Czesław Kościelny · Mirosław Kurkowski
Marian Srebrny

Modern Cryptography Primer

Theoretical Foundations and Practical Applications

Springer

Czesław Kościelny • Mirosław Kurkowski •
Marian Srebrny

Modern Cryptography Primer

Theoretical Foundations and Practical Applications

Springer

Czesław Kościelny
Faculty of Information Technology
Wrocław School of Information Technology
Wrocław, Poland

Mirosław Kurkowski
Inst. of Computer and Information Sciences
Czestochowa University of Technology
Czestochowa, Poland

and

European University of Information
Technology and Economics
Warsaw, Poland

Marian Srebrny
Institute of Computer Science
Polish Academy of Sciences
Warsaw, Poland

and

Section of Informatics
University of Commerce
Kielce, Poland

ISBN 978-3-642-41385-8 ISBN 978-3-642-41386-5 (eBook)
DOI 10.1007/978-3-642-41386-5
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2013955235

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Contents

Chapter 1
Basic Concepts and Historical Overview

1.1 Introduction

Cryptography is the science of transforming, or encoding, information into a form non-comprehensible for anyone who does not know an appropriate key. In such forms information can be securely transferred via any communication channel or stored in data archives with its access restricted or even forbidden (for one reason or another). Cryptography is a part of a broader discipline called *cryptology*, which includes also so-called *cryptanalysis*—the art of breaking codes (ciphers), i.e., regaining the content of encrypted messages without an authorized access to the decryption keys.

1.1.1 Encryption

Cryptography is the art of providing confidentiality of information (messages, documents) through encryption, whenever required, together with means of information security, data integrity, entity and data authentication.

Let us suppose someone (a sender) wishes to deliver some information to someone else (a receiver) via a public channel, e.g., the Internet. Moreover, the sender would like to make sure that no one else, but the intended receiver, can get the content being transmitted. The sender can do so by hiding the information content according to the following scheme. The information content being transferred is called a *plaintext*, or a *cleartext*. The procedure of hiding the content is called *encryption*, and the encrypted message is called its *ciphertext* or *cryptogram*. The reverse procedure of recapturing the content from its cryptogram is called *decryption*. The encryption and decryption algorithms together form a *cipher*. These concepts are illustrated in Fig. 1.1.

Depending on the encryption algorithm, a plaintext can be any information formulated in any way as a sequence of bits, text file, sequence of voice samples, digital video, et cetera. The examples listed come from the pervasive digital world, but

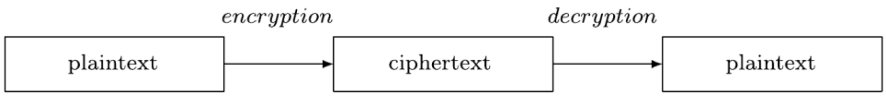

Fig. 1.1 Encryption and decryption

clearly in general one can encrypt information presented in any form whatsoever—it requires only an encryption algorithm to be applied or designed for this purpose. In this book a cipher's input is considered as binary data.

We usually denote a plaintext message by the letter M and its ciphertext by C. Computer output ciphertext is a binary data sequence as well, often of the same size as M, sometimes longer. (In the case of combining encryption with compression, C may turn out smaller than M; encryption itself does not give this effect, usually.) One can view encryption as a function E associating with each given plaintext data its ciphertext data. The encryption procedure can then be written as the mathematical formula:

$$E(M) = C.$$

Similarly, the decryption procedure can be thought of as the function

$$D(C) = M,$$

which takes a cipher text C and outputs its plaintext M.

The goal of decrypting an encrypted message is to recapture the input plaintext; hence the following is required:

$$D(E(M)) = M.$$

1.1.2 Algorithms and Keys

Historically, the security offered by a cipher was to a large extent based on keeping secret its encryption/decryption algorithm. Modern cryptography considers that such ciphers do not provide an adequate level of security. For instance, they cannot be used by a larger group of users. A problem arises when someone would like to leave the group, the others would have to change the algorithm. A similar procedure would apply when someone reveals the algorithm. Another serious concern and source of doubt about secret ciphers is due to the impossibility of having the quality of the algorithms, their standardization and implementations, checked by external experts.

A secret cipher algorithm would have to be uniquely designed for each group of users, which excludes the possibility of ready-to-use software or hardware implementations. Otherwise, an adversary would be able to purchase an identical product and run the same encryption/decryption algorithms. Each group of users would have to design and implement their own cipher. If in such a group there was no good cryptographer and cryptanalyst, the group would not know if its cipher was reliable enough.

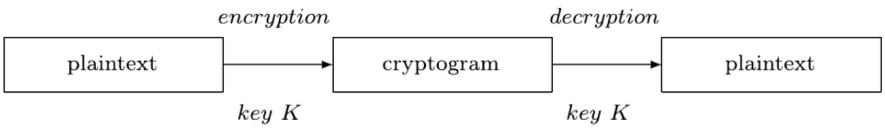

Fig. 1.2 Encryption and decryption with one key

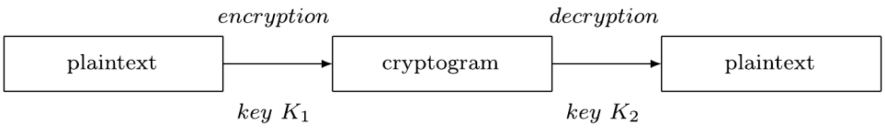

Fig. 1.3 Encryption and decryption with two keys

Modern cryptography solves the above security concerns in such a way that usually the cipher used is publicly known but its encryption/decryption execution uses an extra private piece of information, called a cryptographic key, which is another input parameter. A key is usually denoted by the letter K. It can take one of a wide range or *keyspace* of possible values, usually numbers.

The central idea is that both encryption and decryption functions use a key, and their outputs depend on the keys used, with the following formulae:

$$E(K, M) = C \quad \text{and} \quad D(K, C) = M.$$

In the literature often the following notation appears:

$$E_K(M) = C \quad \text{and} \quad D_K(C) = M$$

where the subscripts indicate the key. Note the following property (see Fig. 1.2):

$$D_K(E_K(M)) = M.$$

Some ciphers use different encryption and decryption keys (Fig. 1.3). This means that the encryption key K_1 is different from the corresponding decryption key K_2. In this case we have the following properties:

$$E_{K_1}(M) = C, \qquad D_{K_2}(C) = M, \qquad D_{K_2}(E_{K_1}(M)) = M.$$

As pointed out above, the security of good ciphers is based on the secrecy of the keys. The algorithms are publicly known and can be analyzed by the best experts. Software and hardware implementations or partial components of the ciphers can be produced and distributed on an industrial scale. Any potential intruder can have access to the algorithms. As long as she does not know our private key and the cipher is good enough, she will not be able to read our cryptograms.

By a cipher or *cryptosystem* we shall mean the two algorithms of encryption and decryption together with (the space of) all the possible plaintexts, cryptograms and keys. There are two general types of ciphers which use keys: symmetric ciphers and public-key ciphers.

Symmetric ciphers, often also called traditional ciphers, *secret-key ciphers*, *single-key algorithms* or *one-key algorithms*, use the same key for encryption and decryption. Here, the same means that each of the two keys can be practically determined (computed) from the other. The keys used in such ciphers have to be kept secret as long as the communication is supposed to be kept secret. Prior to use these keys have to be exchanged over a secure channel between the sender and the receiver. Compromising such a key would enable intruders to encipher and decipher messages and documents.

The basic idea of public key cryptographic algorithms is that encryption and decryption use two different keys, matched in such a way that it is not possible in practice to reconstruct one of them from the other. In such a cryptosystem each user has a unique pair of keys—public and private. The first of them is publicly available. Everybody can use it to encrypt messages. But only the corresponding private key allows decryption. Thus the only person able to run decryption is the one who has the private key.

1.1.3 Strong Cryptosystems Design Principles

An encryption procedure transforms a given *plaintext* document into its enciphered form (*cryptogram*). An encryption algorithm input consists of a plaintext and a key. It outputs the cryptogram. The associated decryption algorithm input consists of a cryptogram and a key, and it outputs the original plaintext.

The basic step in any cryptosystem design is a kind of evaluation of the level of security offered by the resulting system. It can be measured in terms of the computational resources required to break—by any known or foreseeable method—the ciphertexts generated by the system. In the course of many years of research the following conditions have been developed as basic requirements on a strong cryptographic algorithm:

- it should be infeasible to find the plaintext from its cryptogram without knowing the key used;
- reconstructing the secret key should be infeasible.

A good cipher should meet the above criteria also when the cryptanalyst trying to break it has access to some relatively large number of sample plaintexts together with their corresponding cryptograms and knows all the details of the cipher algorithm. It is generally assumed that the cryptanalyst has all the resources (space, devices, technology) feasible currently and in the foreseeable future. Given these assumptions, it should be emphasized strongly that a strong cryptographic system's robustness is based on the secrecy of the private keys.

We do not require that there does not exist a way to break such a cryptosystem. We only require that there is no currently known feasible method to do so. The strong cryptographic algorithms that correspond to the above definition could theoretically be broken, althought in practice it happens very rarely.

The most important rules to date for constructing difficult-to-crack cryptographic code systems were formulated by Claude Elwood Shannon [94] in 1949 (Fig. 1.4). He defined breaking a cipher as finding a method to determine the key and/or cleartext on the basis of its cryptogram. The cryptanalyst can obtain great help from information about certain statistical characteristics of the possible plaintexts such as

Fig. 1.4 Claude E. Shannon

the frequency of occurrences of various characters. On this basis one can determine whether the plaintext is a program written in C, a fragment of prose in Japanese, or an audio file. In each of these cases in every plaintext there is an apparent redundancy of information which can greatly facilitate cryptanalysis. By now many statistical tests based on information theory have been developed, which effectively help in the breaking of ciphers whenever the plaintext statistics parameters are known.

According to Shannon a cryptographic system that allows *excellent protection against unauthorized access* must not provide any statistical information about the encrypted plaintext at all. Shannon proved that this is the case when the number of cryptographic keys is at least as large as the number of possible plaintexts. The key should therefore be of roughly the same or more bits, characters or bytes as the plaintext, with the assumption that no key can be used twice. Shannon's perfect encryption system is called the *single-key system* or *one-time pad*.

According to Shannon to define a mathematical model of a reliable system of strong cryptography it is necessary to be able to reduce the redundancy of plaintext information, so that the redundancy is not carried into the cryptograms. Shannon proposed techniques of *diffusion* and *confusion*, which in practice have been reduced by many crypto designers to some kind of alternation of combining block cipher components with substitutions and permutations.

Claude Elwood Shannon (30 April 1916–24 February 2001) was an eminent American mathematician, founder of information theory, one of the many scholars working during World War II with US military and government agencies as a consultant in the field of cryptology.

1.1.4 Computational Complexity of Algorithms

In this section we introduce the basic concepts of theoretical and practical computational complexity, to the minimum extent that is necessary to understand modern cryptography and the next chapters of this book.[1]

[1]For more on this background topic the reader is referred to [27, 68].

Modern cryptography uses publicly known algorithms. Before their deployment they are subject to objective analysis by independent experts. The private keys are the closely guarded secrets, not the algorithms. Without knowing the appropriate keys no one can encrypt/decrypt documents in any good cryptosystem. According to Shannon's principles the algorithms must be designed in such a way that the complexity of the two tasks previously described as infeasible makes breaking the ciphers practically impossible. Often, however, the high complexity is an estimated upper bound on the performance of one algorithm, with the additional argument that *currently no efficient cipher-breaking algorithm is known.*

An important part of the analysis of encryption and decryption algorithms, and algorithms in general, is their computational complexity, the efficiency of calculations and of solving algorithmic and computational problems and problem instances. In this context, by a computational problem we mean a function of the input data into the output data, that we want to calculate using the analyzed algorithm.

As motivating examples, consider the problem of computing the determinant of an integer-valued matrix and the problem of integer factorization. A given matrix or an integer are called *instances* of these problems, respectively. The bigger the matrix or integer, the more computational resources are needed to calculate the determinant or the prime factors. In general, the bigger the size of the input data, the more resources (time, space, processors) are needed to compute such a problem instance. The complexity of an algorithm is a function of the instance input data size.

One can define the complexity of an algorithm in various ways, however in general it expresses the amount of resources needed by a machine (computer) to perform the algorithm. The resources considered most often are *time* and *space*. For obvious reasons, the amount of these resources can differ greatly from instance to instance depending on several parameters. The time complexity of an algorithm is a function that indicates how much time is needed for its execution. The time here is not measured in seconds or minutes, but in the number of calculation steps, or of bit operations. How many seconds it takes, depends largely on the equipment on which the calculations are performed. Independently from equipment and from the rapid development of computing devices, the complexity of an algorithm is defined as a function expressing how many elementary operations on individual bits have to be performed, in the course of each run of the algorithm, depending on the size of the input data. The time complexity of a given computational problem is the function of the input data size expressing the time complexity of the best algorithm solving the problem.

The time complexity of a given algorithm is defined as the number of elementary operations on input data during one run of the algorithm. By the elementary operations one usually understands the simplest nondecomposable instructions in a certain programming language or in a certain abstract model of computation; e.g., a Turing machine or a (single-core) *Random Access Machine*, RAM. It does not matter what language it is, because what matters is just the proportional order of magnitude of the number of operations, up to a possible (finite) constant multiplicative factor. Without loss of generality, it can be the language Java. Alternatively, one can treat the single instructions (lines) of the algorithm's pseudocode as the elementary operations. In this book we do not refer to anything like that, nor to any abstract machine model. Instead, as elementary we define the arithmetic bit-operations of primary school addition and subtraction of the binary representations of nonnegative integers.

The addition of binary numbers is calculated in the same way as traditional column addition of decimal numbers. You do it by writing one number below the other and adding one column at a time: add up the digits (binary, bits), then write down the resulting 0 or 1, and write down the carry to the next column on the left. Subtraction is calculated just as addition, but instead of adding the binary digits you deduct one bit from the other, and instead of the carry operation you take a *borrow* from the next column.

Multiplication can be done using only the above elementary bit operations. The school division algorithm is much more complicated, but it also refers only to the same elementary operations on single bits. Just add the divisor to some extra parameter (initialized to zero) until it gets bigger than the dividend. Then the number of additions made so far (minus one) makes the resulting quotient of the division, while the dividend minus the final value of the extra parameter (minus the divisor) makes the resulting remainder left over.

In computer architecture the elementary bit operations are implemented in a special section of the central processing unit called the arithmetic logic unit, ALU.

In complexity analysis of a given algorithm it suffices to care about the time of the dominating operation only. That is, the operation performed much longer than all the other operations of the algorithm taken together. In this book, indeed in cryptography in general, the multiplication of two integers is dominating in almost all cases. Sometimes it is integer addition, subtraction, or division. In cryptography, these four basic arithmetic operations can be taken as elementary, the more so as more and more often these operations get hardware implementations in the arithmetic and logical processors, and each of them can be executed in a single clock tick.

Computational complexity is thus a kind of device-independent simplification, an approximation that allows comparisons of the hardness of algorithms (programs) regardless of the machines that can run them. It also neglects a lot of details such as, for instance, the generally much shorter time required for a variety of auxiliary actions, for example, memory register access operations, (sub)procedure calls, passing parameters, etc. It omits all the technical features of the computer on which the calculations are performed. It is generally accepted that the algorithm execution time is proportional to the number of elementary operations performed on the input bits.

Adding two integers m and n written as $|m|$ and $|n|$ binary digits (bits zero or one), respectively, can be done in $\max(|m|, |n|)$ bit-operations. Subtraction has the same estimate. The primary-school multiplication of two $|n|$-bit integers requires at most $|n|^2$ elementary bit-operations. Dividing an $|m|$-bit integer by an $|n|$-bit integer requires at most $|m| \cdot |n|$ time.

In general, the larger the input data, the more resources needed for their processing. However, an analyzed algorithm can do this in a nonuniform, not necessarily monotone, way. The time and space it needs can vary considerably on different input instances of the same size. We distinguish: pessimistic complexity, that is the case of input data on which the analyzed algorithm requires the most resources over all data of that size; expected, or average complexity; and asymptotic complexity, i.e., the limit of the complexity function values on arbitrarily large inputs.

Space complexity refers to how much space on the computer is required. Space complexity of an algorithm (program) is a measure of the amount of memory needed, for example, the number of cells visited on a Turing machine tape. In this book, indeed in modern cryptography in general, the required space is expressed in bits—as the maximum number of bits simultaneously written down (stored) in the course of the analyzed algorithm run.

It is generally considered that a superpolynomial performance of an algorithm, i.e., expressed by a function with asymptotic growth faster than all polynomials with integer coefficients, is infeasible (or intractable) on sufficiently large input data. For example, when we say that there is no known feasible algorithm for integer factorization, we usually mean: no algorithm running in polynomial time.

Theoretical polynomial complexity is not a sufficient criterion for practicality. For example, in 2002 [4] published a polynomial time algorithm checking whether a given natural number is prime (that is, divisible only by 1 and itself). However, the degree of this polynomial is too high for practical use in testing primality of numbers of the size currently interesting in practical applications. These are approximately 1000-bit integers.

One more concept of computational complexity is often called *practical complexity* and measured in seconds on currently available computers and on input data of the size of current interest. Algorithm AKS, mentioned above, has too high practical complexity. Similarly, the currently (August 2012) best attacks on the SHA-1 hash function standard are treated as merely theoretical, because the best of them gives a chance of finding a collision (i.e., two different messages with the same hash) in time corresponding to over 2^{60} SHA-1 evaluations. Nobody can have that much time on currently available computers (without very high extra financial and organizational effort).

The concepts introduced above have been extensively studied in computational complexity theory. The standard reference textbooks are [27, 77].

In modern cryptology the strength of a cipher (in general, a cryptosystem) is usually expressed in terms of the computational complexity of the problem of breaking the analyzed cipher—how much time or space is required to find the secret key or recover the plaintext from its encrypted version with no prior knowledge of the appropriate secret key, even when the cryptanalyst has possibly a large number of pairs plaintext/ciphertext. *How much time is required* refers to the fastest currently known algorithm performing this task. Cryptography can be called cryptocomplexity.

Similarly to time complexity, space complexity is defined as the amount of space required for running an algorithm. It can be measured either by the maximum number of cells in an abstract machine model of the algorithm execution or by the size of actual physical memory space expressed in bits or bytes.

A specific big-oh notation has been introduced for comparison of the rate of growth of functions describing the computational complexity of algorithms. The expression

$$f(n) \in O(g(n))$$

is read and defined as: *function f is at most of order g if and only if there exist positive real c and natural n_o such that for every n greater than or equal to n_o, the value $f(n)$ is at most equal to the product $c \cdot g(n)$*. In symbols it can be written as follows:

$$f(n) = O(g(n)) \Leftrightarrow \exists_{c \in \mathbf{R}_+} \exists_{n_0 \in \mathbf{N}} (n \geq n_0 \Rightarrow f(n) \leq c \cdot g(n)).$$

By way of a simple illustration, we give an estimate of the time cost (computational time complexity) of the algorithms for grade-school addition and multiplication of binary integers.

Consider two binary numbers x and y of bit-length k. Adding these binary numbers is usually realized as k additions of single bits. So, the time complexity is $O(k)$. Multiplying x by y is in the worst case (when y has all ones in the binary notation) $k - 1$ additions of x to x shifted by one bit to the left each time. It requires $(k - 1) \cdot k = k^2 - k$ additions of single bits. So, the time complexity of integer multiplication is $O(k^2)$, i.e., quadratic.[2]

Basic Complexity Classes

- $O(1)$—constant complexity—the algorithm performs a constant number of steps no matter what size its input data is.
- $O(n)$—linear complexity—for each input data size $|n|$, the algorithm performs a number of steps proportional to $|n|$. The growth rate of the running time is linear w.r.t. the input data size.
- $O(n^2)$—quadratic complexity—the algorithm's running time is proportional to the square of the input data size $|n|$.
- $O(n^3)$, $O(n^4)$, ...—polynomial time complexity.
- $O(\log n)$—logarithmic complexity.
- $O(2^n)$—exponential complexity—the algorithm performs a constant number of operations for every subset of the size n input data.
- $O(n!)$—the algorithm performs a constant number of steps on each permutation of the size n input data.

Algorithms of exponential or higher complexity are infeasible. Their running time grows rapidly with increasing n. On large input data size, their running time gets monstrously, inconceivably long. If we imagine that we have two computers, one of which can perform a million operations per second, and the second is a million times faster (which gives the performance of 10^{12} operations per second), the time required by an exponential algorithm (with time complexity $O(2^n)$) is shown in Table 1.1.

[2]See Table 2.1 in [68].

Table 1.1 Running time of an algorithm of class $O(2^n)$

Input size n	20	50	100
10^6 op./s	roughly 1 s	about 35 years	about $4 \cdot 10^{16}$ years
10^{12} op./s	about 10^{-6} s	about 18.5 min	about $4 \cdot 10^{10}$ years

Table 1.2 Enumerating letters of the Latin alphabet

A	B	C	D	...	W	X	Y	Z
↓	↓	↓	↓	...	↓	↓	↓	↓
1	2	3	4	...	23	24	25	26

The table clearly shows that even a big increase of hardware computational power cannot beat the device-independent complexity of an algorithm. The time required for its execution can become unimaginably large. For example, for $n = 100$, in both cases probably we would never see the results.

Here we are talking about asymptotic complexity bounds. This does not exclude the possibility of particular input data instances of even very large size that can be computed very fast by an exponential time complexity algorithm. In practice an algorithm's running time, measured in seconds, behaves irregularly, with many downs and ups. Over the last decade or two a whole domain of research has arisen with significant successful real-life applications of some asymptotic exponential time algorithms, often on some industrial-scale input sizes.

1.2 Simple Stream Ciphers

1.2.1 Caesar Cipher

One of the simplest encryption algorithms is the so-called Caesar cipher, already used by the Roman army.[3] Let us assume that the texts we want to encrypt are written in the 26-letter Latin alphabet excluding capitalization. We assign consecutive positive integers to symbols of the alphabet (see Table 1.2).

The idea of the algorithm consists in replacing each symbol of a plaintext with the symbol whose number is greater by three computing modulo 26 (the last three letters of the alphabet X, Y, Z are replaced with, respectively, A, B, C). If we denote the integer assigned to a letter x by L_x, then we can write the replacement operation mathematically in the following way (addition is performed modulo 26, however we do not replace 26 with 0):

$$C(L_x) = L_x + 3.$$

[3]One can find comprehensive information about this and many other ciphers used in the past in [54]. The history of contemporary cryptography is well discussed in [29]. See also Sect. 7.3 in [68].

An important feature of the Caesar cipher is the distributivity of encryption over the sequence of symbols that forms a plaintext. Formally, we can present it as follows:

$$C(xy) = C(x)C(y).$$

Obviously, it is easy to notice that the algorithm can be modified by changing the number of positions by which symbols are shifted.

Although very simple, the Caesar cipher is a symmetric key algorithm. The key in the original version of the Caesar cipher is equal to 3 (the shift parameter). In the next section we will generalize Caesar's method to all possible permutations of the alphabet.

What is interesting is that the Caesar cipher was used even during World War I by the Russian army. The applied shift parameter was equal to 13.

From the theoretical point of view the cost of encrypting a k-bit ciphertext is linear—it equals $O(k)$. Of course, nowadays it is very easy to break the Caesar cipher even with an unknown shift parameter. Actually, it can be broken without using a computer—a piece of paper and a pen are enough.

1.2.2 XOR Encryption (Vernam Cipher)

Now we are going to present an encryption algorithm known in the literature as the Vernam cipher or simply *XOR*. This algorithm, which requires some mathematical knowledge, uses the *XOR* function. Formally, the latter is a Boolean function (i.e., a function $f : \{0, 1\} \times \{0, 1\} \to \{0, 1\}$) that satisfies the following conditions:

$$f(0, 0) = f(1, 1) = 0 \quad \text{and} \quad f(0, 1) = f(1, 0) = 1.$$

One can easily notice some of its properties. For arbitrary $x, y, z \in \{0, 1\}$ the following equations hold:

1. $f(x, y) = f(y, x)$ (commutativity),
2. $f(x, x) = 0$,
3. $f(x, 0) = x$ (0 is a neutral element of f).
   It can be proved that the function is associative, i.e.,
4. $f(x, f(y, z)) = f(f(x, y), z)$.

When we consider the function *XOR* as an operation defined on the set $\{0, 1\}$, then the above equations can be expressed as follows:

(a) $0 \; XOR \; 0 = 1 \; XOR \; 1 = 0$,
(b) $0 \; XOR \; 1 = 1 \; XOR \; 0 = 1$,
(c) $x \; XOR \; y = y \; XOR \; x$,
(d) $x \; XOR \; x = 0$,
(e) $x \; XOR \; 0 = x$,
(f) $(x \; XOR \; (y \; XOR \; z)) = ((x \; XOR \; y) \; XOR \; z)$.

If we take two sequences of bits $X = (x_1, x_2, \dots, x_n)$ and $Y = (y_1, y_2, \dots, y_n)$, then by $X\ XOR\ Y$ we mean the sequence of values of the *XOR* operation on consecutive entries of sequences X and Y:

$$\begin{aligned} X\ XOR\ Y &= (x_1, x_2, \dots, x_n)\ XOR\ (y_1, y_2, \dots, y_n) \\ &= (x_1\ XOR\ y_1, x_2\ XOR\ y_2, \dots, x_n\ XOR\ y_n). \end{aligned}$$

The above properties allow us to execute the following encryption algorithm. In order to encrypt a k-bit sequence we divide it into blocks with n elements each (in case there are not enough bits for the last block, we fill it with, e.g., zeroes). As a key we take an arbitrary (random) bit sequence of length n: $K = (k_1, k_2, \dots, k_n)$. The ciphertext for a block $A = (a_1, a_2, \dots, a_n)$ is simply expressed by:

$$\begin{aligned} C &= K\ XOR\ A = (k_1\ XOR\ a_1, k_2\ XOR\ a_2, \dots, k_n\ XOR\ a_n) \\ &= (c_1, c_2, \dots, c_n). \end{aligned}$$

The above-mentioned features of the *XOR* operation enable us to specify the method by which the ciphertext is decrypted. In order to reconstruct the block A it is sufficient simply to execute:

$$K\ XOR\ C = K\ XOR\ (K\ XOR\ A) = A.$$

Indeed, for any entry of the ciphertext c_i $(i = 1, \dots, n)$ we have:

$$\begin{aligned} k_i\ XOR\ c_i &= k_i\ XOR\ (k_i\ XOR\ a_i) = (k_i\ XOR\ k_i)\ XOR\ a_i \\ &= 0\ XOR\ a_i = a_i. \end{aligned}$$

The second equation in the above notation follows from the associativity of *XOR*, while the third and the fourth ones follow from properties 2 and 3, respectively.

The cost of this encryption is very low and, as in the case of the Caesar cipher, is equal to $O(k)$ for a k-bit ciphertext. Let us notice that if the key applied is appropriately long (for instance of several bits), then the cipher provides a high security level. Due to its construction it is not vulnerable to any known attacks but the brute force technique. The latter, however, is actually unfeasible in the case of long keys because of its time complexity. Moreover, if the key length is as long as the length of the ciphertext and the key is used only once, then the Vernam cipher turns out to be an ideal cipher that cannot be broken. It can easily be seen that one can adjust a key for a ciphertext of a given length and any plaintext of the same length. The complexity of the brute force method for breaking encryption with a k-bit key equals $O(2^k)$.

Despite its simplicity, the Vernam cipher is still applied: WordPerfect, a very popular text editor, uses it in an only slightly modified version. This encryption scheme is used in many other ciphers, e.g., DES and AES, as well. It is also applied in secure communication with the use of the first quantum communication networks.

1.3 Simple Block Ciphers

1.3.1 *Permutations*

The Caesar cipher is one of the simplest substitution ciphers. Replacing each alphabet letter with another one is performed in a regular manner—it depends on the alphabetical order. One can easily see that this assignment may be done arbitrarily.

Let us now recall some elementary mathematical facts. Given a finite set X, any one-to-one function $f : X \to X$ is called a permutation.

If the cardinality of X is equal to n, then there are $n!$ permutations (one-to-one functions) on X.

The Caesar cipher can be generalized to all possible permutations of the alphabet. In this situation a key is given by a 26-element non-repetitive sequence of integers from 1 to 26. Such a sequence determines the substitution that has to be applied in order to encrypt a message.

It can be seen that for an alphabet with 26 characters the number of all permutations equals 26!, which amounts to about $4 \cdot 10^{26}$, a number that is large even for modern computers. Verification of all possible keys (sequences) would take a great deal of time.

It turns out, however, that substitution ciphers can easily be broken using so-called frequency analysis. Certain letters and combinations of letters occur more often than others. Therefore, it is easy to check which symbols appear in a given ciphertext and with what frequency (for obvious reasons it is better to work with suitably long ciphertext messages).

1.3.2 *Transpositions*

Another simple method of encryption consists of inserting a plaintext in adequately defined (e.g., as geometrical shapes) blocks. Such a block may be represented as an array with a given number of rows and columns. A plaintext is placed in the matrix row-wise. Then the matrix is transposed (rows are replaced with columns). The ciphertext should be read row-wise after such a transposition.

Example 1.1 Let us consider the following plaintext: *ITISASIMPLEEXAMPLE*. If we decide to encrypt the message with a simple transposition cipher using, for instance, a 3×6-matrix, then we obtain the matrix with the plaintext (Table 1.3).

After performing the transposition, we get the matrix presented in Table 1.4:

Finally, we get the ciphertext: *ISILXPTAMEALISPEME*.

In this system, the key is given by the matrix dimensions. For example, in the case of a 6×3-matrix the ciphertext is as follows: *IIXTMAIPMSLPAELSEE*.

Table 1.3 Matrix with the plaintext

I	T	I
S	A	S
I	M	P
L	E	E
X	A	M
P	L	E

Table 1.4 Transposition

I	S	I	L	X	P
T	A	M	E	A	L
I	S	P	E	M	E

Table 1.5 Table for constructing encryption templates

0	1	1	2	3	3	0	3
1	3	2	2	3	0	0	3
3	0	3	2	0	1	0	3
2	1	0	2	1	2	3	2
1	2	1	1	0	3	1	2
2	1	0	1	0	0	2	3

1.3.3 Example of a Simple Transposition Cipher

We are going to explain the principle of constructing such a cipher by a simple, however non-trivial, example. Now, a plaintext and a ciphertext are expressed over a 32-symbol alphabet:

ABCDEFGHIJKLMNOPQRSTUVWXYZ . : , ; -

which consists of 26 capital letters of the Latin alphabet, a space, a period, a colon, a comma, a semicolon, and a hyphen. We assume that a block of a plaintext contains 48 symbols from the above alphabet which are arranged in 6 rows. Given these assumptions, we can apply the following encryption algorithm for a transposition cipher:

1. Create a table with 6 rows and 8 columns and fill it with integers 0, 1, 2 and 3 in such a way that the number of zeroes, ones, twos and threes is the same and equals 12. At the same time, try to minimize the number of adjacent cells containing the same integers. These conditions are satisfied in Table 1.5. Such a table can easily be found, for instance by means of the following program written in Pascal:

```
var i, k, r: Byte;
    l: array[0..3] of Byte;
begin
   repeat
      Randomize;
```

Table 1.6 Block of the plaintext

A	D	I	U	N	K	T	-
L	E	C	T	U	R	E	R
A	F	O	R	Y	Z	M	-
A	P	H	O	R	I	S	M
A	K	T	O	R	K	A	-
A	C	T	R	E	S	S	.

```
        for i:=0 to 3 do l[i]:=0;
        for i:=1 to 6 do
        begin
           for k:=1 to 8 do
           begin
              r:=Random(4); Inc(l[r]);
              Write(r:2);
           end;
           WriteLn
        end;
        WriteLn;
     until (l[0]=l[1]) and (l[0]=l[2]) and(l[0]=l[3])
  end.
```

2. Cut four rectangular pieces of paper that have the same size as the tables from the first step of the algorithm. Then create four templates, numbered 0, 1, 2 and 3. In each template cut 12 holes covering the cells which contain the number of the template.
3. The cryptogram will be written on a blank piece of paper that is exactly the size of the template. Put this piece of paper consecutively against the templates prepared in the second step and insert 48 letters of the plaintext into empty cells (4 times 12 letters).

The decryption algorithm is easier and consists of only one step:

1. Put the templates consecutively against the ciphertext and read the symbols of the plaintext from the holes and write them down in the matrix of six rows and eight columns.

Let us take the part of the Polish-English dictionary shown in Table 1.6 as a plaintext. After encrypting this text according to the algorithm we obtain the cryptogram given in Table 1.7. It can easily be verified that by decrypting this ciphertext according to the algorithm presented above one obtains the proper plaintext.

The just-described substitution block cipher, whose keys are given by encryption templates and the order of their use, can be specified in a more formal manner. If we enumerate 48 symbols of a plaintext consecutively, then the encryption process obtained by applying templates takes the form of the table presented in Table 1.8, from which it follows that the first symbol of the ciphertext corresponds to the first symbol of the plaintext, the second symbol corresponds to the 13th symbol of the

Table 1.7 Block of the cryptogram of the transposition cipher

A	U	R	A	R	K	D	A
E	-	P	H	A	I	U	C
T	N	R	O	K	R	T	E
R	A	-	I	F	S	S	M
O	A	R	Y	L	S	Z	K
T	M	E	-	C	T	O	.

Table 1.8 Encryption permutation

1	13	14	25	37	38	2	39
15	40	26	27	41	3	4	42
43	5	44	28	6	16	7	45
29	17	8	30	18	31	46	32
19	33	20	21	9	47	22	34
35	23	10	24	11	12	36	48

Table 1.9 Decryption permutation

1	7	14	15	18	21	23	27
37	43	45	46	2	3	9	22
26	29	33	35	36	39	42	44
4	11	12	20	25	28	30	32
34	40	41	47	5	6	8	10
13	16	17	19	24	31	38	48

plaintext, the third to the 14th, etc. Using this table, which represents a permutation of the set $\{1, 2, \ldots, 48\}$, one may thus determine the encryption procedure more precisely than by means of encryption templates. What is more, considering this permutation as an encryption key, it is possible to specify the exact number of all possible keys for the above cipher, which is equal to

$$48! = 124139155925360726708622890473733750385214863546777600000000000,$$

which amounts to about $1.241391559 \cdot 10^{62}$. Not a small number, especially when compared to the number of all atoms on our planet which is estimated to be 10^{51}.

In order to decrypt ciphertexts of the cipher in question one has to apply the permutation that is inverse to the encryption permutation presented in Table 1.9. At first sight it seems that the cipher may be broken by trying 48! permutations one by one. Assuming that we would be able to test a million permutations per second (even such an assumption is too optimistic for the current state of technology), it would take around 10^{47} years to break a ciphertext. On the other hand, the age of the universe is estimated to be 10^{11} years. However, if cryptanalysts apply statistical tests, then breaking such a cipher takes them just a couple of seconds.

Table 1.10 Symbols of the plaintext and corresponding symbols of the cryptogram

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
X	H	Y	U	Z	R	.	V	Q	A	:	W	T	,	B	C

Q	R	S	T	U	V	W	X	Y	Z		.	:	,	;	-
S	D	;	E	I	G	L	F	J	K	-	M	P	N	0	

Table 1.11 Table with ciphertext symbols and plaintext symbols corresponding to them

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
J	O	P	R	T	X	V	B	U	Y	Z	W	.	,	;	:

Q	R	S	T	U	V	W	X	Y	Z		.	:	,	;	-
I	F	Q	M	D	H	L	A	C	E	-	G	K	N	S	

Table 1.12 The block of the substitution cipher cryptogram obtained using Table 1.10 which corresponds to the block of the plaintext presented in Table 1.6

X	U	Q	I	,	:	E	
W	Z	Y	E	I	D	Z	D
X	R	B	D	J	K	T	
X	C	V	B	D	Q	;	T
X	:	E	B	D	:	X	
X	Y	E	D	Z	;	;	M

1.3.4 Example of a Substitution Block Cipher

In the easiest case the operation of substitution block ciphers consists in replacing symbols of a plaintext with other symbols one by one. Hence, when performing the encryption algorithm we have to apply a table which contains the rule of this substitution, i.e., the table represents some permutation of the alphabet. Of course, in order to decrypt messages one uses the inverse permutation.

Let us assume that in the considered case the same alphabet as in the example of a transposition cipher is used, the 48-symbol block of a plaintext is the same as previously, and the substitution table presented in Table 1.10 is applied during encryption. Then one obtains immediately a table for use during decryption (Table 1.11).

Now, we can illustrate the process of creating a cryptogram using the substitution cipher. If the plaintext is given by the block presented in Table 1.6, then, after applying Table 1.10 and executing 48 symbol substitution operations, one gets the cryptogram shown in Table 1.12. The obtained cryptogram may yet be easily decrypted since its symbols occur in the same order as the corresponding symbols in the plaintext. For this reason cryptanalysts break substitution ciphers very quickly.

1.3.5 Example of a Product Cipher

A product cipher applies two encryption algorithms sequentially: it starts by encrypting a plaintext by means of the first algorithm, then the obtained cryptogram of the key without repetitions. The empty entries of the matrix are filled with the remaining letters of the alphabet (i.e., those that do not appear in the key).

We obtain the following matrix:

C	R	Y	P	T
O	G	A	H	I
S	K	B	D	E
F	L	M	N	Q
U	V	W	X	Z

The cryptogram is created from a plaintext by appropriate, i.e., with respect to the matrix, substitutions of pairs of letters (if the text has an odd number of symbols, then it is completed with any symbol).

Let us consider the following plaintext: *ENCRYPTIONKEYS*. At the first stage of encryption, the sequence of letters is divided into pairs *EN CR YP TI ON KE YS*. Each pair is transformed with respect to the rectangle contained in the matrix determined by the letters that form the pair (according to the row-wise order). For instance, the pair *EN* is converted to the pair *QD* (as these two letters form the two remaining corners of the rectangle defined by the digraph *EN*). If encrypted letters are placed in the same row/column or they are equal, then we choose the symbols to their right, e.g., *AW* is converted to *HX*, while *FL* is converted to *LM*.

D	E
N	Q

A	H
B	D
M	N
W	X

The whole ciphertext is as follows: *QDRYPTCOFHBSBC*.

Let us present another example of a bigram system.

Example 1.3 Keys are given by two independent permutations of a 25-letter alphabet. We place them in two square matrices (of dimension 5×5).

A	K	N	Y	E		E	R	T	B	O
R	D	U	O	I		W	I	U	M	K
Q	S	W	B	G		N	D	A	S	F
H	C	X	T	Z		Q	X	G	Z	V
V	M	L	P	F		H	Y	P	L	C

A plaintext, for instance *TODAYISABEAUTIFULDAY*, is divided into several rows of a fixed length:

T	O	D	A	Y	I	S	A	B	E
A	U	T	I	F	U	L	D	A	Y

Index

DOI 10.1007/978-3-642-41386-5, © Springer-Verlag Berlin Heidelberg 2013