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:

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

A straightforward way of exponentiation looks like this:

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

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:

$$x^{26} = x^{11010_2} = x^{(h_4h_3h_2h_1h_0)_2}.$$

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       $r = r^2 \bmod n$
          IF $h_i = 1$
1.2           $r = r \cdot x \bmod n$
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., $\log_2 H \;= t+1$. 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 $\overline{MUL}$:

$$\#SQ = t$$
$$\#\overline{MUL} = 0.^{5}\,t$$

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.