Public-key cryptography RSA

NGUYEN Tuong Lan
LIU Yi
Master Informatique
University Lyon 1

Objective:

Our goal in the study is to understand the algorithm RSA, some existence attacks and implement in Java.

Liu's part: I look for understanding and master the algorithm RSA for cryptography by learning:

- All the theorems involved.
- The computations occurred in the RSA algorithm, like congruence relation, etc.
- The algorithms used in RSA like modular multiplicative inverse, modular exponentiation, etc.
- The methods against attack. (Padding Scheme and Padding Method)
- The example of the application of RSA (Folder RSA)

And in the end, I could code my-self a short program to apply the algorithm : generation the keys, encryption, and decryption.

For my part, I firstly tried to show the mathematical strength of algorithm RSA by analyzing the different methods of factoring a big integer(Fermat, Euler and Pollard's Rho). I also provide the comparison of two methods (Fermat and Pollard's Rho) by implementing in Java code these methods. (Folder Factoring).

Secondly, I studied one of Side-Channel Attack: Timing Attack, implemented a simple version in Java code (Folder Timing Attack)

Finally, knowledge of RSA using CRT will finish my part. (I also provide an implementation in Java code) (Folder RSA-CRT).

(Remark: all the results tested in Java files is executed in Dell Inspiron D6000 processor 1.6 Ghz chips Intel Pentium M)

A. RSA

1. History

In cryptography, **RSA** is an algorithm for public-key cryptography. The algorithm was publicly described in 1977 by Ron Rivest, Adi Shamir, and Leonard Adleman at MIT; the letters **RSA** are the initials of their surnames, listed in the same order as on the paper. It is the first algorithm known to be suitable for signing as well as encryption, and one of the first great advances in public key cryptography. RSA is widely used in electronic commerce protocols, and is believed to be secure given sufficiently long keys and the use of up-to-date implementations. [WIKIa]

The RSA algorithm involves three steps: key generation, encryption and decryption

2. Key generation

RSA involves a public key and a private key. The public key can be known to everyone and is used for encrypting messages. Messages encrypted with the public key can only be decrypted using the private key. The keys for the RSA algorithm are generated the following way:

- Choose two distinct prime numbers *p* and *q*
- Compute $n = pq$
  - *n* is used as the modulus for both the public and private keys

Compute the totient: $\varphi(n) = (p-1)(q-1)$. $\varphi(n)$ is also called the Euler's totient function, it is the number of positive integers less than or equal to *n* that are co prime to *n*. *p* and *q* are co-prime because they are all primes. Therefore, $\varphi(pq) = \varphi(p)\varphi(q) = (p-1)(q-1)$

- Choose an integer *e* such that $1 < e < \varphi(n)$, and *e* and $\varphi(n)$ share no factors other than 1 (i.e. *e* and $\varphi(n)$ are co-prime)
  - *e* is released as the public key exponent
- Determine *d* (using modular arithmetic) which satisfies the congruence relation $de \equiv 1 \pmod{\varphi(n)}$;
  - Stated differently, $ed - 1$ can be evenly divided by the totient $(p-1)(q-1)$
  - This relation is also called the modular multiplicative inverse of e modulo $\varphi(n)$ which is d : $e^{-1} \equiv d \mod \varphi(n)$
  - This is often computed using the **Extended Euclidean Algorithm**
  - *d* is kept as the private key exponent

The **extended Euclidean algorithm** [WIKIb]is an extension to the Euclidean algorithm for finding the greatest common divisor (GCD) of integers *a* and *b*: it also finds the integers *x* and *y* in Bézout's identity :

$$ax + by = \gcd(a,b)$$ (Typically either x or y is negative).

The extended Euclidean algorithm is particularly useful when *a* and *b* are co-prime, since *x* is the modular multiplicative inverse of *a* modulo *b*. This is exactly our case because : $de \equiv 1 \pmod{\varphi(n)} \Leftrightarrow de\text{-}1 = k\varphi(n) \Leftrightarrow de - k\varphi(n) = 1$ by the definition of congruence. It verifies also that e and $\varphi(n)$ are co-primes.

With *d* and $\varphi(n)$ given, e the inverse, and k an integer multiple that will be ignored. This is the exact form of equation that the extended Euclidean algorithm solves; the only difference being that $gcd(d, \varphi(n)) = 1$ is predetermined instead of discovered.

There exist several algorithms for resolving this equation:

- The recursive method :

120	x	+	23	y	=	1
(5*23+5)	x	+	23	y	=	1
23	(5x+y)	+	5	x	=	1
..	..	..	..	..	..	..
1	a	+	0	b	=	1

The algorithm of the recursive method is the following:

- If a is divisible by b, the algorithm ends and return the trivial solution x = 0, y = 1.
- Otherwise, repeat the algorithm with b and a modulus b, storing the solution as x' and y'.
- Then, the solution to the current equation is x = y', and y = x' minus y' times totient of a divided by b

Which can be directly translated to this pseudocode:

```
function extended_gcd(a, b)
   if a mod b = 0
      return {0, 1}
   else
      {x, y} := extended_gcd(b, a mod b)
      return {y, x-y*(a div b)}
```

- The iterative method.
- The table method
- Etc...

Notes on the above steps:

- Step 1: For security purposes, the integer *p* and *q* should be chosen uniformly at random and should be of similar bit-length. Prime integers can be efficiently found using a Primarily test.
- Step 2: Choosing *e* with a small hamming weight results in more efficient encryption. Small public exponents (such as *e=3*) could potentially lead to greater security risks.

The **public key** consists of the modulus *n* and the public (or encryption) exponent *e*. The **private key** consists of the modulus *n* and the private (or decryption) exponent *d* which must be kept secret.

- For efficiency the following values may be pre-computed and stored as part of the private key:
  - *p* and *q*: the primes from the key generation,
  - $d \mod (p-1)$ and $d \mod (q-1)$,
  - $q^{-1} \mod (p)$.

3. Encryption

Alice transmits her public key (*n*,*e*) to Bob and keeps the private key secret. Bob then wishes to send message **M** to Alice. He first turns **M** into an integer $0 < m < n$ by using an agreed-upon reversible protocol known as a padding scheme. He then computes the ciphertext *c* corresponding to:

$$c \equiv m^e \pmod{n}.$$

This can be done quickly using the method of exponentiation by squaring. Bob then transmits c to Alice.

4. Decryption

Alice can recover *m* from *c* by using her private key exponent *d* by the following computation:

$$m \equiv c^d \pmod{n}$$

Given *m*, she can recover the original message **M** by reversing the padding scheme. The above decryption procedure works because:

$$c^d \equiv (m^e)^d \equiv m^{ed} \pmod{n}.$$

Now, since $ed = 1 + k\varphi(n).$,

$$m^{ed} \equiv m^{1+k\varphi(n)} \equiv m(m^k)^{\varphi(n)} \equiv m \pmod{n}.$$

The last congruence directly follows from Euler's theorem[WIKIc] when *m* is relatively prime to *n*. Euler's theorem(also known as the Fermat-Euler theorem) states that if *n* is a positive integer and *a* is a positive integer co-prime to *n* then :

$$a^{\varphi(n)} \equiv 1 \ (\ \mathrm{mod}\ \ n\ ).$$

In fact, Eucler's theorem is the generation of Fermat's little theorem which states that if n is a prime number, then for any integer a, a^n-a will be evenly divisible by n

$$a^n \equiv a \ (\ \mathrm{mod}\ \ n\ )$$

A variant of this theorem is :

$$a^{n-1} \equiv 1 \ (\ \mathrm{mod}\ \ n\ )$$

As we know $\varphi(n) = n-1$ if n is prime number. So we can generate Fermat's little theorem to Euler's theorem. We return to Euler's theorem, here $a = m^k$ and we assume m is co-prime to n. So

$$(m^k)^{\varphi(n)} \equiv 1 \ (\ \mathrm{mod}\ \ n\ ) \Rightarrow\ m(m^k)^{\varphi(n)} \equiv m \ (\ \mathrm{mod}\ \ n\ )$$

If m is not co-prime to n, it means m is a multiple of p or q because $m<n$ and p,q are two primes. We use the Chinese remainder theorem[WIKId] to resolve this congruence which is:

Suppose n_1, n_2, …, n_k are positive integers which are pair-wise co-prime. Then, for any given integers a_1,a_2, …, a_k there exists an integer x solving the system of simultaneous congruence

$$x \equiv a_1 \ (\ \mathrm{mod}\ \ n_1\ )$$
$$x \equiv a_2 \ (\ \mathrm{mod}\ \ n_2\ )$$
$$.....$$
$$x \equiv a_k \ (\ \mathrm{mod}\ \ n_K\ )$$

Furthermore, all solutions x to this system are congruent modulo the product $N = n_1n_2…n_k$.

So in using this theorem we can separate the original congruence to:

$$m(m^k)^{(p-1)(q-1)} \equiv m \ (\ \mathrm{mod}\ \ p\ )$$

and

$$m(m^k)^{(p-1)(q-1)} \equiv m \ (\ \mathrm{mod}\ \ q\ )$$

Assuming now m is a multiple of p, so

$$m(m^k)^{(p-1)(q-1)} \equiv 0 \equiv m \ (\ \mathrm{mod}\ \ p\ )$$

m is co-prime to q since m is already a multiple of p and (p, q)are two different primes, so :

$$m(m^{k(p-1)})^{(q-1)} \equiv m \ (\ \mathrm{mod}\ \ q\ )$$ because of Euler's theorem.

So

$$m(m^k)^{(p-1)(q-1)} \equiv m \ (\ \mathrm{mod}\ \ p\times q\ ) \Rightarrow$$
$$m(m^k)^{(p-1)(q-1)} \equiv m \ (\ \mathrm{mod}\ \ n\ )$$

This shows that we get the original message back: $c^d \equiv m \ (\ \mathrm{mod}\ n\ )$

NGUYEN Tuong Lan and LIU YI
University Lyon 1

5. A working example

Here is an example of RSA encryption and decryption. The parameters used here are artificially small, but one can also use OpenSSL to generate and examine a real keypair.

- Choose two prime numbers

$$p = 7 \text{ and } q = 19$$

- Compute $n = pq$

$$n = 7 \times 19 = 133$$

- Compute the totient $\varphi(n) = (p-1)(q-1)$

$$\varphi(n) = (7 - 1)(19 - 1) = 108$$

- Choose $e > 1$ co-prime to 108

$$e = 5$$

- Compute d such that $de \equiv 1 \pmod{\varphi(n)}$ e.g., by computing the modular multiplicative inverse of e modulo $\varphi(n)$:

$$d = 65$$
since $5 \cdot 65 = 325 = 1 + 3 \cdot 108$.

The **public key** is ($n = 133$, $e = 5$). For a padded message m the encryption function is: $c = m^e \bmod n = m^5 \bmod 133$. The **private key** is ($n = 133$, $d = 65$). The decryption function is:

$$m = c^d \bmod n = c^{65} \bmod 133$$

For example, to encrypt $m = 6$, we calculate

$$c = 6^5 \bmod 133 = 62$$

To decrypt $c = 62$, we calculate

$$. \ m = 62^{65} \bmod 133 = 6$$

The last calculation can be computed efficiently using the square-and-multiply algorithm for modular exponentiation [WIKIe]. First, it is required that the exponent *e* be converted to binary notation. That is, *e* can be written as:

$$e = \sum_{i=1}^{n-1} a_i 2^i$$

In such notation, the *length* of *e* is *n* bits. a_i can take the value 0 or 1 for any *i* such that $0 \leq i < n - 1$. By definition, $a_{n-1} = 1$.

The value b^e can then be written as:

$$b^e = b^{(\sum_{i=1}^{n-1} a_i 2^i)} = \prod_{i=0}^{n-1} (b^{2^i})^{a_i}$$

The solution *c* is therefore:

$$c \equiv \prod_{i=0}^{n-1} (b^{2^i})^{a_i} \pmod{n}$$

The application of the method to our example:

$m = C^d \% n$
$= 62^{65} \% 133$
$= 62 * 62^{64} \% 133$
$= 62 * (62^2)^{32} \% 133$
$= 62 * 3844^{32} \% 133$
$= 62 * (3844 \% 133)^{32} \% 133$
$= 62 * 120^{32} \% 133$
$= 62 * 36^{16} \% 133$
$= 62 * 99^8 \% 133$
$= 62 * 92^4 \% 133$
$= 62 * 85^2 \% 133$
$= 62 * 43 \% 133$
$= 2666 \% 133$
$= 6$

In real life situations the primes selected would be much larger, however in our example it would be relatively trivial to factor *n*=133, obtained from the freely available public key back to the primes *p* and *q*. Given *e*, also from the public key, we could then compute *d* and so acquire the private key.

6. Implement in Java (Folder RSA)

We implement a class Java qui generate the keys, encrypted the message, then decrypted the message coded.

B. Attack on RSA

1. Mathematical attack on RSA

1.1 Purpose of mathematical

If we know φ(n) and the public key (the modulus n and exponent e), we can determine the private key d because d.e ≡ 1 (mod φ(n)), and we can find d with Extended Euclidean algorithm. In fact, we use the Bezout identity:

Given a, b, we can find 2 integers x, y such as :

$$a.x + b.y = gcd(a,b)$$

In our case, this equation become d.e + k. φ(n) = 1 because gcd(e, φ(n)) =1. Plus, we have another condition which is gcd (d, φ(n)) =1. Knowing the private key d, we can read the message.

So, how can we determine φ(n)? Knowing φ(n) is equivalent to factoring n. Because

$$n - \varphi(n) + 1 = p.q - (p-1)(q-1) - 1 = p + q.$$

With p.q and p+q, we'll solve the equation in order to determine p and q

$$X^2 - (p+q).X + p.q = 0$$

For example, n= 2773, φ(n) = 2668, so p + q = 106, the equation is (X-47)(X-59), we deduce p = 47, q = 59.

1.2 Problems and difficulties

So with φ(n) and n , we can read any message encrypted. So, our problem is equivalent to the problem of Prime factorization. Prime factorization requires splitting an integer into factors that are prime numbers; and every integer has a unique prime factorization. Multiplying two prime integers together is easy, but as far as we know, factoring the product is much more difficult. Unluckily, we don't have for now an algorithm efficient for factoring an integer. We can take a look at some examples:

Keysize	Total Time	Factor Base	Sieve Memory	Matrix Memory
428	$5.5 * 10^{17}$	600K	24Mbytes	128M
465	$2.5 * 10^{18}$	1.2M	64Mbytes	825Mbytes
512	$1.7 * 10^{19}$	3M	128Mbytes	2 Gbytes
768	$1.1 * 10^{23}$	240M	10Gbytes	160Gbytes
1024	$1.3 * 10^{26}$	7.5G	256Gbytes	10Tbytes

-In 1994, RSA-129 (the key has the length of 426 bits) was factored in March 1994 by Atkins *et al*. [AGL95] using the resources of 1600 computers (which included two fax machines) from the Internet. The factoring took about 4000 to 6000 MIPS years of computation over an eight-month period

-In 2005, F. Bahr, M. Boehm, J. Franke, T. Kleinjung factored a 193-digit number (RSA-640) utilizing 30 2.2GHz-Opteron-CPU took years over a span of 5 months.

This is why the size of the modulus in RSA determines how secure an actual use of RSA is; the larger the modulus, the longer it would take an attacker to factor, and thus the more resistant to attack the RSA modulus is. Therefore, RSA can be called, mathematically, a strong encryption. The difficulties of breaking RSA is rely on the assumption that it is not possible to factor the product of two large prime numbers in polynomial time. In other words, for sufficiently large numbers the time it takes to factor such numbers increases faster than any finite-degree polynomial.

However, we still take a look at some ways of factoring integers.

1.3 Factoring Algorithm

A method tradition is Trial Division. We will divide successfully by 2, 3, 5, 7, 11 … This method is just effective for the number up to 1.000.000 not with a large number. Because the worst case time to factor a composite N is sqrt(N) divisions.

Now we will consider threes others methods: Express N as the difference of 2 squares (Algorithm de Fermat), express N as the sum of two squares in difference ways, finally Pollard's Rho method

1.3.1 Algorithm of Fermat

This method is based on the formula of difference of 2 squares:

$$x^2 - y^2 = (x - y)(x + y)$$

In case of the particular modulus N in RSA n = p.q (p, q are 2 primes), we can express n as: $n = \left(\frac{p+q}{2}\right)^2 - \left(\frac{p-q}{2}\right)^2$. We will follow step by step:

Let k the smallest positive integer so that $k^2 > n$. Consider now: $k^2 - n$. If we can a number h such as $k^2 - n = h^2$, then $n = (k + h)(k - h)$. As n can only factor as p.q. so we will be sure k+h and k-h must be 2 prime numbers. If not the case, we consider $(k+1)^2 - n$. Eventually, we will find g such as $(k+g)^2 - n = h^2$ is a square. So n can be factored as (k+g+h)(k+g-h).

Here an example, n = 6699557, $\sqrt{n} = 2588{,}3$ then we choose k = 2589.

$$k^2 - n = 3364 = 58^2, \text{ so n} = 2531.2647.$$

Take a look now at another example: n = 26504551 $\sqrt{n} = 5148.3$, so we choose k = 5149.

$$k^2 - n = 7650 \text{ is not a square}$$

$$(k+1)^2 - n = 17949 \text{ is not a square}$$
$$(k+2)^2 - n = 28250 \text{ is not a square}$$
$$\ldots\ldots$$
$$(k+691)^2 - n = 2757^2$$

So n can be factored as 3083.8597.

As we can see in 2 examples, this algorithm works if the difference between p and q is small. Unless, we have to compute a lots of operations. We will express g as the function of (p-q) and n. In fact :

$$(k+g)^2 - n = h^2$$
$$(\sqrt{n}+g) = \sqrt{h^2+n} \text{ (as } k \approx \sqrt{n}\text{)}$$

As n = (k+g+h)(k+g-h), and n = p.q. Plus n has only one way of factoring. So p = k+g+h and q=k+g-h. We have h = (p-q)/2.

Finally, we have:

$$g = \sqrt{\left(\frac{p-q}{2}\right)^2 + n} - \sqrt{n}$$

So, if the difference between p and q is large, the number of operations is grand (because we have to try from 1 to g).

1.3.2 Algorithm of Euler

The idea of this method is that if we can express a number n as the sum of 2 squares in 2 difference ways, we can factor n. We will talk about the condition of such expression, now we suppose that n can be displayed as 2 sums of 2 squares.

$$n = a^2 + b^2 = c^2 + d^2$$
(a-c)(a+c)=(d-b)(d+b)

Let k = gcd ((a-c),(d-b)), so

a-c = km, d-b = kh.

(h and m are the integer, and gcd(h,m)=1)

m(a+c)=h(b+d)

And as gcd(m,h)=1, so we must have (b+d) is divisible m. We have (b+d)=m.n. Consquently, a+c=h.n. We have 4 equations:

a-c=k.m
a+c=h.n
d-b=kh
d+b=m.n
We deduce:
a=(km+hn)/2
b=(mn-kh)/2

$$n = \left(\frac{km+hn}{2}\right)^2 + \left(\frac{mn-kh}{2}\right)^2$$

$$n = \frac{k^2.m^2 + h^2.n^2 + 2kmhn + m^2.n^2 + k^2.h^2 - 2kmhn}{4}$$

$$n = \frac{(k^2 + n^2)m^2 + h^2(k^2 + n^2)}{4}$$

$$n = \frac{(k^2 + n^2)(m^2 + h^2)}{4}$$

In our case, we know n is factored as p.q. So we can express p and q.

Euler's factorization method is more effective than Fermat's for integers of which factors are not close and actually much more efficient than trial division if one can find representations of numbers as sums of two squares reasonably easily. However, the great disadvantage of Euler's factorization method is that it cannot be applied to factoring an integer with any prime factor of the form $4k+3$ occurring to an odd power in its prime factorization, as such a number can never be the sum of two squares. As Fermat said, the odd prime p can be expressed by $x^2 + y^2$ if and only if $p \equiv 1 (\mathrm{mod}\, 4)$.

1.3.3 Pollard's Rho method.

Let's say p is one of two factors of n. If you start picking numbers at random (keeping your numbers greater or equal to zero and strictly less than n), then the only time you will get $a \equiv b (\mathrm{mod}\, n)$ is when a and b are identical. However, since p is smaller than n, there is a good chance that $a \equiv b (\mathrm{mod}\, p)$ sometimes when a and b are not identical.

$a \equiv b (\mathrm{mod}\, p)$ means (a-b) is multiple of p, we also know that n is multiple of p. So to find p , all we have to do is to compute gcd (a-b, n).

So, this principal of Pollard's Rho method is to pick a random number *a*. Pick another random number *b*. See if the greatest common divisor of *(a-b)* and *n* is greater than one. If not, pick another random number *c*. Now, check the greatest common divisor of *(c-b)* and *n*. If that is not greater than one, check the greatest common divisor of *(c-a)* and *n*. If that doesn't work, pick another random number *d*. Check *(d-c)*, *(d-b)*, and *(d-a)*. Continue in this way until you find a factor.

Actually, we will do it with faster way. Because we're only concerned with numbers from zero up to (but not including) *n*, we will take all of the values of *f(x)* modulo *n*. We start with some x_1. We then choose to pick our random numbers by $x_{k+1} \equiv f(x_k) \ (mod\ n)$. The choice of function is under the form f(x) = $x^2 + a$

Example:

f(x) = $x^2 - 1$

Again, if n = 1189, this proceeds as follows. Let x(0) = 4.

k	x(2*k-1)	x(2*k)	x(2*k) - x(2*k-1)	d(k)
0		4		
1	15	224	209	1
2	237	285	61	1
3	372	459	222	1
4	227	401	116	29

Thus N = 29*41.

1.3.4 Implement Fermat and Pollard's Rho (Folder Factoring)

For method of Pollard's Rho, we use the function f(x) = $x^2 + c$. The two first number is x and y= x^2. The next step x= f(x) and y= f(f(y)).

We obtain the time of factoring N (in ms)

Number of bit (N)	Method Rho	Fermat
10	0	31
20	94	219
25	200	13375
25	198	3375
25	188	765
32	282	899515
40	8000	
40	8456	
50		

As we can see, from 50 bits, the factoring is too expensive. So, as a conclusion for this part, to break RSA, mathematically, we will concentrate on improving the power of the computer. In fact, we can use an extremely fast factoring machine known as TWIRL. TWIRL promises to break even 1024bits RSA encryption in less than a year for an extremely reasonable price. Using the Number Sieve Factoring theorems TWIRL uses parallel processing to factor large numbers at high speeds. Using the same amount of silicon as a Pentium Xeon Processor and 1 GB of memory, the TWIRL device searches 20,000 times faster than the Intel chip.

2. Implementation Attack

2.1 Introduction

The previous topic shows us the mathematical strength of RSA. However, the research has shown us that it is able to recover the private key of RSA without directly breaking RSA (factoring the modulus n). In fact, Kocher[RCKa96] succeeded in recovering the RSA factors by carefully measuring amount of time required to perform the operation of keys. The method is called Timing attack. Actually, Timing Attack is one of the subsets which belong to Side Channel Attack.

The Side Channel Attack is an attack that bases on information gain from the physical implementation of a cryptosystem such as: timing information, power consumption, electromagnetic leak ....

2.2 General Method

Just for reminding the encryption using RSA, to encrypt a plaintext message *M*, we compute $C = M^e \bmod n$ (e is public key), where *C* is the encoded message, which is called ciphertext. To decrypt the ciphertext *C*, we compute $M = C^d \bmod n$ (d is a private key), which yields the original message. For a timing attack, the attacker needs to have

the target system compute C^d mod n for several carefully selected values of C. And with a measurement of time performance, the attacker can recover bit by bit the private key d.

As we can see above, to create the ciphertext, we have to compute $M = C^d$ mod n which is complex when N, M are large. So, in order to speed up the operation, we'll use the Fast Exponentiation Algorithm (Square and Multiply Algorithm) which is described as below:

We have the binary expression of d is $\sum_{i=0}^{t-1} d_i.2^{t-1-i}$

$d = d_o d_1 .. d_{t-1}$ (t is the length of keys). Normally, $d_0 = 1$, because we know the length of keys.

```
z = C
for j = 1 to t-1 do
    z ≡ z^2 mod n
    if d_j == 1 then
        z ≡ z.C mod n
    endif
endfor
return z
```

(z returned is our message M).

Here, we remark that the operation a mod b is done only when a > b. If that's the case, a mod b = (a-b) mod b. Unless, in case a< b, we have a mod b = a. Therefore, if the j-th bit de private key d is 1 we have an extra operation: $z \equiv z.C \bmod n$ which will put an extra time. Plus, if z.C > n we will also have an extra time of operation. So all we have to do is to create a subset of messages, then separate them in different categories. Finally we measure carefully the time required to perform these operations in order to decide the j-th bit is 1 or 0. We will now consider a set of messages.

2.3 Attack the multiply

Actually, we will choose 2 types of messages: one needs an extra time to operate the reduction when j-th bit of private is 1, second does not need this reduction. With more details, we determine message E and F such as;

$$E^3 < n$$

$$F^2 < n < F^3$$

We will discover d_j.

So, if $d_j = 1$ we have to operate $E^2.E$ mod n and $F^2.F$ mod n. But these two operations don't have the same time to perform. In case of F, we have a reduction $F^3 - n$ (because $n < F^3$) which is not performed in case of E (because $E^3 < n$). If d_j=0, we will perform E^2 mod n and F^2 mod n. Since $E^2 < n$ and F^2<n, we don't have any reduction in two cases, so the performances will take the same time. It means the run time of the algorithm will be longer for F if and only if $d_j = 1$. We will discover d_j.

However, in reality, how can we make a decision about this difference? In fact, instead of choosing 2 types of messages, we choose 2 series of messages: $E_1, E_2 ... E_k ; F_1, F_2, ... F_k$. And then we calculate the time average $timeE = (E_1 + E_2 + ... + E_k)/k; timeF = (F_1 + F_2 + ... + F_k)/k$. If timeE < timeF+e, we decide $d_j = 1$ (here, e can be determined by experiences). Once $d_j = 1$, we can discover bits rested of private key d.

Here an example. The results are obtained by Computer Pentium Pro 200MHz Windows NT. [JFDa98].

Key size	Result			
	without error correction		with error correction	
	sample size	speed	sample size	speed
64	1 500–6 500	> 20 bits/s	1 500–4 500	> 20 bits/s
128	12 000–20 000	2 bits/s	6 000–10 000	4 bits/s
256	70 000–80 000	1 bit/4s	40 000–50 000	1 bit/2s
512	±350 000–400 000	1 bit/65s	200 000 – 300 000	1 bit/37s

2.4 Our implement in Java (Folder TimingAttack)

In this implement, we rewrite the function modPow of class BigInteger. This version is very simple, we chose only 1 E et 1 F. We will show you that, that number of errors increased when length of keys increase.

```
public BigInteger decrypt(BigInteger message) {
      return message.modPow(d, N);
  }

public long decrypt1(BigInteger message, int i){
    BigInteger z;
    BigInteger zz;
    BigInteger zC=null;
    long nano;
    z = message;

          nano=System.nanoTime();
          zz= z.multiply(z).mod(N);
          z=zz;
          if (d.testBit(i)){
                zC = z.multiply(message).mod(N);
                z=zC;
          }

    return System.nanoTime()-nano;
 }
```

For the attack we will execute decrypt1 length of private key d times:

```
for (int i=rsa.d.bitLength()-2;i>-1;i--){
    long timeDecryptedE = rsa.decrypt1(ta.E,i);
    long timeDecryptedF = rsa.decrypt1(ta.F,i);
    if (timeDecryptedE < timeDecryptedF){
        d_recover = d_recover+"1";
        if (!rsa.d.testBit(i)){// if the exact bit is 0, it's error
            k++;
        }
     }else{
        if (!rsa.d.testBit(i)){// if the exact bit is 1, it's error
            k++;
        }
        d_recover = d_recover+"0";
    }
    }
    System.out.println(k);
    System.out.println(d_recover);
```

We compare the recovered key with the private key to determine the number of error bits. We obtain the result following:

Length of private key	Number of bits error
62	*2*
62	*4*
62	*5*
62	*6*
62	*6*
125	*30*
125	*44*
125	*30*
125	*36*
125	*30*
254	*50*
256	*48*
256	*60*
256	*49*
255	*52*
266	*48*
256	*46*

As we can see, the errors is too much, sometimes we can only recover a half of the key. In a reality, we need to try with 100,000 samples of E and F (it depends on the length of key too [JFDa98]) for recover the full key.

C. Against-Attack

1. Chinese reminder theorem (CRT)

1.1 The idea

As we know, the generation of keys RSA is the series of operations modulus. The more keys' length, the more time consuming on computer. More important, RSA using CRT can be saved from Timing Attack. Because we don't compute directly modules d (the private key) but the modulus of factors (we can see it so far in this report). The Chinese have the genius idea which is: We can express the modulus of a number as the modulus of its factors. So we won't compute the modulus directly with d. It will maybe prevent the Timing Attack.

Let N = pq denote a RSA modulus, e a public exponent, and d the corresponding private exponent, so that $d.e \equiv 1 \bmod n$. Let CRT(x; y) mod N denote the number z such that $z \equiv x \bmod p$ and $z \equiv y \bmod q$; note that this number is uniquely determined modulo N..

1.2 Speed up RSA generation of keys with CRT

With help from the CRT we can speed up the decryption of RSA as follows. As a first idea we can compute the result mod p and q separately, i.e. to merge the results of $C^d \pmod p$ and $C^d \pmod q$. This will require twice as many operations as we need to compute two exponentiations. However as partial results need only be calculated modulo p and modulo q respectively, these operations are done with numbers of only half as many bits and hence each multiplication costs only a forth of what it costs for full size numbers. As CRT is almost for free we gain a factor about 2 in running time.

Our goal is to find z such as:

$$z \equiv m^d \bmod n$$

We will compute

$$\begin{aligned} x &\equiv m^{d \bmod (p-1)} \bmod p \\ y &\equiv m^{d \bmod (q-1)} \bmod q \end{aligned} \quad (2)$$

We can use the result of Garner which is [SMYa]

$$z = x + p.([(y - x).(q^{-1} \bmod p)] \bmod q)$$

Implement in Java (folder RSA-CRT)

The result of decryption using CRT is always faster then RSA normal (100 compare to 400-500 ms with the keys de 2048 bits)

2. Padding scheme

When used in practice, RSA is generally combined with some padding scheme. The goal of the padding scheme is to prevent a number of attacks that potentially work against RSA without padding [WIKIa]:

- When encrypting with low encryption exponents (e.g., $e = 3$) and small values of the m, (i.e. $m < n^{1/e}$) the result of m^e is strictly less than the modulus n. In this case, ciphertexts can be easily decrypted by taking the eth root of the ciphertext over the integers.
- Because RSA encryption is a deterministic encryption algorithm – i.e., has no random component – an attacker can successfully launch a chosen plaintext attack against the cryptosystem, by encrypting likely plaintexts under the public key and test if they are equal to the ciphertext. A cryptosystem is called semantically secure if an attacker cannot distinguish two encryptions from each other even if the attacker knows (or has chosen) the corresponding plaintexts. As described above, RSA without padding is not semantically secure.
- RSA has the property that the product of two ciphertexts is equal to the encryption of the product of the respective plaintexts. That is $m_1^e m_2^e \equiv (m_1 m_2)^e \bmod n$. Because of this multiplicative property a chosen-ciphertext attack is possible. E.g. an attacker, who wants to know the decryption of a ciphertext $c = m^e \bmod n$ may ask the holder of the private key to decrypt an unsuspicious-looking ciphertext $c' = cr^e \bmod n$ for some value r chosen by the attacker. Because of the multiplicative property c' is the encryption of $mr \bmod n$. Hence, if the attacker is successful with the attack, he will learn $mr \bmod n$ from which he can derive the message m by multiplying mr with the modular inverse of r modulo n.

To avoid these problems, practical RSA implementations typically embed some form of structured, randomized padding into the value m before encrypting it. This padding ensures that m does not fall into the range of insecure plaintexts, and that a given message, once padded, will encrypt to one of a large number of different possible ciphertexts.

3. Padding method

- **Bit padding**

A single set ('1') bit is added to the message and then as many reset ('0') bits as required are added. The number of reset ('0') bits added will depend on the block boundary to which the message needs to be extended. In bit terms this is "1000 ... 0000", in hex byte terms this is "80 00 ... 00 00". This method can be used to pad messages which are any number of bits long, not necessarily a whole number of bytes long.

- **PKCS7**

Padding is in whole bytes(octets). The value of each added byte is the numbers of bytes that are added, i.e. N bytes, each of value N are added. The number of bytes added will depend on the block boundary to which the message needs to be extended. The padding will be one of:

01
02 02
03 03 03
04 04 04 04
05 05 05 05 05
etc.

Example: In the following example the block size is 12 bytes and padding is required for 4 bytes

... | DD DD DD DD DD DD DD DD | DD DD DD DD 04 04 04 04 |

D. Conclusion

By studying the cryptography algorithm RSA, I have learned

- ✧ many famous theorems such as : Euler's theorem, Fermat's little theorem, Chinese remainder theorem and their applications in RSA.
- ✧ The concepts : congruence relation, Euler's totient function, modular multiplicative inverse, modular exponentiation, padding scheme.
- ✧ The algorithm: extended Euclidean algorithm for resolving the modular multiplicative inverse in order to finding the private key, square-and-multiply algorithm for modular exponentiation.

With mastering the principle of RSA, I find out the vulnerabilities of it, and I also studied how to defend the RSA against an experienced attacker, that's padding scheme. Of course, we have many others types of attack, like factorizing directly n which is p*q or timing attack, but this is not my part, my co-worker will present it blow.

And I created also a program in java to test this algorithm, with the test I do in the end which verifies if the message decrypted is equal to the original message, it proves that this program works well.

For my conclusion, the system using RSA is not always protected in spite of its strength. The Timing Attack showed us that the attackers can break a system without knowing the algorithm of encryption/decryption. As we can see above, the attackers have to know the private key d and then compute the modulus to obtain the message.

And after that, the research has responded this type of attack by using Chinese Remainder Theorem. Then again, we have another type of attack called Fault Attack. In fact, the fault compute x and y1 on (2) can be used for discovering p or q

$$z1 = x + p.([(y1 - x).(q^{-1} \bmod p)] \bmod q)$$

By computing gcd(z1-z,n), we will have p, the first divisor.

Finally, only one regret is that we could not implement a Timing Attack on OpenSSL as we planned because not only the proof of all theorems but also the implementation of the algorithms took us too much time.

Reference:

[SMYa] Cryptanalysis of Two Protocols for RSA with CRT based on Fault Infection Sung-Minh Yen – Dong ryeol Kim

[RCKa96] Timing Attack on implementations of Diffie-Hellman, RSA, DSS and Other Systems Paul C. Kocher

[JFDa98] :A Practical Implementation of the Timing Attack de J.-F. Dhem, F. Koeune, P.-A. Leroux, P. Mestré, J.-J. Quisquater and J.-L. Willems

[WIKIa] http://en.wikipedia.org/wiki/RSA

[WIKIb] http://en.wikipedia.org/wiki/Extended_Euclidean_Algorithm

[WIKIc] http://en.wikipedia.org/wiki/Euler%27s_theorem

[WIKId] http://en.wikipedia.org/wiki/Chinese_remainder_theorem

[WIKIe] http://en.wikipedia.org/wiki/Modular_exponentiation