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:

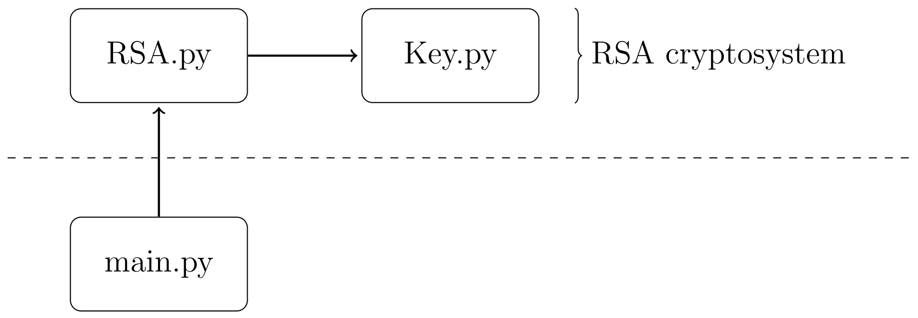


Figure 1.1: System architecture

The system is going to be built using a library pattern and in fact it will allow any program to use its contents to encrypt / decrypt messages. Consider that RSA is 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.

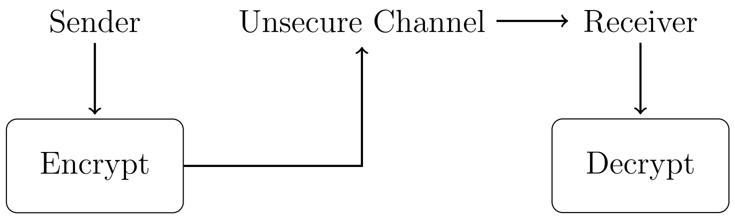


Figure 1.2: Transmission scheme

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 eavesdroppers. The message can only be decrypted by the receiver by using its private key. This can be achieved since the public and the private keys share a mathematical relation. To further understand this concepts, 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 its length in bits designates the **key length**.
- Calculate $\Phi(n) = (p-1) \cdot (q-1)$. Euler's properties are used to calculate Φ.
- Choose an e that satisfies $e < \Phi$ and by having e to be coprime with Φ.

- Calculate 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

The sender should share their public key in the public channel (it does not matter if anyone knows it). The receiver would only need to share its public key in case of digital signature. To demostrate an ecryption by using RSA let's consider the public key of the receiver as: `(n, e)` and its private key `(n, d)`. The algorithm of *Fast Modular Exponentiation* is used to calculate this operation.

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. Consider c to be the cipher text generated by the encryption and this time, by using d (the private key) we can recover the original message.

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

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

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

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

The end result m is the original message that was sent.

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 = None, q = None)	Basic constructor to generate a key given p and q. None values generate a prime from [50, 200]
__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).

```python
from random import randint
from fractions import gcd
from sympy import isprime, randprime

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 = None, q = None):
        # Determine if p and q are defined.
        if not p: p = randprime(50, 200)
        while not q or p == q: q = randprime(50, 200)
        # Check if p and q are primes
        if not isprime(p) or not isprime(q):
            raise Exception("The provided p and q are invalid.")
        # Basic assignments.
        self.p = p
        self.q = q
        self.n = p * q
        self.phi = (p - 1) * (q - 1)
        # Calculate e and d.
        self.__generate_e_and_d()

    def __generate_e_and_d(self):
        gcd = 0
        self.e = self.__generate_e(self.phi)
        gcd, x, y = self.__extended_euclidean_algorithm(self.e, self.phi)
        while gcd != 1 or x < 0:
            self.e = self.__generate_e(self.phi)
            gcd, x, y = self.__extended_euclidean_algorithm(self.e, self.phi)
        self.d = x

    def __generate_e(self, phi):
        e = randint(2, phi - 1)
        while gcd(e, phi) != 1: e = randint(2, phi - 1)
        return e

    def __extended_euclidean_algorithm(self, a, b):
        # Initial x setup. x1 is the last calculated x,
        # and it's initialized to 1
        x1, x = 1, 0
        # Initial y setup (The same as x)
        y1, y = 1, 0
        # While b is diferent from 0
        while b:
            # Calculate the current quotient out of a / b.
            quotient = a // b
            # Set the view value of x and update the last x.
            x, x1 = x1 - quotient * x, x
            # The same for y.
            y, y1 = y1 - quotient * y, y
            # Update a and v values.
            a, b = b, a % b
        # Return the tyiple value g, x, y
        return a, x1, y1

    def length(self):
        return len(format(self.n, 'b'))

    def public(self):
        return {
            'n': self.n,
            'e': self.e
        }

    def private(self):
        return {
            'p': self.p,
            'q': self.q,
            'phi': self.phi,
            'd': self.d
        }

    def print(self):
        print({
            'length': self.length(),
            'public': self.public(),
            'private': self.private()
        })
```

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 = None, q = None)	Method to generate a key given p and q. If no value is provided, they are generated by Key class
encrypt(self, plain_text, e = None, n = None)	Encrypts a message (Might be a number or a string)
decrypt(self, cipher_text, d = None, n = None)	Decrypts an encrypted number or sequence of numbers (string).

```python
from functools import reduce
from RSACryptosystem.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.
        return reduce(lambda x, y: x * y, mod_of_powers) % n

    def generate_key(self, p = None, q = None):
        self.key = Key(p, q)
        return self.key

    def encrypt(self, plain_text, e = None, n = None):
        if not e: e = self.key.e
        if not n: n = self.key.n
        if isinstance(plain_text, str):
            result = []
            for letter in plain_text:
                result.append(self.encrypt(ord(letter), e, n))
        else:
            result = self.__fast_modular_exponentiation(
                plain_text % n, e, n
            )
        return result

    def decrypt(self, cipher_text, d = None, n = None):
        if not d: d = self.key.d
        if not n: n = self.key.n
        if isinstance(cipher_text, list):
            result = ""
            for element in cipher_text:
                result += chr(self.decrypt(element, d, n))
        else:
            result = self.__fast_modular_exponentiation(
                cipher_text, d, n
            )
        return result
```

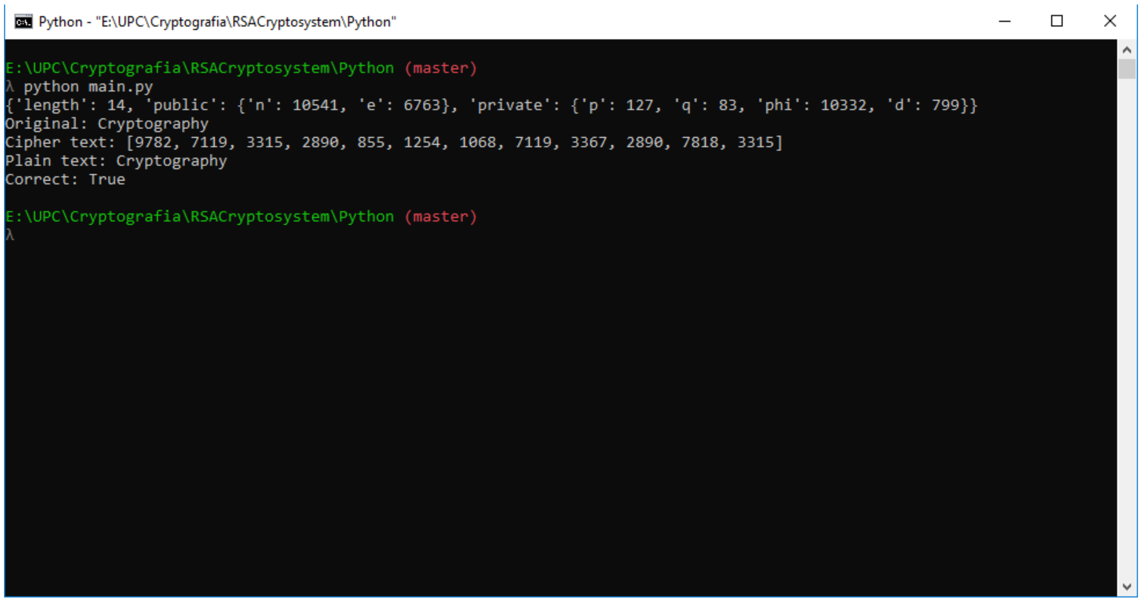
Chapter 3

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.

```python
from RSACryptosystem import RSA

# Initiate the RSA cryptosystem
system = RSA()

# You may also provide two parameters
# p, q to the generate_key() function.
key = system.generate_key()
# This will print to the screen a key representation.
key.print()
# The message to be encrypted
message = 'Cryptography'
print(f'Original: {message}')
# The resulting cipher text of
# encrypting the original message
cipher_text = system.encrypt(message)
print(f'Cipher text: {cipher_text}')
# The resulting plain text after
# decrypting the cipher text recieved
plain_text = system.decrypt(cipher_text)
print(f'Plain text: {plain_text}')
```

```python
# Simple check to see if they match
print(f'Correct: {message == plain_text}')
```



```
E:\UPC\Cryptografia\RSACryptosystem\Python (master)
λ python main.py
{'length': 14, 'public': {'n': 10541, 'e': 6763}, 'private': {'p': 127, 'q': 83, 'phi': 10332, 'd': 799}}
Original: Cryptography
Cipher text: [9782, 7119, 3315, 2890, 855, 1254, 1068, 7119, 3367, 2890, 7818, 3315]
Plain text: Cryptography
Correct: True

E:\UPC\Cryptografia\RSACryptosystem\Python (master)
λ
```

Figure 3.1: Example library output

Chapter 4

Simulation

This simulation exposes a real live simulation to play around with the library.

The following simulation creates a sockets server and clients can connect to chat around. During chatting, special commands may be used to setup the RSA cryptosystem and encrypt messages to a given destination. All the available commands are:

K: e n	Send the public key to the public channel and others register it (should not be used).
K: A	Sends the already generated key to the public channel (should be used).
E: x m	Encrypts m with the given saved public key x. x is printed to the screen when the public key is registed.
m	Sends an insecure message m to the public channel.

The following script is the chat server. It's reponsible of handling all the requests. It recieves all the commands and sends them to all users in the channel. It's basically a broadcaster.

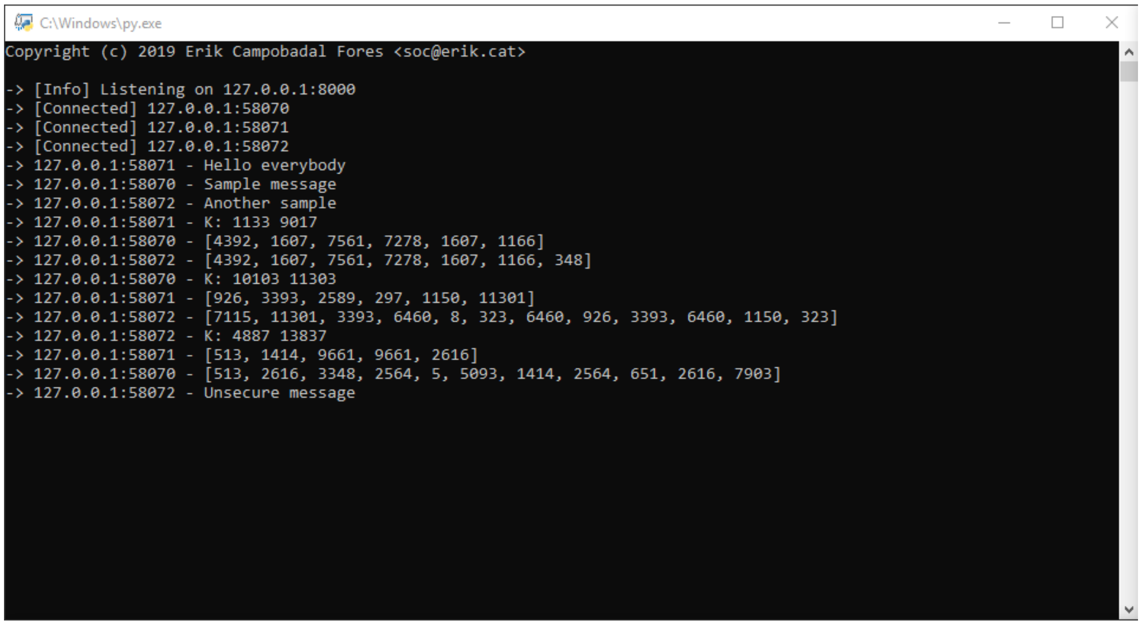
```
# Copyright (c) 2019 Erik Campobadal Fores <soc@erik.cat>
# Distributed under the MIT License

```

```python
from sys import exit
from socket import socket, SHUT_RDWR
from threading import Thread

# Start a socket instance
sk = socket()
# Bind the socket to a given IP and PORT
sk.bind(('127.0.0.1', 8000))
# Start listening for incomming connections
sk.listen(1)
# Helper function
def log(message, arrow = True):
    print(('-> ' if arrow else '') + message)

# Output a message
print('Copyright (c) 2019 Erik Campobadal Fores
↪  <soc@erik.cat>\n')
log('[Info] Listening on 127.0.0.1:8000')

# Stores the current connections.
connections = []

def entry(conn, addr):
    global connections
    connections.append(conn)
    log('[Connected] ' + addr[0] + ':' + str(addr[1]))
    while True:
        try:
            data = conn.recv(1024)
            if not data:
                raise Exception("Client disconnected")
            msg = str(addr[0]) + ':' + str(addr[1]) + ' - ' +
            ↪  data.decode("utf-8")
            log(msg)
            for c in connections:
                if c != conn: c.send(bytes(msg, 'utf-8'))
        except:
            conn.shutdown(SHUT_RDWR)
            log('[Disconected] ' + str(addr[0]) + ':' +
             ↪  str(addr[1]))
            if conn in connections:
                connections.remove(conn)
            exit()

# Start listening for connections.
while True:
    try:
        conn, addr = sk.accept()
        thread = Thread(target = entry, kwargs = {'conn': conn,
         ↪  'addr': addr})
        thread.daemon = True
        thread.start()
    except:
        log('[Closing] Aborting sockets...')
        sk.shutdown(SHUT_RDWR)
        exit()
```

In the other side, the following script corresponds to the chat client, where you connect to the server and it's available to send all the listed commands.

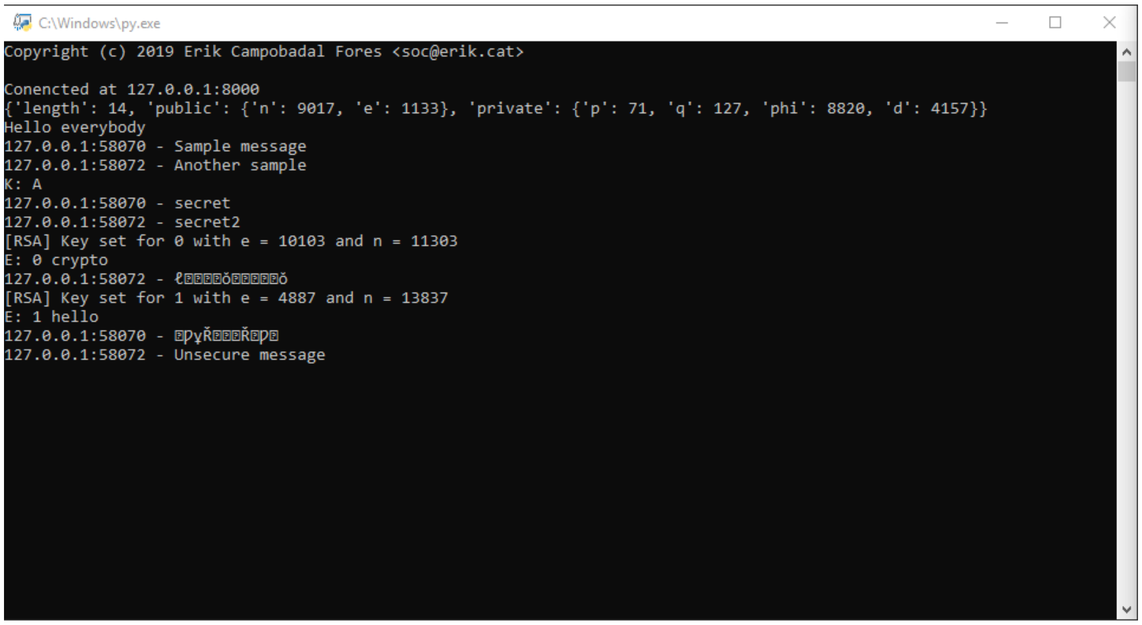
```python
# Copyright (c) 2019 Erik Campobadal Fores <soc@erik.cat>
# Distributed under the MIT License
from RSACryptosystem import RSA
from threading import Thread
from socket import socket, SHUT_RDWR
from time import sleep

# Start a socket instance
sk = socket()
# Connect to a given IP and PORT
sk.connect(('127.0.0.1', 8000))
# Output a message
print('Copyright (c) 2019 Erik Campobadal Fores
 ↪  <soc@erik.cat>\n')
print("Conencted at 127.0.0.1:8000")
# RSA cryptosystem
```

```
rsa = RSA()
# Print the refence key
rsa.generate_key().print()
# Stores the RSA keys
keys = []

# Listener function for the thread
def listener():
    global sk
    while True:
        try:
            data = sk.recv(1024)
            if not data:
                raise Exception("Client disconnected")
            msg = data.decode("utf-8")
            raw = msg[msg.find(' - ') + 3:]
            if raw[0] == '[' and raw[len(raw) - 1] == ']':
                cipher = raw[1:len(raw) - 1].replace(' ', '')
                cipher = [int(i) for i in cipher.split(',')]
                print(msg[:msg.find(' - ') + 3] +
                 ↪  rsa.decrypt(cipher))
            elif raw[:3] == 'K: ':
                key = raw[3:]
                e = int(key[:key.find(' ')])
                n = int(key[key.find(' ') + 1:])
                keys.append((e, n))
                print(f'[RSA] Key set for {len(keys) - 1} with
                 ↪  e = {e} and n = {n}')
            else: print(msg)
        except Exception as e:
            print(e)
            sk.shutdown(SHUT_RDWR)
            exit()

# Start the listener thread.
Thread(target = listener).start()

while True:
    num = input()
    if num[:4] == 'K: A':
        num = f'K: {rsa.key.e} {rsa.key.n}'
    if num[:3] == "E: ":
        begin = num.find(' ', 4)
        index = int(num[3:begin])
        if index >= 0 and index < len(keys):
            # Encrypt the message
            num = str(rsa.encrypt(num[begin + 1:],
             ↪  keys[index][0], keys[index][1]))
    sk.sendall(bytes(num, 'utf-8'))
    sleep(1)
```



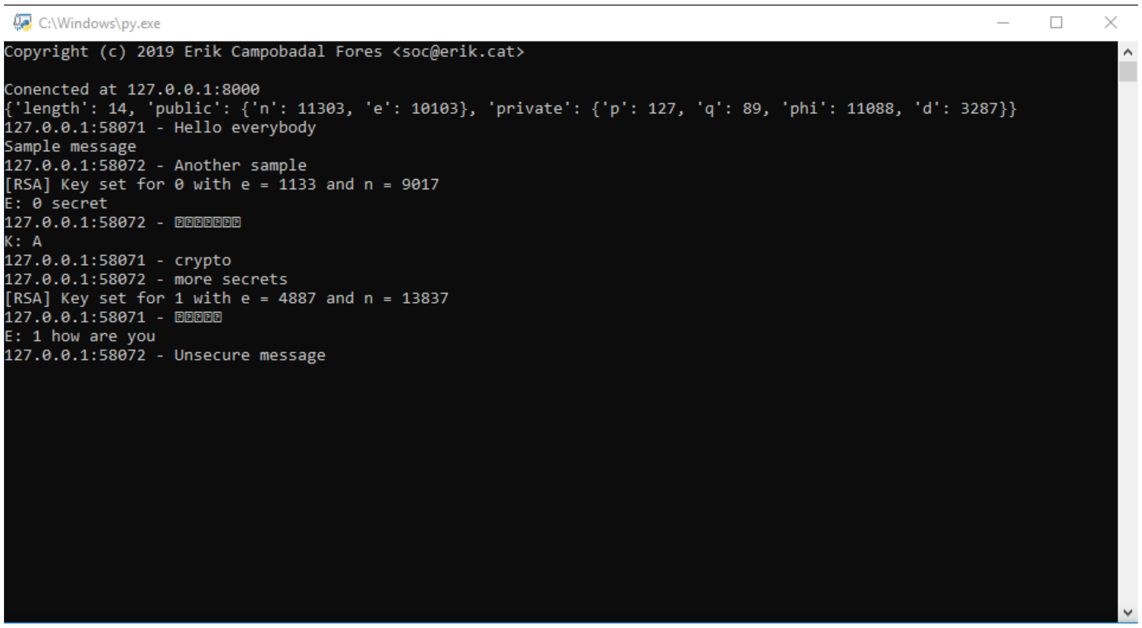
```
C:\Windows\py.exe
Copyright (c) 2019 Erik Campobadal Fores <soc@erik.cat>

-> [Info] Listening on 127.0.0.1:8000
-> [Connected] 127.0.0.1:58070
-> [Connected] 127.0.0.1:58071
-> [Connected] 127.0.0.1:58072
-> 127.0.0.1:58071 - Hello everybody
-> 127.0.0.1:58070 - Sample message
-> 127.0.0.1:58072 - Another sample
-> 127.0.0.1:58071 - K: 1133 9017
-> 127.0.0.1:58070 - [4392, 1607, 7561, 7278, 1607, 1166]
-> 127.0.0.1:58072 - [4392, 1607, 7561, 7278, 1607, 1166, 348]
-> 127.0.0.1:58070 - K: 10103 11303
-> 127.0.0.1:58071 - [926, 3393, 2589, 297, 1150, 11301]
-> 127.0.0.1:58072 - [7115, 11301, 3393, 6460, 8, 323, 6460, 926, 3393, 6460, 1150, 323]
-> 127.0.0.1:58072 - K: 4887 13837
-> 127.0.0.1:58071 - [513, 1414, 9661, 9661, 2616]
-> 127.0.0.1:58070 - [513, 2616, 3348, 2564, 5, 5093, 1414, 2564, 651, 2616, 7903]
-> 127.0.0.1:58072 - Unsecure message
```

Figure 4.1: Socket server



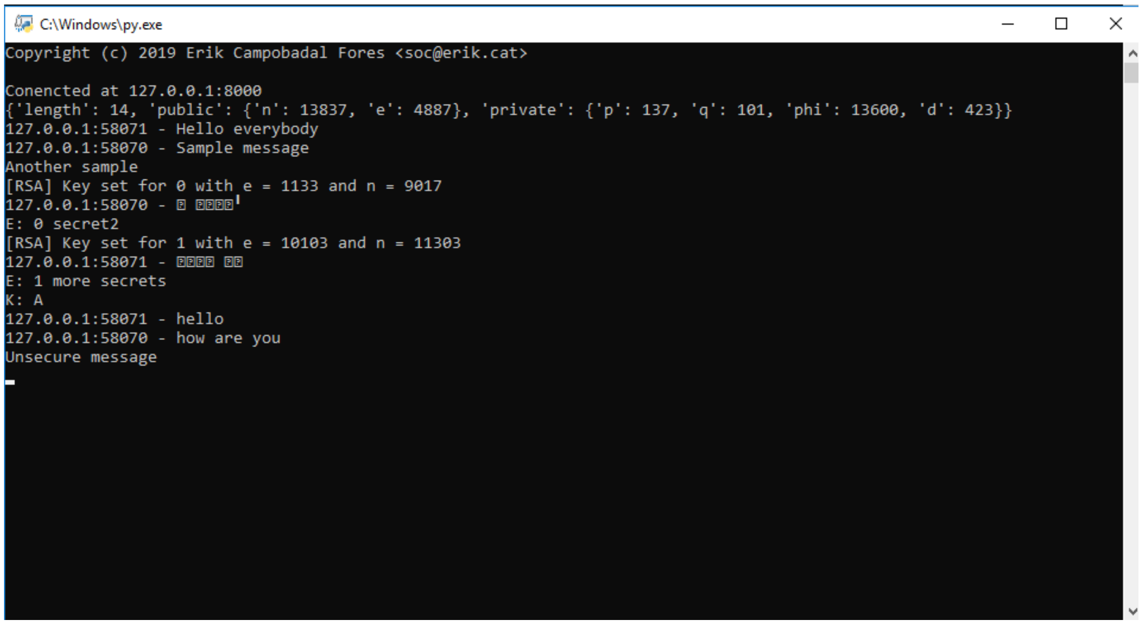
```
Copyright (c) 2019 Erik Campobadal Fores <soc@erik.cat>

Conencted at 127.0.0.1:8000
{'length': 14, 'public': {'n': 9017, 'e': 1133}, 'private': {'p': 71, 'q': 127, 'phi': 8820, 'd': 4157}}
Hello everybody
127.0.0.1:58070 - Sample message
127.0.0.1:58072 - Another sample
K: A
127.0.0.1:58070 - secret
127.0.0.1:58072 - secret2
[RSA] Key set for 0 with e = 10103 and n = 11303
E: 0 crypto
127.0.0.1:58072 - ℓ⍰⍰⍰⍰ŏ⍰⍰⍰⍰⍰⍰ŏ
[RSA] Key set for 1 with e = 4887 and n = 13837
E: 1 hello
127.0.0.1:58070 - ⍰ƿɣŘ⍰⍰⍰Ř⍰ƿ⍰
127.0.0.1:58072 - Unsecure message
```

Figure 4.2: Socket client 1



```
Copyright (c) 2019 Erik Campobadal Fores <soc@erik.cat>

Conencted at 127.0.0.1:8000
{'length': 14, 'public': {'n': 11303, 'e': 10103}, 'private': {'p': 127, 'q': 89, 'phi': 11088, 'd': 3287}}
127.0.0.1:58071 - Hello everybody
Sample message
127.0.0.1:58072 - Another sample
[RSA] Key set for 0 with e = 1133 and n = 9017
E: 0 secret
127.0.0.1:58072 - ⍰⍰⍰⍰⍰⍰⍰
K: A
127.0.0.1:58071 - crypto
127.0.0.1:58072 - more secrets
[RSA] Key set for 1 with e = 4887 and n = 13837
127.0.0.1:58071 - ⍰⍰⍰⍰⍰
E: 1 how are you
127.0.0.1:58072 - Unsecure message
```

Figure 4.3: Socket client 2



```
Copyright (c) 2019 Erik Campobadal Fores <soc@erik.cat>

Conencted at 127.0.0.1:8000
{'length': 14, 'public': {'n': 13837, 'e': 4887}, 'private': {'p': 137, 'q': 101, 'phi': 13600, 'd': 423}}
127.0.0.1:58071 - Hello everybody
127.0.0.1:58070 - Sample message
Another sample
[RSA] Key set for 0 with e = 1133 and n = 9017
127.0.0.1:58070 - ? ????'
E: 0 secret2
[RSA] Key set for 1 with e = 10103 and n = 11303
127.0.0.1:58071 - ???? ??
E: 1 more secrets
K: A
127.0.0.1:58071 - hello
127.0.0.1:58070 - how are you
Unsecure message
```

Figure 4.4: Socket client 3

Chapter 5

Conclusions

RSA Cryptosystems require more that what's explained here to be secure. As you may guess, the same character is encrypted with the same ciphertext every time, leading to a deterministic cryptosystem. However, by using **Optimal asymmetric encryption padding** you can avoid this problem. Deterministic systems are subject to frequency analysis that may crack the system much faster.

The Key length must be quite large for the system to be secure nowadays, leading to a very big n in bit length. Nowadays standards go around key sizes of **2048 bits**.

Chapter 6

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].