Cairo University
Faculty of Engineering

Computer Engineering Department
Third Year

Cryptography and Security
CMP3050
RSA Project
Delivery Document

Ahmed Ihab sec: 1 BN: 2
Ahmed Gamal AbdelSamie sec: 1 BN: 3

Fall 2022

Overview

We have implemented the RSA encryption algorithm using a mixture of Python scripts (.py) and Python Interactive Notebooks (.ipynb). The project is divided into utility functions, chat module using sockets, RSA encryption and decryption, brute force attack, key length vs. time, and chosen ciphertext attack. We have also implemented a GUI and a chat module using Flutter, and Flask.

How to Run

We have used vscode with python and python notebook extensions installed. Any IDE or code editor will be able to run the .py scripts, and the .ipynb files may be run using Google Collab, Jupyter, Kaggle, etc.

The chat app needs vscode, Flutter, and Flask to run

Directions and Explanation

RSA

We have implemented RSA through the following steps:

1- We generated the primes p,q if not supplied by the user using a random number generator and Fermat's primality test, then get n, Φ(n), e, and d
2- Check if the message needs to be divided into blocks, if so it is divided into an array, changed into integers, then encrypted
3- The receiver receives the array of encrypted integers, decrypts each chunk, converts them back to a string, and then concatenates them.

The following is a list of the most important utility functions and their explanation:

- GCD(a,b)

A recursive function that takes numbers a and b and keeps trying to find the GCD of b and a%b

- extendedEuclid(a,b)

finds x and y by recursively being called using b and a%b

- modularExponentiate(base,power,modulo)

uses the square and multiply method according to the reference

- modularInverse(a,n)

uses extendedEuclid and makes sure the result is non-negative

- convertToInt(str)

converts a string to an integer by multiplying by 256 (2^8) and adding the Unicode representation of each character

- convertTostr(int)

converts and integer to a string by repeated addition mod 256 and converting to Unicode

- Encrypt(msg,e,n)

Converts a msg to an integer and uses modular exponentiation

- Decrypt(cipher,d,p,q)

Uses modular exponentiation then converts the result to a string

- divideMsg(msg,n)

since any message represented as an integer can never be bigger than 2^8 power the length of the message, i.e 2^{8*msg_len} , using log to the base 256 we find the length of the needed block of a message that can be safely transmitted and divide the string according to that size

- EncryptEncompass(msg,exponent,modulo)

Combines Encrypt and divideMsg into a single function

- DecryptEncompass(cipher_blocks, d, p, q)

Combines Decrypt and string concatenation into a single function

- getPublicKey(phi_n)

finds a random integer less than phi_n and also co-prime with it

- getPrivateKey(e, p, q)

computes phi_n and finds the modular inverse of e mod phi_n

- nBitRandom(n)

generates a random integer of n binary bits

- fermatPrimalityTest(p)

asserts the relation $a^p = a \ mod \ p$ holds for a 100 different values of a

- generatePrime(n)

generates a random integer of n bits and keeps testing for primality

Chat Module

The chat module is implemented using web sockets, Flutter, and Flask. It allows for real time communication by entering a message, encrypting it, and decrypting it at the receiver. The behind the scenes work of encryption, decryption, and testing is illustrated in json files.

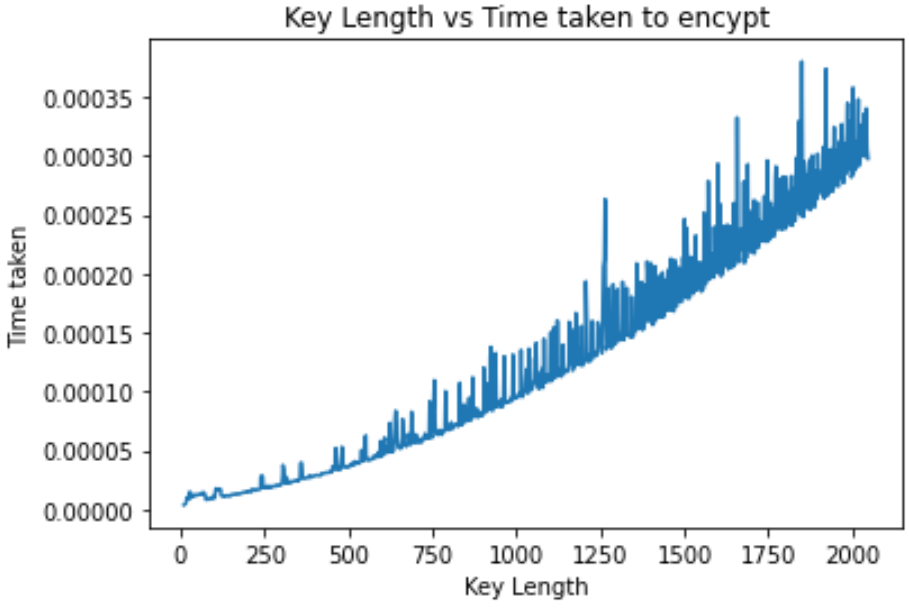
A user has the ability to sign up using his own values of p,q, and e. If any of them is not supplied, they are randomly generated using functions described in the previous section.

Encryption Time vs. Key Length

We start by generating numpy arrays of n phi_n, while maintaining constant e. These 2 arrays are saved as .npy files to be later reused in the graphing function instead of recomputing the primes each time, as it may take a lot of time.

The graphing function encryptionTimeConstE(n) then tries to encrypt a constant message using a constant e (starts small at first then saturates at large values) and different values of n and phi_n. It takes an average of a given number of iterations for encrypting the message, records the time, and graphs it against the key length up to 2048 bit keys.

The graph depicts length of key in binary bits against time taken for encryption in seconds. The graph seems to be exponential which is logical.



This is also the reason why public cryptography systems are used for key distribution and not for sending large messages.

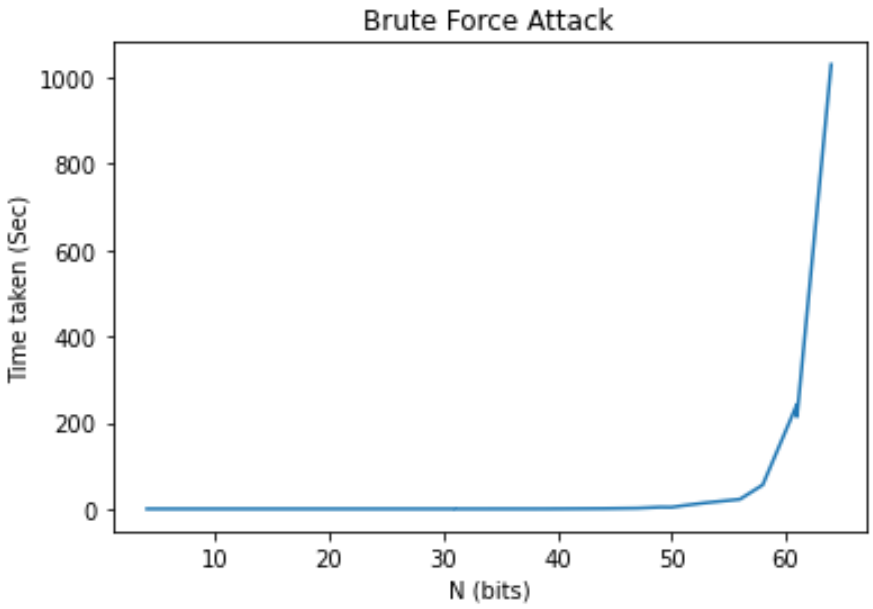
Note: **the implementation of the encryption time vs. key length is in the encryptionTime.ipynb file**

Brute Force Attack

We have implemented the brute force attack by trying to divide n using values starting from 3 till $sqrt(n)$ since the factorization of a number may be found only until its square root. We have tried this process for various values of n and plotted the time needed to find the factorization of n.

Note that the primes are generated in a separate function and saved to a text file, since generating them takes a very long time.

The graph depicts brute forcing against n up till 64 bits and graphs the time in seconds. It follows an exponential distribution, which is reasonable. Increasing the key length increases the time needed to try to break it exponentially, since the key space increases by $2^{increase\_in\_key\_length}$ each iteration



Note: **the implementation of the brute force attack is in the rsa.ipynb file**

Chosen Cipher text Attack

The chosen cipher attack starts by intercepting any cipher text *C* and multiplying it by a factor r^e where is r is a random integer co-prime with n,

and e is the public key, this step is possible with the knowledge of the public key {e, n}

$$C' = C * r^e mod\ n = M^e * r^e mod\ n$$

The attacker then sends *C'* to the sender asking to be decrypted using his private key, so now the sender unsuspectingly computes

$$Y = C'^d = (M^e * r^e)\text{^}d\ \ mod\ n = M * r\ mod\ n$$

This holds since d is the modular inverse of e. Now all the attacker has to do is divide by r, or multiply by its multiplicative inverse, which is possible since r and n are chosen to be co-primes.

The chosen cipher text attack is modeled as a function that takes the cipher text, e, and n. It also takes p and q to simulate the sender's consent to decrypt the attacker's message using his own private key. The function follows the mentioned steps using the utility functions described earlier.

The attack is tested on an input file of random strings. The input file is first encrypted, then the function outputs the cipher and decrypted lines into two other text files. Please note the following file structure:

- input.txt: contains random sentences
- chosenCipher.txt: contains cipher text of input
- chosenCipherOutput: contains the decrypted cipher text after the chosen cipher text attack

Note: **the implementation of the chosen cipher text attack is in the chosenCipherText.ipynb file**