Exploring Cryptographic Models: A Comparative Study of RSA and BB84 for Secure Communication

Cala González Penagos (S4616162) and Mikel Martínez Garrido (S4752295)

University of Groningen, Physics Laboratory III, Supervisor: Marianne Westerhof

Abstract

This scientific report investigates the RSA encryption model and the BB84 quantum cryptography model, exploring their relevance in achieving secure communication. Regarding the RSA protocol, which uses big prime number factorization, the encryption and cracking times were found to increase as the key length n increases. However, for the BB84 protocol completely secure communication could be achieved, as the no-cloning theorem forbids the replication of a certain quantum system. The presence of Eve causes the encryption keys of Bob and Alice to differ due to a 50% *probability of receiving the correct bit with a different bases for Eve with respect to Alice and Bob. Sharing the key bits allows detection of Eve without revealing sensitive information. Openly sharing bases at the end does not compromise key safety, as the security relies on quantum mechanics principles (the uncertainty principle). This highlights the protocol's security and its ability to establish a secure key.*

Introduction

In an interconnected world, secure and effective communication is vital, especially when sensitive information is involved. Traditional cryptographic methods are increasingly vulnerable to emerging threats from computational advances and algorithmic attacks. Hence, the focus has shifted to quantum cryptography, leveraging the principles of quantum mechanics for a robust security model. This paper explores the motivation behind researching the RSA encryption model and the BB84 quantum cryptography model. It highlights how the unique properties of quantum mechanics enable secure communication, in contrast to traditional protocols reliant on mathematical models.

Since the pioneering work of Diffie and Hellman [1], numerous solutions have emerged in the field of cryptography. The RSA cryptographic system [2] is widely recognized as one of the most prominent techniques. Additionally, the Zero Knowledge Proofs (ZKP) technique, primarily based on Number Theory [3], has gained significant interest. These techniques capitalize on the fact that no efficient algorithm for factoring large integers has been discovered, making the computation of large prime numbers their strength.

In this article the centre of attention is set on the most fundamental encryption issue: secure communication between two spatially separated parties, traditionally called Alice (A) and Bob (B), with the wave surrogate Eve (E). For this, the Bennett-Brassard protocol (so-called BB84 protocol) is analyzed, which technically should be secure against wave spoofing.

Theory

RSA protocol

The RSA protocol is a public-key cryptosystem, which means that the message is encrypted by a public key created and publicly broadcast by the receiver, and then it is decrypted by a private key, created and only held by the receiver of the message. The receiver then is the one that must create both the public and private keys, and send the public one to the sender. To create the key, firstly two prime numbers, p and q, are chosen, and the key length is defined to be $n = pq$. Then, Carmichael's $\lambda(n)$ function of the key length n is computed, where $\lambda(n)$ is defined to be the smallest integer m such that [4]:

$$a^m \equiv 1(mod(n)) \tag{1}$$

so $\lambda(n) = LCM(\lambda(p), \lambda(q))$, where p and q are prime, they can be expressed via Euler's totient function ϕ, which for prime powers p is $\phi(p^r) = p^{r-1}(p-1)$. Then it follows that $\lambda(p) = \phi(p) = p-1$, and hence $\lambda(n) = LCM^1(p-1, q-1)$. Afterward, a coprime number e to $\lambda(n)$ is chosen so that $GCD^2(e, \lambda(n)) = 1$. The key length n and the number e are released as the public key. Finally, the private key d is defined to be the modular multiplicative inverse of 3 modulo $\lambda(n)$: $d \equiv e^{-1}(mod(n))$ [4].

Then the message M is encrypted by the sender via the public key as [4]:

$$C = M^e mod(n) \tag{2}$$

And it is then recovered by the receiver via its private key by [4]:

$$C^d = (M^e)^d = M mod(n) \tag{3}$$

The security of this protocol lies in the inefficient factorization of large prime numbers, so with enough computing power is it theoretically always possible to break. Indeed, primality tests take big time to factorize large prime numbers, and no other more efficient tests have been found yet [5], but advances in quantum computing may accelerate this process exponentially [6].

Quantum cryptography and BB84 protocol

Quantum Key Distribution (QKD) allows Alice and Bob to create a common secret key through classical and quantum public channels without the need for a trusted intermediary. Then, in QKD schemes a completely reliable secret encryption key can be generated, for example, following the BB84 protocol. Quantum computation poses a threat to classical encryption systems while also offering a solution through quantum key distribution protocols. Classical encryption methods become vulnerable to quantum attacks, while quantum key distribution leverages quantum mechanics to establish secure communication channels using principles like entanglement and uncertainty.

In this experiment, polarized photons are used as the quantum system. The internal state of a photon, such as its polarization, is represented by a vector ψ in a 2-dimensional Hilbert space $\mathcal{H}$. The photon's state can be described as a linear

[1] LCM stands for Least Common Multiple

[2] GCD stands for Greatest Common Divisor

combination of basis vectors $r_1 = (1,0)$ and $r_2 = (0,1)$ for horizontal and vertical polarization, respectively. Measurements of vertical-versus-horizontal polarization result in the photon adopting a horizontal state with a probability of $cos^2\alpha$ and a vertical state with a probability of $sin^2\alpha$, where α represents the polarization filter angle. Additionally, a diagonal basis consisting of vectors $d_1 = (0.707, 0.707)$ and $d_2 = (0.707, -0.707)$ representing 45° and −45° photons, respectively, can also be used. These bases form the four qubit signal states utilized in the experiment.[7]

$$|0^\circ\rangle = |\psi_{00}\rangle = |0\rangle \tag{4}$$

$$|90^\circ\rangle = |\psi_{10}\rangle = |1\rangle \tag{5}$$

$$|45^\circ\rangle = |\psi_{01}\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \tag{6}$$

$$|-45^\circ\rangle = |\psi_{11}\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \tag{7}$$

Generating two orthonormal qubit bases, $\mathcal{B}_0 = \{|\psi_{00}\rangle, |\psi_{10}\rangle\}$ and $\mathcal{B}_1 = \{|\psi_{01}\rangle, |\psi_{11}\rangle\}$

We establish the following operators to describe a measurement in one of the two bases:

$$\hat{M}_{\mathcal{B}_0} = |0^\circ\rangle\langle 0^\circ| - |90^\circ\rangle\langle 90^\circ| \tag{8}$$

$$\hat{M}_{\mathcal{B}_1} = |45^\circ\rangle\langle 45^\circ| - |-45^\circ\rangle\langle -45^\circ| \tag{9}$$

Then, if a bit set in a certain basis is measured with the other one:

$$\hat{M}_{\mathcal{B}_1}|90^\circ\rangle = \frac{1}{\sqrt{2}}|45^\circ\rangle + \frac{1}{\sqrt{2}}|-45^\circ\rangle \tag{10}$$

As each coefficient squared is the probability of the bit being measured in each state, if a bit 1 is sent in one basis and measured in the other one, the receiver has a 50% probability of measuring either a 0 or a 1, meaning that all information is lost. It will happen for all bases and possible bits, all calculations are in Appendix A. This process is a perfect example of the no-cloning theorem, which states that a certain unknown quantum state cannot be duplicated. The aforementioned theorem is exploited by the BB84 protocol in the key-generating process. If two series of random bases are chosen by Alice and Bob and an array of also random bits is sent, they will agree on every bit they have the same basis for. However, if Eve tries to intercept the key they will just agree on half of the bits with the same bases.

Through subsequent communication over a regular non-quantum channel, which may be susceptible to eavesdropping, they can determine with high probabilities whether the original quantum transmission was tampered with and if such interference was caused by an eavesdropper. If the transmission remains undisturbed, they agree to utilize these shared secret bits as a one-time pad, a well-known method, to encrypt subsequent meaningful communications and ensure their confidentiality [7].

Setup and procedure

The setup used is shown in figure 1.

The sensor electronics had the alignment and the measurement modes, the polarisers 1 and 3 could be adjusted between 0° and 90°, and the polarisers 2 and 4 could range between −45° and 90°. The lasers could emit continuous beams or pulses, depending on how long the button was pressed. Finally, the sensors emitted a blue light when they received the pulse.

First, in the absence of Eve an encryption key was created and a message was transmitted. To create the encryption key an algorithm was used to create a random list of 42 bits and two series of 42 random bases for Alice and Bob. The bits were sent with the random series of Alice and received with the random series of Bob. The encryption key was generated by choosing

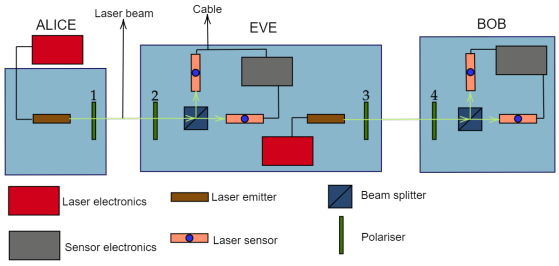

Figure 1: *Schematic of the set-up. Three parts are visible (Alice, Bob, and Eve)*

the bits where Alice and Bob had the same basis. Then, the message was encrypted using binary addition (Table 1) and now sent with a defined series of bases used both by Alice and Bob. The bits received by Bob were decrypted with the encryption key. The same procedure was repeated in the presence of Eve, creating another set of random bases in the key generation process for her. Then, four messages were sent with four differently generated encryption keys for the following Alice, Eve, and Bob fixed bases combinations: (x, x, x), $(+, +, +)$, $(+, x, +)$ and $(x, +, x)$.

Regarding the RSA protocol, an algorithm was created to encrypt and decrypt a message in ASCII [8] through equations 2 and 3. Then, another piece of code was developed to break the message by factorization of the key length n (code in Appendix D). The time needed for encrypting, decrypting, and, cracking the message was recorded for various combinations of prime numbers, and different e values.

Results and discussion

RSA protocol

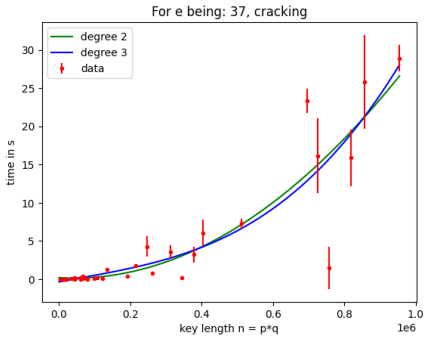

Figure 2: *Key length n vs time needed for cracking the message, for $e = 37$.*

The message is encoded in ASCII language as $(84, 117, 32, 101, 114, 101, 32, 117, 110, 32, 115, 97, 112, 111)$. The encryption time remains low regardless of the value of e and does not increase with longer key lengths n. However, the decryption and code-breaking times increase by a second/third polynomial as the key length increases as seen in Figures 2 and 3, with decryption taking longer than breaking the code.

A maximum key length of 6 digits was used due to computational limitations. Decrypting time was greater than breaking time for the investigated key lengths, indicating the need for larger key lengths to ensure message security. The optimal value for the second part of the public key (e) was found to be 65537,

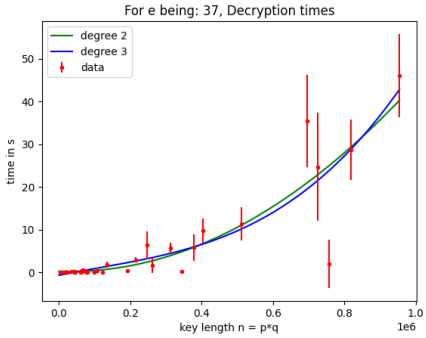

Figure 3: *Graph of key length n vs time needed for decrypting the message, for e = 37.*

as resulted in shorter decrypting times. The RSA protocol was proved to heavily rely on the computing power of both the receiver and potential eavesdropper, with no influence of the last on the message's content. The key length plays a crucial role in the decryption and code-breaking time, and this is due to the significant time required for factorizing large numbers [5].

BB84 protocol

Four different trials have been done: one without Eve, having Bob and Alice on the same basis (+,+), one where Bob, Alice and Eve all shared the same basis (+,+,+), and two with Alice and Bob agreeing on the same basis but with a different one for Eve: (+,x,+) [1] and (x,+,x). In the four trials conducted, the absence of Eve resulted in successful communication between Alice and Bob, with agreement on the encryption key and all received bits. However, when Eve was present, the encryption keys differed, leading to a discrepancy between the transmitted and received messages. When Alice, Bob, and Eve shared the same basis, the accuracy between encrypted and received bits remained at 100%. However, when Eve had a different basis, the accuracy dropped to approximately 50%, indicating a noticeable deviation from the original message. Then, as soon as a yellow bit is found on the decrypted bit that would make the final message differ from the original one, making Eve noticeable.

Basis	**A: + E: x B: +**														
Letter	E					O					H				
Bit	0	0	1	0	0	0	1	1	1	0	0	0	1	1	1
Alice's Key	1	0	0	1	1	0	0	1	0	0	0	0	0	0	1
Encrypted Bit	1	0	1	1	1	0	1	0	1	0	0	0	1	1	0

Received Bit	0	0	0	1	0	1	0	0	0	0	0	1	0	1	0
Bob's Key	1	1	0	1	1	0	0	1	1	0	0	1	1	0	1
Decrypted bit	1	1	0	0	1	1	0	1	1	0	0	0	1	1	1
Letter	Z					W					H				

Table 1: *Eve trial 3: (+,x,+). The green and red indicate the received bits that coincide (or not) with the encrypted bits sent respectively. The yellow decrypted bits indicate the bits that do not agree with the bits sent in the beggining, while blue indicates the bits that agree.*

Is worth to mention that randomly generating bases and bits ensures security by preventing predictable patterns and correlations, making it difficult for eavesdroppers to gain undetected information about the key. Computers use pseudorandom number generators (PRNGs) to generate numbers, which are not truly random. To achieve true randomness, external physical processes like electrical noise, radioactive decay, quantum phenomena, or atmospheric noise can be exploited. Before sending a message, Eve's presence can be detected during key generation. If a bit is sent and received on different bases, complete information loss occurs, eq. 10. Alice and Bob only agree on bits with the same bases at a 50% rate. By checking part of the array, they can detect Eve's presence.

On the other hand, sharing the bases openly at the end in the BB84 protocol does not impact the safety of the key. The security of the key in BB84 relies on the principles of quantum mechanics, specifically the uncertainty principle, rather than the secrecy of the bases. By publicly revealing the bases used during the key exchange, both Alice and Bob can compare a subset of their shared bits to detect any potential interference or eavesdropping by Eve. Any discrepancies in the measured values indicate a potential presence of an eavesdropper, allowing Alice and Bob to discard those bits and establish a secure key. Therefore, openly sharing the bases at the end of the protocol contributes to the security rather than compromising it.

Conclusion

The main focus of this paper repairs on the comparative analysis of the RSA and BB84 protocols, examining their underlying principles and security mechanisms. Regarding the RSA protocol, the message can be intercepted with sufficient computer power. Moreover, for low key lengths n it was found that the encrypting took longer than the cracking of the message.

Regarding the BB84 protocol, 4 trials have been done, one without Eve and three with Eve, including variations on the basis chosen. From the moment that Eve is added, the encryption keys for Bob and Alice differ, since the probability of receiving the correct bit while going from different basess is of the 50% [10]. This makes Eve noticeable since Bob receives a different message that what Alice sent. However by sharing bits used to create the key, this method allows to notice Eve without sending any type of sensitive information. Also, in the BB84 protocol, openly sharing the bases at the end does not compromise the safety of the key. The security of the key is ensured by the principles of quantum mechanics, not the secrecy of the bases. Hence, openly sharing the bases contributes to the security of the protocol, allowing for the establishment of a secure key.

References

[1] Martin E Hellman Whitfield Diffiey. "New directions in cryptography". In: *Information Theory* (22.6 1976), pp. 644–654.

[2] Adi Shamir Ronald L Rivest and Len Adleman. "A method for obtaining digital signatures and public-key cryptosystems". In: *Communications of theACM* (21.2 1978), pp. 120–126.

[3] Teresa Joven. "RSA cryptography: fundamentals and development". In: *Universidad Zaragoza* ().

[4] A. Shamir R. L. Rivest and L. Adleman. "A Method for Obtaining Digital Signatures and PublicKey Cryptosystems". In: *Communications of the ACM* 21.2 (1978).

[5] Maxym Seniv Vitaly Yakovyna Dmytro Fedasyuk and Orest Bilas. "The Performance Testing of RSA Algorithm Software Realization". In: *Physical Review Letters* (2007). DOI: 10.1109/CADSM.2007.4297593.

[6] Nilanjana Datta. "QUANTUM INFORMATION COMPUTATION". In: *DAMTP Cambridge* (2020), p. IIC.

[7] Charles H. Bennett and Gilles Brassard. "QUANTUM CRYPTOGRAPHY: PUBLIC KEY DISTRIBUTION AND COIN TOSSING". In: *Physical Review Letters* ().

[8] Karen M. Strom. *ASCII - Binary Character Table*. URL: http://sticksandstones.kstrom.com/appen.html.

Appendix

A Further Derivations

Exploiting the 4 states spanning the Hilbert space as linear combinations:

$$|-45^\circ\rangle = \frac{1}{\sqrt{2}}|0^\circ\rangle - \frac{1}{\sqrt{2}}|90^\circ\rangle$$

$$|0^\circ\rangle = \frac{1}{\sqrt{2}}|45^\circ\rangle + \frac{1}{\sqrt{2}}|-45^\circ\rangle$$

$$|45^\circ\rangle = \frac{1}{\sqrt{2}}|0^\circ\rangle + \frac{1}{\sqrt{2}}|90^\circ\rangle$$

$$|90^\circ\rangle = \frac{1}{\sqrt{2}}|45^\circ\rangle - \frac{1}{\sqrt{2}}|-45^\circ\rangle$$

And also exploiting the orthogonality $\langle\Phi_a|\Phi_b\rangle = \delta_{a,b}$, how a state in a certain basis behaves when it is received by the other basis can be understood as:

$$\hat{M}_{\mathcal{B}_1}|90^\circ\rangle = |45^\circ\rangle\langle 45^\circ|90^\circ\rangle - |-45^\circ\rangle\langle -45^\circ|90^\circ\rangle$$

$$= |45^\circ\rangle\frac{1}{\sqrt{2}}(\langle 0^\circ|90^\circ\rangle + \langle 90^\circ|90^\circ\rangle) - |-45^\circ\rangle\frac{1}{\sqrt{2}}(\langle 0^\circ|90^\circ\rangle - \langle 90^\circ|90^\circ\rangle)$$

$$= |45^\circ\rangle\frac{1}{\sqrt{2}}(0+1) - |-45^\circ\rangle\frac{1}{\sqrt{2}}(0-1) = \frac{1}{\sqrt{2}}|45^\circ\rangle + \frac{1}{\sqrt{2}}|-45^\circ\rangle \tag{11}$$

The probability of each state is given by its coefficient squared. This means that a bit 1 is sent in the $+$ basis, if it is read in the x basis it will be recorded as a 0 with a probability of 50% and as a 1 with a probability of 50%.

$$\hat{M}_{\mathcal{B}_1}|0^\circ\rangle = |45^\circ\rangle\langle 45^\circ|0^\circ\rangle - |-45^\circ\rangle\langle -45^\circ|0^\circ\rangle$$

$$= |45^\circ\rangle\frac{1}{\sqrt{2}}(\langle 0^\circ|0^\circ\rangle + \langle 90^\circ|0^\circ\rangle) - |-45^\circ\rangle\frac{1}{\sqrt{2}}(\langle 0^\circ|0^\circ\rangle - \langle 90^\circ|0^\circ\rangle)$$

$$= |45^\circ\rangle\frac{1}{\sqrt{2}}(1+0) - |-45^\circ\rangle\frac{1}{\sqrt{2}}(1-0) = \frac{1}{\sqrt{2}}|45^\circ\rangle - \frac{1}{\sqrt{2}}|-45^\circ\rangle \tag{12}$$

The probability of each state is given by its coefficient squared. This means that a bit 0 is sent in the $+$ basis, if it is read in the x basis it will be recorded as a 0 with a probability of 50% and as a 1 with a probability of 50%.

$$\hat{M}_{\mathcal{B}_0}|45^\circ\rangle = |0^\circ\rangle\langle 0^\circ|45^\circ\rangle - |90^\circ\rangle\langle 90^\circ|45^\circ\rangle$$

$$= |0^\circ\rangle\frac{1}{\sqrt{2}}(\langle 45^\circ|45^\circ\rangle + \langle -45^\circ|45^\circ\rangle) - |90^\circ\rangle\frac{1}{\sqrt{2}}(\langle 45^\circ|45^\circ\rangle - \langle -45^\circ|45^\circ\rangle)$$

$$= |0^\circ\rangle\frac{1}{\sqrt{2}}(1+0) - |90^\circ\rangle\frac{1}{\sqrt{2}}(1-0) = \frac{1}{\sqrt{2}}|0^\circ\rangle - \frac{1}{\sqrt{2}}|90^\circ\rangle \tag{13}$$

The probability of each state is given by its coefficient squared. This means that a bit 1 is sent in the x basis, if it is read in the $+$ basis it will be recorded as a 0 with a probability of 50% and as a 1 with a probability of 50%.

$$\hat{M}_{\mathcal{B}_0}|-45^\circ\rangle = |0^\circ\rangle\langle 0^\circ|-45^\circ\rangle - |90^\circ\rangle\langle 90^\circ|-45^\circ\rangle$$

$$= |0^\circ\rangle\frac{1}{\sqrt{2}}(\langle 45^\circ|-45^\circ\rangle + \langle -45^\circ|-45^\circ\rangle) - |90^\circ\rangle\frac{1}{\sqrt{2}}(\langle 45^\circ|-45^\circ\rangle - \langle -45^\circ|-45^\circ\rangle)$$

$$= |0^\circ\rangle\frac{1}{\sqrt{2}}(0+1) - |90^\circ\rangle\frac{1}{\sqrt{2}}(0-1) = \frac{1}{\sqrt{2}}|0^\circ\rangle + \frac{1}{\sqrt{2}}|90^\circ\rangle \tag{14}$$

The probability of each state is given by its coefficient squared. This means that a bit 0 is sent in the x basis, if it is read in the $+$ basis it will be recorded as a 0 with a probability of 50% and as a 1 with a probability of 50%.

B Raw data: other communication tables studied on this experiment and creation of encryption keys

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Basis (+ or x)	x	x	+	x	+	+	x	+	x	+	+	+	x	x
Bit (0 or 1)	1	0	1	0	1	1	0	0	0	1	0	0	0	1
	15	16	17	18	19	20	21	22	23	24	25	26	27	28
Basis (+ or x)	+	x	+	x	+	x	x	x	x	+	x	x	x	+
Bit (0 or 1)	0	0	0	1	1	0	0	0	0	1	1	0	0	0
	29	30	31	32	33	34	35	36	37	38	39	40	41	42
Basis (+ or x)	x	x	+	x	x	x	+	+	x	x	x	x	x	x
Bit (0 or 1)	1	1	1	1	0	1	0	1	1	0	0	0	1	0

Table 7: Key generator for Alice

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Basis (+ or x)	x	+	+	x	x	x	x	+	x	+	x	x	x	+
	15	16	17	18	19	20	21	22	23	24	25	26	27	28
Basis (+ or x)	x	x	x	+	+	x	+	x	+	+	x	+	x	+
	29	30	31	32	33	34	35	36	37	38	39	40	41	42
Basis (+ or x)	x	x	x	x	+	x	x	x	x	x	+	+	x	+

Table 11: Basis selection for Eve

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Basis (+ or x)	x	x	+	x	+	+	x	+	+	+	x	+	+	+
Bit (0 or 1)	1	1	1	0	1	1	0	0	0	1	1	1	0	1
	15	16	17	18	19	20	21	22	23	24	25	26	27	28
Basis (+ or x)	+	x	x	+	x	+	+	+	+	x	+	x	x	+
Bit (0 or 1)	0	0	0	0	0	0	0	1	1	0	1	1	1	0
	29	30	31	32	33	34	35	36	37	38	39	40	41	42
Basis (+ or x)	x	+	+	x	x	+	+	+	x	+	x	x	+	x
Bit (0 or 1)	1	0	1	1	0	1	1	0	1	0	1	0	1	1

Table 8: Key generator for Bob

Figure 4: *Key encryption process for (+,x,+)*

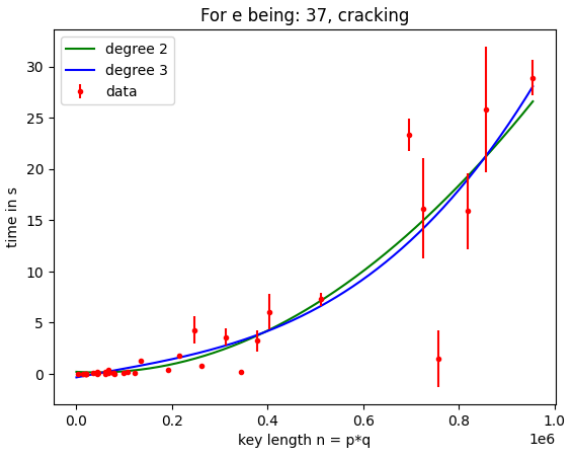

Figure 5: *Key length n vs time needed for cracking the message, for e = 37.*

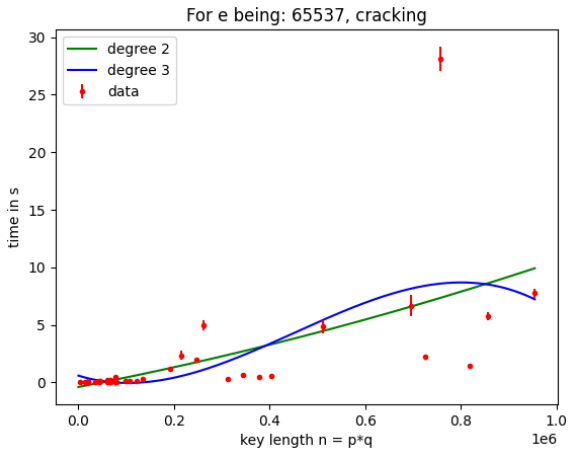

Figure 6: *Key length n vs time needed for cracking the message, for e = 65537.*

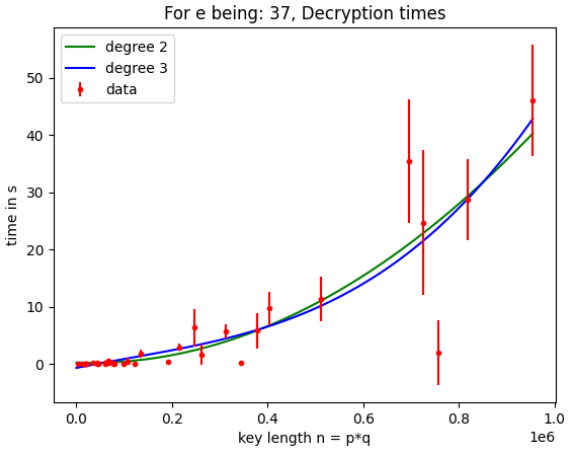

Figure 7: *Key length n vs time needed for decrypting the message, for e = 37.*

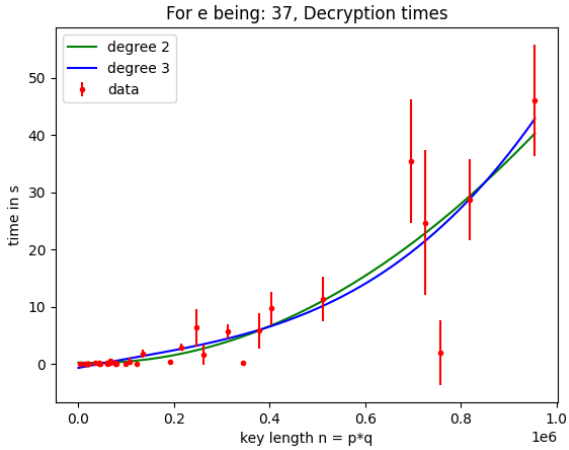

Figure 8: *Key length n vs time needed for decrypting the message, for e = 65537.*

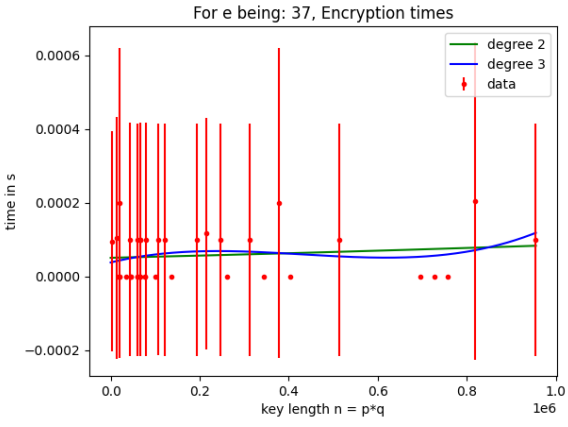

Figure 9: *Key length n vs time needed for encrypting the message, for e = 37.*

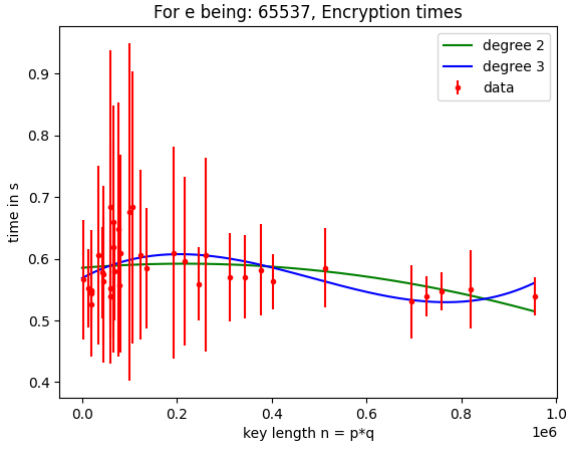

Figure 10: *Key length n vs time needed for encrypting the message, for e = 65537.*

Basis	**A: + B: +**														
Letter	***S***					***P***					***O***				
Bit	1	0	0	1	0	0	1	1	1	1	0	1	1	1	0
Alice's Key	1	0	1	0	0	0	0	1	0	1	1	0	0	0	0
Encrypted Bit	0	0	1	1	0	0	1	0	1	0	1	1	1	1	0
Received Bit	0	0	1	1	0	0	1	0	1	0	1	1	1	1	0
Bob's Key	1	0	0	1	0	0	1	1	1	1	0	1	1	1	0
Decrypted bit	1	0	0	1	0	0	1	1	1	1	0	1	1	1	0
Letter	***S***					***P***					***O***				

Table 2: *Without Eve: (+,+)*

Basis	**A: + E: + B: +**														
Letter	***W***					***O***					***W***				
Bit	1	0	1	1	0	0	1	1	1	0	1	0	1	1	0
Alice's Key	0	1	0	1	0	1	1	1	1	0	0	0	0	1	0
Encrypted Bit	1	1	1	0	0	1	0	0	0	0	1	0	1	0	0
Received Bit	1	1	1	0	0	1	0	0	0	0	1	0	1	0	0
Bob's Key	0	0	0	1	0	1	1	0	1	0	0	0	0	1	1
Decrypted bit	1	1	1	1	0	0	1	0	1	0	1	0	1	1	1
Letter	**#**					***K***					***X***				

Table 3: *Eve trial 1: (+,+,+)*

Basis	**A:** x **E:** + **B:** x														
Letter	***A***					***P***					***O***				
Bit	0	0	0	0	0	0	1	1	1	1	0	1	1	1	0
Alice's Key	0	1	0	0	1	1	0	0	0	1	0	0	0	0	1
Encrypted Bit	0	1	0	0	1	1	1	1	1	0	0	1	1	1	1
Received Bit	1	1	0	1	0	0	0	1	0	1	1	1	0	0	1
Bob's Key	0	1	1	0	1	1	0	0	0	1	0	0	1	0	1
Decrypted bit	1	0	1	1	1	1	0	1	0	0	1	1	1	0	0
Letter	***X***					***U***					**#**				

Table 4: *Eve trial 2: (x,+,x)*

C Python code for encrypting, decrypting and breaking the message

```
# Python for RSA asymmetric cryptographic algorithm.
# For demonstration, values are
# relatively small compared to practical application
import math
import matplotlib.pyplot as plt
import numpy as np

#Greatest common divisor of a and h
def gcd(a, h):
    temp = 0
    while(1):
        temp = a % h
        if (temp == 0):
            return h
        a = h
        h = temp

# Python Program to find the L.C.M. of two input number
```

```
def compute_lcm(x, y):

   # choose the greater number
   if x > y:
       greater = x
   else:
       greater = y

   while(True):
       if((greater % x == 0) and (greater % y == 0)):
           lcm = greater
           break
       greater += 1

   return lcm

def compute_d(e, phi):
    g, x, y = egcd_for_computing_d(e, phi)
    if g != 1:
        raise Exception('modular inverse does not exist')
    else:
        return x % phi

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

def encrypting(message, e, n):
    message_encrypted = []
    for m in message:
        message_encrypted.append(m**e % n)
    return message_encrypted

def decrypting(message_encrypted, d, n):
    message_decrypted = []
    for dec in message_encrypted:
        message_decrypted.append(dec**d % n)
    return message_decrypted

#ENCRYPTING AND DECRYPTING CODE
import time
import matplotlib.pyplot as plt
import statistics as stat
import pylab
import numpy as np
message =(84,117,32,101,114,101,32,117,110,32,115,97,112,111)   #Tu ere un sapo

loops = 10

e_numbers = [3,17,65537]

p_numbers = [13,71,97,101,103,107,179,229,233,239,311,313,317,347,349,353,401,409,419,521,557,641,709,1009,1129,1

q_numbres = [191,191,191,191,191,191,191,191,191,191,191,191,191,191,191,191,191,191,191,191,191,191,191,191,191,
print(len(p_numbers))
print(len(q_numbres))

def time_n_counter(p_all,q_all,e, message,loops):
    if len(p_all) == len(q_all):
        for e in e:
            times_encrypting_list = []
            times_decrypting_list = []

            times_encrypting_averages = []
            times_decrypting_averages = []

            times_encrypting_errors = []
```

```
times_decrypting_errors = []

for counts in range(loops):
    n_values = []
    encrypting_times = []
    decrypting_times = []
    for i in range(len(p_all)):
        p = p_all[i]
        q = q_all[i]
        n=p*q
        n_values.append(n)

        phi = compute_lcm(p-1,q-1)

        while (e < phi):

            # e must be co-prime to phi and
            # smaller than phi.
            if(gcd(e, phi) == 1):
                break
            else:
                e = e+1

        #encrypting time
        start_encrypting = time.time()
        encrypting(message, e, n)
        end_encrypting = time.time()

        encryp = end_encrypting - start_encrypting

        #decrypting time
        start_decrypting = time.time()
        decrypting(encrypting(message, e, n), compute_d(e,phi), n)
        end_decrypting = time.time()

        decryp = end_decrypting - start_decrypting

        #appending times
        encrypting_times.append(encryp)
        decrypting_times.append(decryp)

    times_encrypting_list.append(encrypting_times)
    times_decrypting_list.append(decrypting_times)

#Computing means and stdes for encrypting
for i in range(len(p_all)):
    values = []
    for j in range(loops):
        x = times_encrypting_list[j][i]
        values.append(x)
    means = stat.mean(values)
    devs = stat.stdev(values)
    times_encrypting_averages.append(means)
    times_encrypting_errors.append(devs)

#Computing means and stdes for decrypting
for i in range(len(p_all)):
    values = []
    for j in range(loops):
        x = times_decrypting_list[j][i]
        values.append(x)
    means = stat.mean(values)
    devs = stat.stdev(values)
    times_decrypting_averages.append(means)
    times_decrypting_errors.append(devs)

#DECRYPTING TIMES
# create nth degree polynomial fit
n_decryp = 1
zn_decryp = np.polyfit(n_values, times_decrypting_averages,n_decryp)
```

```
            pn_decryp = np.poly1d(zn_decryp) # construct polynomial

            # create qth degree polynomial fit
            q_decryp = 5
            zq_decryp = np.polyfit(n_values, times_decrypting_averages,q_decryp)
            pq_decryp = np.poly1d(zq_decryp)

            # plot data and fit
            xx_decryp = np.linspace(0, max(n_values), 500)
            pylab.plot(xx_decryp, pn_decryp(xx_decryp),'-g', xx_decryp, pq_decryp(xx_decryp),'-b')
            pylab.errorbar(n_values, times_decrypting_averages, yerr=times_decrypting_errors, fmt='r.')

            # customise graph
            pylab.title(f"For e being: {e}, cracking")
            pylab.legend(['degree '+str(n),'degree '+str(q),'data'])
            pylab.xlabel("key length n = p*q")
            pylab.ylabel("time in s")

            pylab.show()

            #ENCRYPTING TIMES
            # create nth degree polynomial fit
            n_encryp = 1
            zn_encryp = np.polyfit(n_values, times_encrypting_averages,n_encryp)
            pn_encryp = np.poly1d(zn_encryp) # construct polynomial

            # create qth degree polynomial fit
            q_encryp = 5
            zq_encryp = np.polyfit(n_values, times_encrypting_averages,q_encryp)
            pq_encryp = np.poly1d(zq_encryp)

            # plot data and fit
            xx_encryp = np.linspace(0, max(n_values), 500)
            pylab.plot(xx_encryp, pn_encryp(xx_encryp),'-g', xx_encryp, pq_encryp(xx_encryp),'-b')
            pylab.errorbar(n_values, times_encrypting_averages, yerr=times_encrypting_errors, fmt='r.')

            # customise graph
            pylab.title(f"For e being: {e}, cracking")
            pylab.legend(['degree '+str(n),'degree '+str(q),'data'])
            pylab.xlabel("key length n = p*q")
            pylab.ylabel("time in s")

            pylab.show()

    else:
        print("The number of p's and q's is not the same")

time_n_counter(p_numbers, q_numbres, e_numbers, message,loops)

#BREAKING MESSAGE CODE
import time
import matplotlib.pyplot as plt
import statistics as stat
import pylab
import numpy as np
def compute_lcm(x, y):

   # choose the greater number
   if x > y:
       greater = x
   else:
       greater = y

   while(True):
       if((greater % x == 0) and (greater % y == 0)):
           lcm = greater
           break
       greater += 1

   return lcm
```

```
def gcd(a, h):
    temp = 0
    while(1):
        temp = a % h
        if (temp == 0):
            return h
        a = h
        h = temp

def factors_of_a_number(x):
    factors = []
    for i in range(1, x + 1):
        if x % i == 0:
            factors.append(i)
    factors.remove(1)
    factors.remove(x)
    return factors

def compute_d(e, phi):
    g, x, y = egcd_for_computing_d(e, phi)
    if g != 1:
        raise Exception('modular inverse does not exist')
    else:
        return x % phi

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

def encrypting(message, e, n):
    message_encrypted = []
    for m in message:
        message_encrypted.append(m**e % n)
    return message_encrypted

def decrypting(message_encrypted, d, n):
    message_decrypted = []
    for dec in message_encrypted:
        message_decrypted.append(dec**d % n)
    return message_decrypted

def breaking(message, e_all, p_all, q_all, loops):
    for e in e_all:
        times_list = []
        times_averages = []
        times_errors = []
        for counts in range(loops):
            n_values = []
            cracking_times = []
            for i in range(len(p_all)):
                p = p_all[i]
                q = q_all[i]
                n=p*q
                n_values.append(n)

                phi = compute_lcm(p-1,q-1)

                while (e < phi):

                    # e must be co-prime to phi and
                    # smaller than phi.
                    if(gcd(e, phi) == 1):
                        break
                    else:
                        e = e+1
```

```
            #message to break
            encrypted_message_to_break = encrypting(message, e, n)

            #CRACK
            #getting the factors of n
            start_crack = time.time()
            factors = factors_of_a_number(n)
            if len(factors) == 2:
                pass
            else:
                factors = factors_of_a_number(n)

            #computing trial variables
            p_trial, q_trial = factors_of_a_number(n)
            phi_trial = compute_lcm(p_trial-1,q_trial-1)

            trial_d = compute_d(e, phi_trial)

            #possible crack
            possible_final_message = decrypting(encrypted_message_to_break, trial_d, n)

            #veryfing the message
            crack = 0
            while crack == 0:
                for i in range(len(message)):
                    if message[i] == possible_final_message[i]:
                        if i==len(message)-1:
                            crack = 1
                            cracked_message = possible_final_message
                            end_crack = time.time()
                            crack_needed_time = end_crack - start_crack
                            cracking_times.append(crack_needed_time)
                        else:
                            continue
                    else:
                        print("no")

        times_list.append(cracking_times)

    #Computing means and stdes
    for i in range(len(p_all)):
        values = []
        for j in range(loops):
            x = times_list[j][i]
            values.append(x)
        means = stat.mean(values)
        devs = stat.stdev(values)
        times_averages.append(means)
        times_errors.append(devs)

    #plt.errorbar(n_values, times_averages, yerr=times_errors, fmt='o')
    #print(times_errors)
    #plt.title(f"For e being: {e}, cracking")
    #plt.xlabel("key length n = p*q")
    #plt.ylabel("time in s")
    #plt.show()

    # create nth degree polynomial fit
    n = 1
    zn = np.polyfit(n_values, times_averages,n)
    pn = np.poly1d(zn) # construct polynomial

    # create qth degree polynomial fit
    q = 5
    zq = np.polyfit(n_values, times_averages,q)
    pq = np.poly1d(zq)

    # plot data and fit
    xx = np.linspace(0, max(n_values), 500)
    pylab.plot(xx, pn(xx),'-g', xx, pq(xx),'-b')
```

```
        pylab.errorbar(n_values, times_averages, yerr=times_errors, fmt='r.')

        # customise graph
        pylab.title(f"For e being: {e}, cracking")
        pylab.legend(['degree '+str(n),'degree '+str(q),'data'])
        pylab.xlabel("key length n = p*q")
        pylab.ylabel("time in s")

        pylab.show()

message =(84,117,32,101,114,101,32,117,110,32,115,97,112,111)
loops = 10
e_numbers = [3,17,65537]

p_numbers = [13,71,97,101,103,107,179,229,233,239,311,313,317,347,349,353,401,409,419,521,557,641,709,1009,1129,1

q_numbres = [191,191,191,191,191,191,191,191,191,191,191,191,191,191,191,191,191,191,191,191,191,191,191,191,191,

breaking(message, e_numbers, p_numbers, q_numbres, loops)
```