Table of Contents

INTRODUCTION

In fulfilment of the introduction to python programming course, we carried out this programming project to create an encryption-decryption application. For the purpose of the project, we created a graphic user interface which has 2 options, encryption and decryption, The user inputs the public key, and the text to be encrypted, then encrypts it (receives encrypted version of text); and the receiver can input the encrypted text in and his private key and decrypts the text using the decryption button (receives decrypted original text).

In real life it can be used as a stand-alone application, where if a user want to send an encrypted email to others. First encrypt the text then copy it and send to the intended person. Then the reciever can decrypt it with a private key previously agreed on. It could also be included as a module to any messaging system, for encryption before forwarding. In actual practice though there are better algorithms to use such as discrete logarithm problem algorithms, but the idea is basically the same. We suggest RSA because it is sufficiently difficult to secure transmission of message from unwanted access, and not extremely difficult and over complicated to implement.

PROBLEM DESCRIPTION

In digital communication, privacy is critical. Sending and receiving different types of data using open communication channels makes it possible for the transmitted message to be manipulated by any third party who can access the media. This creates problem of ensuring the integrity and confidentiality of the message. Hence when we are transmitting large volumes of personal data in the internet we need a guarantee that the message is recived by the intended destination, not changes, and with its content secured. For this to happen, there are different ways of securing electronic communications such as encryption which is one of the commonly applied methods for securing transmission of messages.

PROPOSED SOLUTION

The proposed solution applies public key cryptography algorithm known as RSA (Rivest–Shamir–Adleman). Basically, it is used to secure data in transmission between sender and reciver. The encryption key can be a public key , however the decryption key is private only known to the receiver

To create those encryption and decryption keys, there are certain steps to be taken. The decryption key has to be difficult to guess or to break. The intention of encryption is to make the decryption difficult it will be impossible to identify the key and decrypt the communicated message.

Therefore to generate the decryption key which is unique, large and combination of random large prime numbers, some stepwise computations has to be performed.

RSA ENCRYPTION ALGORITHM

In this problem solution the identification of a large prime is key factor. Therefore we used the algorithm, Miller–Rabin prime test in order to handle primes of large sizes. It is based on number theory. The theory assumes such that "p" is assumed as a large prime number different from 2 then check for being even. The algorithm works for positive if it is negative the absolute value is used. Then we have large number (P-1) = even. The largest power of 2 -1 is how we get the solution This is key factor for computing the RSA. The following section presents the structure of the program.

STRUCTURE OF THE SOLUTION

To present eaily understandable solution for readers we have classified the overall RSA solution in three parts. The GUI sub program which handles the user interaction with responses expected as well as validation of inputs. The encryption handles most related sub programs from creating random numbers, testing for primality, performing the Euclidian algorithm to obtain the inverse, which is used to create the private and public key, and other subprogram listed below. The final category focuses in decryption of the intended message using the decryption key.

The following are the list of sub programs used.

- **def SQN(base, exp, mod)**: Fast modular exponentiation of large numbers using the Square multiply algorithm.
- **def higBitOrder(n)**: Checks and returns the maximum power of '2' that can be represented in a given number
- **def lowBitOrder(n)**: Returns the maximum power of '2' that divides the given number.
- **def GCD(a,b)**: Returns the greatest common divider of two numbers.
- **def egcd(a, b)**: This extended Euclidian algorithm returns gcd of large primes in a with attached Bezout coefficients (multipliers).

- **def multiplicative_inverse(num, mod):** Retrieves the inverse from the Bezout numbers in the egcd sub program.
- **def millerRabin(n)**: Performs miller-Rabin algorithm to test for the primality of the chosen selected random numbers.
- **def randomPrimeGen()**: Returns random prime number.
- **def modulusCalc()**: Using the random number generated (p,q) returned unique number for encryption sun function .
- **def publicKeyGen()**: Performs the calculation and returns "d" needed for encryption.
- **def privateKeyGen(pubkey)**: Returns private key for decryption.
- **def encryption(plaintext)**: Returns the given text in encrypted format.
- **Def decryption(cipherTuple)**: By using the decryption formular returns the decrypted message.
- **def GUI()**: Function for graphical user interface.
- **def encrypt_click():** Function that handles encrypt button actions
- **def decrypt_click():**Function that handles the decrypt button actions and input validation

DETAILED DESCRIPTION OF SUB-PROGRAMS

I. def SQM(base , exp, mod)

The Square and multiply subprogram for exponentiation performs modular exponentiation by taking string equivalent of binary representation of the exponents. Then, depending on whether you get a 0 or 1 you square or (square and multiply). This sub program takes three arguments: the base which is the main value, exponent of the base, and the modulus. It then perform the exponentiation returns the result.

II. def highBitOrder(n)

This checks and returns the maximum power of two possible in a given number. This is needed in calculating the corresponding low bit (maximum power of '2' which divides it). The binary number for example 01000011 has decimal equivalent 67, or 64+2+1. The highest bit order corresponds to the 64.

The subprogram takes one argument “n” being the given number as explained in the above paragraph and returns the highest bit used to get maximum power of “2” represented in the given number. This is required especially for large integers (encryption keys are supposed to be large primes to be more secure).

III. def lowBitOrder(n)

This sub program is a function that relates to the above one.. As indicated in the previous HighBitOrder function in this case we need to find maximum power of ‘2’ that divides the given number. After calling the previous function, using as an argument the value of the logical expression “n AND –n”, it then returns the resulting low-bit-order.

IV. def GCD(a,b)

This function performs mathematical computing of finding the greatest common division for two given numbers and returns the result. The sub program takes two arguments as step one and checks if the divisor is different from zero. Then by assigning/swaping the numbers continues to find if the division gives a value of zero when perform “%” operation, and the corresponding greatest divisor. This is significant in confirming that the gcd of the two primes is 1 as required for encryption purposes.

V. def egcd(a, b)

This subprogram performs the extended Euclidean algorithm function. In RSA the identification of the Bezout coefficients in the euclidean algorithm is significant in obtaing the inverse number of a prime needed in the key generation. For smaller numbers it is easy to find if their GCD to be “1” or not. However as the number increases in value other methods (extended Euclidian algorithm) is needed to get to the result with simpler computation.

This sub program takes two arguments, (numbers), calculates their gcd and returns the corresponding bezout coefficients of the large primes.

VI. def multiplicative_inverse(num, mod)

This subprogram which is linked to the egcd simply confirms that the gcd equals 1 and then subsequently picks out the inverse (one of the 2 bezout coefficients). We need this mathematical computation to find sub parts of encryption key.

VII. def millerRabin(n)

This sub program is the core of the project. Miller-Rabin is a probalistic algorithm that is used to test for primality and find the needed prime numbers to perform encryption and provide the encryption key. Generally what is performed in the encryption process can be best described as follows:

- Finding a modulus n = pq, where p and q are primes.
- Calculating the order of 'n' as u(n) = (p-1)*(q -1).
- Generating the public key of RSA as a pair (n; e), where the greatest common divisor of e and u(n) = 1.
- Generating the private key as a pair (n; d), where d is the inverse of e modulo u(n), i.e. ed = 1 (mod u(n)).
- The encryption and decryption functions are as follows:
  - ♣ E(x) = x^(e) (mod n)
  - ♣ D(x) = y^(d) (mod n)

Hence it is vital to have a fast means of confirming the primality of the chosen random numbers.

How the algorithm works is; the subprogram accepts as an argument a random number then checks if the number is even or not to quickly eliminate those numbers and save computation power when possible. If it is even it drops it and returns 'False'. When the condition is satisfied, it checks if the number is in the first 200 prime list also to save computation power, if prime it returns 'True'. If not then it continues to find the low-bit-order of the number in order to find the divisor gotten with this low-bit-order. This divisor is then used in finding a "witness" (a number) which proves that our random number is "probably a prime" or conclusively prove that is is definitely not a prime (this).

The function is iterated a couple of times to increase the "probability", in the second part of miller rabin; and the unchanging values are stored in a dictionary list to save costly calculations.

VIII. def randomPrimeGen()

This sub program is needed to find the random prime numbers specified in a specific range of number. In this project case it is limited between 256-65536. Then the sub program returns random prime number for the operation.

IX. def modulusCalc()

This sub program calculates the modulus by using the random numbers generated (p,q) and returns a unique number for encryption subfunction .This operation is needed to generate the number that will be used in calculation of the keys needed for RSA.

X. def publicKeyGen()

This sub program uses the previous described sub programs such as random prime number generator, modulus calculation, and then checks the selected numbers GCD as well as millerRabin part in a while statement. After the condition is satisfied and computation performed, the function returns a public key tuple.

XI. def privateKeyGen(pubkey)

This subprogram is needed to create the private key that is required for the user to decrypt the received message. After identifying (e,mode2,phi) from the public_key tuple received as an argument, the function finds private key “d” by calling the sub function that performs the inverse. After it performs the mathematical operation it returns private key for decryption.

XII. def encryption(plaintext)

This subprogram’s main task is encrypt the given plain text by using public key and outputing an encrypted text for transmission. The sub program takes one argument which is the plain text. Then it uses the public key which is generated by the function call of publicKeyGen and SQM function to perform the necessary operation of encrypting the ASCII values of the plain text. Finally returns the given text in encrypted format.

XIII. def decryption(cipherTuple)

This sub program carries out the decryption and returns the decrypted message. It takes one argument which is the encrypted text. By using the decryption key which is the private key, it performs the operation to get the original ASCII values of the message (plain text) and convert’s these to characters. So decrypting the crypto-text gives back the plaintext and the system works.

XIV. def GUI()

This Function creates the graphical user interface, creates the textbox for inputing and outputing text for decryption and encryption. It also creates the buttons for decryption and encryption as well as textbox area for inputting the modulus and the private key when decrypting.

XV. def encrypt_click()

This function handles the actions which occur when the encryption button is clicked, linking the encryption functions with the text to be encrypted, and subsequently displaying the encrypted text.

XVI. def decrypt_click ()

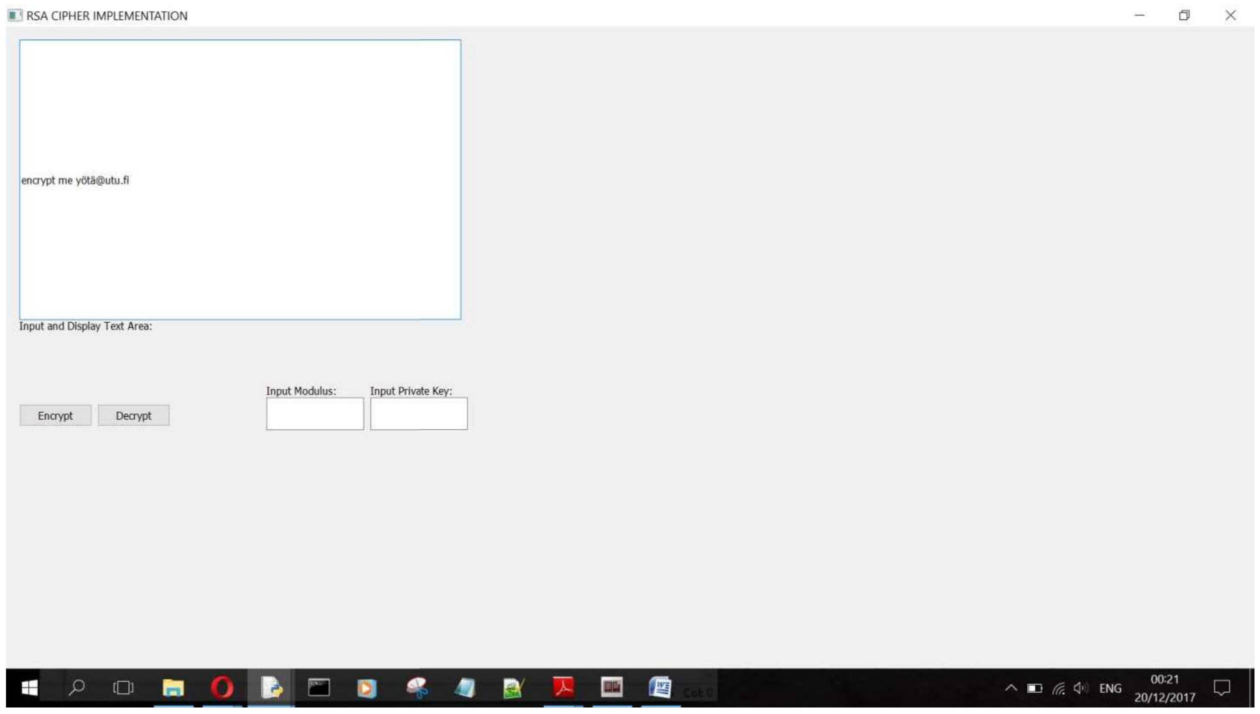
This function handles the actions which occur when the decryption button is clicked, linking the decryption functions and other related functions with the text to be decrypted, the modulus and the private key, and subsequently displaying the decrypted text. It also validates the input by the user, checking if the are in the appropriate format.

CONCLUSION

Cryptography is one of the fields that improves the secure transmission of data between sender and receiver. There are several types of cryptographic applications and algorithms in the field. Encryption using RSA can be applied in different area for securing data. In this project we tried to design RSA encryption technique in python programming language. For the algorithm to work we implemented several sub programs to perform specific tasks of the mathematical computation to arrive to the needed output. The public key and the private key must be generated in a way that will reduce the risk of being easily identified or broken easily by unauthorized users.

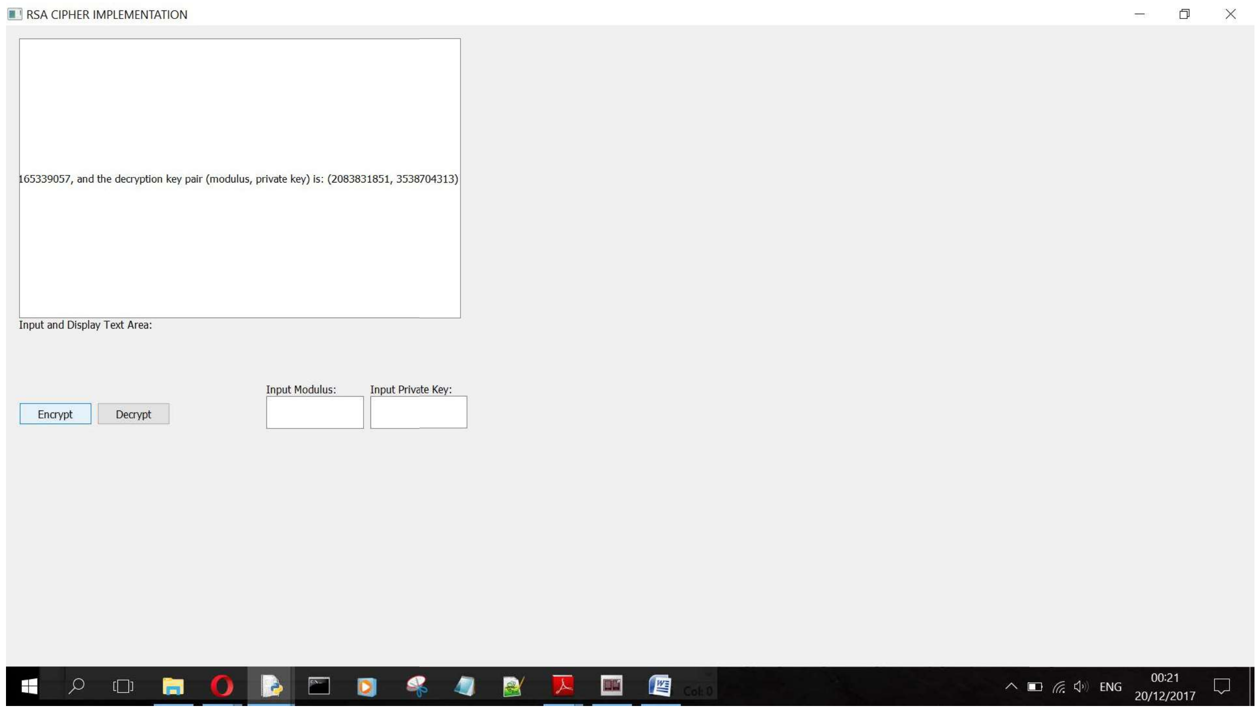
However, our solution has some inherent weaknesses; such as the using the python generated random numbers which are just pseudo random numbers and have a high propability of being guessed by skilled adversaries. Similarly our encryption is displayed as a list of distinct numbers, making it quite possible to run analysis and attempt hacks simply by picking the first elements of the encryption and the plain text. Also due to limitations in our calculation of the low bit order, our algorithm can only use keys not greater than 64bits which is comparatively easy to crack. There might be need for slight modifications of the code to accommodate other python versions. Furthermore despite RSA being a good encryption algorithm however it relies on the difficulty of factorization problems which in comparison, discreet logarithm problems are better and much more difficult problems for use in security. Even though discreet algorithm is better than RSA we have choosen to carry out the project using RSA , since the level of complexity of other NP-hard problems and implementation of their algorithm are temporarily beyond our capacity and scope of this course.

IMPLEMENTATION SCREENSHOTS



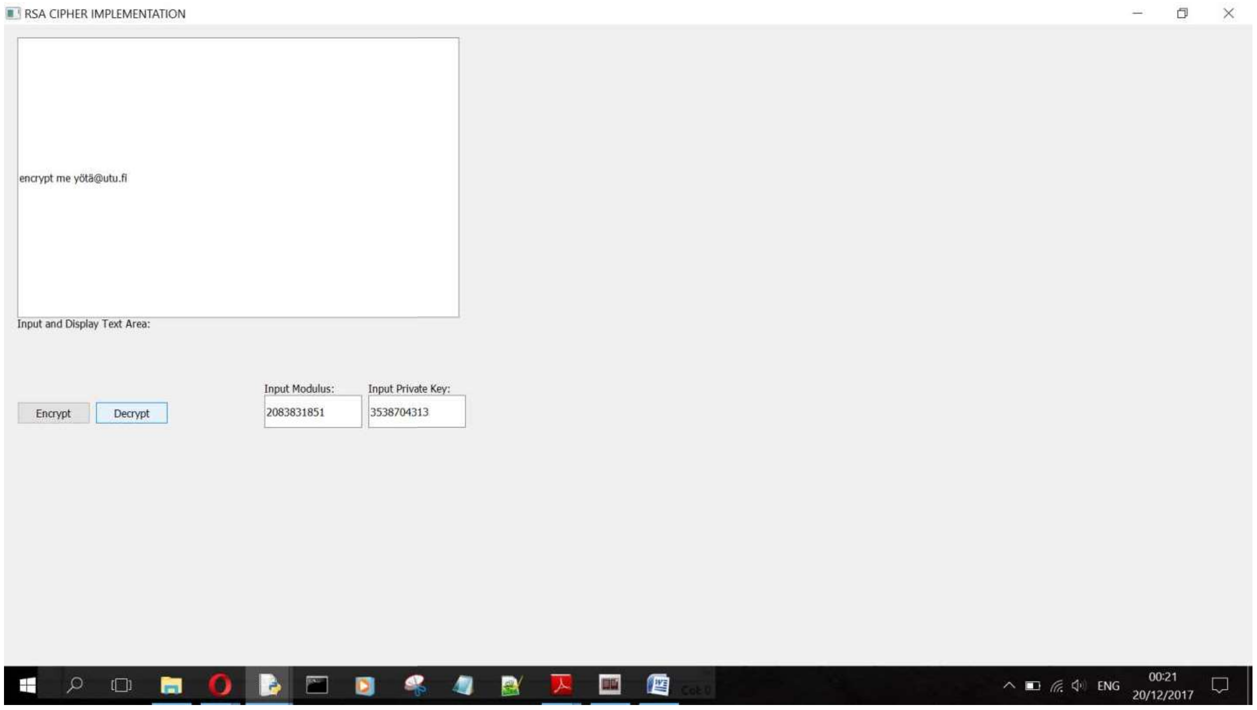
encrypt me yötä@utu.fi

Fig 1. Input



165339057, and the decryption key pair (modulus, private key) is: (2083831851, 3538704313)

Fig 1. Encryption



encrypt me yötä@utu.fi

Input Modulus: 2083831851

Input Private Key: 3538704313

Fig.2 Decryption