Programming and Algorithms: RSA Cryptography

Arthur Duarte de Paula Coutinho Marques and Luísa Vieira da Silva

December 14, 2023

All code used for this project are available at GitHub

1 Introduction

Cryptography comes from the Greek "crypos" that means "secret" or "occult", and i's defined as the study of the act of encrypting messages in a way that only the sender and the receiver can decipher it. Currently, cryptography systems are largely used to send emails and to set up passwords for banking and social media [2].

The most known and commonly used system is RSA cryptography was invented in 1978 by R. L. Rivest, A. Shamir and L. Adlemen, thus, the acronym RSA stands for the initials of its three creators [2]. RSA uses a system of public and private keys: the receiver and the sender must have, respectively, a private and public key. Without the knowledge of both key is very unlikely that someone will be able to decipher the message.

The RSA algorithm is called an asymmetric cryptography algorithm, as it uses two different, mathematically linked keys. As their names suggest, a public key can be shared publicly, while a private key is secret and must not be shared with anyone [2]. Figure 1 illustrates how the RSA can be used to transmit messages safely.

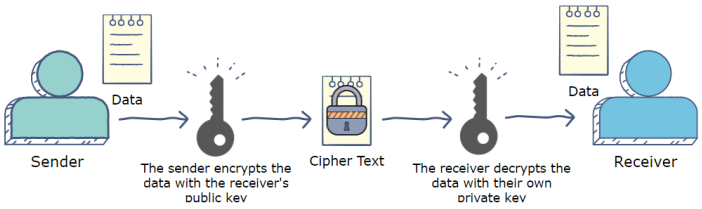

Figure 1: Simplified flowchart to explain asymmetric cryptography [7].

2 The RSA Algorithm

The RSA algorithm is based on the utilization of prime numbers and the basic steps described below:

1. Two random prime numbers p and q are chosen – the prime numbers must be big to increase the security of the encryption: the smallest number used for RSA cryptography had 100 digits (100 – 330 bits) and the last successfully broken RSA number was composed of 230 digits (762 bits) [8].

2. With p and q defined, it's possible to obtain $n = p * q$ and $\phi = (1 - p)(1 - q)$. Note that, ϕ is Euler's totient function [7].

3. A number e, which is part of the public key, must be chosen with the condition that 1 $<$ e $<$ ϕ. However, a commonly used value of e is 65,537, which is a Fermat prime and is relatively secure against known attacks [5].

4. Next step is to calculate d such that $e^d = 1 * mod(\phi)$. Therefore, d can be found using the extended Euclidean algorithm, better explained in section 4 [3].

5. The public key is then defined as (e, n), while the private key is composed of (n, d).

6. To transmit the message the sender must convert it into numbers. For this, it's possible to use a predefined table in a common conversion process that uses the ASCII alphabet, as shown in Figure 2 [4].

7. Given a public key (e, n), a plain text message in number format (P) can be encrypted calculating $C = P^e * mod(n)$, in which C is the encrypted message [7].

8. Finally, to decrypt the message the inverse operation can be done using the formula $P = C^d mod(n)$ [7].

9. The last step is to convert the decrypted message in number format back to its text format using the same table mentioned in the encryption step.

A	B	C	D	E	F	G	H	I	J	K	L	M
65	66	67	68	69	70	71	72	73	74	75	76	77
N	O	P	Q	R	S	T	U	V	W	X	Y	Z
78	79	80	81	82	83	84	85	86	87	88	89	90

Figure 2: Table for converting the letters of the alphabet into numbers [4].

3 Security of the RSA algorithm

As described above, to decode a message it's necessary to know the private key (n, d). Since the public key (n, e) already informs the value n, one would only need to find out

p and q through the factorization of n, then calculate ϕ and use the extended Euclidean algorithm to finally obtain d. However, factorizing n into two large prime numbers takes a lot of computational power, thus, with an increase on the size of the prime numbers p and q, becomes it's exponentially harder to find d.

According to Bonfim (2017), in 2005, the company RSA Laboratory launched a challenge for breaking keys of different sizes, one of them, with 193 algorithms, took 5 months and 80 computers to be broken. That's why, it's, to this day, impossible to decipher messages encrypted with RSA when the numbers p and q are large enough.

4 The Extended Euclidean Algorithm

The Euclidean Algorithm is used to find the Greatest Common Divisor (GCD) of two numbers a and b. To do so, it repeatedly divides the larger number by the smaller number until the remainder of the division reaches zero. The last non-zero remainder is the GCD [6].

The Extended Euclidean Algorithm is an extension of the Euclidean Algorithm, and it is used not only to find the greatest common divisor (GCD) of two numbers but also to determine coefficients that satisfy Bézout's identity. Bézout's identity states that for any integers a and b, there exist integers s and t such that $as + bt = GCD(a, b)$ [3].

The Extended Euclidean Algorithm determines the values of s and t through iterative back substitutions to rewrite the division algorithm equation until it finds an equation that is a linear combination of the original numbers [6]. This has a wide range of application, such as in modular arithmetic and to find the modular multiplicative inverse of a number [6].

An illustration of how the algorithm works can be seen in Figure 3.

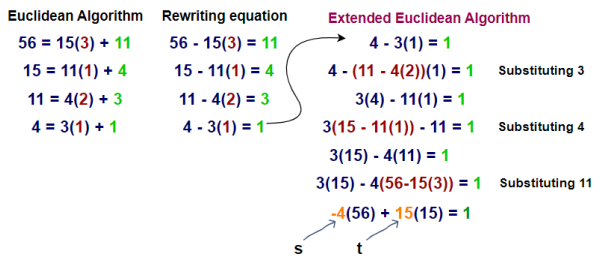

Figure 3: Simplified explanation of the Extended Euclidean Algorithm [6].

5 Objectives

On that note, the main objectives of this project are:

- Develop a Python code to reproduce the RSA algorithm.
- Create a user-friendly interface to generate keys, decrypt and encrypt a message.

- Simulate a hack attack that tries to find d based on the public key. The aim is to show how the computational power required to find the d increases as the size of the prime numbers p and q increases.

6 Implementing in Python

6.1 RSA Algorithm

The complete algorithm to generate keys is divided into small functions. To start, functions that generate the prime numbers p and q are defined.

```
def is_prime(self, num):
        for divisor in range(2, num):
            if num % divisor == 0:
                return False
        return True

def generate_prime(self, bits):
    while True:
        num = random.getrandbits(bits)
        if self.is_prime(num) and num != 0:
            return num
```

Listing 1: Function to generate prime numbers (Python)

First the function `is_prime()` is created to verify if a number is a prime number. After that, the function `generate_prime()` generates a random number, with the size of bits specified by the user. The function then calls back to the `is_prime()` function to verify if the number generated is indeed a prime number and if the condition is satisfied then the function returns this number.

After that, the function `extended_gcd()` is defined to perform the extended Euclidean algorithm to find the greatest common divisor between two numbers and the `modinv()` function finds the modular inverse of a number using the `extended_gcd()` function.

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

def modinv(self, a, m):
    g, x, y = self.extended_gcd(a, m)
    if g != 1:
        raise Exception('No modular inverse')
    else:
        return x % m
```

Listing 2: Functions for the Extended Euclidian algorithm and modular inverse (Python)

With the main functions defined, it's finally possible to generate the keys with a new function called `generate_keys()`, that has as input the number of bits provided by the user.

```python
def generate_keys(self, bits):
    p = self.generate_prime(bits)
    q = self.generate_prime(bits)

    self.n = p * q
    self.phi_n = (p - 1) * (q - 1)

    self.e = 65537
    self.d = self.modinv(self.e, self.phi_n)

    return p, q, self.n, self.phi_n, self.e, self.d
```

Listing 3: Function to generate public and private keys (Python)

The prime numbers p and q are randomly generated using the `generate_prime()` function, then n and *phi* are calculated based on its formulas. The number e is by default 65537 and d is calculated using the function `modinv()` that runs the extended Euclidean algorithm and calculates its modular inverse.

To encrypt the message, firstly the function `ord()` is used to transform each letter into a number, after that, the function `pow()` is used to perform the operation $(char^e)mod(n)$ for each character of the text, which returns the encrypted message.

```python
def rsa_encrypt(self, message, public_key):
    n, e = public_key
    encrypted_message = [pow(ord(char), e, n) for char in message]
    print(encrypted_message)
    return encrypted_message
```

Listing 4: Function encrypt a message (Python)

To decrypt the message, previously, the function `pow()` was used to return the value of i to the power of d, modulus n, representing the operation $(i^d)mod(n)$, where i corresponds to each character in the message. Subsequently, the function `chr()` converted the resultant numbers back to letters.

However, issues appeared during decryption of the last character of a message, when testing the encryption and decryption processes on different computers. The decryption process was, thus, adjusted as follows:

```python
def rsa_decrypt(self, message, private_key):
    n, d = private_key
    decrypted_message = [pow(byte, d, n) for byte in message[:-1]]
    partial_decryption = ''.join(chr(char) for char in decrypted_message)

    last_byte = pow(message[-1], d, n)
    last_char = chr(last_byte)

    full_decrypted_message = partial_decryption + last_char
    return full_decrypted_message
```

Listing 5: Function decrypt a message (Python)

This new approach ensures a more accurate decryption by treating the last character separately from the rest of the decryption process. Decryption now successfully reconstructs the original message, even when transmitted by different systems, using the public key generated by the decryptor to encrypt a message on the encrypting system and then decrypt it on the decrypting one.

6.2 RSA Modules

The RSA algorithm developed was divided in two modules that can be used on different computers. Firstly, the file decryption.py uses Kivy package to create the user interface and it's where the user enters the key size (bits) to generate the public and private key. In addition, it's where one can enter the encrypted message to decrypt it.

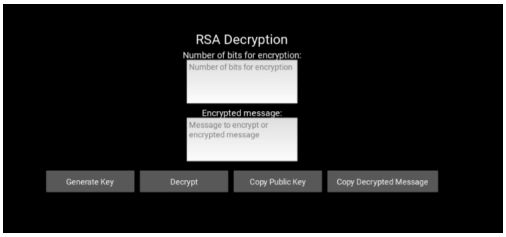

Figure 4: Interface for decryption module (Python)

The second module, file encrypted.py, allows the user to enter the public key generated by the decryption.py file and type in the message they want to encrypt. Once the encrypted message is generated, the user can copy and send it back to the receiver, who generated the keys and will decrypt the message.

Figure 5: Interface for encryption module (Python)

6.3 RSA Simulation

The RSA simulation file runs the process of encrypting and decrypting multiple times to compare how long it takes, on average, to decrypt different key sizes. It also plots the graph of number of bits over time of encryption and decryption. To get a better representation of the average time of decrypting for each key size, this code runs a high number of iterations (over 1000) and the graph is plotted based on the mean time of all repetitions.

6.4 RSA Hack simulation

This file tries to break the encrypted message using brute force. It runs from 5 bits to 15 bits, 10 times each, to have a better measure of how much time it takes. It also plots the chart of it.

Factoring a large number n into two smaller numbers, p and q, is a complex problem, and efficient algorithms are not known. The difficulty of this problem is the basis for the security of encryption algorithms. Therefore, to attempt a hack attack the approach chosen was to create a function that attempts to crack the RSA encryption by iteratively testing potential values of d to find the correct private key.

This function initiates the brute force process by incrementally increasing a variable ($possible_d$) and checking if the condition $pow(e, possible_d, n) == 1$. This is because manipulating the equation $d * e = 1 modphi(n)$ makes it possible to obtain $d * e\phi(n) = 1$ [1]. Therefore, the equation $pow(e, possible_d, n) == 1$ holds true if $possible_d$ is equal to the private key d.

In this code, the multiprocessing module is used to use multiple CPU cores concurrently, aiming to speed up the computation-intensive tasks. This parallel execution enables to process multiple key size at the same time. As a result, it significantly reduces the overall time required to the encryption, decryption, and hacking for multiple key sizes.

7 Results

To demonstrate how the encryption and decryption time grows with the increase in size of keys, the RSA algorithm runs a simple message ("Hello, world!") from 5 to 30 bits and the time of executing is recorded. The results can be observed in Figure 6.

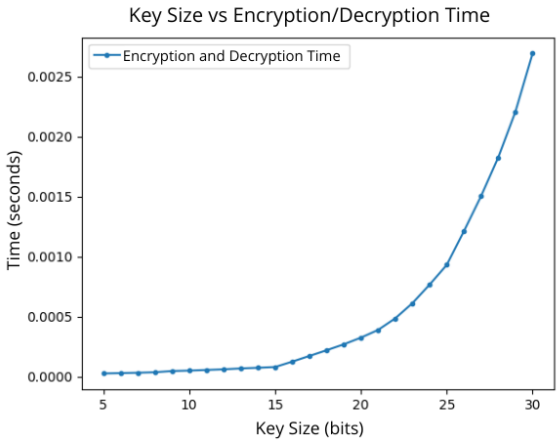

Figure 6: Graph of Encryption/Decryption time over key sizes

Since the keys are random, to get an average decryption time for each bit size, the code ran 10000 times, and the average time was used for plotting the graph. It's clear that the

graph behaves exponentially as key sizes increase. Of course, in a real-life example, actual RSA keys have, usually, over 100 bits, but, even on this smaller scale example, it's possible to understand how the size of keys make the decryption process much more complex and computationally expensive even when both public and private key are known.

In addition, it was simulated a "hack attack", meaning that the code for decryption was modified in order to simulate how much time does it take to break the encryption only knowing the public key and the encrypted message. The time for decrypting a message knowing both keys (blue line) and decrypting it when only the public key is known (orange line) is shown in Figure 7 for different key sizes (from 5 to 15).

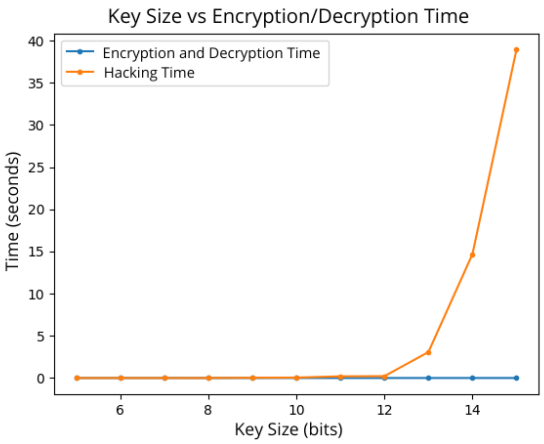

Figure 7: Graph of Encryption/Decryption time (blue line) and Hack time (orange line) over key sizes.

Once again, to get an average of time, for each bit, the code ran 10 different times, and this average is shown in the graph. Just as observed in Figure 2, the hacking time also grows exponentially with increase of key sizes. However, it increases a lot faster than for the case in which both keys are known.

This reinforces the discussion about the security of RSA algorithms, because, even for small keys, it's possible to see a rapidly exponential growth, therefore, it's expected that for bigger keys (over 100 bits) the amount of time and computational power required to successfully break those keys, will make this task undoable. Thus, it's possible to observe that the security of the algorithm increases with the increase of bits used to generate the keys.

8 Conclusion

In this project, it was possible to develop an code to reproduce the RSA algorithm for encryption of messages. A user-friendly interface was created and allowed the development

of two separate modules that enable users to exchange encrypted messages safely in different computers and locations.

The times of encryption and decryption were tested for different keys sizes, and it was proven, as expected, that the increase in bits exponentially increases the decryption time and the security of the algorithm. In addition, a hack attack was simulated to demonstrate that, without knowledge of the private key, using brute force to unravel an encrypted message becomes exponentially harder as the sizes of keys increases, and therefore, would be unachievable for large enough keys.

A proposed improvement for this project is to correctly fix the rsa_decrypt function. Currently, the function divides the last element from the original array to decrypt it separately, but the correct option would be to decrypt the entire array together, without splitting it. However, this was done because, when trying to encrypt a message in one system and decrypt it in another, there was a persistent problem with the decryption of the last character, although no it was not possible to identify the reason why (when encrypting and decrypting a message using the same computer, no problem was detected and the whole message was successfully decrypted without dividing the array).

References

[1] Khan Academy. Modular inverses. https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/modular-inverses#:~:text=Step%202.-,The%20modular%20inverse%20of%20A%20mod%20C%20is%20the%20B,A%20*%20B%20mod%20C%20%3D%201&text=Simple!

[2] D. H. Bonfim. Criptografia rsa. Dissertação de mestrado, Programa de Pós Graduação em Mestrado Profissional em Matemática em Rede Nacional, Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo (USP), São Carlos, São Paulo, Brasil, 2017.

[3] Brilliant.org. Extended euclidean algorithm. https://brilliant.org/wiki/extended-euclidean-algorithm/, 2023.

[4] Brilliant.org. Rsa encryption. https://brilliant.org/wiki/rsa-encryption/, 2023.

[5] Don Coppersmith. Small solutions to polynomial equations, and low exponent rsa vulnerabilities. *Journal of Cryptology*, 10(4):233–260, 1997.

[6] Educative. What is extended euclidean algorithm? https://www.educative.io/answers/what-is-extended-euclidean-algorithm, 2023.

[7] Educative. What is rsa algorithm? https://www.educative.io/answers/what-is-the-rsa-algorithm, 2023.

[8] Hacking na Web. Entendendo algoritmo rsa (de verdade). https://hackingnaweb.com/criptografia/entendendo-algoritmo-rsa-de-verdade/, 2019.