Théorie des codes - TP 6
ZZ3 F5 - Réseaux et Sécurité Informatique
Cryptographie Moderne : Chiffrement RSA

La méthode de cryptographie RSA [a] a été inventée en 1977 par Ron Rivest, Adi Shamir et Len Adleman, à la suite de la découverte de la cryptographie à clé publique par Diffie et Hellman. Le RSA est encore le système cryptographique à clé publique le plus utilisé de nos jours. Il est intéressant de remarquer que son invention est fortuite : au départ, Rivest, Shamir et Adleman voulaient prouver que tout système à clé publique possède une faille. RSA a été breveté par le Massachusetts Institute of Technology en 1983 aux États-Unis. Le brevet a expiré le 21 septembre 2000.

[a] Disponible :https://people.csail.mit.edu/rivest/Rsapaper.pdf

Ron Rivest, Adi Shamir et Len Adleman

Chiffrement de RSA

Introduction : Le chiffrement RSA est souvent utilisé pour communiquer une clef de chiffrement symétrique, qui permet alors de poursuivre l'échange de façon confidentielle : Alice envoie à Bob une clef de chiffrement symétrique qui peut ensuite être utilisée par Alice et Bob pour échanger des données.
Création des clefs : Le chiffrement RSA repose sur un utilisateur (Alice) qui crée son couple (clef publique / clef privée) en utilisant la procédure suivante :

- Choisir deux grands nombres premiers distincts p et q ;
- Calculer $n = p \times q$, appelé *module de chiffrement* ;
- Calculer $\phi(n) = (p-1)(q-1)$, qui est la valeur de l'*indicatrice d'Euler* en n ;
- Choisir un entier naturel e premier avec $\phi(n)$ et strictement inférieur à $\phi(n)$, appelé *exposant de chiffrement* ;
- Calculer d, l'inverse de e modulo $\phi(n)$ appelé *exposant de déchiffrement* ;
- Publier la paire $K_e = (e, n)$ comme *clef publique RSA* ;
- Garder secrète la paire $K_d = (d, n)$ qui est la *clef privée RSA*.

Chiffrement : Si M est un entier naturel strictement inférieur à n représentant un message, alors le message chiffré sera représenté par

$$E(M) = C \equiv M^e \mod n$$

L'entier naturel C étant choisi strictement inférieur à n.

Déchiffrement : Pour déchiffrer C, on utilise d, l'inverse de e modulo $(p-1)(q-1)$, et on retrouve le message clair M par

$$D(C) = M \equiv C^d \mod n.$$

D'après le théorème RSA on remarque bien que $D_{K_d}(E_{K_e}(M)) \equiv M^{ed} \mod n = M^{1+\alpha\phi(n)} \mod n = M$.

Implémentation : Pour déterminer un grand nombre premier, on utilise un procédé qui fournit à la demande un entier impair aléatoire d'une taille suffisante puis on réalise un test de primalité pour déterminer s'il est ou non premier. On réalise plusieurs essais, sachant que le théorème des nombres premiers assure que l'on trouve un nombre premier au bout d'un nombre raisonnable d'essais. Pratiquement, il est nécessaire d'avoir un test de primalité très rapide. On utilise donc un test probabiliste comme le *test de primalité de Miller-Rabin* (cf annexe 1). Ce test ne garantit pas exactement que le nombre soit premier, mais seulement une (très) forte probabilité qu'il le

soit.
Le calcul de $M \equiv C^d \mod n$ ne se fait pas en calculant C^d, puis le reste modulo n, car cela demanderait de manipuler des entiers beaucoup trop grands. Il existe des méthodes efficaces pour le calcul de l'*exponentiation modulaire* (cf annexe 1). L'exposant de déchiffrement peut se calculer efficacement par l'*algorithme d'Euclide étendu* ou par le *théorème d'Euler* (cf annexe 1).

Dans ce TP, nous allons réaliser un chiffrement et déchiffrement RSA en utilisant la librairie GMP (*Arithmetic without limitation*).

1. Dans un premier temps, utilisez au maximum les fonctionnalités de la librairie GMP pour mettre en place la génération des clefs RSA. Vous utiliserez les fonctions : `mpz_nextprime`, `mpz_powm` et `mpz_invert`. Vous trouverez plus de détails dans l'annexe 2.

2. Vous testerez votre algorithme de chiffrement et de déchiffrement sur quelques données aléatoires.

3. Dans un second temps, nous allons remplacer les différentes fonctions fournis par GMP en les codant par nous même (cf. annexe 1). Vous remplacerez la fonction `mpz_powm` par l'algorithme d'exponentiation rapide, la fonction `mpz_nextprime` par une fonction utilisant le test de primalité de Rabin-Miller et la fonction `mpz_invert` par l'utilisation de l'algorithme d'Euclide étendu ou le théorème d'Euler.

Appendix 1 RSA Implementation

Euclidean algorithm , or Euclid's algorithm, is an efficient method for computing the greatest common divisor (GCD) of two numbers, the largest number that divides both of them without leaving a remainder.

Algorithm 1 Euclidean algorithm : GCD(a,b)

```
Require: two integers $a$ and $b$ ;
Ensure: GCD($a$,$b$)
if $b = 0$ then
    return a
else
    Compute recursively GCD($b$,$a \mod b$)
end if
```

Exponentiation by squaring is a general method for fast computation of large positive integer powers of a number.

Algorithm 2 Exponentiation by squaring : $m \equiv g^k \mod p$

```
Require: three integers $g$, $k$ and $p$ ;
Ensure: Compute $m$
if $k < 0$ then
   $g \leftarrow 1/g$
   $k \leftarrow -k$
end if
if $k = 0$ then
   return $m = 1$
end if
$y \leftarrow 1$
while $k > 1$ do
   if $k$ is even then
      $g \leftarrow g \times g \mod p$
      $k \leftarrow k/2$
   else
      $y \leftarrow g \times y$
      $g \leftarrow g \times g$
      $k \leftarrow (k-1)/2$
   end if
end while
return $m = g \times y \mod p$
```

Miller-Rabin primality test or Rabin-Miller primality test is a primality test : an algorithm which determines whether a given number is prime.

Algorithm 3 Rabin-Miller primality test

```
Require: $n > 2$, an odd integer to be tested for primality ;
Require: $k$, a parameter that determines the accuracy of the test
Ensure: composite if $n$ is composite, otherwise probably prime
Write $n-1$ as $t \times 2^s$ with $t$ odd by factoring powers of 2 from $n-1$
loop
   Repeat $k$ times
   Pick $a$ randomly in the range $[2, n-1]$
   $x \leftarrow a^t \mod n$
   if $x = 1$ or $x = n-1$ then
      Do next LOOP
   end if
   for $r = 1...s-1$ do
      $x \leftarrow x \times x \mod n$
      if $x = 1$ then
         return $n$ is composite
      end if
      if $x = n-1$ then
         Do next LOOP
      end if
   end for
   return $n$ is composite
end loop
return $n$ is probably prime
```

Appendix 2 GMP - Arithmetic without limitation

GNU MP[1] is a portable library written in C for arbitrary precision arithmetic on integers, rational numbers, and floating-point numbers. It aims to provide the fastest possible arithmetic for all applications that need higher precision than is directly supported by the basic C types. Many applications use just a few hundred bits of precision ; but some applications may need thousands or even millions of bits. GMP is designed to give good performance for both, by choosing algorithms based on the sizes of the operands, and by carefully keeping the overhead at a minimum.

Random Number Functions. Sequences of pseudo-random numbers in GMP are generated using a variable of type `gmp_randstate_t`, which holds an algorithm selection and a current state. Such a variable must be initialized by a call to one of the `gmp_randinit` functions, and can be seeded with one of the `gmp_randseed` functions.

Number Theoretic Functions.
Function : `void mpz_nextprime (mpz_t rop, const mpz_t op)`. Set `rop` to the next prime greater than `op`. This function uses a probabilistic algorithm to identify primes. For practical purposes it's adequate, the chance of a composite passing will be extremely small.

Function : `void mpz_gcd (mpz_t rop, const mpz_t op1, const mpz_t op2)`
Set `rop` to the greatest common divisor of `op1` and `op2`. The result is always positive even if one or both input operands are negative. Except if both inputs are zero ; then this function defines gcd(0,0) = 0.

Function : `int mpz_invert (mpz_t rop, const mpz_t op1, const mpz_t op2)` Compute the inverse of `op1` modulo `op2` and put the result in `rop`. If the inverse exists, the return value is non-zero and `rop` will satisfy 0 <= `rop` < abs(`op2`) (with `rop` = 0 possible only when abs(`op2`) = 1, i.e., in the somewhat degenerate zero ring). If an inverse doesn't exist the return value is zero and `rop` is undefined. The behaviour of this function is undefined when `op2` is zero.

Exponentiation Functions.
Function : `void mpz_powm (mpz_t rop, const mpz_t base, const mpz_t exp, const mpz_t mod)`
Set `rop` to (`base` raised to `exp`) modulo `mod`. Negative exp is supported if an inverse base^-1 mod mod exists (see `mpz_invert`). If an inverse doesn't exist then a divide by zero is raised.

Conversion Functions.
Function : `char * mpz_get_str (char *str, int base, const mpz_t op)`
Convert `op` to a string of digits in base `base`. The base argument may vary from 2 to 62 or from -2 to -36.
For base in the range 2..36, digits and lower-case letters are used ; for -2..-36, digits and upper-case letters are used ; for 37..62, digits, upper-case letters, and lower-case letters (in that significance order) are used.

[1] Documentation : `https://gmplib.org/manual/`, `https://gmplib.org/gmp-man-5.0.1.pdf`

Appendix 3 An Introduction to the OpenSSL command line tool

First steps : OpenSSL is a C library that implements the main cryptographic operations like symmetric encryption, public-key encryption, digital signature, hash functions and so on... OpenSSL also implements obviously the famous Secure Socket Layer (SSL) protocol. OpenSSL is avaible for a wide variety of platforms. This tutorial[2] shows some basics funcionalities of the OpenSSL command line tool. You should able to check for the version.

```
> openssl version
OpenSSL 0.9.8zg 14 July 2015
```

OpenSSL has got many commands. Here is the way to list them :

```
> openssl list-standard-commands
asn1parse
ca
ciphers
crl
...
```

Secret key encryption algorithms : OpenSSL implements numerous secret key algorithms. To see the complete list :

```
>  openssl list-cipher-commands
aes-128-cbc
aes-128-ecb
aes-192-cbc
aes-192-ecb
aes-256-cbc
aes-256-ecb
...
```

If I want for example to encrypt the text "Vive l'ISIMA" with the AES algorithm using CBC mode and a key of 256 bits, I simply write :

```
> touch sample.txt
> echo "Vive l'ISIMA" > sample.txt
> openssl enc -aes-256-cbc -in sample.txt -out encrypted.bin
enter aes-256-cbc encryption password: hello
Verifying - enter aes-256-cbc encryption password: hello
```

The secret key of 256 bits is computed from the password. Note that of course the choice of password "hello" is really insecure. The output file `encrypted.bin` is binary. If I want to decrypt this file I write :

```
> openssl enc -aes-256-cbc -d -in encrypted.bin -pass pass:hello
Vive l'ISIMA
```

Public Key Cryptography : To illustrate how OpenSSL manages public key algorithms we are going to use the famous RSA algorithm. Other algorithms exist of course, but the principle remains the same.

Key generation : First we need to generate a pair of public/private key. In this example we create a pair of RSA key of 1024 bits.

[2] Extract from `http://users.dcc.uchile.cl/~pcamacho/tutorial/crypto/openssl/openssl_intro.html`

```
> openssl genrsa -out key.pem 1024
Generating RSA private key, 1024 bit long modulus
.++++++
...++++++
e is 65537 (0x10001)
```

The generated file has got both public and private key. Obviously the private key must be kept in a secure place, or better must be encrypted. But before let's have a look at the file `key.pem`. The private key is coded using the Privacy Enhanced Email (PEM) standard.

```
> cat key.pem
-----BEGIN RSA PRIVATE KEY-----
MIICXAIBAAKBgQDhXbl9APKkVn9qGrzLkc0tAVK15dNE+nd/EILMbliQ8aGa7dpF
...
guJThRXZ+IJLIxxSFBixTOVoOrzjECgyTIvVdAO8kWI=
-----END RSA PRIVATE KEY-----
```

The next line allows to see the details of the RSA key pair (modulus, public and private exponent between others).

```
>  openssl rsa -in key.pem -text -noout
Private-Key: (1024 bit)
modulus:                                              n = p * q
    00:e1:5d:b9:7d:00:f2:a4:56:7f:6a:1a:bc:cb:91:
    ...
    2d:58:60:62:5e:58:a9:3a:cf
publicExponent: 65537 (0x10001)                       e
privateExponent:                                      d
    39:fc:01:63:96:40:d7:e3:a1:78:cd:54:52:14:3b:
    ...
    2f:f7:36:b0:85:17:a1:61
prime1:                                               p
    00:f6:ce:a3:a6:57:8b:b3:15:7f:b0:d4:dc:c9:a4:
    ...
    80:0e:8b:31:c7
prime2:                                               q
    00:e9:c2:a3:d4:a7:eb:5d:e9:58:04:aa:40:b7:10:
    ...
    75:33:db:ae:b9
exponent1:                                            d mod (p-1)
    31:14:cc:f4:a2:8a:54:95:dc:eb:e4:98:f7:bc:b1:
    ...
    73:31:f9:37
exponent2:                                            d mod (q-1)
    00:d2:b0:a1:33:61:aa:0d:79:ec:e0:3f:87:96:b8:
    ...
    ce:5a:50:4d:d9
coefficient:                                          (1/q) mod q
    32:8a:0b:ea:a1:9f:cb:b9:1e:1b:6b:2e:4f:e6:d6:
    ...
    03:bc:91:62
```

The `-noout` option allows to avoid the display of the key in base 64 format [3]. Numbers in hexadecimal format can be seen (except the public exponent by default is always 65537 for 1024 bit

[3] https://en.wikipedia.org/wiki/Base64

keys) : the modulus, the public exponent, the private, the two primes that compose the modules and three other numbers that are use to optimize the algorithm.

So now it's time to encrypt the private key :

```
> openssl rsa -in key.pem -des3 -out enc-key.pem
writing RSA key
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
```

The key file will be encrypted using a secret key algorithm which secret key will be generated by a password provided by the user. In this example the secret key algorithm is triple des (3-des). The private key alone is not of much interest as other users need the public key to be able to send you encrypted messages. So let ?s extract the public from the file `key.pem`.

```
> openssl rsa -in key.pem -pubout -out pub-key.pem
```

Encryption : We are ready to perform encryption.

```
> openssl rsautl -encrypt -in <input_file> -inkey <key> -out <output_file>
```

Where :

- `input_file` is the file to encrypt. This file must no be longer that 116 bytes =928 bits because RSA is a block cipher, and this command is low level command, i.e. it does not do the work of cutting your text in piece of 1024 bits (less indeed because a few bits are used for special purposes) ;
- `key` File that contains the public key. If this file contains only the public key (not both private and public), then the option `-pubin` must be used ;
- `output_file` the encrypted file.

To decrypt only replace `-encrypt` by `-decrypt`, and invert the input/output file as for decryption the input is the encrypted text, and the output the plain text.