RSA cryptosystem

Security and secret in the codification of information

Èrik Campobadal Forés

Documentation and source codes presented for
the final project of cryptography

ICT Systems Engineering
Polytechnic University of Catalonia
Catalonia, Spain
January 2019

Contents

List of Figures

Chapter 1

Introduction

The goal of this document is to provide insight on how to implement a RSA cryptosystem using **Python 3**. The system will be composed of two diferent modules named `RSA` and `Key`. Containing the RSA class and the RSA's Key class respectively.

The system's architecture will be the following:

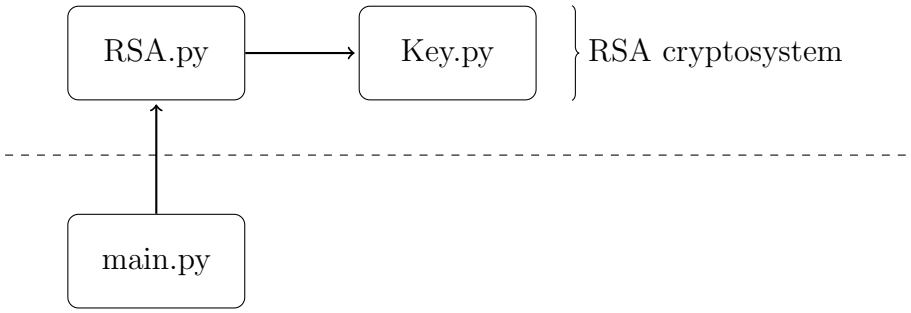

Figure 1.1: System architecture

The system is going to be built using a library pattern and in fact it will allow any program to use it's contents to encrypt / decrypt messages. Consider RSA is also a slow algorithm compared to symmetric algorithms.

1.1 Rivest–Shamir–Adleman and public key cryptography

RSA (Rivest–Shamir–Adleman) is a cryptosystem that was one of the first public-key based system. That means that RSA have a public key and a private key used to share information between a sender and a receiver.

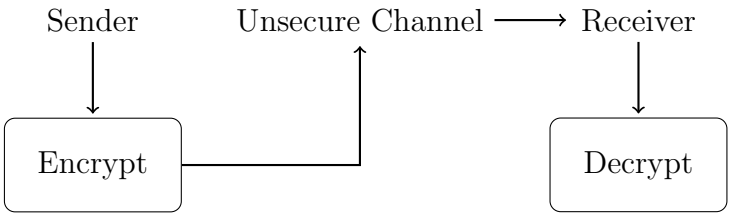

Figure 1.2: Transmission scheme

That said, the sender must first encrypt the message using the public key of the receiver. Once the message is encrypted, it can be shared in a public channel that can be insecure by the pressence of evasdroppers. The message can only be decrypted by the receiver by using it's private key. This can be achieved since the public and the private keys share a mathematical relation. To further explain this, the following explanation corresponds to the RSA cryptosystem.

1.1.1 Key Generation

To generate a RSA key you need to follow the following steps:

- Choose two prime numbers p and q. Those numbers should be randomly chosen and should have a similar length in bits.
- Calculate $n = p \cdot q$. The result n will be used as the modulus for both keys (private, public) and it's length in bits designates the **key length**.
- Calculate $\Phi = (p-1) \cdot (q-1)$. Euler's properties are used to calculate Φ.
- Calculate e given $e < \Phi$ and e to be coprime with Φ.
- Determine d given $e \cdot d \equiv 1 \mod \Phi$ (Modular multiplicative inverse)

The public key will be: `(n, e)` and the private key will be `(n, d)`

1.1.2 Encryption

Both, the sender and the receiver should share their public key in the public channel (it does not matter if anyone know it). To demostrate an ecryption by using RSA let's consider the public key of the receiver as: `(n, e)` and it's private key `(n, d)`.

If a sender wants to send the receiver a message it needs to encrypt the message by using the following mathematical formula:

$$c \equiv m^e \mod n$$

Consider c to be the cipher text generated by the encryption.

1.1.3 Decryption

To decrypt, the same formula applies, but this time changing the e with d and m with c.

$$p \equiv c^d \mod n$$

$$p \equiv (m^e)^d \mod n$$

$$p \equiv m^{e \cdot d} \mod n \quad \rightarrow \quad p \equiv m^1 \mod n$$

$$\boxed{p \equiv m \mod n}$$

Chapter 2

Implementation

This chapter will include the implementation details of how to implement the given RSA Cryptosystem using **Python 3**. The implementation, as mentioned earlier, will be done as a library containing two modules `Key.py` and `RSA.py`

2.1 RSA Key

The RSA Key module (`Key.py`) is the one responsible of the key creation for the RSA cryptosystem.

The Key class will be used inside the RSA class to determine the current used key. Keep in mind that the Key is able to automatically execute the steps mentioned in the key generation process given two integers p and q. This is done in the constructor and further saved into the class attributes.

The class atributes and methods are defined below. The method names contains the attributes needed to work. They also have an brief explanation of what they do.

p, q, n, phi, e, d	The basic storage attributes for the RSA key as defined in the introduction.
__init__(self, p, q)	Basic constructor to generate a key given p and q.
__generate_e_and_d(self)	Method to generate an e and d given Φ and e.
__generate_e(self, phi)	Generate an e based on Φ.
__extended_euclidean_algorithm(self, a, b)	The Extended Euclidean Algorithm given $a = e$ and $b = \Phi$ to compute d considering $d \cdot e \equiv$ mod Φ.
length(self)	Returnns the length of the key.
public(self)	Returns the public key.
private(self)	Returns the private key.
print(self)	Prints the whole key (Public + Private).

```
from random import randint
from fractions import gcd

class Key:

    # The p and q used by the key
    p, q = 0, 0
    # n = p * q
    n = 0
    # phi = (p - 1) * (q - 1)
    phi = 0
    # The e and d used by the key. d == e**-1 mod phi
    e, d = 0, 0

    def __init__(self, p, q):
        # Basic assignments.
        self.p = p
        self.q = q
        self.n = p * q
        self.phi = (p - 1) * (q - 1)
```

```
21          # Calculate e and d.
22          self.__generate_e_and_d()
23
24      def __generate_e_and_d(self):
25          gcd = 0
26          self.e = self.__generate_e(self.phi)
27          gcd, x, y = self.__extended_euclidean_algorithm(self.e,
            ↪  self.phi)
28          while gcd != 1 or x < 0:
29              self.e = self.__generate_e(self.phi)
30              gcd, x, y =
                ↪  self.__extended_euclidean_algorithm(self.e,
                ↪  self.phi)
31          self.d = x
32
33      def __generate_e(self, phi):
34          e = randint(2, phi - 1)
35          while gcd(e, phi) != 1: e = randint(2, phi - 1)
36          return e
37
38      def __extended_euclidean_algorithm(self, a, b):
39          # Initial x setup. x1 is the last calculated x,
40          # and it's initialized to 1
41          x1, x = 1, 0
42          # Initial y setup (The same as x)
43          y1, y = 1, 0
44          # While b is diferent from 0
45          while b:
46              # Calculate the current quotient out of a / b.
47              quotient = a // b
48              # Set the view value of x and update the last x.
49              x, x1 = x1 - quotient * x, x
50              # The same for y.
51              y, y1 = y1 - quotient * y, y
52              # Update a and v values.
53              a, b = b, a % b
54          # Return the tyiple value g, x, y
55          return a, x1, y1
```

```
56
57      def length(self):
58          return len(format(self.n, 'b'))
59
60      def public(self):
61          return {
62              'n': self.n,
63              'e': self.e
64          }
65
66      def private(self):
67          return {
68              'p': self.p,
69              'q': self.q,
70              'phi': self.phi,
71              'd': self.d
72          }
73
74      def print(self):
75          print({
76              'length': self.length(),
77              'public': self.public(),
78              'private': self.private()
79          })
```

2.2 RSA Cryptosystem

The RSA Class module (`RSA.py`) is the one responsible of the RSA cryptosystem.

The RSA class will be responsible of generating keys, encrypting and decrypting messages. The key generation will be dispatched to the Key class, responsible of this things. The RSA will only store and use the key.

The class atributes and methods are defined below. The method names contains the attributes needed to work. They also have an brief explanation of what they do.

key	Stores the RSA key class used to encrypt / decrypt messages.
__fast_modular_exponentiation(self, a, z, n)	Method to calculate $x \equiv a^z \mod n$.
generate_key(self, p, q)	Method to generate a key given p and q
encrypt(self, plain_text)	Encrypts a message (Might be a number or a string)
decrypt(self, cipher_text)	Decrypts an encrypted number or sequence of numbers (string).

```
from functools import reduce
from RSA.Key import Key

class RSA:

    # Store the RSA key currently in use.
    key = None

    def __fast_modular_exponentiation(self, a, z, n):
        # Transform the exponent into binary (powers of 2).
        binary = format(z, 'b')
        # Reverse the binary value of the exponent.
        binary_rev = binary[::-1]
        # Save the powers of 2 needed to later use
        powers_of_two = []
        # Add the power of 2^i.
        for i,e in enumerate(binary_rev):
            # Only add if the digit is 1. (1 << i == 2**i)
            if e == '1': powers_of_two.append(1 << i)
        # Calculate mod C of the powers of two.
        mod_of_powers, current_exp = [], 1
        while current_exp <= powers_of_two[-1]:
            if current_exp in powers_of_two:
                mod_of_powers.append(a**current_exp % n)
            current_exp *= 2
        # We multiply the items in the list to get the result
        ↪  and do result mod n.
```

```
27          return reduce(lambda x, y: x * y, mod_of_powers) % n
28
29      def generate_key(self, p, q):
30          self.key = Key(p, q)
31          return self.key
32
33      def encrypt(self, plain_text):
34          if isinstance(plain_text, str):
35              result = []
36              for letter in plain_text:
37                  result.append(self.encrypt(ord(letter)))
38          else:
39              result = self.__fast_modular_exponentiation(
40                  plain_text % self.key.n, self.key.e, self.key.n
41              )
42          return result
43
44      def decrypt(self, cipher_text):
45          if isinstance(cipher_text, list):
46              result = ""
47              for element in cipher_text:
48                  result += chr(self.decrypt(element))
49          else:
50              result = self.__fast_modular_exponentiation(
51                  cipher_text, self.key.d, self.key.n
52              )
53          return result
```

Chapter 3

Library Usage

To use the library you simply need to import it as a regular `Python 3` library and use the RSA class to access the given public methods. That said, the following example may ilustrate a sample usage case.

```
1  from RSA.RSA import RSA
2
3  system = RSA()
4
5  key = system.generate_key(61, 53)
6  key.print()
7
8  message = 'Cryptography'
9  cipher_text = system.encrypt(message)
10 plain_text = system.decrypt(cipher_text)
11 print(f'Original: {message}')
12 print(f'Cipher text: {cipher_text}')
13 print(f'Plain text: {plain_text}')
14 print(f'Correct: {message == plain_text}')
```

Chapter 4

References

Khan Academy. (2019). *Fast modular exponentiation*. [online] Available at: https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/fast-modular-exponentiation [Accessed 12 Jan. 2019].

Loyola University Chicago. (2019). *Extended Euclidean Algorithm*. 1st ed. [ebook] Boston, Massachusetts: Dr. Andrew Harrington, p.2. Available at: https://anh.cs.luc.edu/331/notes/xgcd.pdf [Accessed 12 Jan. 2019].

Wikipedia. (2019). *RSA (cryptosystem)*. [online] Available at: https://en.wikipedia.org/wiki/RSA_(cryptosystem) [Accessed 12 Jan. 2019].

Python. (2019). *Python 3.7.2 Documentation*. [online] Available at: https://docs.python.org/3/ [Accessed 12 Jan. 2019].