Chapter 10

Cryptography

"From times immemorial, humanity has gotten frequent, often cruel, reminders that many things are easier to do than to reverse."
L. Levin [**?**]

SOMEWHAT ROUGH STILL

The importance of cryptography in today's online world needs no introduction. Here we focus on the complexity issues that underlie this field. The traditional task of cryptography was to allow two parties to *encrypt* their messages so that eavesdroppers gain no information about the message. (See Figure 10.1.) Various encryption techniques have been invented throughout history with one common characteristic: sooner or later they were broken.

Figure unavailable in pdf file.

Figure 10.1: People sending messages over a public channel (e.g., the internet) wish to use encryption so that eavesdroppers learn "nothing."

In the post **NP**-completeness era, a crucial new idea was presented: the code-breaker should be thought of as a resource-bounded computational device. Hence the security of encryption schemes ought to be proved by *reducing* the task of breaking the scheme into the task of solving some computationally intractable problem (say requiring exponential time complexity or circuit size), thus one could hope to design encryption schemes that are efficient enough to be used in practice, but whose breaking will require, say, millions of years of computation time.

Early researchers tried to base the security of encyption methods upon the (presumed) intractability of **NP**-complete problems. This effort has not succeeded to date, seemingly because **NP**-completeness concerns the intractability of problems in the *worst-case* whereas cryptography seems to need problems that are intractable on *most instances*. After all, when we encrypt email, we require that decryption should be difficult for an eavesdropper for all (or *almost all*) messages, not just for a few messages. Thus the concept most useful in this chapter will be *average-case complexity*[1]. We will see a class of functions called *one-way functions* that are easy to compute but hard to invert for most inputs —they are alluded to in Levin's quote above. Such functions exist under a variety of assumptions, including the famous assumption that factoring integers requires time super-polynomial time in the integer's bit-length to solve in the average case (e.g., for a product of two random primes).

Furthermore, in the past two decades, cryptographers have taken on tasks above and beyond the basic task of encryption—from implementing digital cash to maintaining the privacy of individuals in public databases. (We survey some applications in Section 10.4.) Surprisingly, many of these tasks can be achieved using the same computational assumptions used for encryption. A crucial ingredient in these developments turns out to be an answer to the question: "What is a random string and how can we generate one?" The complexity-theoretic answer to this question leads to the notion of a *pseudorandom generator*, which is a central object; see Section 10.2. This notion is very useful in itself and is also a template for several other key definitions in cryptography, including that of encryption (see Section 10.4).

Private key versus public key: Solutions to the encryption problem today come in two distinct flavors. In *private-key cryptography,* one assumes that the two (or more) parties participating in the protocol share a private "key" —namely, a statistically random string of modest size—that is not known to the eavesdropper[2]. In a *public-key encryption system* (a concept introduced by Diffie and Hellman in 1976 [**?**]) we drop this assumption. Instead, a party P picks a pair of keys: an *encryption* key and *decryption* key, both chosen at random from some (correlated) distribution. The encryption key will be used to encrypt messages to P and is considered public —i.e.,

[1]A problem's average-case and worst-case complexities can differ radically. For instance, 3COL is **NP**-complete on general graphs, but on most n-node graphs is solvable in quadratic time or less. A deeper study of average case complexity appears in Chapter 15.

[2]Practically, this could be ensured with a face-to-face meeting that might occur long before the transmission of messages.

published and known to everybody including the eavesdropper. The decryption key is kept secret by P and is used to decrypt messages. A famous public-key encryption scheme is based upon the RSA function of Example 10.4. At the moment we do not know how to base public key encryption on the sole assumption that one-way functions exist and current constructions require the assumption that there exist one-way functions with some special structure (such as RSA, factoring-based, and Lattice-based one way functions). Most topics described in this chapter are traditionally labeled *private key cryptography*.

10.1 Hard-on-average problems and one-way functions

A basic cryptographic primitive is a *one-way function*. Roughly speaking, this is a function f that is easy to compute but hard to invert. Notice that if f is not one-to-one, then the inverse $f^{-1}(x)$ may not be unique. In such cases "inverting" means that given $f(x)$ the algorithm is able to produce some preimage, namely, any element of $f^{-1}(f(x))$. We say that the function is one-way function if inversion is difficult for the "average" (or "many") x. Now we define this formally; a discussion of this definition appears below in Section 10.1.1. A function family (g_n) is a family of functions where g_n takes n-bit inputs. It is *polynomial-time computable* if there is a polynomial-time TM that given an input x computes $g_{|x|}(x)$.

Definition 10.1 (One-way function)
A family of functions $\{f_n : \{0,1\}^n \mapsto \{0,1\}^{m(n)}\}$ is $\epsilon(n)$ *one-way* with *security* $s(n)$ if it is polynomial-time computable and furthermore for every algorithm A that runs in time $s(n)$,

$$\mathbf{Pr}_{x\in\{0,1\}^n}[A \text{ inverts } f_n(x)] \leq \epsilon(n). \tag{1}$$

Now we give a few examples and discuss the evidence that they are hard to invert "on average inputs."

Example 10.2
The first example is motivated by the fact that finding the prime factors of a given integer is the famous FACTORING problem, for which the best current algorithm has running time about $2^{O(n^{1/3})}$ (and even that bounds relies on the truth of some unproven conjectures in number theory). The

hardest inputs for current algorithms appear to be of the type $x \cdot y$, where x, y are random primes of roughly equal size.

Here is a first attempt to define a one-way function using this observation. Let $\{f_n\}$ be a family of functions where $f_n : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^{2n}$ is defined as $f_n([x]_2, [y]_2) = [x \cdot y]_2$. If x and y are primes —which by the Prime Number Theorem happens with probability $\Theta(1/n^2)$ when x, y are random n-bit integers— then f_n seems hard to invert. It is widely believed that there are $c > 1, f > 0$ such that family f_n is $(1 - 1/n^c)$-one-way with security parameter 2^{n^f}.

An even harder version of the above function is obtained by using the existence of a randomized polynomial-time algorithm A (which we do not describe) that, given 1^n, generates a random n-bit prime number. Suppose A uses m random bits, where $m = \text{poly}(n)$. Then A may be seen as a (deterministic) mapping from m-bit strings to n-bit primes. Now let function $\tilde{f}_m$ map (r_1, r_2) to $[A(r_1) \cdot A(r_2)]_2$, where $A(r_1), A(r_2)$ are the primes output by A using random strings r_1, r_2 respectively. This function seems hard to invert for almost all r_1, r_2. (Note that any inverse (r_1', r_2') for $\tilde{f}_m(r_1, r_2)$ allows us to factor the integer $A(r_1) \cdot A(r_2)$ since unique factorization implies that the prime pair $A(r_1'), A(r_2')$ must be the same as $A(r_1), A(r_2)$.) It is widely conjecture that there are $c > 1, f > 0$ such that $\tilde{f}_n$ is $1/n^c$-one-way with security parameter 2^{n^f}.

The FACTORING problem, a mainstay of modern cryptography, is of course the inverse of multiplication. Who would have thought that the humble multiplication, taught to children in second grade, could be the source of such power? The next two examples also rely on elementary mathematical operations such as exponentiation, albeit with modular arithmetic.

Example 10.3

Let $p_1, p_2, \ldots$ be a sequence of primes where p_i has i bits. Let g_i be the generator of the group $Z^*_{p_i}$, the set of numbers that are nonzero mod p_i. Then for every $y \in 1, .., p_i - 1$, there is a unique $x \in \{1, .., p - 1\}$ such that

$$g_i^x \equiv y \pmod{p_i}.$$

Then $x \to g_i^x \pmod{p_i}$ is a permutation on $1, .., p_i - 1$ and is conjectured to be one-way. The inversion problem is called the *DISCRETE LOG* problem. We show below using random self-reducibility that if it is hard on worst-case inputs, then it is hard on average.

We list some more conjectured one-way functions.

Example 10.4
RSA function. Let $m = pq$ where p, q are large random primes and e be a random number coprime to $\phi(m) = (p-1)(q-1)$. Let Z_m^* be the set of integers in $[1, \ldots, m]$ coprime to m. Then the function is defined to be $f_{p,q,e}(x) = x^e \pmod{m}$. This function is used in the famous RSA public-key cryptosystem.

Rabin function. For a composite number m, define $f_m(x) = x^2 \pmod{m}$. If we can invert this function on a $1/\text{poly}(\log m)$ fraction of inputs then we can factor m in $\text{poly}(\log m)$ time (see exercises).

Both the RSA and Rabin functions are useful in public-key cryptography. They are examples of *trapdoor* one-way functions: if the factors of m (the "trapdoor" information) are given as well then it is easy to invert the above functions. Trapdoor functions are fascinating objects but will not be studied further here.

Random subset sum. Let $m = 10n$. Let the inputs to f be n positive m-bit integers $a_1, a_2, \ldots, a_n$, and a subset S of $\{1, 2, \ldots, n\}$. Its output is $(a_1, a_2, \ldots, a_n, \sum_{i \in S} a_i)$. Note that f maps $n(m+1)$-bit inputs to $nm + m$ bits.

When the inputs are randomly chosen, this function seems hard to invert. It is conjectured that there is $c > 1, d > 0$ such that this function is $1/n^c$-one-way with security 2^{n^d}.

10.1.1 Discussion of the definition of one-way function

We will always assume that the the one-way function under consideration is such that the security parameter $s(n)$ is superpolynomial, i.e., larger than n^k for every $k > 0$. The functions described earlier are actually believed to be one-way with a larger security parameter 2^{n^ϵ} for some fixed $\epsilon > 0$.

Of greater interest is the error parameter $\epsilon(n)$, since it determines the *fraction* of inputs for which inversion is easy. Clearly, a continuum of values is possible, but two important cases to consider are (i) $\epsilon(n) = (1 - 1/n^c)$ for some fixed $c > 0$, in other words, the function is difficult to invert on at least $1/n^c$ fraction of inputs. Such a function is often called a *weak* one-way function. The simple one-way function f_n of Example 10.2 is conjectured to be of this type. (ii) $\epsilon(n) < 1/n^k$ for every $k > 1$. Such a function is called a *strong* one-way function.

Yao showed that if weak one-way functions exist then so do strong one-way functions. We will prove this surprising theorem (actually, something close to it) in Chapter 18. We will not use it in this chapter, except as a justification for our intuition that strong one-way functions exist. (Another justification is of course the empirical observation that the candidate one-way functions mentioned above do seem appear difficult to invert on most inputs.)

10.1.2 Random self-reducibility

Roughly speaking, a problem is *random-self-reducible* if solving the problem on any input x reduces to solving the problem on a sequence of random inputs $y_1, y_2, \ldots,$ where each y_i is uniformly distributed among all inputs. To put it more intuitively, the worst-case can be reduced to the average case. Hence the problem is either easy on *all* inputs, or hard on *most* inputs. (In other words, we can exclude the possibility that problem is easy on almost all the inputs but not all.) If a function is one-way and also randomly self-reducible then it must be a strong one-way function. This is best illustrated with an example.

THEOREM 10.5
Suppose A is an algorithm with running time $t(n)$ that, given a prime p, a generator g for $\mathbb{Z}_p^*$, and an input $g^x(\mod p)$, manages to find x for δ fraction of $x \in \mathbb{Z}_p^*$. Then there is a randomized algorithm A' with running time $O(\frac{1}{\delta \log 1/\epsilon}(t(n)+poly(n)))$ that solves DISCRETE LOG on every input with probability at least $1-\epsilon$.

PROOF: Suppose we are given $y = g^x(\mod p)$ and we are trying to find x. Repeat the following trial $O(1/(\delta \log 1/\epsilon))$ times: "Randomly pick $r \in \{0, 1, \ldots, p-2\}$ and use A to try to compute the logarithm of $y \cdot g^r (\bmod p)$. Suppose A outputs z. Check if $g^{z-r}(\text{mod}p)$ is y, and if so, output $z - r(\text{mod}(p-1))$ as the answer."

The main observation is that if r is randomly chosen, then $y \cdot g^r(\mod p)$ is randomly distributed in $\mathbb{Z}_p^*$ and hence the hypothesis implies that A has a δ chance of finding its discrete log. After $O(1/(\delta \log 1/\epsilon)$ trials, the probability that A failed every time is at most ϵ. ■

Corollary 10.6
If for any infinite sequence of primes $p_1, p_2, \ldots,$ DISCRETE LOG mod p_i is hard on worst-case $x \in \mathbb{Z}^*_{p_i}$, then it is hard for almost all x.

Later as part of the proof of Theorem 10.14 we give another example of random self-reducibility: linear functions over $GF(2)$.

10.2 What is a random-enough string?

Cryptography often becomes much easier if we have an abundant supply of random bits. Here is an example.

Example 10.7 (One-time pad)
Suppose the message sender and receiver share a long string r of random bits that is not available to eavesdroppers. Then secure communication is easy. To encode message $m \in \{0,1\}^n$, take the first n bits of r, say the string s. Interpret both strings as vectors in $GF(2)^n$ and encrypt m by the vector $m+s$. The receiver decrypts this message by adding s to it (note that $s+s=0$ in $GF(2)^n$). If s is statistically random, then so is $m+s$. Hence the eavesdropper provably cannot obtain even a single bit of information about m *regardless of how much computational power he expends.*

Note that reusing s is a strict no-no (hence the name "one-time pad"). If the sender ever reuses s to encrypt another message m' then the eavesdropper can add the two vectors to obtain $(m+s)+(m'+s) = m+m'$, which is some nontrivial information about the two messages.

Of course, the one-time pad is just a modern version of the old idea of using "codebooks" with a new key prescribed for each day.

One-time pads are conceptually simple, but impractical to use, because the users need to agree in advance on a secret pad that is large enough to be used for all their future communications. It is also hard to generate because sources of quality random bits (e.g., those based upon quantum phenomena) are often too slow. Cryptography's suggested solution to such problems is to use a pseudorandom generator. This is a deterministically computable function $g : \{0,1\}^n \to \{0,1\}^{n^c}$ (for some $c > 1$) such that if $x \in \{0,1\}^n$ is randomly chosen, then $g(x)$ "looks" random. Thus so long as users have been provided a common n-bit random string, they can use the generator to produce n^c "random looking" bits, which can be used to encrypt n^{c-1} messages of length n. (In cryptography this is called a *stream cipher.*)

Clearly, at this point we need an answer to the question posed in the Section's title! Philosophers and statisticians have long struggled with this question.

EXAMPLE 10.8
What is a random-enough string? Here is Kolmogorov's definition: *A string of length n is random if no Turing machine whose description length is $< 0.99n$ (say) outputs this string when started on an empty tape.* This definition is the "right" definition in some philosophical and technical sense (which we will not get into here) but is not very useful in the complexity setting because checking if a string is random according to this definition is undecidable.

Statisticians have also attempted definitions which boil down to checking if the string has the "right number" of patterns that one would expect by the laws of statistics, e.g. the number of times 11100 appears as a substring. (See Knuth Volume 3 for a comprehensive discussion.) It turns out that such definitions are too weak in the cryptographic setting: one can find a distribution that passes these statistical tests but still will be completely insecure if used to generate the pad for the one-time pad encryption scheme.

10.2.1 Blum-Micali and Yao definitions

Now we introduce two complexity-theoretic definitions of pseudorandomness due to Blum-Micali and Yao in the early 1980s. For a string $y \in \{0,1\}^n$ and $S \subseteq [n]$, we let $y|_S$ denote the projection of Y to the coordinates of S. In particular, $y|_{[1..i]}$ denotes the first i bits of y.

The Blum-Micali definition is motivated by the observation that one property (in fact, the defining property) of a statistically random sequence of bits y is that given $y|_{[1..i]}$, we cannot predict y_{i+1} with odds better than 50/50 *regardless of the computational power available to us.* Thus one could define a "pseudorandom" string by considering predictors that have limited computational resources, and to show that they cannot achieve odds much better than 50/50 in predicting y_{i+1} from $y|_{[1..i]}$. Of course, this definition has the shortcoming that any single finite string would be predictable for a trivial reason: it could be hardwired into the program of the predictor Turing machine. To get around this difficulty the Blum-Micali definition (and also Yao's definition below) defines pseudorandomness for distributions of strings rather than for individual strings. Furthermore, the definition concerns an infinite sequence of distributions, one for each input size.

Definition 10.9 (Blum-Micali)
Let $\{g_n\}$ be a polynomial-time computable family of functions, where $g_n : \{0,1\}^n \to \{0,1\}^m$ and $m = m(n) > n$. We say the family is $(\epsilon(n), t(n))$-*unpredictable* if for every probabilistic polynomial-time algorithm A that runs in time $t(n)$ and every large enough input size n,

$$\Pr[A(g(x)_{[1..i]}) = g(x)_{i+1}] \leq \frac{1}{2} + \epsilon(n),$$

where the probability is over the choice of $x \in \{0,1\}^n$, $i \in \{1,\ldots,n\}$, and the randomness used by A.

If for every fixed k, the family $\{g_n\}$ is $(1/n^c, n^k)$-unpredictable for every $c > 1$, then we say in short that it is *unpredictable by polynomial-time algorithms.*

Remark 10.10
Allowing the tester to be an arbitrary polynomial-time machine makes perfect sense in a cryptographic setting where we wish to assume nothing about the adversary except an upperbound on her computational power.

Pseudorandom generators proposed in the pre-complexity era, such as the popular linear or quadtratic congruential generators do not satisfy the Blum-Micali definition because bit-prediction can in fact be done in polynomial time.

Yao gave an alternative definition in which the tester machine is given access to the entire string at once. This definition implicitly sets up a test of randomness analogous to the more famous Turing test for intelligence (see Figure 10.2). The tester machine A is given a string $y \in \{0,1\}^{n^c}$ that is produced in one of two ways: it is either drawn from the uniform distribution on $\{0,1\}^{n^c}$ or generated by taking a random string $x \in \{0,1\}^n$ and stretching it using a deterministic function $g : \{0,1\}^n \to \{0,1\}^{n^c}$. The tester is asked to output "1" if the string looks random to it and 0 otherwise. We say that g is a pseudorandom generator if no polynomial-time tester machine A has a great chance of being able to determine which of the two distributions the string came from.

Definition 10.11 ([**?**])
Let $\{g_n\}$ be a polynomial-time computable family of functions, where $g_n : \{0,1\}^n \to \{0,1\}^m$ and $m = m(n) > n$. We say it is a $(\delta(n), s(n))$-*pseudorandom generator* if for every probabilistic algorithm A running in time $s(n)$ and for all large enough n

$$|\mathbf{Pr}_{y\in\{0,1\}^{n^c}}[A(y) = 1] - \mathbf{Pr}_{x\in\{0,1\}^n}[A(g_n(x)) = 1]| \le \delta(n). \qquad (2)$$

We call $\delta(n)$ the *distinguishing probability* and $s(n)$ the *security parameter*.

If for every $c', k > 1$, the family is $(1/n^{c'}, n^k)$-pseudorandom then we say in short that it is a *pseudorandom generator*.

Figure unavailable in pdf file.

Figure 10.2: Yao's definition: If $c > 1$ then $g : \{0,1\}^n \to \{0,1\}^{n^c}$ is a *pseudorandom generator* if no polynomial-time tester has a good chance of distinguishing between truly random strings of length n^c and strings generated by applying g on random n-bit strings.

10.2.2 Equivalence of the two definitions

Yao showed that the above two definitions are equivalent —up to minor changes in the security parameter, a family is a pseudorandom generator iff it is (bitwise) unpredictable. The *hybrid argument* used in this proof has become a central idea of cryptography and complexity theory.

The nontrivial direction of the equivalence is to show that pseudorandomness of the Blum-Micali type implies pseudorandomness of the Yao type. Not surprisingly, this direction is also more important in a practical sense. Designing pseudorandom generators seems easier for the Blum-Micali definition —as illustrated by the Goldreich-Levin construction below— whereas Yao's definition seems more powerful for applications since it allows the adversary unrestricted access to the pseudorandom string. Thus Yao's theorem provides a bridge between what we can prove and what we need.

Theorem 10.12 (Prediction vs. Indistinguishability [**?**])
Let Let $g_n : \{0,1\}^n \to \{0,1\}^{N(n)}$ be a family of functions where $N(n) = n^k$ for some $k > 1$.
If g_n is $(\frac{\epsilon(n)}{N(n)}, 2t(n))$-unpredictable where $t(n) \ge N(n)^2$ then it is $(\epsilon(n), t(n))$-pseudorandom.
Conversely, if g_n is $(\epsilon(n), t(n))$-pseudorandom, then it is $(\epsilon(n), t(n))$-unpredictable.

PROOF: The converse part is trivial since a bit-prediction algorithm can in particular be used to distinguish $g(x)$ from random strings of the same length. It is left to the reader.

Let N be shorthand for $N(n)$. Suppose g is not $(\epsilon(n), t(n))$-pseudorandom, and A is a distinguishing algorithm that runs in $t(n)$ time and satisfies:

$$\left| \Pr_{x \in B^n}[A(g(x)) = 1] - \Pr_{y \in \{0,1\}^N}[A(y) = 1] \right| > \epsilon(n). \tag{3}$$

By considering either A or the algorithm that is A with the answer flipped, we can assume that the $|\cdot|$ can be removed and in fact

$$\Pr_{x \in B^n}[A(g(x)) = 1] - \Pr_{y \in \{0,1\}^N}[A(y) = 1] > \epsilon(n). \tag{4}$$

Consider B, the following bit-prediction algorithm. Let its input be $g(x)|_{\leq i}$ where $x \in \{0,1\}^n$ and $i \in \{0, \ldots, N-1\}$ are chosen uniformly at random. B's program is: "*Pick bits $u_{i+1}, u_{i+2}, \ldots, u_N$ randomly and run A on the input $g(x)|_{\leq i} u_{i+1} u_{i+2} \ldots u_N$. If A outputs 1, output u_{i+1} else output $\overline{u_{i+1}}$.*" Clearly, B runs in time less than $t(n) + O(N(n)) < 2t(n)$. To complete the proof we show that B predicts $g(x)_{i+1}$ correctly with probability at least $\frac{1}{2} + \frac{\epsilon(n)}{N}$.

Consider a sequence of $N+1$ distributions $\mathcal{D}_0$ through $\mathcal{D}_N$ defined as follows (in all cases, $x \in \{0,1\}^n$ and $u_1, u_2, \ldots, u_N \in \{0,1\}$ are assumed to be chosen randomly)

$$\begin{aligned} \mathcal{D}_0 &= u_1 u_2 u_3 u_4 \cdots u_N \\ \mathcal{D}_1 &= g(x)_1 u_2 u_3 \cdots u_N \\ \vdots &\quad \vdots \\ \mathcal{D}_i &= g(x)_{\leq i} u_{i+1} \cdots u_N \\ \vdots &\quad \vdots \\ \mathcal{D}_N &= g(x)_1 g(x)_2 \cdots g(x)_N \end{aligned}$$

Furthermore, we denote by $\overline{\mathcal{D}_i}$ the distribution obtained from $\mathcal{D}_i$ by flipping the ith bit (i.e., replacing $g(x)_i$ by $\overline{g(x)_i}$). If $\mathcal{D}$ is any of these $2(N+1)$ distributions then we denote $\Pr_{y \in \mathcal{D}}[A(y) = 1]$ by $q(\mathcal{D})$. With this notation we rewrite (4) as

$$q(\mathcal{D}_N) - q(\mathcal{D}_0) > \epsilon(n). \tag{5}$$

Furthermore, in $\mathcal{D}_i$, the $(i+1)$th bit is equally likely to be $g(x)_{i+1}$ and $\overline{g(x)_{i+1}}$, so

$$q(\mathcal{D}_i) = \tfrac{1}{2}(q(\mathcal{D}_{i+1}) + q(\overline{\mathcal{D}_{i+1}})), \tag{6}$$

Now we analyze the probability that B predicts $g(x)_{i+1}$ correctly. Since i is picked randomly we have

$$\Pr_{i,x}[B \text{ is correct}] = \frac{1}{N}\sum_{i=0}^{n-1}\frac{1}{2}\Big(\Pr_{x,\mathbf{u}}[B\text{'s guess for } g(x)_{i+1} \text{ is correct} \mid u_{i+1} = g(x)_{i+1}]$$
$$+ \quad \Pr_{x,\mathbf{u}}[B\text{'s guess for } g(x)_{i+1} \text{ is correct} \mid u_{i+1} = \overline{g(x)_{i+1}}]\Big).$$

Since B's guess is u_{i+1} iff A outputs 1 this is

$$\begin{aligned}
&= \frac{1}{2N}\sum_{i=0}^{N-1}(q(\mathcal{D}_{i+1}) + 1 - q(\overline{\mathcal{D}_{i+1}}))\\
&= \frac{1}{2} + \frac{1}{2N}\sum_{i=0}^{N-1}(q(\mathcal{D}_{i+1}) - q(\overline{\mathcal{D}_{i+1}}))
\end{aligned}$$

From (6), $q(\mathcal{D}_{i+1}) - q(\overline{\mathcal{D}_{i+1}}) = 2(q(\mathcal{D}_{i+1}) - q(\mathcal{D}_i))$, so this becomes

$$\begin{aligned}
&= \frac{1}{2} + \frac{1}{2N}\sum_{i=0}^{N-1}2(q(\mathcal{D}_{i+1}) - q(\mathcal{D}_i))\\
&= \frac{1}{2} + \frac{1}{N}(q(\mathcal{D}_N) - q(\mathcal{D}_0))\\
&> \frac{1}{2} + \frac{\epsilon(n)}{N}.
\end{aligned}$$

This finishes our proof. ■

10.3 One-way functions and pseudorandom number generators

Do pseudorandom generators exist? Surprisingly the answer (though we will not prove it in full generality) is that they do if and only if *one-way functions* exist.

THEOREM 10.13
One-way functions exist iff pseudorandom generators do.

Since we had several plausible candidates for one-way functions in Section 10.1, this result helps us design pseudorandom generators using those candidate one-way functions. If the pseudorandom generators are ever proved to be insecure, then the candidate one-way functions were in fact not one-way, and so we would obtain (among other things) efficient algorithms for FACTORING and DISCRETE LOG.

The "if" direction of Theorem 10.13 is trivial: if g is a pseudorandom generator then it must also be a one-way function since otherwise the algorithm that inverts g would be able to distinguish its outputs from random strings. The "only if" direction is more difficult and involves using a one-way function to explicitly construct a pseudorandom generator. We will do this only for the special case of one-way functions that are *permutations*, namely, they map $\{0,1\}^n$ to $\{0,1\}^n$ in a one-to-one and onto fashion. As a first step, we describe the Goldreich-Levin theorem, which gives an easy way to produce one pseudorandom bit, and then describe how to produce n^c pseudorandom bits.

10.3.1 Goldreich-Levin hardcore bit

Let $\{f_n\}$ be a one-way permutation where $f_n : \{0,1\}^n \to \{0,1\}^n$. Clearly, the function $g : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^{2n}$ defined as $g(x,r) = (f(x),r)$ is also a one-way permutation. Goldreich and Levin showed that given $(f(x),r)$, it is difficult for a polynomial-time algorithm to predict $x \odot r$, the scalar product of x and r (mod 2). Thus even though the string $(f(x),r)$ in principle contains all the information required to extract (x,r), it is computationally difficult to extract even the single bit $x \odot r$. This bit is called a *hardcore* bit for the permutation. Prior to the Goldreich-Levin result we knew of hardcore bits for some specific (conjectured) one-way permutations, not all.

THEOREM 10.14 (GOLDREICH-LEVIN THEOREM)
Suppose that $\{f_n\}$ is a family of $\epsilon(n)$)-one-way permutation with security $s(n)$. Let $S(n) = (\min\left\{s(n), \frac{1}{\epsilon(n)}\right\})^{1/8}$ Then for all algorithms A running in time $S(n)$

$$\mathbf{Pr}_{x,r\in\{0,1\}^n}[A(f_n(x),r) = x \odot r] \le \frac{1}{2} + O(\frac{1}{S(n)}). \tag{7}$$

PROOF: Suppose that some algorithm A can predict $x \odot r$ with probability $1/2 + \delta$ in time $t(n)$. We show how to invert $f_n(x)$ for $O(\delta)$ fraction of the inputs in $O(n^3 t(n)/\delta^4)$ time, from which the theorem follows.

Claim 10.15
Suppose that

$$\mathbf{Pr}_{x,r\in\{0,1\}^n}[A(f_n(x),r) = x \odot r] \geq \frac{1}{2} + \delta. \tag{8}$$

Then for at least δ fraction of x's

$$\mathbf{Pr}_{r\in\{0,1\}^n}[A(f_n(x),r) = x \odot r] \geq \frac{1}{2} + \frac{\delta}{2}. \tag{9}$$

Proof: We use an averaging argument. Suppose that p is the fraction of x's satisfying (9). We have $p \cdot 1 + (1-p)(1/2+\delta/2) \geq 1/2 + \delta$. Solving this with respect to p, we obtain

$$p \geq \frac{\delta}{2(1/2-\delta/2)} \geq \delta.$$

■

We design an inversion algorithm that given $f_n(x)$, where $x \in_R \{0,1\}^n$, will try to recover x. It succeeds with high probability if x is such that (9) holds, in other words, for at least δ fraction of x. Note that the algorithm can always check the correctness of its answer, since it has $f_n(x)$ available to it and it can apply f_n to its answer and see if $f_n(x)$ is obtained.

Warmup: *Reconstruction when the probability in (9) is $\geq 3/4 + \delta$.*

Let P be any program that computes some unknown linear function over $GF(2)^n$ but errs on some inputs. Specifically, there is an unknown vector $x \in GF(2)^n$ such that

$$\Pr_r[P(r) = x \cdot r] = 3/4 + \delta. \tag{10}$$

Then we show to add a simple "correction" procedure to turn P into a probabilistic program P' such that

$$\forall\ r \quad \Pr[P'(r) = x \cdot r] \geq 1 - \tfrac{1}{n^2}. \tag{11}$$

(Once we know how to compute $x \cdot r$ for every r with high probability, it is easy to recover x bit-by-bit using the observation that if e_i is the n-bit vector that is 1 in the ith position and zero elsewhere then $x \cdot e_i = a_i$, the ith bit of a.)

"*On input r, repeat the following trial $O(\log n/\delta^2)$ times. Pick y randomly from $GF(2)^n$ and compute the bit $P(r+y) + P(y)$. At the end, output the majority value.*"

The main observation is that when y is randomly picked from $GF(2)^n$ then $r + y$ and y are both randomly distributed in $GF(2)^n$, and hence the probability that $P(r + y) \neq a \cdot (r + y)$ or $P(y) \neq a \cdot y$ is at most $2 \cdot (1/4 - \delta) = 1/2 - 2\delta$. Thus with probability at least $1/2 + 2\delta$, each trial produces the correct bit. Then Chernoff bounds imply that probability is at least $1 - 1/n^2$ that the final majority is correct.

General Case:

The idea for the general case is very similar, the only difference being that this time we want to pick $r_1, \ldots, r_m$ so that we already "know" $x \odot r_i$. The preceding statement may appear ridiculous, since knowing the inner product of x with $m \geq n$ random vectors is, with high probability, enough to reconstruct x (see exercises). The explanation is that the r_i's will not be completely random. Instead, they will be pairwise independent. Recall the following construction of a set of pairwise independent vectors: Pick k random vectors $t_1, t_2, \ldots, t_k \in GF(2)^n$ and for each nonempty $S \subseteq \{1, \ldots, k\}$ define $Y_S = \sum_{i \in S} t_i$. This gives $2^k - 1$ vectors and for $S \neq S'$ the random variables $Y_S, Y_{S'}$ are independent of each other.

Now let us describe the observation at the heart of the proof. Suppose $m = 2^k - 1$ and our random strings $r_1, \ldots, r_m$ are $\{Y_S\}$'s from the previous paragraph. Then $x \odot Y_S = x \odot (\sum_{i \in S} t_i) = \sum_{i \in S} x \odot t_i$. Hence if we know $x \odot t_i$ for $i = 1, \ldots, k$, we also know $x \odot Y_S$. Of course, we don't actually know $x \odot t_i$ for $i = 1, \ldots, k$ since x is unknown and the t_i's are random vectors. But we can just try all 2^k possibilities for the vector $(x \odot t_i)_{i=1,\ldots,k}$ and run the rest of the algorithm for each of them. Whenever our "guess" for these innerproducts is correct, the algorithm succeeds in producing x and this answer can be checked by applying f_n on it (as already noted). Thus the guessing multiplies the running time by a factor 2^k, which is only m. This is why we can assume that we know $x \odot Y_S$ for each subset S.

The details of the rest of the algorithm are similar to before. *Pick m pairwise independent vectors Y_S's such that, as described above, we "know" $x \odot Y_S$ for all S. For each $i = 1, 2, \ldots, n$, and each S run A on the input $(f_n(x), Y_S \oplus e_i)$ (where $Y_S \oplus e_i$ is Y_S with its ith entry flipped). Compute the majority value of $A(f_n(x), Y_S \oplus e_i) - x \odot Y_S$ among all S's and use it as your guess for x_i.*

Suppose $x \in GF(2)^n$ satisfies (9). We will show that this algorithm produces all n bits of x with probability at least $1/2$. Fix i. For each i, the guess for x_i is a majority of m bits. The expected number of bits among these that agree with x_i is $m(1/2 + \delta/2)$, so for the majority vote to result in the incorrect answer it must be the case that the number of incorrect values deviates from its expectation by more than $m\delta/2$. Now, we can bound the variance of this random variable and apply Chebyshev's inequality (Lemma A.14 in the Appendix) to conclude that the probability of such a deviation is $\leq \frac{4}{m\delta^2}$.

Here is the calculation using Chebyshev's inequality. Let ξ_S denote the event that A produces the correct answer on $(f_n(x), Y_S \oplus e_i)$. Since x satisfies (9) and $Y_S \oplus e_i$ is randomly distributed over $GF(2)^n$, we have $E(\xi_S) = 1/2 + \delta/2$ and $Var(\xi_S) = E(\xi_S)(1 - E(\xi_S)) < 1$. Let $\xi = \sum_S \xi_S$ denote the number of correct answers on a sample of size m. By linearity of expectation, $E[\xi] = m(1/2 + \delta/2)$. Furthermore, the Y_S's are pairwise independent, which implies that the same is true for the outputs ξ_S's produced by the algorithm A on them. Hence by pairwise independence $Var(\xi) < m$. Now, by Chebyshev's inequality, the probability that the majority vote is incorrect is at most $\frac{4Var(\xi)}{m^2\delta^2} \leq \frac{4}{m\delta^2}$.

Finally, setting $m > 8/n\delta^2$, the probability of guessing the ith bit incorrectly is at most $1/2n$. By the union bound, the probability of guessing the whole word incorrectly is at most $1/2$. Hence, for every x satisfying (9), we can find the preimage of $f(x)$ with probability at least $1/2$, which makes the overall probability of inversion at least $\delta/2$. The running time is about $m^2 n\times$ (running time of A), which is $\frac{n^3}{\delta^4} \times t(n)$, as we had claimed. ■

10.3.2 Pseudorandom number generation

We saw that if f is a one-way permutation, then $g(x, r) = (f(x), r, x \odot r)$ is a pseudorandom generator that stretches $2n$ bits to $2n + 1$ bits. Stretching to even more bits is easy too, as we now show. Let $f^i(x)$ denote the i-th iterate of f on x (i.e., $f(f(f(\cdots(f(x)))))$ where f is applied i times).

THEOREM 10.16
If f is a one-way permutation then $g_N(x, r) = (r, x \odot r, f(x) \odot r, f^2(x) \odot r, \ldots, f^N(x) \odot r)$ is a pseudorandom generator for $N = n^c$ for any constant $c > 0$.

PROOF: Since any distinguishing machine could just reverse the string as a first step, it clearly suffices to show that the string $(r, f^N(x) \odot r, f^{N-1}(x) \odot r, \ldots, f(x) \odot r, x \odot r)$ looks pseudorandom. By Yao's theorem (Theorem 10.12), it suffices to show the difficulty of bit-prediction. For contradiction's sake, assume there is a PPT machine A such that when $x, r \in \{0,1\}^n$ and $i \in \{1, \ldots, N\}$ are randomly chosen,

$$\Pr[A \text{ predicts } f^i(x) \odot r \text{ given } (r, f^N(x) \odot r, f^{N-1}(x) \odot r, \ldots, f^{i+1}(x) \odot r)] \geq \frac{1}{2} + \epsilon.$$

We describe an algorithm B that given $f(z), r$ where $z, r \in \{0,1\}^n$ are randomly chosen, predicts the hardcore bit $z \odot r$ with reasonable probability, which contradicts Theorem 10.14.

Algorithm B picks $i \in \{1, \ldots, N\}$ randomly. Let $x \in \{0,1\}^n$ be such that $f^i(x) = z$. There is of course no efficient way for B to find x, but for any $l \geq 1$, B can efficiently compute $f^{i+l}(x) = f^{l-1}(f(z))$! So it produces the string $r, f^N(x) \odot r, f^{N-1}(x) \odot r, \ldots, f^{i+1}(x) \odot r$ and uses it as input to A. By assumption, A predicts $f^i(x) \odot r = z \odot r$ with good odds. Thus we have derived a contradiction to Theorem 10.14. ■

10.4 Applications

Now we give some applications of the ideas introduced in the chapter.

10.4.1 Pseudorandom functions

Pseudorandom functions are a natural generalization of (and are easily constructed using) pseudorandom generators. This is a function $g : \{0,1\}^m \times \{0,1\}^n \to \{0,1\}^m$. For each $K \in \{0,1\}^m$ we denote by $g|_K$ the function from $\{0,1\}^n$ to $\{0,1\}^m$ defined by $g|_K(x) = g(K, x)$. Thus the family contains 2^m functions from $\{0,1\}^n$ to $\{0,1\}^m$, one for each K.

We say g is a pseudorandom function generator if it passes a "Turing test" of randomness analogous to that in Yao's definition of a pseudorandom generator (Definition 10.11).

Recall that the set of all functions from $\{0,1\}^n$ to $\{0,1\}^m$, denoted $\mathcal{F}_{n,m}$, has cardinality $(2^m)^{2^n}$. The PPT machine is presented with an "oracle" for a function from $\{0,1\}^n$ to $\{0,1\}^n$. The function is one of two types: either a function chosen randomly from $\mathcal{F}_{n,m}$, or a function $f|_K$ where $K \in \{0,1\}^m$ is randomly chosen. The PPT machine is allowed to query the oracle in any points of its choosing. We say $f|_K$ is a pseudorandom function generator if for all $c > 1$ the PPT has probability less than n^{-c} of detecting which of the two cases holds. (A completely formal definition would resemble Definition 10.1 and talk about a family of generators, one for each n. Then m is some function of n.)

Now we describe a construction of a pseudorandom function generator g from a length-doubling pseudorandom generator $f : \{0,1\}^m \to \{0,1\}^{2m}$. For any $K \in \{0,1\}^m$ let T_K be a complete binary tree of depth n whose each node is labelled with an m-bit string. The root is labelled K. If a node in the tree has label y then its left child is labelled with the first m bits of

Figure unavailable in pdf file.

Figure 10.3: Constructing a pseudorandom function from $\{0,1\}^n$ to $\{0,1\}^m$ using a random key $K \in \{0,1\}^m$ and a length-doubling pseudorandom generator $g : \{0,1\}^m \to \{0,1\}^{2m}$.

$f(y)$ and the right child is labelled with the last m bits of $f(y)$. Now we define $g(K, x)$. For any $x \in \{0,1\}^n$ interpret x as a label for a path from root to leaf in T_K in the obvious way and output the label at the leaf. (See Figure 10.3.)

We leave it as an exercise to prove that this construction is correct.

A pseudorandom function generator is a way to turn a random string K into an implicit description of an exponentially larger "random looking" string, namely, the table of all values of the function $g|_K$. This has proved a powerful primitive in cryptography; see the next section. Furthermore, pseudorandom function generators have also figured in a very interesting explanation of why current lowerbound techniques have been unable to separate **P** from **NP**; see Chapter **??**.

10.4.2 Private-key encryption: definition of security

We hinted at a technique for private-key encryption in our discussion of a one-time pad (including the pseudorandom version) at the start of Section 10.2. But that discussion completely omitted what the *design goals* of the encryption scheme were. This is an important point: design of insecure systems often traces to a misunderstanding about the type of security ensured (or not ensured) by an underlying protocol.

The most basic type of security that a private-key encryption should ensure is *semantic security*. Informally speaking, this means that whatever can be computed from the encrypted message is also computable without access to the encrypted message and knowing only the *length* of the message. The formal definition is omitted here but it has to emphasize the facts that we are talking about an ensemble of encryption functions, one for each message size (as in Definition 10.1) and that the encryption and decryption is done by probabilistic algorithms that use a shared private key, and that *for every message* the guarantee of security holds with high probability with respect to the choice of this private key.

Now we describe an encryption scheme that is semantically secure. Let $f : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$ be a pseudorandom function generator. The two parties share a secret random key $K \in \{0,1\}^n$. When one of them wishes to send a message $x \in \{0,1\}^n$ to the other, she picks a random string $r \in \{0,1\}^n$ and transmits $(r, x \oplus f_K(r))$. To decrypt the other party computes $f_K(r)$ and then XORs this string with the last n bits in the received text.

We leave it as an exercise to show that this scheme is semantically secure.

10.4.3 Derandomization

The existence of pseudorandom generators implies subexponential deterministic algorithms for **BPP**: this is usually referred to as *derandomization* of **BPP**. (In this case, the derandomization is only partial since it results in a subexponential deterministic algorithm. Stronger complexity assumptions imply a full derandomization of **BPP**, as we will see in Chapter 17.)

Theorem 10.17
If for every $c > 1$ there is a pseudorandom generator that is secure against circuits of size n^c, then $\mathbf{BPP} \subseteq \cap_{\varepsilon>0}\mathbf{DTIME}(2^{n^\varepsilon})$.

Proof: Let us fix an $\varepsilon > 0$ and show that $\mathbf{BPP} \subseteq \mathbf{DTIME}(2^{n^\varepsilon})$.

Suppose that M is a **BPP** machine running in n^k time. We can build another probabilistic machine M' that takes n^ε random bits, streches them to n^k bits using the pseudorandom generator and then simulates M using this n^k bits as a random string. Obviously, M' can be simulated by going over all binary strings n^ε, running M' on each of them, and taking the majority vote.

It remains to prove that M and M' accept the same language. Suppose otherwise. Then there exists an infinite sequence of inputs $x_1, \dots, x_n, \dots$ on which M distinguishes a truly random string from a pseudorandom string with a high probability, because for M and M' to produce different results, the probability of acceptance should drop from $2/3$ to below $1/2$. Hence we can build a distinguisher similar to the one described in the previous theorem by hardwiring these inputs into a circuit family. ■

The above theorem shows that the existence of hard problems implies that we can reduce the randomness requirement of algorithms. This "hardness versus randomness" tradeoff is studied more deeply in Chapter 17.

Remark 10.18
There is an interesting connection to *discrepancy theory*, a field of mathematics. Let $\mathcal{S}$ be a set of subsets of $\{0,1\}^n$. Subset $A \subset \{0,1\}^n$ has *discrepancy ϵ with respect to $\mathcal{S}$* if for every $s \in \mathcal{S}$,

$$\left| \frac{|s \cap A|}{|S|} - \frac{|A|}{2^n} \right| \leq \epsilon.$$

Our earlier result that $\mathbf{BPP} \subseteq \mathbf{P}/poly$ showed the *existence* of polynomial-size sets A that have low discrepancy for all sets defined by polynomial-time Turing machines (we only described discrepancy for the universe $\{0,1\}^n$ but one can define it for all input sizes using $\limsup$). The goal of derandomization is to explicitly construct such sets; see Chapter 17.

10.4.4 Tossing coins over the phone and bit commitment

How can two parties A and B toss a fair random coin over the phone? (Many cryptographic protocols require this basic primitive.) If only one of them actually tosses a coin, there is nothing to prevent him from lying about the result. The following fix suggests itself: both players toss a coin and they take the XOR as the shared coin. Even if B does not trust A to use a fair coin, he knows that as long as his bit is random, the XOR is also random. Unfortunately, this idea also does not work because the player who reveals his bit first is at a disadvantage: the other player could just "adjust" his answer to get the desired final coin toss.

This problem is addressed by the following scheme, which assumes that A and B are polynomial time turing machines that cannot invert one-way permutations. The protocol itself is called *bit commitment*. First, A chooses two strings x_A and r_A of length n and sends a message $(f_n(x_A), r_A)$, where f_n is a one-way permutation. This way, A commits the string x_A without revealing it. Now B selects a random bit b and conveys it. Then A reveals x_A and they agree to use the XOR of b and $(x_A \odot r_A)$ as their coin toss. Note that B can verify that x_A is the same as in the first message by applying f_n, therefore A cannot change her mind after learning B's bit. On the other hand, by the Goldreich–Levin theorem, B cannot predict $x_A \odot r_A$ from A's first message, so this scheme is secure.

10.4.5 Secure multiparty computations

This concerns a vast generalization of the setting in Section 10.4.4. There are k parties and the ith party holds a string $x_i \in \{0,1\}^n$. They wish to compute $f(x_1, x_2, \ldots, x_k)$ where $f : \{0,1\}^{nk} \to \{0,1\}$ is a polynomial-time computable function known to all of them. (The setting in Section 10.4.4 is a subcase whereby each x_i is a bit —randomly chosen as it happens— and f is XOR.) Clearly, the parties can just exchange their inputs (suitably encrypted if need be so that unauthorized eavesdroppers learn nothing) and then each of them can compute f on his/her own. However, this leads to all of them knowing each other's input, which may not be desirable in many situations. For instance, we may wish to compute statistics (such as the average) on the combination of several medical databases that are held by different hospitals. Strict privacy and nondisclosure laws may forbid hospitals from sharing information about individual patients. (The original example Yao gave in introducing the problem was of k people who wish to compute the average of their salaries without revealing their salaries to each other.)

We say that a multiparty protocol for computing f is *secure* if at the end no party learns anything new apart from the value of $f(x_1, x_2, \ldots, x_k)$. The formal definition is inspired by the definition of a pseudorandom generator, and states that for each i, the bits received by party i during the protocol should be computationally indistinguishable from completely random bits[3].

It is completely nonobvious why such protocols must exist. Yao [**?**] proved existence for $k = 2$ and Goldreich, Micali, Wigderson [**?**] proved existence for general k. We will not describe this protocol in any detail here except to mention that it involves "scrambling" the circuit that computes f.

10.4.6 Lowerbounds for machine learning

In *machine learning* the goal is to learn a succinct function $f : \{0,1\}^n \to \{0,1\}$ from a sequence of type $(x_1, f(x_1)), (x_2, f(x_2)), \ldots,$ where the x_i's are randomly-chosen inputs. Clearly, this is impossible in general since a random function has no succinct description. But suppose f has a succinct description, e.g. as a small circuit. Can we learn f in that case?

The existence of pseudorandom functions implies that even though a function may be polynomial-time computable, there is no way to learn it from examples in polynomial time. In fact it is possible to extend this impossibility result (though we do not attempt it) to more restricted function families such as $\mathbf{NC}^1$ (see Kearns and Valiant [**?**]).

10.5 Recent developments

The earliest cryptosystems were designed using the SUBSET SUM problem. They were all shown to be insecure by the early 1980s. In the last few years,

[3]Returning to our medical database example, we see that the hospitals can indeed compute statistics on their combined databases without revealing any information to each other —at least any information that can be extracted feasibly. Nevetheless, it is unclear if current privacy laws allow hospitals to perform such secure multiparty protocols using patient data— an example of the law lagging behind scientific progress.

interest in such problems —and also the related problems of computing approximate solutions to the shortest and nearest lattice vector problems— has revived, thanks to a one-way function described in Ajtai [?], and a public-key cryptosystem described in Ajtai and Dwork [?] (and improved on since then by other researchers). These constructions are secure on *most* instances iff they are secure on *worst-case* instances. (The idea used is a variant of random self-reducibility.)

Also, there has been a lot of exploration of the exact notion of security that one needs for various cryptographic tasks. For instance, the notion of semantic security in Section 10.4.2 may seem quite strong, but researchers subsequently realized that it leaves open the possibility of some other kinds of attacks, including *chosen ciphertext* attacks, or attacks based upon *concurrent* execution of several copies of the protocol. Achieving security against such exotic attacks calls for many ideas, most notably *zero knowledge* (a brief introduction to this concept appears in Section ??).

Chapter notes and history

In the 1940s, Shannon speculated about topics reminiscent of complexity-based cryptography. The first concrete proposal was made by Diffie and Hellman [?], though their cryptosystem was later broken. The invention of the RSA cryptosystem (named after its inventors Ron Rivest, Adi Shamir, and Len Adleman) [?] brought enormous attention to this topic. In 1981 Shamir [?] suggested the idea of replacing a one-time pad by a pseudorandom string. He also exhibited a weak pseudorandom generator assuming the average-case intractability of the RSA function. The more famous papers of Blum and Micali [?] and then Yao [?] laid the intellectual foundations of private-key cryptography. (The hybrid argument used by Yao is a stronger version of one in an earlier important manuscript of Goldwasser and Micali [?] that proposed probabilistic encryption schemes.) The construction of pseudorandom functions in Section 10.4.1 is due to Goldreich, Goldwasser, and Micali [?]. The question about tossing coins over a telephone was raised in an influential paper of Blum [?]. Today complexity-based cryptography is a vast field with several dedicated conferences. Goldreich [?]'s two-volume book gives a definitive account.

A scholarly exposition of number theoretic algorithms (including generating random primes and factoring integers) appears in Victor Shoup's recent book [?] and the book of Bach and Shallit [?].

Theorem 10.13 and its very technical proof is in Håstad et al. [?] (the relevant conference publications are a decade older).

Our proof of the Goldreich-Levin theorem is usually attributed to Rackoff (unpublished).

Exercises

§1 Show that if **P** = **NP** then one-way functions and pseudorandom generators do not exist.

§2 (Requires just a little number theory). Prove that if some algorithm inverts the Rabin function $f_m(x) = x^2 \pmod{m}$ on a $1/\text{poly}(\log m)$ fraction of inputs then we can factor m in $\text{poly}(\log m)$ time.

Hint: Suppose $m = pq$ where p, q are prime numbers. Then x^2 has 4 "square roots" modulo m.

§3 Show that if f is a one-way permutation then so is f^k (namely, $f(f(f(\cdots(f(x)))))$ where f is applied k times) where $k = n^c$ for some fixed $c > 0$.

§4 Assuming one-way functions exist, show that the above fails for one-way functions.

Hint: You have to design a one-way function where f^k is not one-way.

§5 Suppose $a \in \mathbf{GF(2)}^m$ is an unknown vector. Let $r_1, r_2, \ldots, r_m \in \mathbf{GF(2)}^m$ be randomly chosen, and $a \odot r_i$ revealed to us for all $i = 1, 2, \ldots, m$. Describe a deterministic algorithm to reconstruct a from this information, and show that the probability (over the choice of the r_i's) is at least 1/4 that it works.

Hint: You need to show that a certain determinant is nonzero.

This shows that the "trick" in Goldreich-Levin's proof is necessary.

§6 Suppose somebody holds an unknown n-bit vector a. Whenever you present a randomly chosen subset of indices $S \subseteq \{1, \ldots, n\}$, then with probability at least $1/2+\epsilon$, she tells you the parity of the all the bits in a indexed by S. Describe a guessing strategy that allows you to guess a (an n bit string!) with probability at least $(\frac{\epsilon}{n})^c$ for some constant $c > 0$.

§7 Suppose $g:\{0,1\}^n \to \{0,1\}^{n+1}$ is any pseudorandom generator. Then use g to describe a pseudorandom generator that stretches n bits to n^k for any constant $k > 1$.

§8 Show the correctness of the pseudorandom function generator in Section 10.4.1.

Hint: Use a hybrid argument which replaces the labels on the first k levels of the tree by completely random strings. Note that the random labels do not need to be assigned ahead of time — this would take at least 2^k time —but can be assigned on the fly whenever they are needed by the distinguishing algorithm.

§9 Formalize the definition of semantic security and show that the encryption scheme in Section 10.4.2 is semantically secure.

Hint: First show that for all message pairs x, y their encryptions are indistinguishable by polynomial-time algorithms. Why does this suffice?