Information Systems Security Mandatory TP 3 - RSA and pseudo Station-to-Station

November 5th, 2025

Submit on **Moodle** your Python 3 files **.py**, before **Tuesday, November 25th, 2025 at 11:59 pm (23h59)**.

Your code should be **commented**.

Goal

The goal of this TP will be :

- To fully implement the RSA encryption system, including algorithms to generate big prime numbers, fast exponentiation, and euclide's extended algorithm.
- Use RSA for signatures, with the help of the hashing function SHA-256 (for SHA-256, use the hashlib library and its sha256() function).
- And then use that signature function to implement a simplified version of the Station-to-Station protocol allowing to generate session keys.

We'll use big numbers : prime numbers of minimum 512 bits for RSA (i.e. n of minimum 1024 bits), and a safe prime number of minimum 512 bits for Station-to-station exchange.

Even with this kind of big numbers, almost everything in your code should be instantaneous if the algorithms are implemented correctly (doing fast exponentiation correctly is especially important, as correctly applying the mod n every time it is needed is what allows us to keep computations from becoming too long).
Only the prime number generations should take some time (standard

prime generation can easily take a few seconds, and safe prime generation can easily take a few minutes), but these two parts are not necessary since the primes and safe prime are given (see last section on results and examples).

RSA

First, you need to implement RSA encryption. We will need three main algorithms to do that :

1. **Fermat primality test**, to determine if numbers are prime (normally used to generate p and q, instead you will use it to verify that the p,q you are given are indeed prime)
2. **Euclide extended algorithm**, to compute the decryption exponent d
3. **Fast exponentiation**, for the encryption/decryption itself.

With these tools, we can implement RSA from key generation to the encryption and decryption. Implement these three algorithms yourself. For example, that means you can use the pow() function from python with two arguments, but not with three (which does fast expo) as you have to implement fast exponentiation yourself (that being said, it is a good idea to use that function affter your finished your own fast expo function, to compare and verify if it works correctly).
As a reminder, here's a short review of what is needed for RSA keys : p and q are big prime numbers (generally 512 or 1024 bits minimum, here we will use a p and q with 512 bits), $n = p \cdot q$, and then we also need to choose the encryption exponent, $e \in [2, \varphi(n) - 2]$ which also has to be prime with $\varphi(n)$ (i.e. $gcd(e, \varphi(n)) = 1$), and to compute the inverse exponent d (such that $e \cdot d \equiv 1 \mod \varphi(n)$).

Fermat's primality test

To generate RSA keys, we need to generate big prime numbers. The method used is to generate a random number, and then verify if it is prime with Fermat's primality test, and try again until we find one that is prime.

In this case, we give you the prime numbers, and you will use the test only to verify that these numbers are indeed prime numbers.

We will use Fermat's primality test : Let n be the number we want to verify is prime or not. We choose a 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. 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 :

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

Note that in practice, we need to apply the modulo each time we do a multiplication in this last formula (to prevent numbers from becoming too big).

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 $x^{ed} \mod n = x^{k \cdot \phi(n)+1} \mod n = x$).

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 $r_{i+1} == 0$
$r_{i+1} := r_{i-1} - q_i \cdot r_i$;
```

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

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

RSA Signature

Now that the RSA encryption system is done, we'll use it to sign messages. Which means, we're using our private key (n, d) to sign messages, and anyone can use our public key (n,e) to verify the signature.

We will sign with a signature with appendix, i.e. the signature is added as an appendix to the message. We define the signature S as follows :

S = SHA-256$(message)^d \mod n$

So the message is hashed with SHA-256, and then we sign this hash with our private key.

To verify the signature S, the receiver will compute the SHA-256 hash of the message m, and apply the public key exponent e to S. Both should give the same result, i.e. the SHA-256 hash of m. If the result is the same, the receiver accepts the signature.

You should use the hashlib library for sha256 (see more details in the given file "TP3examples.py").

Simplified "Station-to-Station" Key generation

Now, we'll use this signature scheme to implement some simplified version of the Station-to-Station Key generation protocol. This protocol allows two people to create symmetric session keys in a public, non-confidential environment.

In this case, we consider that Alice and Bob were able to exchange their authentic public keys for RSA signature beforehand.

Then, the protocol is the following, with S_A being Alice's signature (as defined earlier using RSA signature keys and SHA-256), and similarly S_B is Bob's signature.

You will also need a way to generate a safe prime (i.e. a number p such that $\frac{p-1}{2}$ is also prime) for the pseudo Station-to-Station protocol. The easiest way to do that is to generate a prime number, check if $\frac{p-1}{2}$ is also prime, and if not try again until you find a safe prime.

1. (Initialisation) We choose a big "safe prime" number p (512 bits minimum), and a generator α of $\mathbb{Z}_p^*$. Both numbers are public. These can be generated by Alice as she starts the protocol, or given by a trusted third party (in this TP, they are given to you, see examples).
2. $A \rightarrow B : \alpha^x \mod p$ (Alice generates a secret $x \in [2, p-2]$, and sends to Bob $\alpha^x \mod p$).
   In this case, Alice also sends p and α (everything is sent in clear).
3. $A \leftarrow B : \alpha^y \mod p,\ S_B(\alpha^x \mod p, \alpha^y \mod p, K))$, i.e. Bob chooses a secret $y \in [2, p-2]$, computes the session key K, $K = (\alpha^x)^y \mod p$, and signs a bloc that contains both public parts α^x and α^y as well as the key K.
   Note : these three elements inside the signature are concatenated. Treat each of these three values as a binary, and concatenate all three to obtain one big binary. Then convert that back to an integer (and then to bytes to give it to the sha256 function, see python file with examples)
4. $A \rightarrow B : S_A(\alpha^y \mod p, \alpha^x \mod p, K)$ (Alice computes the key K, and then she has all three parts of the message, so she can verify Bob's signature. If the signature is correct, then she send a similar signature to Bob, but with both public parts reversed).
   Note : the three elements inside the signature are concatenated, see the previous step.
5. And finally, Bob verifies if Alice's signature is correct.

(Note that this is not the Station-to-Station protocol, just a simplified version. In a standard Station-to-Station, we would not include K in the signature, but rather encrypt the signature with a symmetric encryption scheme (for ex. AES) with key K)

To implement this part, just do as if you were Alice and Bob, and have different functions "take turns" between both.

Examples and expected results

You should print all the following examples and results to show your code works as intended.

All the given values for the examples are in the python file "TP3examples.py". This file also contains in the comments all the expected results for each value you have to compute.

Here are, for each part of the TP, the list of what's given to you and what you have to compute and show :

For RSA encryption :

- p_A, q_A, and $n_A = p_A \cdot q_A$ are both prime numbers and n used in Alice's keys, these are given.
- The encryption exponent e_A for Alice's public key is also given.
- And the message m_1 is also given.

And with these, you are expected to :

- Show that p and q are indeed prime with Fermat's primality test.
- Compute the decryption exponent $d_A = ...$
- Compute the ciphertext $c_1 = ...$
- And then show that the decryption works and finds the original message m_1.

For RSA signature :

- p_B, q_B, n_B, e_B and d_B (all the elements of Bob's key) are given.
- And the message to sign m_2 as well.

For which you are expected to give the following results :

- Compute the signature S = ...
- And then show that the verification of the signature indeed validates that the signature is correct.

And finally for station-to-station :

All of these are given :

- p_A, q_A, n_A, e_A and d_A for the RSA signature keys of Alice (same pair of keys from the first example for RSA encryption),
- p_B, q_B, n_B, e_B and d_B for the RSA signature keys of Bob (same pair of keys from the second example for RSA signature),
- the safe prime number P_{safe},
- the generator α,
- the random choice x for Alice
- the random choice y for Bob

And you are expected to use these to run all the steps of the protocol between Alice and Bob (messages from Alice and Bob and vice-versa, in order), to compute :

- $A \rightarrow B$: Alice sends to Bob $\alpha^x \mod p = ...$
- $A \leftarrow B$: Bob responds to Alice with $\alpha^y \mod p = ...$
  and his signature $S_B(\alpha^x \mod p, \alpha^y \mod p, K)) = ...$
  and also computes the session key $K = (\alpha^x)^y \mod p = ...$
- $A \rightarrow B$: Alice then computes the session key, then verifies Bob's signature, and if it is correct, computes her own signature that the sends to Bob : $S_A(\alpha^y \mod p, \alpha^x \mod p, K) = ...$
- And finally Bob's verifies Alice's signature.
- Include all these message exchanges, and also show what is the session key that they both computed.