Chapter 6
Introduction to Public-Key Cryptography

Before we learn about the basics of public-key cryptography, let us recall that the term *public-key cryptography* is used interchangeably with *asymmetric cryptography*; they both denote exactly the same thing and are used synonymously.

As stated in Chap. 1, symmetric cryptography has been used for at least 4000 years. Public-key cryptography, on the other hand, is quite new. It was publicly introduced by Whitfield Diffie, Martin Hellman and Ralph Merkle in 1976. Much more recently, in 1997 British documents which were declassified revealed that the researchers James Ellis, Clifford Cocks and Graham Williamson from the UK's Government Communications Headquarters (GCHQ) discovered and realized the principle of public-key cryptography a few years earlier, in 1972. However, it is still being debated whether the government office fully recognized the far-reaching consequences of public-key cryptography for commercial security applications.

In this chapter you will learn:

- A brief history of public-key cryptography
- The pros and cons of public-key cryptography
- Some number theoretical topics that are needed for understanding public-key algorithms, most importantly the extended Euclidean algorithm

Christof Paar et al., *Understanding Cryptography*

6.1 Symmetric vs. Asymmetric Cryptography

In this chapter we will see that asymmetric, i.e., public-key, algorithms are very different from symmetric algorithms such as AES or DES. Most public-key algorithms are based on number-theoretic functions. This is quite different from symmetric ciphers, where the goal is usually *not* to have a compact mathematical description between input and output. Even though mathematical structures are often used for small blocks *within* symmetric ciphers, for instance, in the AES S-Box, this does not mean that the entire cipher forms a compact mathematical description.

Symmetric Cryptography Revisited

In order to understand the principle of asymmetric cryptography, let us first recall the basic symmetric encryption scheme in Fig. 6.1.

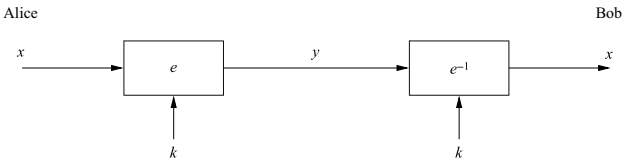

Fig. 6.1 Principle of symmetric-key encryption

Such a system is symmetric with respect to two properties:

1. The *same secret key* is used for encryption and decryption.
2. The encryption and decryption *function* are very similar (in the case of DES they are essentially identical).

There is a simple analogy for symmetric cryptography, as shown in Fig. 6.2. Assume there is a safe with a strong lock. Only Alice and Bob have a copy of the key for the lock. The action of encrypting of a message can be viewed as putting the message in the safe. In order to read, i.e., decrypt, the message, Bob uses his key and opens the safe.

Modern symmetric algorithms such as AES or 3DES are very secure, fast and are in widespread use. However, there are several shortcomings associated with symmetric-key schemes, as discussed below.

Key Distribution Problem The key must be established between Alice and Bob using a secure channel. Remember that the communication link for the message is not secure, so sending the key over the channel directly — which would be the most convenient way of transporting it — can't be done.

Number of Keys Even if we solve the key distribution problem, we must potentially deal with a very large number of keys. If each pair of users needs a separate pair of keys in a network with n users, there are

Fig. 6.2 Analogy for symmetric encryption: a safe with one lock

$$\frac{n \cdot (n-1)}{2}$$

key pairs, and every user has to store $n-1$ keys securely. Even for mid-size networks, say, a corporation with 2000 people, this requires more than 4 million key pairs that must be generated and transported via secure channels. More about this problem is found in Sect. 13.1.3. (There are smarter ways of dealing with keys in symmetric cryptography networks as detailed in Sect. 13.2; however, those approaches have other problems such as a single point of failure.)

No Protection Against Cheating by Alice or Bob Alice and Bob have the same capabilities, since they possess the same key. As a consequence, symmetric cryptography cannot be used for applications where we would like to prevent cheating by either Alice or Bob as opposed to cheating by an outsider like Oscar. For instance, in e-commerce applications it is often important to prove that Alice actually sent a certain message, say, an online order for a flat screen TV. If we only use symmetric cryptography and Alice changes her mind later, she can always claim that Bob, the vendor, has falsely generated the electronic purchase order. Preventing this is called *nonrepudiation* and can be achieved with asymmetric cryptography, as discussed in Sect. 10.1.1. Digital signatures, which are introduced in Chap. 10, provide nonrepudiation.

Fig. 6.3 Analogy for public-key encryption: a safe with public lock for depositing a message and a secret lock for retrieving a message

Principles of Asymmetric Cryptography

In order to overcome these drawbacks, Diffie, Hellman and Merkle had a revolutionary proposal based on the following idea: It is not necessary that the key possessed by the person who *encrypts* the message (that's Alice in our example) is secret. The crucial part is that Bob, the receiver, can only *decrypt* using a secret key. In order to realize such a system, Bob publishes a public encryption key which is known to everyone. Bob also has a matching secret key, which is used for decryption. Thus, Bob's key k consists of two parts, a public part, k_{pub}, and a private one, k_{pr}.

A simple analogy of such a system is shown in Fig. 6.3. This systems works quite similarly to the good old mailbox on the corner of a street: Everyone can put a letter in the box, i.e., encrypt, but only a person with a private (secret) key can retrieve letters, i.e., decrypt. If we assume we have cryptosystems with such a functionality, a basic protocol for public-key encryption looks as shown in Fig. 6.4.

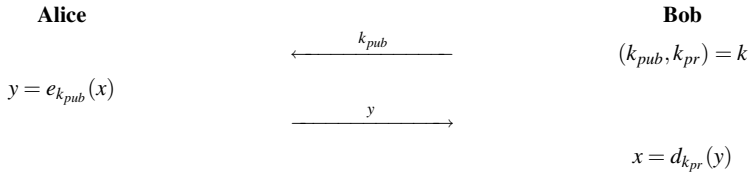

Fig. 6.4 Basic protocol for public-key encryption

By looking at that protocol you might argue that even though we can encrypt a message without a secret channel for key establishment, we still cannot exchange a key if we want to encrypt with, say, AES. However, the protocol can easily be modified for this use. What we have to do is to *encrypt a symmetric key*, e.g., an AES key, using the public-key algorithm. Once the symmetric key has been decrypted by Bob, both parties can use it to encrypt and decrypt messages using symmetric ciphers. Figure 6.5 shows a basic key transport protocol where we use AES as the symmetric cipher for illustration purposes (of course, one can use any other symmetric algorithm in such a protocol). The main advantage of the protocol in Fig. 6.5 over the protocol in Fig. 6.4 is that the payload is encrypted with a symmetric cipher, which tends to be much faster than an asymmetric algorithm.

From the discussion so far, it looks as though asymmetric cryptography is a desirable tool for security applications. The question remains how one can build public-key algorithms. In Chaps. 7, 8 and 9 we introduce most asymmetric schemes of practical relevance. They are all built from one common principle, the one-way function. The informal definition of it is as follows:

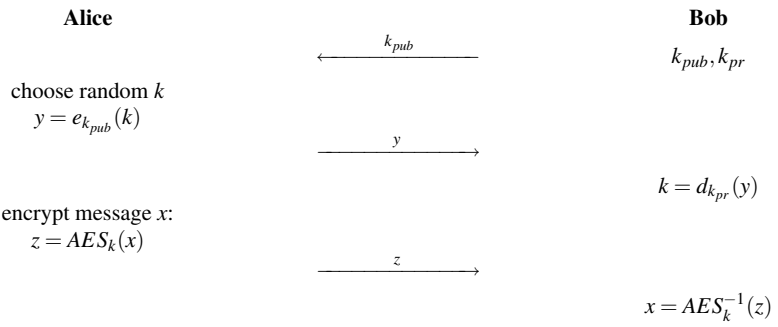

Fig. 6.5 Basic key transport protocol with AES as an example of a symmetric cipher

Definition 6.1.1 One-way function
A function $f()$ is a one-way function if:

1. $y = f(x)$ *is computationally easy, and*
2. $x = f^{-1}(y)$ *is computationally infeasible.*

Obviously, the adjectives "easy" and "infeasible" are not particularly exact. In mathematical terms, a function is easy to compute if it can be evaluated in polynomial time, i.e., its running time is a polynomial expression. In order to be useful in practical crypto schemes, the computation $y = f(x)$ should be sufficiently fast that it does not lead to unacceptably slow execution times in an application. The inverse computation $x = f^{-1}(y)$ should be so computationally intensive that it is not feasible to evaluate it in any reasonable time period, say, 10,000 years, when using the best known algorithm.

There are two popular one-way functions which are used in practical public-key schemes. The first is the integer factorization problem, on which RSA is based. Given two large primes, it is easy to compute the product. However, it is very difficult to factor the resulting product. In fact, if each of the primes has 150 or more decimal digits, the resulting product cannot be factored, even with thousands of PCs running for many years. The other one-way function that is used widely is the discrete logarithm problem. This is not quite as intuitive and is introduced in Chap. 8.

6.2 Practical Aspects of Public-Key Cryptography

Actual public-key algorithms will be introduced in the next chapters, since there is some mathematics we must study first. However, it is very interesting to look at the principal security functions of public-key cryptography which we address in this section.

6.2.1 Security Mechanisms

As shown in the previous section, public-key schemes can be used for encryption of data. It turns out that we can do many other, previously unimaginable, things with public-key cryptography. The main functions that they can provide are listed below:

Main Security Mechanisms of Public-Key Algorithms:

Key Establishment There are protocols for establishing secret keys over an insecure channel. Examples for such protocols include the Diffie–Hellman key exchange (DHKE) or RSA key transport protocols.

Nonrepudiation Providing nonrepudiation and message integrity can be realized with digital signature algorithms, e.g., RSA, DSA or ECDSA.

Identification We can identify entities using challenge-and-response protocols together with digital signatures, e.g., in applications such as smart cards for banking or for mobile phones.

Encryption We can encrypt messages using algorithms such as RSA or Elgamal.

We note that identification and encryption can also be achieved with symmetric ciphers, but they typically require much more effort with key management. It looks as though public-key schemes can provide all functions required by modern security protocols. Even though this is true, the major drawback in practice is that encryption of data is very computationally intensive — or more colloquially: extremely slow — with public-key algorithms. Many block and stream ciphers can encrypt about one hundred to one thousand times faster than public-key algorithms. Thus, somewhat ironically, public-key cryptography is rarely used for the actual encryption of data. On the other hand, symmetric algorithms are poor at providing nonrepudiation and key establishment functionality. In order to use the best of both worlds, most practical protocols are *hybrid protocols* which incorporate both symmetric and public-key algorithms. Examples include the SSL/TLS potocol that is commonly used for secure Web connections, or IPsec, the security part of the Internet communication protocol.

6.2.2 The Remaining Problem: Authenticity of Public Keys

From the discussion so far we've seen that a major advantage of asymmetric schemes is that we can freely distribute public keys, as shown in the protocols in Figs. 6.4 and 6.5. However, in practice, things are a bit more tricky because we still have to assure the *authenticity* of public keys. In other words: Do we really know that a certain public key belongs to a certain person? In practice, this issue is often

solved with what is called *certificates*. Roughly speaking, certificates bind a public key to a certain identity. This is a major issue in many security application, e.g., when doing e-commerce transactions on the Internet. We discuss this topic in more detail in Sect. 13.3.2.

Another problem, which is not as fundamental, is that public-key algorithms require very long keys, resulting in slow execution times. The issue of key lengths and security is discussed below.

6.2.3 Important Public-Key Algorithms

In the previous chapters, we learned about some block ciphers, DES and AES. However, there exist many other symmetric algorithms. Several hundred algorithms have been proposed over the years and even though a lot were found not to be secure, there exist many cryptographically strong ones as discussed in Sect. 3.7. The situation is quite different for asymmetric algorithms. There are only three major families of public-key algorithms which are of practical relevance. They can be classified based on their underlying computational problem.

Public-Key Algorithm Families of Practical Relevance

Integer-Factorization Schemes Several public-key schemes are based on the fact that it is difficult to factor large integers. The most prominent representative of this algorithm family is RSA.

Discrete Logarithm Schemes There are several algorithms which are based on what is known as the discrete logarithm problem in finite fields. The most prominent examples include the Diffie–Hellman key exchange, Elgamal encryption or the Digital Signature Algorithm (DSA).

Elliptic Curve (EC) Schemes A generalization of the discrete logarithm algorithm are elliptic curve public-key schemes. The most popular examples include Elliptic Curve Diffie–Hellman key exchange (ECDH) and the Elliptic Curve Digital Signature Algorithm (ECDSA).

The first two families were proposed in the mid-1970s, and elliptic curves were proposed in the mid-1980s. There are no known attacks against any of the schemes if the parameters, especially the operand and key lengths, are chosen carefully. Algorithms belonging to each of the families will be introduced in Chaps. 7, 8 and 9. It is important to note that each of the three families can be used to provide the main public-key mechanisms of key establishment, nonrepudiation through digital signatures and encryption of data.

In addition to the three families above, there have been proposals for several other public-key schemes. They often lack cryptographic maturity, i.e., it is not known how robust they are against mathematical attacks. Multivariate quadratic

(MQ) or some lattice-based schemes are examples of this. Another common problem is that they have poor implementation characteristics, like key lengths in the range of megabytes, e.g., the McEliece cryptosystems. However, there are also some other schemes, for instance, hyperelliptic curve cryptosystems, which are both as efficient and secure as the three established families shown above, but which simply have not gained widespread adoption. For most applications it is recommended to use public-key schemes from the three established algorithm families.

6.2.4 Key Lengths and Security Levels

All three of the established public-key algorithm families are based on number-theoretic functions. One distinguishing feature of them is that they require arithmetic with very long operands and keys. Not surprisingly, the longer the operands and keys, the more secure the algorithms become. In order to compare different algorithms, one often considers the *security level*. An algorithm is said to have a "security level of n bit" if the best known attack requires 2^n steps. This is a quite natural definition because symmetric algorithms with a security level of n have a key of length n bit. The relationship between cryptographic strength and security is not as straightforward in the asymmetric case, though. Table 6.1 shows recommended bit lengths for public-key algorithms for the four security levels 80, 128, 192 and 256 bit. We see from the table that RSA-like schemes and discrete-logarithm schemes require very long operands and keys. The key length of elliptic curve schemes is significantly smaller, yet still twice as long as symmetric ciphers with the same cryptographic strength.

Table 6.1 Bit lengths of public-key algorithms for different security levels

Algorithm Family	Cryptosystems	Security Level (bit)			
		80	128	192	256
Integer factorization	RSA	1024 bit	3072 bit	7680 bit	15360 bit
Discrete logarithm	DH, DSA, Elgamal	1024 bit	3072 bit	7680 bit	15360 bit
Elliptic curves	ECDH, ECDSA	160 bit	256 bit	384 bit	512 bit
Symmetric-key	AES, 3DES	80 bit	128 bit	192 bit	256 bit

You may want to compare this table with the one given in Sect. 1.3.2, which provides information about the security estimations of symmetric-key algorithms. In order to provide long-term security, i.e., security for a timespan of several decades, a security level of 128 bit should be chosen, which requires fairly long keys for all three algorithm families.

An undesired consequence of the long operands is that public-key schemes are extremely arithmetically intensive. As mentioned earlier, it is not uncommon that one public-operation, say a digital signature, is by 2–3 orders of magnitude slower than the encryption of one block using AES or 3DES. Moreover, the computational

complexity of the three algorithm families grows roughly with the cube bit length. As an example, increasing the bit length from 1024 to 3076 in a given RSA signature generation software results in an execution that is $3^3 = 27$ times slower! On modern PCs, execution times in the range of several 10 msec to a few 100 msec are common, which does not pose a problem for many applications. However, public-key performance can be a more serious bottleneck in constrained devices where small CPUs are prevalent, e.g., mobile phones or smart cards, or on network servers that have to compute many public-key operations per second. Chaps. 7, 8 and 9 introduce several techniques for implementing public-key algorithms reasonably efficiently.

6.3 Essential Number Theory for Public-Key Algorithms

We will now study a few techniques from number theory which are essential for public-key cryptography. We introduce the Euclidean algorithm, Euler's phi function as well as Fermat's Little Theorem and Euler's theorem. All are important for asymmetric algorithms, especially for understanding the RSA crypto scheme.

6.3.1 Euclidean Algorithm

We start with the problem of computing the *greatest common divisor (gcd)*. The gcd of two positive integers r_0 and r_1 is denoted by

$$\gcd(r_0, r_1)$$

and is the largest positive number that divides both r_0 and r_1. For instance $\gcd(21,9) = 3$. For small numbers, the gcd is easy to calculate by factoring both numbers and finding the highest common factor.

Example 6.1. Let $r_0 = 84$ and $r_1 = 30$. Factoring yields

$$\begin{aligned} r_0 &= 84 = 2 \cdot 2 \cdot 3 \cdot 7 \\ r_1 &= 30 = 2 \cdot 3 \cdot 5 \end{aligned}$$

The gcd is the product of all common prime factors:

$$2 \cdot 3 = 6 = \gcd(30, 84)$$

$\diamond$

For the large numbers which occur in public-key schemes, however, factoring often is not possible, and a more efficient algorithm is used for gcd computations, the Euclidean algorithm. The algorithm, which is also referred to as Euclid's algorithm, is based on the simple observation that

$$\gcd(r_0, r_1) = \gcd(r_0 - r_1, r_1),$$

where we assume that $r_0 > r_1$, and that both numbers are positive integers. This property can easily be proven: Let $\gcd(r_0, r_1) = g$. Since g divides both r_0 and r_1, we can write $r_0 = g \cdot x$ and $r_1 = g \cdot y$, where $x > y$, and x and y are coprime integers, i.e., they do not have common factors. Moreover, it is easy to show that $(x - y)$ and y are also coprime. It follows from here that:

$$\gcd(r_0 - r_1, r_1) = \gcd(g \cdot (x - y), g \cdot y) = g.$$

Let's verify this property with the numbers from the previous example:

Example 6.2. Again, let $r_0 = 84$ and $r_1 = 30$. We now look at the gcd of $(r_0 - r_1)$ and r_1:

$$\begin{aligned} r_0 - r_1 &= 54 = 2 \cdot 3 \cdot 3 \cdot 3 \\ r_1 &= 30 = 2 \cdot 3 \cdot 5 \end{aligned}$$

The largest common factor still is $2 \cdot 3 = 6 = \gcd(30, 54) = \gcd(30, 84)$.

◇

It also follows immediately that we can apply the process iteratively:

$$\gcd(r_0, r_1) = \gcd(r_0 - r_1, r_1) = \gcd(r_0 - 2r_1, r_1) = \cdots = \gcd(r_0 - m\, r_1, r_1)$$

as long as $(r_0 - m\, r_1) > 0$. The algorithm uses the fewest number of steps if we choose the maximum value for m. This is the case if we compute:

$$\gcd(r_0, r_1) = \gcd(r_0 \bmod r_1, r_1).$$

Since the first term $(r_0 \bmod r_1)$ is smaller than the second term r_1, we usually swap them:

$$\gcd(r_0, r_1) = \gcd(r_1, r_0 \bmod r_1).$$

The core observation from this process is that we can **reduce the problem of finding the gcd of two given numbers to that of the gcd of two smaller numbers.** This process can be applied recursively until we obtain finally $\gcd(r_l, 0) = r_l$. Since each iteration preserves the gcd of the previous iteration step, it turns out that this final gcd is the gcd of the original problem, i.e.,

$$\gcd(r_0, r_1) = \cdots = \gcd(r_l, 0) = r_l.$$

We first show some examples for finding the gcd using the Euclidean algorithm and then discuss the algorithm a bit more formally.

Example 6.3. Let $r_0 = 27$ and $r_1 = 21$. Fig. 6.6 gives us some feeling for the algorithm by showing how the lengths of the parameters shrink in every iteration. The shaded parts in the iteration are the new remainders $r_2 = 6$ (first iteration), and $r_3 = 3$ (second iteration) which form the input terms for the next iterations. Note

that in the last iteration the remainder is $r_4 = 0$, which indicates the termination of the algorithm. ◇

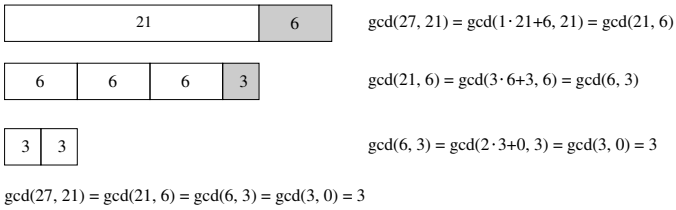

Fig. 6.6 Example of the Euclidean algorithm for the input values $r_0 = 27$ and $r_1 = 21$

It is also helpful to look at the Euclidean algorithm with slightly larger numbers, as happens in Example 6.4.

Example 6.4. Let $r_0 = 973$ and $r_1 = 301$. The gcd is then computed as

$973 = 3 \cdot 301 + 70$	$\gcd(973,301) = \gcd(301,70)$
$301 = 4 \cdot 70 + 21$	$\gcd(301,70) = \gcd(70,21)$
$70 = 3 \cdot 21 + 7$	$\gcd(70,21) = \gcd(21,7)$
$21 = 3 \cdot 7 + 0$	$\gcd(21,7) = \gcd(7,0) = 7$

◇

By now we should have an idea of Euclid's algorithm, and we can give a more formal description of the algorithm.

Euclidean Algorithm
Input: positive integers r_0 and r_1 with $r_0 > r_1$
Output: $\gcd(r_0, r_1)$
Initialization: $i = 1$
Algorithm:

```
1     DO
1.1       i  = i + 1
1.2       r_i = r_{i-2} mod r_{i-1}
      WHILE r_i ≠ 0
2     RETURN
          gcd(r_0, r_1) = r_{i-1}
```

Note that the algorithm terminates if a remainder with the value $r_i = 0$ is computed. The remainder computed in the previous iteration, denoted by r_{l-1}, is the gcd of the original problem.

The Euclidean algorithm is very efficient, even with the very long numbers typically used in public-key cryptography. The number of iterations is close to the number of digits of the input operands. That means, for instance, that the number of iterations of a gcd involving 1024-bit numbers is 1024 times a constant. Of course, algorithms with a few thousand iterations can easily be executed on today's PCs, making the algorithms very efficient in practice.

6.3.2 Extended Euclidean Algorithm

So far, we have seen that finding the gcd of two integers r_0 and r_1 can be done by recursively reducing the operands. However, it turns out that finding the gcd is not the main application of the Euclidean algorithm. An extension of the algorithm allows us to compute modular inverses, which is of major importance in public-key cryptography. In addition to computing the gcd, the *extended Euclidean algorithm* (EEA) computes a linear combination of the form:

$$\gcd(r_0, r_1) = s \cdot r_0 + t \cdot r_1$$

where s and t are integer coefficients. This equation is often referred to as *Diophantine equation*.

The question now is: how do we compute the two coefficients s and t? The idea behind the algorithm is that we execute the standard Euclidean algorithm, but we express the current remainder r_i in every iteration as a linear combination of the form

$$r_i = s_i r_0 + t_i r_1. \tag{6.1}$$

If we succeed with this, we end up in the last iteration with the equation:

$$r_l = \gcd(r_0, r_1) = s_l r_0 + t_l r_1 = s r_0 + t r_1.$$

This means that the last coefficient s_l is the coefficient s in Eq. (6.1) we are looking for, and also $t_l = t$. Let's look at an example.

Example 6.5. We consider the extended Euclidean algorithm with the same values as in the previous example, $r_0 = 973$ and $r_1 = 301$. On the left-hand side, we compute the standard Euclidean algorithm, i.e., we compute new remainders $r_2, r_3, \ldots$. Also, we have to compute the integer quotient q_{i-1} in every iteration. On the right-hand side we compute the coefficients s_i and t_i such that $r_i = s_i r_0 + t_i r_1$. The coefficients are always shown in brackets.

i	$r_{i-2} = q_{i-1} \cdot r_{i-1} + r_i$	$r_i = [s_i]r_0 + [t_i]r_1$
2	$973 = 3 \cdot 301 + 70$	$70 = [1]r_0 + [-3]r_1$
3	$301 = 4 \cdot 70 + 21$	$21 = 301 - 4 \cdot 70$ $= r_1 - 4(1r_0 - 3\,r_1)$ $= [-4]r_0 + [13]r_1$
4	$70 = 3 \cdot 21 + 7$	$7 = 70 - 3 \cdot 21$ $= (1r_0 - 3r_1) - 3(-4r_0 + 13r_1)$ $= [13]r_0 + [-42]r_1$
	$21 = 3 \cdot 7 + 0$	

The algorithm computed the three parameters $\gcd(973, 301) = 7$, $s = 13$ and $t = -42$. The correctness can be verified by:

$$\gcd(973, 301) = 7 = [13]973 + [-42]301 = 12649 - 12642.$$

$\diamond$

You should carefully watch the algebraic steps taking place in the right column of the example above. In particular, observe that the linear combination on the right-hand side is always constructed with the help of the *previous* linear combinations. We will now derive recursive formulae for computing s_i and r_i in every iteration. Assume we are in iteration with index i. In the two previous iterations we computed the values

$$r_{i-2} = [s_{i-2}]r_0 + [t_{i-2}]r_1 \tag{6.2}$$
$$r_{i-1} = [s_{i-1}]r_0 + [t_{i-1}]r_1 \tag{6.3}$$

In the current iteration i we first compute the quotient q_{i-1} and the new remainder r_i from r_{i-1} and r_{i-2}:

$$r_{i-2} = q_{i-1} \cdot r_{i-1} + r_i.$$

This equation can be rewritten as:

$$r_i = r_{i-2} - q_{i-1} \cdot r_{i-1}. \tag{6.4}$$

Recall that our goal is to represent the new remainder r_i as a linear combination of r_0 and r_1 as shown in Eq. (6.1). The core step for achieving this happens now: in Eq. (6.4) we simply substitute r_{i-2} by Eq. (6.2) and r_{i-1} by Eq. (6.3):

$$r_i = (s_{i-2}r_0 + t_{i-2}r_1) - q_{i-1}(s_{i-1}r_0 + t_{i-1}r_1)$$

If we rearrange the terms we obtain the desired result:

$$\begin{aligned} r_i &= [s_{i-2} - q_{i-1}s_{i-1}]r_0 + [t_{i-2} - q_{i-1}t_{i-1}]r_1 \\ r_i &= [s_i]r_0 + [t_i]r_1 \end{aligned} \tag{6.5}$$

Eq. (6.5) also gives us immediately the recursive formulae for computing s_i and t_i, namely $s_i = s_{i-2} - q_{i-1}s_{i-1}$ and $t_i = t_{i-2} - q_{i-1}t_{i-1}$. These recursions are valid

for index values $i \geq 2$. Like any recursion, we need starting values for s_0, s_1, t_0, t_1. These initial values (which we derive in Problem 6.13) can be shown to be $s_0 = 1, s_1 = 0, t_0 = 0, t_1 = 1$.

Extended Euclidean Algorithm (EEA)
Input: positive integers r_0 and r_1 with $r_0 > r_1$
Output: $\gcd(r_0, r_1)$, as well as s and t such that $\gcd(r_0, r_1) = s \cdot r_0 + t \cdot r_1$.
Initialization:
$s_0 = 1 \quad t_0 = 0$
$s_1 = 0 \quad t_1 = 1$
$i = 1$
Algorithm:

1 DO
1.1 $i = i + 1$
1.2 $r_i = r_{i-2} \bmod r_{i-1}$
1.3 $q_{i-1} = (r_{i-2} - r_i)/r_{i-1}$
1.4 $s_i = s_{i-2} - q_{i-1} \cdot s_{i-1}$
1.5 $t_i = t_{i-2} - q_{i-1} \cdot t_{i-1}$
WHILE $r_i \neq 0$
2 RETURN
$\gcd(r_0, r_1) = r_{i-1}$
$s = s_{i-1}$
$t = t_{i-1}$

As mentioned above, the main application of the EEA in asymmetric cryptography is to compute the inverse modulo of an integer. We already encountered this problem in the context of the affine cipher in Chap. 1. For the affine cipher, we were required to find the inverse of the key value a modulo 26. With the Euclidean algorithm, this is straightforward. Let's assume we want to compute the inverse of r_1 mod r_0 where $r_1 < r_0$. Recall from Sect. 1.4.2 that the inverse only exists if $\gcd(r_0, r_1) = 1$. Hence, if we apply the EEA, we obtain $s \cdot r_0 + t \cdot r_1 = 1 = \gcd(r_0, r_1)$. Taking this equation modulo r_0 we obtain:

$$\begin{aligned} s \cdot r_0 + t \cdot r_1 &\equiv 1 \\ s \cdot 0 + t \cdot r_1 &\equiv 1 \bmod r_0 \\ r_1 \cdot t &\equiv 1 \bmod r_0 \end{aligned} \tag{6.6}$$

Equation (6.6) is exactly the definition of the inverse of r_1. That means, that t itself is the inverse of r_1:

$$t = r_1^{-1} \bmod r_0.$$

Thus, if we need to compute an inverse $a^{-1} \bmod m$, we apply the EEA with the input parameters m and a. The output value t that is computed is the inverse. Let's look at an example.

Example 6.6. Our goal is to compute $12^{-1} \bmod 67$. The values 12 and 67 are relatively prime, i.e., $\gcd(67,12) = 1$. If we apply the EEA, we obtain the coefficients s and t in $\gcd(67,12) = 1 = s \cdot 67 + t \cdot 12$. Starting with the values $r_0 = 67$ and $r_1 = 12$, the algorithm proceeds as follows:

i	q_{i-1}	r_i	s_i	t_i
2	5	7	1	-5
3	1	5	-1	6
4	1	2	2	-11
5	2	1	-5	**28**

This gives us the linear combination

$$-5 \cdot 67 + 28 \cdot 12 = 1$$

As shown above, the inverse of 12 follows from here as

$$12^{-1} \equiv 28 \mod 67.$$

This result can easily be verified

$$28 \cdot 12 = 336 \equiv 1 \mod 67.$$

◇

Note that the s coefficient is not needed and is in practice often not computed. Please note also that the result of the algorithm can be a negative value for t. The result is still correct, however. We have to compute $t = t + r_0$, which is a valid operation since $t \equiv t + r_0 \bmod r_0$.

For completeness, we show how the EEA can also be used for computing multiplicative inverses in Galois fields. In modern cryptography this is mainly relevant for the derivation of the AES S-Boxes and for elliptic curve public-key algorithms. The EEA can be used completely analogously with polynomials instead of integers. If we want to compute an inverse in a finite field $GF(2^m)$, the inputs to the algorithm are the field element $A(x)$ and the irreducible polynomial $P(x)$. The EEA computes the auxiliary polynomials $s(x)$ and $t(x)$, as well as the greatest common divisor $\gcd(P(x),A(x))$ such that:

$$s(x)P(x) + t(x)A(x) = \gcd(P(x),A(x)) = 1$$

Note that since $P(x)$ is irreducible, the gcd is always equal to 1. If we take the equation above and reduce both sides modulo $P(x)$, it is straightforward to see that the auxiliary polynomial $t(x)$ is equal to the inverse of $A(x)$:

$$s(x)\,0 + t(x)\,A(x) \equiv 1 \bmod P(x)$$
$$t(x) \equiv A^{-1}(x) \bmod P(x)$$

We give at this point an example of the algorithm for the small field $GF(2^3)$.

Example 6.7. We are looking for the inverse of $A(x) = x^2$ in the finite field $GF(2^3)$ with $P(x) = x^3 + x + 1$. The initial values for the $t(x)$ polynomial are: $t_0(x) = 0$, $t_1(x) = 1$

Iteration	$r_{i-2}(x)$	$= [q_{i-1}(x)]\, r_{i-1}(x) + [r_i(x)]$	$t_i(x)$
2	$x^3 + x + 1$	$= [x]\, x^2 + [x+1]$	$t_2 = t_0 - q_1 t_1 = 0 - x\,1 \equiv x$
3	x^2	$= [x]\,(x+1) + [x]$	$t_3 = t_1 - q_2 t_2 = 1 - x\,(x) \equiv 1 + x^2$
4	$x+1$	$= [1]\, x + [1]$	$t_4 = t_2 - q_3 t_3 = x - 1\,(1 + x^2)$
			$t_4 \equiv 1 + x + x^2$
5	x	$= [x]\, 1 + [0]$	Termination since $r_5 = 0$

Note that polynomial coefficients are computed in $GF(2)$, and since addition and multiplication are the same operations, we can always replace a negative coefficient (such as $-x$) by a positive one. The new quotient and the new remainder that are computed in every iteration are shown in brackets above. The polynomials $t_i(x)$ are computed according to the recursive formula that was used for computing the integers t_i earlier in this section. The EEA terminates if the remainder is 0, which is the case in the iteration with index 5. The inverse is now given as the last $t_i(x)$ value that was computed, i.e., $t_4(x)$:

$$A^{-1}(x) = t(x) = t_4(x) = x^2 + x + 1.$$

Here is the check that $t(x)$ is in fact the inverse of x^2, where we use the properties that $x^3 \equiv x + 1 \bmod P(x)$ and $x^4 \equiv x^2 + x \bmod P(x)$:

$$\begin{aligned} t_4(x) \cdot x^2 &= x^4 + x^3 + x^2 \\ &\equiv (x^2 + x) + (x+1) + x^2 \bmod P(x) \\ &\equiv 1 \bmod P(x) \end{aligned}$$

$\diamond$

Note that in every iteration of the EEA, one uses long division (not shown above) to determine the new quotient $q_{i-1}(x)$ and the new remainder $r_i(x)$.

The inverse Table 4.2 in Chap. 4 was computed using the extended Euclidean algorithm.

6.3.3 Euler's Phi Function

We now look at another tool that is useful for public-key cryptosystems, especially for RSA. We consider the ring $\mathbb{Z}_m$, i.e., the set of integers $\{0, 1, \ldots, m-1\}$. We are

interested in the (at the moment seemingly odd) problem of knowing *how many* numbers in this set are relatively prime to m. This quantity is given by *Euler's phi function*, which is defined as follows:

Definition 6.3.1 Euler's Phi Function
The number of integers in $\mathbb{Z}_m$ relatively prime to m is denoted by $\Phi(m)$.

We first look at some examples and calculate Euler's phi function by actually counting all the integers in $\mathbb{Z}_m$ which are relatively prime.

Example 6.8. Let $m = 6$. The associated set is $\mathbb{Z}_6 = \{0,1,2,3,4,5\}$.
$\gcd(0,6) = 6$
$\gcd(1,6) = 1$ $\star$
$\gcd(2,6) = 2$
$\gcd(3,6) = 3$
$\gcd(4,6) = 2$
$\gcd(5,6) = 1$ $\star$
Since there are two numbers in the set which are relatively prime to 6, namely 1 and 5, the phi function takes the value 2, i.e., $\Phi(6) = 2$.
$\diamond$

Here is another example:

Example 6.9. Let $m = 5$. The associated set is $\mathbb{Z}_5 = \{0,1,2,3,4\}$.
$\gcd(0,5) = 5$
$\gcd(1,5) = 1$ $\star$
$\gcd(2,5) = 1$ $\star$
$\gcd(3,5) = 1$ $\star$
$\gcd(4,5) = 1$ $\star$
This time we have four numbers which are relatively prime to 5, hence, $\Phi(5) = 4$.
$\diamond$

From the examples above we can guess that calculating Euler's phi function by running through all elements and computing the gcd is extremely slow if the numbers are large. In fact, computing Euler's phi function in this naïve way is completely out of reach for the large numbers occurring in public-key cryptography. Fortunately, there exists a relation to calculate it much more easily if we know the factorization of m, which is given in following theorem.

Theorem 6.3.1 *Let m have the following canonical factorization*

$$m = p_1^{e_1} \cdot p_2^{e_2} \cdot \ldots \cdot p_n^{e_n},$$

where the p_i are distinct prime numbers and e_i are positive integers, then

$$\Phi(m) = \prod_{i=1}^{n} (p_i^{e_i} - p_i^{e_i - 1}).$$

Since the value of n, i.e., the number of distinct prime factors, is always quite small even for large numbers m, evaluating the product symbol $\prod$ is computationally easy. Let's look at an example where we calculate Euler's phi function using the relation:

Example 6.10. Let $m = 240$. The factorization of 240 in the canonical factorization form is

$$m = 240 = 16 \cdot 15 = 2^4 \cdot 3 \cdot 5 = p_1^{e_1} \cdot p_2^{e_2} \cdot p_3^{e_3}$$

There are three distinct prime factors, i.e., $n = 3$. The value for Euler's phi functions follows then as:

$$\Phi(m) = (2^4 - 2^3)(3^1 - 3^0)(5^1 - 5^0) = 8 \cdot 2 \cdot 4 = 64.$$

That means that 64 integers in the range $\{0, 1, \ldots, 239\}$ are coprime to $m = 240$. The alternative method, which would have required to evaluate the gcd 240 times, would have been much slower even for this small number.

◇

It is important to stress that we need to know the factorization of m in order to calculate Euler's phi function quickly in this manner. As we will see in the next chapter, this property is at the heart of the RSA public-key scheme: Conversely, if we know the factorization of a certain number, we can compute Euler's phi function and decrypt the ciphertext. If we do not know the factorization, we cannot compute the phi function and, hence, cannot decrypt.

6.3.4 Fermat's Little Theorem and Euler's Theorem

We describe next two theorems which are quite useful in public-key crpytography. We start with *Fermat's Little Theorem*.[1] The theorem is helpful for primality testing and in many other aspects of public-key cryptography. The theorem gives a seemingly surprising result if we do exponentiations modulo an integer.

[1] You should not confuse this with Fermat's Last Theorem, one of the most famous number-theoretical problems, which was proved in the 1990s after 350 years.

Theorem 6.3.2 Fermat's Little Theorem
Let a be an integer and p be a prime, then:

$$a^p \equiv a \pmod{p}.$$

We note that arithmetic in finite fields $GF(p)$ is done modulo p, and hence, the theorem holds for all integers a which are elements of a finite field $GF(p)$. The theorem can be stated in the form:

$$a^{p-1} \equiv 1 \pmod{p}$$

which is often useful in cryptography. One application is the computation of the inverse in a finite field. We can rewrite the equation as $a \cdot a^{p-2} \equiv 1 \pmod{p}$. This is exactly the definition of the multiplicative inverse. Thus, we immediately have a way for inverting an integer a modulo a prime:

$$a^{-1} \equiv a^{p-2} \pmod{p} \tag{6.7}$$

We note that this inversion method holds only if p is a prime. Let's look at an example:

Example 6.11. Let $p = 7$ and $a = 2$. We can compute the inverse of a as:

$$a^{p-2} = 2^5 = 32 \equiv 4 \bmod 7.$$

This is easy to verify: $2 \cdot 4 \equiv 1 \bmod 7$.
$\diamond$

Performing the exponentiation in Eq. (6.7) is usually slower than using the extended Euclidean algorithm. However, there are situations where it is advantageous to use Fermat's Little Theorem, e.g., on smart cards or other devices which have a hardware accelerator for fast exponentiation anyway. This is not uncommon because many public-key algorithms require exponentiation, as we will see in subsequent chapters.

A generalization of Fermat's Little Theorem to any integer moduli, i.e., moduli that are not necessarily primes, is *Euler's theorem*.

Theorem 6.3.3 Euler's Theorem
Let a and m be integers with $\gcd(a,m) = 1$*, then:*

$$a^{\Phi(m)} \equiv 1 \pmod{m}.$$

Since it works modulo m, it is applicable to integer rings $\mathbb{Z}_m$. We show now an example for Euler's theorem with small values.

Example 6.12. Let $m = 12$ and $a = 5$. First, we compute Euler's phi function of m:

$$\Phi(12) = \Phi(2^2 \cdot 3) = (2^2 - 2^1)(3^1 - 3^0) = (4-2)(3-1) = 4.$$

Now we can verify Euler's theorem:

$$5^{\Phi(12)} = 5^4 = 25^2 = 625 \equiv 1 \bmod 12.$$

$\diamond$

It is easy to show that Fermat's Little Theorem is a special case of Euler's theorem. If p is a prime, it holds that $\Phi(p) = (p^1 - p^0) = p - 1$. If we use this value for Euler's theorem, we obtain: $a^{\Phi(p)} = a^{p-1} \equiv 1 \pmod{p}$, which is exactly Fermat's Little Theorem.

6.4 Discussion and Further Reading

Public-Key Cryptography in General Asymmetric cryptography was introduced in the landmark paper by Whitfield Diffie and Martin Hellman [58]. Ralph Merkle independently invented the concept of asymmetric cryptography but proposed an entirely different public-key algorithm [121]. There are a few good accounts of the history of public-key cryptography. The treatment in [57] by Diffie is recommended. Another good overview on public-key cryptography is [127]. A very instructive and detailed history of elliptic curve cryptography, including the relatively intense competition between RSA and ECC during the 1990s, is described in [100]. More recent development in asymmetric cryptography is tracked by the Workshop on Public-Key Cryptography (PKC) series.

Modular Arithmetic With respect to the mathematics introduced in this chapter, the introductory books on number theory recommended in Sect. 1.5 make good sources for further reading. In practical terms, the Extended Euclidean Algorithm (EEA) is the most crucial, since virtually all implementations of public-key schemes incorporate it, especially modular inversion. An important acceleration technique for the scheme is the binary EEA. Its advantage over the standard EEA is that it replaces divisions by bit shifts. This is in particular attractive for the very long numbers occurring in public-key schemes.

Alternative Public-Key Algorithms In addition to the three established families of asymmetric schemes, there exist several others. First, there are algorithms which have been broken or are believed to be insecure, e.g., knapsack schemes. Second, there are generalizations of the established algorithms, e.g., hyperelliptic curves, algebraic varieties or non-RSA factoring-based schemes. These schemes use the same one-way function, that is, integer factorization or the discrete logarithm in certain groups. Third, there are asymmetric algorithms which are based on different one-way functions. Four families of one-way function are of particular interest: hash-based, code-based, lattice-based and multivariate quadratic (MQ) public-key algorithms. There are, of course, reasons why they are not as widely used today.

In most cases, they have either practical drawbacks, such as very long keys (sometimes in the range of several megabytes), or the cryptographic strength is not well understood. Since about 2005, there has been growing interest in the cryptographic community in such asymmetric schemes. This is in part motivated by the fact that no quantum computing attacks are currently known against these four families of alternative asymmetric schemes. This is in contrast to RSA, discrete logarithm, and elliptic curve schemes and their variants, which are all vulnerable to attacks using quantum computers [153]. Even though it is not clear whether quantum computers will ever exist (the most optimistic estimates state that they are still several decades away), the alternative public-key algorithms are at times collectively referred to as *post-quantum cryptography*. A recent book [18] and a new workshop series [36, 35] provide more information about this area of active research.

6.5 Lessons Learned

- Public-key algorithms have capabilities that symmetric ciphers don't have, in particular digital signature and key establishment functions.
- Public-key algorithms are computationally intensive (a nice way of saying that they are *slow*), and hence are poorly suited for bulk data encryption.
- Only three families of public-key schemes are widely used. This is considerably fewer than in the case of symmetric algorithms.
- The extended Euclidean algorithm allows us to compute modular inverses quickly, which is important for almost all public-key schemes.
- Euler's phi function gives us the number of elements smaller than an integer n that are relatively prime to n. This is an important function for the RSA crypto scheme.

Problems

6.1. As we have seen in this chapter, public-key cryptography can be used for encryption and key exchange. Furthermore, it has some properties (such as nonrepudiation) which are not offered by secret key cryptography.

So why do we still use symmetric cryptography in current applications?

6.2. In this problem, we want to compare the computational performance of symmetric and asymmetric algorithms. Assume a fast public-key library such as OpenSSL [132] that can decrypt data at a rate of 100 Kbit/sec using the RSA algorithm on a modern PC. On the same machine, AES can decrypt at a rate of 17 Mbit/sec. Assume we want to decrypt a movie stored on a DVD. The movie requires 1 GByte of storage. How long does decryption take with either algorithm?

6.3. Assume a (small) company with 120 employees. A new security policy demands encrypted message exchange with a symmetric cipher. How many keys are required, if you are to ensure a secret communication for every possible pair of communicating parties?

6.4. The level of security in terms of the corresponding bit length directly influences the performance of the respective algorithm. We now analyze the influence of increasing the security level on the runtime.

Assume that a commercial Web server for an online shop can use either RSA or ECC for signature generation. Furthermore, assume that signature generation for RSA-1024 and ECC-160 takes 15.7 ms and 1.3 ms, respectively.

1. Determine the increase in runtime for signature generation if the security level from RSA is increased from 1024 bit to 3072 bit.
2. How does the runtime increase from 1024 bit to 15,360 bit?
3. Determine these numbers for the respective security levels of ECC.
4. Describe the difference between RSA and ECC when increasing the security level.

Hint: Recall that the computational complexity of both RSA and ECC grows with the cube of bit length. You may want to use Table 6.1 to determine the adequate bit length for ECC, given the security level of RSA.

6.5. Using the basic form of Euclid's algorithm, compute the greatest common divisor of

1. 7469 and 2464
2. 2689 and 4001

For this problem use only a pocket calculator. Show every iteration step of Euclid's algorithm, i.e., don't write just the answer, which is only a number. Also, for every gcd, provide the chain of gcd computations, i.e.,

$$\gcd(r_0, r_1) = \gcd(r_1, r_2) = \cdots .$$

6.6. Using the extended Euclidean algorithm, compute the greatest common divisor and the parameters s,t of

1. 198 and 243
2. 1819 and 3587

For every problem check if $s\,r_0 + t\,r_1 = \gcd(r_0, r_1)$ is actually fulfilled. The rules are the same as above: use a pocket calculator and show what happens in every iteration step.

6.7. With the Euclidean algorithm we finally have an efficient algorithm for finding the multiplicative inverse in Z_m that is much better than exhaustive search. Find the inverses in Z_m of the following elements a modulo m:

1. $a = 7$, $m = 26$ (affine cipher)
2. $a = 19$, $m = 999$

Note that the inverses must again be elements in Z_m and that you can easily verify your answers.

6.8. Determine $\phi(m)$, for $m = 12, 15, 26$, according to the definition: Check for each positive integer n smaller m whether $\gcd(n,m) = 1$. (You do *not* have to apply Euclid's algorithm.)

6.9. Develop formulae for $\phi(m)$ for the special cases when

1. m is a prime
2. $m = p \cdot q$, where p and q are primes. This case is of great importance for the RSA cryptosystem. Verify your formula for $m = 15, 26$ with the results from the previous problem.

6.10. Compute the inverse a^{-1} mod n with Fermat's Theorem (if applicable) or Euler's Theorem:

- $a = 4$, $n = 7$
- $a = 5$, $n = 12$
- $a = 6$, $n = 13$

6.11. Verify that Euler's Theorem holds in Z_m, $m = 6, 9$, for all elements a for which $\gcd(a,m) = 1$. Also verify that the theorem does not hold for elements a for which $\gcd(a,m) \neq 1$.

6.12. For the affine cipher in Chapter 1 the multiplicative inverse of an element modulo 26 can be found as

$$a^{-1} \equiv a^{11} \bmod 26.$$

Derive this relationship by using Euler's Theorem.

6.13. The extended Euclidean algorithm has the initial conditions $s_0 = 1, s_1 = 0, t_0 = 0, t_1 = 1$. *Derive* these conditions. It is helpful to look at how the general iteration formula for the Euclidean algorithm was derived in this chapter.