Computational Complexity: A Modern Approach

Sanjeev Arora and Boaz Barak
Princeton University

http://www.cs.princeton.edu/theory/complexity/

complexitybook@gmail.com

Chapter 9

Cryptography

"Human ingenuity cannot concoct a cipher which human ingenuity cannot resolve."
E. A. Poe, 1841

"In designing a good cipher ... it is not enough merely to be sure none of the standard methods of cryptanalysis work– we must be sure that no method whatever will break the system easily. This, in fact, has been the weakness of many systems. ... The problem of good cipher design is essentially one of finding difficult problems, subject to certain other conditions. This is a rather unusual situation, since one is ordinarily seeking the simple and easily soluble problems in a field."
C. Shannon [Sha49b]

"While the **NP** *complete problems show promise for cryptographic use, current understanding of their difficulty includes only worst case analysis. For cryptographic purposes, typical computational costs must be considered."*
W. Diffie and M. Hellman [DH76]

Cryptography is much older than computational complexity. Ever since people began to write, they invented methods for "secret writing" that would be difficult to decipher for others. But the numerous methods of *encryption* or "secret writing" devised over the years all had one common characteristic— sooner or later they were broken. But everything changed in 1970's, when thanks to the works of several researchers, *modern cryptography* was born, whereby computational complexity was used to argue about the security of the encryption schemes. In retrospect this connection seems like a natural one, since the codebreaker has bounded computational resources (even if she has computers at her disposal) and therefore to ensure security one should try to ensure that the codebreaking problem is computationally difficult.

Another notable difference between modern cryptography and the older notion is that the security of encryption no longer relies upon the the encryption technique being kept secret. In modern cryptography, the encryption technique itself is well-known, yet nevertheless it is hard to break. Furthermore, modern cryptography is about much more than just encryption, and the security of all these schemes is proved by means of *reductions* similar (though not identical) to those used in the theory of **NP**-completeness.

This new focus on building system from basic problems via reductions enabled modern cryptography to achieve two seemingly contradictory goals. On the one hand these new schemes are much more secure— systems such as the RSA encryption [RSA78] have withstood more attacks by talented mathematicians assisted with state of the art computers than every previous encryption in history. On the other hand, their security requirements are much more stringent— we require the schemes to remain secure even when the encryption key is known to the attacker (i.e., *public key* encryption), and even when the attacker gets access to encryptions and decryptions of text of her choice (so-called *chosen plaintext* and *chosen ciphertext* attacks). Moreover, modern cryptography provides schemes that go much beyond simple encryption— tools such as digital signatures, zero knowledge proofs, electronic voting and auctions, and more. All of these are shown to be secure against *every* polynomial-time attack, and not just attacks we can think of today, as long as the underlying computational problem is indeed hard.

Research on modern cryptography led to significant insights that had impact and applications in complexity theory and beyond that. One is the notion of *pseudorandomness.* Philosophers and scientists have struggled for years to define when to consider a bit string "random enough." Cryptography's answer to this question is that it suffices if this string is drawn from a distribution that "looks random" to all *efficient* (i.e., polynomial-time) observers (see Section 9.2.3). This notion is crucial for the construction of many cryptographic schemes, but is also extremely useful in other areas where random bits are needed. For example cryptographic pseudorandom generators can be used to reduce the randomness requirements of *probabilistic algorithms* such as the ones we saw in Chapter 7; see also Chapter 20. Another insight is the notion of *simulation.* A natural question in cryptography is how one can demonstrate that an attacker cannot learn *anything* about some secret information from observing the behavior of parties holding this information. Cryptography's answer is to show that the attacker's observations can be *simulated* without any access to the secret information. This is epitomized in the notion of *zero knowledge proofs* covered in Section 9.4, and used in many other cryptographic applications.

We start the chapter in Section 9.1 with Shannon's definition of *perfectly secret* encryption and the limitations of such systems. These limitations lead us to consider encryptions that are only *computationally secret*— secure for polynomial-time eavesdroppers— which we construct in Section 9.2 using *pseudorandom generators.* Then, in Section 9.2.3 we show how these generators can be constructed from weaker assumptions. In Section 9.4 we describe *zero knowledge proofs*, a fascinating concept that has had deep implications for cryptography and complexity alike. Finally, in Section 9.5 we mention how these concepts can be used to achieve security in a variety of settings. Cryptography is a huge topic, and so naturally this chapter covers only a tiny sliver of it; the chapter notes contain some excellent choices for further reading. Cryptography is intimately related to notions such as *average-case complexity*, *hardness amplifications* and *derandomization*, see chapters 18, 19, and 20.

9.1 Perfect secrecy and its limitations

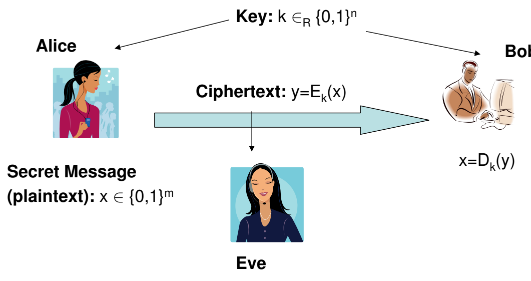
Key: $k \in_R \{0,1\}^n$
Alice
Bob
Ciphertext: $y=E_k(x)$
$x=D_k(y)$
Secret Message (plaintext): $x \in \{0,1\}^m$
Eve

Figure 9.1 In a private key encryption, Alice and Bob share a secret key k chosen at random. To send a plaintext message x to Bob, Alice sends $y = \mathsf{E}_k(x)$ where $\mathsf{E}(\cdot)$ is the *encryption* function that takes a key k and plaintext x to compute the ciphertext y. Bob can decode x by running the decryption algorithm D on inputs k, y.

The fundamental task of cryptography is *encryption.* In this Section we describe this task at a high level, and discuss what it could possibly mean for encryption to be *secure.*

We introduce a simple idea for encryption called the *one time pad* and discuss its strengths and limitations.

The basic setting is described in Figure 9.1— Alice wants to send a secret message x (known as the *plaintext*) to Bob, but her adversary Eve is eavesdropping on the communication channel between Alice and Bob. Thus Alice will "scramble" the plaintext x using an *encryption algorithm* E to obtain a *ciphertext* y which she sends to Bob. Presumably it will be hard or even impossible for Eve to decode the plaintext x from the ciphertext y, but Bob will be able to do so using the *decryption algorithm* D.

Of course, Bob is seeing the same information that Eve is, so in order to do something that Eve cannot, Bob has to know something that Eve doesn't. In the simple setting of private key encryption we assume that Alice and Bob share some secret string k (known as the *key*) that is chosen at random. (Presumably, Alice and Bob met beforehand and agreed on the key k.)

Thus, the encryption scheme is composed of a pair of algorithms (E, D) each taking a key and a message (where we write the key input as a subscript), such that for every key k and plaintext x

$$\mathsf{D}_k(\mathsf{E}_k(x)) = x\,. \tag{1}$$

The condition (1) says nothing about the *security* of the scheme, and could be satisfied by the trivial "encryption" that just outputs the plaintext message. It turns out that defining security is quite subtle. A first attempt at a definition might be to say that a scheme is secure if Eve cannot compute x from $\mathsf{E}_k(x)$, but this may not be sufficient, because this does not rule out the possibility of Eve computing some *partial information* on x. For example, if Eve knows that the plaintext is either the message "`buy`" or "`sell`" then it will be enough for her to learn only the first character of the message, even if she can't recover it completely. Shannon gave the following definition of secure private key encryption that ensures Eve does not learn *anything* about the plaintext from the ciphertext:

Definition 9.1 *(Perfect Secrecy)* Let (E, D) be an encryption scheme for messages of length m and with a key of length n satisfying (1). We say that (E, D) is *perfectly secret* if for every pair of messages $x, x' \in \{0,1\}^m$, the distributions $\mathsf{E}_{U_n}(x)$ and $\mathsf{E}_{U_n}(x')$ are identical.[1] $\diamondsuit$

In a perfectly secret encryption, the ciphertext that Eve sees always has the same distribution, regardless of the plaintext, and so Eve gets absolutely no information on the plaintext. It might seem like a condition so strong that it's impossible to satisfy, but in fact there's a very simple perfectly secret encryption scheme. In the *one-time pad* scheme, to encrypt a message $x \in \{0,1\}^n$ we choose a random key $k \in_{\text{R}} \{0,1\}^n$ and encrypt x by simply sending $x \oplus k$ ($\oplus$ denotes bitwise XOR— vector addition modulo 2). The receiver can recover the message x from $y = x \oplus k$ by XOR'ing y once again with k. It's not hard to see that the ciphertext is distributed uniformly regardless of the plaintext message encrypted, and hence the one-time pad is perfectly secret (see Exercise 9.1).

Of course, as the name suggests, a "one-time pad" must never be reused on another message. If two messages x, x' are encoded using the same pad k, this gives Eve both $k \oplus x$ and $k \oplus x'$, allowing her to compute $(k \oplus x) \oplus (k \oplus x') = x \oplus x'$, which is some nontrivial information about the messages. In fact, one can show that no perfectly secret encryption scheme can use a key size shorter than the message size (see Exercise 9.2).

9.2 Computational security, one-way functions, and pseudorandom generators

Though a one-time pad does provide perfect secrecy, it fails utterly as a practical solution to today's applications where one wishes to securely exchange megabytes or even gigabytes of information. Our discussion above implies that perfect secrecy would require private keys

[1]Recall that U_n denotes the uniform distribution over $\{0,1\}^n$.

that are as long as the messages, and it is unclear such huge keys can be securely exchanged among users. Ideally we want to keep the shared secret key fairly small, say a few hundred bits long. Obviously, to allow this we must relax the perfect secrecy condition somehow. As stated in the introduction, the main idea will be to design encryption schemes that are secure only against eavesdroppers that are *efficient* (i.e., run in polynomial-time). However, the next Lemma shows that even with this restriction on the eavesdropper, achieving perfect secrecy is impossible with small key sizes if $\mathbf{P} = \mathbf{NP}$. Hence assuming $\mathbf{P} \neq \mathbf{NP}$ will be *necessary* for proceeding any further. In fact we will rely on assumptions stronger than $\mathbf{P} \neq \mathbf{NP}$ —specifically, the assumption that a *one-way function exists*— and it is an important research problem to weaken the assumption (ideally to just $\mathbf{P} \neq \mathbf{NP}$) under which cryptographic schemes can be proved secure.

Lemma 9.2 *Suppose that* $\mathbf{P} = \mathbf{NP}$. *Let* (E, D) *be any polynomial-time computable encryption scheme satisfying (1) with key shorter than the message. Then, there is a polynomial-time algorithm* A *satisfying that for every input length* m*, there is a pair of messages* $x_0, x_1 \in \{0,1\}^m$ *such that*

$$\Pr_{\substack{b \in_R \{0,1\} \\ k \in_R \{0,1\}^n}} [A(\mathsf{E}_k(x_b)) = b] \geq 3/4\,, \tag{2}$$

where $n < m$ *denotes the key length for messages of length* m. ♢

Such an algorithm breaks the security of the encryption scheme since, as demonstrated by the "`buy`"/"`sell`" example of Section 9.1, a minimal requirement from an encryption is that Eve cannot tell which one of two random messages was encrypted with probability much better than $1/2$.

Proof of Lemma 9.2: Let (E, D) be an encryption for messages of length m and with key length $n < m$ as in the lemma's statement. Let $S \subseteq \{0,1\}^*$ denote the support of $\mathsf{E}_{U_n}(0^m)$. Note that $y \in S$ if and only if $y = \mathsf{E}_k(0^m)$ for some k, and hence if $\mathbf{P} = \mathbf{NP}$ then membership in S can be efficiently verified. Our algorithm A will be very simple— on input y, it outputs 0 if $y \in S$, and 1 otherwise. We claim that setting $x_0 = 0^m$, there exists some $x_1 \in \{0,1\}^m$ such that (2) holds.

Indeed, for every message x, let D_x denote the distribution $\mathsf{E}_{U_n}(x)$. By the definition of A and the fact that $x_0 = 0^m$, $\Pr[A(D_{x_0}) = 0] = 1$. Because

$$\Pr_{\substack{b \in_R \{0,1\} \\ k \in_R \{0,1\}^n}} [A(\mathsf{E}_k(x_b)) = b] = \frac{1}{2}\Pr[A(D_{x_0}) = 0] + \frac{1}{2}\Pr[A(D_{x_1}) = 1]$$

$$= \frac{1}{2} + \frac{1}{2}\Pr[A(D_{x_1}) = 1]\,,$$

it suffices to show that there's some $x_1 \in \{0,1\}^m$ such that $\Pr[A(D_{x_1}) = 1] \geq 1/2$. In other words, it suffices to show that $\Pr[D_{x_1} \in S] \leq 1/2$ for some $x_1 \in \{0,1\}^m$.

Suppose otherwise that $\Pr[D_x \in S] > 1/2$ for every $x \in \{0,1\}^m$. Define $S(x,k)$ to be 1 if $\mathsf{E}_k(x) \in S$ and to be 0 otherwise, and let $T = \mathsf{E}_{x \in_R \{0,1\}^m, k \in \{0,1\}^n}[S(x,k)]$. Then under our assumption, $T > 1/2$. But reversing the order of expectations, we see that

$$T = \mathop{\mathsf{E}}_{k \in \{0,1\}^n}\Big[\mathop{\mathsf{E}}_{x \in \{0,1\}^m}[S(x,k)]\Big] \leq 1/2\,,$$

where the last inequality follows from the fact that for every fixed key k, the map $x \mapsto \mathsf{E}_k(x)$ is one-to-one and hence at most $2^n \leq 2^m/2$ of the x's can be mapped under it to a set S of size $\leq 2^n$. Thus we obtained a contradiction to the assumption that $\Pr[D_x \in S] > 1/2$ for every $x \in \{0,1\}^m$. ■

Before proceeding further, we make a simple definition that will greatly simplify notation throughout this chapter.

Definition 9.3 *(Negligible functions)*
A function $\epsilon : \mathbb{N} \to [0,1]$ is called *negligible* if $\epsilon(n) = n^{-\omega(1)}$ (i.e., for every c and sufficiently large n, $\epsilon(n) < n^{-c}$).

Because negligible functions tend to zero very fast as their input grows, events that happen with negligible probability can be safely ignored in most practical and theoretical settings.

9.2.1 One way functions: definition and some examples

The above discussion suggests that complexity-theoretic conjectures are necessary to prove the security of encryption schemes. Now we introduce an object that is useful not only in this context but also many others in cryptography. This is a *one-way function*: a function that is easy to compute but hard to invert for a polynomial-time algorithm:

Definition 9.4 *(One way functions)*
A polynomial-time computable function $f : \{0,1\}^* \to \{0,1\}^*$ is a *one-way function* if for every probabilistic polynomial-time algorithm A there is a negligible function $\epsilon : \mathbb{N} \to [0,1]$ such that for every n,

$$\Pr_{\substack{x \in_{\mathrm{R}} \{0,1\}^n \\ y = f(x)}}[A(y) = x' \text{ s.t. } f(x') = y] < \epsilon(n)\,.$$

Conjecture 9.5
There exists a one-way function.

Exercise 9.5 asks you to show that Conjecture 9.5 implies that $\mathbf{P} \neq \mathbf{NP}$. Most researchers believe Conjecture 9.5 is true because there are several examples for functions that no one has yet been able to invert. Now we describe several.

Multiplication: Simple multiplication turns out to be hard to invert. That is, the function that treats its input $x \in \{0,1\}^n$ as describing two $n/2$-bit numbers A and B and outputs $A \cdot B$ is believed to be one way. Inverting this function is known as the *integer factorization problem*. Of course, it's easy to factor a number N using at most N (or even only $\sqrt{N}$) trial divisions. But if N is an n-bit number this is an exponential in n number of operations. At the moment no polynomial (i.e., polylog(N)) time algorithm is known for this problem, and the best factoring algorithm runs in time $2^{O(\log^{1/3} N \sqrt{\log\log N})}$ [LLMP90].[2]

A more standard implementation of a one-way function based on factoring is the following. Treat the input $x \in \{0,1\}^n$ as randomness that is used to generate two random $n^{1/3}$-bit primes P and Q. (We can do so by generating random numbers and testing their primality using the algorithm described in Chapter 7.) Then output $P \cdot Q$.

Factoring integers has captured the attention of mathematicians for at least two millennia, way before the invention of computers. Yet no efficient factorization algorithm was found, leading to the conjecture that no such algorithm exists. Then this function is indeed one-way, though this conjecture is obviously much stronger than the conjecture that $\mathbf{P} \neq \mathbf{NP}$ or the conjecture that *some* one-way function exists.

RSA and Rabin functions: (These examples require a bit of number theory; see Section A.3 for a quick review) The *RSA function*[3] is another very popular candidate for a one-way function. We assume that for every input length n there is an n-bit composite integer N that was generated in some way, and some number e that is coprime to $\varphi(N) = |\mathbb{Z}_N^*|$ where $\mathbb{Z}_N^*$ is the multiplicative group of numbers coprime

[2] If A and B are chosen randomly then it's not hard to find a *some* prime factor of $A \cdot B$, since $A \cdot B$ will have a small prime factor with high probability. But finding the all the prime factors or even finding any representation of $A \cdot B$ as the multiplication of two numbers each no larger than $2^{n/2}$ can be shown to be equivalent (up to polynomial factor) to factoring the product of two random primes.

[3] RSA are the initials of this function's discoverers— Rivest, Shamir, and Adleman; see the chapter notes.

to N. (Typically N would be generated as a product of two $n/2$-long primes; e is often set to be simply 3.) The function $RSA_{N,e}$ treats its input as a number X in $\mathbb{Z}^*_N$ and outputs $X^e \pmod N$).[4] It can be shown that because e is coprime to $\varphi(N)$, this function is one-to-one on Z^*_N. A related candidate one-way function is the *Rabin function* that given a number N that is the product of two odd primes P, Q such that $P, Q = 1 \pmod 4$), maps $X \in QR_N$ into $X^2 \pmod N$), where QR_N is the set of *quadratic residues* modulo N (an element $X \in \mathbb{Z}^*_N$ is a quadratic residue modulo N if $X = W^2 \pmod N$ for some $W \in \mathbb{Z}^*_N$). Again, it can be shown that this function is one-to-one on QR_N.

While both the RSA function and the Rabin function are believed to be hard to invert, inverting them is actually easy if one knows the factorization of N. In the case of the RSA function, the factorization can be used to compute $\varphi(N)$ and from that the number d such that $d = e^{-1} \pmod{\varphi(N)}$. It's not hard to verify that the function $Y^d \pmod N$ is the inverse of the function $X^e \pmod N$. In the case of the Rabin function, if we know the factorization then we can use the Chinese Remainder Theorem to reduce the problem of taking a square root modulo N to taking square roots modulo the prime factors of N, which can be done in polynomial time. Because these functions are conjectured hard to invert but become easy to invert once you know certain information (i.e., N's factorization), they are known as *trapdoor* one-way functions, and are crucial to obtaining *public key* cryptography. It it known that inverting Rabin's function is in fact *computationally equivalent* to factoring N (see Exercise 9.7). No such equivalence is known for the RSA function.

Levin's universal one-way function: There is a function $f_{\mathcal{U}}$ that has a curious property: if there exists *some* one-way function f then $f_{\mathcal{U}}$ is also a one-way function. For this reason, the function $f_{\mathcal{U}}$ is called a *universal* one-way function. It is defined as follows: treat the input as a list $x_1, \ldots, x_{\log n}$ of $n/\log n$ bit long strings. Output $M_1^{n^2}(x_1), \ldots, M_{\log n}^{n^2}(x_n)$ where M_i denotes the i^{th} Turing machine according to some canonical representation and we define $M^t(x)$ to be the output of the Turing machine M on input x if M uses at most t computational steps on input x. If M uses more than t computational steps on x then we define $M^t(x)$ to be the all-zeroes string $0^{|x|}$. Exercise 9.6 asks you to prove the universality of $f_{\mathcal{U}}$.

There are also examples of candidate one-way functions that have nothing to do with number theory (e.g., one-way functions arising from block ciphers such as the AES [DR02]).

9.2.2 Encryption from one-way functions

Now we show that one-way functions can be used to to design secure encryption schemes with keys much shorter than the message length.

Theorem 9.6 *(Encryption from one-way function)*
Suppose that one-way functions exist. Then for every $c \in \mathbb{N}$ there exists a computationally secure encryption scheme (E, D) *using n-length keys for n^c-length messages.*

Of course to make sense of Theorem 9.6, we need to define the term "computationally secure". The idea is to follow the intuition that a secure encryption should not reveal any partial information about the plaintext to a polynomial-time eavesdropper, but due to some subtleties, the actual definition is somewhat cumbersome. Thus, for the sake of presentation we'll use a simpler relaxed definition that an encryption is "computationally secure" if any individual bit of the plaintext chosen at random cannot be guessed by the eavesdropper

[4]We can map the input to $\mathbb{Z}^*_N$ by simply reducing the input modulo N— the probability (over the choice of the input) that the result will not be coprime to N is negligible.

with probability non-negligibly higher than 1/2. That is, we say that a scheme (E, D) using length n keys for length m messages is computationally secure if for every probabilistic polynomial-time A, there's a negligible function $\epsilon : \mathbb{N} \to [0,1]$ such that

$$\Pr_{\substack{k \in_R \{0,1\}^n \\ x \in_R \{0,1\}^m}} [A(\mathsf{E}_k(x)) = (i,b) \text{ s.t. } x_i = b] \le 1/2 + \epsilon(n)\,. \qquad (3)$$

The full-fledged, stronger notion of computational security (whose standard name is *semantic security*) is developed in Exercise 9.9, where it is also shown that Theorem 9.6 holds also for this stronger notion.

9.2.3 Pseudorandom generators

Recall the one-time pad idea of Section 9.1, whose sole limitation was the need for a shared random string whose length is the same as the combined length of all the messages that need to be transmitted. The main idea in the proof of Theorem 9.6 is to show how to take a small random key of length n and stretch it to a much larger string of length m that is still "random enough" that it provides security against polynomial-time eavesdroppers when used as a one-time pad. This stretching of the random string uses a tool called a *pseudorandom generator*, which has applications even beyond cryptography.

Example 9.7
What is a random-enough string? Scientists have struggled with this question before. 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 [Knu73] 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.

Cryptography's answer to the above question is simple and satisfying. First, instead of trying to describe what it means for a single string to be "random-looking" we focus on distributions on strings. Second, instead of focusing on individual tester algorithms as the statisticians did, we say that the distribution has to "look" like the uniformly random distribution to *every* polynomial-time algorithm. Such a distribution is called *pseudorandom.* The distinguisher algorithm is given a sample string that is drawn from either the uniform distribution or the unknown distribution. The algorithm outputs "1" or "0" depending upon whether or not this string looks random to it. (An example of such an algorithm is the statistics-based tester of Example 9.7.) The distribution is said to be *pseudorandom* if the probability that the polynomial-time algorithm outputs 1 is essentially the same on the two distributions, *regardless* of which algorithm is used[5].

[5]Note that this definition is reminiscent of a "blind test": for instance we say that an artificial sweetner is "sugar-like" if the typical consumer cannot tell the difference between it and sugar in a blind-test. However, the definition of a pseudorandom distribution is more stringent since the distribution has to fool *all* distinguisher algorithms. The analogous notion for a sweetner would require it to taste like sugar to every human.

Definition 9.8 *(Secure pseudorandom generators)*
Let $G : \{0,1\}^* \to \{0,1\}^*$ be a polynomial-time computable function. Let $\ell : \mathbb{N} \to \mathbb{N}$ be a polynomial-time computable function such that $\ell(n) > n$ for every n. We say that G is a *secure pseudorandom generator of stretch* $\ell(n)$, if $|G(x)| = \ell(|x|)$ for every $x \in \{0,1\}^*$ and for every probabilistic polynomial-time A, there exists a negligible function $\epsilon : \mathbb{N} \to [0,1]$ such that

$$\Big|\Pr[A(G(U_n)) = 1] - \Pr[A(U_{\ell(n)}) = 1]\Big| < \epsilon(n)\,,$$

for every $n \in \mathbb{N}$.

Theorem 9.9 *(Pseudorandom generators from one-way functions* [HILL99]*)*
If one-way functions exist, then for every $c \in \mathbb{N}$, there exists a secure pseudorandom generator with stretch $\ell(n) = n^c$.

Definition 9.8 states that it's infeasible for polynomial-time adversaries to distinguish between a completely random string of length $\ell(n)$ and a string that was generated by applying the generator G to a much shorter random string of length n. Thus, it's not hard to verify that Theorem 9.9 implies Theorem 9.6: if we modify the one-time pad encryption to generate its n^c-length random key by applying a secure pseudorandom generator with stretch n^c to a shorter key of length n, then a polynomial-time eavesdropper would not be able to tell the difference. To see this, assume there is an adversary A that can predict a bit of the plaintext with probability noticeably larger than $1/2$, thus violating the computational security requirement (3). Then because such prediction is impossible when the key is truly random (see Exercise 9.3), A can be used to distinguish between a pseudorandom and truly random key, thus contradicting the security of the generator as per Definition 9.8. ■

9.3 Pseudorandom generators from one-way permutations

We will prove only the special case of Theorem 9.9 when the one-way function is a permutation:

Lemma 9.10 *Suppose that there exists a one-way function $f : \{0,1\}^* \to \{0,1\}^*$ such that f is one-to-one for every $x \in \{0,1\}^*$, $|f(x)| = |x|$. Then, for every $c \in \mathbb{N}$, there exists a secure pseudorandom generator with stretch n^c.* ♢

The proof of Lemma 9.10 does demonstrate some of the ideas behind the proof of the more general Theorem 9.9. Moreover, these ideas, including the *hybrid argument* and the *Goldreich-Levin Theorem*, are of independent interest and had found several applications in other areas of Computer Science.

9.3.1 Unpredictability implies pseudorandomness

To prove Lemma 9.10 it will be useful to have the following alternative characterization of pseudorandom generators. Historically, this definition was the original definition proposed for the notion of *pseudorandom generator* and the proof that it is equivalent to Definition 9.8 was a major discovery.

Let $G : \{0,1\}^* \to \{0,1\}^*$ be a polynomial-time computable function with stretch $\ell(n)$ (i.e., $|G(x)| = \ell(|x|)$ for every $x \in \{0,1\}^*$). We call G *unpredictable* if for every probabilistic polynomial-time B there is a negligible function $\epsilon : \mathbb{N} \to [0,1]$ such that

$$\Pr_{\substack{x \in_{\mathrm{R}} \{0,1\}^n \\ y = G(x) \\ i \in_{\mathrm{R}} [\ell(n)]}}[B(1^n, y_1, \ldots, y_{i-1}) = y_i] \le 1/2 + \epsilon(n)\,. \qquad (4)$$

In other words, predicting the ith bit given the first $i-1$ bits (where i is a randomly chosen index) is difficult for every polynomial-time algorithm.

Clearly, if G is a pseudorandom generator then it is also unpredictable. Indeed, if $y_1, \ldots, y_{\ell(n)}$ were uniformly chosen bits then it would be impossible to predict y_i given $y_1, \ldots, y_{i-1}$, and hence if such a predictor exists when $y = G(x)$ for a random x, then the predictor can distinguish between the distribution $U_{\ell(n)}$ and $G(U_n)$. Interestingly, the converse direction also holds:

Theorem 9.11 *(Unpredictability implies pseudorandomness* [Yao82a]*)*
Let $\ell : \mathbb{N} \to \mathbb{N}$ be some polynomial-time computable function, and $G : \{0,1\}^* \to \{0,1\}^*$ be a polynomial-time computable function such that $|G(x)| = \ell(|x|)$ for every $x \in \{0,1\}^*$. If G is unpredictable then it is a secure pseudorandom generator. Moreover, for every probabilistic polynomial-time algorithm A, there exists a probabilistic polynomial-time B such that for every $n \in \mathbb{N}$ and $\epsilon > 0$, if $\Pr[A(G(U_n)) = 1] - \Pr[A(U_{\ell(n)}) = 1] \geq \epsilon$, then

$$\Pr_{\substack{x \in_R \{0,1\}^n \\ y = G(x) \\ i \in_R [\ell(n)]}}[B(1^n, y_1, \ldots, y_{i-1}) = y_i] \geq 1/2 + \epsilon/\ell(n)$$

Proof: First, note that the main result does follow from the "moreover" part. Indeed, suppose that G is not a pseudorandom generator and hence there is some algorithm A and constant c such that

$$\left|\Pr[A(G(U_n)) = 1] - \Pr[A(U_{\ell(n)}) = 1]\right| \geq n^{-c} \tag{5}$$

for infinitely many n's. Then we can ensure (perhaps by changing A to the algorithm $1 - A$ that flips the one-bit answer of A), that for infinitely many n's, (5) holds without the absolute value. For every such n, we'll get a predictor B that succeeds with probability $1/2 + n^{-c}/\ell(n)$, contradicting the unpredictability property.

We turn now to proving this "moreover" part. Let A be some probabilistic polynomial-time algorithm that is supposedly more likely to output 1 on input from the distribution $G(U_n)$ than on input from $U_{\ell(n)}$. Our algorithm B will be quite simple: on input 1^n, $i \in [\ell(n)]$ and $y_1, \ldots, y_{i-1}$, Algorithm B will choose $z_i, \ldots, x_{\ell(n)}$ independently at random, and compute $a = A(y_1, \ldots, y_{i-1}, z_i, \ldots, z_{\ell(n)})$. If $a = 1$ then B surmises its guess for z_i is correct and outputs z_i; otherwise it outputs $1 - z_i$.

Let $n \in \mathbb{N}$ and $\ell = \ell(n)$ and suppose that $\Pr[A(G(U_n)) = 1] - \Pr[A(U_{\ell(n)}) = 1] \geq \epsilon$. We'll show that

$$\Pr_{\substack{x \in_R \{0,1\}^n \\ y = G(x) \\ i \in_R [\ell]}}[B(1^n, y_1, \ldots, y_{i-1}) = y_i] \geq 1/2 + \epsilon/\ell\,. \tag{6}$$

To analyze B's performance, we define the following ℓ distributions $\mathcal{D}_0, \ldots, \mathcal{D}_\ell$ over $\{0,1\}^\ell$. (This technique is called the *hybrid argument*.) For every i, the distribution $\mathcal{D}_i$ is obtained as follows: choose $x \in_R \{0,1\}^n$ and let $y = G(x)$, output $y_1, \ldots, y_i, z_{i+1}, \ldots, z_\ell$, where $z_{i+1}, \ldots, z_\ell$ are chosen independently at random in $\{0,1\}$. Note that $\mathcal{D}_0 = U_\ell$ while $\mathcal{D}_\ell = G(U_n)$. For every $i \in \{0,..,\ell\}$, define $p_i = \Pr[A(\mathcal{D}_i) = 1]$. Note that $p_\ell - p_0 \geq \epsilon$. Thus, writing

$$p_\ell - p_0 = (p_\ell - p_{\ell-1}) + (p_{\ell-1} - p_{\ell-2}) + \cdots + (p_1 - p_0)\,,$$

we get that $\sum_{i=1}^{\ell}(p_i - p_{i-1}) \geq \epsilon$ or in other words, $\mathsf{E}_{i \in [\ell]}[p_i - p_{i-1}] \geq \epsilon/\ell$. We will prove (6) by showing that for every i,

$$\Pr_{\substack{x \in_R \{0,1\}^n \\ y = G(x)}}[B(1^n, y_1, \ldots, y_{i-1}) = y_i] \geq 1/2 + (p_i - p_{i-1})\,.$$

Recall that B makes a guess z_i for y_i and invokes A to obtain a value a, and then outputs z_i if $a = 1$ and $1 - z_i$ otherwise. Thus B predicts y_i correctly if either $a = 1$ and $y_i = z_i$ or

$a \neq 1$ and $y_i = 1 - z_i$, meaning that the probability this event happens is

$$\tfrac{1}{2} \Pr[a = 1 | z_i = y_i] + \tfrac{1}{2}(1 - \Pr[a = 1 | z_i = 1 - y_i]) . \tag{7}$$

Yet, one can verify that conditioned on $z_i = y_i$, B invokes A with the distribution $\mathcal{D}_i$, meaning that $\Pr[a = 1 | z_i = y_i] = p_i$. On the other hand if we don't condition on z_i then the distribution B invokes A is equal to $\mathcal{D}_{i-1}$. Hence,

$$p_{i-1} = \Pr[a = 1] = \tfrac{1}{2} \Pr[a = 1 | z_i = y_i] + \tfrac{1}{2} \Pr[a = 1 | z_i = 1 - y_i] = \tfrac{1}{2} p_i + \tfrac{1}{2} \Pr[a = 1 | z_i = 1 - y_i] .$$

Plugging this into (7) we get that B predicts y_i with probability $\tfrac{1}{2} + p_i - p_{i-1}$. ■

9.3.2 Proof of Lemma 9.10: The Goldreich-Levin Theorem

Let f be some one-way permutation. To prove Lemma 9.10 we need to use f to come up with a pseudorandom generator with arbitrarily large polynomial stretch $\ell(n)$. It turns out that the crucial step is obtaining a pseudorandom generator that extends its input by one bit (i.e., has stretch $\ell(n) = n + 1$). This is achieved by the following theorem:

Theorem 9.12 *(Goldreich-Levin Theorem* [GL89]*)*
Suppose that $f : \{0,1\}^* \to \{0,1\}$ is a one-way function such that f is one-to-one and $|f(x)| = |x|$ for every $x \in \{0,1\}^*$. Then, for every probabilistic polynomial-time algorithm A there is a negligible function $\epsilon : \mathbb{N} \to [0,1]$ such that

$$\Pr_{x,r \in_R \{0,1\}^n} [A(f(x), r) = x \odot r] \leq \tfrac{1}{2} + \epsilon(n) ,$$

where $x \odot r$ is defined to be $\sum_{i=1}^n x_i r_i \pmod 2$.

Theorem 9.12 immediately implies that the function $G(x, r) = f(x), r, x \odot r$ is a secure pseudorandom generator that extends its input by one bit (mapping $2n$ bits into $2n+1$ bits). Indeed, otherwise by Theorem 9.11 there would be a predictor B for this function. But because f is a permutation over $\{0,1\}^n$, the first $2n$ bits of $G(U_{2n})$ are completely random and independent, and hence cannot be predicted from their predecessors with probability better than $\tfrac{1}{2}$. This means that a predictor for this function would have to succeed at predicting the $2n+1^{th}$ bit from the previous $2n$ bits with probability noticeably larger than $\tfrac{1}{2}$, which exactly amounts to violating Theorem 9.12.

Proof of Theorem 9.12: Suppose, for the sake of contradiction, that there is some probabilistic polynomial-time algorithm A that violates the theorem's statement. We'll use A to show a probabilistic polynomial-time algorithm B that inverts the permutation f, in contradiction to the assumption that it is one way. Specifically, we will show that if for some n,

$$\Pr_{x,r \in_R \{0,1\}^n} [A(f(x), r) = x \odot r] \geq \tfrac{1}{2} + \epsilon , \tag{8}$$

then B will run in $O(n^2/\epsilon^2)$ time and invert the one-way permutation f on inputs of length n with probability at least $\Omega(\epsilon)$. This means that if A's success probability is more than $\tfrac{1}{2} + n^{-c}$ for some constant c and infinitely many n's, then B runs in polynomial-time and inverts the one-way permutation with probability at least $\Omega(n^{-c})$ for infinitely many n's.

Let n, ϵ be such that (8) holds. Then by a simple averaging argument, for at least an $\epsilon/2$ fraction of the x's, the probability over r that $A(f(x), r) = x \odot r$ is at least $\tfrac{1}{2} + \epsilon/2$. We'll call such x's *good*, and show an algorithm B that with high probability inverts $f(x)$ for every good x.

To restate the scenario here (and point out its connection to the program checking idea introduced in Chapter 8, which came later historically speaking) is that we are given a "black box" that computes an unknown linear function $x \mapsto x \odot r$ for at least $1/2 + \epsilon/2$ fraction of r's, and we have to give an efficient algorithm that runs in $\text{poly}(|X| + 1/\epsilon)$ time that reconstructs x.

As a warm-up, note that if $\Pr_r[A(f(x), r) = x \odot r] = 1$, then it is easy to recover x from $f(x)$: just run $A(f(x), e^1), \ldots, A(f(x), e^n)$ where e^i is the string whose i^{th} coordinate is equal to one and all the other coordinates are zero. Clearly, $x \odot e^i$ is the i^{th} bit of x, and hence by making these n calls to A we can recover x completely. Of course, this idea breaks down if $\Pr_r[A(f(x), r) = x \odot r]$ is less than 1. Below, we first describe a simpler reconstruction algorithm that works when this probability is 0.9. The more general algorithm extends this simpler algorithm.

Recovery for success probability 0.9: Now suppose that for an $\Omega(\epsilon)$ fraction of x's, we had $\Pr_r[A(f(x), r) = x \odot r] \geq 0.9$. For such an x, we cannot trust that $A(f(x), e^i) = x \odot e^i$, since it may be that $e^1, \ldots, e^n$ are among the $2^n/10$ strings r on which A answers incorrectly. Still, there is a simple way to bypass this problem: if we choose $r \in_{\text{R}} \{0,1\}^n$ then the string $r \oplus e^i$ is also uniformly distributed. Hence by the union bound,

$$\Pr_r[A(f(x), r) \neq x \odot r \text{ or } A(f(x), r \oplus e^i) \neq x \odot (r \oplus e^i)] \leq 0.2\,.$$

But $x \odot (r \oplus e^i) = (x \odot r) \oplus (x \odot e^i)$, which means that if we choose r at random, and compute $z = A(f(x), r)$ and $z' = A(f(x), r \odot e^i)$, then $z \oplus z'$ will be equal to the i^{th} bit of x with probability at least 0.8. To obtain *every* bit of x, we amplify this probability to $1 - 1/(10n)$ by taking majorities. Specifically, we use the following algorithm:

Algorithm B:

1. Choose $r^1, \ldots, r^m$ independently at random from $\{0,1\}^n$ (we'll specify m shortly).
2. For every $i \in [n]$:
   - Compute the values $z_1 = A(f(x), r^1), z'_1 = A(f(x), r^1 \odot e^i), \ldots, z_m = A(f(x), r^m), z'_m = A(f(x), r^m \oplus e^i)$.
   - Guess that x_i is the majority value among $\{z_j \oplus z'_j\}_{j \in [m]}$.

We claim that if $m = 200n$ then for every $i \in [n]$, the majority value will be correct with probability at least $1 - 1/(10n)$ (and hence B will recover *every* bit of x with probability at least 0.9). To prove the claim, we define the random variable Z_j to be 1 if both $A(f(x), r^j) = x \odot r^j$ and $A(f(x), r^j \oplus e^i) = x \odot (r^j \oplus e^i)$; otherwise $Z_j = 0$. Note that the variables $Z_1, \ldots, Z_m$ are independent and by our previous discussion $\mathsf{E}[Z_j] \geq 0.8$ for every j. It suffices to show that with probability $1 - 1/(10n)$, more than $m/2$ of the Z_j's are equal to 1. In other words, letting $Z = Z_1 + \ldots Z_m$, it suffices to show that $\Pr[Z \leq m/2] \leq 1/(10n)$. But, since $\mathsf{E}[Z] = \sum_j \mathsf{E}[Z_j] \geq 0.8m$, all we need to do is bound $\Pr[|Z - E[Z]| \geq 0.3m]$. By Chebychev's Inequality (Lemma A.12),[6]

$$\Pr\left[|Z - E[Z]| \geq k\sqrt{\mathsf{Var}(Z)}\right] \leq 1/k^2\,.$$

In our case, because the Z_j's are independent 0/1 random variables, $\mathsf{Var}(Z) = \sum_{j=1}^m \mathsf{Var}(Z_j)$ and $\mathsf{Var}(Z_j) \leq 1$ for every j, implying that

$$\Pr[|Z - E[Z]| \geq 0.3m] \leq \tfrac{1}{(0.3\sqrt{m})^2}\,,$$

which is smaller than $1/(10n)$ by our choice of $m = 200n$.

[6]We could have gotten an even better bound using the Chernoff Inequality, but this analysis is easier to extend to the general case of lower success probability.

Recovery for success probability $1/2 + \epsilon/2$: The above analysis crucially used the unrealistic assumption that for many x's, $A(f(x), r)$ is correct with probability at least 0.9 over r. It's not hard to see that once this probability falls below 0.75, that analysis breaks down, since we no longer get any meaningful information by applying the union bound on the events $A(f(x), r) = x \odot r$ and $A(f(x), r \oplus e^i) = x \odot (r \oplus e^i)$. Unfortunately, in general our only guarantee is that if x is good then this probability is at least $1/2 + \epsilon/2$ (which could be much smaller than 0.75). The crucial insight needed to extend the proof is that all of the above analysis would still carry over even if the strings $r^1, \ldots, r^m$ are only chosen to be *pairwise independent* as opposed to fully independent. Indeed, the only place where we used independence is to argue that the random variables $Z_1, \ldots, Z_m$ satisfy $\mathsf{Var}(\sum_j Z_j) = \sum_j \mathsf{Var}(Z_j)$ and this condition holds also for pairwise independent random variables (see Claim A.13).

We'll now show how to pick $r^1, \ldots, r^m$ in a pairwise indpendent fashion in such a way that we "know" each $x \odot r^i$ already. This may seem ridiculous since x is unknown, and indeed the catch is that we can do it only thanks to some exhaustive guessing, to be made clear soon. Set k such that $m \leq 2^k - 1$ and do as follows:

1. Choose k strings $s^1, \ldots, s^k$ independently at random from $\{0,1\}^n$.
2. For every $j \in [m]$, we associate a unique nonempty set $T_j \subseteq [k]$ with j in some canonical fashion and define $r^j = \sum_{t \in T_j} s^t \pmod 2$. That is, r^j is the XOR of all the strings among $s^1, \ldots, s^k$ that belong to the j^{th} set.

It can be shown that the strings $r^1, \ldots, r^m$ are pairwise independent (see Exercise 8.4). Moreover, for every $x \in \{0,1\}^n$, $x \odot r^j = \sum_{t \in T_j} x \odot s^t$. This means that if we know the k values $x \odot s^1, \ldots, x \odot s^k$ then we can deduce the m values $x \odot r^1, \ldots x \odot r^m$. This is where exhaustive guessing comes in. Since $2^k = O(m)$, we can enumerate over *all possible guesses* for $x \odot s^1, \ldots, x \odot s^k$ in polynomial time. This leads us to the following algorithm B' to invert $f(\cdot)$:

Algorithm B':

Input: $y \in \{0,1\}^n$, where $y = f(x)$ for an unknown x.

We assume that x is "good" and hence $\Pr_r[A(f(x), r) = x \odot r] \geq 1/2 + \epsilon/2$. (We don't care how B performs on x's that are not good.)

Operation: Let $m = 200n/\epsilon^2$ and k be the smallest such that $m \leq 2^k - 1$. Choose $s^1, \ldots, s^k$ independently at random in $\{0,1\}^k$, and define $r^1, \ldots, r^m$ as above. For every string $w \in \{0,1\}^k$ do the following:

- Run the algorithm B from above under the assumption that $x \odot s^t = w_t$ for every $t \in [k]$. That is, for every $i \in [n]$, we compute our guess z_j for $x \odot r^j$ by setting $z_j = \sum_{t \in T_j} w_t$. We compute the guess z'_j for $x \odot (r_j \oplus e^i)$ as before by setting $z'_j = A(y, r^j \oplus e^i)$.
- As before, for every $i \in [n]$, our guess for x_i is the majority value among $\{z_j \oplus z'_j\}_{j \in [m]}$.
- We test whether our guess for $x = x_1, \ldots, x_n$ satisfies $f(x) = y$. If so, we halt and output x.

The analysis is almost identical to the previous case. In one of the 2^k iterations we will guess the correct values $w_1, \ldots, w_k$ for $x \odot s^1, \ldots, x \odot s^k$. We'll show that in this particular iteration, for every $i \in [n]$ Algorithm B' guesses x_i correctly with probability at least $1 - 1/(10n)$. Indeed, fix some $i \in [n]$ and define the random variables $Z_1, \ldots, Z_m$ as we did before: Z_j is a 0/1 variable that equals 1 if both $z_j = x \odot r^j$ and $z'_j = x \odot (r^j \oplus e^i)$. In the iteration where we chose the right values $w_1, \ldots, w_k$, it always holds that $z_j = x \odot r^j$ and hence Z_j depends only on the second event, which happens with probability at least

$1/2 + \epsilon/2$. Thus, all that is needed is to show that for $m = 100n/\epsilon^2$, if $Z_1, \ldots, Z_m$ are pairwise independent 0/1 random variables, where $\mathsf{E}[Z_j] \geq 1/2 + \epsilon/2$ for every j, then $\Pr[\sum_j Z_j \leq m/2] \leq 1/(10n)$. But this follows immediately from Chebychev's Inequality. ■

Getting arbitrarily large expansion

Theorem 9.12 provides us with a secure pseudorandom generator of stretch $\ell(n) = n+1$, but to complete the proof of Lemma 9.10 (and to obtain useful encryption schemes with short keys) we need to show a generator with arbitrarily large polynomial stretch. This is achieved by the following theorem:

Theorem 9.13 *If f is a one-way permutation and $c \in \mathbb{N}$, then the function G that maps $x, r \in \{0,1\}^n$ to $r, f^\ell(x) \odot r, f^{\ell-1}(x) \odot r, \cdots, f^1(x) \odot r$, where $\ell = n^c$ is a secure pseudorandom generator of stretch $\ell(2n) = n + n^c$. (f^i denotes the function obtained by applying the function f i times to the input.)* ♢

Proof: By Yao's theorem (Theorem 9.11), 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,

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

We will show a probabilistic polynomial-time algorithm B that on such n's will predict $x \odot r$ from $f(x), r$ with probability at least $1/2 + \epsilon$. Thus, if A has non-negligible success then B violates Theorem 9.12.

Algorithm B is given r and y such that $y = f(x)$ for some x. It will then pick $i \in \{1, \ldots, N\}$ randomly, and compute the values $f^{\ell-i}(y), \ldots, f(y)$ and output $a = A(r, f^{\ell-i-1}(y) \odot r, \ldots, f(y) \odot r, y \odot r)$. Because f is a permutation, this is exactly the same distribution obtained where we choose $x' \in_R \{0,1\}^n$ and set $x = f^i(x')$, and hence A will predict $f^i(x') \odot r$ with probability $1/2 + \epsilon$, meaning that B predicts $x \odot r$ with the same probability. ■

9.4 Zero knowledge

Normally we think of a proof as presenting the evidence that some statement is true, and hence typically after carefully reading and verifying a proof for some statement, you learn much more than the mere fact that the statement is true. But does it have to be this way? For example, suppose that you figured out how to schedule all of the flights of some airline in a way that saves them millions of dollars. You want to *prove* to the airline that there exists such a schedule, without actually *revealing* the schedule to them (at least not before you receive your well-deserved payment). Is this possible?

A similar scenario arises in the context of authentication— suppose a company has a sensitive building, that only a select group of employees is allowed to enter. One way to enforce this is to choose two random prime numbers P and Q and reveal these numbers to the selected employees, while revealing $N = P \cdot Q$ to the guard outside the building. The guard will be instructed to let inside only a person demonstrating knowledge of N's factorization. But is it possible to demonstrate such knowledge without revealing the factorization?

It turns out this is in fact possible to do, using the notion of *zero knowledge proof.* Zero knowledge proofs are interactive probabilistic proof systems, just like the systems we encountered in Chapter 8. However, in addition to the *completeness* property (prover can convince the verifier to accept with high probability) and *soundness* property (verifier will reject false statements with high probability), we require an additional *zero knowledge* property, that roughly speaking, requires that the verifier does not learn anything from the interaction apart from the fact that the statement is true. That is, zero knowledge requires that *whatever the verifier learns after participating in a proof for a statement x, she could* *have computed by herself, without participating in any interaction.* Below we give the formal definition for zero knowledge proofs of **NP** languages. (One can define zero knowledge also for languages outside **NP**, but the zero knowledge condition makes it already highly non-trivial and very useful to obtain such proof systems even for languages in **NP**.)

Definition 9.14 *(Zero knowledge proofs)*
Let L be an **NP**-language, and let $p(\cdot), M$ be a polynomial and Turing machine that demonstrate this. That is, $x \in L \Leftrightarrow \exists_{u\in\{0,1\}^{p(|x|)}}$ s.t. $M(x,y) = 1$.
A pair P, V of interactive probabilistic polynomial-time algorithms is called a *zero knowledge proof* for L, if the following three condition hold:

Completeness: For every $x \in L$ and u a certificate for this fact (i.e., $M(x,u) = 1$), $\Pr[\mathsf{out}_V\langle P(x,u), V(x)\rangle] \geq 2/3$, where $\langle P(x,u), V(x)\rangle$ denotes the interaction of P and V where P gets x, u as input and V gets x as input, and $\mathsf{out}_V I$ denotes the output of V at the end of the interaction I.

Soundness: If $x \notin L$, then for *every* strategy P^* and input u, $\Pr[\mathsf{out}_V\langle P^*(x,u), V(x)\rangle] \leq 1/3$. (The strategy P^* needs not run in polynomial time.)

Perfect Zero Knowledge: For every probabilistic polynomial-time interactive strategy V^*, there exists an expected probabilistic polynomial-time (stand-alone) algorithm S^* such that for every $x \in L$ and u a certificate for this fact,

$$\mathsf{out}_{V^*}\langle P(x,u), V^*(x)\rangle \equiv S^*(x)\,. \tag{9}$$

(That is, these two random variables are identically distributed.) This algorithm S^* is called the *simulator* for V^*, as it simulates the outcome of V^*'s interaction with the prover without any access to such an interaction.

The zero knowledge condition means that the verifier cannot learn anything new from the interaction, even if she does not follow the protocol but rather uses some other strategy V^*. The reason is that she could have learned the same thing by just running the stand-alone algorithm S^* on the publicly known input x. The perfect zero knowledge condition can be relaxed by requiring that the distributions in (9) have small *statistical distance* (see Section A.2.6) or are *computationally indistinguishable* (see Exercise 9.17). The resulting notions are called respectively *statistical zero knowledge* and *computational zero knowledge* and are central to cryptography and complexity theory. The class of languages with statistical zero knowledge proofs, known as **SZK**, has some fascinating properties, and is believed to lie strictly between **P** and **NP** (see [Vad99] for an excellent survey). In contrast, it is known ([GMW86], see also [BCC86]) that if one-way functions exist then *every* **NP** language has a computational zero knowledge proof, and this result has significant applications to the design of cryptographic protocols (see the chapter notes).

The idea of using *simulation* to demonstrate security is also central to many aspects of cryptography. Asides from zero knowledge, it is used in the definition of semantic security for encryptions (see Exercise 9.9), secure multiparty computation (Section 9.5.4) and many other settings. In all these cases security is defined as the condition that an attacker cannot learn or do anything that she could not have done in an idealized and "obviously secure" setting (e.g., in encryption in the ideal setting the attacker doesn't see even the ciphertext, while in zero knowledge in the ideal setting there is no interaction with the prover).

Example 9.15
We show a perfect zero knowledge proof for the language GI of graph isomorphism. The language GI is in **NP** and has a trivial proof satisfying completeness and soundness— send the isomorphism to the verifier. But that proof is not known to be zero knowledge, since we do not know of a polynomial-time algorithm that can find the isomorphism between two given isomorphic graphs.

Zero-knowledge proof for Graph Isomorphism:

Public input: A pair of graphs G_0, G_1 on n vertices. (For concreteness, assume they are represented by their adjacency matrices.)

Prover's private input: A permutation $\pi : [n] \to [n]$ such that $G_1 = \pi(G_0)$, where $\pi(G)$ denotes the graph obtained by transforming the vertex i into $\pi(i)$ (or equivalently, applying the permutation π to the rows and columns of G's adjacency matrix).

Prover's first message: Prover chooses a random permutation $\pi_1 : [n] \to [n]$ and sends to the verifier the adjacency matrix of $\pi_1(G_1)$.

Verifier's message: Verifier chooses $b \in_{\text{R}} \{0, 1\}$ and sends b to the prover.

Prover's last message: If $b = 1$, the prover sends π_1 to the verifier. If $b = 0$, the prover sends $\pi_1 \circ \pi$ (i.e., the permutation mapping n to $\pi_1(\pi(n))$) to the verifier.

Verifier's check: Letting H denote the graph received in the first message and π the permutation received in the last message, the verifier accepts if and only if $H = \pi(G_b)$.

Clearly, if both the prover and verifier follow the protocol, then the verifier will accept with probability one. For soundness, we claim that if G_0 and G_1 are *not* isomorphic, then the verifier will reject with probability at least $1/2$ (this can be reduced further by repetition). Indeed, in that case regardless of the prover's strategy, the graph H that he sends in his first message cannot be isomorphic to both G_0 and G_1, and there has to exist $b \in \{0, 1\}$ such that H is not isomorphic to G_b. But the verifier will choose this value b with probability $1/2$, and then the prover will not be able to find a permutation π such that $H = \pi(G_b)$, and hence the verifier will reject.

Let V^* be some verifier strategy. To show the zero knowledge condition, we use the following simulator S^*: On input a pair of graphs G_0, G_1, the simulator S^* chooses $b' \in_{\text{R}} \{0, 1\}$, a random permutation π on $[n]$ and computes $H = \pi(G_{b'})$. It then feeds H to the verifier V^* to obtain its message $b \in \{0, 1\}$. If $b = b'$ then S^* sends π to V^* and outputs whatever V^* outputs. Otherwise (if $b \neq b'$) the simulator S^* restarts from the beginning.
The crucial observation is that S^*'s first message is distributed in exactly the same way as the prover's first message— a random graph that is isomorphic to G_0 and G_1. This also means that H reveals nothing about the choice of b', and hence the probability that $b' = b$ is $1/2$. If this happens, then the messages H and π that V^* sees are distributed identically to the distribution of messages that it gets in a real interaction with the prover. Because S^* succeeds in getting $b' = b$ with probability $1/2$, the probability it needs k iterations is 2^{-k}, which means that its expected running time is $T(n)\sum_{k=1}^{\infty} 2^{-k} = O(T(n))$, where $T(n)$ denotes the running time of V^*. Thus, S^* runs in expected probabilistic polynomial-time.[7]

9.5 Some applications

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

[7]In Chapter 18 we will see a stricter notion of expected probabilistic polynomial-time (see Definition 18.4). This simulator satisfies this stricter notion as well.

9.5.1 Pseudorandom functions

Pseudorandom functions are a natural generalization of pseudorandom generators. This is a family of functions that although are efficiently computable and have a polynomial-size representation (and hence are far from being random), are indistinguishable from random functions to an observer with input/output access to the function. This is of course reminiscent of the definition of a pseudorandom generator, whose output also has to pass a "blind test" versus a truly random string. The difference here is that the object being talked about is a function, whose truth table has exponential size. Hence the distinguishing algorithm only has the ability to ask for the value of the function at any inputs of its choosing.

Definition 9.16 Let $\{f_k\}_{k\in\{0,1\}^*}$ be a family of functions such that $f_k : \{0,1\}^{|k|} \to \{0,1\}^{|k|}$ for every $k \in \{0,1\}^*$, and there is a polynomial-time algorithm that computes $f_k(x)$ given $k \in \{0,1\}^*, x \in \{0,1\}^{|k|}$. We say that the family is *pseudorandom* if for every probabilistic polynomial-time oracle[8] Turing machine A there is a negligible function $\epsilon : \mathbb{N} \to [0,1]$ such that

$$\left| \Pr_{k\in_R\{0,1\}^n}\left[A^{f_k(\cdot)}(1^n) = 1\right] - \Pr_{g\in_R\mathcal{F}_n}\left[A^g(1^n) = 1\right] \right| < \epsilon(n)$$

for every n, where $\mathcal{F}_n$ denotes the set of all functions from $\{0,1\}^n$ to $\{0,1\}^n$. ♢

One can verify that if $\{f_k\}$ is a pseudorandom function family, then for every polynomial $\ell(n)$, the function G that maps $k \in \{0,1\}^n$ to $f_k(1),\ldots,f_k(\ell(n))$ (where we use some canonical encoding of the numbers $1,\ldots,\ell(n)$ as strings in $\{0,1\}^n$) is a secure pseudorandom generator. Thus, pseudorandom functions imply the existence of secure pseudorandom generators of arbitrary polynomial stretch. It turns out that the converse is true as well:

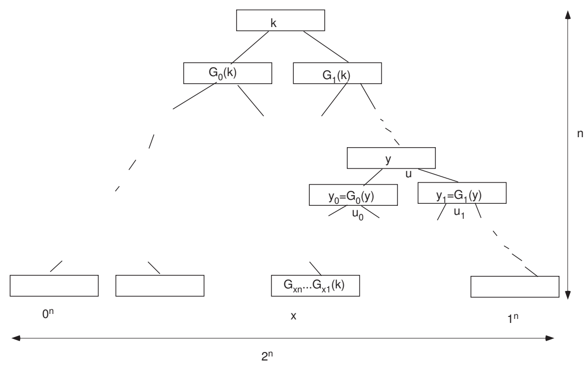

Figure 9.2 The pseudorandom function $f_k(x)$ outputs the label of the x^{th} node in a depth n binary tree where the root is labeled by k, and the children u_0, u_1 of every node u labeled y are labeled by $G_0(y)$ and $G_1(y)$.

Theorem 9.17 ([GGM84]) *Suppose that there exists a secure pseudorandom generator G with stretch $\ell(n) = 2n$. Then there exists a pseudorandom function family.* ♢

PROOF: Let G be a secure pseudorandom generator as in the theorems statement mapping length-n strings to length-$2n$ strings. For every $x \in \{0,1\}^n$, we denote by $G_0(x)$ the first n bits of $G(x)$, and by $G_1(x)$ the last n bits of $G(x)$. For every $k \in \{0,1\}^n$ we will define the function $f_k(\cdot)$ as follows:

$$f_k(x) = G_{k_n}(G_{k_{n-1}}(\cdots(G_{k_1}(x))\cdots) \qquad (10)$$

for every $x \in \{0,1\}^n$. Note that $f_k(x)$ can be computed by making n invocations of G, and hence clearly runs in polynomial time. Another way to view f_k is given in Figure 9.2—think of a full depth n binary tree whose root is labeled by k, and where we label the two

[8]See Section 3.4 for the definition of oracle Turing machines.

children of a vertex labeled by y with the values $G_0(y)$ and $G_1(y)$ respectively. Then, $f_k(x)$ denotes the label of the x^{th} leaf of this tree. Of course, actually writing the tree down would take exponential time and space, but as is shown by (10), we can compute the label of each leaf in polynomial time by following the length n path from the root to this leaf.

Why is this function family pseudorandom? We'll show this by transforming a T-time algorithm A that distinguishes between f_{U_n} and a random function with bias ϵ into a poly$(n)T$-time algorithm B that distinguishes between U_{2n} and $G(U_n)$ with bias $\epsilon/(nT)$.

Assume without loss of generality that A makes exactly T queries to its oracle (we can ensure that by adding superfluous queries). Now, we can implement an oracle $\mathcal{O}$ to f_{U_n} in the following way: the oracle $\mathcal{O}$ will label vertices of the depth n full binary tree as needed. Initially, only the root is labeled by a random string k. Whenever, a query of A requires the oracle to label the children u_0, u_1 of a vertex v labeled by y, the oracle will invoke G on y to obtain $y_0 = G_0(y)$ and $y_1 = G_1(y)$ and then label u_0, u_1 with y_0, y_1 respectively and delete the label y of u. Note that indeed, once u_0 and u_1 are labeled, we have no further need for the label of u. Following the definition of f_k, the oracle $\mathcal{O}$ answers a query x with the label of the x^{th} vertex. Note that $\mathcal{O}$ invokes the generator G at most Tn times. By adding superfluous invocations we can assume $\mathcal{O}$ invokes the generator exactly Tn times.

Now for every $i \in \{0, \ldots, Tn\}$ define the oracle $\mathcal{O}_i$ as follows: the oracle $\mathcal{O}_i$ follows the operation of $\mathcal{O}$, but for the first i invocations of G, instead of the labels y_0, y_1 of the children of a node labeled y by setting $y_0 = G_0(y)$ and $y_1 = G_1(y)$, the oracle $\mathcal{O}_i$ chooses both y_0 and y_1 independently at random from $\{0,1\}^n$. Note that $\mathcal{O}_0$ is the same as the oracle $\mathcal{O}$ to f_{U_n}, but $\mathcal{O}_{nT}$ is an oracle to a completely random function. Let $p_i = \Pr[A^{\mathcal{O}_i}(1^n) = 1]$. Then, as in the proof of Theorem 9.11, we may assume $p_{Tn} - p_0 \geq \epsilon$ and deduce that $\mathsf{E}_{i \in_R [Tn]}[p_i - p_{i-1}] \geq \epsilon/(Tn)$. Our algorithm B to distinguish U_{2n} from $G(U_n)$ will do as follows: on input $y \in \{0,1\}^{2n}$, choose $i \in_R [Tn]$ and execute A with access to the oracle $\mathcal{O}_{i-1}$, using random values for the first $i-1$ invocations of G. Then, in the i^{th} invocation use the value y instead of the result of invoking G. In all the rest of the invocations B runs G as usual, and at the end outputs what A outputs. One can verify that for every choice of i, if the input y is distributed as U_{2n} then B's output is distributed as $A^{\mathcal{O}_i}(1^n)$, while if it is distributed according to $G(U_n)$, B's output is distributed as $A^{\mathcal{O}_{i-1}}(1^n)$. ■

A pseudorandom function generator is a way to turn a random string $k \in \{0,1\}^n$ into an implicit description of an exponentially larger "random looking" string, namely, the table of all values of the function f_k. This has proved a powerful primitive in cryptography. For example, while we discussed encryption schemes for a single message, in practice we often want to encrypt many messages with the same key. Pseudorandom functions allow Alice and Bob to share an "exponentially large one-time pad". That is, Alice and Bob can share a key $k\{0,1\}^n$ of a pseudorandom function, and whenever she wants to encrypt a message $x \in \{0,1\}^n$ for Bob, Alice will choose $r \in_R \{0,1\}^n$, and send $(r, f_k(r) \oplus x)$. Bob can find x since he knows the key k, but for an adversary that does not know the key, it looks as if Alice sent two random strings (as long as she doesn't choose the same string r to encrypt two different messages, but this can only happen with exponentially small probability). Pseudorandom functions are also used for *message authentication codes*. If Alice and Bob share a key k of a pseudorandom function, then when Alice sends a message x to Bob, she can append the value $f_k(x)$ to this message. Bob can verify that the pair (x, y) he receives satisfies $y = f_k(x)$. An adversary Eve that controls the communication line between Alice and Bob cannot change the message x to x' without being detected, since the probability that Eve can predict the value of $f_k(x')$ is negligible (after all, a random function is unpredictable). Furthermore, pseudorandom function generators have also figured in a very interesting explanation of why current lower bound techniques have been unable to separate **P** from **NP**; see Chapter 23.

9.5.2 Derandomization

The existence of pseudorandom generators implies subexponential deterministic algorithms for **BPP**: this is usually referred to as *derandomization* of **BPP**. That is, if $L \in \mathbf{BPP}$ then for every $\epsilon > 0$ there is a 2^{n^ϵ}-time deterministic algorithm A such that for every sampleable distribution of inputs $\{X_n\}$ where $X_n \in \{0,1\}^n$, $\Pr[A(X_n) = L(X_n)] > 0.99$. (Note that the randomness is only over the choice of the inputs— the algorithm A is deterministic.) The algorithm A works by simply reducing the randomness of the probabilistic algorithm for L to n^ϵ using a pseudorandom generator, and then enumerating over all the possible inputs for the pseudorandom generator. We will see stronger derandomization results for **BPP** in Chapter 20.

9.5.3 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. 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. Now B selects a random bit b and sends it to A. 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. (For this reason, we say that A's first message is a cryptographic *commitment* to the bit $x_A \odot r_A$.) On the other hand, by Theorem 9.12, B cannot predict $x_A \odot r_A$ from A's first message, and so cannot bias her bit according to the choice of $x_A \odot r_A$.

9.5.4 Secure multiparty computations

This concerns a vast generalization of the setting in Section 9.5.3. 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 9.5.3 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 zero knowledge and says that whatever a party or a coalition of parties learn during the protocol can be simulated in an ideal setting where they only get to send their inputs to some trusted authority that computes f on these inputs and broadcasts the result. Amazingly, there are protocols to achieve this task securely for every number of parties and for every polynomial-time computable f— see the chapter notes.[9]

[9]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. It is unclear if current privacy laws allow hospitals to perform such secure multiparty protocols using patient data— this an example of the law lagging behind scientific progress.

9.5.5 Lower bounds 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 to more restricted function families such as $\mathbf{NC}^1$ (see Kearns and Valiant [KV89]).

Chapter notes and history

We have chosen to model potential eavesdroppers, and hence also potential inverting algorithms for the one-way functions as probabilistic polynomial-time Turing machines. An equally justifiable choice is to model these as *polynomial-sized circuits* or, equivalently, probabilistic polynomial-time Turing machines that can have some input-length dependent polynomial-sized constants "hard-wired" into them as advice. All the results of this chapter hold for this choice as well, and in fact some proofs and definitions become slightly simpler. We chose to use uniform Turing machines to avoid making this chapter dependant on Chapter 6.

Goldreich's book [Gol04] is a good source for much of the material of this chapter (and more than that), while the undergraduate text [KL07] is a gentler introduction for the basics. For more coverage of recent topics, especially in applied cryptography, see Boneh and Shoup's upcoming book [BS08]. For more on computational number theory, see the books of Shoup [Sho05] and Bach and Shallit [BS96].

Kahn's book [Kah96] is an excellent source for the fascinating history of cryptography over the ages. Up until the mid 20th century, this history followed Edgar Alan Poe's quote in the chapter's start— every cipher designed and widely used was ultimately broken. Shannon [Sha49b] was the first to rigorously study the security of encryptions. He showed the results presented in Section 9.1, giving the first formal definition of security and showing that to satisfy it it's necessary and sufficient to have the key as large as the message. Shannon realized that computational difficulty is the way to bypass this bound, though he did not have a concrete approach how to do that. This is not surprising since the mathematical study of efficient computation (i.e., algorithm design and complexity theory) only really began in the 1960's, and with this study came the understanding of the dichotomy between polynomial time and exponential time.

Around 1974, Diffie and Hellman and independently Merkle began to question the age-old notion that secure communication requires sharing a secret key in advance. This resulted in the groundbreaking paper of Diffie and Hellman [DH76] that put forward the notion of *public key cryptography*. This paper also suggested the first implementation of this notion— what is known today as the *Diffie-Hellman key exchange* protocol, which also immediately yields a public key encryption scheme known today as El-Gamal encryption. But, to fully realize their agenda of both confidential and authenticated communication without sharing secret keys, Diffie and Hellman needed *trapdoor permutations* which they conjectured to exist but did not have a concrete implementation for.[10] The first construction for such trapdoor permutations was given by Rivest, Shamir, and Adleman [RSA78]. The resulting encryption and signature schemes were quite efficient and are still the most widely used such schemes today. Rivest et al conjectured that the security of their trapdoor permutation is equivalent to the factoring problem, though they were not able to prove it (and no proof has been found in the years since). Rabin [Rab79] later showed a trapdoor permutation that is in fact equivalent to the factoring problem.

Interestingly, similar developments also took place within the closed world of the intelligence community and in fact somewhat before the works of [DH76, RSA78], although this only came to light more than twenty years later [Ell99]. In 1970, James Ellis of the British intelligence agency

[10]Diffie and Hellman actually used the name "public key encryption" for the concept today known as trapdoor permutations. Indeed, trapdoor permutations can be thought of as a variant of public key encryptions with a deterministic (i.e., not probabilistic) encryption function. But following the work [GM82], we know that the use of probabilistic encryption is both essential for strong security, and useful to get encryption without using trapdoor permutations (as is the case in the Diffie-Hellman / El-Gamal encryption scheme).

GCHQ also realized that it might be possible to have secure encryption without sharing secret keys. No one in the agency had found a possible implementation for this idea until in 1973, Clifford Cocks suggested to use a trapdoor permutation that is a close variant of the RSA trapdoor permutation, and a few months later Malcolm Williamson discovered what we know today as the Diffie-Hellman key exchange. (Other concepts such as digital signatures, Rabin's trapdoor permutations, and public key encryption from the codes/lattices seem not to have been anticipated in the intelligence community.) Perhaps it is not very surprising that these developments happened in GCHQ before their discovery in the open literature, since between Shannon's work and the publication of [DH76], cryptography was hardly studied outside of the intelligence community.

Despite the well justified excitement they generated, the security achieved by the RSA and Diffie-Hellman schemes on their own was not fully satisfactory, and did not match the kind of security that Shannon showed the one-time pad can achieve in the sense of not revealing even partial information about the message. Goldwasser and Micali [GM82] showed how such strong security can be achieved, in a paper that was the basis and inspiration for many of the works that followed achieving strong notions of security for encryption and other tasks. Another milestone was reached by Goldwasser, Micali and Rivest [GMR84], who gave strong security definitions for digital signatures and showed how these can be realized under the assumption that integer factorization is hard.

Pseudorandom generators were used in practice since the early days of computing. Shamir [Sha81] was the first to explicitly connect intractability to pseudorandomness, by showing that if the RSA function is one-way then there exists a generator that can be proven to satisfy a certain weak pseudorandomness property (block unpredictability). Blum and Micali [BM82] defined the stronger notion of next bit unpredictability and showed a factoring-based generator satisfying it. Yao [Yao82a] defined the even stronger definition of pseudorandomness as fooling all polynomial-time tests (Definition 9.8), and proved that this notion is *equivalent* to next-bit unpredictability (Theorem 9.11). The Goldreich-Levin theorem was proven in [GL89], though we presented an unpublished proof due to Rackoff . Theorem 9.9 (pseudorandom generators from one-way functions) and its very technical proof is by Håstad, Impagliazzo, Luby and Levin [HILL99] (the relevant conference publications are a decade older). The construction of pseudorandom functions in Section 9.5.1 is due to Goldreich, Goldwasser, and Micali [GGM84].

Zero knowledge proofs were invented by Goldwasser, Micali and Rackoff [GMR85], who also showed a zero knowledge proof for problem of quadratic residuosity (see also Example 8.9). Goldreich, Micali and Wigderson [GMW86] showed that if one-way functions exist then there is a computational zero knowledge proof system for every language in **NP**. The zero knowledge protocol for graph isomorphism of Example 9.15 is also from the same paper. Independently, Brassard, Chaum and Crépeau [BCC86] gave a perfect zero knowledge argument for **NP** (where the soundness condition is computational, and the zero knowledge condition is with respect to unbounded adversaries), under a specific hardness assumption.

Yao [Yao82b] suggested the first protocol for realizing securely *any* two party functionality, as described in Section 9.5.4, but his protocol only worked for passive (also known as "eavesdropping" or "honest but curious") adversaries. Goldreich, Micali and Wigderson [GMW87] extended this result for every number of parties and also showed how to use zero knowledge proofs to achieve security also against *active* attacks, a paradigm that has been used many times since.

Some early cryptosystems were designed using the SUBSET SUM problem, but many of those were broken by the early 1980s. In the last few years, 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 [Ajt96], and a public-key cryptosystem described in Ajtai and Dwork [AD97] (and improved on since then by other researchers). These constructions are secure on *most* instances if and only if they are secure on *worst-case* instances. (The idea used is a variant of random self-reducibility.) Oded Regev's survey [Reg06] as well as his lecture notes (available from his home page) are a good source for more information on this fascinating topic (see also the older book [MG02]). The hope is that such ideas could eventually be used to base cryptography on *worst-case* type conjectures such as $\mathbf{P} \neq \mathbf{NP}$ or $\mathbf{NP} \cap \mathbf{coNP} \nsubseteq \mathbf{BPP}$, but there are still some significant obstacles to achieving this.

Much research has been devoted to exploring the exact notions of security that one needs for various cryptographic tasks. For instance, the notion of semantic security (see Section 9.2.2 and Exercise 9.9) may seem quite strong, but it turns out that for most applications it does not suffice and we need the stronger notion of *chosen ciphertext security* [RS91, DDN91]. See the Boneh-Shoup book [BS08] for more on this topic. Zero knowledge proofs play a central role in achieving security in such settings.

Exercises

9.1 Prove that the one-time pad encryption is perfectly secret as per Definition 9.1.

9.2 Prove that if (E, D) is a scheme satisfying (1) with message-size m and key-size $n < m$, then there exist two messages $x, x' \in \{0,1\}^m$ such that $\mathsf{E}_{U_n}(x)$ is not the same distribution as $\mathsf{E}_{U_n}(x')$. H460

9.3 Prove that in the one-time pad encryption, no eavesdropper can guess any bit of the plaintext with probability better than $1/2$. That is, prove that for every function A, if (E, D) denotes the one-time pad encryption then

$$\Pr_{\substack{k \in_R \{0,1\}^n \\ x \in_R \{0,1\}^n}}[A(\mathsf{E}_k(x)) = (i, b) \text{ s.t. } x_i = b] \leq 1/2\,.$$

Thus, the one-time pad satisfies in a strong way the condition (3) of computational security.

9.4 Exercise 9.2 and Lemma 9.2 show that for security against unbounded time adversaries (or efficient time if $\mathbf{P} = \mathbf{NP}$) we need key as large as the message. But they actually make an implicit subtle assumption: that the encryption process is *deterministic*. In a *probabilistic encryption scheme*, the encryption function E may be probabilistic: that is, given a message x and a key k, the value $\mathsf{E}_k(x)$ is not fixed but is distributed according to some distribution $Y_{x,k}$. Of course, because the decryption function is only given the key k and not the internal randomness used by E, we modify the requirement (1) to require $\mathsf{D}_k(y) = x$ for *every* y in the support of $\mathsf{E}_k(x)$. Prove that even a probabilistic encryption scheme cannot have key that's significantly shorter than the message. That is, show that for every probabilistic encryption scheme (D, E) using n-length keys and $n+10$-length messages, there exist two messages $x_0, x_1 \in \{0,1\}^{n+10}$ and function A such that

$$\Pr_{\substack{b \in_R \{0,1\} \\ k \in_R \{0,1\}^n}}[A(\mathsf{E}_k(x_b)) = b] \geq 9/10\,. \tag{11}$$

Furthermore, prove that if $\mathbf{P} = \mathbf{NP}$ then this function A can be dun in polynomial time. H460

9.5 Show that if $\mathbf{P} = \mathbf{NP}$ then one-way functions do not exist.

9.6 **(a)** Show that if there exists a one-way function f then there exists a one-way function g that is computable in n^2 time. H460

(b) Show that if there exists a one-way function f then the function $f_{\mathcal{U}}$ described in Section 9.2.1 is one way.

9.7 Prove that if there's a polylog(M) algorithm to invert the Rabin function $f_M(X) = X^2 \pmod{M}$ of Section 9.2.1 on a $1/\operatorname{polylog}(M)$ fraction of its inputs then we can factor M in polylog(M) time. H460

9.8 Let $\{(p_n, g_n)\}_{n \in \mathbb{N}}$ be some sequence of pairs of n-bit numbers such that p_n is prime and g_n is a generator of the group $\mathbb{Z}^*_{p_n}$, and there is a deterministic polynomial-time algorithm such that $S(1^n) = (p_n, g_n)$ for every $n \in \mathbb{N}$.

Suppose A is an algorithm with running time $t(n)$ that on input $g_n^x \pmod{p_n}$, manages to find x for $\delta(n)$ fraction of $x \in \{0, .., p_n - 1\}$. Prove that for every $\epsilon > 0$, there is a randomized algorithm A' with running time $O(\frac{1}{\delta \log 1/\epsilon}(t(n) + \operatorname{poly}(n)))$ such that for *every* $x \in \{0, .., p_n - 1\}$, $\Pr[A'(g_n^x \pmod{p_n}) = x] \geq 1 - \epsilon$. This property is known as the *self reducibility* of the discrete logarithm problem. H460

9.9 We say that a sequence of random variables $\{X_n\}_{n \in \mathbb{N}}$ where $X_n \in \{0,1\}^{m(n)}$ for some polynomial $m(\cdot)$ is *sampleable* if there's a probabilistic polynomial-time algorithm D such that X_n is equal to the distribution $D(1^n)$ for every n. Let (E, D) be an encryption scheme such that for every n, (E, D) uses length n keys to encrypt length $m(n)$ messages for some polynomial $m(\cdot)$. We say that (E, D) is *semantically secure*, if for every sampleable sequence $\{X_n\}$ (where $X_n \in \{0,1\}^{m(n)}$, every polynomial-time computable function $f : \{0,1\}^* \to \{0,1\}$, and every probabilistic polynomial-time algorithm A, there exists negligible function $\epsilon : \mathbb{N} \to [0,1]$ and a probabilistic polynomial-time algorithm B such that

$$\Pr_{\substack{k \in_R \{0,1\}^n \\ x \in_R X_n}}[A(\mathsf{E}_k(x)) = f(x)] \leq \Pr_{x \in_R X_n}[B(1^n) = f(x)] + \epsilon(n)\,.$$

That is, A cannot compute $f(x)$ given an encryption of x better than just guessing it using the knowledge of the distribution X_n.

(a) Prove that if (E, D) is semantically secure then it's also satisfy the condition of "computational security" of Section 9.2.2.

(b) Prove that if G is a pseudorandom generator mapping $\{0,1\}^n$ to $\{0,1\}^m$, then the encryption $\mathsf{E}_k(x) = x \oplus G(k), \mathsf{D}_k(y) = y \oplus G(k)$is semantically secure. H460

(c) Prove that semantic security is equivalent to its special case where for every n, X_n is the uniform distribution over a pair of strings x_0^n, x_1^n and f is the function that maps x_0^n to 0 and x_1^n to 1 for every n. H460

9.10 Show that if there exists a secure pseudorandom generator with stretch $\ell(n) = n+1$ then for every c there exists a pseudorandom generator with stretch $\ell(n) = n^c$. H460

9.11 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 $c > 0$.

9.12 Assuming one-way functions exist, show that the above fails for one-way functions. That is, design a one-way function f where f^{n^c} is not one-way for some constant c.

9.13 Suppose $x \in \{0,1\}^m$ is an unknown vector. Let $r^1, \ldots, r^m \in \{0,1\}^m$ be randomly chosen, and $x \odot r_i$ revealed to us for all $i = 1, 2, \ldots, m$. Describe a deterministic algorithm to reconstruct x from this information, and show that the probability (over the choice of the r^i's) is at least 1/4 that it works. This shows that if $r^1, \ldots, r^m$ are fully independent then we cannot guess $x \odot r^1, \ldots, x \odot r^m$ with probability much better than 2^{-m} (and hence it was crucial to move to a merely pairwise independent collection of vectors in the proof of Theorem 9.12). H460

9.14 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 $c > 0$.

9.15 Say that two sequences $\{X_n\}, \{Y_n\}$ of random variables, where $X_n, Y_n \in \{0,1\}^{m(n)}$ for some polynomial $m(n)$, are *computationally indistinguishable* if for every probabilistic polynomial-time A there exists a negligible function $\epsilon : \mathbb{N} \to [0,1]$ such that

$$\left|\Pr[A(X_n) = 1] - \Pr[A(Y_n) = 1]\right| < \epsilon(n)$$

for every n. Prove that:

(a) If $f : \{0,1\}^* \to \{0,1\}^*$ is a polynomial-time computable function and $\{X_n\}, \{Y_n\}$ are computationally indistinguishable, then so are the sequences $\{f(X_n)\}, \{f(Y_n)\}$.

(b) A polynomial-time computable function G with stretch $\ell(n)$ is a secure pseudorandom generator if and only if the sequences $\{U_{\ell(n)}\}$ and $\{G(U_n)\}$ are computationally indistinguishable.

(c) An encryption scheme (E, D) with $\ell(n)$-length messages for n-length keys is semantically secure if and only if for every pair of probabilistic polynomial time algorithms X_0, X_1, where $|X_0(1^n)| = |X_1(1^n)| = \ell(n)$, the sequences $\{\mathsf{E}_{U_n}(X_0(1^n))\}$ and $\{\mathsf{E}_{U_n}(X_1(1^n))\}$ are computationally indistinguishable.

9.16 Suppose that one-way permutations exist. Prove that there exists a pair of polynomially sampleable computationally indistinguishable distributions $\{G_n\}$ and $\{H_n\}$ over n-vertex graphs such that for every n, G_n and H_n are n-vertex graphs, and $\Pr[G_n \text{ is 3-colorable}] = 1$ but $\Pr[H_n \text{ is 3-colorable}] = 0$. (A graph G is 3-colorable if G's vertices can be colored in one of three colors so that no two neighboring vertices have the same color, see Exercise 2.2). H460

9.17 Say that a language L has a *computational zero knowledge* proof if it satisfies the relaxation of Definition 9.14 where condition (9) is replaced by the condition that $\{\mathsf{out}_{v^*}\langle P(X_n, U_n), V^*(X_n)\rangle\}$ and $\{S^*(X_n)\}$ are computationally indistinguishable for every sampleable distribution (X_n, U_n) such that $|X_n| = n$ and $\Pr[M(X_n, U_n) = 1] = 1$,

(a) Prove that if there exists a computational zero knowledge proof for some language L that is **NP**-complete via a Levin reduction (Section 2.3.6), then there exists a computational zero knowledge proof for every $L \in \mathbf{NP}$.

(b) Prove that the following protocol (due to Blum [Blu87]) is a computational zero knowledge proof system with completeness 1 and soundness error $1/2$ for the language of Hamiltonian circuits:[11]

Common input Graph G on n vertices.

Prover's private input A Hamiltonian cycle C in the graph.

Prover's first message Choose a random permutation π on the vertices of G, and let M be the adjacency matrix of G with its rows and columns permuted according to π. For every $i, j \in [n]$, choose $x^{i,j}, r^{i,j} \in_R \{0,1\}^n$ and send to the verifier $f(x^{i,j}), r^{i,j}, (x^{i,j} \odot r^{i,j}) \oplus M_{i,j}$.

Verifier's message Verifier chooses $b \in_R \{0,1\}$ and sends b to prover.

Prover's last message If $b = 0$, the prover sends to the verifier all randomness used in the first message. That is, the prover reveals the permutation π, the matrix M, and reveals $x_{i,j}$ for every $i, j \in [n]$. If $b = 1$, the prover computes C' which is the permuted version of the cycle C (i.e., C' contains $(\pi(i), \pi(j))$ for every edge $\overline{i\,j} \in C$). It then sends C' to the verifier, and reveals only the randomness corresponding to these edges. That is, for every $(i, j) \in C'$ it sends $x_{i,j}$ to the verifier.

[11] The soundness error can be reduced by repetition.

Verifier's check If $b = 0$, the verifier checks that the prover's information is consistent with its first message— that M is the permuted adjacency matrix of G according to π, and that the values $x_{i,j}$ are consistent with $M_{i,j}$ and the values $y_{i,j}$ that the prover sent in its first message. If $b = 1$ then the verifier checks that C' is indeed a Hamiltonian cycle, and that the values the prover sent are consistent with its first message and with $M_{i,j} = 1$ for every $(i, j) \in C'$. The verifier accepts if and only if these checks succeed.