5 - The RSA algorithm

Jean-Baptiste Campesato

1 Introduction

How can someone send a secret message in a way that only the recipient could read the content even if the message happens to have been intercepted by a third party? There are several ways to do so.

Early cipher algorithms relied on a unique key which had to be used both by the sender to encrypt the message and by the recipient to decrypt it. A very simple example is Caesar's cipher which consists in shifting letters according to the key.

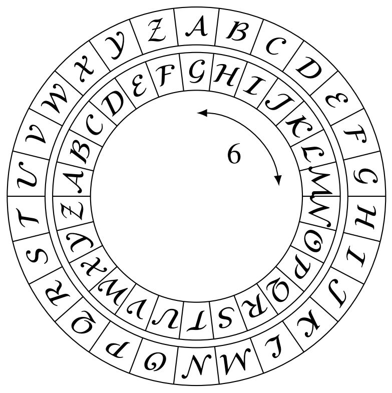
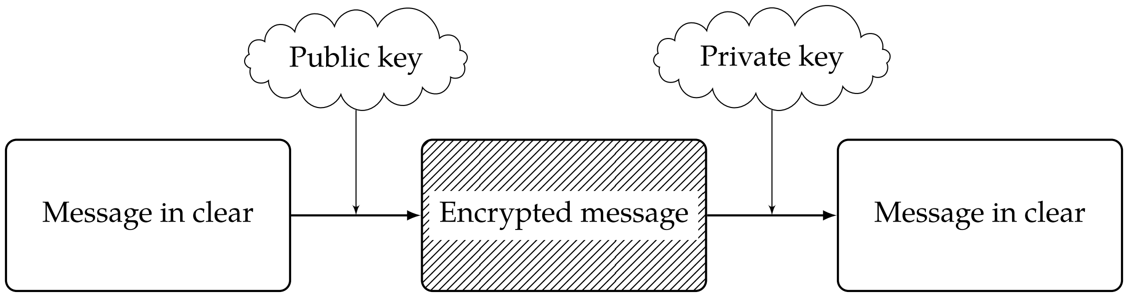


Figure 1: Caesar ciphering with key=6

A major weakeness of such a system is that the key has to be disclosed to all the participants, increasing the risk for the key to be compromised by a third party.

Asymmetric algorithms allow to reduce this weakeness by using two keys: a public key which is used to encrypt messages and which can be widely shared without compromising the exchanges, and a private key which is kept only by the recipient to decrypt the data. The idea is that anyone can encrypt messages using the public key but only the recipient can decrypt them with his private key. Particularly, the knowledge of the public key should not be enough to decrypt messages.



Asymmetric cryptographic systems were theoretically developed in the mid 70s and the first concrete algorithm of this kind was the RSA algorithm which appeared in 1978 and is named from the initials of its authors (Ron Rivest, Adi Shamir, and Leonard Adleman). Actually, the British secret service developed a similar algorithm as early as 1973 but it was kept confidential until the 90s.

In this chapter, we are going to explain the RSA algorithm which relies on modular arithmetic. The original proof of work used Fermat's little theorem but the RSA algorithm is actually easier to explain using Euler's theorem (which is a generalization of Fermat's little theorem, as you already know).

The robustness of this cipher relies on the fact that we don't know yet an efficient algorithm in order to find the prime decomposition of a given positive integer. This last difficult problem will not be addressed in this chapter.

2 Generation of the keys

The recipient, that we will call Alice, picks two distinct prime numbers p and q. She sets $n \coloneqq pq$ and then she chooses $e \in \mathbb{N}$ such that $\gcd(e, \varphi(n)) = 1$. Then the public key is (n, e). Alice can publicly provide this key to people willing to send her a crypted message.

Since $\gcd(e, \varphi(n)) = 1$, e admits a multiplicative inverse modulo $\varphi(n)$, i.e. there exists $d \in \mathbb{N}$ such that $ed \equiv 1 \pmod{\varphi(n)}$. Indeed, there exist $u, v \in \mathbb{Z}$ such that $eu + \varphi(n)v = 1$ (and we can easily find such a Bézout's relation using Euclid's algorithm). Then we take $d = u + k\varphi(n)$ for a suitable $k \in \mathbb{Z}$ for d to be positive. Then the private key is (n, d). Alice should not share this key with anyone else.

Note that in order to find a suitable d, it is necessary to know $\varphi(n)$ and e. Alice knows the prime numbers p and q that she used to define n so she can easily compute $\varphi(n) = (p-1)(q-1)$. But the shared information is only the public key (n, e). Although it is theoretically possible to find $\varphi(n)$, there is no known efficient algorithm to compute $\varphi(n)$ directly from n. Nonetheless, if a third party were able to quickly compute the prime factorization of a positive integer, then it could computes $\varphi(n)$ allowing him to recover (n, d) from (n, e).

The prime numbers p and q should be chosen wisely so that there is no known efficient algorithm to recover p and q from n using our current computing power. For instance, not only p and q should be large enough but $\delta = |p - q|$ should be large too. Indeed, assume that $p < q$ then $q = p + \delta$. Thus $\sqrt{n} = p\sqrt{1 + \frac{\delta}{p}} \sim p + \frac{\delta}{2}$. Hence, according to Proposition 3 of Chapter 3, it is enough to check whether numbers less than $\sqrt{n}$ divides n, and from the above estimation, p could be obtained after less than $\frac{\delta}{2}$ attempts (starting from $\sqrt{n}$).

3 How to encrypt a message

The sender, that we are going to call Bob, wants to send a secret message to Alice. But he wants to make sure that only her can read the content. First, Bob obtains her public key (n, e).

A message is going to be an element of $m \in \{0, 1, \dots, n-1\}$ (in practice, Alice and Bob need to agree on how to reduce a human readable message into a sequence of natural numbers less than n, that's the goal of the various protocols used in computer sciences).

Then, there exists a unique $c \in \{0, 1, \dots, n-1\}$ such that $c \equiv m^e \pmod{n}$. It is going to be the crypted message. Bob sends c to Alice, and Alice will use her private key in order to recover m from c.

4 How to decrypt a message

Alice just received the secret message c from Bob. He told her that it was encrypted using her public key (n, e). Since she knows her private key (n, d), Alice can find the unique $k \in \{0, 1, \dots, n-1\}$ such that $k \equiv c^d \pmod{n}$.

We claim that $m = k$. Indeed, since $ed = 1 + l\varphi(n)$ for some $l \in \mathbb{N}$, we obtain using Euler's theorem that

$$k \equiv c^d \pmod{n} \equiv m^{ed} \pmod{n} = m^{1+l\varphi(n)} \pmod{n} \equiv m \times \left(m^{\varphi(n)}\right)^l \pmod{n} \equiv m \times 1^l \pmod{n} \equiv m \pmod{n}$$

We conclude since k has a unique representative in $\{0, 1, \dots, n-1\}$ and $m, k \in \{0, 1, \dots, n-1\}$.

Note that the above proof doesn't work when $\gcd(m,n)\neq 1$, i.e. when $p|m$ or $q|m$ (because we can't apply Euler's theorem). Nonetheless, it is still true that $m^{ed}\equiv m \pmod n$ in this case (you will prove it during next week tutorials).

5 An example

Alice wants to create a pair of keys for the RSA algorithm so that people could send her secret messages. She picked the prime numbers $p=13$ and $q=17$ then $n=221$ and $\varphi(n)=12\times 16=192$. Then she picks $e=11$, which is a suitable choice since $\gcd(192,11)=1$.

Using Euclid's algorithm, Alice obtains the Bézout relation $192\times(-2)+11\times(35)=1$. Therefore, she sets $d=35$ so that $ed\equiv 1 \pmod{192}$. Finally, she shares the public key $(n,e)=(221,11)$ on her website and preciously keeps the private key $(n,d)=(221,35)$ for herself only.

Later, Bob wants to send the private message $m=149\in\{0,1,2,\dots,220\}$ to Alice. He finds on her website her public key and computes $m^e=149^{11}\equiv 89 \pmod{221}$. So the encrypted message is $c=89\in\{0,1,2,\dots,220\}$. He sends it to Alice by e-mail.

After receiving the e-mail, Alice computes $c^d=89^{35}\equiv 149 \pmod{221}$ and she recovers the original message $m=149$.

6 In practice

It is not difficult to find a Bézout relation using Euclid's algorithm. But two other things seem not to be very practical in the above example:

1. How to generate the prime numbers p and q?
2. The computations seem to involve very large numbers which are not suitable to computers (149^{11} is already a very large number).

The first problem is a little bit tricky. In practice we generate a random odd number k of the wanted order of magnitude and we check whether it is prime or not. If not, we take the next odd number and we repeat the process. According to the prime number theorem[1] we could expect a prime number before $\frac{\ln(k)}{2}$ attempts.

Nonetheless, we don't know efficient algorithms to check whether a number is prime or not. Instead, we usually use probabilistic primality tests (so they can fail, but with a very low probability).
Some algorithms rely on Fermat's little theorem: if p is prime then $\forall a\in\mathbb{Z},\ a^p\equiv a \pmod p$.
Therefore, since $24^{221}\equiv 176 \pmod{221}$, we know that 221 is not prime.
Nonetheless, it is possible for such a congruence to hold even for a non-prime number a, for instance we have $2^{341}\equiv 2 \pmod{341}$ although $341=11\times 31$.

The second problem has easy workarounds. First, note that we don't need to actually compute m^e. Indeed, we only need a representative modulo n. More precisely, given $m,e,n\in\mathbb{N}$, we want to find (the unique) $c\in\{0,1,\dots,n-1\}$ such that $m^e\equiv c \pmod n$. One naive way to avoid very large numbers consists in iteratively multiplying by c and to reduce to a representative in $\{0,\dots,n\}$ before the next step.
For instance, in order to compute $149^{11} \pmod{221}$, we would do:

1. $149^1\equiv 149 \pmod{221}$
2. $149^2=149\times149=22201\equiv 101 \pmod{221}$
3. $149^3=101\times149=15049\equiv 21 \pmod{221}$
4. $149^4=21\times149=3129\equiv 35 \pmod{221}$
5. $149^5=35\times149=5215\equiv 132 \pmod{221}$
6. $149^6=132\times149=19668\equiv 220 \pmod{221}$
7. $149^7=220\times149=32780\equiv 72 \pmod{221}$
8. $149^8=72\times149=10728\equiv 120 \pmod{221}$
9. $149^9=120\times149=17880\equiv 200 \pmod{221}$
10. $149^{10}=200\times149=29800\equiv 186 \pmod{221}$
11. $149^{11}=186\times149=27714\equiv 89 \pmod{221}$

Note that no involved number exceeded 32780 whereas $149^{11}=80361669864744786813914 9$ (actually $149\times 220=32780$ is the largest number we could have obtained).

[1]For k large enough, there are *about* $\frac{k}{\ln(k)}$ prime numbers less than or equal to k.

We even have even far more efficient algorithms.

First, write the exponent in binary $e = \overline{a_r a_{r-1} \dots a_1 a_0}^2 = \sum_{i=0}^{r} a_i 2^i$ where $a_i \in \{0, 1\}$. Then

$$m^e = m^{\sum a_i 2^i} = \prod_{i=0}^{r} \left(m^{2^i}\right)^{a_i}$$

So we just need to compute successive squares: $m^{2^{i+1}} = \left(m^{2^i}\right)^2$ *(actually we only need it modulo n)*.

Implementation in Julia:

```julia
function fastpowmod(m,e,n::Integer)
   n > 0 || error("n must be positive")
   e >= 0 || error("e must be non-negative")
   r = 1
   while e > 0
      if (e & 1) > 0
         r = (r*m)%n
      end
      e >>= 1
      m = (m^2)%n
   end
   return r>0 ? r : r+n
end
```

Output:

```
julia> fastpowmod(149,11,221)
89
```

7 A simple implementation in Julia

Source:

```julia
using Primes

struct PublicKey
   n::Integer
   e::Integer
end

struct PrivateKey
   n::Integer
   d::Integer
end

function gen_keys(p::Integer, q::Integer, e::Integer)
   isprime(p) || error("p must be a prime number")
   isprime(q) || error("q must be a prime number")
   e>0 || error("e must be positive")
   phi = (p-1)*(q-1)
   (g,u,v) = gcdx(e,phi)
   g == 1 || error("phi(n) and e must be coprime")
   u<0 ? d=(u%phi)+phi : d=u%phi
   n = p*q
   return PublicKey(n,e),PrivateKey(n,d)
end

function encrypt(m::Integer, k::PublicKey)
   0 <= m || error("m must be non-negative")
   m < k.n || error("m is too large")
   return powermod(m,k.e,k.n)
end

function decrypt(c::Integer, k::PrivateKey)
   0 <= c || error("c must be non-negative")
   c < k.n || error("c is too large")
   return powermod(c,k.d,k.n)
end

(pbk,pvk) = gen_keys(13,17,11)
println("The public key is (n,e)=($(pbk.n),$(pbk.e)), give it to people
   willing to send you a secret message!")
println("The private key is (n,d)=($(pvk.n),$(pvk.d)), don't share it!")
m = 149
println("Original message: $m")
c = encrypt(m, pbk)
println("Encrypted message: $c")
println("Decrypted message: $(decrypt(c,pvk))")
```

Output:

```
[mat246@Pavilion mat246]$ julia rsa.j
The public key is (n,e)=(221,11), give it to people willing to send you a
   secret message!
The private key is (n,d)=(221,35), don't share it!
Original message: 149
Encrypted message: 89
Decrypted message: 149
```