# Information Systems Security
# Mandatory TP 4 - RSA Implementation and Theoretical Questions

December 1st, 2021

**Submit both** :

- **Your Python files .py** for the RSA implementation,
- **And your report in .pdf** for the theoretical questions,

Before **Tuesday, December 21st, 2021 at 11:59 PM (23h59)**.

Your code should be **commented**, and all your answers should be **justified**.

## Goal

This TP has two parts : one practical and one theoretical.

First, in the practical part, you will implement RSA as an asymmetrical encryption system.

Then, the theoretical part is a collection of shorts questions about various topics from the course.

## Practical Part : RSA

For this practical part, you will implement RSA encryption and decryption.

### RSA

To implement RSA, you will need these three algorithms :

1. **A prime number generator**, using **Fermat primality test**

2. **Fast exponentiation**
3. **Euclide extended algorithm**

With these tools, you can implement a RSA Key generator, and then the encryption and decryption. Implement these three algorithms yourself (For example, you can use the pow() function with two arguments, but not with three (as it does fast exponentiation already)).

We'll use big numbers : prime numbers of minimum 513 bits for p and q in RSA (i.e. $n$ of minimum 1025 bits)

Consider the message to be a simple value of 1024 bits or less.

**Even with this kind of big numbers, everything in your code should be instantaneous if the algorithms are implemented correctly (fast exponentiation should be done correctly, and used everywhere when we exponentiate modulo something). Only prime number generation may take a few seconds maximum if you're really unlucky with random generation.**

**Prime number generator**

To generate keys, you need to generate big prime numbers.

The method used is to generate a random number, and then verify if it is prime, and try again until we find one that is prime.

We will use Fermat's primality test : Let n be the chosen random number, we choose another random number $a$, such that $1 < a < n$, and we compute $a^{n-1} \mod n$ (using fast exponentiation of course). If the result is not equal to 1, then for sure, n is not a prime number (and we try another $n$). If it's equal to 1, then n is "probably" prime. To be sure, we repeat this test enough times, changing $a$ each time. If n passes this test many times, we can be confident that it is prime.

You can consider that repeating this process 20 times is enough.

**Fast Exponentiation**

To compute efficiently big exponents in a modular environment, we use what is called fast exponentiation :

Let's say we want to compute $3^{42} \mod 25$. The fast exponentiation idea is to use the exponents which are a power of two :

$3^1 \mod 25 = 3$
$3^2 \mod 25 = 9$

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

Then, we can write the power as $42 = 32 + 8 + 2$, with powers of two, and we can easily compute :

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

## Euclide's extended algorithm

This algorithm is used to find the decryption exponent $d$ for RSA.

$d$ is the inverse of $e$ modulo $\phi(n)$ ($ed \mod \Phi(n) = 1$ implies $x^e d \mod n = x^(k\phi(n) + 1) \mod n = x$).

With this algorithm, for two integers a and b, $a \geq b$, we can obtain $r = pgdc(a, b)$ and $s, t$ such that $s \cdot a + t \cdot b = r$ (these are called Bezout coefficients). If a and b are co-prime, then s is the inverse of a modulo b, and t is the inverse of b modulo a.

So in this case, we will use it with $a = \phi(n)$ and $b = e$. We will find the inverse t of $e \mod \phi(n)$. Then $d = t \mod Phi(n)$ (we need this to verify that $t$ is such that $0 < t < Phi(n)$, as Bezout may return negative numbers).

Here is the algorithm : ($\div$ 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 $r_i, s_i, t_i$;
```

(Note that when $r_{i+1} = 0$, you have to break out of this loop instantaneously, as finishing the loop leads to computing $q_{i+1}$ which has a zero division)

## Theoretical part

Answer the following questions. Justify your answers.

### Exercise 1 : Entropy and Asymetrical Ciphers

1. Given the ASCII 8-bit alphabet (we consider every character to be printable and usable), compute the entropy of choosing randomly a key for Caesar cipher.
2. Given the ASCII 8-bit alphabet (we consider every character to be printable and usable), compute the entropy of choosing randomly a key for monoalphabetical substitution.
3. Given the ASCII 8-bit alphabet (we consider every character to be printable and usable), compute the entropy of choosing randomly a key of length k for a Vigenere cipher.
4. From now on, we consider a deck of 54 cards : we have four colors ($\spadesuit, \heartsuit, \diamondsuit, \clubsuit$) with 13 cards each (numbered from 1 to 10, plus the Jack, Queen and King), and two Jokers.
   What would be the entropy of drawing a suite of 5 cards from the deck one by one, with replacement (i.e. we put the card back in the deck and shuffle again before drawing another card) ?
5. And what would be the entropy of a suite of 5 cards drawn from the deck without replacement ?
6. And if we only keep hands where we have a hand of 5 cards with one Joker and one card from each color, what is the entropy ?

### Exercise 2 : Symmetrical and Asymmetrical Encryption

1. Of the three following functions $f, g$ and $h$, which one would be usable as an encryption function for a block cipher with 128 bit block size ? Why ?

   Here, x is a 128 bit block, that we can write as $x = x_1 \parallel x_2$ that are its two halves of 64 bits, and k is a 128 bit key that we can express as two halves ($k_1$, $k_2$) the same way :

   - $f(x) = SHA - 256(AES(x, k))$
   - $g(x) = DES(x_1, k_2) * DES(x_2, k_1)$ *($*$ is the multiplication)*
   - $h(x) = DES(x_1 \oplus x_2, k_1) \cdot DES(x_1 \oplus k_1, k_2)$ *($\cdot$ is the concatenation)*
2. You get the following keys for RSA : (n=143, e=21), (n=147, e=37), (n=161, e=35).
   Which one of these three keys is a valid RSA key ? Compute d for that valid key.
3. Given the prime number p=23, build a set of keys for an ElGamal encryption.

## Exercise 3 : Tryout for a MAC

We want to use AES in CBC mode as a MAC : the AES 128 bit key $k$ is our secret MAC key, and to create a hash, we do an XOR of all the cipher blocks from the AES encryption. We consider IV=0.

That means we have one entry, which is a message m, one key k, and the output is a 128 bit hash (that is created by XORing all ciphers from AES) :

$AES - MAC(m) = AES - MAC(m_1 || m_2 ... || m_n) = c_1 \oplus c_2 \oplus ... \oplus c_n$.

Where the $m_i$ are the blocks of the message, and the $c_i$ are the resulting ciphers from the block cipher encryption (reminder, we use CBC mode).

Note that the ciphers are not sent : we only send the message and the resulting MAC (thus the ciphers are not known, they can only be computed by the sender and the receiver as they both have the key)

1. If we suppose this system has second pre-image resistance and is secure, what computing power should we need to find a second pre-image (in other terms, how many times should we expect to try before finding a second pre-image) ?
2. If we suppose this system has collision resistance and is secure, what computing power should we need to find a collision (in other terms, how many times should we expect to try before finding a second pre-image) ?
3. Now, let us suppose we change the key and use a 256 bit key instead. How would that change the two previous results ? Why ?
4. You have access to one pair (message, MAC). Can you use this to falsify the MAC for another message (i.e. build another pair (message,MAC) without the key ?
5. Now suppose you have access to one pair (message, ciphertext). Can you falsify the MAC for another message ?

## Exercise 4 : Trying signature schemes with RSA

Let (d,n) be the RSA private key of A, and (e,n) the corresponding public key. A uses these keys to sign messages, by computing $y = m^d \mod n$, and sending (m,y) to B.

To validate the signature, B computes $x = y^e \mod n$, and accepts if and only if $x = m$.

1. Find a message easy to falsify when you possess only the public key.

2. B chooses a number x, and computes $x^{-1} \mod n$. Then he asks A to sign $m = x^e k \mod n$. What will B be able to obtain, without A noticing ? What do we call this process ?

3. Show that you can falsify the signature of a given message m (m different from 0, 1, p and q) with only two chosen signatures, i.e. you can choose two messages, obtain the signatures, and then falsify the signature for m.

4. Alice decides to use the previous signature scheme, and she decides it's easier to use the same pair of keys to sign and to encrypt/decrypt messages.
   Why is that a bad idea, and what does Alice give access to when she signs messages ?

5. Now, let us try another signature scheme : this time, the signature is $s = h(m)^d \mod n$, where $h(m) = SHA - 256(m)$, and we send the pair $(m, s)$ to Bob. Do we still have all the previous problems ?