Chapter 2

# Cryptography and Complexity Theory

## 1. Introduction

There are two main approaches to defining the security of a cryptosystem or a cryptographic protocol (below we use the unifying term a *cryptographic scheme* for both notions) in theoretical cryptography, namely, the information theory approach and the complexity theory approach. Under the information theory approach, we assume that the adversary does not have even theoretical possibility to obtain the required information. The classical example here is the Vernam cipher with a one-time pad, which is perfectly secure against a passive adversary.

In practice, the security level of the overwhelming majority of cryptographic schemes is not that high. Moreover, it is usually easy to suggest an algorithm for the adversary to perform the information retrieval task, at least in principle. Consider the following example.

**Example 1** (Public key cryptosystems)**.** A *public key cryptosystem* is completely determined by three algorithms: the key generation algorithm, the encryption algorithm and the decryption algorithm. The key generation algorithm $G$ is public; anybody may feed it with a random string $r$ of required length and get a pair of keys $(K_1, K_2)$ as

the output. The open key $K_1$ becomes public, and the secret key $K_2$ and the random string $r$ remain private. The encryption algorithm $E_{K_1}$ and the decryption algorithm $D_{K_2}$ are such that if $(K_1, K_2)$ is a pair of keys generated by the algorithm $G$, then $D_{K_2}(E_{K_1}(m)) = m$ for an arbitrary plaintext $m$. We assume, for simplicity, that the ciphertext has the same length $n$ as the plaintext. Also, we assume that the plaintext, the cryptogram, and both keys are binary strings.

Now, suppose an adversary attacks this cryptosystem. He knows the public key $K_1$, but not the secret key $K_2$. After tapping into a cryptogram $d$, the adversary tries to find the message $m$ such that $d = E_{K_1}(m)$. Since the encryption algorithm is public, it is possible simply to search through all messages $m_i$ of length $n$, to apply the encryption algorithm to each such message, and to compare the encrypted message $d_i = E_{K_1}(m_i)$ with $d$. The message $m$ such that $d = E_{K_1}(m)$ is the required one. With a bit of luck, the answer will be found sufficiently quickly. In the worst case the search will take time proportional to $2^n T(n)$, where $T(n)$ is the time required for the computation of $E_{K_1}$ on a message of length $n$. If the message length is about 1000 bits, then no computer can carry on such an exhaustive search.

This is the first way to attack a cryptosystem. This algorithm of searching for the plaintext is called the exhaustive search algorithm, or the brute force method. Another simple algorithm is the try-and-guess method. This obvious algorithm requires very short calculations, but the success probability is extremely low (if the text is sufficiently long). In practice, an adversary can attack a cryptosystem in various ways and use much more sophisticated techniques to find the plaintext. It is natural to view a cryptosystem as secure if any such algorithm requires an amount of calculations not available in practice, or if the probability of success is negligible. (We must take into account that the adversary can use randomized algorithms as well as deterministic ones.) This is the complexity theory approach to the security definition. In order to make it precise we must

1) give a formal definition of a scheme of a given type;

2) give a formal definition of the security of this scheme;



3) know how to prove that a scheme of a given type is secure.

A number of problems arises on this way.

First, in a specific cryptographic scheme the values of parameters are fixed. For example, a standard cryptosystem uses keys of length, say, 256 or 512 bits. On the contrary, the complexity analysis requires an asymptotic setting. In this approach, mathematical models of cryptographic schemes are studied. These models depend on a parameter, called the security parameter, which can take arbitrary large values (usually it runs over all natural numbers).

Second, the definition of security depends on the task the adversary tries to perform and on the information about the scheme available to him. The definition of security must be given and analyzed in each case separately.

Third, we state precisely what amount of calculations should be considered nonrealizable. The discussion above implies that this value cannot be constant; it must be a function of the growing security parameter. According to Edmond's thesis, an algorithm is considered to be efficient if the computation time is bounded by a polynomial in the input length (in the security parameter in our case). Otherwise the computations by means of the given algorithm are considered to be infeasible. Note also that the cryptographic schemes themselves *must be* efficient, i.e., all the required calculations should take at most polynomial time.

Finally, we must give a bound for the negligible probability. The standard cryptographic agreement treats the probability as negligible if it does not exceed $1/p(n)$ for an arbitrary polynomial $p$ and for all sufficiently large values of the security parameter $n$.

Accepting the definitions above, we reduce the problem to proving the nonexistence of a polynomial time algorithm performing the adversary's task. However, here we encounter an extremely serious obstacle: the modern state of complexity theory does not allow one to justify superpolynomial lower bounds for problems of a given class. This means that all the known proofs of security are based on some unproved assumptions. This is why the research is usually concentrated on the search for the weakest possible sufficient

(or necessary and sufficient, if available) conditions for the existence of secure schemes of each type. The assumptions are usually either general (or complexity theory based) assumptions, or number theory assumptions, that is, assumptions concerning the intractability of some number theory problems. Both these types are usually called cryptographic.

Below, we review briefly some interesting mathematical objects arising on the boundary between complexity theory and cryptography. A more detailed survey can be found in [**5**].

## 2. Cryptography and the P ≠ NP conjecture

Even mathematicians who are not specialists in complexity theory are usually familiar with the classes P and NP and the famous P ≠ NP conjecture.

Let us recall briefly the necessary notions. Let $\Sigma^*$ denote the set of all finite sequences in the two-letter alphabet $\Sigma = \{0, 1\}$. In complexity theory, subsets $L \subseteq \Sigma^*$ are usually called *languages*. A Turing machine[1] $M$ is called *polynomial time* if there is a polynomial $p$ such that $M$ stops at each input word of length $n$ after executing at most $p(n)$ operations. A Turing machine $M$ *recognizes* a language $L$ if it stops in the accepting state for each input word $x \in L$, and it stops in the rejecting state for each input word $y \notin L$. The class $P$ consists of all languages recognizable by a polynomial time Turing machine. A function $f : \Sigma^* \to \Sigma^*$ is *computable in polynomial time* if there exists a polynomial time Turing machine such that, taking a word $x \in \Sigma^*$ as an input, the word $f(x)$ is written on the tape when it stops. A language $L$ belongs to the class NP if there is a predicate $P(x, y) : \Sigma^* \times \Sigma^* \to \{0, 1\}$ computable in polynomial time and a polynomial $p$ such that $L = \{x | \exists y P(x, y) \& |y| \leq p(|x|)\}$. Thus, $L$ belongs to the class NP if for any word of length $n$ in $L$ we can guess a string of length polynomial in $n$ and then verify our conjecture by means of the predicate $P$. Obviously, P ⊆ NP. Whether this inclusion is strict, is one of the most famous unsolved mathematical

[1] For a good introduction to Turing machines, see, e.g., [**69**, Chapter 1]. (*Added in translation.*)



problems.[2] The majority of researchers think that the inclusion is indeed strict (that is, P ≠ NP). The class NP contains a subclass of languages of maximal complexity (so-called *NP-complete* languages); an NP-complete language is polynomial only if P=NP.

Below, we shall also need the notion of a probabilistic Turing machine. The new state of an ordinary Turing machine (which is also called *deterministic*) is totally determined by the current state and the tape symbol under the head. The new state of a probabilistic machine may depend also on a random variable taking values 0, 1 with probability 1/2. A different approach consists in considering an additional tape containing an infinite random binary string. The random tape can be read in only one direction, and the new state of the machine can depend on the symbol under the head on the random tape.

Now let us turn to the following natural question: is the validity of the P ≠ NP conjecture a necessary and sufficient condition for the existence of secure cryptographic schemes?

The necessity of this condition is obvious in many cases. Let us return to Example 1 at the beginning of this chapter. Define the following language:

$$L = \{(K_1, d, i) \mid \text{there is a message } m \text{ such that } E_{K_1}(m) = d \text{ and the } i\text{th bit of } m \text{ is } 1\}.$$

It is clear that $L \in$ NP: instead of the exhaustive search described in the Introduction we can simply guess the plaintext $m$ and verify whether $E_{K_1}(m) = d$ and the $i$th bit of $m$ is 1 in polynomial time. If these statements are true, then the word $(K_1, d, i)$ is approved; otherwise it is rejected.

If P = NP, then there is a deterministic algorithm recognizing the language $L$. Taking $K_1$ and $d$ as the input this algorithm determines the plaintext $m$ bit by bit. Therefore, this cryptosystem is insecure.

The same approach (guess the secret key and verify the correctness of the guess in polynomial time) is applicable, in principle, to

[2] See, for example, the URL of the Clay Mathematics Institute, http://www.claymath.org/prizeproblems/index.htm.

other cryptographic schemes as well. However, some cases cause technical difficulties related to the nonunique recovering of the required value (the plaintext, the secret key, and so on).

Turning to the sufficiency of the condition P $\neq$ NP, it is natural to choose an NP-complete problem and to use it for the construction of a cryptographic scheme such that the corresponding break-in problem is NP-complete. Such attempts, related first of all to cryptographic systems with a public key, were made in the early eighties, but failed. These attempts led to the following conclusion: even if P $\neq$ NP, any NP-complete problem may happen to be difficult only on a certain infinite sequence of input words. In other words, the definition of the class NP measures the complexity in the worst case, whereas the security of a cryptographic scheme requires adversary's task to be complex "almost everywhere". Hence, it became clear that the N $\neq$ NP assumption must be replaced with a much more restrictive one. For this restrictive assumption, the existence of one-way functions was chosen.

## 3. One-way functions

Informally speaking, a one-way function is an efficiently computable function such that there are no efficient algorithms for inverting it.[3] By inverting a function we understand a process of finding an (arbitrary) preimage of a given value in the target space (note that, generally speaking, it may happen that there is no inverse function).

Since the notion of one-way function is central in mathematical cryptography, we present a formal definition.

Let $\Sigma^n = \{0,1\}^n$ denote the set of all binary strings of length $n$. A function $f$ is a family of functions $\{f_n\}$, $f_n : \Sigma^n \to \Sigma^m$, where

[3]There exists the following real-life analog of the notion of one-way function. Suppose we have a large telephone book where subscribers' names are listed in the alphabetical order. Then it is rather easy to find a telephone number given the name. But what about the opposite task: to find the name given a telephone number? The same example can be used to clarify the intuition behind the notion of trapdoor function (see Chapter 1). Now we have two telephone books. The first is ordered by names as above, the second, by telephone numbers. The first book is made publicly available, while we keep the second as our secret. The task to find subscriber's name given a telephone number remains intractable for anyone but us.



$m = m(n)$. For simplicity we assume that $n$ runs over the entire set of positive integers, and that each mapping $f_n$ is defined everywhere.

A function $f$ is *honest* if there is a polynomial $q$ such that $n \leq q(m(n))$ for all $n$.

**Definition 1.** An honest function $f$ is called a *one-way function* if the following conditions are satisfied.

1) There is a polynomial algorithm producing the output $f(x)$ for each input $x$.

2) Let $A$ be an arbitrary polynomial probabilistic Turing machine and $x$ a random string from $\Sigma^n$ (we use the notation $x \in_R \Sigma^n$). Then for any polynomial $p$ and for all sufficiently large $n$ we have

$$\Pr\{f(A(f(x))) = f(x)\} < \frac{1}{p(n)}.$$

The probability here is determined by the random choice of $x$ and the random data used by the machine $A$.

Condition 2) means essentially that any polynomial time probabilistic Turing machine $A$ can reconstruct $x$ from a given $y$ only with a negligibly small probability.

Note that the honesty requirement is important. If $f$ shrinks input words too much, then $A$ may require more than polynomial time in $m$, the length of $f(x)$, simply for writing out the input word.

It is clear that if a one-way function exists, then P $\neq$ NP. It can happen, however, that P $\neq$ NP and there are no one-way functions.

The existence of one-way functions is necessary for the security of many types of cryptographic systems. In some cases this fact is rather simple. Let us return to Example 1. Consider a function $f$ such that $f(r) = K_1$. The algorithm $G$ computes it in polynomial time. Let us show that if $f$ is not a one-way function, then the cryptosystem is insecure. Suppose there exists a polynomial time algorithm $A$ inverting $f$ with probability at least $1/p(n)$ for some polynomial $p$. Here $n$ is the length of $K_1$. The adversary can use the key $K_1$ as the input for the algorithm $A$ and obtain some preimage $r' \in f^{-1}(K_1)$ with probability at least $1/p(n)$. Using $r'$ as the input for $G$ the adversary obtains a pair of keys $(K_1, K_2')$. Although $K_2'$ does

not necessarily coincide with $K_2$, we have $D_{K_2'}(E_{K_1}(m)) = m$ for any plaintext $m$, by definition of a cryptosystem. Since the probability of finding $K_2'$ is at least $1/p(n)$, and this value is not negligible, this cryptosystem is insecure.

For other cryptographic schemes, the proof is not so simple. The necessity of the existence of one-way functions for security of a number of cryptographic schemes is proved by Impagliazzo and Luby in [**38**].

It follows from the discussion above that the assumption about the existence of one-way functions is the weakest possible cryptographic assumption sufficient for the existence of secure cryptographic schemes of different types. The efforts of specialists in theoretical cryptography are concentrated on understanding whether this assumption is indeed sufficient. The following example shows that this problem is in fact complicated. Suppose $f$ is a one-way function, and we are trying to construct a *cryptosystem with a secret key*. In such a system, there is only one key, and this key is known both to the sender of the encrypted text and to the recipient. The encryption algorithm $E_K$ and the decryption algorithm $D_K$ depend on this secret key $K$ and satisfy the condition $D_K(E_K(m)) = m$ for any plaintext $m$. It is clear that if the adversary intercepts the ciphertext $d = f(m)$, then he can reconstruct $m$ only with a negligible probability. But if $f$ is a one-way function, then how can the legal recipient of $d$ reconstruct the text $m$? Besides, if $f$ is a one-way function, this only means that the adversary cannot compute the entire message. And this is a rather low security level. One would expect that an adversary knowing the cryptogram $d$ should not be able to compute even a single bit of the plaintext.

By now, it is proved that the existence of one-way functions is a necessary and sufficient condition for the existence of secure cryptosystems with a secret key, as well as secure cryptographic protocols of various types, including electronic signature protocols. On the other hand, the result of Impagliazzo and Rydich [**40**] confirms rather strongly that the security of some types of cryptographic schemes (including key distribution protocols of Diffie–Hellman type) requires more restrictive assumptions than simply the existence of one-way



functions. However, we cannot explain this result in the present chapter.

## 4. Pseudorandom generators

The Vernam cipher uses one-time pads. Is it possible to get rid of this defect at the expense of some weakening of the security? Here is one way to solve this problem. Suppose the sender and the recipient have a common secret key $K$ of length $n$ and generate a sequence $r = g(K)$ of some length $q(n)$, where $q$ is a polynomial, by means of a sufficiently efficient algorithm $g$. Such a cryptosystem (we denote it by $Cr$) allows one to encrypt a single message or a sequence of messages $m$ of length up to $q(n)$ bits using the formula $d = r \oplus m$, where $m$ is the bitwise sum modulo 2. The decryption is given by the formula $m = d \oplus r$. Shannon's results imply that such a cryptosystem is not perfectly secure, that is, not secure against an arbitrary adversary (this statement is easy to verify directly). But what if we deal with a polynomially restricted adversary having only polynomial time probabilistic algorithms at his disposal? What conditions should the sequence $r$ and the algorithm $g$ satisfy to make the cryptosystem $Cr$ secure? The search for an answer to this question has led to the notion of pseudorandom generator introduced by Blum and Micali in [**12**].

Let $g : \{0,1\}^n \to \{0,1\}^{q(n)}$ be a function computable in polynomial time in $n$. Such a function is called a *generator*. Informally, a generator $g$ is *pseudorandom* if the sequences it generates cannot be distinguished from random sequences of the same length $q(n)$ by a polynomial time probabilistic algorithm. The formal definition is as follows.

Let $A$ be a polynomial time probabilistic Turing machine fed with binary strings of length $q(n)$ and generating a single bit. Set

$$P_1(A,n) = \Pr\{A(r) = 1 \mid r \in_R \{0,1\}^{q(n)}\}.$$

Here the probability is the function of the random choice of the message $r$ and random values the machine $A$ uses. Let

$$P_2(A,n) = \Pr\{A(g(s)) = 1 \mid s \in_R \{0,1\}^n\}.$$

Here the probability is the function of the random choice of the string $s$ and random values the machine $A$ uses. We emphasize that the function $g$ is computed by a deterministic algorithm.

**Definition 2.** A generator $g$ is called a *cryptographically secure pseudorandom generator* if for each polynomial time probabilistic Turing machine $A$, every polynomial $p$, and all sufficiently large $n$ we have

$$|P_1(A,n) - P_2(A,n)| < 1/p(n).$$

For brevity, cryptographically secure pseudorandom generators will be called just pseudorandom generators. This abbreviation is standard in the literature.

It is easy to show that the existence of pseudorandom generators implies the existence of one-way functions. Indeed, $g$ itself must be a one-way function. We leave the proof of this simple fact as an exercise to the reader. The question on whether the existence of pseudorandom functions is also sufficient for the existence of pseudorandom generators remained open for a long time. In 1982, Yao [**67**] constructed a pseudorandom generator under the assumption that there are *one-way permutations*, i.e., length-preserving bijective one-way functions. In a subsequent series of papers this assumption was weakened, and finally, in 1989–1990, Impagliazzo, Levin, and Luby [**39**] and Håstad [**37**] obtained the following result.

**Theorem 1.** *Pseudorandom generators exist if and only if one-way functions exist.*

Pseudorandom generators are useful not only in cryptography, but also in complexity theory and in other areas of discrete mathematics. We are not going to discuss these applications in the present chapter. Instead, we illustrate their usefulness with the cryptosystem $Cr$ described at the beginning of this section, where for the algorithm $g$ we use a pseudorandom generator.

First of all, we must define the security of a cryptosystem with a secret key. Let $E_K$ be the encryption algorithm of a cryptosystem with a secret key. Denote the result by $d = E_K(m)$; here $K$ is a secret key consisting of $n$ bits, and $m$ is a plaintext consisting of $q(n)$ bits. Let $m_i$ denote the $i$th bit of the plaintext. Let $A$ be a



polynomial time probabilistic Turing machine fed with the ciphertext $d$ and producing the pair $(i,\sigma)$, where $i \in \{1,\dots,q(n)\}$, $\sigma \in \{0,1\}$. Informally, a cryptosystem is secure if there is no Turing machine $A$ that can compute some bit of the plaintext with probability essentially greater than that obtained by a simple guess.

**Definition 3.** A cryptosystem is *secure* if for any polynomial time probabilistic Turing machine $A$, any polynomial $p$, and all sufficiently large $n$ we have

$$\Pr\{A(d) = (i,\sigma)\,\&\,\sigma = m_i \mid K \in_R \{0,1\}^n, m \in_R \{0,1\}^{q(n)}\} < 1/2 + 1/p(n).$$

The probability (for brevity, we use the notation $Pr$ for it below) is a function of the random choice of the secret key $K$, the random choice of the plaintext $m$ in the set of all binary strings of length $q(n)$, and random variables used by the machine $A$.

Let us show that the cryptosystem $Cr$ using a pseudorandom generator as $g$ is secure in the sense of this definition. Assume the opposite, i.e., let there exist a polynomial time probabilistic algorithm $A$ and a polynomial $p$ such that $Pr \geq 1/2+1/p(n)$ for infinitely many values of $n$. Consider the algorithm $B$ which gets a binary string $r$ of length $q(n)$ as an input, chooses $m \in_R \{0,1\}^{q(n)}$, computes $d = m \oplus r$ and calls $A$ as a subroutine with $d$ as the input. After receiving a pair $(i,\sigma)$ from $A$ the algorithm $B$ checks whether indeed $m_i = \sigma$, produces 1 if the statement is true, 0 otherwise, and stops. It is easy to see that $B$ requires polynomial time (in $n$). Now we can verify that $B$ distinguishes between pseudorandom strings generated by $g$ and random strings of length $q(n)$. Indeed, if the input strings $r$ of $B$ are random, then the distribution of the cryptogram $d$ coincides with that in $Cr$, and therefore, by assumption, $Pr \geq 1/2+1/p(n)$ for infinitely many $n$. This contradicts the definition of a pseudorandom generator and proves that $Cr$ is secure.

The assumption that plaintexts are picked uniformly at random is rather unrealistic. But we used it only for the sake of simplicity. One can prove the security of the cryptosystem $Cr$ in more general settings as well. It is interesting to consider e.g. the following one. The adversary knows beforehand that one of the messages $m_1, m_2 \in$

$\{0,1\}^{q(n)}$ will be transmitted, each with probability 1/2. His task is to guess which one, after seeing the ciphertext $d$. We leave it to the reader as an exercise to prove the security of the cryptosystem $Cr$ in this setting (to this end, Definition 3 must be modified).

Of course, there are different ways to define the security of a cryptosystem with a secret key. For example, we may consider security against an attack with the choice of the public key: the adversary has an opportunity to choose in advance a polynomial number of plaintexts and get their encrypted versions; afterwards he gets the ciphertext and tries to compute at least one bit of the corresponding plaintext. It is easy to show that the cryptosystem $Cr$ with a pseudorandom generator used for $g$ is secure against the attack with the plaintext choice as well.

Thus we showed that pseudorandom generators allow us to construct secure cryptosystems. The main direction of research here is the construction of efficient pseudorandom generators based on various cryptographic assumptions. The efficiency is measured by the number of operations required to compute each subsequent bit of the pseudorandom sequence.

## 5. Zero-knowledge proofs

Suppose Alice knows a proof of some theorem, and she wishes to convince Bob that the theorem is true. Of course, Alice can simply give the proof to Bob for verification. But then Bob will be able to prove the theorem to other people himself, not referring to Alice. Is there a way for Alice to convince Bob that she knows the proof without any hint about what the proof actually is? The protocols with zero knowledge satisfy these two, apparently contradictory, requirements. These protocols were introduced by Goldwasser, Micali, and Rackoff [**33**] in 1989.

They considered the following model. Suppose Alice and Bob have probabilistic Turing machines $\mathbf{P}$ and $\mathbf{V}$, respectively, in their disposal. The computational resources of Alice are unlimited, while the machine $\mathbf{V}$ is polynomial time. The machines $\mathbf{P}$ and $\mathbf{V}$ can exchange messages through a common communication tape. After writing a message on the communication tape, the machine gets into



the waiting state, which it leaves after the answer is written on the tape. In addition, the machines $\mathbf{P}$ and $\mathbf{V}$ have a common input tape, where the input word $x$ is written. Alice proves the statement "$x \in L$", where $L$ is a fixed language known to both Alice and Bob. In order to avoid trivial cases, the language $L$ must be complicated, e.g., it may be NP-complete; otherwise Bob would be able to prove that $x \in L$ by himself. Essentially, the proof protocol consists of Bob's random questions, Alice's answers, and Bob's verification that the answers are correct. The protocol stops when the machine $\mathbf{V}$ stops; it produces 1 if the proof is accepted and 0 otherwise.

Let $A$ and $B$ be two probabilistic Turing machines interacting through a common communication tape. We denote by $[B(x), A(x)]$ the random output of the machine $A$ after the machines $A$ and $B$ work on an input word $x$, and by $|x|$ the length of the word $x$.

**Definition 4.** An *interactive proof for a language* $L$ is a pair $(\mathbf{P}, \mathbf{V})$ of interacting Turing machines such that the following two conditions are satisfied.

1 (Completeness). For each $x \in L$ we have

$$\Pr\{[\mathbf{P}(x), \mathbf{V}(x)] = 1\} = 1.$$

2 (Correctness). For each Turing machine $\mathbf{P}^*$, every polynomial $p$, and any word $x \notin L$ of sufficiently large length we have

$$\Pr\{[\mathbf{P}^*(x), \mathbf{V}(x)] = 1\} < 1/p(|x|).$$

The completeness property means that if the input word belongs to the language $L$ and both Alice and Bob follow the protocol, then the proof is always accepted. The correctness requirement protects Bob from possible Alice's dishonesty in case Alice tries to deceive Bob and to prove a false statement. In this case Alice may deviate from the protocol in any possible way, i.e., she can use an arbitrary Turing machine $\mathbf{P}^*$ instead of $\mathbf{P}$. The requirement guarantees that the probability of a successful trick is negligible.

**Definition 5.** An interactive proof protocol for a language $L$ is called a *zero-knowledge proof* if, in addition to conditions 1 and 2 in the previous definition, it satisfies one more condition:

3 (Zero-knowledge property). For any polynomial time probabilistic Turing machine $\mathbf{V}^*$ there exists a probabilistic Turing machine $\mathbf{M}_{\mathbf{V}^*}$ running in expected polynomial time in $|x|$ and such that for all $x \in L$ we have

$$\mathbf{M}_{\mathbf{V}^*}(x) = [\mathbf{P}(x), \mathbf{V}^*(x)].$$

The machine $\mathbf{M}_{\mathbf{V}^*}$ is called a simulating machine for $\mathbf{V}^*$. The average working time is expected to be bounded by a polynomial in $|x|$. This means that $\mathbf{M}_{\mathbf{V}^*}$ can work for a rather long time depending on the values of the random variables it uses. However, the probability that this time will exceed some polynomial bound is low. For each machine $\mathbf{V}^*$, its own simulating machine must be constructed; it can use the machine $\mathbf{V}^*$ as a subroutine. We denote by $\mathbf{M}_{\mathbf{V}^*}(x)$ the random output of the machine $\mathbf{M}_{\mathbf{V}^*}$ on the input word $x$.

The zero-knowledge property protects Alice from possible Bob's dishonesty in case Bob tries to get additional information by deviating from the protocol, that is, using another machine $\mathbf{V}^*$ instead of $\mathbf{V}$. Condition 3 means that in this way Bob can obtain only the information he would be able to compute himself in polynomial time without using the protocol.

Consider as an example the zero-knowledge protocol for the GRAPH ISOMORPHISM language suggested by Goldreich, Micali, and Wigderson [**35**]. An input word consists of a pair of graphs $G_1 = (U, E_1)$ and $G_0 = (U, E_0)$. Here $U$ is the set of graph vertices, which can be identified with the set $\{1, \ldots, n\}$, and $E_0$ and $E_1$ are the sets of edges such that $|E_1| = |E_0| = m$. Two graphs $G_1$ and $G_0$ are *isomorphic* if there exists a permutation $\varphi$ of the set $U$ such that $(u, v) \in E_0$ if and only if $(\varphi(u), \varphi(v)) \in E_1$ (in this case we write $G_1 = \varphi G_0$). The problem of graph isomorphism recognition is a well-known mathematical problem, and up to now no polynomial solution for it is known. On the other hand, it is not known whether this problem is NP-complete, although there is solid evidence that it is not.

**GI protocol.** Let $\varphi$ denote an isomorphism between $G_1$ and $G_0$. The cycle below is iterated $m$ times; the random values used in the iterations are independent.



1. The machine $\mathbf{P}$ chooses a random permutation $\pi$ on the set $U$, computes $H = \pi G_1$ and sends the graph $H$ to $\mathbf{V}$.

2. The machine $\mathbf{V}$ generates a random bit $\alpha$ and sends it to $\mathbf{P}$.

3. If $\alpha = 1$, then $\mathbf{P}$ sends the permutation $\pi$ to $\mathbf{V}$; otherwise it sends the permutation $\pi \circ \varphi$.

4. If the permutation received by $\mathbf{V}$ is not an isomorphism between $G_\alpha$ and $H$, then $\mathbf{V}$ rejects the proof and stops. Otherwise the cycle is repeated.

If the verification result of Round 4 was positive in all $m$ iterations, then the machine $\mathbf{V}$ accepts the proof.

Note that if the machine $\mathbf{P}$ gets the isomorphism $\varphi$ as an additional input word, then it does not require unlimited computational resources to implement the protocol. Moreover, in this case $\mathbf{P}$ can be a polynomial time probabilistic Turing machine.

**Theorem 2** (see [**35**]). *The protocol GI is a zero-knowledge proof for the GRAPH ISOMORPHISM language.*

The completeness of the GI protocol is obvious.

In order to prove that the protocol is correct, it suffices to note that the bit $\alpha$ generated by the machine $\mathbf{V}$ in Round 2 tells the machine $\mathbf{P}$ which of the two graphs, $G_0$ or $G_1$, should be proved to be isomorphic to $H$. If $G_0$ and $G_1$ are not isomorphic, then $H$ may be isomorphic to at most one of them. In this case, the verification of Round 4 gives a positive result with probability $\leq 1/2$, which leads to the probability $\leq 1/2^m$ after $m$ iterations.

The proof of the zero-knowledge property is much more complicated. Therefore, we present only the main idea. Note first that the main goal of the machine $\mathbf{V}^*$ consists in obtaining as much information about the isomorphism between $G_0$ and $G_1$ as possible. It is natural to assume that, in contrast to $\mathbf{V}$, its output will consist not of a single bit, but of all the information obtained as the result of the protocol implementation, including the contents of its random tape, the graph $H$ of Step 1, and the permutation of Step 3. The simulating machine $\mathbf{M}_{\mathbf{V}^*}$ must be able to produce the same random strings, graphs and permutations, and it is assumed that the permutation $\varphi$

is unknown to it! Therefore, $\mathbf{M}_{\mathbf{V}^*}$ tries to guess the bit $\alpha$ generated by the machine $\mathbf{V}^*$ in Round 2. In order to do this, $\mathbf{M}_{\mathbf{V}^*}$ generates a random bit $\beta$, a random permutation $\psi$, and computes $H = \psi G_\beta$. Then $\mathbf{M}_{\mathbf{V}^*}$ saves the state of the machine $\mathbf{V}^*$ (including the contents of the random tape) and calls $\mathbf{V}^*$ as a subroutine feeding it with the graph $H$. The machine $\mathbf{V}^*$ returns some bit $\alpha$. If $\alpha = \beta$, then this iteration of simulation is successful, since $\mathbf{M}_{\mathbf{V}^*}$ is able to demonstrate the required isomorphism. If $\alpha \neq \beta$, then $\mathbf{M}_{\mathbf{V}^*}$ restores the state of $\mathbf{V}^*$ saved before and repeats the attempt.

If we weaken the requirement that the random values $\mathbf{M}_{\mathbf{V}^*}(x)$ and $[\mathbf{P}(x), \mathbf{V}^*(x)]$ in the definition of zero knowledge are equal and require instead that their probability distributions be "almost equal", then we obtain another class of proofs, *statistical zero-knowledge proofs*.

One more class consists of *computational zero-knowledge proofs*. In this case we require that the model machine would produce a probability distribution that cannot be distinguished from $[\mathbf{P}(x), \mathbf{V}^*(x)]$ by a polynomial time probabilistic algorithm (the meaning of the term "distinguish" is similar to that in the definition of a pseudorandom generator).

We emphasize that in all three definitions of zero knowledge the conditions are posed only on the simulating machine behavior on words belonging to the language under consideration.

Zero-knowledge proofs are interesting not only as nontrivial mathematical objects. Their applications are also important. Identification protocols form the most natural and important kind of these applications (see Chapter 3). By such a protocol, Alice can prove her authentication to Bob.

Suppose, for example, that Alice is a smart bank card with implemented algorithm $\mathbf{P}$, and Bob is a bank computer executing the program $\mathbf{V}$. Before starting the execution of a bank operation the bank should be convinced that the bank card is authentic and identify its owner, or, in the language of cryptography, the card must pass an authentication procedure. In principle, the GI protocol above fits this purpose. In this case the bank computer memory contains a pair of graphs $(G_0, G_1)$ assigned to Alice, and the smart card contains the same pair of graphs and the isomorphism $\varphi$. It is assumed that



nobody except Alice (and, possibly, Bob) knows this isomorphism, whence the GI protocol allows the card to prove its authenticity. The completeness property implies that the card will always prove its authenticity. The correctness property protects the bank from an intruder that tries to pass the authentication procedure under the name of Alice after tampering with one or more authentication protocols of the card. Of course, in this case it makes no sense to prove that the pair of graphs $(G_0, G_1)$ belongs to the GRAPH ISOMORPHISM language, since it is chosen from the language. Instead, Alice proves that she knows an isomorphism $\varphi$. Interactive proofs of this kind are called *knowledge proofs*.

From the point of view of applications, an important property of the GI protocol, as well as other knowledge proof protocols, is that the algorithm $\mathbf{P}$ having $\varphi$ as an additional input is polynomial in time. Any other zero-knowledge protocol having the algorithm $\mathbf{P}$ possessing this property is also applicable. However, in practice the GI protocol and other protocols of this type are not efficient because of the large number of iterations, very long messages, and so on. Researchers' efforts concentrate on the search for efficient provably secure protocols.