Chapter 7

One-Way Functions

As mentioned several times so far, one-way functions and trapdoor functions play a fundamental role in modern cryptography. In this chapter, we elaborate on these functions. More specifically, we introduce the topic in Section 7.1, overview and discuss some candidate one-way functions in Section 7.2, elaborate on integer factorization algorithms and algorithms to compute discrete logarithms in Sections 7.3 and 7.4, address hard-core predicates in Section 7.5, briefly introduce the notion of elliptic curve cryptography in Section 7.6, and conclude with final remarks in Section 7.7.

7.1 INTRODUCTION

In Section 2.1.1 and Definition 2.1, we introduced the notion of a one-way function. More specifically, we said that a function $f : X \rightarrow Y$ is one way if $f(x)$ can be computed efficiently for all $x \in X$, but $f^{-1}(y)$ cannot be computed efficiently for $y \in_R Y$ (see Figure 2.1). We also noted that this definition is not precise in a mathematically strong sense and that we must first introduce some complexity-theoretic basics (mainly to define more precisely what we mean by saying that we can or we cannot compute efficiently). Because we have done this in Chapter 6, we are now ready to better understand and more precisely define the notion of a one-way function. This is done in Definition 7.1.

Definition 7.1 (One-way function) *A function $f : X \rightarrow Y$ is* one way *if the following two conditions are fulfilled:*

- *The function f is easy to compute, meaning that $f(x)$ can be computed efficiently for all $x \in X$. Alternatively speaking, there is a probabilistic polynomial-time (PPT) algorithm A that outputs $A(x) = f(x)$ for all $x \in X$.*

- *The function f is hard to invert, meaning that it is not known how to efficiently compute $f^{-1}(f(x))$ for all $x \in X$ or $f^{-1}(y)$ for $y \in_R Y$. Alternatively speaking, there is no known PPT algorithm A that outputs $A(f(x)) = f^{-1}(f(x))$ for all $x \in X$ or $A(y) = f^{-1}(y)$ for $y \in_R Y$.*

X and Y are often set to $\{0,1\}^*$. In either case, A is not required to find the correct x; it is only required to find some inverse of y (if the function f is injective, then the only inverse of y is x).

Another way to express the second condition in Definition 7.1 is to say that any PPT algorithm A attempting to invert the one-way function on an element in its range will succeed with no more than a negligible probability (i.e., smaller than any polynomial in the size of the input, where the probability is taken over the elements in the domain of the function and the internal coin tosses of A). The statement is probabilistic (i.e., A is not unable to invert the function, but it has a very low success probability). More formally,

$$\Pr[A(f(x), 1^n) \in f^{-1}(f(x))] \leq \frac{1}{p(n)}$$

for every PPT algorithm A, all $x \in X$, every polynomial p, and all sufficiently large n (with n representing the binary length of x). In this notation, the algorithm A is given $f(x)$ and the security parameter 1^n (expressed in unary representation). The only purpose of the second argument is to allow A to run in time polynomial in the length of x, even if $f(x)$ is much shorter than x. This rules out the possibility that a function is considered one way only because the inverting algorithm does not have enough time to print the output. Typically, f is a length-preserving function, and in this case the auxiliary argument 1^n is redundant.

Note that the notation given earlier is not standard, and that there are many notations used in the literature to express the same idea. For example, the following notion is also frequently used in the literature.

$$\Pr[(f(z) = y : x \overset{u}{\leftarrow} \{0,1\}^n; y \leftarrow f(x); z \leftarrow A(y, 1^n)] \leq \frac{1}{p(n)}$$

It basically says the same thing: if x is selected uniformly from $\{0,1\}^n$, y is assigned $f(x)$, and z is assigned $A(y, 1^n)$, then the probability that $f(z)$ equals y is negligible.

A few words concerning the notion of negligible probability are in place. We consider the success probability of a PPT algorithm A to be negligible if it is bound by a polynomial fraction. It follows that repeating A polynomially (in the input length) many times yields a new algorithm that also has a success probability that is negligible. Put in other words, events that occur with negligible probability remain negligible even if the experiment is repeated polynomially many times. This property is important for complexity-theoretic considerations.

In some literature, a distinction is made between strong one-way functions (as discussed earlier) and weak one-way functions, and it is then shown that the former can be constructed from the latter. The major difference is that whereas one only requires some nonnegligible fractions of the inputs on which it is hard to invert a weak one-way function, a strong one-way function must be hard to invert on all but a negligible fraction of the inputs. For the purpose of this book, we don't delve into the details of this distinction, and hence we don't distinguish between strong and weak one-way functions accordingly.

If X and Y are the same, then a one-way function $f : X \rightarrow X$ represents a *one-way permutation*. Hence, one-way permutations are just special cases of one-way functions, namely ones in which the domain and the range are the same.

Having in mind the notion of a one-way function, the notion of a *trapdoor function* (or *trapdoor one-way function*) is simple to explain and understand. According to Definition 2.2, a one-way function $f : X \rightarrow Y$ is a *trapdoor function* if there exist some extra information—called the *trapdoor*—with which f can be inverted efficiently—that is, there is a (deterministic or probabilistic) polynomial-time algorithm A that outputs $A(f(x)) = f^{-1}(f(x))$ for all $x \in X$ or $A(y) = f^{-1}(y)$ for $y \in_R Y$. Consequently, the notion of a trapdoor function can be defined by prepending the words "unless some extra information (i.e., the trapdoor) is known" in the second condition of Definition 7.1. More formally, a trapdoor function can be defined as suggested in Definition 7.2.

Definition 7.2 (Trapdoor function) *A one-way function $f : X \rightarrow Y$ is a* trapdoor function *if there is a trapdoor information t and a PPT algorithm I that can be used to efficiently compute $x' = I(f(x), t)$ with $f(x') = f(x)$.*

Many cryptographic functions required to be one way (or preimage resistant) output bitstrings of fixed size. For example, cryptographic hash functions are required to be one way and output strings of 128, 160, or more bits (see Chapter 8). Given such a function, one may be tempted to ask how expensive it is to invert it (i.e., one may ask for the computational complexity of inverting the hash function). Unfortunately, the (complexity-theoretic) answer to this question is not particularly useful. If the cryptographic hash function outputs n-bit values, then 2^n tries are always sufficient to invert the function or to find a preimage for a given hash value (2^{n-1} tries are sufficient on the average). Because 2^n is constant for any fixed $n \in \mathbb{N}$, the computational complexity to invert the hash function is $O(1)$, and hence one cannot say that inverting it is intractable. If we want to use complexity-theoretic arguments, then we cannot live with a constant n. Instead, we must make n variable, and it must be possible to let n grow arbitrarily large. Consquently, we must work with potentially infinite *families*[1] *of one-way functions* (i.e., at least one for each n). The notion of a family of one-way functions is formally captured in Definition 7.3.

Definition 7.3 (Family of one-way function) *A family of functions* $F = \{f_i : X_i \rightarrow Y_i\}_{i \in I}$ *is a* family of one-way functions *if the following three conditions are fulfilled:*

- *I is an infinite index set;*
- *Every $i \in I$ selects a function $f_i : X_i \rightarrow Y_i$ from the family;*
- *Every $f_i : X_i \rightarrow Y_i$ is a one-way function according to Definition 7.1.*

A family of one-way functions $\{f_i : X_i \rightarrow Y_i\}_{i \in I}$ is a *family of one-way permutations* if every f_i is a permutation over the domain X_i (i.e., $Y_i = X_i$). Furthermore, it is a *family of trapdoor functions* if every f_i is a trapdoor function with trapdoor information t_i.

In this book, we often talk about one-way functions and trapdoor functions when we should actually be talking about families of such functions. We make this simplification because we think that it is more appropriate and simpler to understand. In either case, we want to emphasize that there is no function—or family of functions—known to be one way (in a mathematically strong sense) and that the current state of knowledge in complexity theory does not allow us to prove the existence of one-way functions, even using more traditional assumptions as $\mathcal{P} \neq \mathcal{NP}$. Hence, only a few functions are conjectured to be one way. These candidate one-way functions are overviewed and discussed next.

7.2 CANDIDATE ONE-WAY FUNCTIONS

There are a couple of functions that are conjectured to be one way. For example, a symmetric encryption system that encrypts a fixed plaintext message yields such a function.[2] For example, the use of DES in this construction can be shown to be one way assuming that DES is a family of pseudorandom functions. Another simple example is the integer multiplication function $f : (x, y) \mapsto xy$ for $x, y \in \mathbb{Z}$. As discussed later in this chapter, no efficient algorithm is known to find the prime factors of a large integer.

1 In some literature, the terms "classes," "collections," or "ensembles" are used instead of "families."

2 For example, UNIX systems use such a function to store the user passwords in protected form.

The following three functions, which are conjectured to be one way, have many applications in (public key) cryptography:

- Discrete exponentiation function;
- RSA function;
- Modular square function.

The fact that these functions are conjectured to be one way means that we don't know how to efficiently invert them. The best algorithms we have at hand are super-polynomial—that is, they have an exponential or subexponential running time behavior (some of the algorithms are briefly overviewed in Sections 7.3 and 7.4). The three candidate one-way functions are addressed next.

7.2.1 Discrete Exponentiation Function

From the real numbers, we know that the exponentiation and logarithm functions are inverse to each other and that they can both be computed efficiently. This makes us believe that this must be the case in all algebraic structures. There are, however, algebraic structures in which we can compute the exponentiation function efficiently, but in which no known algorithm can be used to efficiently compute the logarithm function. For example, let p be a prime number and g be a generator (or primitive root) of $\mathbb{Z}_p^*$. The function

$$\begin{array}{rcl} \mathrm{Exp}_{p,g} : \mathbb{Z}_{p-1} & \longrightarrow & \mathbb{Z}_p^* \\ x & \longmapsto & g^x \end{array}$$

is then called *discrete exponentiation function* to the base g. It defines an isomorphism from the additive group $\langle \mathbb{Z}_{p-1}, + \rangle$ to the multiplicative group $\langle \mathbb{Z}_p^*, \cdot \rangle$—that is, $\mathrm{Exp}_{p,g}(x+y) = \mathrm{Exp}_{p,g}(x) \cdot \mathrm{Exp}_{p,g}(y)$. Because $\mathrm{Exp}_{p,g}$ is bijective, it has an inverse function that is defined as follows:

$$\begin{array}{rcl} \mathrm{Log}_{p,g} : \mathbb{Z}_p^* & \longrightarrow & \mathbb{Z}_{p-1} \\ x & \longmapsto & \log_g x \end{array}$$

It is called the *discrete logarithm function*. For any $x \in \mathbb{Z}_p^*$, the discrete logarithm function computes the *discrete logarithm* of x to the base g, denoted by

$\log_g x$. This value refers to the element of $\mathbb{Z}_{p-1}$ to which g must be set to the power of in order to get x.

$\text{Exp}_{p,g}$ is efficiently computable, for example, by using the square-and-multiply algorithm (i.e., Algorithm 3.3). Contrary to that (and contrary to the logarithm function in the real numbers), no efficient algorithm is known to exist for computing discrete logarithms for sufficiently large prime numbers p. All known algorithms have a super-polynomial running time, and it is widely believed that $\text{Log}_{p,g}$ is not efficiently computable.

Earlier in this chapter we said that—in order to use complexity-theoretic arguments—we must consider families of one-way functions. In the case of the discrete exponentiation function, we may use p and g as indexes for an index set I. In fact, I can be defined as follows:

$$I := \{(p,g) \mid p \in \mathbf{P};\ g \text{ a generator of } \mathbb{Z}_p^*\}$$

Using this index set, we can formally define the Exp family (i.e., the family of discrete exponentiation functions)

$$\text{Exp} := \{\text{Exp}_{p,g} : \mathbb{Z}_{p-1} \longrightarrow \mathbb{Z}_p^*,\ x \longmapsto g^x\}_{(p,g)\in I}$$

and the Log family (i.e., the family of discrete logarithm functions)

$$\text{Log} := \{\text{Log}_{p,g} : \mathbb{Z}_p^* \longrightarrow \mathbb{Z}_{p-1},\ x \longmapsto \log_g x\}_{(p,g)\in I}.$$

If we want to employ the Exp family as a family of one-way functions, then we must make sure that it is hard to invert, meaning that it is not known how to efficiently compute discrete logarithms. This is where the *discrete logarithm assumption* (DLA) as formally expressed in Definition 7.4 comes into play.

Definition 7.4 (Discrete logarithm assumption) *Let $I_k := \{(p,g) \in I \mid |p| = k\}$ for $k \in \mathbb{N}$,[3] $p(k)$ be a positive polynomial, and $A(p,g,y)$ be a PPT algorithm. Then the DLA says that there exists a $k_0 \in \mathbb{N}$, such that*

3 This means that the index set I consists of disjoint subsets I_k (i.e., $I = \bigcup_{k\in\mathbb{N}} I_k$). Consequently, k may be considered the security parameter of $i = (p,g) \in I_k$.

$$\Pr[A(p,g,y) = \text{Log}_{p,g}(y) : (p,g) \overset{u}{\leftarrow} I_k; y \overset{u}{\leftarrow} \mathbb{Z}_p^*] \leq \frac{1}{p(k)}$$

for all $k \geq k_0$.

In this terminology, the PPT algorithm A models an adversary who tries to compute the discrete logarithm of y to the base g, or, equivalently, to invert the discrete exponentiation function $\text{Exp}_{p,g}$. Furthermore, the term

$$y \overset{u}{\leftarrow} \mathbb{Z}_p^*$$

suggests that y is uniformly distributed, meaning that all $y \in \mathbb{Z}_p^*$ occur with the same probability (i.e., $\Pr[y] = 1/|\mathbb{Z}_p^*| = 1/(p-1)$). This is just another way of saying that $y \in_R \mathbb{Z}_p^*$. Similarly, the term

$$(p,g) \overset{u}{\leftarrow} I_k$$

suggests that the pair (p,g) is uniformly distributed, meaning that all $(p,g) \in I_k$ occur with the same probability. Consequently, the probability statement of Definition 7.4 can be read as follows: if we randomly select both the index $i = (p,g) \in I_k$ with security parameter k and $y = g^x$, then the probability that the PPT algorithm A successfully computes and outputs $\text{Log}_{p,g}(y)$ is negligible (i.e., smaller than any polynomial bound). This means that $\text{Exp}_{p,g}$ cannot be inverted by A for all but a negligible fraction of input values.

Even if the security parameter k is very large, there may be pairs (p,g) such that A can correctly compute $\text{Log}_{p,g}(y)$ with a probability that is nonnegligible. For example, if $p-1$ has only small prime factors, then there is an efficient algorithm due to Steve Pohlig and Martin E. Hellman that can be used to compute the discrete logarithm function [1]. In either case, the number of such special pairs (p,g) is negligibly small as compared to all keys with security parameter k. If (p,g) is randomly (and uniformly) chosen from I_k, then the probability of obtaining such a pair (i.e., a pair for which A can compute discrete logarithms) is negligibly small.

Under the DLA, the Exp family represents a family of one-way functions. It is used in many public key cryptosystems, including, for example, the ElGamal public key cryptosystem (see Sections 14.2.3 and 15.2.2) and the Diffie-Hellman key exchange protocol (see Section 16.3). Furthermore, several problems are centered around the DLA and the conjectured one-way property of the discrete exponential function. The most important problems are the *discrete logarithm problem* (DLP) captured in Definition 7.5, the (computational) *Diffie-Hellman problem* (DHP) captured in Definition 7.6, and the *Diffie-Hellman decision problem* (DHDP) captured in Definition 7.7. The problems can be specified in arbitrary cyclic groups.

Definition 7.5 (Discrete logarithm problem) *Let G be a cyclic group, g be a generator of G, and h be an arbitrary element in G. The DLP is to determine an integer x such that $g^x = h$.*

Definition 7.6 (Diffie-Hellman problem) *Let G be a cyclic group, g be a generator in G, and x and y be two integers smaller than the order of G (i.e., $x, y < |G|$). The terms g^x and g^y then represent two elements in G. The DHP is to determine g^{xy} from g^x and g^y.*

Definition 7.7 (Diffie-Hellman decision problem) *Let G be a cyclic group, g be a generator of G, and r, s, and t be three positive integers smaller than the order of G (i.e., $r, s, t < |G|$). The terms g^r, g^s, g^t, and g^{rs} then represent elements in G. The DHP is to determine whether g^{rs} or g^t solves the DHP for g^r and g^s. Alternatively speaking, the DHDP is to distinguish the triples $\langle g^r, g^s, g^{rs} \rangle$ and $\langle g^r, g^s, g^t \rangle$ when they are given in random order.*

When giving all of these problems, it may be interesting to know how they are related. This is done by giving complexity-theoretic reductions from one problem to another (see Definition 6.10 for the notion of a polynomial-time reduction). In fact, in can be shown that DHP $\leq_P$ DLP (i.e., the DHP polytime reduces to the DLP) and that DDHP $\leq_P$ DHP (i.e., the DDHP polytime reduces to the DHP) in an arbitrary finite Abelian group. So the DLP is the hardest among the problems (i.e., if one is able to solve the DLP, then one is trivially able to solve the DHP and the DDHP). The exact relationship and the complexity of the corresponding proof (if it is known in the first place) depend on the actual group in use. In many cyclic groups, the DLP and the DHP have been shown to be computationally equivalent [2, 3]. There are, however, groups in which one can solve the DDHP in polynomial time, but the fastest known algorithm to solve the DHP requires subexponential time. In order to better understand the DLP and the underlying DLA, it is worthwhile to have a look at the currently available algorithms to compute discrete logarithms. This is done in Section 7.4.

7.2.2 RSA Function

Let n be the product of two distinct primes p and q (i.e., $n = pq$), and e be relatively prime to $\phi(n)$. Then the function

$$\begin{array}{rrcl} \mathrm{RSA}_{n,e} : & \mathbb{Z}_n^* & \longrightarrow & \mathbb{Z}_n^* \\ & x & \longmapsto & x^e \end{array}$$

is called the *RSA function*. It computes the e^{th} power for $x \in \mathbb{Z}_n^*$. To compute the inverse function, it is required to compute e^{th} roots. If the inverse d of e modulo $\phi(n)$ is known, then the following RSA function can be used to compute the inverse of $\mathrm{RSA}_{n,e}$:

$$\begin{array}{rrcl} \mathrm{RSA}_{n,d} : & \mathbb{Z}_n^* & \longrightarrow & \mathbb{Z}_n^* \\ & x & \longmapsto & x^d \end{array}$$

$\mathrm{RSA}_{n,e}$ can be efficiently computed by modular exponentiation. In order to compute $\mathrm{RSA}_{n,d}$, however, one must know either d or the prime factors of n (i.e., p and q). As of this writing, no polynomial-time algorithm to compute $\mathrm{RSA}_{n,d}$ is known if p, q, or d are not known. Hence, $\mathrm{RSA}_{n,d}$ can only be computed if any of these values is known, and hence these values represent trapdoors for $\mathrm{RSA}_{n,e}$.

If we want to turn the RSA function into a family of one-way functions, then we must define an index set I. This can be done as follows:

$$I := \{(n,e) \mid n = pq; p,q \in \mathbf{P}; p \neq q; 0 < e < \phi(n); (e,\phi(n)) = 1\}$$

Using this index set, the family of RSA functions can be defined as follows:

$$\mathrm{RSA} := \{\mathrm{RSA}_{n,e} : \mathbb{Z}_n^* \longrightarrow \mathbb{Z}_n^*,\ x \longmapsto x^e\}_{(n,e)\in I}$$

This family of RSA functions is called the *RSA family*. Because each RSA function $\mathrm{RSA}_{n,e}$ represents a permutation over $\mathbb{Z}_n^*$, the RSA family represents a family of one-way (or trapdoor) permutations.

It is assumed and widely believed that the RSA family is a family of trapdoor permutations, meaning that $\mathrm{RSA}_{n,e}$ is hard to invert (for sufficiently large and properly chosen n). The *RSA assumption* formally expressed in Definition 7.8 makes the one-way property of the RSA family explicit.

Definition 7.8 (RSA assumption) *Let $I_k := \{(n,e) \in I \mid n = pq; |p| = |q| = k\}$ for $k \in \mathbb{N}$, $p(k) \in \mathbb{Z}[\mathbb{N}]$ be a positive polynomial, and $A(p,g,y)$ be a PPT algorithm. Then the RSA assumption says that there exists a $k_0 \in \mathbb{N}$, such that*

$$\Pr[A(n,e,y) = \text{RSA}_{n,d}(y) : (n,e) \overset{u}{\leftarrow} I_k; y \overset{u}{\leftarrow} \mathbb{Z}_n^*] \leq \frac{1}{p(k)}$$

for all $k \geq k_0$.

Again, the PPT algorithm A models the adversary who tries to compute $\text{RSA}_{n,d}(y)$ without knowing the trapdoor information. The RSA assumption may be interpreted in an analogous way to the DLA. The fraction of keys (n,e) in I_k, for which A has a significant chance to succeed, must be negligibly small if the security parameter k is sufficiently large.

There is also a stronger version of the RSA assumption known as the *strong RSA assumption*. The strong RSA assumption differs from the RSA assumption in that the adversary can select the public exponent e: given a modulus n and a ciphertext c, the adversary must compute any plaintext m and public exponent e such that $c = m^e \pmod{n}$. For the purpose of this book, we don't use the strong RSA assumption anymore.

If we accept the RSA assumption, then the *RSA problem* (RSAP) captured in Definition 7.9 is intractable.

Definition 7.9 (RSA problem) *Let (n,e) be a public key and $c = m^e \pmod{n}$. The RSAP is to determine m (i.e., the e^{th} root of c modulo n) if the private key (n,d) and the factorization of n (i.e., p and q) are not known.*

The RSA assumption and the RSAP are at the core of many public key cryptosystems, including, for example, the RSA public key cryptosystem (see Sections 14.2.1 and 15.2.1). Because the prime factors of n (i.e., p and q) represent a(nother) trapdoor for $\text{RSA}_{n,d}$ (in addition to d), somebody who is able to factor n is also able to compute $\text{RSA}_{n,d}$ and to invert $\text{RSA}_{n,e}$ accordingly. Consequently, one must make the additional assumption that it is computationally infeasible (for the adversary one has in mind) to factor n. This is where the *integer factoring assumption* (IFA) as formally expressed in Definition 7.10 comes into play.

Definition 7.10 (Integer factoring assumption) *Let $I_k := \{n \in I \mid n = pq; |p| = |q| = k\}$ for $k \in \mathbb{N}$, $p(k)$ be a positive polynomial, and $A(n)$ be a PPT algorithm. Then the IFA says that there exists a $k_0 \in \mathbb{N}$, such that*

$$\Pr[A(n) = p : n \overset{u}{\leftarrow} I_k] \leq \frac{1}{p(k)}$$

for all $k \geq k_0$.

If we accept the IFA, then the *integer factoring problem* (IFP) captured in Definition 7.11 is intractable.

Definition 7.11 (Integer factoring problem) *Let n be a positive integer (i.e., $n \in \mathbb{N}$). The IFP is to determine the prime factors of n (i.e., to determine $p_1, \ldots, p_k \in \mathbf{P}$ and $e_1, \ldots, e_k \in \mathbb{N}$) such that*

$$n = p_1^{e_1} \cdots p_k^{e_k}.$$

The IFP is well defined, because every positive integer can be factored uniquely up to a permutation of its prime factors (see Theorem 3.5). Note that the IFP must not always be intractable, but that it must be possible to easily find an instance of the IFP that is intractable.

Again using complexity-theoretic arguments, one can show that RSAP $\leq_P$ IFP (i.e., the RSAP polytime reduces to the IFP). This means that one can invert the RSA function if one can solve the IFP. The converse, however, is not known to be true (i.e., it is not known whether there exists a simpler way to invert the RSA function than to solve the IFP). In order to better understand the RSAP and the underlying RSA assumption, it is worthwhile to have a look at the currently available integer factorization algorithms. This is done in Section 7.3.

7.2.3 Modular Square Function

Similar to the exponentiation function, the square function can be computed and inverted efficiently in the real numbers, but it is not known how to invert it efficiently in a cyclic group. If, for example, we consider $\mathbb{Z}_n^*$, then modular squares can be computed efficiently, but modular square roots can only be computed efficiently if the prime factorization of n is known. In fact, it can be shown that computing square roots in $\mathbb{Z}_n^*$ and factoring n are computationally equivalent. Consequently, the modular square function looks like a candidate one-way function. Unfortunately, the modular square function (in its general form) is neither injective nor surjective. It can, however, be made injective and surjective (and hence bijective) if the domain and range are both restricted to QR_n (i.e., the set of quadratic residues or squares modulo n), with n being a Blum integer (see Definition 3.33). The function

$$\begin{array}{rcccl} \text{Square}_n & : & QR_n & \longrightarrow & QR_n \\ & & x & \longmapsto & x^2 \end{array}$$

is then called *square function*. It is bijective, and hence the inverse function

$$\begin{array}{rrcl} \text{Sqrt}_n : & QR_n & \longrightarrow & QR_n \\ & x & \longmapsto & x^{1/2} \end{array}$$

exists and is called the *square root function*. Either function maps an element of QR_n to another element of QR_n. The function represents a permutation.

To turn the square function into a one-way function (or one-way permutation, respectively), we must have an index set I. Taking into account that n must be a Blum integer, the index set can be defined as follows:

$$I := \{n \mid n = pq; p, q \in \mathbf{P}; p \neq q; |p| = |q|; p, q \equiv 3 \pmod 4\}$$

Using this index set, we can define the following family of square functions:

$$\text{Square} := \{\text{Square}_n : QR_n \longrightarrow QR_n,\ x \longmapsto x^2\}_{n \in I}$$

This family is called the *Square family*, and the family of inverse functions can be defined as follows:

$$\text{Sqrt} := \{\text{Sqrt}_n : QR_n \longrightarrow QR_n,\ x \longmapsto x^{1/2}\}_{n \in I}$$

It is called the *Sqrt family*. The Square family of trapdoor permutations is used by several public key cryptosystems, including, for example, the Rabin public key cryptosystem (see Section 14.2.2). For every $n \in I$, the prime factors p and q represent a trapdoor. Hence, if we can solve the IFP, then we can trivially invert the Square family. We look at algorithms to solve the IFP next.

7.3 INTEGER FACTORIZATION ALGORITHMS

Many algorithms can be used to solve the IFP.[4] They can be divided into two broad categories:

4 http://mathworld.wolfram.com/PrimeFactorizationAlgorithms.html

- *Special-purpose algorithms* depend on and take advantage of special properties of the integer n to be factored, such as its size, the size of its smallest prime factor p, or the prime factorization of $p-1$.
- Contrary to that, *general-purpose algorithms* depend on nothing (i.e., they work for all values of n).

In practice, algorithms of both categories are combined and used one after another. If one is given a large integer n with no clue about the size of its prime factors, then one usually employs special-purpose algorithms that are optimized to find small prime factors before one turns to the less efficient general-purpose algorithms.

7.3.1 Special-Purpose Algorithms

Examples of special-purpose algorithms include trial division, P±1, Pollard Rho, and the elliptic curve method (ECM).

7.3.1.1 Trial Division

If n is composite, then at least one prime factor is at most $\sqrt{n}$. Consequently, one can always factorize n by trying to divide it by all primes up to $\lfloor\sqrt{n}\rfloor$. This simple algorithm is called *trial division*. Its running time is $O(p)$, where p is the smallest prime factor of n (i.e., the one that is found first). In the worst case, this equals to

$$O(\sqrt{n}) = O(e^{\ln\sqrt{n}}) = O(e^{\ln(n^{1/2})})$$

if the smallest prime factor of n is about $\sqrt{n}$ (this occurs if n has two prime factors of about the same size). Consequently, the worst-case running time function of the trial division integer factorization algorithm is exponential in $\ln n$. If, for example, n is 1,024 bits long, then the algorithm requires

$$\sqrt{2^{1024}} = (2^{1024})^{1/2} = 2^{1024/2} = 2^{512}$$

trial divisions in the worst case. This is certainly beyond what is feasible today, and hence the trial division algorithm can only be used to factorize n if n is sufficiently small (e.g., smaller than 10^{12}) or smooth (i.e., it has only small prime factors). In either case, the space requirements of the trial division algorithm are negligible.

7.3.1.2 P±1

In the 1970s, John M. Pollard developed and proposed two special-purpose integer factorization algorithms that are optimized to find small prime factors. The first algorithm is known as P$-$1.

Let n be the integer to be factorized and p be some (yet unknown) prime factor of n, for which $p-1$ is B-smooth—that is, $p-1$ is the product of possibly many prime numbers that are smaller than or equal to B (see Definition 3.28). If k is the product of all prime numbers that are smaller than or equal to B, then k is a multiple of $p-1$. Now consider what happens if we take a small integer (e.g. $a=2$) and set it to the power of k. Fermat's Little Theorem (i.e., Theorem 3.7) tells us that

$$a^k \equiv 1 \pmod{p},$$

and hence p divides a^k-1. On the other hand, p must also divide n (remember that p is supposed to be a prime factor of n), and hence p divides the greatest common divisor of a^k-1 and n (i.e., $p \mid gcd(a^k-1,n)$). Note that k might be very large, but a^k-1 can always be reduced modulo n.

Note that one knows neither the prime factorization of $p-1$ nor the bound before one starts the algorithm. So one has to begin with an initially chosen bound B and perhaps increase it during the execution of the algorithm. Consequently, the algorithm is practical only if B is not too large. For the typical size of prime numbers in use today (e.g., for the RSA public key cryptosystem), the probability that one can factorize n using Pollard's P$-$1 algorithm is pretty small. Nevertheless, the mere existence of the algorithm is a reason that some cryptographic standards require RSA moduli to have prime factors p for which $p-1$ has at least one large prime factor. In the literature, such primes are frequently called *strong*.

In either case, the running time of Pollard's P$-$1 algorithm is $O(|t|)$, where t is the largest prime power dividing $p-1$. Pollard's P$-$1 algorithm was later modified and a corresponding P$+$1 algorithm was proposed.

7.3.1.3 Pollard Rho

The second algorithm developed and proposed by Pollard in the 1970s is known as *Pollard Rho*. The basic idea is to have the algorithm successively draw random numbers less than n. If p is a (yet unknown) prime factor of n, then it follows from the birthday paradox (see Section 8.1) that after about $p^{1/2} = \sqrt{p}$ rounds one has drawn x_i and x_j with $x_i \neq x_j$ and $x_i \equiv x_j \pmod{p}$. If this occurs, one knows that p divides the greatest common divisor of $x_i - x_j$ and n (i.e., $p \mid gcd(x_i - x_j, n)$).

The Pollard Rho algorithm has a running time of

$$O(\sqrt{p})$$

where p is the smallest prime factor of n, or

$$O(\sqrt[4]{n}) = O(n^{1/4}) = O(e^{\ln(n^{1/4})})$$

in the worst case. Consequently, the Pollard Rho algorithm is an algorithm that is exponential in $\ln n$, and as such it can only be used if p is small compared to n. For the size of the integers that are used today, the algorithm is still impractical. It was, however, used on the factorization of the eighth Fermat number

$$F_8 = 2^{2^8} + 1 = 2^{256} + 1,$$

which unexpectedly turned out to have a small prime factor. In either case, the space requirements of the Pollard Rho algorithm are small.

7.3.1.4 ECM

In the 1980s, Hendrik W. Lenstra developed and proposed the ECM [4]. It can be best thought of as a generalization or randomized version of Pollard's P-1 algorithm. The success of Pollard's P-1 algorithm depends on n having a divisor p such that $p - 1$ is smooth. If no such p exists, then the algorithm fails. The ECM randomizes the choice, replacing the group $\mathbb{Z}_p$ used in Pollard's P-1 algorithm by a random (or pseudorandom) elliptic curve over $GF(p)$.

The ECM has a subexponential running time. Its average-case (worst-case) running time is $L_p[\frac{1}{2}, \sqrt{2}]$ ($L_n[\frac{1}{2}, 1]$). The worst case occurs when p is roughly $\sqrt{n}$, which is often the case when one uses RSA or some other public key cryptosystem. So, although the ECM cannot be considered a threat against the standard RSA public key cryptosystem that uses two primes, it must nevertheless be taken into account when one implements the so-called multiprime RSA system, where the modulus may have more than two prime factors.

7.3.2 General-Purpose Algorithms

Examples of general-purpose integer factorization include continued fraction, the quadratic sieve (QS), and the number field sieve (NFS).

7.3.2.1 Continued Fraction

The continued fraction algorithm was developed and proposed in the 1970s [5]. It has a subexponential running time and was the the fastest integer factoring algorithm in use for quite a long time (i.e., until the QS was developed).

7.3.2.2 QS

In the 1980s, Carl Pomerance developed and proposed the QS [6]. Like many other general-purpose integer factorization algorithms, the QS is based on an idea that is due to Fermat. If we have two integers x and y with

$$x \not\equiv \pm y \pmod{n}$$

and

$$x^2 \equiv y^2 \pmod{n}, \tag{7.1}$$

then we can factorize n with a success probability of $1/2$. Let $n = pq$, and we want to use x and y to find p or q. First, we note that $x^2 \equiv y^2 \pmod{n}$ means that

$$x^2 - y^2 = (x-y)(x+y) = 0 \pmod{n}.$$

Because $n = pq$, the four following cases are possible:

1. $p|x-y$ and $q|x+y$;
2. $p|x+y$ and $q|x-y$;
3. $p|x-y$ and $q|x-y$ (but neither p nor q divides $x+y$);
4. $p|x+y$ and $q|x+y$ (but neither p nor q divides $x-y$).

All of these cases are equally probable and occur with a probability of $1/4$. If we then compute

$$d = gcd(x-y, n),$$

then d refers to p in case 1, q in case 2, n in case 3, and 1 in case 4. Hence, in cases 1 and 2 we have indeed found a prime factor of n. So the success probability is in fact $1/2$ (as mentioned earlier).

So the question (most general-purpose integer factorization algorithms try to answer) is how to find two integers x and y that satisfy equivalence (7.1).

The general approach is to choose a set of t relatively small primes $S = \{p_1, p_2, \ldots, p_t\}$ (the so-called factor base) and to proceed with the following two steps:

- First, one computes $b_i \equiv a_i^2 \pmod{n}$ for arbitrary a_i, and this value is expressed as the product of powers of the primes in S. In this case, b_i can be represented as a vector in a t-dimensional vector space. This step is called the *relation collection stage*, and it is highly parallelizable.
- Second, if we have collected enough (e.g., $t+1$) values for b_i, then a solution of equivalence (7.1) can be found by performing the Gaussian elimination on the matrix $B = [b_i]$. This step is called the *matrix step* and cannot be parallelized. It works on a huge (sparse) matrix and eventually comes up with a nontrivial factor of n.

Obviously, the choice of the number of primes of S is very important for the performance of the QS. If it is too small, then the relation collection stage may take very long (because a very small proportion of numbers factor over a small set of primes). If, however, it is too large (and too many primes are put into S), then the matrix may become too large to be reduced efficiently.

In either case, the QS has a subexponential running time of $L_n[\frac{1}{2}, c]$ for some constant c. As mentioned later, a variation of the QS was used in 1994 when RSA-129 was successfully factorized.

7.3.2.3 NFS

The NFS was developed and proposed in the 1990s (e.g., [7]). It is conceptually similar to the QS and is currently the fastest general-purpose integer factorization algorithm. It has a running time of $L_n[\frac{1}{3}, c]$ for $c = 1.923$ and was used in 1999 to factorize RSA-155 (see the following section). Furthermore, there are several variations of it, including, for example, the special number field sieve (SNFS) and the general number field sieve (GNFS).

7.3.3 State of the Art

When the RSA public key cryptosystem was published, a famous challenge was posted in the August 1977 issue of *Scientific American* [8]. In fact, US$100 were offered to anyone who could decrypt a message that was encrypted using a 129-digit integer acting as modulus. The number became known as RSA-129, and it was not factored until 1994 (with a distributed implementation of a variation of the QS algorithm [9]). Just to give an impression of the size of such an integer, RSA-129 and its prime factors are reprinted here:

$$\begin{aligned} RSA-129 &= 1143816257578888676692357799761466120102182967212 \\ &\quad 4236256256184293570693524573389783059712356395870 \\ &\quad 5058989075147599290026879543541 \\ &= 3490529510847650949147849619903898133417764638493 \\ &\quad 387843990820577 \\ &\quad * \\ &\quad 3276913299326670954996198819083446141317764296799 \\ &\quad 2942539798288533 \end{aligned}$$

Today, the RSA Factoring Challenge is sponsored by RSA Laboratories to learn more about the actual difficulty of factoring large integers of the type used in the RSA public key cryptosystem.[5] In 1999, a group of researchers completed the factorization of the 155-digit (512-bit) RSA Challenge Number, and in December 2003, researchers at the University of Bonn (Germany) completed the factorization of the 174-digit (576-bit) RSA Challenge Number. The next numbers to factor (in the RSA Factoring Challenge) are 640, 704, 768, 896, 1,024, 1,536, and 2,048 bits long.

In the past, a couple of proposals have been made to use specific hardware devices to speed up integer factoring algorithms. For example, TWINKLE is a device that can be used to speed up the first step in the QS algorithm—that is, find pairs (x, y) of distinct integers that satisfy equivalence (7.1) [10]. TWIRL is a more recent proposal [11].

7.4 ALGORITHMS FOR COMPUTING DISCRETE LOGARITHMS

There are basically two categories of algorithms to solve the DLP (and to compute discrete logarithms accordingly):

- Algorithms that attempt to exploit special characteristics of the group in which the DLP must be solved;

5 http://www.rsasecurity.com/rsalabs/challenges/factoring/

- Algorithms that do not attempt to exploit special characteristics of the group in which the DLP must be solved.

Algorithms of the first category are often called *special-purpose algorithms*, whereas algorithms of the second category are called *generic algorithms*. Let's start with the second category of algorithms.

7.4.1 Generic Algorithms

Let G be a cyclic group and g be a generator in this group. The difficulty of computing discrete logarithms to the base g in G then depends on whether we know the order of the group (i.e., $|G|$). If we don't know $|G|$, then the *Baby-step giant-step algorithm* is the best we can do. It has a running time of $O(\sqrt{|G|} \log |G|)$ and memory requirements of $O(\sqrt{|G|})$. If, however, we know $|G|$, then we can do better. In this case, we can use *Pollard's ρ-algorithm*, which is slightly more efficient than the Baby-step giant-step algorithm. In fact, Pollard's ρ-algorithm has a running-time complexity of $O(\sqrt{|G|})$ and requires almost no memory. It has been shown that this running time is a lower bound for any general-purpose algorithm to compute discrete logarithms in a cyclic group (if the factorization of the group order is not known) [12].

If, in addition to $|G|$, we also know the factorization of $|G|$, then we can use the *Pohlig-Hellman algorithm*, which has a running time of $O(\sqrt{q} \log q)$ (where q is the largest prime factor of $|G|$). This result implies, for example, that in DLP-based cryptosystems for $\mathbb{Z}_p^*$, $p-1$ must have at least one large prime factor (as already mentioned in Section 7.2.1).

7.4.2 Special-Purpose Algorithms

If we are talking about special-purpose algorithms, then we are talking about specific groups. If, for example, we are talking about $\mathbb{Z}_p^*$, then there are basically two algorithms to solve the DLP in a subexponential running time.

- The *index calculus algorithm* has a running time of $L_p[\frac{1}{2}, c]$ for some small constant c;
- The NFS algorithm can also be used to compute discrete logarithms. Remember that it has a running time of $L_p[\frac{1}{3}, 1.923]$.

Consequently, the NFS algorithm is the algorithm of choice to solve the DLP in $\mathbb{Z}_p^*$.

7.4.3 State of the Art

Given that the NFS algorithm can be used to factor integers and compute discrete logarithms in $\mathbb{Z}_p^*$, we note that the state of the art in computing discrete logarithms in $\mathbb{Z}_p^*$ is comparable to the state of the art in factoring integers. This suggests that we must also work with 1,024-bit prime numbers p. Special care must be taken that $p-1$ does not have only small prime factors. Otherwise, the Pohlig-Hellman algorithm [1] can be used to efficiently compute discrete logarithms.

If we are not working in $\mathbb{Z}_p^*$, then the special-purpose algorithms mentioned earlier do not work, and the state of the art in computing discrete logarithms is worse than the state of the art in factoring integers. In this case, we have to use general-purpose algorithms (that do not have subexponential running times). This fact is, for example, exploited by elliptic curve cryptography.

7.5 HARD-CORE PREDICATES

The fact that f is a one-way function does not mean that $f(x)$ necessarily hides all information about x. Nevertheless, it seems likely that there is at least some information (e.g., one bit) about x that is hard to guess from $f(x)$, given that x in its entirety is hard to compute. One may ask if it is possible to point to specific bits of x that are hard to compute and how hard to compute they are. These questions can be answered in the affirmative. A number of results are known that give a particular bit of x, which is hard to guess given $f(x)$ for some particular one-way functions.

A *hard-core predicate* for f is a predicate about x that cannot be computed from $f(x)$. More formally, a hard-core predicate B can be defined as suggested in Definition 7.12 and illustrated in Figure 7.1.

Definition 7.12 (Hard-core predicate) *Let $f : X \rightarrow Y$ be a one-way function. A* hard-core predicate *of f is a Boolean predicate $B : X \rightarrow \{0, 1\}$, such that the following two conditions hold:*

- *$B(x)$ can be computed efficiently for all $x \in X$. Alternatively speaking, there is a PPT algorithm A that on input x outputs $B(x)$ for all $x \in X$.*
- *It is not known how to efficiently compute $B(x)$ for all $y = f(x) \in Y$. Alternatively speaking, there is no known PPT algorithm A that on input $f(x)$ outputs $B(x)$ for all $x \in X$.*

Again, there are many possibilities to express the same properties. For example, the second condition can also be expressed as follows: for every PPT A and for all constants c, there exists a k_0 such that

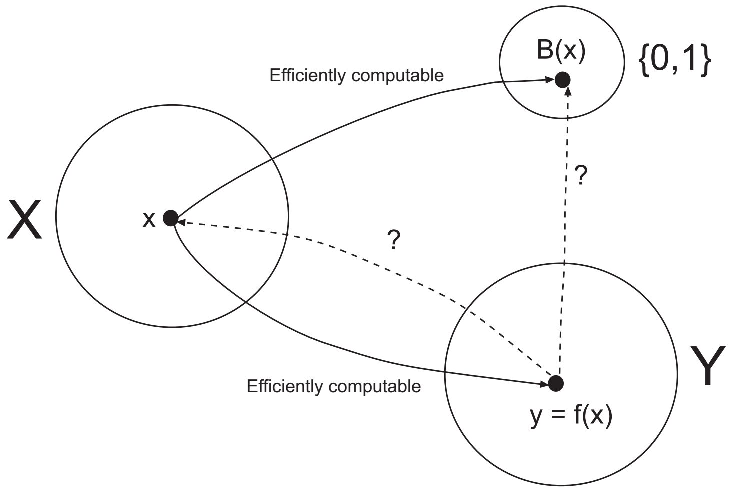

Figure 7.1 A hard-core predicate of a one-way function.

$$\Pr[A(f(x)) = B(x)] < \frac{1}{2} + \frac{1}{k^c}$$

for all $k > k_0$, where the probability is taken over over the random coin tosses of A and random choices of x of length k (i.e., the success probability of A is only negligibly smaller than $1/2$). It is simple and straightforward to extend the notion of a hard-core predicate for a family of one-way functions.

Historically, the notion of a hard-core predicate was first captured by Manuel Blum and Silvio Micali in a paper on pseudorandom number generation [13]. In fact, they showed that the *most significant bit* (MSB) is a hard-core predicate for the Exp family. This is in contrast to the RSA family, for which the *least significant bit* (LSB) represents a hard-core predicate [14]. In the context of probabilistic encryption, Oded Goldwasser and Micali showed that Square has a hard-core predicate, as well [15]. Andrew C. Yao[6] generalized the notion of a hard-core predicate and showed that given any one-way function f, there is a predicate $B(x)$ that is as hard to guess from $f(x)$ as to invert f [16].

6 In 2000, Andrew Chi-Chih Yao won the Turing Award for his seminal work on the theory of computation in general, and pseudorandom number generation, cryptography, and communication complexity in particular.

7.6 ELLIPTIC CURVE CRYPTOGRAPHY

Most public key cryptosystems get their security from the assumed intractability of inverting a one-way function (as discussed earlier). Against this background, it is important to note that inverting a one-way function is not necessarily equally difficult in all algebraic structures one may think of. If we look at inverting the discrete exponentiation function in $\mathbb{Z}_p^*$, then there are known algorithms that are subexponential. This need not be the case in all possible groups (in which the function is assumed to be one way). In fact, ECC has become popular (and important) mainly because groups have been found in which subexponential algorithms to invert the discrete exponentiation function (i.e., compute discrete logarithms) are not known to exist.[7] This basically means that one has to use a general-purpose (and exponential-time) algorithm to compute discrete logarithms and break the security of the corresponding public key cryptosystem accordingly. Note, however, that it is not known whether subexponential algorithms in these groups exist; we simply don't know them.

The fact that subexponential algorithms are not known to exist has the positive side effect (from the cryptographer's viewpoint) that the resulting elliptic curve cryptosystems are equally secure with smaller key sizes than their conventional counterparts. This is important for implementations in which key sizes and performance are important issues (e.g., smartcards). For example, to reach the security level of 1,024 (2,048) bits in a conventional public key cryptosystem (e.g., RSA), it is estimated that 163 (224) bits are sufficient for an elliptic curve cryptosystem (e.g., [17]). This is a nonnegligible factor that can speed up implementations considerably.

Most elliptic curve cryptosystems are based on the *elliptic curve discrete logarithm problem* (ECDLP) in such a group. Similar to the DLP, the ECDLP can be defined as suggested in Definition 7.13. Again, the ECDLP is assumed to be computationally intractable.

Definition 7.13 (Elliptic Curve Discrete Logarithm Problem) *Let $E(\mathbb{F}_q)$ be an elliptic curve over $\mathbb{F}_q$, P be a point on $E(\mathbb{F}_q)$ of order n, and Q be another point on $E(\mathbb{F}_q)$. The ECDLP is to determine an integer x (with $0 \leq x < n$), such that $Q = xP$.*

Based on the intractability assumption of the ECDLP, Neal Koblitz [18] and Victor Miller [19] independently proposed using elliptic curves to implement public key cryptosystems based on the DLP. This proposal dates back to the mid 1980s, and since then many public key cryptosystems have been reformulated in an elliptic

7 An interesting online tutorial about elliptic curves in general, and ECC in particular, is available at http://www.certicom.com/resources/ecc_tutorial/ecc_tutorial.html.

curve setting. Examples include Diffie-Hellman, ElGamal, and DSA. Today, many books address ECC and elliptic curve cryptosystems in detail (e.g., [20–23]). You may refer to any of these books if you want to get more involved in elliptic curves and ECC.

Since 1985, the ECDLP has received considerable attention from leading mathematicians around the world. It is currently believed that the ECDLP is much harder than integer factorization or DLP. More specifically, there is no algorithm known that has a subexponential running time in the worst case. A few vulnerabilities and potential attacks should be considered with care and kept in mind when elliptic curves are used. For example, it was shown that the ECDLP can be reduced to the DLP in extension fields of F_q, where the index-calculus methods can be applied [24]. However, this reduction algorithm is only efficient for a special class of elliptic curves known as *supersingular curves*. Moreover, there is a simple test to ensure that an elliptic curve is not supersingular and hence not vulnerable to this attack. Consequently, it is possible to avoid them in the first place. Some other vulnerabilities and potential attacks can be found in the literature.

A distinguishing feature of ECC is that each user may select a different elliptic curve $E(F_q)$—even if all users use the same underlying finite field F_q. From a security viewpoint, this flexibility is advantageous (because the elliptic curve can be changed periodically). From a practical viewpoint, however, this flexibility is also disadvantageous (because it makes interoperability much more difficult and because it has led to a situation in which the field of ECC is tied up in patents). Note that there is (more or less) only one way to implement a conventional public key cryptosystem, such as RSA, but usually many ways to implement an elliptic curve cryptosystem. In fact, one can work with different finite fields, different elliptic curves over these fields, and a wide variety of representations of the elements on these curves. Each choice has advantages and disadvantages, and one can construct an efficient curve for each application. Consequently, the relevant standardization bodies, such as the Institute of Electrical and Electronics Engineers (IEEE),[8] ISO/IEC JTC1, the American National Standards Institute (ANSI), and the National Institute of Standards and Technology (NIST), are working hard to come up with ECC standards and recommendations that are commonly accepted and widely deployed.[9]

7.7 FINAL REMARKS

In this chapter, we elaborated on one-way functions and trapdoor functions. More specifically, we defined the notion of a family of one-way functions or trapdoor

8 http://grouper.ieee.org/groups/1363

9 http://www.certicom.com/resources/standards/eccstandards.html

functions, and we overviewed and discussed some functions that are conjectured to be one way or trapdoor. More specifically, we looked at the discrete exponentiation function, the RSA function, and the modular square function. We further looked at hard-core predicates and algorithms for factoring integers or computing discrete logarithms. Curiously, factoring integers and computing discrete logarithms seem to have essentially the same difficulty (and computational complexity), at least as indicated by the current state-of-the-art algorithms.

Most public key cryptosystems in use today are based on one (or several) of the conjectured one-way functions mentioned earlier. This is also true for ECC, which works in cyclic groups in which known special-purpose algorithms to compute discrete logarithms do not work. From a practical viewpoint, ECC is interesting because it allows us to use smaller keys (compared to other public key cryptosystems). This is advantageous especially when it comes to implementing cryptographic systems and applications in environments that are restricted in terms of computational resources (e.g., smartcards). For the purpose of this book, however, we don't make a major distinction between public key cryptosystems that are based on the DLP and public key cryptosystems that are based on the ECDLP.

In either case, it is sometimes recommended to use cryptosystems that combine different candidate one-way functions in one way or another. If one of these functions then turns out not to be one way, then the other functions still remain and keep on securing the cryptosystem. Obviously, this strategy becomes useless if all functions turn out not to be one way.

References

[1] Pohlig, S., and M.E. Hellman, "An Improved Algorithm for Computing Logarithms over GF(p)," *IEEE Transactions on Information Theory*, Vol. 24, January 1978, pp. 106–110.

[2] Maurer, U.M., "Towards the Equivalence of Breaking the Diffie-Hellman Protocol and Computing Discrete Logarithms," *Proceedings of CRYPTO '94*, Springer-Verlag, LNCS 839, 1994, 271–281.

[3] Maurer, U.M., and S. Wolf, "The Diffie-Hellman Protocol," *Designs, Codes, and Cryptography*, Special Issue on Public Key Cryptography, Vol. 19, No. 2-3, 2000, pp. 147–171.

[4] Lenstra, H.W., "Factoring Integers with Elliptic Curves," *Annals of Mathematics*, Vol. 126, 1987, pp. 649–673.

[5] Morrison, M.A., and J. Brillhart, "Method of Factoring and the Factorization of $\mathbb{F}_7$," *Mathematics of Computation*, Vol. 29, 1975, pp. 183–205.

[6] Pomerance, C., "The Quadratic Sieve Factoring Algorithm," *Proceedings of EUROCRYPT '84*, Springer-Verlag, 1984, pp. 169–182.

[7] Lenstra, A.K., and H.W. Lenstra, *The Development of the Number Field Sieve*. Springer-Verlag, LNCS 1554, New York, 1993.

[8] Gardner, M., “A New Kind of Cipher That Would Take Millions of Years to Break,” *Scientific American*, Vol. 237, pp. 120–124.

[9] Atkins, D., “The Magic Words Are Squeamish Ossifrage,” *Proceedings of ASIACRYPT ’94*, Springer-Verlag, LNCS 917, 1995, pp. 263–277.

[10] Shamir, A., “Factoring Large Numbers with the TWINKLE Device,” *Proceedings of CHES ’99*, Springer-Verlag, LNCS 1717, 1999, pp. 2–12.

[11] Shamir, A., and E. Tromer, “Factoring Large Numbers with the TWIRL Device,” *Proceedings of CRYPTO 2003*, Springer-Verlag, LNCS 2729, 2003, pp. 1–26.

[12] Shoup, V., “Lower Bounds for Discrete Logarithms and Related Problems,” *Proceedings of EUROCRYPT ’97*, Springer-Verlag, LNCS 1233, 1997, pp. 256–266.

[13] Blum, M., and S. Micali, “How to Generate Cryptographically Strong Sequences of Pseudo-Random Bits,” *SIAM Journal of Computing*, Vol. 13, No. 4, November 1984, pp. 850–863.

[14] Alexi, W.B., et al., “RSA/Rabin Functions: Certain Parts Are as Hard as the Whole,” *SIAM Journal of Computing*, Vol. 17, No. 2, April 1988, pp. 194–209.

[15] Goldwasser, S., and S. Micali, “Probabilistic Encryption,” *Journal of Computer and System Sciences*, Vol. 28, No. 2, April 1984, pp. 270–299.

[16] Yao, A.C., “Theory and Application of Trapdoor Functions,” *Proceedings of 23rd IEEE Symposium on Foundations of Computer Science*, IEEE Press, Chicago, 1982, pp. 80–91.

[17] Lenstra, A.K., and E.R. Verheul, “Selecting Cryptographic Key Sizes,” *Journal of Cryptology*, Vol. 14, No. 4, 2001, pp. 255–293.

[18] Koblitz, N., “Elliptic Curve Cryptosystems,” *Mathematics of Computation*, Vol. 48, No. 177, 1987, pp. 203–209.

[19] Miller, V., “Use of Elliptic Curves in Cryptography,” *Proceedings of CRYPTO ’85*, LNCS 218, Springer-Verlag, 1986, pp. 417–426.

[20] Koblitz, N.I., *A Course in Number Theory and Cryptography*, 2nd edition. Springer-Verlag, New York, 1994.

[21] Blake, I., G. Seroussi, and N. Smart, *Elliptic Curves in Cryptography*, Cambridge University Press, Cambridge, UK, 2000.

[22] Washington, L.C., *Elliptic Curves: Number Theory and Cryptography.* Chapman & Hall/CRC, Boca Raton, FL, 2003.

[23] Hankerson, D., A. Menezes, and S.A. Vanstone, *Guide to Elliptic Curve Cryptography.* Springer-Verlag, New York, NY, 2004.

[24] Menezes, A., T. Okamoto, and S.A. Vanstone, “Reducing Elliptic Curve Logarithms to Logarithms in a Finite Field,” *IEEE Transactions on Information Theory*, Vol. 39, 1993, pp. 1639–1646.