Chapter 7
The RSA Cryptosystem

My comments in red.

Ghaith

After Whitfield Diffie and Martin Hellman introduced public-key cryptography in their landmark 1976 paper [58], a new branch of cryptography suddenly opened up. As a consequence, cryptologists started looking for methods with which public-key encryption could be realized. In 1977, Ronald Rivest, Adi Shamir and Leonard Adleman (cf. Fig. 7.1) proposed a scheme which became the most widely used asymmetric cryptographic scheme, RSA.

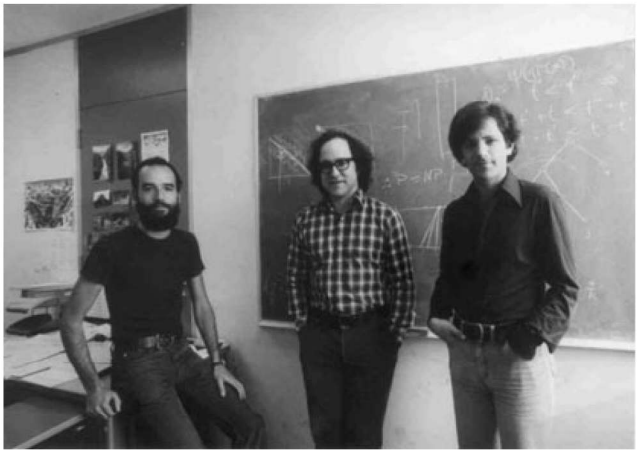

Fig. 7.1 An early picture of Adi Shamir, Ron Rivest, and Leonard Adleman (reproduced with permission from Ron Rivest)

In this chapter you will learn:

- How RSA works
- Practical aspects of RSA, such as computation of the parameters, and fast encryption and decryption
- Security estimations
- Implementational aspects

7.1 Introduction

The RSA crypto scheme, sometimes referred to as the Rivest–Shamir–Adleman algorithm, is currently the most widely used asymmetric cryptographic scheme, even though elliptic curves and discrete logarithm schemes are gaining ground. RSA was patented in the USA (but not in the rest of the world) until 2000.

There are many applications for RSA, but in practice it is most often used for:

- encryption of small pieces of data, especially for key transport
- digital signatures, which is discussed in Chap. 10, e.g., for digital certificates on the Internet

However, it should be noted that RSA encryption is not meant to replace symmetric ciphers because it is several times slower than ciphers such as AES. This is because of the many computations involved in performing RSA (or any other public-key algorithm) as we learn later in this chapter. Thus, the main use of the encryption feature is to securely exchange a key for a symmetric cipher (key transport). In practice, RSA is often used together with a symmetric cipher such as AES, where the symmetric cipher does the actual bulk data encryption.

The underlying one-way function of RSA is the integer factorization problem: Multiplying two large primes is computationally easy (in fact, you can do it with paper and pencil), but factoring the resulting product is very hard. Euler's theorem (Theorem 6.3.3) and Euler's phi function play important roles in RSA. In the following, we first describe how encryption, decryption and key generation work, then we talk about practical aspects of RSA.

7.2 Encryption and Decryption

RSA encryption and decryption is done in the integer ring $\mathbb{Z}_n$ and modular computations play a central role. Recall that rings and modular arithmetic in rings were introduced in Sect. 1.4.2. RSA encrypts plaintexts x, where we consider the bit string representing x to be an element in $\mathbb{Z}_n = \{0, 1, \ldots, n-1\}$. As a consequence the binary value of the plaintext x must be less than n. The same holds for the ciphertext. Encryption with the public key and decryption with the private key are as shown below:

RSA Encryption Given the public key $(n, e) = k_{pub}$ and the plaintext x, the encryption function is:

$$y = e_{k_{pub}}(x) \equiv x^e \bmod n \tag{7.1}$$

where $x, y \in \mathbb{Z}_n$.

RSA Decryption Given the private key $d = k_{pr}$ and the ciphertext y, the decryption function is:

$$x = d_{k_{pr}}(y) \equiv y^d \bmod n \tag{7.2}$$

where $x, y \in \mathbb{Z}_n$.

In practice, x, y, n and d are very long numbers, usually 1024 bit long or more. The value e is sometimes referred to as *encryption exponent* or *public exponent*, and the private key d is sometimes called *decryption exponent* or *private exponent*. If Alice wants to send an encrypted message to Bob, Alice needs to have his public key (n, e), and Bob decrypts with his private key d. We discuss in Sect. 7.3 how these three crucial parameters d, e, and n are generated.

Even without knowing more details, we can already state a few requirements for the RSA cryptosystem:

1. Since an attacker has access to the public key, it must be computationally infeasible to determine the private-key d given the public-key values e and n.
2. Since x is only unique up to the size of the modulus n, we cannot encrypt more than l bits with one RSA encryption, where l is the bit length of n.
3. It should be relatively easy to calculate $x^e \bmod n$, i.e., to encrypt, and $y^d \bmod n$, i.e., to decrypt. This means we need a method for fast exponentiation with very long numbers.
4. For a given n, there should be many private-key/public-key pairs, otherwise an attacker might be able to perform a brute-force attack. (It turns out that this requirement is easy to satisfy.)

* Important point

7.3 Key Generation and Proof of Correctness

A distinctive feature of all asymmetric schemes is that there is a set-up phase during which the public and private key are computed. Depending on the public-key scheme, key generation can be quite complex. As a remark, we note that key generation is usually not an issue for block or stream ciphers.

Here are the steps involved in computing the public and private-key for an RSA cryptosystem.

Instead of the the Euler totient function φ(n), Carmichael's totient function λ(n) can be used for calculating the private exponent d.

Since φ(n) is always divisible by λ(n) the algorithm works as well.

RSA Key Generation

Output: public key: $k_{pub} = (n, e)$ and private key: $k_{pr} = (d)$

1. Choose two large primes p and q.
2. Compute $n = p \cdot q$.
3. Compute $\Phi(n) = (p-1)(q-1)$.
4. Select the public exponent $e \in \{1, 2, \ldots, \Phi(n) - 1\}$ such that

$$\gcd(e, \Phi(n)) = 1.$$

5. Compute the private key d such that

$$d \cdot e \equiv 1 \bmod \Phi(n)$$

The condition that $\gcd(e, \Phi(n)) = 1$ ensures that the inverse of e exists modulo $\Phi(n)$, so that there is always a private key d.

Two parts of the key generation are nontrivial: Step 1, in which the two large primes are chosen, as well as Steps 4 and 5 in which the public and private key are computed. The prime generation of Step 1 is quite involved and is addressed in Sect. 7.6. The computation of the keys d and e can be done at once using the extended Euclidean algorithm (EEA). In practice, one often starts by first selecting a public parameter e in the range $0 < e < \Phi(n)$. The value e must satisfy the condition $\gcd(e, \Phi(n)) = 1$. We apply the EEA with the input parameters n and e and obtain the relationship:

$$\gcd(\Phi(n), e) = s \cdot \Phi(n) + t \cdot e$$

If $\gcd(e, \Phi(n)) = 1$, we know that e is a valid public key. Moreover, we also know that the parameter t computed by the extended Euclidean algorithm is the inverse of e, and thus:

$$d = t \bmod \Phi(n)$$

In case that e and $\Phi(n)$ are not relatively prime, we simply select a new value for e and repeat the process. Note that the coefficient s of the EEA is not required for RSA and does not need to be computed.

We now see how RSA works by presenting a simple example.

Example 7.1. Alice wants to send an encrypted message to Bob. Bob first computes his RSA parameters in Steps 1–5. He then sends Alice his public key. Alice encrypts the message ($x = 4$) and sends the ciphertext y to Bob. Bob decrypts y using his private key.

Alice		**Bob**
message $x = 4$		1. choose $p = 3$ and $q = 11$
		2. $n = p \cdot q = 33$
		3. $\Phi(n) = (3-1)(11-1) = 20$
		4. choose $e = 3$
		5. $d \equiv e^{-1} \equiv 7 \bmod 20$
	$\xleftarrow{k_{pub}=(33,3)}$	
$y = x^e \equiv 4^3 \equiv 31 \bmod 33$		
	$\xrightarrow{y=31}$	
		$y^d = 31^7 \equiv 4 = x \bmod 33$

Note that the private and public exponents fulfill the condition $e \cdot d = 3 \cdot 7 \equiv 1 \bmod \Phi(n)$.

◇

Practical RSA parameters are much, much larger. As can be seen from Table 6.1, the RSA modulus n should be at least 1024 bit long, which results in a bit length for p and q of 512. Here is an example of RSA parameters for this bit length:

$$
\begin{aligned}
p &= E0DFD2C2A288ACEBC705EFAB30E4447541A8C5A47A37185C5A9 \\
&\quad CB98389CE4DE19199AA3069B404FD98C801568CB9170EB712BF \\
&\quad 10B4955CE9C9DC8CE6855C6123_h \\
q &= EBE0FCF21866FD9A9F0D72F7994875A8D92E67AEE4B515136B2 \\
&\quad A778A8048B149828AEA30BD0BA34B977982A3D42168F594CA99 \\
&\quad F3981DDABFAB2369F229640115_h \\
n &= CF33188211FDF6052BDBB1A37235E0ABB5978A45C71FD381A91 \\
&\quad AD12FC76DA0544C47568AC83D855D47CA8D8A779579AB72E635 \\
&\quad D0B0AAAC22D28341E998E90F82122A2C06090F43A37E0203C2B \\
&\quad 72E401FD06890EC8EAD4F07E686E906F01B2468AE7B30CBD670 \\
&\quad 255C1FEDE1A2762CF4392C0759499CC0ABECFF008728D9A11ADF_h \\
e &= 40B028E1E4CCF07537643101FF72444A0BE1D7682F1EDB553E3 \\
&\quad AB4F6DD8293CA1945DB12D796AE9244D60565C2EB692A89B888 \\
&\quad 1D58D278562ED60066DD8211E67315CF89857167206120405B0 \\
&\quad 8B54D10D4EC4ED4253C75FA74098FE3F7FB751FF5121353C554 \\
&\quad 391E114C85B56A9725E9BD5685D6C9C7EED8EE442366353DC39_h \\
d &= C21A93EE751A8D4FBFD77285D79D6768C58EBF283743D2889A3 \\
&\quad 95F266C78F4A28E86F545960C2CE01EB8AD5246905163B28D0B \\
&\quad 8BAABB959CC03F4EC499186168AE9ED6D88058898907E61C7CC \\
&\quad CC584D65D801CFE32DFC983707F87F5AA6AE4B9E77B9CE630E2 \\
&\quad C0DF05841B5E4984D059A35D7270D500514891F7B77B804BED81_h
\end{aligned}
$$

What is interesting is that the message x is first raised to the eth power during encryption and the result y is raised to the dth power in the decryption, and the result of this is again equal to the message x. Expressed as an equation, this process is:

$$d_{k_{pr}}(y) = d_{k_{pr}}(e_{k_{pub}}(x)) \equiv (x^e)^d \equiv x^{de} \equiv x \bmod n. \tag{7.3}$$

This is the essence of RSA. We will now prove why the RSA scheme works.

Proof. We need to show that decryption is the inverse function of encryption, $d_{k_{pr}}(e_{k_{pub}}(x)) = x$. We start with the construction rule for the public and private key: $d \cdot e \equiv 1 \bmod \Phi(n)$. By definition of the modulo operator, this is equivalent to:

$$d \cdot e = 1 + t \cdot \Phi(n),$$

where t is some integer. Inserting this expression in Eq. (7.3):

$$d_{k_{pr}}(y) \equiv x^{de} \equiv x^{1+t\cdot\Phi(n)} \equiv x^{t\cdot\Phi(n)} \cdot x^1 \equiv (x^{\Phi(n)})^t \cdot x \mod n. \tag{7.4}$$

This means we have to prove that $x \equiv (x^{\Phi(n)})^t \cdot x \mod n$. We use now Euler's Theorem from Sect. 6.3.3, which states that if $\gcd(x,n) = 1$ then $1 \equiv x^{\Phi(n)} \bmod n$. A minor generalization immediately follows:

$$1 \equiv 1^t \equiv (x^{\Phi(n)})^t \bmod n, \tag{7.5}$$

where t is any integer. For the proof we distinguish two cases:

First case: $\gcd(x,n) = 1$

Euler's Theorem holds here and we can insert Eq. (7.5) into (7.4):

$$d_{k_{pr}}(y) \equiv (x^{\Phi(n)})^t \cdot x \equiv 1 \cdot x \equiv x \bmod n. \qquad q.e.d.$$

This part of the proof establishes that decryption is actually the inverse function of encryption for plaintext values x which are relatively prime to the RSA modulus n. We provide now the proof for the other case.

Second case: $\gcd(x,n) = \gcd(x, p\cdot q) \neq 1$

Since p and q are primes, x must have one of them as a factor:

$$x = r \cdot p \quad \text{or} \quad x = s \cdot q,$$

where r,s are integers such that $r < q$ and $s < p$. Without loss of generality we assume $x = r \cdot p$, from which follows that $\gcd(x,q) = 1$. Euler's Theorem holds in the following form:

$$1 \equiv 1^t \equiv (x^{\Phi(q)})^t \bmod q,$$

where t is any positive integer. We now look at the term $(x^{\Phi(n)})^t$ again:

$$(x^{\Phi(n)})^t \equiv (x^{(q-1)(p-1)})^t \equiv ((x^{\Phi(q)})^t)^{p-1} \equiv 1^{(p-1)} = 1 \mod q.$$

Using the definition of the modulo operator, this is equivalent to:

$$(x^{\Phi(n)})^t = 1 + u \cdot q,$$

where u is some integer. We multiply this equation by x:

$$\begin{aligned} x \cdot (x^{\Phi(n)})^t &= x + x \cdot u \cdot q \\ &= x + (r \cdot p) \cdot u \cdot q \\ &= x + r \cdot u \cdot (p \cdot q) \\ &= x + r \cdot u \cdot n \\ x \cdot (x^{\Phi(n)})^t &\equiv x \mod n. \end{aligned} \tag{7.6}$$

Inserting Eq. (7.6) into Eq. (7.4) yields the desired result:

$$d_{k_{pr}} = (x^{\Phi(n)})^t \cdot x \equiv x \bmod n.$$

□

If this proof seems somewhat lengthy, please remember that the correctness of RSA is simply assured by Step 5 of the RSA key generation phase. The proof becomes simpler by using the Chinese Remainder Theorem which we have not introduced.

Do we need to implement fast exponentiation algorithem, or we can use a C++ library that does it?

7.4 Encryption and Decryption: Fast Exponentiation

Unlike symmetric algorithms such as AES, DES or stream ciphers, public-key algorithms are based on arithmetic with very long numbers. Unless we pay close attention to how to realize the necessary computations, we can easily end up with schemes that are too slow for practical use. If we look at RSA encryption and decryption in Eqs. (7.1) and (7.2), we see that both are based on modular exponentiation. We restate both operations here for convenience:

$$\begin{aligned} y &= e_{k_{pub}}(x) \equiv x^e \bmod n \quad \text{(encryption)} \\ x &= d_{k_{pr}}(y) \equiv y^d \bmod n \quad \text{(decryption)} \end{aligned}$$

A straightforward way of exponentiation looks like this:

$$x \xrightarrow{SQ} x^2 \xrightarrow{MUL} x^3 \xrightarrow{MUL} x^4 \xrightarrow{MUL} x^5 \ldots$$

where SQ denotes squaring and MUL multiplication. Unfortunately, the exponents e and d are in general very large numbers. The exponents are typically chosen in the range of 1024–3072 bit or even larger. (The public exponent e is sometimes chosen to be a small value, but d is always very long.) Straightforward exponentiation as shown above would thus require around 2^{1024} or more multiplications. Since the number of atoms in the visible universe is estimated to be around 2^{300}, computing 2^{1024} multiplications to set up one secure session for our Web browser is not

too tempting. The central question is whether there are considerably faster methods for exponentiation available. The answer is, luckily, yes. Otherwise we could forget about RSA and pretty much all other public-key cryptosystems in use today, since they all rely on exponentiation. One such method is the *square-and-multiply algorithm*. We first show a few illustrative examples with small numbers before presenting the actual algorithm.

Example 7.2. Let's look at how many multiplications are required to compute the simple exponentiation x^8. With the straightforward method:

$$x \xrightarrow{SQ} x^2 \xrightarrow{MUL} x^3 \xrightarrow{MUL} x^4 \xrightarrow{MUL} x^5 \xrightarrow{MUL} x^6 \xrightarrow{MUL} x^7 \xrightarrow{MUL} x^8$$

we need seven multiplications and squarings. Alternatively, we can do something faster:

$$x \xrightarrow{SQ} x^2 \xrightarrow{SQ} x^4 \xrightarrow{SQ} x^8$$

which requires only three squarings that are roughly as complex as a multiplication. ◇

This fast method works fine but is restricted to exponents that are powers of 2, i.e., values e and d of the form 2^i. Now the question is, whether we can extend the method to arbitrary exponents? Let us look at another example:

Example 7.3. This time we have the more general exponent 26, i.e., we want to compute x^{26}. Again, the naïve method would require 25 multiplications. A faster way is as follows:

$$x \xrightarrow{SQ} x^2 \xrightarrow{MUL} x^3 \xrightarrow{SQ} x^6 \xrightarrow{SQ} x^{12} \xrightarrow{MUL} x^{13} \xrightarrow{SQ} x^{26}.$$

This approach takes a total of six operations, two multiplications and four squarings. ◇

Looking at the last example, we see that we can achieve the desired result by performing two basic operations:

1. *squaring* the current result,
2. *multiplying* the current result by the base element x.

In the example above we computed the sequence SQ, MUL, SQ, SQ, MUL, SQ. However, we do not know the sequence in which the squarings and multiplications have to be performed for other exponents. One solution is the *square-and-multiply* algorithm. It provides a systematic way for finding the sequence in which we have to perform squarings and multiplications by x for computing x^H. Roughly speaking, the algorithm works as follows:

The algorithm is based on scanning the bit of the exponent from the left (the most significant bit) to the right (the least significant bit). In every iteration, i.e., for every exponent bit, the current result is squared. If and only if the currently

scanned exponent bit has the value 1, a multiplication of the current result by x is executed following the squaring.

This seems like a simple if somewhat odd rule. For better understanding, let's revisit the example from above. This time, let's pay close attention to the exponent bits.

Example 7.4. We again consider the exponentiation x^{26}. For the square-and-multiply algorithm, the binary representation of the exponent is crucial:

$$x^{26} = x^{11010_2} = x^{(h_4 h_3 h_2 h_1 h_0)_2}.$$

The algorithm scans the exponent bits, starting on the left with h_4 and ending with the rightmost bit h_0.

Step		
#0	$x = x^{\mathbf{1}_2}$	inital setting, bit processed: $h_4 = 1$
#1a	$(x^1)^2 = x^2 = x^{\mathbf{10}_2}$	SQ, bit processed: h_3
#1b	$x^2 \cdot x = x^3 = x^{10_2} x^{1_2} = x^{\mathbf{11}_2}$	MUL, since $h_3 = 1$
#2a	$(x^3)^2 = x^6 = (x^{11_2})^2 = x^{\mathbf{110}_2}$	SQ, bit processed: h_2
#2b		no MUL, since $h_2 = 0$
#3a	$(x^6)^2 = x^{12} = (x^{110_2})^2 = x^{\mathbf{1100}_2}$	SQ, bit processed: h_1
#3b	$x^{12} \cdot x = x^{13} = x^{1100_2} x^{1_2} = x^{\mathbf{1101}_2}$	MUL, since $h_1 = 1$
#4a	$(x^{13})^2 = x^{26} = (x^{1101_2})^2 = x^{\mathbf{11010}_2}$	SQ, bit processed: h_0
#4b		no MUL, since $h_0 = 0$

To understand the algorithm it is helpful to closely observe how the binary representation of the exponent evolves. We see that the first basic operation, squaring, results in a left shift of the exponent, with a 0 put in the rightmost position. The other basic operation, multiplication by x, results in filling a 1 into the rightmost position of the exponent. Compare how the highlighted exponents change from iteration to iteration.

◇

Here is the pseudo code for the square-and-multiply algorithm:

Square-and-Multiply for Modular Exponentiation
Input:
base element x
exponent $H = \sum_{i=0}^{t} h_i 2^i$ with $h_i \in 0, 1$ and $h_t = 1$
and modulus n
Output: $x^H \bmod n$
Initialization: $r = x$
Algorithm:

```
1   FOR i = t − 1 DOWNTO 0
1.1     r = r^2 mod n
        IF h_i = 1
1.2         r = r · x mod n
2   RETURN (r)
```

The modulo reduction is applied after each multiplication and squaring operation in order to keep the intermediate results small. It is helpful to compare this pseudo code with the verbal description of the algorithm above.

We determine now the complexity of the square-and-multiply algorithm for an exponent H with a bit length of $t+1$, i.e., $\lceil \log_2 H \rceil = t+1$. The number of squarings is independent of the actual value of H, but the number of multiplications is equal to the Hamming weight, i.e., the number of ones in its binary representation. Thus, we provide here the average number of multiplication, denoted by $\overline{MUL}$:

$$\#SQ = t$$
$$\#\overline{MUL} = 0.5t$$

Because the exponents used in cryptography have often good random properties, assuming that half of their bits have the value one is often a valid approximation.

Example 7.5. How many operations are required on average for an exponentiation with a 1024-bit exponent?

Straightforward exponentiation takes $2^{1024} \approx 10^{300}$ multiplications. That is completely impossible, no matter what computer resources we might have at hand. However, the square-and-multiply algorithm requires only

$$1.5 \cdot 1024 = 1536$$

squarings and multiplications on average. This is an impressive example for the difference of an algorithm with linear complexity (straightforward exponentiation) and logarithmic complexity (square-and-multiply algorithm). Remember, though, that each of the 1536 individual squarings and multiplications involves 1024-bit numbers. That means the number of integer operations on a CPU is much higher than 1536, but certainly doable on modern computers.

◇

7.5 Speed-up Techniques for RSA

Our presentation could include techniques to speed up RSA?

As we learned in Sect. 7.4, RSA involves exponentiation with very long numbers. Even if the low-level arithmetic involving modular multiplication and squaring as well as the square-and-multiply algorithm are implemented carefully, performing a full RSA exponentiation with operands of length 1024 bit or beyond is computationally intensive. Thus, people have studied speed-up techniques for RSA since its invention. We introduce two of the most popular general acceleration techniques in the following.

7.5.1 *Fast Encryption with Short Public Exponents*

A surprisingly simple and very powerful trick can be used when RSA operations with the public key e are concerned. This is in practice encryption and, as we'll learn later, verification of an RSA digital signature. In this situation, the public key e can be chosen to be a very small value. In practice, the three values $e = 3$, $e = 17$ and $e = 2^{16} + 1$ are of particular importance. The resulting complexities when using these public keys are given in Table 7.1.

Table 7.1 Complexity of RSA exponentiation with short public exponents

Public key e	e as binary string	#MUL + #SQ
3	11_2	2
17	10001_2	5
$2^{16}+1$	$1\,0000\,0000\,0000\,0001_2$	17

These complexities should be compared to the $1.5t$ multiplications and squarings that are required for exponents of full length. Here $t + 1$ is the bit length of the RSA modulus n, i.e., $\lceil \log_2 n \rceil = t + 1$. We note that all three exponents listed above have a low Hamming weight, i.e., number of ones in the binary representation. This results in a particularly low number of operations for performing an exponentiation. Interestingly, RSA is still secure if such short exponents are being used. Note that the private key d still has in general the full bit length $t + 1$ even though e is short.

An important consequence of the use of short public exponents is that encryption of a message and verification of an RSA signature is a very fast operation. In fact, for these two operations, RSA is in almost all practical cases the fastest public-key scheme available. Unfortunately, there is no such easy way to accelerate RSA when the private key d is involved, i.e., for decryption and signature generation. Hence, these two operations tend to be slow. Other public-key algorithms, in particular elliptic curves, are often much faster for these two operations. The following section shows how we can achieve a more moderate speed-up when using the private exponent d.

7.5.2 Fast Decryption with the Chinese Remainder Theorem

We cannot choose a short private key without compromising the security for RSA. If we were to select keys d as short as we did in the case of encryption in the section above, an attacker could simply brute-force all possible numbers up to a given bit length, i.e., 50 bit. But even if the numbers are larger, say 128 bit, there are key recovery attacks. In fact, it can be shown that the private key must have a length of at least $0.3t$ bit, where t is the bit length of the modulus n. In practice, e is often chosen short and d has full bit length. What one does instead is to apply a method which is based on the Chinese Remainder Theorem (CRT). We do not introduce the CRT itself here but merely how it applies to accelerate RSA decryption and signature generation.

Our goal is to perform the exponentiation $x^d \bmod n$ efficiently. First we note that the party who possesses the private key also knows the primes p and q. The basic idea of the CRT is that rather than doing arithmetic with one "long" modulus n, we do two individual exponentiations modulo the two "short" primes p and q. This is a type of transformation arithmetic. Like any transform, there are three steps: transforming into the CRT domain, computation in the CRT domain, and inverse transformation of the result. Those three steps are explained below.

Transformation of the Input into the CRT Domain

We simply reduce the base element x modulo the two factors p and q of the modulus n, and obtain what is called the modular representation of x.

$$x_p \equiv x \bmod p$$
$$x_q \equiv x \bmod q$$

Exponentiation in the CRT Domain

With the reduced versions of x we perform the following two exponentiations:

$$y_p = x_p^{d_p} \bmod p$$
$$y_q = x_q^{d_q} \bmod q$$

where the two new exponents are given by:

$$d_p \equiv d \bmod (p-1)$$
$$d_q \equiv d \bmod (q-1)$$

Note that both exponents in the transform domain, d_p and d_q, are bounded by p and q, respectively. The same holds for the transformed results y_p and y_q. Since the two

primes are in practice chosen to have roughly the same bit length, the two exponents as well as y_p and y_q have about half the bit length of n.

Inverse Transformation into the Problem Domain

The remaining step is now to assemble the final result y from its modular representation (y_p, y_q). This follows from the CRT and can be done as:

$$y \equiv [q\,c_p]\,y_p + [p\,c_q]\,y_q \bmod n \tag{7.7}$$

where the coefficients c_p and c_q are computed as:

$$c_p \equiv q^{-1} \bmod p, \qquad c_q \equiv p^{-1} \bmod q$$

Since the primes change very infrequently for a given RSA implementation, the two expressions in brackets in Eq. (7.7) can be precomputed. After the precomputations, the entire reverse transformation is achieved with merely two modular multiplications and one modular addition.

Before we consider the complexity of RSA with CRT, let's have a look at an example.

Example 7.6. Let the RSA parameters be given by:

$$\begin{aligned} p &= 11 & e &= 7 \\ q &= 13 & d &\equiv e^{-1} \equiv 103 \bmod 120 \\ n &= p \cdot q = 143 \end{aligned}$$

We now compute an RSA decryption for the ciphertext $y = 15$ using the CRT, i.e., the value $y^d = 15^{103} \bmod 143$. In the first step, we compute the modular representation of y:

$$\begin{aligned} y_p &\equiv 15 \equiv 4 \bmod 11 \\ y_p &\equiv 15 \equiv 2 \bmod 13 \end{aligned}$$

In the second step, we perform the exponentiation in the transform domain with the short exponents. These are:

$$\begin{aligned} d_p &\equiv 103 \equiv 3 \bmod 10 \\ d_q &\equiv 103 \equiv 7 \bmod 12 \end{aligned}$$

Here are the exponentiations:

$$\begin{aligned} x_p &\equiv y_p^{d_p} = 4^3 = 64 \equiv 9 \bmod 11 \\ x_q &\equiv y_q^{d_q} = 2^7 = 128 \equiv 11 \bmod 13 \end{aligned}$$

In the last step, we have to compute x from its modular representation (x_p, x_q). For this, we need the coefficients:

$$c_p = 13^{-1} \equiv 2^{-1} \equiv 6 \bmod 11 \qquad c_q = 11^{-1} \equiv 6 \bmod 13$$

The plaintext x follows now as:

$$\begin{aligned} x &\equiv [qc_p]x_p + [pc_q]x_q \bmod n \\ x &\equiv [13 \cdot 6]9 + [11 \cdot 6]11 \bmod 143 \\ x &\equiv 702 + 726 = 1428 \equiv 141 \bmod 143 \end{aligned}$$

◇

If you want to verify the result, you can compute $y^d \bmod 143$ using the square-and-multiply algorithm.

We will now establish the computational complexity of the CRT method. If we look at the three steps involved in the CRT-based exponentiation, we conclude that for a practical complexity analysis the transformation and inverse transformation can be ignored since the operations involved are negligible compared to the actual exponentiations in the transform domain. For convenience, we restate these CRT exponentiations here:

$$\begin{aligned} y_p &= x_p^{d_p} \bmod p \\ y_q &= x_q^{d_q} \bmod q \end{aligned}$$

If we assume that n has $t+1$ bit, both p and q are about $t/2$ bit long. All numbers involved in the CRT exponentiations, i.e., x_p, x_q, d_p and d_q, are bound in size by p and q, respectively, and thus also have a length of about $t/2$ bit. If we use the square-and-multiply algorithm for the two exponentiations, each requires on average approximately $1.5t/2$ modular multiplications and squarings. Together, the number of multiplications and squarings is thus:

$$\#SQ + \#MUL = 2 \cdot 1.5t/2 = 1.5t$$

This appears to be exactly the same computational complexity as regular exponentiation without the CRT. However, each multiplication and squaring involves numbers which have a length of only $t/2$ bit. This is in contrast to the operations without CRT, where each multiplication was performed with t-bit variables. Since the complexity of multiplication decreases quadratically with the bit length, each $t/2$-bit multiplication is four times faster than a t-bit multiplication.[1] Thus, *the total speed-up obtained through the CRT is a factor of 4.* This speed-up by four can be very valuable in practice. Since there are hardly any drawbacks involved, CRT-based exponentiations are used in many cryptographic products, e.g., for Web browser encryption. The method is also particularly valuable for implementations on smart

[1] The reason for the quadratic complexity is easy to see with the following example. If we multiply a 4-digit decimal number *abcd* by another number *wxyz*, we multiply each digit from the first operand with each digit of the second operand, for a total of $4^2 = 16$ digit multiplications. On the other hand, if we multiply two numbers with two digits, i.e., *ab* times *wx*, only $2^2 = 4$ elementary multiplications are needed.

cards, e.g., for banking applications, which are only equipped with a small microprocessor. Here, digital signing is often needed, which involves the secret key d. By applying the CRT for signature computation, the smart card is four times as fast. For example, if a regular 1024-bit RSA exponentiation takes 3 sec, using the CRT reduces that time to 0.75 sec. This acceleration might make the difference between a product with high customer acceptance (0.75 sec) and a product with a delay that is not acceptable for many applications (3 sec). This example is a good demonstration how basic number theory can have direct impact in the real world.

7.6 Finding Large Primes

There is one important practical aspect of RSA which we have not discussed yet: generating the primes p and q in Step 1 of the key generation. Since their product is the RSA modulus $n = p \cdot q$, the two primes should have about half the bit length of n. For instance, if we want to set up RSA with a modulus of length $\lceil \log_2 n \rceil = 1024$, p and q should have a bit length of about 512 bit. The general approach is to generate integers at random which are then checked for primality, as depicted in Fig. 7.2, where RNG stands for random number generator. The RNG should be non predictable because if an attacker can compute or guess one of the two primes, RSA can be broken easily as we will see later in this chapter.

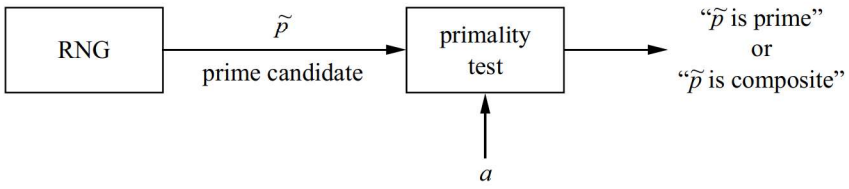

Fig. 7.2 Principal approach to generating primes for RSA

In order to make this approach work, we have to answer two questions:

1. How many random integers do we have to test before we have a prime? (If the likelihood of a prime is too small, it might take too long.)
2. How fast can we check whether a random integer is prime? (Again, if the test is too slow, the approach is impractical.)

It turns out that both steps are reasonably fast, as is discussed in the following.

7.6.1 How Common Are Primes?

Now we'll answer the question whether the likelihood that a randomly picked integer p is a prime is sufficiently high. We know from looking at the first few positive

integers that primes become less dense as the value increases:

$$2,3,5,7,11,13,17,19,23,29,31,37,\ldots$$

The question is whether there is still a reasonable chance that a random number with, say, 512 bit, is a prime. Luckily, this is the case. The chance that a randomly picked integer $\tilde{p}$ is a prime follows from the famous prime number theorem and is approximately $1/\ln(\tilde{p})$. In practice, we only test odd numbers so that the likelihood doubles. Thus, the probability for a random odd number $\tilde{p}$ to be prime is

$$P(\tilde{p} \text{ is prime}) \approx \frac{2}{\ln(\tilde{p})}.$$

In order to get a better feeling for what this probability means for RSA primes, let's look at an example:

Example 7.7. For RSA with a 1024-bit modulus n, the primes p and q each should have a length of about 512 bits, i.e., $p,q \approx 2^{512}$. The probability that a random odd number $\tilde{p}$ is a prime is

$$P(\tilde{p} \text{ is prime}) \approx \frac{2}{\ln(2^{512})} = \frac{2}{512\ln(2)} \approx \frac{1}{177}.$$

This means that we expect to test 177 random numbers before we find one that is a prime.

◇

The likelihood of integers being primes decreases slowly, proportional to the bit length of the integer. This means that even for very long RSA parameters, say with 4096 bit, the density of primes is still sufficiently high.

7.6.2 Primality Tests

The other step we have to do is to decide whether the randomly generated integers $\tilde{p}$ are primes. A first idea could be to factor the number in question. However, for the numbers used in RSA, factorization is not possible since p and q are too large. (In fact, we especially choose numbers that cannot be factored because factoring n is the best known attack against RSA.) The situation is not hopeless, though. Remember that we are *not* interested in the factorization of $\tilde{p}$. Instead we merely need the statement whether the number being tested is a prime or not. It turns out that such primality tests are computationally much easier than factorization. Examples for primality tests are the Fermat test, the Miller–Rabin test or variants of them. We introduce primality test algorithms in this section.

Practical primality tests behave somewhat unusually: if the integer $\tilde{p}$ in question is being fed into a primality test algorithm, the answer is either

1. "$\tilde{p}$ is composite" (i.e., not a prime), which is always a true statement, or
2. "$\tilde{p}$ is prime", which is only true with a high probability.

If the algorithm output is "composite", the situation is clear: The integer in question is not a prime and can be discarded. If the output statement is "prime", $\tilde{p}$ is probably a prime. In rare cases, however, an integers prompts a "prime" statement but it *lies*, i.e., it yields an incorrect positive answer. There is way to deal with this behavior. Practical primality tests are *probabilistic algorithms*. That means they have a second parameter a as input which can be chosen at random. If a composite number $\tilde{p}$ together with a parameter a yields the incorrect statement "$\tilde{p}$ is prime", we repeat the test a second time with a different value for a. The general strategy is to test a prime candidate $\tilde{p}$ so often with several different random values a that the likelihood that the pair $(\tilde{p},a)$ lies every single time is sufficiently small, say, less than 2^{-80}. Remember that as soon as the statement "$\tilde{p}$ is composite" occurs, we know for certain that $\tilde{p}$ is not a prime and we can discard it.

Fermat Primality Test

One primality test is based on Fermat's Little Theorem, Theorem (6.3.2).

Fermat Primality Test
Input: prime candidate $\tilde{p}$ and security parameter s
Output: statement "$\tilde{p}$ is composite" or "$\tilde{p}$ is likely prime"
Algorithm:

```
1     FOR i = 1 TO s
1.1       choose random a ∈ {2,3,...,p̃ − 2}
1.2       IF a^(p̃−1) ≢ 1
1.3           RETURN ("p̃ is composite")
2     RETURN ("p̃ is likely prime")
```

The idea behind the test is that Fermat's theorem holds for all primes. Hence, if a number is found for which $a^{\tilde{p}-1} \not\equiv 1$ in Step 1.2, it is certainly not a prime. However, the reverse is not true. There could be composite numbers which in fact fulfill the condition $a^{\tilde{p}-1} = 1$. In order to detect them, the algorithm is run s times with different values of a.

Unfortunately, there are certain composite integers which behave like primes in the Fermat test for many values of a. These are the *Carmichael numbers*. Given a Carmichael number C, the following expression holds for all integers a for which $\gcd(a,C) = 1$:

$$a^{C-1} \equiv 1 \bmod C$$

Such special composites are very rare. For instance, there exist approximately only $100,000$ Carmichael numbers below 10^{15}.

Example 7.8. Carmichael Number
$n = 561 = 3 \cdot 11 \cdot 17$ is a Carmichael number since

$$a^{560} \equiv 1 \bmod 561$$

for all $\gcd(a, 561) = 1$.
◇

If the prime factors of a Carmichael numbers are all large, there are only few bases a for which Fermat's test detects that the number is actually composite. For this reason, in practice the more powerful Miller–Rabin test is often used to generate RSA primes.

Miller–Rabin Primality Test

In contrast to Fermat's test, the Miller–Rabin test does not have any composite numbers for which a large number of base elements a yield the statement "prime". The test is based on the following theorem:

Theorem 7.6.1 *Given the decomposition of an odd prime candidate* $\tilde{p}$

$$\tilde{p} - 1 = 2^u r$$

where r *is odd. If we can find an integer* a *such that*

$$a^r \not\equiv 1 \bmod \tilde{p} \qquad \text{and} \qquad a^{r2^j} \not\equiv \tilde{p} - 1 \bmod \tilde{p}$$

for all $j = \{0, 1, \ldots, u-1\}$, *then* $\tilde{p}$ *is composite. Otherwise, it is probably a prime.*

We can turn this into an efficient primality test.

Miller–Rabin Primality Test
Input: prime candidate $\tilde{p}$ with $\tilde{p}-1=2^u r$ and security parameter s
Output: statement "$\tilde{p}$ is composite" or "$\tilde{p}$ is likely prime"
Algorithm:

```
1     FOR i = 1 TO s
          choose random a ∈ {2,3,...,p̃ − 2}
1.2       z ≡ a^r mod p̃
1.3       IF z ≠ 1 and z ≠ p̃ − 1
1.4           FOR j = 1 TO u − 1
                  z ≡ z^2 mod p̃
                  IF z = 1
                      RETURN ("p̃ is composite")
1.5           IF z ≠ p̃ − 1
                  RETURN ("p̃ is composite")
2     RETURN ("p̃ is likely prime")
```

Step 1.2 is computed by using the square-and-multiply algorithm. The IF statement in Step 1.3 tests the theorem for the case $j=0$. The FOR loop 1.4 and the IF statement 1.5 test the right-hand side of the theorem for the values $j=1,\ldots,u-1$.

It can still happen that a composite number $\tilde{p}$ gives the incorrect statement "prime". However, the likelihood of this rapidly decreases as we run the test with several different random base elements a. The number of runs is given by the security parameter s in the Miller–Rabin test. Table 7.2 shows how many different values a must be chosen in order to have a probability of less than 2^{-80} that a composite is incorrectly detected as a prime.

Table 7.2 Number of runs within the Miller–Rabin primality test for an error probability of less than 2^{-80}

Bit lengths of $\tilde{p}$	Security parameter s
250	11
300	9
400	6
500	5
600	3

Example 7.9. Miller–Rabin Test
Let $\tilde{p}=91$. Write $\tilde{p}$ as $\tilde{p}-1=2^1\cdot 45$. We select a security parameter of $s=4$. Now, choose s times a random value a:

1. Let $a=12$: $z=12^{45}\equiv 90 \bmod 91$, hence, $\tilde{p}$ is likely prime.
2. Let $a=17$: $z=17^{45}\equiv 90 \bmod 91$, hence, $\tilde{p}$ is likely prime.
3. Let $a=38$: $z=38^{45}\equiv 90 \bmod 91$, hence, $\tilde{p}$ is likely prime.

4. Let $a = 39$: $z = 39^{45} \equiv 78 \bmod 91$, hence, $\tilde{p}$ is composite.

Since the numbers 12, 17 and 38 give incorrect statements for the prime candidate $\tilde{p} = 91$, they are called "liars for 91".

◇

Should we implement Padding in our implementation?

7.7 RSA in Practice: Padding

What we described so far is the so-called "schoolbook RSA" system which has several weaknesses. In practice RSA has to be used with a padding scheme. Padding schemes are extremely important, and if not implemented properly, an RSA implementation may be insecure. The following properties of schoolbook RSA encryption are problematic:

- RSA encryption is deterministic, i.e., for a specific key, a particular plaintext is always mapped to a particular ciphertext. An attacker can derive statistical properties of the plaintext from the ciphertext. Furthermore, given some pairs of plaintext–ciphertext, partial information can be derived from new ciphertexts which are encrypted with the same key.
- Plaintext values $x = 0$, $x = 1$, or $x = -1$ produce ciphertexts equal to 0, 1, or -1.
- Small public exponents e and small plaintexts x might be subject to attacks if no padding or weak padding is used. However, there is no known attack against small public exponents such as $e = 3$.

RSA has another undesirable property, namely that it is *malleable*. A crypto scheme is said to be malleable if the attacker Oscar is capable of transforming the ciphertext into another ciphertext which leads to a known transformation of the plaintext. Note that the attacker does not decrypt the ciphertext but is merely capable of manipulating the plaintext in a predictable manner. This is easily achieved in the case of RSA if the attacker replaces the ciphertext y by $s^e y$, where s is some integer. If the receiver decrypts the manipulated ciphertext, he computes:

$$(s^e y)^d \equiv s^{ed} x^{ed} \equiv s x \bmod n.$$

Even though Oscar is not able to decrypt the ciphertext, such targeted manipulations can still do harm. For instance, if x were an amount of money which is to be transferred or the value of a contract, by choosing $s = 2$ Oscar could exactly double the amount in a way that goes undetected by the receiver.

A possible solution to all these problems is the use of padding, which embeds a random structure into the plaintext before encryption and avoids the above mentioned problems. Modern techniques such as *Optimal Asymmetric Encryption Padding (OAEP)* for padding RSA messages are specified and standardized in Public Key Cryptography Standard #1 (PKCS #1).

Let M be the message to be padded, let k be the length of the modulus n in bytes, let $|H|$ be the length of the hash function output in bytes and let $|M|$ be the

length of the message in bytes. A hash function computes a message digest of fixed length (e.g., 160 or 256 bit) for every input. More about hash functions is found in Chap. 11. Furthermore, let L be an optional label associated with the message (otherwise, L is an empty string as default). According to the most recent version PKCS#1 (v2.1), padding a message within the RSA encryption scheme is done in the following way:

1. Generate a string PS of length $k - |M| - 2|H| - 2$ of zeroed bytes. The length of PS may be zero.
2. Concatenate $Hash(L)$, PS, a single byte with hexadecimal value `0x01`, and the message M to form a data block DB of length $k - |H| - 1$ bytes as

$$DB = Hash(L)||PS||\texttt{0x01}||M.$$

3. Generate a random byte string $seed$ of length $|H|$.
4. Let $dbMask = MGF(seed, k - |H| - 1)$, where MGF is the mask generation function. In practice, a hash function such as SHA-1 is often used as MFG.
5. Let $maskedDB = DB \oplus dbMask$.
6. Let $seedMask = MGF(maskedDB, |H|)$.
7. Let $maskedSeed = seed \oplus seedMask$.
8. Concatenate a single byte with hexadecimal value `0x00`, $maskedSeed$ and $maskedDB$ to form an encoded message EM of length k bytes as

$$EM = \texttt{0x00}||maskedSeed||maskedDB.$$

Figure 7.3 shows the structure of a padded message M.

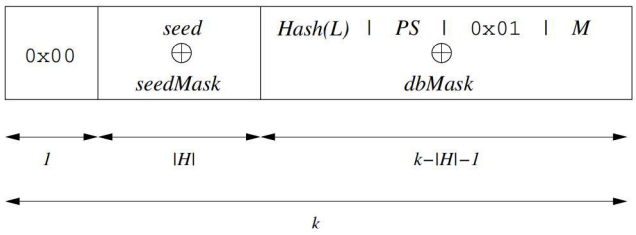

Fig. 7.3 RSA encryption of a message M with Optimal Asymmetric Encryption Padding (OAEP)

On the decryption side, the structure of the decrypted message has to be verified. For instance, if there is no byte with hexadecimal value `0x01` to separate PS from M, a decryption error occurred. In any case, returning a decryption error to the user (or a potential attacker!) should not reveal any information about the plaintext.

We could include types of attacks against RSA in our paper/presenation.

7.8 Attacks

There have been numerous attacks proposed against RSA since it was invented in 1977. None of the attacks are serious, and moreover, they typically exploit weaknesses in the way RSA is implemented or used rather than the RSA algorithm itself. There are three general attack families against RSA:

1. Protocol attacks
2. Mathematical attacks
3. Side-channel attacks

We comment on each of them in the following.

Protocol Attacks

Protocol attacks exploit weaknesses in the way RSA is being used. There have been several protocol attacks over the years. Among the better known ones are the attacks that exploit the malleability of RSA, which was introduced in the previous section. Many of them can be avoided by using padding. Modern security standards describe exactly how RSA should be used, and if one follows those guidelines, protocol attacks should not be possible.

Mathematical Attacks

The best mathematical cryptanalytical method we know is factoring the modulus. An attacker, Oscar, knows the modulus n, the public key e and the ciphertext y. His goal is to compute the private key d which has the property that $e \cdot d \equiv \mod \Phi(n)$. It seems that he could simply apply the extended Euclidean algorithm and compute d. However, he does not know the value of $\Phi(n)$. At this point factoring comes in: the best way to obtain this value is to decompose n into its primes p and q. If Oscar can do this, the attack succeeds in three steps:

$$\begin{aligned} \Phi(n) &= (p-1)(q-1) \\ d^{-1} &\equiv e \bmod \Phi(n) \\ x &\equiv y^d \bmod n. \end{aligned}$$

In order to prevent this attack, the modulus must be sufficiently large. This is the sole reason why moduli of 1024 or more bit are needed for a RSA. The proposal of the RSA scheme in 1977 sparked much interest in the old problem of integer factorization. In fact, the major progress that has been made in factorization in the last three decades would most likely not have happened if it weren't for RSA. Table 7.3 shows a summary of the RSA factoring records that have occurred since the beginning of the 1990s. These advances have been possible mainly due to improvements in factoring algorithms, and to a lesser extent due to improved computer technology.

Even though factoring has become easier than the RSA designers had assumed 30 years ago, factoring RSA moduli beyond a certain size still is out of reach.

Table 7.3 Summary of RSA factoring records since 1991

Decimal digits	Bit length	Date
100	330	April 1991
110	364	April 1992
120	397	June 1993
129	426	April 1994
140	463	February 1999
155	512	August 1999
200	664	May 2005

Of historical interest is the 129-digit modulus which was published in a column by Martin Gardner in *Scientific American* in 1997. It was estimated that the best factoring algorithms of that time would take 40 trillion ($4 \cdot 10^{13}$) years. However, factoring methods improved considerably, particularly during the 1980s and 1990s, and it took in fact less than 30 years.

Which exact length the RSA modulus should have is the topic of much discussion. Until recently, many RSA applications used a bit length of 1024 bits as default. Today it is believed that it might be possible to factor 1024-bit numbers within a period of about 10–15 years, and intelligence organizations might be capable of doing it possibly even earlier. Hence, it is recommended to choose RSA parameters in the range of 2048–4096 bits for long-term security.

Side-Channel Attacks

A third and entirely different family of attacks are side-channel attacks. They exploit information about the private key which is leaked through physical channels such as the power consumption or the timing behavior. In order to observe such channels, an attacker must typically have direct access to the RSA implementation, e.g., in a cell phone or a smart card. Even though side-channel attacks are a large and active field of research in modern cryptography and beyond the scope of this book, we show one particularly impressive such attack against RSA in the following.

Figure 7.4 shows the power trace of an RSA implementation on a microprocessor. More precisely, it shows the electric current drawn by the processor over time. Our goal is to extract the private key d which is used during the RSA decryption. We clearly see intervals of high activity between short periods of less activity. Since the main computational load of RSA is the squarings and multiplication during the exponentiation, we conclude that the high-activity intervals correspond to those two operations. If we look more closely at the power trace, we see that there are high activity intervals which are short and others which are longer. In fact, the longer ones appear to be about twice as long. This behavior is explained by the square-and-multiply algorithm. If an exponent bit has the value 0, only a squaring is per-

formed. If an exponent bit has the value 1, a squaring together with a multiplication is computed. But this timing behavior reveals immediately the key: A long period of activity corresponds to the bit value 1 of the secret key, and a short period to a key bit with value 0. As shown in the figure, by simply looking at the power trace we can identify the secret exponent. Thus we can learn the following 12 bits of the private key by looking at the trace:

operations:	*S*	*SM*	*SM*	*S*	*SM*	*S*	*S*	*SM*	*SM*	*SM*	*S*	*SM*
private key:	0	1	1	0	1	0	0	1	1	1	0	1

Obviously, in real-life we can also find all 1024 or 2048 bits of a full private key. During the short periods with low activity, the square-and-multiply algorithm scans and processes the exponent bits before it triggers the next squaring or squaring-and-multiplication sequence.

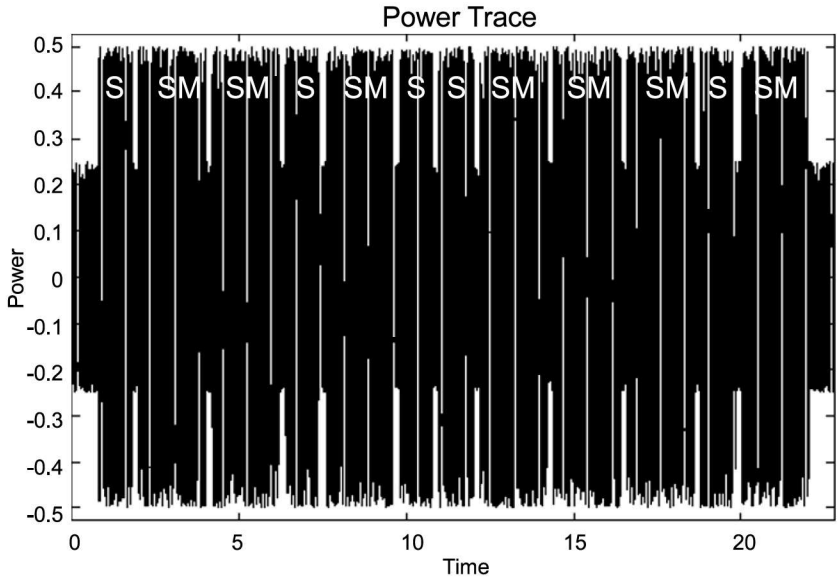

Fig. 7.4 The power trace of an RSA implementation

This specific attack is classified as simple power analysis or SPA. There are several countermeasures available to prevent the attack. A simple one is to execute a multiplication with dummy variables after a squaring that corresponds to an exponent bit 0. This results in a power profile (and a run time) which is independent of the exponent. However, countermeasures against more advanced side-channel attacks are not as straightforward.

7.9 Implementation in Software and Hardware

RSA is the prime example (almost literally) for a public-key algorithm that is very computationally intensive. Hence, the implementation of public-key algorithms is much more crucial than that of symmetric ciphers like 3DES and AES, which are significantly faster. In order to get an appreciation for the computational load, we develop a rough estimate for the number of integer multiplications needed for an RSA operation.

We assume a 2048-bit RSA modulus. For decryption we need on average 3072 squaring and multiplications, each of which involves 2048-bit operands. Let's assume a 32-bit CPU so that each operand is represented by $2048/32 = 64$ registers. A single long-number multiplication requires now $64^2 = 4096$ integer multiplications since we have to multiply every register of the first operand with every register of the second operand. In addition, we have to modulo reduce each of these multiplications. The best algorithms for doing this also require roughly $64^2 = 4096$ integer multiplications. Thus, in total, the CPU has to perform about $4096 + 4096 = 8192$ integer multiplications for a single long-number multiplication. Since we have 3072 of these, the number of integer multiplications for one decryption is:

$$\#(\text{32-bit mult}) = 3072 \times 8192 = 25,165,824$$

Of course, using a smaller modulus results in fewer operations, but given that integer multiplications are among the most costly operations on current CPUs, it is probably clear that the computational demand is quite impressive. Note that most other public key schemes have a similar complexity.

The extremely high computational demand of RSA was, in fact, a serious hindrance to its adoption in practice after it had been invented. Doing hundreds of thousands of integer multiplications was out of question with 1970s-style computers. The only option for RSA implementations with an acceptable run time was to realize RSA on special hardware chips until the mid- to late 1980s. Even the RSA inventors investigated hardware architecture in the early days of the algorithm. Since then much research has focused on ways to quickly perform modular integer arithmetic. Given the enormous capabilities of state-of-the-art VLSI chips, an RSA operation can today be done in the range of 100 μs on high-speed hardware.

Similarly, due to Moore's Law, RSA implementations in software have become possible since the late 1980s. Today, a typical decryption operation on a 2 GHz CPU takes around 10 ms for 2048-bit RSA. Even though this is sufficient for many PC applications, the throughput is about $100 \times 2048 = 204,800$ bit/s if one uses RSA for encryption of large amounts of data. This is quite slow compared to the speed of many of today's networks. For this reason RSA and other public-key algorithms are not used for bulk data encryption. Rather, symmetric algorithms are used that are often faster by a factor of 1000 or so.

7.10 Discussion and Further Reading

RSA and Variants The RSA cryptosystem is widely used in practice and is well standardized in bodies such as PKCS#1 [149]. Over the years several variants have been proposed. One generalization is to use a modulus which is composed of more than two primes. Also proposed have been multipower moduli of the form $n = p^2 q$ [162] as well as multifactor ones where $n = pqr$ [45]. In both cases speed-ups by a factor of approximately 2–3 are possible.

There are also several other crypto schemes which are based on the integer factorization problem. A prominent one is the Rabin scheme [140]. In contrast to RSA, it can be shown that the Rabin scheme is equivalent to factoring. Thus, it is said that the cryptosystem is *provable secure*. Other schemes which rely on the hardness of integer factorization include the probabilistic encryption scheme by Blum–Goldwasser [28] and the Blum Blum Shub pseudo-random number generator [27]. The *Handbook of Applied Cryptography* [120] describes all the schemes mentioned in a coherent form.

Implementation The actual performance of an RSA implementation heavily depends on the efficiency of the arithmetic used. Generally speaking, speed-ups are possible at two levels. On the higher level, improvements of the square-and-multiply algorithm are an option. One of the fastest methods is the sliding window exponentiation which gives an improvement of about 25% over the square-and-multiply algorithm. A good compilation of exponentiation methods is given in [120, Chap. 14]. On the lower layer, modular multiplication and squaring with long numbers can be improved. One set of techniques deals with efficient algorithms for modular reduction. In practice, Montgomery reduction is the most popular choice; see [41] for a good treatment of software techniques and [72] for hardware. Several alternatives to the Montgomery method have also been proposed over the years [123]; [120, Chap. 14]. Another angle to accelerate long number arithmetic is to apply fast multiplication methods. Spectral techniques such as the fast Fourier transform (FFT) are usually not applicable because the operands are still too short, but methods such as the Karatsuba algorithm [99] are very useful. Reference [17] gives a comprehensive but fairly mathematical treatment of the area of multiplication algorithms, and [172] describes the Karatsuba method from a practical viewpoint.

Attacks Breaking RSA analytically has been a subject of intense investigation for the last 30 years. Especially during the 1980s, major progress in factorization algorithms was made, which was not in small part motivated by RSA. There have been numerous other attempts to mathematically break RSA, including attacks against short private exponents. A good survey is given in [32]. More recently, proposals have been made to build special computers whose sole purpose is to break RSA. Proposals include an optoelectronic factoring machine [151] and several other architectures based on conventional semiconductor technology [152, 79].

Side channel attacks have been systematically studied in academia and industry since the mid- to late 1990s. RSA, as well as most other symmetric and asymmetric schemes, are vulnerable against differential power analysis (DPA), which is more

powerful than the simple power analysis (SPA) shown in this section. On the other hand, numerous countermeasures against DPA are known. Good references are *The Side Channel Cryptanalysis Lounge* [70] and the excellent book on DPA [113]. Related implementation-based attacks are *fault injection attacks* and *timing attacks*. It is important to stress that a cryptosystem can be mathematically very strong but still be vulnerable to side-channel attacks.

7.11 Lessons Learned

- RSA is the most widely used public-key cryptosystem. In the future, elliptic curve cryptosystems will probably catch up in popularity.
- RSA is mainly used for key transport (i.e., encryption of keys) and digital signatures.
- The public key e can be a short integer. The private key d needs to have the full length of the modulus. Hence, encryption can be significantly faster than decryption.
- RSA relies on the integer factorization problem. Currently, 1024-bit (about 310 decimal digits) numbers cannot be factored. Progress in factorization algorithms and factorization hardware is hard to predict. It is advisable to use RSA with a 2048-bit modulus if one needs reasonable long-term security, especially with respect to extremely well funded attackers.
- "Schoolbook RSA" allows several attacks, and in practice RSA should be used together with padding.

Problems

7.1. Let the two primes $p = 41$ and $q = 17$ be given as set-up parameters for RSA.

1. Which of the parameters $e_1 = 32, e_2 = 49$ is a valid RSA exponent? Justify your choice.
2. Compute the corresponding private key $K_{pr} = (p, q, d)$. Use the extended Euclidean algorithm for the inversion and point out every calculation step.

7.2. Computing modular exponentiation efficiently is inevitable for the practicability of RSA. Compute the following exponentiations x^e mod m applying the square-and-multiply algorithm:

1. $x = 2$, $e = 79$, $m = 101$
2. $x = 3$, $e = 197$, $m = 101$

After every iteration step, show the exponent of the intermediate result in binary notation.

7.3. Encrypt and decrypt by means of the RSA algorithm with the following system parameters:

1. $p = 3$, $q = 11$, $d = 7$, $x = 5$
2. $p = 5$, $q = 11$, $e = 3$, $x = 9$

Only use a pocket calculator at this stage.

7.4. One major drawback of public-key algorithms is that they are relatively slow. In Sect. 7.5.1 we learned that an acceleration technique is to use short exponents e. Now we study short exponents in this problem in more detail.

1. Assume that in an implementation of the RSA cryptosystem one modular squaring takes 75% of the time of a modular multiplication. How much quicker is one encryption on average if instead of a 2048-bit public key the short exponent $e = 2^{16} + 1$ is used? Assume that the square-and-multiply algorithm is being used in both cases.
2. Most short exponents are of the form $e = 2^n + 1$. Would it be advantageous to use exponents of the form $2^n - 1$? Justify your answer.
3. Compute the exponentiation x^e mod 29 of $x = 5$ with both variants of e from above for $n = 4$. Use the square-and-multiply algorithm and show each step of your computation.

7.5. In practice the short exponents $e = 3$, 17 and $2^{16} + 1$ are widely used.

1. Why can't we use these three short exponents as values for the exponent d in applications where we want to accelerate decryption?
2. Suggest a minimum bit length for the exponent d and explain your answer.

7.6. Verify the RSA with CRT example in the chapter by computing $y^d = 15^{103}$ mod 143 using the square-and-multiply algorithm.

7.7. An RSA encryption scheme has the set-up parameters $p = 31$ and $q = 37$. The public key is $e = 17$.

1. Decrypt the ciphertext $y = 2$ using the CRT.
2. Verify your result by encrypting the plaintext without using the CRT.

7.8. Popular RSA modulus sizes are 1024, 2048, 3072 and 4092 bit.

1. How many random odd integers do we have to test on average until we expect to find one that is a prime?
2. Derive a simple formula for any arbitrary RSA modulus size.

7.9. One of the most attractive applications of public-key algorithms is the establishment of a secure session key for a private-key algorithm such as AES over an insecure channel.

Assume Bob has a pair of public/private keys for the RSA cryptosystem. Develop a simple protocol using RSA which allows the two parties Alice and Bob to agree on a shared secret key. Who determines the key in this protocol, Alice, Bob, or both?

7.10. In practice, it is sometimes desirable that both communication parties influence the selection of the session key. For instance, this prevents the other party from choosing a key which is a *weak key* for a symmetric algorithm. Many block ciphers such as DES and IDEA have weak keys. Messages encrypted with weak keys can be recovered relatively easily from the ciphertext.

Develop a protocol similar to the one above in which both parties influence the key. Assume that both Alice and Bob have a pair of public/private keys for the RSA cryptosystem. Please note that there are several valid approaches to this problem. Show just one.

7.11. In this exercise, you are asked to attack an RSA encrypted message. Imagine being the attacker: You obtain the ciphertext $y = 1141$ by eavesdropping on a certain connection. The public key is $k_{pub} = (n, e) = (2623, 2111)$.

1. Consider the encryption formula. All variables except the plaintext x are known. Why can't you simply solve the equation for x?
2. In order to determine the private key d, you have to calculate $d \equiv e^{-1} \bmod \Phi(n)$. There is an efficient expression for calculating $\Phi(n)$. Can we use this formula here?
3. Calculate the plaintext x by computing the private key d through factoring $n = p \cdot q$. Does this approach remain suitable for numbers with a length of 1024 bit or more?

7.12. We now show how an attack with chosen ciphertext can be used to break an RSA encryption.

1. Show that the *multiplicative property* holds for RSA, i.e., show that the product of two ciphertexts is equal to the encryption of the product of the two respective plaintexts.

2. This property can under certain circumstances lead to an attack. Assume that Bob first receives an encrypted message y_1 from Alice which Oscar obtains by eavesdropping. At a later point in time, we assume that Oscar can send an innocent looking ciphertext y_2 to Bob, and that Oscar can obtain the decryption of y_2. In practice this could, for instance, happen if Oscar manages to hack into Bob's system such that he can get access to decrypted plaintext for a limited period of time.

7.13. In this exercise, we illustrate the problem of using nonprobabilistic cryptosystems, such as schoolbook RSA, imprudently. Nonprobabilistic means that the same sequence of plaintext letters maps to the same ciphertext. This allows traffic analysis (i.e., to draw some conclusion about the cleartext by merely observing the ciphertext) and in some cases even to the total break of the cryptoystem. The latter holds especially if the number of possible plaintexts is small. Suppose the following situation:

Alice wants to send a message to Bob encrypted with his public key pair (n,e). Therefore, she decides to use the ASCII table to assign a number to each character ($Space \rightarrow 32$, ! $\rightarrow 33$, ..., $A \rightarrow 65$, $B \rightarrow 66$, ..., $\sim\!\!\rightarrow 126$) and to encrypt them separately.

1. Oscar eavesdrops on the transferred ciphertext. Describe how he can successfully decrypt the message by exploiting the nonprobabilistic property of RSA.
2. Bob's RSA public key is $(n,e) = (3763, 11)$. Decrypt the ciphertext

$$y = 2514, 1125, 333, 3696, 2514, 2929, 3368, 2514$$

   with the attack proposed in 1. For simplification, assume that Alice only chose capital letters A–Z during the encryption.
3. Is the attack still possible if we use the OAEP padding? Exactly explain your answer.

7.14. The modulus of RSA has been enlarged over the years in order to thwart improved attacks. As one would assume, public-key algorithms become slower as the modulus length increases. We study the relation between modulus length and performance in this problem. The performance of RSA, and of almost any other public-key algorithm, is dependent on how fast modulo exponentiation with large numbers can be performed.

1. Assume that one modulo multiplication or squaring with k-bit numbers takes $c \cdot k^2$ clock cycles, where c is a constant. How much slower is RSA encryption/decryption with 1024 bits compared to RSA with 512 bits on average? Only consider the encryption/decryption itself with an exponent of full length and the square-and-multiply algorithm.
2. In practice, the Karatsuba algorithm, which has an asymptotical complexity that is proportional to $k^{\log_2 3}$, is often used for long number multiplication in cryptography. Assume that this more advanced technique requires $c' \cdot k^{\log_2 3} = c' \cdot k^{1.585}$ clock cycles for multiplication or squaring where c' is a constant. What is the

ratio between RSA encryption with 1024 bit and RSA with 512 bit if the Karatsuba algorithm is used in both cases? Again, assume that full-length exponents are being used.

7.15. (Advanced problem!) There are ways to improve the square-and-multiply algorithm, that is, to reduce the number of operations required. Although the number of squarings is fixed, the number of multiplications can be reduced. Your task is to come up with a modified version of the square-and-multiply algorithm which requires fewer multiplications. Give a detailed description of how the new algorithm works and what the complexity is (number of operations).
Hint: Try to develop a generalization of the square-and-multiply algorithm which processes more than one bit at a time. The basic idea is to handle k (e.g., $k = 3$) exponent bit per iteration rather than one bit in the original square-and-multiply algorithm.

7.16. Let us now investigate side-channel attacks against RSA. In a simple implementation of RSA without any countermeasures against side-channel leakage, the analysis of the current consumption of the microcontroller in the decryption part directly yields the private exponent. Figure 7.5 shows the power consumption of an implementation of the square-and-multiply algorithm. If the microcontroller computes a squaring or a multiplication, the power consumption increases. Due to the small intervals in between the loops, every iteration can be identified. Furthermore, for each round we can identify whether a single squaring (short duration) or a squaring followed by a multiplication (long duration) is being computed.

1. Identify the respective rounds in the figure and mark these with S for squaring or SM for squaring and multiplication.
2. Assume the square-and-multiply algorithm has been implemented such that the exponent is being scanned from left to right. Furthermore, assume that the starting values have been initialized. What is the private exponent d?
3. This key belongs to the RSA setup with the primes $p = 67$ and $q = 103$ and $e = 257$. Verify your result. (Note that in practice an attacker wouldn't know the values of p and q.)

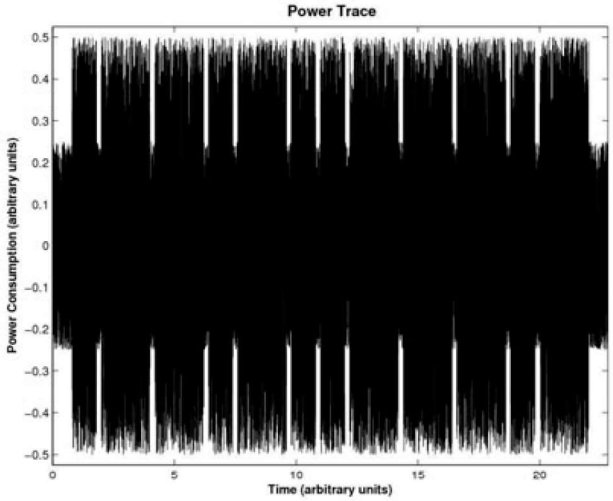

Fig. 7.5 Power consumption of an RSA decryption