**Department of Computer Engineering**



**Cairo University**

**Faculty of Engineering**

**Computer Security – Cryptography**

**Assignment**

**RSA-Encryption**

**Submitted to:**

*Eng. Khaled*

**Submitted by:**

|  |  |
| --- | --- |
| Name | Abdelrahman Hamdy Ahmed |
| Section | 1 |
| BN | 36 |
| Code | 9202833 |
| Seat No. | 173036 |
| Email | abdelrahmanhamdy49@gmail.com |

**Introduction**

The RSA Algorithm is a widely used cryptographic algorithm that plays a crucial role in securing communications and data across the internet. With the correct key size, the algorithm may be impossible to break and that is what gives it its strength. It is a public-key encryption algorithm, meaning that it uses a public-private key pair for encryption and decryption. RSA is based on the mathematical properties of prime numbers and modular arithmetic, and its security relies on the difficulty of factoring large composite numbers into their prime factors. We will attempt such an attack in this project and we’ll also analyze different key sizes with their computation time. Now let’s dive into the steps of implementing the RSA algorithm!

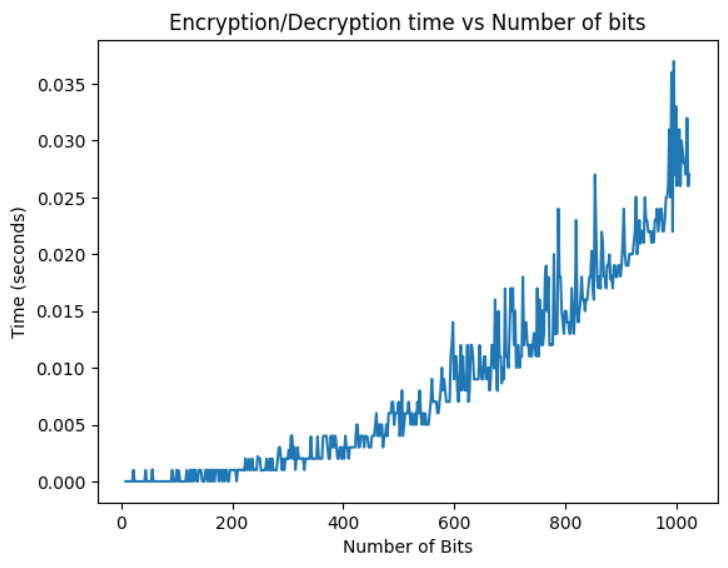
**The Algorithm**

1. Generate 2 random prime numbers p & q within a given range
2. Compute **n** 🡪 n = p \* q
3. Compute **phiN** 🡪 phiN = (p – 1) \* (q – 1)
4. Generate a random number **e** where **1 < e < phiN** such that e and phiN are **coprime** 🡪 gcd(e, phiN) = 1
5. We now have our **public key** which is represented by **(e, n)**
6. Next step is to generate the private key, so we start by first computing a number called **d** which we can get by calculating the multiplicative inverse of e modulo phiN 🡪 **d = e-1 mod phiN**
7. We now have our **private key** which is represented by **(d, n)**
8. Prompt the user for inputting the plain text and process it. This is done by changing all characters in the text to **lowercase** and **appending spaces** to the string until its length is divisible by 5. (We will know the reason behind this throughout the next steps)
9. Select the first 5 characters of the processed plain text (GROUP\_SIZE) and **encode** it into a large number by transforming each character to its corresponding number. **0 -> 9: 0 -> 9**, **a -> z: 10 -> 35**, any other characters or spaces: **36**.

10) **Encrypt** the encoded text M into cipher text C using the **public key** 🡪 **C = Me mod n**

11) **Repeat** steps 9 & 10 for the **next 5 characters** of the text and append the result into a final cipher text which will be sent to the user for decryption.

12) **Decrypt** the final appended cipher text by first obtaining the encoded message using the **private key** 🡪 **M = Cd mod n** & then decode the obtained message to restore it back to characters. This is done by reversing the operation we did when encoding the message before encryption. The decryption operation is also done in **groups of 5** & the final result is obtained from appending each output of decrypting these 5 characters.



* Key size doesn’t impact the encryption and decryption speed significantly. As shown in the above plot, with an increased number of bits, the time taken for encrypting and decrypting the message varies in an increasing random distribution which ranges from 0 to 0.035 seconds for 8 to 1024 bits. (Almost the same)

**Client-Server Connection**

To implement a chat module, I used socket programming between a server and one client. To ensure that both can send and receive at the same time, I used multithreading where there is a thread for sending messages and another one for receiving them. The output is printed in the terminal whenever a message is received or sent.

At the beginning, the server creates a socket and binds to it through the provided host and port. Then, it listens to whenever a client joins and accepts their connection based on the port number and their address. The client connects to the socket and therefore a connection is now established between the server and client.

Before chatting, the server sends the number of bits for the key prompted by the user to the client. Once both of them generates a pair of public-private keys, the server sends its public key to the client and vice versa. This is because messages sent from the server will be encrypted using the public key of the client so that he can decrypt it only using his private key. This will ensure confidentiality and also protect the secrecy of data sent across the communication line.

Once the public keys are exchanged, they are used for encrypting messages typed in. The chat is now working between both users **securely**!

**The Attack**

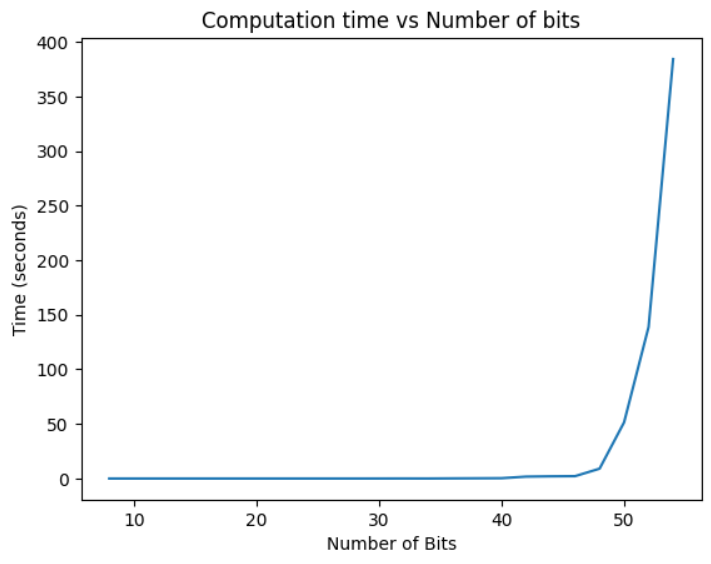
I also built a program that attempts to attack RSA and break it using prime factorization. Analysis was done on the program by inputting different key sizes for the public key. How is this done? Let’s check it out!

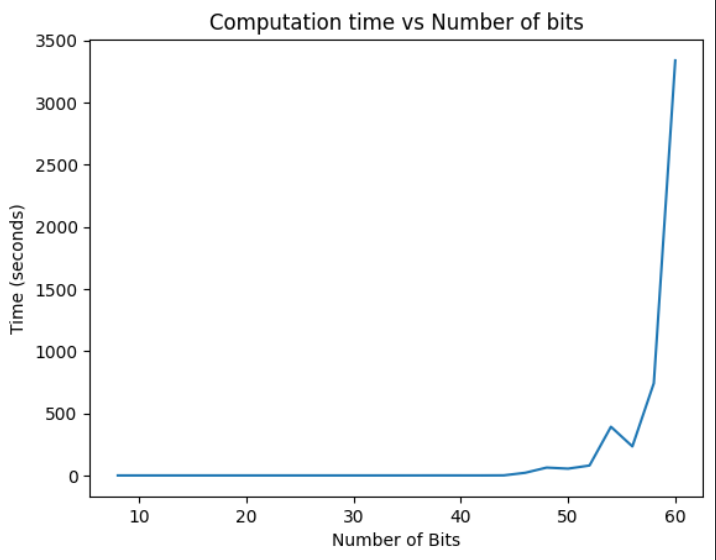
**Steps**

1. Since the attacker already knows the **public key (e, n)**… All what needs to be done is to get the **prime factors of n** by trying all numbers from the square root of n down to 2 & checking if any of them divides n. If one of the numbers works, then this is our p. We then **get q by simply dividing n by p**. The task of getting the first prime factor of n is what **takes the most time** and is essentially the main reason why RSA is very difficult to break for large key sizes.
2. Once we have got p & q, it’s now easy to obtain phiN 🡪 **phiN = (p – 1) \* (q – 1)**
3. Next, we already have **e** from the public key, so it’s easy to compute **d** just like we did before 🡪**d = e-1 mod phiN**
4. We now have the **private key (d, n)…** Therefore, we continue by **decrypting** the cipher text the same way we did above and of course decode it to obtain the actual plain text.
5. **Compare** the obtained plain text with the original true plain text and the actual private key with the obtained private key (Extra check) to ensure that the attack was successful, else failed.

**Analysis and Plot (Number of bits 🡪 Size of n)**

* **Time of algorithm breaking against Key size**





**Conclusion**

As we can see from the above graph, the time taken to break the RSA algorithm for a number of bits less than approximately 48 was minimal and didn’t even take more than 20 seconds, but as we increased the key size further more from 50 and beyond, it took the attack much more time to compute the prime factorization of n & decrypt the message. For example, for a key size of 54, the computation time was 380 seconds while for a key size of 32 (Size of p & q 🡪 16 bit), it was roughly less than a second. The plot hit an exponential rising curve for a 60 bit key, the time was more than 3000 seconds! We can also notice that for a 56 bit key, the time taken decreased and this is purely because the operation for getting the prime numbers is completely random and could vary in the times for any number of bits. For a 64 bit key & more, the time taken was too long that I couldn’t record it and this shows how it’s difficult to break it with such a key, keeping in mind that in practice, a key of size 1024 is used! Imagine how long it would take to calculate the factors of n, yes, forever!