A data encryption application: Development Proposal

Diogo Vilas Boas

Lusofona University of Porto, Portugal
diogovilasboas98@gmail.com

Abstract. This paper follows the development of a particular software which uses a cryptographic algorithm to encrypt and decrypt data. Here, we will us e RSA (Rivest Shamir Adleman). The first part of the paper will discuss the mathematics involved in the creation of the keys and in the encryption and decryption of the data. While in the second part will be looking into the integration of mathematics involved when developing the software. This part will highlight some code of the project and provide an explanation of the methodology used. Concluding the paper with the results of performance tests done to the software and with a commentary on those results.

Keywords: RSA, Encryption, Decryption, Cryptography, Software, Public Key.

1 Introduction

Cryptography has as aim enabling users to maintain communication in a secure way over a channel which is not reliable to guarantee privacy and/or authenticity. In the ideal world, we wouldn't be able to decipher or make modifications without the permission of the person who encrypted the message [1].

It is not known for sure when the cryptography appeared, but there is some evidence of it in the earliest types of writing. In some way, every one, early on, wanted to have a secret language where they could communicate without the information being comprised. This was particularly useful in times of diplomacy and in times of war [2].

Normally, throughout history, you would have a password or a method, if it was used backward, you could decrypt the message that was encrypted. This wasn't the best way to secure a message because you would have to share that password or method so that other people could decrypt it [2]. The problem comes with this exchange. If it doesn't assure the secrecy of the password or method, encrypting it be comes useless.

In spite of Diffie Hellman method being secure to use it to communicate, it didn't implement digital signatures which means the receiver didn't know who or what the source of the encrypted messages was. So, when Ronald Rivest, Adi Shamir, and Richard Adleman read the paper of Diffie Hellman, they realized this flaw and started searching for a mathematical solution to solve this problem. And this was how RSA was born [3].

So, since RSA is such a good algorithm, it was proposed to me to develop one implementation of a software that uses RSA to encrypt and decrypt text, in order to learn a little bit more about the RSA and it’s challenges when anyone tries to implement it.

2 Mathematics of the encryption and decryption

RSA uses mathematics to encrypt and decrypt messages. Therefore, in this section, it will be focused on how those mathematic principles work and explain them.

2.1 Basic Concepts

Prime Numbers are numbers that are only divisible for themselves and by 1.They are like building blocks for numbers,almost all numbers can be created by multiplying prime numbers [4].

Euclidean Division is the process that divides two integers, which results in a quotient and remainder. It normally follows this structure [4]:

$$\frac{dividend}{divisor} = quotient + remainder$$

Modulo Operation also known by modulus gives the remainder of Euclidean division. After giving two positive numbers, it returns a number that ranges between 0 and a unit below of the divisor [4].

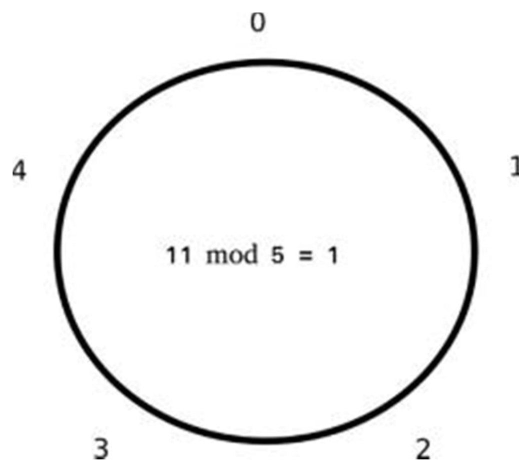

Fig.1. Modulo operation between 11 and 5.

$$dividend \bmod divisor = dividend - (quotient * divisor)$$

Looking at figure 1 and equation 2, we can notice that as the dividend increases the module numbers will repeat. In the case of figure 1 as we increase the the dividend of the module of 5, the module will repeat 0, 1, 2, 3, 4, in this exact order[4].

Greatest common divisor(gcd) says that for certain integers a and b are bigger than zero, than gcd(a,b) returns the largest integer that a and b are divisible by[4].

Euler's theorem says that if n and a are coprime then

$$a\varphi(n) \equiv 1 mod n$$

where φ(n) is Eule's totient function. The notation is explained in the article modular arithmetic. Which returns the number of cop rimes of n. Two numbers are coprime if the greatest common divisor of both of them is 1 [4].

Chinese Remainder Theorem(CRT) is a theorem that solves simultaneous linear congruence's, where the moduli are coprime [5].

The first step is to multiply the modili resulting a number N [5]:

$$N = x1 * x2 * x3 ... xk$$

Secondly, for each i= 1, 2 ... k, we calculate [5]:

$$yi = \frac{N}{xi}$$

Then, for the same i 's, we calculate [5]:

$$zi \equiv yi^{-1} mod xi$$

And finally:

$$x = \sum ai * yi * zi$$

Where x is solution of the system of congruence's, and x mod N is the only solution modulo N [5].

2.2 Generating Keys

RSA usually needs 5 numbers, p, q, N, e and d [4]. The p and q are really huge prime numbers where the ideal size would be 1024 bits. So the easiest way to generate this numbers is by generating a random number and then verifying if that number is prime, using primality test. If it is not a prime, repeats the process until it is prime [4].

The N is the result of the multiplication of p and q [4].

The e is a number between 1 and φ(N) such that [4]:

$$gcd(e, \varphi(N)) = 1$$

The d is the solution of [4]

$$d \equiv e^{-1} mod N$$

Now that we have the 5 numbers, the public key is the N and e and the private key is the d [4].

2.3 Encrypting and Decrypting

To encrypt message we need to transform the text or file to an array ofnumbers so that we can use RSA [4]. After doing this, we have our message(m) in a number form and then the encrypted cipher (c) is the result of [4]

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

3 Development Proposal

My development proposal uses python as main and only programming language and besides that uses PyQt5 which is a graphics interface framework. Now in the next sections, it will be explained the code from the project and what was the though process behind it.

3.1 Generating Keys

```
def generate_keypair(self):
    #a
    p = random.randint(2, 9007199254740991)
    q = random.randint(2, 9007199254740991)
    while not self.prime(p):
        p += 1
    while not self.prime(q) and p!=q:
        q += 1
    #b
    n = p * q
    #c
    phi = (p - 1) * (q - 1)
    #d
    e = random.randint(1, phi)
    g = self.gcd(e, phi)

    while g != 1:
        e = random.randint(2, phi)
        g,y,x = self.gcd(e,phi)
    #e
    d = self.multiplicative_inverse(e, phi)
    #f
    self.p= p
    self.q= q
    self.e= e
    self.d= d
    self.n= n
    #g
    self.publicKey=self.arrayToBase64([e,n])
    #h
    self.dP = d % (p-1)
    self.dQ = d % (q-1)
```

Fig.2. Function generate_keypair

The image above(Fig. 2.) represents the part of the code that generates the public and private keys. But there is many steps to being able to generate them, so it is divided by letters. From the letter a to letter b, two huge prime numbers(p and q) are created by randomly picking up a number between 2 and 9007199254740991 which is , using the random.randint, and then it verifies if it is a prime by using the primality algorithm that is represented by the function prime. Then, if it is not a prime number, it increments one until it is prime.

```
def prime(self, num):
    if num == 2:
        return True
    if num < 2 or num % 2 == 0:
        return False
    for n in range(3, int(num**0.5)+2, 2):
        if num % n == 0:
            return False
    return True
```

Fig.3. Function prime

From the letter b to c, the N is calculated by multiplying the previous prime numbers. From the letter c to d, we setup the phi variable by using the formula.

$$\varphi = (p - 1)(q - 1)$$

From the letter d to e, the e is calculated by choosing a number between 1 and phi(N), and then checking if the gcd(e,phi(N))=1. If it is not, it repeats the process until it is true.

```
def gcd(self,a, b):
    if a == 0:
        return (b, 0, 1)
    else:
        g, y, x = self.gcd(b % a, a)
        return (g, x - (b // a) * y, y)
```

Fig. 4. Function gcd

From the letter e to f, the d, the private key, is calculated by choosing a number that is a multiplicative inverse of e which is calculated with the function multiplicative_inverse.

```
def multiplicative_inverse(self,a, m):
    g, x, y = self.gcd(a, m)
    if g != 1:
        raise Exception('modular inverse does not exist')
    else:
        return x % m
```

Fig 5. Function multiplicative_inverse

From the letter f to g, we attach the numbers to the class, so that we can access them anywhere inside the class RSA(the class that is responsible to apply the RSA algorithm).

From the letter g to h, we create the public key that is the N and the e in base 64, using the function arrayToBase64.

```
def arrayToBase64(self,array):
    length= len(array)
    string=""+array[0].__str__()
    if(length>1):
        for i in range(1,length):
            string+=" "+array[i].__str__()
    return base64.b64encode(string.encode('utf-8'))
```

Fig.6. Function arrayToBase64

Lastly, from the h to the end of the function, the Chinese Remainder Theorem is setup. This will be explained in greater detail in the Decoding section.

3.2 Encrypting

```
def encrypt(self, publickey, plaintext):
    #a
    key, n = self.base64ToNumber(publickey)
    #b
    cipher = [pow(ord(char), key, n) for char in plaintext]
    #c
    cipher=self.arrayToBase64(cipher)
    return cipher
```

Fig.7. Function encrypt

The function above, encrypt (Fig. 7.) , it encrypts a string given a public key. From a to b, it takes the N and the key which is the e,that we mention above, using the function base64ToNumber because the public key is in base 64.

```
def base64ToNumber(self,msg):
    numbers = base64.b64decode(msg).decode('utf-8')
    numbers = ciphertext.split(' ')
    length=len(numbers)
    for i in range(length):
        numbers[i]=int(numbers[i])
    return numbers
```

Fig.8. Function base64ToNumber

From b to c, we encrypt each letter by turning the letter into a number, using the ASCII converter included in the python, and then raising it to the key. After this, it is done the modulus of N, and the result is a cipher encrypted using the RSA algorithm.

Lastly, from c to the end of the function, it converts the array of ciphers to base64 and returns it.

3.3 Decrypting

```
def decrypt(self,ciphertext):
    #a
    ciphertext=self.base64ToNumber(ciphertext)
    #b
    key=self.d
    n=self.n
    #c
    plain=''
    for i in range(len(ciphertext)):
        m1= pow(ciphertext[i], self.dP, self.p)
        m2= pow(ciphertext[i], self.dQ, self.q)
        qInv = (1/self.q) % self.p
        h = qInv * (m1 - m2)
        m = m2 + (h * self.q)

        plain += chr(int(m))
    return ''.join(plain)
```

Fig.9. Function decrypt

The image above show us the function decrypt, that decrypts a ciphertext given.

From a to b, it transforms the string ciphertext to numbers using the function base64ToNumber.

From b to c, it gets the private key and part of the public key and puts it in local variables.

From c to end of the function, it decrypts each numner by implementing the Chinese Remainder Theorem which divides the d into two smaller numbers(dQ and dP from generate_keypair) and then uses Garner's formula. So that, the operations be faster than powering it to d and moduling it to N [6].

4 Implementation and Results

In this section, the tests that were made to the software are going to be shared as their results. And then, I will give my interpretation about them.

4.1 First test

The first test was to encrypt the message "Hello World!!" using the public key of another window of the software. And decrypt it using the other window.

Fig.10. First part of the test

Fig.11. Second part of the test

The images above show the result of the test which successfully passed.

4.2 Second test

The second test consisted in picking the RSA class and adding a main, where it created to RSA objects(a and b) and then randomly generating a random message. And then encrypt the message using b public key using a. After this decrypts using b. Then checks if it the message decrypted is equal to the message originated. If it is, prints “yes”, it is not prints “not”. It repeats this process 100 times.

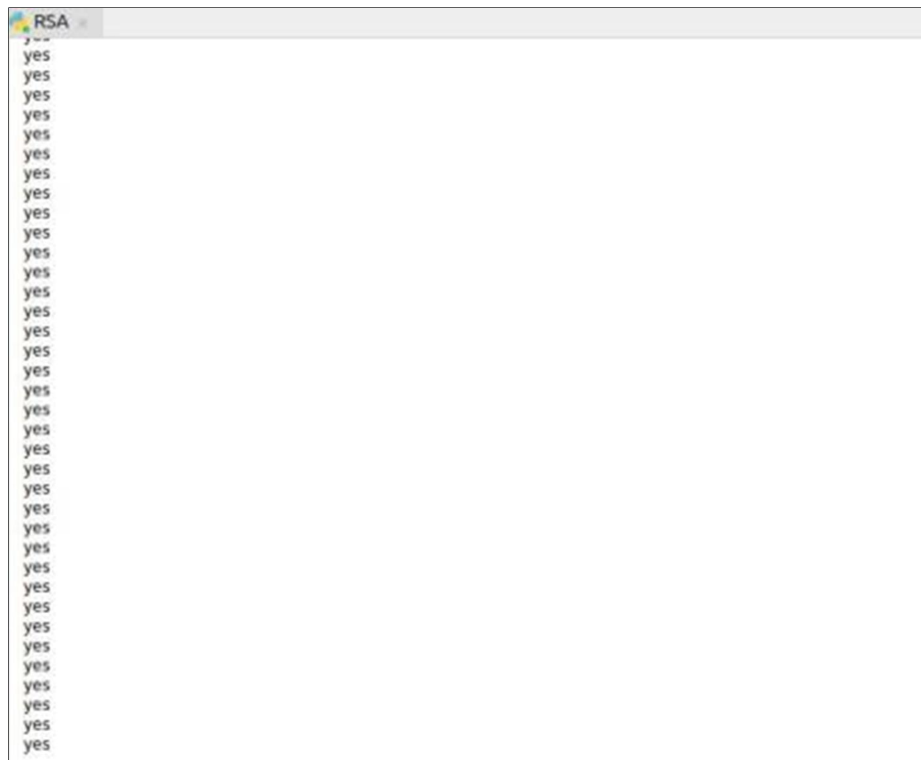

Fig.12. Part of the results of the second test

The image above shows part of the output of the test. The output was all “yes” which means the software passed the test successfully and it shows to be reliable.

4.3 Final Notes

The software seems to be pretty reliable when it comes to encrypting and decrypting. Although, it takes some time launch (5 seconds) because it has to generate the keys, it works in real time after launching. So we can conclude that this software was a success.

5 Conclusion

While in development, many adversities were faced. One of them was choosing the right programming language, where big numbers needed to be supported. If not, it wouldn't be possible making any operations with the numbers generated.

Another adversity was the mathematics of RSA. Although they are simple, they are not very intuitive. Which you need to spend some time understanding the mathematics involved. That is one of the reasons that I put an entire chapter on the mathematical concepts before explaining the method.

The last adversity, it was bugs in code. Although it is predictable that you will have bugs when developing a software, it does not mean they do not give some headaches. So, those where

treated with patience. Which means, many features I would have loved to put in code needed to be cut off, so that it was possible to finish the project in time.

References

1. Coron, J.: "What is cryptography?". IEEE Security & Privacy Magazine Volume 4 issue 1, (2016), doi:10.1109/MSP.2006.29
2. Davies, D.: "A brief history of cryptography". Information Security Technical Report Volume 2 issue 2 ,(1997) , doi: 10.1016/s1363-4127(97)81323-4
3. "Public Key Cryptography (PKC) History", https://www.livinginternet.com/i/is_crypt_pkc_inv.htm. Last acessed 21 Nov 2018
4. Blakley, G.; Borosh, I.: ”Rivest-Shamir-Adleman public key cryptosystems do not always conceal messages” Computers & Mathematics with Applic ations Volume 5 issue 3, (1979) doi:10.1016/0898-1221(79)90039-7
5. Schmid, H.; Mahler, K.: "On the Chinese Remainder Theorem", Mathematische Nachrichten, (1958), doi:10.1002/mana.19580180112
6. "Using the CRT with RSA", https://www.di-mgt.com.au/crt_rsa.html, Last acessed 21 Dec 2018