) \stackrel{\text{def}}{=} (f(x), s)$, where $|s| = 2|x|$. Let $c > 0$ be a constant, and $l(n) \stackrel{\text{def}}{=} \lceil c \log_2 n \rceil$. Let $b_i(x,s)$ denote the inner-product mod 2 of the binary vectors x and $(s_{i+1}, ..., s_{i+n})$, where $s = (s_1, ..., s_{2n})$. Then the function $h(x,s) \stackrel{\text{def}}{=} b_1(x,s) \cdots b_{l(|x|)}(x,s)$ is a hard-core of the function g_2.*

The proof of the theorem follows by combining a proposition concerning the structure of the specific function h with a general lemma concerning hard-core functions. Loosely speaking, the proposition "reduces" the problem of approximating $b(x,r)$ given $g(x,r)$ to the problem of approximating the exclusive-or of any non-empty set of the bits of $h(x,s)$ given $g_2(x,s)$, where b and g are the hard-core and the one-way function presented in the previous subsection. Since we know that the predicate $b(x,r)$ cannot be approximated from $g(x,r)$, we conclude that no exclusive-or of the bits of $h(x,s)$ can be approximated from $g_2(x,s)$. The general lemma states that, for every "logarithmically shrinking" function h' (i.e., h' satisfying $|h'(x)| = O(\log |x|)$), the function h' is a hard-core of a function f' if and only if the exclusive-or of any non-empty subset of the bits of h' cannot be approximated from the value of f'.

Proposition 2.5.5 *Let f, g_2 and b_i's be as above. Let $I(n) \subseteq \{1, 2, ..., l(n)\}$, $n \in \mathbb{N}$, be an arbitrary sequence of non-empty subsets, and let $b_{I(|x|)}(x,s) \stackrel{\text{def}}{=} \oplus_{i \in I(|x|)} b_i(x,s)$. Then, for every probabilistic polynomial-time algorithm A', every polynomial $p(\cdot)$, and all sufficiently large n's*

$$\mathsf{Pr}\left(A'(g_2(U_{3n})) = b_{I(n)}(U_{3n})\right) < \frac{1}{2} + \frac{1}{p(n)}$$

Proof: The proof is by a "reducibility" argument. It is shown that the problem of approximating $b(X_n, R_n)$ given $(f(X_n), R_n)$ is reducible to the problem of approximating

$b_{I(n)}(X_n, S_{2n})$ given $(f(X_n), S_{2n})$, where X_n, R_n and S_{2n} are independent random variable and the last is uniformly distributed over $\{0,1\}^{2n}$. The underlying observation is that, for every $|s| = 2 \cdot |x|$,

$$b_I(x,s) = \oplus_{i \in I} b_i(x,s) = b(x, \oplus_{i \in I} \mathrm{sub}_i(s)$$

where $\mathrm{sub}_i(s_1, ..., s_{2n}) \stackrel{\text{def}}{=} (s_{i+1}, ..., s_{i+n})$. Furthermore, the reader can verify that for every non-empty $I \subseteq \{1, ..., n\}$, the random variable $\oplus_{i \in I} \mathrm{sub}_i(S_{2n})$ is uniformly distributed over $\{0,1\}^n$, and that given a string $r \in \{0,1\}^n$ and such a set I one can efficiently select a string uniformly in the set $\{s : \oplus_{i \in I} \mathrm{sub}_i(s) = r\}$. (Verification of both claims is left as an exercise.)

Now, assume to the contradiction, that there exists an efficient algorithm A', a polynomial $p(\cdot)$, and an infinite sequence of sets (i.e., $I(n)$'s) and n's so that

$$\mathsf{Pr}\left(A'(g_2(U_{3n})) = b_{I(n)}(U_{3n})\right) \geq \frac{1}{2} + \frac{1}{p(n)}$$

We first observe that for n's satisfying the above inequality we can find in probabilistic polynomial time (in n) a set I satisfying

$$\mathsf{Pr}\left(A'(g_2(U_{3n})) = b_I(U_{3n})\right) \geq \frac{1}{2} + \frac{1}{2p(n)}$$

(i.e., by going over all possible I's and experimenting with algorithm A' on each of them). Of course we may be wrong here, but the error probability can be made exponentially small.

We now present an algorithm for approximating $b(x,r)$, from $y \stackrel{\text{def}}{=} f(x)$ and r. On input y and r, the algorithm first finds a set I as described above (this stage depends only on $|x|$ which equals $|r|$). Once I is found, the algorithm uniformly select a string s so that $\oplus_{i \in I} \mathrm{sub}_i(s) = r$, and return $A'(y,s)$. Evaluation of the success probability of this algorithm is left as an exercise. ∎

Lemma 2.5.6 (Computational XOR Lemma): *Let f and h be arbitrary length regular functions, and let $l(n) \stackrel{\text{def}}{=} |h(1^n)|$. Let D be an algorithm. Denote*

$$p \stackrel{\text{def}}{=} \mathsf{Pr}\left(D(f(X_n), h(X_n)) = 1\right) \quad \textit{and} \quad q \stackrel{\text{def}}{=} \mathsf{Pr}\left(D(f(X_n), R_{l(n)}) = 1\right)$$

where X_n and R_l are as above. Let G be an algorithm that on input y, S (and $l(n)$), selects r uniformly in $\{0,1\}^{l(n)}$, and outputs $D(y,r) \oplus 1 \oplus (\oplus_{i \in S} r_i)$, where $r = r_1 \cdots r_l$ and $r_i \in \{0,1\}$. Then,

$$\mathsf{Pr}\left(G(f(X_n), I_l, l(n)) = \oplus_{i \in I_l}(h_i(X_n))\right) = \frac{1}{2} + \frac{p-q}{2^{l(n)}-1}$$

where I_l is a randomly chosen non-empty subset of $\{1, ..., l(n)\}$ and $h_i(x)$ denotes the i^{th} bit of $h(x)$.

It follows that, for logarithmically shrinking h's, the existence of an efficient algorithm that distinguishes (with a gap which is not negligible in n) the random variables $(f(X_n), h(X_n))$ and $(f(X_n), R_{l(n)})$ implies the existence of an efficient algorithm that approximates the exclusive-or of a random non-empty subset of the bits of $h(X_n)$ from the value of $f(X_n)$ with an advantage that is not negligible. On the other hand, it is clear that any efficient algorithm, which approximates an exclusive-or of an non-empty subset of the bits of h from the value of f, can be easily modified to distinguish $(f(X_n), h(X_n))$ from $(f(X_n), R_{l(n)})$. Hence, for logarithmically shrinking h's, the function h is a hard-core of a function f if and only if the exclusive-or of any non-empty subset of the bits of h cannot be approximated from the value of f.

Proof: All that is required is to evaluate the success probability of algorithm G. We start by fixing an $x \in \{0,1\}^n$ and evaluating $\mathsf{Pr}(G(f(x), I_l, l) = \oplus_{i \in I_l}(h_i(x))$, where I_l is a uniformly chosen non-empty subset of $\{1, ..., l\}$ and $l \stackrel{\rm def}{=} l(n)$. Let B denote the set of all non-empty subsets of $\{1, ..., l\}$. Define, for every $S \in B$, a relation $\equiv_S$ so that $y \equiv_S z$ if and only if $\oplus_{i \in S} y_i = \oplus_{i \in S} z_i$, where $y = y_1 \cdots y_l$ and $z = z_1 \cdots z_l$. By the definition of G, it follows that on input $(f(x), S, l)$ and random choice $r \in \{0,1\}^l$, algorithm G outputs $\oplus_{i \in S}(h_i(x))$ if and only if either "$D(f(x), r) = 1$ and $r \equiv_S h(x)$" or "$D(f(x), r) = 0$ and $r \not\equiv_S h(x)$". By elementary manipulations, we get

$$
\begin{aligned}
s(x) &\stackrel{\rm def}{=} \mathsf{Pr}(G(f(x), I_l, l) = \oplus_{i \in I_l}(h_i(x))) \\
&= \sum_{S \in B} \frac{1}{|B|} \mathsf{Pr}(G(f(x), S, l) = \oplus_{i \in S}(h_i(x)) \\
&= \sum_{S \in B} \frac{1}{2 \cdot |B|} (\mathsf{Pr}(D(f(x), R_l) = 1 \mid R_l \equiv_S h(x)) + \mathsf{Pr}(D(f(x), R_l) = 0 \mid R_l \not\equiv_S h(x))) \\
&= \frac{1}{2} + \frac{1}{2|B|} \sum_{S \in B} (\mathsf{Pr}(D(f(x), R_l) = 1 \mid R_l \equiv_S h(x)) - \mathsf{Pr}(D(f(x), R_l) = 1 \mid R_l \not\equiv_S h(x))) \\
&= \frac{1}{2} + \frac{1}{2|B|} \cdot \frac{1}{2^{l-1}} \cdot \left(\sum_{S \in B} \sum_{r \equiv_S h(x)} \mathsf{Pr}(D(f(x), r) = 1) - \sum_{S \in B} \sum_{r \not\equiv_S h(x)} \mathsf{Pr}(D(f(x), x) = 1) \right) \\
&= \frac{1}{2} + \frac{1}{2^l \cdot |B|} \cdot \left(\sum_r \sum_{S \in E(r, h(x))} \mathsf{Pr}(D(f(x), r) = 1) - \sum_r \sum_{S \in N(r, h(x))} \mathsf{Pr}(D(f(x), r) = 1) \right)
\end{aligned}
$$

where $E(r, z) \stackrel{\rm def}{=} \{S \in B : r \equiv_S z\}$ and $N(r, z) \stackrel{\rm def}{=} \{S \in B : r \not\equiv_S z\}$. Observe that for every $r \neq z$ it holds that $|N(r, z)| = 2^{l-1}$ (and $|E(r, z)| = 2^{l-1} - 1$). On the other hand, $E(z, z) = B$ (and $N(z, z) = \emptyset$). Hence, we get

$$
\begin{aligned}
s(x) &= \frac{1}{2} + \frac{1}{2^l |B|} \sum_{r \neq h(x)} ((2^{l-1} - 1) \cdot \mathsf{Pr}(D(f(x), r) = 1) - 2^{n-1} \cdot \mathsf{Pr}(D(f(x), r) = 1)) \\
&\quad + \frac{1}{2^l |B|} \cdot |B| \cdot \mathsf{Pr}(D(f(x), h(x)) = 1)
\end{aligned}
$$

$$= \frac{1}{2} + \frac{1}{|B|} \cdot (\mathsf{Pr}(D(f(x), h(x)) = 1) - \mathsf{Pr}(D(f(x), R_n) = 1))$$

Thus

$$\mathsf{E}(s(X_n)) = \frac{1}{2} + \frac{1}{|B|} \cdot (\mathsf{Pr}(D(f(X_n), h(X_n)) = 1) - \mathsf{Pr}(D(f(X_n), R_n) = 1))$$

and the lemma follows. ■

2.6 * Efficient Amplification of One-way Functions

The *amplification* of weak one-way functions into strong ones, presented in Theorem 2.3.2, has no practical value. Recall that this amplification transforms a function f which is hard to invert on a non-negligible fraction (i.e., $\frac{1}{p(n)}$) of the strings of length n into a function g which is hard to invert on all but a negligible fraction of the strings of length $n^2p(n)$. Specifically, it is shown that an algorithm running in time $T(n)$ which inverts g on a $\epsilon(n)$ fraction of the strings of length $n^2p(n)$ yields an algorithm running in time $\text{poly}(p(n), n, \frac{1}{\epsilon(n)}) \cdot T(n)$ which inverts f on a $1 - \frac{1}{p(n)}$ fraction of the strings of length n. Hence, if f is "hard to invert in practice on a $\frac{1}{1000}$ fraction of the strings of length 100" then all we can say is that g is "hard to invert in practice on a $\frac{999}{1000}$ fraction of the strings of length 1,000,000". In contrast, an efficient amplification of one-way functions, as given below, should relate the difficulty of inverting the (weak one-way) function f on strings of length n to the difficulty of inverting the (strong one-way) function g on the strings of length $O(n)$ (rather than relating it to the to the difficulty of inverting the function g on the strings of length $\text{poly}(n)$). The following definition is natural for a general discussion of amplification of one-way functions.

Definition 2.6.1 (quantitative one-wayness): *Let $T:\mathbb{N}\mapsto\mathbb{N}$ and $\epsilon:\mathbb{N}\mapsto\mathbb{R}$ be polynomial-time computable functions. A polynomial-time computable function $f:\{0,1\}^*\mapsto\{0,1\}^*$ is called $\epsilon(\cdot)$-*one-way with respect to time $T(\cdot)$ *if for every algorithm, A', with running-time bounded by $T(\cdot)$ and all sufficiently large n's*

$$\mathsf{Pr}\left(A'(f(U_n)) \not\in f^{-1}f(U_n)\right) > \epsilon(n)$$

Using this terminology we review what we know already about amplification of one-way functions. A function f is weakly one-way if there exists a polynomial $p(\cdot)$ so that f is $\frac{1}{p(\cdot)}$-one-way with respect to polynomial time. A function f is strongly one-way if, for every polynomial $p(\cdot)$, the f is $(1 - \frac{1}{p(\cdot)})$-one-way with respect to polynomial time. The amplification result of Theorem 2.3.2 can be generalized and restated as follows. If there exist a polynomial-time computable function f which is $\frac{1}{\text{poly}(\cdot)}$-one-way with respect to time $T(\cdot)$ then there exist a polynomial-time computable function g which is $(1 - \frac{1}{\text{poly}(\cdot)})$-one-way with respect to time $T'(\cdot)$, where $T'(\text{poly}(n)) = T(n)$ (i.e., in other words, $T'(n) = T(n^\epsilon)$ for some $\epsilon > 0$). In contrast, an efficient amplification of one-way functions, as given below, should state that the above should hold with respect to $T'(O(n)) = T(n)$ (i.e., in other words, $T'(n) = T(\epsilon \cdot n)$ for some $\epsilon > 0$). Such a result can be obtained for *regular* one-way functions. A function f is called *regular* if there exists a polynomial-time computable function $m : \mathbb{N} \mapsto \mathbb{N}$ and a polynomial $p(\cdot)$ so that, for every y in the range of f, the number of preimages (of length n) of y under f, is between $\frac{m(n)}{p(n)}$ and $m(n) \cdot p(n)$. In this book we only review the result for one-way permutations (i.e., length preserving 1-1 functions).

Theorem 2.6.2 (Efficient amplification of one-way permutations): *Let $p(\cdot)$ be a polynomial and $T : \mathbb{N} \mapsto \mathbb{N}$ be a polynomial-time computable function. Suppose that f is a polynomial-time computable permutation which is $\frac{1}{p(\cdot)}$-one-way with respect to time $T(\cdot)$. Then, there exists a polynomial-time computable permutation F so that, for every polynomial-time computable function $\epsilon : \mathbb{N} \mapsto [0, 1]$, the function F is $(1 - \epsilon(\cdot))$-one-way with respect to time $T'_\epsilon(\cdot)$, where $T'_\epsilon(O(n)) \stackrel{\text{def}}{=} \frac{\epsilon(n)^2}{\text{poly}(n)} \cdot T(n)$.*
The constants, in the O-notation and in the poly-notation, depend on the polynomial $p(\cdot)$.

The key to the amplification of a one-way permutation f is to apply f on many different arguments. In the proof of Theorem 2.3.2, f is applied to unrelated arguments (which are disjoint parts of the input). This makes the proof relatively easy, but also makes the construction very inefficient. Instead, in the construction presented in the proof of the current theorem, we apply the one-way permutation f on related arguments. The first idea which comes to mind is to apply f iteratively many times, each time on the value resulting from the previous application. This will not help if easy instances for the inverting algorithm keep being mapped, by f, to themselves. We cannot just hope that this will not happen. The idea is to use randomization between successive applications. It is important that we use only a small amount of randomization,, since the “randomization” will be encoded into the argument of the constructed function. The randomization, between successive applications of f, takes the form of a random step on an expander graph. Hence a few words about these graphs and random walks on them are in place.

A graph $G = (V, E)$ is called an *(n, d, c)-expander* if it has n vertices (i.e., $|V| = n$), every vertex in V has degree d (i.e., G is d-regular), and G has the following *expansion property* (with *expansion factor* $c > 0$): for every subset $S \subset V$ if $|S| \leq \frac{n}{2}$ then $|N(S)| \geq c \cdot |S|$, where $N(S)$ denotes the vertices in $V - S$ which have neighbour in S (i.e., $N(S) \stackrel{\text{def}}{=} \{u \in V - S : \exists v \in S \text{ s.t. } (u, v) \in E\}$). By *explicitly constructed expanders* we mean a family of graphs $\{G_n\}_{n \in \mathbb{N}}$ so that G_n is a $(2^{2n}, d, c)$ expander (d and c are the same for all graphs in the family) having a polynomial-time algorithm that on input a description of a vertex in an expander outputs its adjacency list (vertices in G_n are represented by binary strings of length $2n$). Such expender families do exist. By a *random walk* on a graph we mean the sequence of vertices visited by starting at a uniformly chosen vertex and randomly selecting at each step one of the neighbouring vertices of the current vertex, with uniform probability distribution. The expanding property implies (via a non-trivial proof) that the vertices along random walks on an expander have surprisingly strong "random properties". In particular, for every l, the probability that vertices along an $O(l)$-step long random walk hit a subset, S, is approximately the same as the probability that at least one of l independently chosen vertices hits S.

We remind the reader that we are interested in successively applying the permutation f, while interleaving randomization steps between successive applications. Hence, before applying permutation f, to the result of the previous application, we take one random step on an expender. Namely, we associate the domain of the given one-way permutation with the vertex set of the expander. Our construction alternatively applies the given one-way permutation, f, and randomly moves from the vertex just reached to one of its neighbours. A key observation is that the composition of an expander with any permutation on its vertices yields an expander (with the same expansion properties). Combining the properties of random walks on expanders and a "reducibility" argument, the construction is showed to amplify the one-wayness of the given permutation in an efficient manner.

Construction 2.6.3 *Let $\{G_n\}_{n\in\mathbb{N}}$ be a family of d-regular graphs, so that G_n has vertex set $\{0,1\}^n$ and self-loops at every vertex. Consider a labeling of the edges incident to each vertex (using the labels $1,2,...,d$). Define $g_l(x)$ be the vertex reachable from vertex x by following the edge labeled l. Let $f:\{0,1\}^*\mapsto\{0,1\}^*$ be a 1-1 length preserving function. For every $k\geq 0$, $x\in\{0,1\}^n$, and $\sigma_1,\sigma_2,...,\sigma_k\in\{1,2,...,d\}$, define*

$$F(x,\sigma_1\sigma_2...\sigma_k) = \sigma_1, F(g_{\sigma_1}(f(x)),\sigma_2,...,\sigma_k)$$

(with $F(x,\lambda)=x$). For every $k:\mathbb{N}\mapsto\mathbb{N}$, define $F_{k(\cdot)}(\alpha)\stackrel{\rm def}{=}F(x,\sigma_1,..,\sigma_t)$, where $t=k(|x|)$ and $\sigma_i\in\{1,2,...,d\}$.

Proposition 2.6.4 *Let $\{G_n\}$, f, $k:\mathbb{N}\mapsto\mathbb{N}$, and $F_{k(\cdot)}$ be as in Construction 2.6.3 (above), and suppose that $\{G_n\}_{n\in\mathbb{N}}$ is an explicitly constructed family of d-regular expander graphs, and f is polynomial-time computable. Suppose that $\alpha:\mathbb{N}\mapsto\mathbb{R}$ and $T:\mathbb{N}\mapsto\mathbb{N}$ are polynomial-time computable, and f is $\alpha(\cdot)$-one-way with respect to time $T:\mathbb{N}\to\mathbb{N}$. Then, for every polynomial-time computable $\varepsilon:\mathbb{N}\mapsto\mathbb{R}$, the function $F_{k(\cdot)}$ is polynomial-time computable as well as $(1-\varepsilon(\cdot))\beta(\cdot)$-one-way with respect to time $T':\mathbb{N}\to\mathbb{N}$, where $\beta(n)\stackrel{\rm def}{=}(1-(1-\alpha(n))^{k(n)/2})$ and $T'(n+k(n)\cdot\log_2 d)\stackrel{\rm def}{=}\frac{\varepsilon(n)^2\alpha(n)}{k(n)\cdot n}\cdot T(n)$.*

Theorem 2.6.2 follows by applying the proposition $\delta+1$ times, where δ is the degree of the polynomial $p(\cdot)$ (specified in the hypothesis that f is $\frac{1}{p(\cdot)}$-one-way). In all applications of the proposition we use $k(n)\stackrel{\rm def}{=}3n$. In the first δ applications we use any $\varepsilon(n)<\frac{1}{7}$. The function resulting from the $i^{\rm th}$ application of the proposition, for $i\leq\delta$, is $\frac{1}{2n^{\delta-i}}$-one-way. In particular, after δ applications, the resulting function is $\frac{1}{2}$-one-way. (It seems that the notion of $\frac{1}{2}$-one-wayness is worthy of special attention, and deserves a name as *mostly one-way*.) In the last (i.e., $\delta+1^{\rm st}$) application we use $\varepsilon(n)=\epsilon(n)$. The function resulting of the last (i.e., $\delta+1^{\rm st}$) application of the proposition satisfies the statement of Theorem 2.6.2.

The proposition itself is proven as follows. First, we use the fact that f is a permutation to show, that the graph $G_f=(V,E_f)$, obtained from $G=(V,E)$ by letting $E_f \stackrel{\rm def}{=} \{(u,f(v)) : (u,v)\in E\}$, has the same expansion property as the graph G. Next, we use the known relation between the expansion constant of a graph and the ratio of the two largest eigenvalues of its adjacency matrix to prove that with appropriate choice of the family $\{G_n\}$ we can have this ratio bounded below by $\frac{1}{\sqrt{2}}$. Finally, we combine the following two Lemmata.

Lemma 2.6.5 (Random Walk Lemma): *Let G be a d-regular graph having a normalized (by factor $\frac{1}{d}$) adjacency matrix for which the ratio of the first and second eigenvalues is smaller than $\frac{1}{\sqrt{2}}$. Let $\mu \le 1/2$ and S be a subset of measure μ of the expender's nodes. Then a random walk of length $2k$ on the expander hits S with probability at least $1-(1-\mu)^k$.*

The proof of the Random Walk Lemma regards probability distributions oven the expander vertex-set as linear combinations of the eigenvectors of the adjacency matrix. It can be shown that the largest eigenvalue is 1, and the eigenvector associated to it is the uniform distribution. Going step by step, we bound from above the probability mass assigned to random walks which do not pass through the set S. At each step, the component of the current distribution, which is in the direction of the first eigenvector, losses a factor μ of its weight (this represents the fraction of the paths which enter S in the current step). The problem is that we cannot make a similar statement with respect to the other components. Yet, using the bound on the second eigenvalue, it can be shown that in each step these components are "pushed" towards the direction of the first eigenvector. The details, being of little relevance to the topic of the book, are omitted.

Lemma 2.6.6 (Reducibility Lemma): *Let $\alpha,\beta:\mathbb{N}\mapsto[0,1]$, and $G_{f,n}$ be a d-regular graph on 2^n vertices satisfying the following* random path property*: for every measure $\alpha(n)$ subset, S, of $G_{f,n}$'s nodes, at least a fraction $\beta(n+k(n)\cdot\log_2 d)$ of the paths of length $k(n)$ passes through a node in S (typically $\beta(n+k(n)\log_2 d)>\alpha(n)$). Suppose that f is $(\alpha(\cdot)+\exp(\cdot))$-one-way with respect to time $T(\cdot)$. Then, for every polynomial-time computable $\varepsilon:\mathbb{N}\mapsto\mathbb{R}$, the function $F_{k(\cdot)}$, defined above, is $(1-\varepsilon(\cdot))\beta(\cdot)$-one-way with respect to time $T':\mathbb{N}\to\mathbb{N}$, where $\beta(n+k(n)\log_2 d)\stackrel{\rm def}{=}(1-(1-\alpha(n))^{k(n)/2})$ and $T'(n+k(n)\log_2 d)\stackrel{\rm def}{=}\frac{\varepsilon(n)^2\alpha(n)}{k(n)n}\cdot T(n)$.*

Proof Sketch: The proof is by a "reducibility argument". Assume for contradiction that $F_{k(\cdot)}$ defined as above can be inverted in time $T'(\cdot)$ with probability at least $1-(1-\varepsilon(m))\cdot\beta(m)$ on inputs of length $m\stackrel{\rm def}{=}n+k(n)\log_2 d$. Amplify A to invert $F_{k(\cdot)}$ with overwhelming probability on a $1-\beta(m)$ fraction of the inputs of length m (originally A inverts each such point with probability $>\varepsilon(m)$, as we can ignore inputs inverted with probability smaller than $\varepsilon(m)$). Note that inputs to A correspond to $k(n)$-long paths on the graph G_n. Consider the set, denoted B_n, of paths (x,p) such that A inverts $F_{k(n)}(x,p)$ with overwhelming probability.

In the sequel, we use the shorthands $k \stackrel{\rm def}{=} k(n)$, $m \stackrel{\rm def}{=} n + k\log_2 d$, $\varepsilon \stackrel{\rm def}{=} \varepsilon(m)$, $\beta \stackrel{\rm def}{=} \beta(m)$, $\alpha \stackrel{\rm def}{=} \alpha(n)$, and $B \stackrel{\rm def}{=} B_n$. Let P_v be the set of all k-long paths which pass through v, and B_v be the subset of B containing paths which pass through v (i.e., $B_v = B \cap P_v$). Define v as *good* if $|B_v|/|P_v| \geq \varepsilon\beta/k$ (and *bad* otherwise). Intuitively, a vertex v is called good if at least a $\varepsilon\beta/k$ fraction of the paths going through v can be inverted by A. Let $B' = B - \cup_{v \text{ bad}} B_v$; namely B' contain all "invertible" paths which pass solely through good nodes. Clearly,
Claim 2.6.6.1: The measure of B' in the set of all paths is greater than $1-\beta$.
Proof: Denote by $\mu(S)$ the measure of the set S in the set of all paths. Then

$$\begin{aligned}
\mu(B') &= \mu(B) - \mu(\cup_{v \text{ bad}} B_v) \\
&\geq 1 - (1-\epsilon)\beta - \sum_{v \text{ bad}} \mu(B_v) \\
&> 1 - \beta + \epsilon\beta - \sum_v (\epsilon\beta/k)\mu(P_v) \\
&> 1 - \beta \quad \Box
\end{aligned}$$

Using the random path property, we have
Claim 2.6.6.2: The measure of good nodes is at least $1-\alpha$.
Proof: Otherwise, let S be the set of bad nodes. If S has measure α then, by the random path property, it follows the fraction of path which pass through vertices of S is at least β. Hence, B', which cannot contain such paths can contain only a $1-\beta$ fraction of all paths in contradiction to Claim 2.6.6.1. $\Box$

The following algorithm for inverting f, is quite natural. The algorithm uses as subroutine an algorithm, denoted A, for inverting $F_{k(\cdot)}$. Inverting f on y is done by placing y on a random point along a randomly selected path $\overline{p}$, taking a walk from y according to the suffix of $\overline{p}$, and asking A for the preimage of the resulting pair under F_k.
Algorithm for inverting f:
On input y, `repeat` $\frac{kn}{\epsilon\beta}$ `times`:

1. Select randomly $i \in \{1,2,...,k\}$, and $\sigma_1, \sigma_2, ..., \sigma_k \in \{1,2,...,d\}$;
2. Compute $y' = F(g_{\sigma_i}(y), \sigma_{i+1}...\sigma_k)$;
3. Invoke A to get $x' \leftarrow A(\sigma_1\sigma_2, ..., \sigma_k, y')$;
4. Compute $x = F(x', \sigma_1...\sigma_{i-1})$;
5. If $f(x) = y$ then halt and output x.

Analysis of the inverting algorithm (for a good x):
Since x is good, a random path going through it (selected above) corresponds to an "invertible path" with probability at least $\epsilon\beta/k$. If such a path is selected then we obtain

the inverse of $f(x)$ with overwhelming probability. The algorithm for inverting f repeats the process sufficiently many times to guarantee overwhelming probability of selecting an "invertible path".

By Claim 2.6.6.2, the good x's constitute a $1-\alpha$ fraction of all n-bit strings. Hence, the existence of an algorithm inverting $F_{k(\cdot)}$, in time $T'(\cdot)$ with probability at least $1-(1-\varepsilon(\cdot))\beta(\cdot)$, implies the existence of an algorithm inverting f, in time $T(\cdot)$ with probability at least $1-\alpha(\cdot)-\exp(\cdot)$. This constitutes a contradiction to the hypothesis of the lemma, and hence the lemma follows. ■

2.7 Miscellaneous

2.7.1 Historical Notes

The notion of a one-way function originates from the paper of Diffie and Hellman [DH76]. Weak one-way functions were introduced by Yao [Y82]. The RSA function was introduced by Rivest, Shamir and Adleman [RSA78], whereas squaring modulo a composite was introduced and studied by Rabin [R79]. The suggestion for basing one-way functions on the believed intractability of decoding random linear codes is taken from [BMT78,GKL88], and the suggestion to base one-way functions on the subset sum problem is taken from [IN89].

The equivalence of existence of weak and strong one-way functions is implicit in Yao's work [Y82]. The existence of universal one-way functions is stated in Levin's work [L85]. The efficient amplification of one-way functions, presented in Section 2.6, is taken from Goldreich el. al. [GILVZ], which in turn uses ideas originating in [AKS].

> Author's Note: *GILVZ = Goldreich, Impagliazzo, Levin, Venkatesan and Zuckerman (FOCS90); AKS = Ajtai, Komolos and Szemeredi (STOC87).*

The concept of hard-core predicates originates from the work of Blum and Micali [BM82]. That work also proves that a particular predicate constitutes a hard-core for the "DLP function" (i.e., exponentiation in a finite field), provided that this function is one-way. Consequently, Yao proved that the existence of one-way functions implies the existence of hard-core predicates [Y82]. However, Yao's construction, which is analogous to the contraction used for the proof of Theorem 2.3.2, is of little practical value. The fact that the inner-product mod 2 is a hard-core for any one-way function (of the form $g(x,r)=(f(x),r)$) was proven by Goldreich and Levin [GL89]. The proof presented in this book, which follows ideas originating in [ACGS84], is due to Charles Rackoff.

Hard-core predicates and functions for specific collections of permutations were suggested in [BM82,LW,K88,ACGS84,VV84]. Specifically, Kalisky [K88], extending ideas of [BM82,LW], proves that the intractability of various discrete logarithm problems yields hard-core functions for the related exponentiation permutations. Alexi el. al. [ACGS84], building on work by Ben-Or et. al. [BCS83], prove that the intractability of factoring yields hard-core functions for permutations induced by squaring modulo a composite number.

2.7.2 Suggestion for Further Reading

Our exposition of the RSA and Rabin functions is quite sparse in details. In particular, the computational problems of generating uniformly distributed "certified primes" and of "primality checking" deserve much more attention. A probabilistic polynomial-time algorithm for generating uniformly distributed primes together with corresponding certificates of primality has been presented by Bach [BachPhd]. The certificate produced, by this algorithm, for a prime P consists of the prime factorization of $P-1$, together with certificates for primality of these factors. This recursive form of certificates for primality originates in von-Pratt's proof that the set of primes is in $\mathcal{NP}$ (cf. [vP]). However, the above procedure is not very practical. Instead, when using the RSA (or Rabin) function in practice, one is likely to prefer an algorithm that generates integers at random and checks them for primality using fast primality checkers such as the algorithms presented in [SSprime,Rprime]. One should note, however, that these algorithms do not produce certificates for primality, and that with some (small) probability may assert that a composite number is a prime. Probabilistic polynomial-time algorithms (yet not practical ones) that, given a prime, produce a certificate for primality, are presented in [GKprime,AHprime]

> Author's Note: *SSprime = Solovay and Strassen, Rprime = Rabin, GKprime = Goldwasser and Kilian, AHprime = Adleman and Haung.*

The subset sum problem is known to be easy in two special cases. One case is the case in which the input sequence is constructed based on a simple "hidden sequence". For example, Merkle and Hellman [MH78], suggested to construct an instance of the subset-sum problem based on a "hidden super increasing sequence" as follows. Let $s_1, ..., s_n, M \stackrel{\rm def}{=} s_{n+1}$ be a sequence satisfying, $s_i > \sum_{j=1}^{i-1} s_j$, for every i, and let w be relatively prime to M. Such a sequence is called *super increasing*. The instance consists of $(x_1, ..., x_n)$ and $\sum_{i\in I} x_i$, for $I \subseteq \{1, ..., n\}$, where $x_i \stackrel{\rm def}{=} w \cdot s_i \bmod M$. It can be shown that knowledge of both w and M allows easy solution of the subset sum problem for the above instance. The hope was that, when w and M are not given, solving the subset-sum problem is hard even for instances generated based on a super increasing sequence (and this would lead to a trapdoor one-way function). However, the hope did not materialize. Shamir presented an efficient algorithm for solving the subset-sum problem for instances with a hidden super increasing sequence [S82]. Another case for which the subset sum problem is known to be easy is the case of *low density* instances. In these instances the length of the elements in binary representation is considerably larger than the number of elements (i.e. $|x_1| = \cdots = |x_n| = (1+\epsilon)n$ for some constant $\epsilon > 0$). For further details consult the original work of Lagarias and Odlyzko [LO85] and the later survey of Brickell and Odlyzko [BO88].

For further details on hard-core functions for the RSA and Rabin functions the reader is directed to Alexi el. al. [ACGS84]. For further details on hard-core functions for the "DLP function" the reader is directed to Kalisky's work [K88].

The theory of average-case complexity, initiated by Levin [L84], is somewhat related to the notion of one-way functions. For a survey of this theory we refer the reader to [BCGL].

Loosely speaking, the difference is that in our context it is required that the (efficient) "generator" of hard (on-the-average) instances can easily solve them himself, whereas in Levin's work the instances are hard (on-the-average) to solve even for the "generator". However, the notion of average-case reducibility introduced by Levin is relevant also in our context.

> Author's Note: *BCGL = Ben-David, Chor, Goldreich and Luby (JCSS, April 1992).*

Readers interested in further details about the best algorithms known for the factoring problem are directed to Pomerance's survey [P82]. Further details on the best algorithms known for the discrete logarithm problem (DLP) can be found in Odlyzko's survey [O84]. In addition, the reader is referred to Bach and Shalit's book on computational number theory [BS92book]. Further details about expander graphs, and random walks on them, can be found in the book of Alon and Spencer [AS91book].

> Author's Note: *Updated versions of the surveys by Pomerance and Odlyzko do exist.*

2.7.3 Open Problems

The efficient amplification of one-way functions, originating in [GILVZ], is only known to work for special types of functions (e.g., regular ones). We believe that presenting (and proving) an efficient amplification of arbitrary one-way functions is a very important open problem. It may also be instrumental for more efficient constructions of pseudorandom generators based on arbitrary one-way functions (see Section 3.5).

An open problem of more practical importance is to try to present hard-core functions with larger range for the RSA and Rabin functions. Specifically, assuming that squaring mod N is one-way, is the function which returns the first half of x a hard-core of squaring mod N? Some support to a positive answer is provided by the work of Shamir and Shrift [SS90]. A positive answer would allow to construct extremely efficient pseudorandom generators and public-key encryption schemes based on the conjectured intractability of the factoring problem.

2.7.4 Exercises

Exercise 1: *Closing the gap between the motivating discussion and the definition of one-way functions*: We say that a function $h:\{0,1\}^* \mapsto \{0,1\}^*$ is *hard on the average but easy with auxiliary input* if there exists a probabilistic polynomial-time algorithm, G, such that

1. There exists a polynomial-time algorithm, A, such that $A(x,y) = h(x)$ for every (x,y) in the range of G (i.e., for every (x,y) so that (x,y) is a possible output of $G(1^n)$ for some input 1^n).

2. for every probabilistic polynomial-time algorithm, A', every polynomial $p(\cdot)$, and all sufficiently large n's

$$\mathsf{Pr}(A'(X_n) = h(X_n)) < \frac{1}{p(n)}$$

where $(X_n, Y_n) \stackrel{\text{def}}{=} G(1^n)$ is a random variable assigned the output of G.

Prove that if there exist "hard on the average but easy with auxiliary input" functions then one-way functions exist.

Exercise 2: *One-way functions and the $\mathcal{P}$ vs. $\mathcal{NP}$ question (part 1):* Prove that the existence of one-way functions implies $\mathcal{P} \neq \mathcal{NP}$.

Guideline: For every function f define $L_f \in \mathcal{NP}$ so that if $L_f \in \mathcal{P}$ then there exists a polynomial-time algorithm for inverting f.

Exercise 3: *One-way functions and the $\mathcal{P}$ vs. $\mathcal{NP}$ question (part 2):* Assuming that $\mathcal{P} \neq \mathcal{NP}$, construct a function f so that the following three claims hold:

1. f is polynomial-time computable;
2. there is no polynomial-time algorithm that always inverts f (i.e., successfully inverts f on every y in the range of f); and
3. f is not (even weakly) one-way. Furthermore, there exists a polynomial-time algorithm which inverts f with exponentially small failure probability, where the probability space is (again) of all possible choices of input (i.e., $f(x)$) and internal coin tosses for the algorithm.

Guideline: Consider the function f_{sat} defined so that $f_{\text{sat}}(\phi, \tau) = (\phi, 1)$ if τ is a satisfying assignment to propositional formulae ϕ, and $f_{\text{sat}}(\phi, \tau) = (\phi, 0)$ otherwise. Modify this function so that it is easy to invert on most instances, yet inverting f_{sat} is reducible to inverting its modification.

Exercise 4: Let f be a strongly one-way function. Prove that for every probabilistic polynomial-time algorithm A, and for every polynomial $p(\cdot)$ the set

$$B_{A,p} \stackrel{\text{def}}{=} \{x : \mathsf{Pr}(A(f(x)) \in f^{-1}f(x)) \geq \frac{1}{p(|x|)}\}$$

has negligible density in the set of all strings (i.e., for every polynomial $q(\cdot)$ and all sufficiently large n it holds that $\frac{|B \cap \{0,1\}^n|}{2^n} < \frac{1}{q(n)}$).

Exercise 5: *Another definition of non-uniformly one-way functions*: Consider the definition resulting from Definition 2.2.6 by allowing the circuits to be probabilistic (i.e., have an auxiliary input which is uniformly selected). Prove that the resulting new definition is equivalent to the original one.

Exercise 6: Define $f_{\rm add} : \{0,1\}^* \mapsto \{0,1\}^*$ so that $f_{\rm add}(xy) = {\rm prime}(x) + {\rm prime}(y)$, where $|x| = |y|$ and ${\rm prime}(z)$ is the smallest prime which is larger than z. Prove that $f_{\rm add}$ is not a one-way function.

> **Guideline:** Don't try to capitalize on the possibility that ${\rm prime}(N)$ is too large, e.g., larger than $N + {\rm poly}(\log N)$. It is unlikely that such a result, in number theory, can be proven. Furthermore, it is generally believed that there exists a constant c such that, for all integer $N \geq 2$, it holds that ${\rm prime}(N) < N + \log_2^c N$. Hence, it is likely that $f_{\rm add}$ is polynomial-time computable.

Exercise 7: (Suggested by Bao Feng): Refute the following conjecture.

> For every (length preserving) one-way function f, the function $f'(x) \stackrel{\rm def}{=} f(x) \oplus x$ is one-way too.

> **Guideline:** Let g be a (length preserving) one-way function, and consider f defined on pairs of strings of the same length so that $f(y,z) \stackrel{\rm def}{=} (g(y) \oplus z, z)$.

Exercise 8: Prove that *one-way functions cannot have a polynomial-size range.* Namely, prove that if f is (even weakly) one-way then for every polynomial $p(\cdot)$ and all sufficiently large n's it holds $|\{f(x) : x \in \{0,1\}^n\}| > p(n)$.

Exercise 9: Prove that *one-way functions cannot have polynomially bounded cycles.* Namely, for every function f define ${\rm cyc}_f(x)$ to be the smallest positive integer i such that applying f for i times on x yields x. Prove that if f is (even weakly) one-way then for every polynomial $p(\cdot)$ and all sufficiently large n's it holds $\mathsf{E}({\rm cyc}_f(U_n)) > p(n)$, where U_n is a random variable uniformly distributed over $\{0,1\}^n$.

Exercise 10: *on the improbability of strengening Theorem 2.3.2 (part 1)*: Suppose that the definition of weak one-way function is further weakened so that it is required that every probabilistic polynomial-time algorithm fails to inverts the function with non-negligible probability. That is, the order of quantifiers in Definition 2.2.2 is reversed (we now have "for every algorithm there exists a polynomial" rather than "there exists a polynomial so that for every algorithm"). Demonstrate the difficulty of extending the proof of Theorem 2.3.2 to this case.

> **Guideline:** Suppose that there exists a family of algorithms, one per each polynomial $t(\cdot)$, so that an algorithm with time bound $t(n)$ fails to invert the function with probability $1/t(n)$. Demonstrate the plausibility of such a family.

Exercise 11: *on the improbability of strengening Theorem 2.3.2 (part 2)* (due to S. Rudich): Suppose that the definition of a strong one-way function is further strengthen so that it is required that every probabilistic polynomial-time algorithm fails to inverts the function with some *specified* negligible probability (e.g., $2^{-\sqrt{n}}$). Demonstrate the difficulty of extending the proof of Theorem 2.3.2 to this case.

Guideline: Suppose that that we construct the strong one-way function g as in the original proof. Note that you *can* prove that any algorithm that works separately on each block of the function g, can invert it only with exponentially low probability. However, there may be an inverting algorithm, A, that inverts the function g with probability ϵ. Show that any inverting algorithm for the weakly one-way function f that uses algorithm A as a black-box "must" invoke it at least $\frac{1}{\epsilon}$ times.

Exercise 12: *collections of one-way functions and one-way functions*: Represent a collection of one-way functions, (I, D, F), as a single one-way function. Given a one-way function f, represent it as a collection of one-way functions.
(Remark: the second direction is quite trivial.)

Exercise 13: *a convention for collections of one-way functions*: Show that without loss of generality, algorithms I and D of a collection (of one-way functions) can be modified so that each of them uses a number of coins which exactly equals the input length. (Hint: Apply padding first on 1^n, next on the coin tosses and output of I, and finally to the coin tosses of D.)

Exercise 14: *justification for a convention concerning one-way collections*: Show that giving the index of the function to the inverting algorithm is essential for a meaningful definition of a collection of one-way functions. (Hint: Consider a collection $\{f_i : \{0,1\}^{|i|} \mapsto \{0,1\}^{|i|}\}$ where $f_i(x) = x \oplus i$.)

Exercise 15: *Rabin's collection and factoring*: Show that the Rabin collection is one-way if and only if factoring integers which are the product of two primes of equal binary expsansion is intractable in a strong sense (i.e., every efficient algorithm succeeds with negligible probability).

Guideline: For one direction use the Chinese Reminder Theorem and an efficient algorithm for extracting square roots modulo a prime. For the other direction observe that an algorithm for extracting square roots modulo a composite N can be use to get two integers x and y such that $x^2 \equiv y^2 \bmod N$ and yet $x \not\equiv \pm y \bmod N$. Also, note that such a pair, (x, y), yields a split of N (i.e., two integers $a, b \neq 1$ such that $N = a \cdot b$).

Exercise 16: *clawfree collections imply one-way functions*: Let (I, D, F) be a clawfree collection of functions (see Subsection 2.4.5). Prove that, for every $\sigma\{0,1\}$, the triplet (I, D, F_σ), where $F_\sigma(i,x) \stackrel{\text{def}}{=} F(\sigma, i, x)$, is a collection of strong one-way functions. Repeat the exercise when replacing the word 'functions' by 'permutations'. (I, D, F) be a clawfree collection of functions

Exercise 17: *more on the inadequacy of graph isomorphism as a basis for one-way functions*: Consider another suggestion to base one-way functions on the conjectured difficulty of the Graph Isomorphism problem. This time we present a collection of functions, defined by the algorithmic triplet $(I_{\rm GI}, D_{\rm GI}, F_{\rm GI})$. On input 1^n, algorithm $I_{\rm GI}$ selects uniformly a $d(n)$-regular graph on n vertices (i.e., each of the n vertices in the graph has degree $d(n)$). On input a graph on n vertices, algorithm $D_{\rm GI}$ randomly selects a permutation in the symmetric group of n elements (i.e., the set of permutations of n elements). On input a (n-vertex) graph G and a (n-element) permutation π, algorithm $F_{\rm GI}$ returns $f_G(\pi) \stackrel{\rm def}{=} \pi G$.

1. Present a polynomial-time implementation of $I_{\rm GI}$.
2. In light of the known algorithms for the Graph Isomorphism problem, which values of $d(n)$ should be definitely avoided?
3. Using a known algorithm, prove that the above collection does not have a one-way property, no matter which function $d(\cdot)$ one uses.

(A search into the relevant literature is indeed required for items (2) and (3).)

Exercise 18: Assuming the existence of one-way functions, prove that there exist a one-way function f so that *no* single bit of the preimage constitutes a hard-core predicate.

> **Guideline:** Given a one-way function f construct a function g so that $g(x,I,J) \stackrel{\rm def}{=} (f(x_{I\cap J}), x_{\overline{I}\cup\overline{J}}, I, J)$, where $I,J \subseteq \{1,2,...|x|\}$, and x_S denotes the string resulting by taking only the bits of x with positions in the set S (i.e., $x_{i_1,...,i_s} \stackrel{\rm def}{=} x_{i_1}\cdots x_{i_s}$, where $x = x_1\cdots x_{|x|}$).

Exercise 19: *hard-core predicate for a 1-1 function implies that the function is one-way*: Let f be a 1-1 function (you may assume for simplicity that it is length preserving) and let b be a hard-core for f.

1. Prove that if f is polynomial-time computable then it is strongly one-way.
2. Prove that (regardless of whether f is polynomial-time computable or not) f must be weakly one-way. Furthermore, for every $\delta > \frac{1}{2}$, the function f cannot be inverted on a δ fraction of the instances.

Exercise 20: *An unbiased hard-core predicate* (suggested by Erez Petrank): Assuming the existence of one-way functions, prove the existence of hardcore predicates which are unbiased (i.e., the predicate b satisfies $\mathsf{Pr}(b(U_n)=1) = \frac{1}{2}$).

> **Guideline:** Slightly modify the predicate defined in Theorem 2.5.2.

Exercise 21: In continuation to the proof of Theorem 2.5.2, we present guidelines for a more efficient inverting algorithm. In the sequel it will be more convenient to use arithmetic of reals instead of that of Boolean. Hence, we denote $b'(x,r) = (-1)^{b(r,x)}$ and $G'(y,r) = (-1)^{G(y,r)}$.

1. Prove that for every x it holds that $\mathsf{E}(b'(x,r)\cdot G'(f(x), r+e^i)) = s'(x)\cdot(-1)^{x_i}$, where $s'(x) \stackrel{\rm def}{=} 2\cdot(s(x)-\frac{1}{2})$.

2. Let v be an l-dimensional Boolean vector, and let R be a uniformly chosen l-by-n Boolean matrix. Prove that for every $v \neq u \in \{0,1\}^l$ it holds that vR and uR are pairwise independent and uniformly distributed in $\{0,1\}^n$.
3. Prove that $b'(x, vR) = b'(xR^T, v)$, for every $x \in \{0,1\}^n$ and $v \in \{0,1\}^l$.
4. Prove that, with probability at least $\frac{1}{2}$, there exists $\sigma \in \{0,1\}^l$ so that for every $1 \le i \le n$ the sign of $\sum_{v \in \{0,1\}^l} b'(\sigma, v) G'(f(x), vR + e^i))$ equals the sign of $(-1)^{x_i}$. (Hint: $\sigma \stackrel{\rm def}{=} xR^T$.)
5. Let B be an 2^l-by-2^l matrix with the (σ, v)-entry being $b'(\sigma, v)$, and let $\overline{g}^i$ be an 2^l-dimensional vector with the $v^{\rm th}$ entry equal $G'(f(x), vR + e^i)$. The inverting algorithm computes $\overline{z}_i \leftarrow Bg^i$, for all i's, and forms a matrix Z in which the columns are the $\overline{z}_i$'s. The output is a row that when applying f to it yields $f(x)$. Evaluate the success probability of the algorithm. Using the special structure of matrix B, show that the product Bg^i can be computed in time $l \cdot 2^l$.
   Hint: B is the Sylvester matrix, which can be written recursively as
   $$S_k = \begin{pmatrix} S_{k-1} & S_{k-1} \\ S_{k-1} & \overline{S_{k-1}} \end{pmatrix}$$
   where $S_0 = +1$ and $\overline{M}$ means flipping the $+1$ entries of M to -1 and vice versa.

Author's Note: *First draft written mainly in Summer of 1991.*