Sécurité des Systèmes d'Information
Mandatory TP 2 - RSA and SHA-2

06 Novembre 2019

Submit your TP on **Moodle**, as Python 3 files **.py**, before **November 27, 2019 at 4pm (16h00)**.

Your code should be **commented**.

Goal

The goal of this TP is to implement both RSA ans the hash function SHA-2.

RSA

First, for RSA, you need to implement the following algorithms :

1. The Fast Exponentiation algorithm
2. A Prime Number Generator (using the Fermat Primality Test)
3. Euclide's Extended Algorithm (to compute inverses)
4. The RSA Key Generator
5. The RSA Encryption, using the previously generated keys
6. The RSA Decryption, with the most efficient way, using the Chinese Remainder Theorem.

Fast Exponentiation

The goal is to easily compute $a^k \mod n$ for big numbers and exponents.

For example, let's compute manually for $3^{42} \mod 25$. We compute every exponent that is a power of 2 :

$3^1 \mod 25 = 3$
$3^2 \mod 25 = 9$
$3^4 \equiv 3^2 \cdot 3^2 \equiv 9 \cdot 9 \equiv 81 \mod 25 = 6$
$3^8 \equiv 6 \cdot 6 \equiv 36 \mod 25 = 11$
$3^{16} \equiv 121 \mod 25 = 21$
$3^{32} \equiv 21 \cdot 21 \equiv 441 \mod 25 = 16$

And since $42 = 32 + 8 + 2$, we have :

$$3^{42} \equiv 3^{32} \cdot 3^8 \cdot 3^2 \equiv 16 \cdot 11 \cdot 9 \equiv 1584 \mod 25 = 9$$

Prime Number Generator

First, generate a big random number n.

Then, to see if it is prime, we use the Fermat primality test :

Choose a random number a between 2 and $n-1$. Compute $a^{n-1} \mod n$. If it is not 1, n is not prime, and we try again with a new n randomly chosen. If it is 1, n is "probably prime".
We repeat this process by choosing a randomly again and again to improve the probability of n to be prime.

Euclide's Extended Algorithm

We have two numbers $a \geq b$. This algorithm will compute $r = pgdc(a, b)$ and s, t such that $s \cdot a + t \cdot b = r$ (i.e. the coefficients of Bézout's identity). If a and b are relatively prime, then s is the inverse of a modulo b, and t is the inverse of b modulo a.

Then for RSA, if $a = \phi(n)$ and $b = e$, we can use r to see if e and $\phi(n)$ are prime, and if it is the case, then t is the inverse of $e \mod \phi(n)$.

Here is the algorithm (note that $\div$ is the **integer division**) :

```
$r_0$ := a;
$r_1$ := b;
```

```
$s_0 := 1;$
$s_1 := 0;$
$t_0 := 0;$
$t_1 := 1;$
$q_1 := r_0 \div r_1;$

repeat until $r_{i+1} == 0$
$r_{i+1} := r_{i-1} - q_i * r_i;$
$s_{i+1} := s_{i-1} - q_i * s_i;$
$t_{i+1} := t_{i-1} - q_i * t_i;$
$q_{i+1} := r_i \div r_{i+1};$
end repeat;

return $r_i, s_i, t_i;$
```

Warning : The algorithm gives us the coefficients of Bézout's identity, which can be positive or negative numbers. So you need to verify that $0 \leq t \leq a-1$ (and $0 \leq s \leq b-1$).

Chinese Remainder Theorem

We only consider the special case with two congruences :

If we have $p, q \in \mathbb{Z}$, $p, q > 1$ et $pgdc(p, q) = 1$, then the following equation system :

$$x = a \mod p$$
$$x = b \mod q$$

has a unique solution modulo $p \cdot q$, given by $x = a \cdot q \cdot (q^{-1} \mod p) + b \cdot p \cdot (p^{-1} \mod q)$

(Note that some parts of this formula, like $(q^{-1} \mod p)$ and $(p^{-1} \mod q)$, can be pre-computed at the same time as the key generation, to avoid computing them each time we decrypt.)

SHA-256

You will implement SHA-2, or more specifically SHA-256, one of the versions of SHA-2.

First Step : Padding

First, you need to pad your message (of length L) to a multiple of 512 bits, following this algorithm :

- Add one "1" to your message,
- Add K "0" to your message, with $K \in \mathbb{N}$ the smallest integer such that $L + 1 + K + 64$ is a multiple of 512,
- Add L (length of the initial message) as a 64 bits integer to your message.

Second Step : Merkle-Damgard structure

SHA-256 uses the Merkle-Damgard structure to create a fixed length output :

- Slice your message into 512 bit blocks,
- You have an initial value IV of 256 bits (given later in the TP),
- Give the 256 IV and the 512 bit block to SHA-256, which will compute a 256 bit output value h,
- Then take this value h and the next 512 bit block of the message, and give them to SHA-256, which will output a new 256 bit cipher,
- Repeat this process until all 512 bit blocks of the message have been used,
- The last 256 bit output is your hash.

Third Step : The SHA-256 one-way compression function

Now we just need the SHA-256 "Box", which is a one-way compression function taking two blocks, one of 512 bits (from the message) and one of 256 bits (The previous cipher, or IV in the first step).

This box works with 32 bit words. Additions are made modulo 2^{32}

- First, we need to create a 64 words (each one of 32 bits) list. Let us slice the 512 bit block from the message into 32 bit words, which will be $W_1, W_2, ..., W_16$ (the first 16 words).

  Then, the other words are defined with this pseudo-code (Be careful as there's both shifts and cyclic shifts : **rightrotate** is a rotation (cyclic shift) of the word to the right, **rightshift** is a shift (non cyclic) to the right) :

  ```
  For i from 17 to 64 :
  $s_0 = (w_{i-15}$ rightrotate 7) xor ($w_{i-15}$ rightrotate 18) xor ($w_{i-15}$ rightshift 3)
  ```

$$s_1 = (w_{i-2} \text{ rightrotate } 17) \text{ xor } (w_{i-2} \text{ rightrotate } 19) \text{ xor } (w_{i-2} \text{ rightshift } 10)$$

$$w_i = w_{i-16} + s_0 + w_{i-7} + s_1$$

- Then, we will do 64 iterations of a compression function. That compression function needs 3 things :
  An initial 256 bit hash, noted as eight 32 bit words $h_0, h_1, h_2, h_3, h_4, h_5, h_6, h_7$ (this is either the previous hash, or IV in the first round),
  And two 32 words for each iteration $i \in \{1, 2, ..., 64\}$. The first one is W_i,
  And the second one is a constant K_i (that constant is different for each of the 64 iterations, and these are given later in the TP).
  This will give you eight 32 bit words, noted a, b, c, d, e, f, g, h.
  The compression function is explained in the next subsection.
- Finally, after the 64 iterations defined in the next section, add the 256 bits (as eight 32 bit words, each time modulo 2^{32}) obtained with the initial hash (or IV) :

$$newh_0 = h_0 + a \mod 2^{32}$$
$$newh_1 = h_1 + b \mod 2^{32}$$
$$...$$
$$newh_7 = h_7 + h \mod 2^{32}$$

Last Step : The Compression Function

And finally, we just need to define the compression function :

We start with the previous hash or IV as our initial eight words, noted a,b,c,d,e,f,g,h. The W_i and K_i are the words previously defined from the Third Step (see next subsection for constants K_i)

Then, we follow this algorithm (rightrotate is again the rotation to the right (cyclic shift), and **"xor", "not"** and **"and"** are bitwise operators):

All additions (+) are one again modulo 2^{32}.

For i from 1 to 64 :

$$X_1 := (e \text{ rightrotate } 6) \text{ xor } (e \text{ rightrotate } 11) \text{ xor } (e \text{ rightrotate } 25)$$
$$CH := (e \text{ and } f) \text{ xor } ((\text{ not } e) \text{ and } g)$$
$$X_2 := (a \text{ rightrotate } 2) \text{ xor } (a \text{ rightrotate}13) \text{ xor } (a \text{ rightrotate } 22)$$

$$\begin{aligned}
MAJ &:= (a \text{ and } b) \text{ xor } (a \text{ and } c) \text{ xor } (b \text{ and } c) \\
temp1 &:= h + X_1 + CH + K_i + W_i \\
temp2 &:= X_2 + MAJ \\
h &:= g \\
g &:= f \\
f &:= e \\
e &:= d + temp1 \\
d &:= c \\
c &:= b \\
b &:= a \\
a &:= temp1 + temp2
\end{aligned}$$

The last a,b,...,h you get are the ones used in the last point of the Third Step.

SHA-256 Constants

You need two constants, IV (256 bits) and the 64 words of 32 bits in K. These constants are already in the python file **"SHAConstants.py"** :

IV (256 bits) is defined in hexadecimal as the eight 32 bit words :

$$\begin{pmatrix} h0 := 0x6a09e667 \\ h1 := 0xbb67ae85 \\ h2 := 0x3c6ef372 \\ h3 := 0xa54ff53a \\ h4 := 0x510e527f \\ h5 := 0x9b05688c \\ h6 := 0x1f83d9ab \\ h7 := 0x5be0cd19 \end{pmatrix}$$

(which, fun fact, are defined by the first 32 bits of the fractional parts of the square roots of the first 8 primes (from 2 to 19)).

The K_i (32 bit words) are defined in hexadecimal as :
K[0..63] :=
0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5, 0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174, 0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da, 0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967, 0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85, 0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070, 0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3, 0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2

(fun fact part two, these are the first 32 bits of the fractional parts of the cube roots of the first 64 primes (from 2 to 311))

Exercices

Exercice 1 : RSA

Implement RSA, with all the algorithms described in this TP, and big enough numbers (generate prime numbers which are at least 2^{20}).

Exercice 2 : SHA-2

programmation : Implement the hash function SHA-2.