Why is algebra important? - Public-key encryption

Definition

A public-key encryption scheme is a set of encryption transformations $\{E_e : e \in \mathcal{K}\}$ and a set of decryption transformations $\{D_d : d \in \mathcal{K}\}$. For each $e \in \mathcal{K}$ there is a corresponding $d \in \mathcal{K}$ such that $D_d(E_e(M)) = M$, $\forall M$. Furthermore, after choosing such a pair (e, d), the *public key* e (or the public parameter) is made public, while the associated *secret key* d is kept secret. For the scheme to be secure, it must be computationally infeasible to compute d as well as computing $E_e^{-1}(C)$, knowing the public value e.

The RSA public-key encryption scheme

Definition

The *RSA public-key encryption scheme* works as follows. Let $n = pq$, where p and q are two large primes. Let $\mathcal{M} = \mathcal{C} = \mathbb{Z}_n$. Pick a number e relatively prime to $\phi(n)$ and calculate a number d such that $ed = 1 \bmod \phi(n)$. The public key is the two numbers (n, e) and the (public) encryption transformation $E(M)$ is

$$E(M) = M^e \bmod n.$$

The secret key is the number d (as well as p, q and $\phi(n)$) and the secret decryption transformation $D(C)$ is

$$D(C) = C^d \bmod n.$$

Verify that decrypting a ciphertext returns the encrypted plaintext.

$$D(C) = C^d = (M^e)^d = M^{ed} \bmod n.$$

Now we note that $ed = 1 \bmod \phi(n)$, which means that we can write

$$ed = 1 + t \cdot \phi(n),$$

for some integer t. S we can continue

$$D(C) = M^{ed} = M^{(1+t\cdot\phi(n))} = M \cdot M^{t\cdot\phi(n)} \bmod n.$$

From Euler's formula we know that $x^{\phi(n)} = 1$ for any $x \in \mathbb{Z}_n^*$. So assuming that M is invertible we have

$$D(C) = M \cdot M^{t\cdot\phi(n)} = M \cdot 1 \bmod n.$$

Example

Let $p = 47$ and $q = 167$ be two primes. Their product is $n = pq = 7849$.

Compute $\phi(n) = (p - 1)(q - 1) = 7636$. Next, we must choose a value e such that $\gcd(e, \phi(n)) = 1$. We can check that $e = 25$ is such a value.

When $\gcd(e, \phi(n)) = 1$ we know that $e = 25$ has a multiplicative inverse in $\mathbb{Z}_{\phi(n)}$. We use Euclidean algorithm and Bezout's lemma to find the inverse $d = 2749$, i.e.,

$$e \cdot d = 25 \cdot 2749 = 1 \bmod 7636.$$

We publish our public key $(n, e) = (7849, 25)$.

Example

Anyone with access to our public key can now send us an encrypted message $M \in \mathbb{Z}_{7849}$, by calculating

$$C = M^e \bmod n.$$

Alice sends message $M = 2728$. She then computes

$$C = 2728^{25} = 2401 \bmod 7849.$$

When Bob receives the ciphertext C, he computes

$$M = C^d = 2401^{2749} \bmod n$$

and, as by magic, $M = 2401^{2749} = 2728 \bmod n$ so he recover the correct message.

Security of the RSA cryptosystem

If we can factor n then we can compute $\phi(n)$ and d.

RSA relies on the *factoring problem*.

This does not mean that breaking RSA is equivalent to solving a factorization problem. It is not known whether RSA can be broken without factoring n.

PROJECT 1: The factoring problem

Let p and q be two prime numbers and let $N = pq$. Our factoring problem is given as follows.

Problem: Given N, find the primes p and q such that $N = pq$.

Trial division. N must be divisible by a number of size at most $\lfloor \sqrt{N} \rfloor$. So we just test whether

$$N \bmod p = 0,$$

for all $p = 2, 3, \ldots, \lfloor \sqrt{N} \rfloor$.

- The complexity of trial division is exponential ($< N^{\beta} \cdot C$ for some constants β and C),
- Modern factoring algorithms has a complexity that lies somewhere between polynomial and exponential time, subexponential time.
- Two well known methods are called *Quadratic Sieve* and *Number field Sieve*.

- The Quadratic Sieve algorithm was the fastest factoring algorithm up to early 1990s.
- When the target numbers grew larger, the Number field Sieve started to outperform the Quadratic Sieve, due to a better asymptotic performance.
- Recently, we have seen several cases of 512 bit numbers (150 decimal digit numbers) being factored through a large computational effort.
- RSA with a 512 bit composite number is not secure enough. The minimum recommended length is currently 1024 bits.

Your task

Exercise 2: Write a program that implements a simple version of the *Quadratic Sieve algorithm*, described in the sequel. Use this program to factor the 25 digit number you have been given from the assistant.

A very basic trick

If we in some way can find (or produce) two different numbers x and y such that

$$x^2 = y^2 \bmod N$$

then we have

$$x^2 - y^2 = (x - y)(x + y) = 0 \bmod N,$$

and this means that $(x - y)(x + y) = K \cdot p \cdot q$ for some integer K.

We can separate in two possible cases,

- either p divides $(x - y)$ and q divides $(x + y)$, or vice versa;
- or both p and q divides $(x - y)$ and none of them divides $(x + y)$, or vice versa.

We calculate

$$d = \gcd(x - y, N).$$

In the first case we will then get $d = p$ or $d = q$ whereas in the second case we get $d = N$ or $d = 1$. If the first case appear we can factor N, but if the second case appears we can not.

How to find x, y such that $x^2 = y^2 \bmod N$?

- Define a *factorbase* F to be the set of prime numbers less than a certain bound B, called the *smoothness bound*. The number of primes in F is denoted $|F|$.
  Example: If $B = 12$, then $F = \{2, 3, 5, 7, 11\}$.
- Define a number x to be B-*smooth* if it can be factored over the factorbase F, i.e., it can be written as a product of primes all smaller than B.

Example: $N = 264$ is 12-smooth since $N = 264 = 2^3 \cdot 3 \cdot 11$.

Simplified Quadratic sieve

- Generate many numbers x_i, $i = 1, 2, \ldots$, such that if $x_i^2 = y_i \bmod N$ then y_i is a B-smooth number (y_i factors over F).
- We could select a random number r and test whether $r^2 \bmod N$ is B-smooth. We can increase our chances if we select the numbers r of a special form.

We select our numbers r to test as

$$r = \lfloor \sqrt{k \cdot N} \rfloor + j, \tag{1}$$

for $j = 1, 2, \ldots$ and $k = 1, 2, \ldots$.

Note that

$$r^2 = (\lfloor \sqrt{k \cdot N} \rfloor + j)^2 = (\lfloor \sqrt{k \cdot N} \rfloor)^2 + 2 \cdot (\lfloor \sqrt{k \cdot N} \rfloor) \cdot j + j^2,$$

which means that $r^2 \bmod N$ is a number of the same order as $\sqrt{N}$.

The probability that such a number will factor over F is much larger than for a randomly selected r.

Example: If $N = 77$ then $r = \lfloor \sqrt{1 \cdot N} \rfloor + 1 = 8 + 1 = 9$ and $r^2 \bmod 77 = 4$ is 12-smooth.

Second step

After this first step, we have a set of numbers x_i, $i = 1, 2, \ldots, L$, such that if $x_i^2 = y_i \bmod N$ then y_i factors over F.

Now we are going to combine them to build two numbers x and y such that $x^2 = y^2 \bmod N$.

If we assume that $L > |F|$, say $L = |F| + 5$, then there exists a suitable selection of x_i's such that

$$x_{i_1}^2 \cdot x_{i_2}^2 \cdot \ldots x_{i_{L'}}^2 = y_{i_1} \cdot y_{i_2} \cdot \ldots y_{i_{L'}} \bmod N,$$

and $y_{i_1} \cdot y_{i_2} \cdot \ldots y_{i_{L'}} = Y^2$ for some Y.

Why? Gaussian elimination.

Let the factor base F be $F = \{p_1, p_2, \ldots, p_{|F|}\}$.

$$\begin{array}{rcl} x_1^2 & = & p_1^{e_{11}} \cdot p_2^{e_{12}} \cdot \ldots \cdot p_{|F|}^{e_{1|F|}} \\ x_2^2 & = & p_1^{e_{21}} \cdot p_2^{e_{22}} \cdot \ldots \cdot p_{|F|}^{e_{2|F|}} \\ \vdots & & \vdots \\ x_L^2 & = & p_1^{e_{L1}} \cdot p_2^{e_{L2}} \cdot \ldots \cdot p_{|F|}^{e_{L|F|}}, \end{array}$$

Our objective is to obtain an equation where the right hand side is of the form y^2 for some y.

This is true if all exponents involved e_{ij}, $j = 1, \ldots |F|$, are *even*. If so, the right hand side

$$p_1^{e_{i1}} \cdot p_2^{e_{i2}} \cdot \ldots \cdot p_{|F|}^{e_{i|F|}} = \left(p_1^{e_{i1}/2} \cdot p_2^{e_{i2}/2} \cdot \ldots \cdot p_{|F|}^{e_{i|F|}/2}\right)^2$$

and this gives $y = p_1^{e_{i1}/2} \cdot p_2^{e_{i2}/2} \cdot \ldots \cdot p_{|F|}^{e_{i|F|}/2}$.

The remaining question is how to get the right hand side of one equation to have only even exponents.

Start by selecting an equation for which the exponent for p_1 (e_{i1}) is odd, say

$$x_1^2 = p_1^{e_{11}} \cdot p_2^{e_{12}} \cdot \ldots \cdot p_{|F|}^{e_{1|F|}}.$$

Now go through all other equations. For each equation for which the exponent for p_1 (e_{i1}) is odd, say

$$x_i^2 = p_1^{e_{i1}} \cdot p_2^{e_{i2}} \cdot \ldots \cdot p_{|F|}^{e_{i|F|}},$$

we replace this equation by

$$(x_1 \cdot x_i)^2 = p_1^{e_{11}+e_{i1}} \cdot p_2^{e_{12}+e_{i2}} \cdot \ldots \cdot p_{|F|}^{e_{1|F|+e_{i|F|}}}.$$

After going through all the exponents for all the primes, we should, with $L = |F| + 5$, end up with 5 equations with all even exponents on the right hand side. We then have several pairs x, y such that $x^2 = y^2 \bmod N$, and this should give us a high probability of finding the factors of N.

the above procedure is equivalent to solving of a system of binary linear equations, where even exponent corresponds to the value 0 and odd exponent corresponds to the value 1

An important note is the following. Let $r_1^2 = b_1$, where b_1 factors over F. Then the number $r = 2 \cdot r_1$ by $r^2 = (r_1 \cdot 2)^2 = 4 \cdot r_1^2 = b$ also produces a relation $r^2 = b$ where b factors over F. Note that for these two relations the binary row in the matrix above will be the same.

Therefore, before you insert a row in the binary matrix M above, make sure that it is not already present.

Final words

The size of the factor base should for best performance in your case be as large as possible. Select a size that your computer can handle, for example $L = 1000$.

You need to be able to do arithmethic with large numbers (30 digit numbers). The standard integer representation in a programming language is usually not sufficient. You need to use some library supporting large integers (or implement it yourself!).

The second problem you might run into is memory management. Note that the second step of the algorithm requires a substantial amount of memory for large factor bases.