1 Overview

Aller Anfang ist schwer: All beginnings are difficult.
— German proverb

Defendit numerus: There is safety in numbers.
— Anonymous

The ability to quote is a serviceable substitute for wit.
— W. Somerset Maugham

1.1 Introduction

It is rather difficult to give a precise definition of *cryptography*. Loosely speaking, it is the science (or art or technology) of preventing access to sensitive data by parties who are not authorized to access the data. Secure transmission of messages over a public channel is the first, simplest and oldest example of a cryptographic protocol. For assessing the security of these protocols, one studies their possible weak points, namely the strategies for breaking them. This study is commonly referred to as *cryptanalysis*. And, finally, the study of both cryptography and cryptanalysis is known as *cryptology*.

$$\boxed{\text{Cryptology} = \text{Cryptography} + \text{Cryptanalysis}}$$

The science of cryptology is rather old. It naturally developed as and when human beings felt the need for privacy and secrecy. The rapid deployment of the Internet in the current years demands that we look into this subject with a renewed interest. Newer requirements tailored to Internet applications have started cropping up and as a result newer methods, protocols and algorithms are coming up. The most startling discoveries include that of the key-exchange protocol by Diffie and Hellman in 1976 and that of the RSA cryptosystem by Rivest, Shamir and Adleman in 1978. They opened up a new branch of cryptology, namely *public-key cryptology*. Historically, public-key technology came earlier than the Internet, but it is the latter that makes an extensive use of the former.

This book is an attempt to introduce to the reader the vast and interesting branch of public-key cryptology. One of the most distinguishing features of public-key cryptology is that it involves a reasonable amount of abstract mathematics which often comes in the way of a complete understanding to an uninitiated reader. This book tries to bridge the gap. We develop the required mathematics in necessary and sufficient details.

This chapter is an overview of the topics that the rest of the book deals with. We start with a description of the most common cryptographic protocols. Then we introduce the public-key paradigm and discuss the source of its security. We use certain mathematical terms and notations throughout this chapter. If the reader is not already familiar with these terms, there is nothing to worry about. As we have just claimed, we will introduce the mathematics in the later chapters. The exposition of this chapter is expected to give the reader an overview of the area of public-key cryptography and also the requisite motivation for learning the mathematical tools that follow.

1.2 Common Cryptographic Primitives

As claimed at the outset of this chapter, it is rather difficult to give a precise definition of the term *cryptography*. The best way to understand it is by examples. In this section, we briefly describe the common problems that cryptography deals with.

1.2.1 The Classical Problem: Secure Transmission of Messages

To start with, we introduce the legendary figures of cryptography: Alice, Bob and Carol. Alice wants to send a message to Bob over a public communication channel like the Internet and wants to ensure that nobody other than Bob can make out the meaning of the message. A third party like Carol, who has access to the communication channel, can intercept the message. But the message should be wrapped or transformed before transmission in such a way that knowledge of some secret piece of information is needed to unwrap or transform back the message. It is Bob who has this information, but not Carol (nor Dorothy nor Emily nor . . .).

It is expedient to point out here that Alice, Bob and Carol need not be human beings. They can stand for organizations (like banks) or, more correctly, for computers or computer programs run by individuals or organizations. It is, therefore, customary to call them *parties*, *entities* or *subjects* instead of persons or characters. In the cryptology jargon, Carol has got several names used interchangeably: *adversary*, *eavesdropper*, *opponent*, *intruder*, *attacker* and *enemy* are the most common ones. When a message transmission like that just mentioned is involved, Alice is called the *sender* and Bob is called the *receiver* of the message.

It is a natural strategy to put the message in a box and lock the box using a *key*, called the *encryption key*. A matching *decryption key* is needed to unlock the box and retrieve the message. The process of putting the message in the box is commonly called *encoding* and that of locking the box is called *encryption*. The reverse processes, namely unlocking the box and taking the message out of the box are respectively called *decryption* and *decoding*. This is precisely the classical *encryption–decryption protocol* of cryptography.[1]

In the world of electronic communication, a message M is usually a bit string, and encoding, encryption, decryption and decoding are well-defined transformations of bit strings. If we denote by f_e the transformation function consisting of encoding and encryption, then we get a new bit string $C = f_e(M, K_e)$, where K_e stands for the encryption key. This bit string C is sent over the communication channel. After Bob receives C, he uses the reverse transformation f_d (decryption followed by decoding) to get the original message M back; that is, $M = f_d(C, K_d)$. Note that the decryption key K_d is needed as an argument to f_d. If Carol does not know this, she cannot compute M. We conventionally call M the *plaintext message* and C the *ciphertext message*.

The encoding and decoding operations do not make use of keys and can be performed by anybody. (It should not be difficult to put a letter in or take a letter out of an unlocked box!) One might then wonder why it is necessary to do these transformations instead of applying the encryption and decryption operations directly on M and C respectively. With whatever we have discussed so far, we cannot give a full answer to this question. For the answer, we will need to wait until we reach the later chapters. We only mention here that the encryption algorithms often require as input some mathematical entities (like integers or elements of a field) which are *logically* not bit strings. But that's not all! As we see later, the additional transformations often add to the security of the protocols. On the other hand, for a general discussion, it is often unnecessary to start from the encoding process and end at the decoding process. As a result, we will assume, unless otherwise stated, that M is the input to the encryption routine and the output of the decryption routine, in which case f_e and f_d stand for the encryption and decryption functions only.

[1]Some people prefer to use the terms *enciphering* and *deciphering* in place of the words *encryption* and *decryption* respectively.

Symmetric-key or secret-key cryptography

In the simplest form of locking mechanism, one has $K_e = K_d$. That is, the same key, called the *symmetric key* or the *secret key*, is used for both encryption and decryption. Common examples of such symmetric-key algorithms include DES (Data Encryption Standard) together with its various modifications like the Triple DES and DES-X, IDEA (International Data Encryption Algorithm), SAFER (Secure And Fast Encryption Routine), FEAL (Fast Encryption Algorithm), Blowfish, RC5 and AES (Advanced Encryption Standard). We will not describe all these algorithms in this book. Interested readers can look at the abundant literature to know more about them.

Asymmetric-key or public-key cryptography

The biggest disadvantage of using a secret-key system is that Alice and Bob must agree upon the key $K_e = K_d$ *secretly*, for example by personal contact or over a secure channel. This is a serious limitation and is not often practical nor even possible. Another drawback of secret-key systems is that every pair of parties needs a key for communication. Thus, if there are n entities communicating over a net, the number of keys would be of the order of n^2. Also, each entity has to *remember* $\mathrm{O}(n)$ keys for communicating with other entities. In practice, however, an entity does not communicate with every other entity on the net. Yet the total number of keys to be remembered by an entity could be quite high.

Both these problems can be avoided by using what is called an *asymmetric-key* or a *public-key* protocol. In such a protocol, each entity decides a *key pair* (K_e, K_d), makes the encryption key K_e public and keeps the decryption key K_d secret. K_e is also called the *public key* and K_d the *private key*. Anybody who wants to send a message to Bob gets Bob's public key, encrypts the message with the key, and sends the ciphertext to Bob. Upon receiving the ciphertext, Bob uses his private key to decrypt the message. One may view such a lock as a self-locking padlock. Anybody can lock a box with a self-locking padlock, but opening it requires a key which only Bob possesses.

The source of security of such a system is based on the difficulty of computing the private key K_d given the public key K_e. It is apparent that K_e and K_d are sort of *inverses* of each other, because the former is used to generate C from M and the latter is used to generate M from C. This is where mathematics comes into the picture. We mention a few possible constructions of key pairs in the next section and the rest of the book deals with an in-depth study of these public-key protocols.

Attractive as it looks, public-key protocols have a serious drawback, namely they are orders of magnitude slower than their secret-key counterparts. This is of concern, if huge amounts of data need to be encrypted and decrypted. This shortcoming can be overcome by using both secret-key and public-key protocols in tandem as follows: Alice generates a secret key (say, for AES), encrypts the message by the secret key and the secret key by the public key of Bob and sends both the encrypted message and the encrypted secret key. Bob first decrypts the encrypted secret key using his private key and uses this decrypted secret key to decrypt the message. Since secret keys are usually short bit strings (most commonly of length 128 bits), the slow performance of the public-key algorithms causes little trouble. But at the same time, Alice and Bob are relieved of having a previous secret meeting or communication for agreeing on the secret key. Moreover, neither Alice nor Bob needs to remember the secret key. During every session of message transmission, a random secret key can be generated and later destroyed, when the communication is over.

1.2.2 Key Exchange

There is an alternative method by which Alice and Bob can exchange secret information (like AES keys) over a public communication channel. Let us first see how this can be done in the physical lock-and-key scenario. Alice generates a secret, puts it in a box, locks the box with her own key and sends it to Bob. Bob, upon receiving the locked box, adds a second lock to it and sends the doubly locked box back to Alice. Alice then removes her lock and again sends the box to Bob. Finally, Bob uses his key to unlock the box and retrieve the secret. A third party (Carol) that can access the box during the three communications finds it locked by Alice or Bob or both. Since Carol does not possess the keys to these locks, she cannot open the box to discover the secret.

This process can be abstractly described as follows: Alice and Bob first independently generate key pairs (A_{K_e}, A_{K_d}) and (B_{K_e}, B_{K_d}) respectively. Alice then sends A_{K_e} to Bob and Bob sends B_{K_e} to Alice. The private keys A_{K_d} and B_{K_d} are not disclosed. They also agree upon a function g with which Alice computes $g_A = g(A_{K_d}, B_{K_e})$ and Bob computes $g_B = g(B_{K_d}, A_{K_e})$. If $g_A = g_B$, then this common value can be used as a shared secret between Alice and Bob.

Our intruder Carol knows g and taps the values of A_{K_e} and B_{K_e}. So the function g should be such that a knowledge of these values alone does not suffice for the computation of $g_A = g_B$. One of the private keys A_{K_d} or B_{K_d} is needed for the computation. Since (A_{K_e}, A_{K_d}) and (B_{K_e}, B_{K_d}) are key pairs, it is assumed that private keys are difficult to compute from the knowledge of the corresponding public keys.

Such a technique of exchanging secret values over an insecure channel is called a *key-exchange* or a *key-agreement* protocol. It is important to point out here that such a protocol is usually based on the public-key paradigm; that is to say, we do not know secret-key counterparts for a key-exchange protocol. Since a shared secret between the communicating parties is usually short, the low speed of public-key algorithms is really not a concern in this case.

1.2.3 Digital Signatures

A digital signature is yet another application of the public-key paradigm. Suppose Alice wants to sign a message M in such a way that the signature S can be verified by anybody but nobody other than Alice would be able to generate the signature S on the message M. This can be achieved as follows: Alice generates a key pair (K_e, K_d), makes K_e public and keeps K_d secret. She now uses the decryption function f_d to generate the signature, that is, $S = f_d(M, K_d)$. The signature S is then made public. Anybody who has access to Alice's public key K_e applies the reverse transformation f_e to get back the message $M = f_e(S, K_e)$.

If Carol signs the message M with a different key K'_d, then she generates the signature $S' = f_d(M, K'_d)$. Now, since K'_d and K_e are not matching keys, verification using K_e gives $M' = f_e(S', K_e)$, which is different from M. If we assume that M is a message written in a human-readable language (like English), then M' would generally look like a meaningless sequence of characters which is neither English nor any sensible string to a human reader. So the signature verifier would then immediately conclude that this is a case of forged signature.

Such a scheme of generating digital signatures is called a *signature scheme with message recovery*. It is obvious that this is the same as our encrypt–decrypt scheme with the sequence of encryption and decryption steps reversed. If the message M to be signed is quite long, using this algorithm calls for a large execution time both for signature generation and for verification. It is, therefore, customary to use another variant of signature schemes called *signature schemes with appendix* that we describe now.

Instead of applying the decryption transform directly on M, Alice first computes a *short* representative $H(M)$ of her message M. Her signature now becomes the pair $S = (M, \sigma)$, where $\sigma = f_d(H(M), K_d)$. Typically, a hash function (see Section 1.2.6) is used to compute the representative $H(M)$ from M and is assumed to be a public knowledge. Now anybody can verify the signature by checking if the equality $H(M) = f_e(\sigma, K_e)$ holds. If a key different from K_d is used to generate the signature, one would (in general) get a value $\sigma' \neq \sigma$ and the signature forging will be detected by observing that $H(M) \neq f_e(\sigma', K_e)$.

1.2.4 Entity Authentication

By *entity authentication*, we mean a process in which one entity called the *claimant* proves its identity to another entity called the *prover*. Entity-authentication techniques, thus, tend to prevent impersonation of an entity by an intruder. Both secret-key and public-key techniques are used for entity-authentication schemes.

The simplest example of an entity-authentication scheme is the use of *passwords*, as in a computer where a user (the claimant) tries to gain access to some resources in a computer (the prover) by proving its identity using a password. Password schemes are mostly based on secret-key techniques. For example, the UNIX password system is based on encrypting the zero message (a string of 64 zero bits) using a repeated application of a variant of the DES algorithm with 64 bits of the user input (the password) as the key. Password-based authentication schemes are fixed and time-invariant and are often called *weak authentication* schemes.

We see applications of public-key techniques in *challenge–response authentication schemes* (also called *strong authentication schemes*). Assume that an entity, Alice, wants to prove her identity to another entity, Bob. Alice generates a key pair (K_e, K_d), makes K_e public and keeps K_d secret. Now, Bob chooses a random message M, encrypts M using Alice's public key—that is, computes $C = f_e(M, K_e)$—and sends C to Alice. Alice, upon reception of C, decrypts it using her private key K_d; that is, she regenerates $M = f_d(C, K_d)$ and sends M to Bob. Bob compares this value of M with the one he generated, and if a match occurs, Bob becomes sure that the entity who is claiming to be Alice possesses the knowledge of Alice's private key. If Carol uses any private key other than K_d for the decryption, she gets a message M' different from M and thereby cannot prove to Bob her identity as Alice. This is how this scheme prevents impersonation of Alice by Carol.

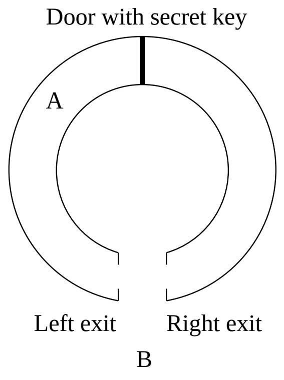

Figure 1.1 Zero-knowledge proofs

Entity authentication is often carried out using another interesting technique called *zero-knowledge proof*. In such a protocol, the prover (or any third party listening to the conversation) gains *no knowledge* regarding the secret possessed by the claimant, but develops the desired confidence regarding the claim by the claimant of the *possession* of the secret. We provide here an informal example explaining zero-knowledge proofs.

Let us think of a circular cave as shown in Figure 1.1. The cave has two exits, left and right, denoted by L and R respectively. The cave also has a door inside it, which is invisible outside the cave. Alice (A) wants to prove to Bob (B) that she possesses a key to this door without showing him the key or the process of unlocking the door with the key. Bob stations himself somewhere outside the exits of the cave. Alice enters the cave and randomly chooses the left or right wing of the cave (and goes there). She does not disclose this choice to Bob, because Bob is not allowed to know the session secrets too. Once Alice is placed in the cave, Bob makes a random choice from L and R and asks Alice (using cell phones or by shouting loudly) to come out of the cave via that chosen exit. Suppose Bob challenges Alice to use L. If Alice is in the left wing, she can come out of the cave using L. If Alice is in the right wing, she must use her secret key to open the central door to come to the left wing and then go out using exit L. If Alice does *not* possess the secret key, she can succeed in obeying Bob's directive with a probability of half. If this procedure is repeated t times, then the probability that Alice succeeds on all occasions without possessing the secret key is $(1/2)^t = 1/2^t$. By choosing t appropriately, Bob can make the probability of accepting a false claim arbitrarily small. For example, if $t = 20$, then the chance is less than one in a million that Alice can establish a false claim.

Thus, if Alice succeeds every time, Bob gains the desired confidence that Alice actually possesses the secret. However, during this entire process, Bob can obtain no information regarding Alice's secrets (the key and the choices of wings). Another important aspect of this interaction is that Alice has no way of predicting Bob's questions, preventing impostors (of Alice) from fooling Bob.

1.2.5 Secret Sharing

Suppose that a secret piece of information is to be distributed among n entities in such a way that $n-1$ (or fewer) entities are unable to construct the secret. All of the n entities must participate to reveal the secret. As usual, let us assume that the secret is an l-bit string. A simple strategy would be to break the string into n parts and provide each entity with a part. This method is, however, not really attractive, because it gives partial information about the secret. Thus, for example, if a 256-bit long bit string is to be distributed equally among 16 entities, any 15 of them working together can reconstruct the secret by trying only $2^{16} = 65536$ possibilities for the unknown 16 bits.

We now describe an alternative strategy that does not suffer from this drawback. Once again, we break the secret string into n parts and consider the parts as integers $a_0, \ldots, a_{n-1}$. We construct the polynomial $f(x) = x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$ and give the integers $f(1), f(2), \ldots, f(n)$ to the entities. When all of the entities cooperate, the linear system of equations $f(i) = i^n + a_{n-1}i^{n-1} + \cdots + a_1 i + a_0$, $1 \leqslant i \leqslant n$, can be solved to find out the unknown coefficients $a_0, \ldots, a_{n-1}$ which, in turn, reveal the secret. On the other hand, if $n-1$ or less entities cooperate, they get an underspecified system of equations in n unknowns, from which the actual solution is not readily available.

The secret-sharing problem can be generalized in the following way: to distribute a secret among n parties in such a way that any m or more of the parties can reconstruct the secret (for some $m \leqslant n$), whereas any $m-1$ or less parties cannot do the same. A polynomial of degree m as in the above example readily adapts to this generalized situation.

1.2.6 Hashing

A function which converts bit strings of arbitrary lengths to bit strings of a fixed (finite) length is called a *hash function*. Hash functions play a crucial role in cryptography. We have already seen an application of it for designing a digital signature scheme with appendix. If H is a hash function, a pair of input values (strings) x_1 and x_2 for which $H(x_1) = H(x_2)$ is called a *collision* for H. For any hash function H, collisions must exist, since H is a map from an infinite set to a finite set. However, for cryptographic purposes we want that collisions should be difficult to obtain. More specifically, a cryptographic hash function H should satisfy the following desirable properties:

First pre-image resistance Except for a *small* set of hash values y it should be difficult to find an input x with $H(x) = y$. We exclude a small set of values, because an adversary might prepare (and maintain) a list of pairs $(x, H(x))$ for certain values of x of her choice. If the given value of y is the second coordinate of one pair in her list, she can produce the corresponding input value x easily.

Second pre-image resistance Given a pair $(x, H(x))$, it should be difficult to find an input x' different from x with $H(x) = H(x')$.

Collision resistance It should be difficult to find two different input strings x, x' with $H(x) = H(x')$.

Hash functions are also called *message digests* and can be used with a secret key. Popular examples of unkeyed hash functions are SHA-1, MD5 and MD2, whereas those for keyed hash functions include HMAC and CBCMAC.

1.2.7 Certification

So far we have seen several protocols which are based on the use of public keys of remote entities, but have never questioned the authenticity of public keys. In other words, it is necessary to ascertain that a public key is really owned by a remote entity. *Public-key certificates* are used to that effect. These are data structures that bind public-key values to entities. This binding is achieved by having a trusted *certification authority* digitally sign each certificate.

Typically a certificate is issued for a period of validity. However, it is possible that a certificate becomes invalid before its date of expiry for several reasons, like possible or suspected compromise of the private key. Under such circumstances it is necessary that the certification authority *revokes* the certificate and maintains a list called *certificate revocation list* (CRL) of revoked certificates. When Alice verifies the authenticity of Bob's public-key certificate by verifying the digital signature of the authority and does not find the certificate in the CRL, she gains the desired confidence in using Bob's public key.

The X 5.09 *public-key infrastructure* specifies Internet standards for certificates and CRLs.

1.3 Public-key Cryptography

In this section, we give a short introduction to the realization of public-key cryptosystems. More specifically, we list some of the computationally intensive mathematical problems and describe how the (apparent) intractability of these problems can be used for designing key pairs. We use some mathematical terms that we will introduce later in this book.

1.3.1 The Mathematical Problems

The security of the public-key cryptosystems is based on the *presumed* difficulty of solving certain mathematical problems.

The integer factorization problem (IFP) Given the product $n = pq$ of two distinct prime integers p and q, find p and q.

The discrete logarithm problem (DLP) Let G be a finite cyclic (multiplicatively written) group with cardinality n and a generator g. Given an element $a \in G$, find an integer x (or *the* integer x with $0 \leqslant x \leqslant n-1$) such that $a = g^x$ in G. Three different types of groups are commonly used for cryptographic applications: the multiplicative group of a finite field, the group of rational points on an elliptic curve over a finite field and the Jacobian of a hyperelliptic curve over a finite field. By an abuse of notation, we often denote the DLP over finite fields as simply DLP, whereas the DLP in elliptic curves and hyper-elliptic curves is referred to as the *elliptic curve discrete logarithm problem* (ECDLP) and the *hyperelliptic curve discrete logarithm problem* (HECDLP).

The Diffie–Hellman problem (DHP) Let G and g be as above. Given elements g^a and g^b of G, compute the element g^{ab}. As in the case of the DLP, the DHP can be applied to the multiplicative group of finite fields, the group of rational points on an elliptic curve and the Jacobian of a hyperelliptic curve.

We show in the next section how (the intractability of) these problems can be exploited to create key pairs for various cryptosystems. These computational problems are termed *difficult*, *intractable*, *infeasible* or *intensive* in the sense that there are no *known* algorithms to solve these problems in time polynomially bounded by the input size. The best-known algorithms are *subexponential* or even fully *exponential* in some cases. This means that if the input size is chosen to be sufficiently large, then it is infeasible to compute the private key from a knowledge of the public key in a reasonable amount of time. This, in turn, implies (not provably, but as the current state of the art stands) that encryption or signature verification can be done rather quickly (in polynomial time), but the converse process of decryption or signature generation cannot be done in feasible time, unless one knows the private key. As a result, encryption (or signature verification) is called a *trapdoor one-way function*, that is, a function which is easy to compute but for which the inverse is computationally infeasible, unless some additional information (the trapdoor) is available.

It is, however, not known that these problems are really computationally infeasible, that is, there is no proof of the fact that these problems cannot be solved in polynomial time. As a result, the public-key cryptographic systems based on these problems are *not provably secure*.

1.3.2 Realization of Key Pairs

In RSA and similar cryptosystems, one generates two (distinct) suitably large primes p and q and computes the product $n = pq$. Then $\phi(n) = (p-1)(q-1)$, where ϕ denotes Euler's totient function. One then chooses a random integer e with $\gcd(e, \phi(n)) = 1$. There exists an integer d such that $ed \equiv 1 \pmod{\phi(n)}$. The integer e is used as the public key, whereas the integer d is used as the private key.

If the IFP can be solved fast, one can also compute $\phi(n)$ easily, and subsequently d can be computed from e using the (polynomial-time) extended GCD algorithm. This is why[2] we say that the RSA cryptosystem derives its security from the intractability of the IFP.

In order to see how RSA encryption and decryption work, let the plaintext message be encoded as an integer m with $2 \leqslant m < n$. The ciphertext message is generated (as an integer) as $c = m^e \pmod{n}$. Decryption is analogous, that is, $m = c^d \pmod{n}$. The correctness of the algorithm follows from the fact that $ed \equiv 1 \pmod{\phi(n)}$. It is, however, not proved that one *has to* know d or $\phi(n)$ or the factorization of n in order to decrypt an RSA-encrypted message. But at present no better methods are known.

[2]The problem of factoring $n = pq$ is polynomial-time equivalent to computing $\phi(n) = (p-1)(q-1)$.

Let us now consider the discrete logarithm problem. Let G be a finite cyclic multiplicative group (as those mentioned above) where it is easy to multiply two elements, but where it is difficult to compute discrete logarithms. Let g be a generator of G. In order to set up a random key pair over such a group, one chooses the private key as a random integer d, $2 \leqslant d < n$, where n is the cardinality of G. The public key e is then computed as an element of G as $e = g^d$.

Applications of encryption–decryption schemes based on the key pair (g^d, d) are given in Chapter 5. Now, we only remark that many such schemes (like the ElGamal scheme) derive their security from the DHP instead of the DLP, whereas the other schemes (like the Nyberg–Rueppel scheme) do so from the DLP. It is assumed that these two problems are computationally equivalent (at least for the groups of our interest). Obviously, if one assumes availability of a solution of the DLP, one has a solution for the DHP too ($g^{ab} = (g^a)^b$). The reverse implication is not clear.

1.3.3 Public-key Cryptanalysis

As we pointed out earlier, (most of) the public-key cryptosystems are not provably secure in the sense that they are based on the *apparent* difficulty of solving certain computational problems. It is expedient to know how difficult these problems are. No non-trivial complexity–theoretic statements are available for these problems, and as such it is worthwhile to study the algorithms known till date to solve these problems. Unfortunately, however, many of the algorithms of this kind are often much more complicated than the algorithms for building the corresponding cryptographic systems. One needs to acquire more mathematical machinery in order to understand (and augment) these cryptanalytic algorithms. We devote Chapter 4 to a detailed discussion on these algorithms.

In specific situations, one need not always use these computationally intensive algorithms. Access to a party's decryption equipment may allow an adversary to gain partial or complete information about the private key by *watching* a decryption process. For example, an adversary (say, the superuser) might have the capability to read the contents of the memory holding a private key during some decryption process. For another possibility, think of RSA decryption which involves a modular exponentiation. If the standard square-and-multiply algorithm (Algorithm 3.9) is used for this purpose and the adversary can tap some hardware details (like machine cycles or power fluctuations) during a decryption process, she can guess a significant number of the bits in the private key. Such attacks, often called *side-channel attacks*, are particularly relevant for cryptographic applications based on smart cards.

A cryptographic system is (believed to be) strong if and only if there are no good known mechanisms to break it. It is, therefore, for the sake of security that we must study cryptanalysis. Cryptography and cryptanalysis are deeply intertwined and a complete study of one must involve the other.

1.4 Some Cryptographic Terms

In cryptology, there are different models of attacks or attackers.

1.4.1 Models of Attacks

So far we have assumed that an adversary can *only* read messages during transmission over a channel. Such an adversary is called a *passive adversary*. An *active adversary*, on the other hand, can mutilate or delete messages during transmission and/or generate false messages. An attack mounted by an active (resp.[3] a passive) adversary is called an *active* (resp. a *passive*) *attack*. In this book, we will mostly concentrate on passive attacks.

1.4.2 Models of Passive Attacks

A two-party communication involves transmission of ciphertext messages over a communication channel. A passive attacker can read these ciphertext messages. In practice, however, an attacker might have more control over the choice of ciphertext and/or plaintext messages. Based on these capabilities of the attacker we have the following types of attacks.

Ciphertext-only attack This is the weakest model of the adversary. Here the attacker has absolutely no choices on the ciphertext messages that flow in the channel and also on the corresponding plaintext messages. Using only these ciphertext messages the attacker has to obtain a private key and/or a plaintext message corresponding to a new ciphertext message.

Known-pair attack In this kind of attack (also called *known-plaintext* or *known-ciphertext attack*), the attacker uses her knowledge of some plaintext–ciphertext pairs. If many such pairs are available to the attacker, she can use these pairs to deduce a pattern based on which she can subsequently gain some information on a new plaintext for which the ciphertext is available. In a public-key scheme, the adversary can generate as many such pairs as she wants, because in order to generate such a pair it is sufficient to have a knowledge of the receiver's public key. Thus a public-key encryption scheme must provide sufficient security against known plaintext attacks.

Chosen-plaintext attack In this kind of attack, the attacker knows some plaintext–ciphertext pairs in which the plaintexts are *chosen* by the attacker. As discussed earlier, such an attack is easily mountable for a public-key encryption scheme.

Adaptive chosen-plaintext attack This is similar to the chosen-plaintext attack with the additional possibility that the attacker chooses the plaintexts in the known plaintext–ciphertext pairs sequentially and *adaptively* based on the knowledge of the previous pairs. This kind of attack can be easily mounted on public-key encryption systems.

Chosen-ciphertext attack The attacker has knowledge of some plaintext–ciphertext pairs in which the ciphertexts are *chosen* by the attacker. Such an attack is not directly mountable on a public-key scheme, since obtaining a plaintext from a chosen ciphertext requires knowledge of the private key. However, if the attacker has access to the receiver's decryption equipment, the machine can divulge the plaintexts corresponding to the ciphertexts that the attacker supplies to the machine. In this context, we assume that the machine does not reveal the private key itself, that is, it has the key

[3]Throughout the book, resp. stands for respectively.

stored secretly somewhere in its hardware which the attacker cannot directly access. However, the attacker can run the machine to know the plaintexts corresponding to the ciphertexts of her choice. Later (when the attacker no longer has access to the decryption equipment) the known pairs may be exploited to obtain information about the plaintext corresponding to a new ciphertext.

Adaptive chosen-ciphertext attack This is similar to the chosen-ciphertext attack with the additional possibility that the attacker chooses the ciphertexts in the known pairs sequentially and *adaptively* based on her knowledge of the previously generated plaintext–ciphertext pairs. This attack is mountable in a scenario described in connection with chosen-ciphertext attacks.

For a digital signature scheme, there are equivalent names for these types of attacks. The attacker is assumed to have access to the public key of the signer, because this key is used for signature verification. An attempt to forge signatures based only on the knowledge of this verification key is called a *key-only attack*. The adversary may additionally possess knowledge of some message–signature pairs. An attack based on this knowledge is called a *known-pair* or *known-message* or *known-signature attack*. If the messages are *chosen* by the adversary, we call the attack a *chosen-message attack*. If the adversary generates the sequence of messages in a chosen-message attack adaptively (based on the previously generated message–signature pairs), we have an *adaptive chosen-message attack*. An (adaptive or non-adaptive) chosen-message attack can be mounted, if the attacker gains access to the signer's signature generation equipment, or if the signer is willing to sign arbitrary messages provided by the adversary.

The attacker can choose some signatures and generate the corresponding messages by encrypting them with the signer's public key. The private-key operation on these messages generates the signatures chosen by the attacker. This gives *chosen-signature* and *adaptive chosen-signature attacks* on a digital signature scheme. Now the adversary cannot directly control the messages to sign. On the other hand, such an attack is easily mountable, because it utilizes only some public knowledge (the signer's public key). Indeed, one may treat chosen-signature attacks as variants of key-only attacks.

1.4.3 Public Versus Private Algorithms

So far, we have assumed that all the parties connected to a network *know* the algorithms used in a cryptographic scheme. The security of the scheme is based on the difficulty of obtaining some secret information (the secret or private key).

It, however, remains possible that two parties communicate using an algorithm unknown to other entities. Top-secret communications (for example, during wars or diplomatic transactions) often use private cryptographic algorithms. In this book, we will not deal with such techniques. Our attention is focused mostly on Internet applications in which public knowledge of the algorithms is of paramount importance (for the sake of universal applicability and convenience).

In short, this book is going to deal with a world in which only public public-key algorithms are deployed and in which adversaries are usually passive. A restricted model of the world though it may be, it is general and useful enough to concentrate on. Let us begin our journey!

Chapter Summary

This chapter provides an overview of the problems that *cryptology* deals with. The first and oldest cryptographic primitive is *encryption* for secure transmission of messages. Some other primitives are *key exchange*, *digital signature*, *authentication*, *secret sharing*, *hashing*, and *digital certificates*. We then highlight the difference between *symmetric* (secret-key) and *asymmetric* (*public-key*) *cryptography*. The relevance of some computationally intractable mathematical problems in public-key cryptography is discussed next, and the working of a prototype public-key cryptosystem (RSA) is explained. We finally discuss different models of attacks on cryptosystems.

Not uncommonly, some people think that cryptology also deals with intrusion, viruses, and Trojan horses. We emphasize that this is never the case. *Data and network security* is the branch that deals with these topics. Cryptography is also a part of this branch, but not conversely. Imagine that your house is to be secured against theft. First, you need a good lock—that is, cryptography. However, a lock has nothing to prevent a thief from entering the house after breaking the window panes. A bad butler who leaks secret information of the house to the outside world also does not come under the jurisdiction of the lock. Securing your house requires adopting sufficient guards against all these possibilities of theft. In this book, we will study only the technology of manufacturing and breaking locks.