6.4 Implementation issues

As you have seen, RSA relies on the factoring problem. This means that we need to use a really large number n or otherwise the enemy will simply factor the number. A typical RSA modulus could have a length of, say, 1024 bits. With such large numbers, we need to examine how the different arithmetic operations are done.

Let us examine the implementation of the RSA operation, i.e., computing

$$M^e \bmod n.$$

Assuming that calculations are in $\mathbb{Z}_n$, so we need $L = \lceil \log_2 n \rceil$ bits to store a value in $\mathbb{Z}_n$. First note that it is in general a good idea to reduce modulo n as soon as one receives an intermediate value outside the interval $[0, n-1]$.

Example 6.4. *Calculating* $2728^3 \bmod 7849$ *is done by first computing* $2728^2 = 7441984$*, then computing* $7441984 \bmod 7849 = 1132$*. Continuing,* $1132 * 2728 = 3088096$ *and finally* $3088096 \bmod 7849 = 3439$*. So* $2728^3 \bmod 7849 = 3439$*.*

Next, we must find out how we can do exponentiation in a fast way, i.e., computing $M^e \bmod n$ when e is a large number. A trivial approach would be to multiply M with itself $e-1$ times, and reducing the intermediate values modulus n in each step. But in a real RSA system e could be of size 2^{1024}, requiring the same number of multiplications. So this is not possible.

Fortunately, there is a much more efficient way of computing $M^e \bmod n$, known as the *square and multiply algorithm*. In this algorithm we first rewrite e as a sum of powers of 2, i.e.,

$$e = e_0 + e_1 \cdot 2 + e_2 \cdot 2^2 + \cdots + e_{L-1} 2^{L-1},$$

where $e_i \in \{0, 1\}, 0 \leq i \leq L-1$. We then perform $L-1$ squarings of M, computing the values

$$M^2, M^4 = (M^2)^2, M^8 = (M^4)^2, \ldots,$$

in $\mathbb{Z}_n$. Finally we perform at most $L-1$ multiplications by computing M^e as

$$M^e = M^{e_0} \cdot (M^2)^{e_1} \cdot (M^4)^{e_2} \cdots (M^{2^{L-1}})^{e_{L-1}}.$$

Example 6.5. *Compute* 2728^{25} *in* $\mathbb{Z}_n$ *using the square and multiply algorithm.*

We rewrite 25 *as* $25 = 16 + 8 + 1$ *and get*

$$2728^{25} = 2728 \cdot 2728^8 \cdot 2728^{16}.$$

Performing a number of squarings we get

$$2728^2 = 1132, 2728^4 = (1132)^2 = 2037, 2728^8 = (2037)^2 = 5097, 2728^{16} = (5097)^2 = 7068.$$

Performing the multiplications we get

$$2728^{25} = 2728 \cdot 5097 \cdot 7068 = 2401.$$

There are many different versions of the general theme of the square and multiply algorithm.

6.5 Primality testing

WEhen setting up an RSA cryptosystem it is necessary to generate two large prime numbers p and q. The way to do this in practice is to randomly select a candidate number and then test whether it is a prime number. The question is then how to check whether a given number m is a prime or not.

A naive approach would be to test by trial division, i.e., we check if $x|m$ for all integers x, where $2 \leq x \leq \sqrt{m}$. The problem is that if m is a 1024 bit number then $\sqrt{m}$ is of size 512 bits and 2^{512} tests is not possible.

The solution to the problem of efficiently finding prime numbers is the *probabilistic* algorithms for testing primality. Let us introduce the concept by looking at the Fermat test. Recall that Fermat's Little Theorem states that

$$a^{m-1} = 1 \bmod m,$$

if m is a prime and a such that $1 \leq a \leq m-1$. When m is not a prime we cannot really know what we will receive when computing a^{m-1}. But if $a^{m-1} \neq 1 \bmod m$ we know for certain that m is not a prime number.

The approach should now be quite clear. Select a random a and compute $A = a^{m-1}$. If $A \neq 1$ we know that m is composite, whereas if $A = 1$ we cannot say for certain if m is prime or not. We then proceed and select a new random a and repeat the procedure. If we have repeated this a large number of times and never received a proof of compositeness, we finally end with the conclusion that m is a prime number. There is of course a small probability that we came to the wrong conclusion, but if we repeated the test enough we would argue that this probability is very small. For the Fermat test it can be shown that if m is composite, then we get $a^{m-1} = 1$ with probability less than $1/2$. The error probability after repeating the test k time would then be less than $1/2^k$.

A composite m such that $a^{m-1} = 1 \bmod m$ is said to be a pseudo-prime to the base a. If a composite m is such that it is a pseudo-prime for every base a with $\gcd(a, m) = 1$, the the number is called a Carmichael number. The smallest Carmichael number is 561. For a Carmichael number, the Fermat test will fail for any choice of bases. Carmichael numbers are rare but not extremely rare. For this reason the Fermat test is usually avoided. Instead one uses a slightly more advanced test. One such test is the Miller-Rabin test.

- Randomly choose a base a.
- Write $m - 1 = 2^b \cdot q$, where q is odd.
- If $a^q = 1 \bmod m$ or $a^{2^c \cdot q} = -1 \bmod m$ for any $c < b$ then return "m probably prime" and go back and choose a new base, otherwise return "m definitely not prime".

One can prove that $P(\text{"}m \text{ probably prime"}|m\text{not prime} < 1/4$.

6.6 Factoring

As factoring the publicly known RSA modulus n would break RSA it is interesting to know something more on the difficulty of factoring. We have a number of different factoring algorithms. Some of them apply to numbers of special form whereas others are general in the sence that they can factor any number.

We start by giving one interesting algorithm of a special kind, *Pollard's* $(p-1)$*-method.*

Let $n = pq$ be the number to be factored. Let $p-1$ factor as

$$p - 1 = q_1 q_2 \cdots q_k,$$

where each q_i is a prime power. The condition for the algorithm to work is that each unknown $q_i < B$, where B is a predetermined "bound". The Pollard's $(p-1)$-method then computes the value $a = 2^{B!} \bmod n$ and computes the unknown prime p as $p = \gcd(a-1, n)$.

The explanation behind this simple algorithm is as follows. Assuming that every unknown prime power $q_i < B$, we have that

$$(p-1)|B!.$$

We compute $a = 2^{B!} \bmod n$ in the algorithm. Now let $a' = 2^{B!} \bmod p$. Since $p|n$ we must have $a = a' \bmod p$. Fermat's little theorem states that

$$2^{p-1} = 1 \bmod p$$

and since $(p-1)|B!$ we also have $a' = 1 \bmod p$ leading to $a = 1 \bmod p$. So $p|(a-1)$ and since $p|n$ we have $p = \gcd(a-1, n)$.

This simple algorithm gives us a very important conclusion. If p or q are chosen such that $p-1$ or $q-1$ factors in only small prime powers there is a very efficient factoring algorithm. The primes p and q must then be chosen such that $p-1$ and $q-1$ each contains a large prime in their factorisation. This is not a problem, however. A usual approach is to generate a random bprime p_1 and then test whether $p = 2p_1 + 1$ is a prime number. If so, we choose p.

6.6.1 Modern factoring algorithms

See project 1.

For more information on the current status of factoring algorithms, read on the web.

6.7 Digital signatures

6.8 Hash functions

6.9 Diffie-Hellman Key Exchange and discrete logatithms