

Final Year Project
RSA Public-Key Cryptography

By: Stephen Jones

Supervisor: Dr Ana Salagean

Abstract

Computer security is becoming increasingly important to companies and organisations all over the world. Computers are being used for communications between companies that may be in different cities or even different countries. These communications must be encrypted to ensure they remain secure while in transit. This project looks at the general principles of cryptography. It then focuses on RSA, a public key cryptographic algorithm. This algorithm is implemented in three stages, firstly using small numbers, to show the mathematics behind the algorithm. The second implementation extends this to using the GNU Multiple Precision Library, to allow the program to handle large numbers. The third implementation extends this further to include simple text file encryption.

Contents

1	Introduction	5
1.1	Cryptography	5
1.1.1	Symmetric (Secret-Key) Cryptography	6
1.1.2	Asymmetric (Public-Key) Cryptography	6
1.1.3	Symmetric Vs. Asymmetric	6
2	Project Goals	7
2.1	Required Objectives	7
2.2	Optional Objectives	7
3	Requirements.....	7
3.1	Understanding the Algorithms.....	7
3.1.1	Key Generation	7
3.1.2	Encryption and Decryption.....	7
3.2	Large Number Key Generation	7
3.3	Large Number Encryption and Decryption	8
3.4	File Encryption and Decryption	9
3.5	Performance.....	9
3.6	General Requirements	10
3.7	Optional Requirements	10
4	Background to RSA	10
4.1	What is RSA	10
4.2	History of RSA	10
4.3	Uses of RSA.....	10
4.4	Strength	10
4.4.1	The RSA Problem	10
4.4.2	The Problem of Factoring Large Numbers	11
4.5	Algorithms Used	11
4.5.1	Public/Private key generation	11
4.5.2	Encryption/Decryption.....	11
5	Work Plan.....	12
6	Implementation 1 – Small Numbers	12
6.1	Research and Design	12
6.2	Implementation of Code	13
6.2.1	Menu – menu.c.....	13
6.2.2	Key Generation – gen_keys.c	13
6.3	Encryption/Decryption – cipher.c	14
6.3.1	cipher().....	14
6.4	Testing.....	14
6.4.1	Testing 1 – Key Generation:.....	14
6.4.2	Testing 2 – Encryption/Decryption:	16
6.5	Evaluation	17
7	Implementation 2 – Large Numbers	18
7.1	Library Choice.....	18
7.1.1	BigDigits Library	18
7.1.2	GMP Library.....	18
7.2	Function Definitions.....	18
7.3	Code	19
7.3.1	Menu – menu_big.c.....	20
7.3.2	Key Generation – gen_keys_big.c.....	20
7.4	Encryption/Decryption – cipher_big.c.....	20
7.5	Testing.....	21
7.5.1	Key Generation, Encryption and Decryption	21

- 7.5.2 Key Length 23
- 7.6 Evaluation 24
- 8 Implementation 3 – File Encryption 24
 - 8.1 Function Definitions..... 24
 - 8.2 Code 25
 - 8.2.1 Menu – menu_file.c 25
 - 8.2.2 Key Generation – gen_keys_file.c 25
 - 8.3 Encryption/Decryption – cipher_file.c 25
 - 8.3.1 encrypt_file() 25
 - 8.3.2 decrypt_file() 26
 - 8.4 Testing..... 26
 - 8.4.1 Key Generation 26
 - 8.4.2 Encryption and Decryption..... 26
 - 8.5 Evaluation 27
- 9 Performance Analysis 27
 - 9.1 Key Generation 27
 - 9.1.1 Results 28
 - 9.2 Encryption/Decryption..... 28
 - 9.2.1 Results 28
 - 9.3 Evaluation 28
- 10 Evaluation & Conclusion 29
 - 10.1 Project Achievements 29
 - 10.2 Problems Encountered..... 29
 - 10.3 Personal achievements 29
 - 10.4 Final Conclusion 30
- 11 Further Work..... 30
 - 11.1 Binary File Encryption 30
 - 11.2 Chinese Remainder Theorem 30
 - 11.3 Open PGP..... 30
- 12 Acknowledgements 31
- 13 References 31
 - 13.1 Websites 31
 - 13.2 Books..... 31
 - 13.3 Papers..... 31
 - 13.4 Libraries..... 31
- 14 Bibliography 32
 - 14.1 Books..... 32
 - 14.2 Websites 32
 - 14.3 Papers..... 32
- 15 Appendix A – Gantt Chart 33
- 16 Appendix B – Testing 35
- 17 Appendix C – Key C Source Files 41
 - 17.1 Small Number Implementation – menu.c..... 41
 - 17.2 Small Number Implementation – gen_keys.c 43
 - 17.3 Small Number Implementation – cipher.c..... 46
 - 17.4 Large Number Implementation – gen_keys_big.c 48
 - 17.5 Large Number Implementation – cipher_big.c..... 50
 - 17.6 File Implementation – cipher_file.c..... 52

RSA Public-Key Cryptography - Implementation and Evaluation

1 Introduction

Computer security is a growing problem for companies and organisations all over the world. Data and information must be kept secure and protected against unauthorised access. This problem is fairly simple when data is stored on a single computer; the computer can be locked in a room and require users to login to get access to it. When data needs to be shared between a number of different computers the problem of keeping it secure is much greater.

Networks and the Internet are getting more widely used for data transmission and email, therefore it is becoming increasingly important to ensure that any data transmitted across an insecure network is encrypted to prevent unauthorised access.

As technology advances and the processor speeds of computers increase, existing cryptography methods become weaker and easier to crack, because of this there is a need for them to be updated and improved to ensure they stay strong and the transmitted information remains secure. Creating new algorithms and making improvements to existing ones should ensure this.

1.1 Cryptography

Cryptography is the science of keeping messages secure. It involves ciphering and deciphering messages usually for transmission over some unsecured network such as the Internet.

The definition of cryptography as given by www.webopedia.com:

“The art of protecting information by transforming it (encrypting it) into an unreadable format, called cipher text. Only those who possess a secret key can decipher (or decrypt) the message into plain text. Encrypted messages can sometimes be broken by cryptanalysis, also called code breaking, although modern cryptography techniques are virtually unbreakable.”

The process to transmit a message over a network is as follows:

- The sender and recipient agree on an encryption algorithm or method.
- These algorithms are mathematically proven to be secure and usually use an extra parameter called a key. The key, or at least part of it, should be kept private so that only the sender and recipient know it.
- The original message, known as the ‘plaintext’, is encrypted into a coded equivalent using the pre-defined algorithm
- This encrypted message, known as the ‘ciphertext’, can then be securely transmitted over an unsecured network.
- If the message gets intercepted during the transmission it is almost impossible for somebody to read without knowing the encryption algorithm and having access to the key.
- The recipient can then decrypt the ciphertext using the decrypting part of the same algorithm that was used to encrypt.

There are two types of cryptography:

- Symmetric Cryptography (also known as secret-key cryptography)
- Asymmetric Cryptography (also known as public-key cryptography)

1.1.1 Symmetric (Secret-Key) Cryptography

This method of cryptography involves using a single key, which only the sender and recipient know. Both the sender and the recipient use the same key to encrypt and decrypt message. This method of cryptography takes less computing power to encrypt and decrypt messages than the asymmetric method, therefore it is a much more efficient method. The main problem with symmetric cryptography is that the key must be sent using a different method before it can be used; an asymmetric method is usually implemented for this.

DES, AES and Blowfish are all examples of symmetric cryptography.

1.1.2 Asymmetric (Public-Key) Cryptography

The second form of cryptography is Public-Key cryptography. This involves generating a key pair, consisting of a public key and a private key. In the simplest terms, the key is a single number or set of numbers, which in some way are mathematically related. The public key is used to encrypt a message and the private key to decrypt it. The recipient makes their public key available to the sender, usually via an Internet or network repository. The sender then encrypts the message with an algorithm that uses the recipients' public key as a parameter. The message can then be transmitted to the recipient. Once encrypted a message can only be decrypted using the matching private key from the key pair. To decrypt the message a second algorithm is applied to the ciphertext using the private key.

As the name suggests the public key is made publicly available. Therefore if a person intercepts the message, they will also have access to the public key. The keys in the key pair must be related in a way such that it is not possible to calculate the private key from the public key, but still used together to encrypt and decrypt messages.

RSA is the most well known asymmetric cryptography.

1.1.3 Symmetric Vs. Asymmetric

Deciding whether to use symmetric or asymmetric cryptography is an important decision for anyone who is planning to send encrypted messages. Symmetric cryptography is less computationally intensive than Asymmetric therefore would be the preferred method. But this has the added problem of securely communicating the key to both the sender and recipient. Meeting face-to-face to exchange the key would be the most secure way of doing this but is not always practical. One solution to the problem is to use asymmetric cryptography to encrypt a key that is then sent securely, via a network or the Internet. This key can then be used for symmetric cryptography to send the original the message.

The focus of this project is the RSA algorithm, a form of asymmetric cryptography. RSA is used in TLS and SSL to communicate the master secret key between the sender and receiver before using a symmetric method such as AES to send the bulk of the transmission.

2 Project Goals

The overall aim of this project is to implement the key generation, encryption and decryption algorithms as specified by the RSA specification. The implementation will have to be as efficient as possible to ensure quick execution times of the various parts.

2.1 Required Objectives

- Gain knowledge and understanding of the RSA algorithms for key generation, encryption and decryption.
- Create a program to generate RSA public and private keys of at least 2048 bits.
- Create a program capable of encrypting and decrypting large numbers using RSA.
- Create a program capable of encrypting and decrypting simple text files using RSA.
- Ensure the file encryption program is efficient and that the time taken to generate keys and encrypt or decrypt files is reasonable.

2.2 Optional Objectives

- Implementing the Chinese Remainder Theorem to help with the decryption of messages.
- Looking into OpenPGP keys and using them as the public/private key pairs in my encryption/decryption programs.

3 Requirements

3.1 Understanding the Algorithms

3.1.1 Key Generation

I will start by trying to understand the algorithms involved in RSA encryption. To help me do this I will implement a program that will generate the public and private keys using small numbers. Small numbers are those that will fit into the integer data type (–32,768 to 32,767 for integer, 0 to 65,535 for unsigned integer). These programs will be implemented without using any special library functions.

3.1.2 Encryption and Decryption

I will also have to understand the algorithms needed to encrypt and decrypt numbers and files. First I will try to implement the encryption and decryption algorithms using small numbers and no library functions.

As small numbers are being used in this section of the implementation, the key generation, encryption and decryption will be faster than with longer keys but will be extremely insecure.

3.2 Large Number Key Generation

Once I am sure I have a good enough understanding of the algorithm used to generate the public and private key pair, I will continue with implementing a program to create 2048 keys. As this will require using very large numbers, I will need to use special library functions to be able to handle them. The keys should be generated as quickly as possible and should conform to the standards given in the

RSA specification as shown below. The full specification can be found on the RSA website: <http://www.rsasecurity.com>.

Definition of the RSA Public Key, Section 3.1 of the RSA specification:

“... an RSA public key consists of two components:

n the RSA modulus, a positive integer
e the RSA public exponent, a positive integer

In a valid RSA public key, the RSA modulus n is a product of u distinct odd primes r_i , $i = 1, 2, \dots, u$, where $u \geq 2$, and the RSA public exponent e is an integer between 3 and $n - 1$ satisfying $\text{GCD}(e, l(n)) = 1$, where $l(n) = \text{LCM}(r_1 - 1, \dots, r_u - 1)$. By convention, the first two primes r_1 and r_2 may also be denoted p and q respectively.”

Definition of the RSA Private Key, Section 3.2 of the RSA specification:

“... an RSA private key may have either of two representations.

1. The first representation consists of the pair (n, d) , where the components have the following meanings:

n the RSA modulus, a positive integer
d the RSA private exponent, a positive integer

[...]

In a valid RSA private key with the first representation, the RSA modulus n is the same as in the corresponding RSA public key and is the product of u distinct odd primes r_i , $i = 1, 2, \dots, u$, where $u \geq 2$. The RSA private exponent d is a positive integer less than n satisfying $e \cdot d \equiv 1 \pmod{l(n)}$, where e is the corresponding RSA public exponent and $l(n)$ is defined as in Section 3.1.”

The second method of showing the RSA Private key mentioned above is for use with the Chinese Remainder Theorem which is part of an optional objective and will not be looked at in this project unless time allows.

3.3 Large Number Encryption and Decryption

Once the key generation algorithm has been implemented using large numbers I can go on to implement the encryption and decryption algorithms. Again I will need to use the special library functions to implement this section. This implementation must also meet the standards given in the RSA specification as shown below:

Definition of RSA encryption, Section 5.1 of the RSA specification:

“RSAEP $((n, e), m)$

Input: (n, e) RSA public key
 M message representative, an integer between 0 and $n - 1$

Output: c ciphertext representative, an integer between 0 and $n - 1$

Error: “message representative out of range”

Assumption: RSA public key (n, e) is valid

Steps:

1. If the message representative m is not between 0 and $n - 1$, output “message representative out of range” and stop.
2. Let $c = m^e \bmod n$.
3. Output c .”

Definition of RSA encryption, Section 5.1 of the RSA specification:

“RSADP (K, c)

Input: K RSA private key, where K has one of the following forms:

— a pair (n, d)

[...]

c ciphertext representative, an integer between 0 and $n - 1$

Output: m message representative, an integer between 0 and $n - 1$

Error: “ciphertext representative out of range”

Assumption: RSA private key K is valid

Steps:

1. If the ciphertext representative c is not between 0 and $n - 1$, output “ciphertext representative out of range” and stop.
2. The message representative m is computed as follows.
 - a. If the first form (n, d) of K is used, let $m = c^d \bmod n$.
- [...]
3. Output m .”

Again the second method referred to in the decryption section relates to the Chinese Remainder Theorem.

3.4 File Encryption and Decryption

Once I have successfully implemented the algorithms to generate large keys and used them to encrypt and decrypt large numbers I will go on to implement a program capable of encrypting and decrypting simple text files. This implementation should be as efficient as possible.

3.5 Performance

In cryptography the efficiency of a program is important. If it takes too long to generate a key pair or encrypt and decrypt a message people will not spend the time using it. This increases the risk of the messages being sent unencrypted. Therefore I must ensure that the time taken for my program to generate keys and encrypt or decrypt a file is reasonable based on the key size and file size.

3.6 General Requirements

The programs will be coded using the C programming language with which I will have to re-familiarise myself before I start this project.

None of the programs will need a complex graphical interface; a simple command line menu system will be enough. This should remove some of the complexity of the coding and removes the overhead required to display a graphical interface.

3.7 Optional Requirements

If time is available I will look into the use of the Chinese remainder theorem to help with the programming of the decryption algorithm. This should help to decrease the time it takes to decrypt numbers or files using my program.

I could also look into the use of OpenPGP certificates and digital signatures for encryption and decryption and their compatibility with my implementations.

4 Background to RSA

To help with my understanding of RSA and the algorithms involved, I will first look into the background of RSA, where it has come from, what it is used for and its strengths and weaknesses.

4.1 What is RSA

RSA is a public-key based asymmetric cryptography algorithm which is used for encryption, decryption and signing. It was one of the first algorithms created for this purpose and it is still widely used today. It is considered a very secure algorithm if used with sufficiently long key lengths.

4.2 History of RSA

RSA was originally described in 1977 by three mathematicians; Ron Rivest, Adi Shamir and Len Adleman and takes its name from the initials of these three people. It was patented by MIT in the United States in 1983, the patent expired in 2000.

4.3 Uses of RSA

The RSA algorithm became a standard encryption method for many pieces of software, especially Internet related software such as Microsoft Internet Explorer and Netscape Navigator. It is also used in the TLS (Transport Layer Security) and SSL (Secure Socket Layer) protocols which provide security for Internet communications such as email and web browsing.

4.4 Strength

As there is no known algorithm which could break the RSA encryption efficiently, it is considered infeasible to try, as it would take a large amount of computing power and a long time to manage it. RSA's strength is due to two mathematical problems, as described below.

4.4.1 The RSA Problem

This is the task of trying to compute a value for m , where $c = m^e \bmod n$, where (e, n) is the RSA Public Key, and c is the ciphertext. There is currently no known way to compute this that would be efficient, but if a way is found it would give a method for breaking all RSA encryption.

4.4.2 The Problem of Factoring Large Numbers

The best way to attempt to break an RSA encryption is to factorise the numbers in the public key to find the two prime numbers originally used when it was generated. The numbers used in the private key can then be worked out much more quickly and efficiently. This method is thought infeasible if the keys are sufficiently long. In 2005 the largest number successfully factorised was a 663-bit number. Modern RSA keys are usually 1024 to 2048 bits long, which is of sufficient length to make them unbreakable at the moment. It is still thought that 1024 bit numbers will become breakable within a short time. Therefore it is currently recommended that at least 2048 bit RSA keys be used.

Peter Shor suggested that using a Quantum Computer could greatly improve the efficiency of factoring large numbers. If such a computer was developed it would mean the RSA algorithm would become unusable because it would be easy to break. A quantum computer with this power is not likely to be developed for many years.

4.5 Algorithms Used

4.5.1 Public/Private key generation

The generation of public keys is defined using the following steps:

- Generate two random prime numbers p and q
- Calculate n the product of p and q
- Calculate the totient of n , $\varphi(n)$
Totient: The totient of a number n is defined to be the number of prime integers less than or equal to n and coprime to n . In our case this can easily be calculated as $\varphi(n) = (p-1)(q-1)$
- Find an integer e which is coprime to $\varphi(n)$ and $3 < e < n-1$
Coprime: a and b are coprime if they have no common factor other than 1
- Calculate d to satisfy the congruence relation $de=1(\text{mod } \varphi(n))$.
Congruence Relation: this can be defined where a and b have the same remainder when divided by n using the following expression $a \equiv b (\text{mod } n)$. d can also be calculated as the multiplicative inverse of $e \text{ mod } r$

Once these numbers have been calculated, n is used as the modulus for both public and private keys, e is the public key exponent and d is the private key exponent.

4.5.2 Encryption/Decryption

The encryption and decryption algorithms are defined as follows:

To encrypt messages the formula $c = m^e \text{ mod } n$ is used. Where the output from the algorithm; c is the ciphertext. m is the original message, given by the user. e is the public key exponent and n is the modulus both taken from the public key file.

To decrypt messages the algorithm $m=c^d \text{ mod } n$ is used. Where the output from the algorithm; m is the original message. c is the ciphertext, given by the user. d is the private key exponent and n is the modulus both taken from the private key file.

5 Work Plan

This project is structured with several implementations, each an extension of the previous one and built off the previous implementations code. An incremental approach would be the best way to tackle this project.

The first implementation will cover coding the algorithms using small numbers. The second implementation is extending the program to handle large numbers. The final implementation is adding functionality to allow text file encryption and decryption.

A Gantt chart is included in *Appendix A* to show my expected timetable for the implementations.

6 Implementation 1 – Small Numbers

The main aim of this section of the project was to ensure that I understood the algorithms needed for RSA cryptography.

6.1 Research and Design

The first part of the project required me to get a reasonable understanding of the algorithms used in RSA cryptography. To do this I looked at a number of resources; the main ones used are listed here. For a full list see the Bibliography.

RSA Specification: <http://www.rsa.com>

As part of the requirement of the project was to meet the RSA specification, I used it to find out exactly what RSA was, started to understand what algorithms were needed for the project and the way the algorithms worked. The terminology in the specification is fairly technical so to get an initial understanding of the algorithms was quite difficult.

Wikipedia: <http://www.wikipedia.org>

This website gives a complete explanation and a simple example run through of the algorithms used for key generation, encryption and decryption. It was a very useful resource to gain an understanding of the basics of the algorithms required. The RSA page also contained links to the relevant pages for the various mathematical concepts required.

Cryptography and Network Security Principles, by William Stallings

This book again had good explanations and a run through of the algorithms used in RSA. It also explained the mathematical principles behind the algorithm and examples of how they could be calculated.

Once I had a good understanding of the algorithms needed I then implemented them using small numbers (ones which are small enough to fit into the standard data types in C). As only small numbers are used, this implementation did not need to be secure and as it is only being used to gain an understanding of the algorithms it was not a requirement to be efficient.

After looking at the way the algorithms worked, I decided it would be easier to first implement the key generation in one C file and the encryption/decryption in another. In the key generation file I decided to implement each section of the algorithm in a separate function and write the numbers for the keys to two files. I then created a menu system to make it more user friendly and link it all together.

6.2 Implementation of Code

The final version of the small number implementation consisted of the following files:

Source Files:

- ***cipher.c*** – this file contained the functions to encrypt and decrypt numbers using the two key files.
- ***gen_keys.c*** – this file contained the functions used to generate the numbers for the keys and write the two key files.
- ***menu.c*** – this contained the main program which the user ran, it links the other files together and includes the function to write the menu system.

Text Files:

- ***private_key.txt*** – a text file containing the two numbers which make up the private key.
- ***public_key.txt*** – a text file containing the two numbers which make up the public key.

Other Files:

- ***menu*** – the main program file, this is a compiled version of ***menu.c***.

Code listing of key functions can be found in *Appendix C*.

6.2.1 Menu – menu.c

A simple menu system was created to give access to all the functions of the program from one central point. This makes the program easier for the user as there is only one compiled program file. The ***main()*** function is within ***menu.c*** and displays the program menu for the user and processed their choice by running the relevant function. ***menu.c*** also contains a function to display the key files; ***public_key.txt*** and ***private_key.txt*** to the user.

6.2.2 Key Generation – gen_keys.c

The following sections were implemented as functions in ***gen_keys.c***. The ***gen_keys()*** function uses each of the sections in turn to calculate the relevant values needed for the two keys. Once the values for the keys have been generated they are written out to two files, ***private_key.txt*** and ***public_key.txt***.

6.2.2.1 Generating *p* and *q*

p and ***q*** are two prime numbers which are chosen randomly and are not equal. They were generated by choosing a random number ***x***, then checking if ***x*** was prime. A prime number is defined as a number that only divides by 1 and itself without a remainder.

A way of checking if ***x*** is prime would be to see if it divides equally by any number less than itself. This is a very inefficient method, especially if the numbers are large. To greatly decrease the time it takes to check if a number is prime, it only needs to be divided by numbers up to \sqrt{x} . Another way to reduce the time it takes is to check the remainder when ***x*** is divided by 2 and then check only the odd numbers, because if ***x*** divides by any even number it will also divide by 2. If ***x*** will divide exactly by 2 or any odd number less than \sqrt{x} it is not a prime number. This is still a relatively inefficient method but, as only small numbers were involved at this stage,

it is possible to generate the numbers this way. It would not be practical if larger numbers were being used.

6.2.2.2 Calculating n and r

n is the product of p and q so $n = p * q$

r is the totient of n . This is defined as the number of positive integers which are less than or equal to and coprime to n . In this case as n is the product of the two prime numbers p and q , the totient r is the result of $(p-1) * (q-1)$.

6.2.2.3 Calculating e

e is a random number which is coprime to r . Two numbers are defined to be coprime if they have no common factors other than 1, therefore their greatest common denominator is 1.

To calculate e , a random number is generated and then the greatest common denominator of the random number and r is calculated. This process is repeated until the random number generated and r have a greatest common denominator of 1. An iterative implementation of the Euclidean Algorithm was used to calculate the greatest common denominators.

6.2.2.4 Calculating d

As e and r are coprime, d can be calculated as the multiplicative inverse of $e \bmod r$. The Extended Euclidean Algorithm can be used to calculate d given e and r . An iterative implementation of the algorithm was implemented.

6.3 Encryption/Decryption – *cipher.c*

The encryption and decryption functions are provided by *cipher.c*. This file contains the two functions *encrypt()* and *decrypt()* which get called from *menu.c*. These functions work in a similar way, they read the numbers stored in the relevant key file and get an input from the user of either the plaintext or ciphertext. These values are then passed to the *cipher()* function which performs encryption or decryption.

6.3.1 *cipher()*

The encryption and decryption functions use different parameters but are the same, because of this a single *cipher()* function can be implemented. *cipher()* accepts the values passed from *encrypt()* or *decrypt()*, it then calculates $m^e \bmod n$ where m is the number being encrypted or decrypted, e is the exponent and n is the modulus. *cipher()* uses a square and multiply algorithm to perform this calculation.

6.4 Testing

There are a number of different things which need testing in the system, and a number of ways of testing them. The testing methods I chose are listed below:

6.4.1 Testing 1 – Key Generation:

I will start my testing by checking if the keys are being generated correctly in relation to the algorithms given in the RSA specification.

6.4.1.1 Checking p and q

To check the program is generating prime numbers correctly. I will use the program to generate 10 pairs of prime numbers p and q . I will then check if they are prime

against a list of prime numbers less than 256 shown in *Table 1*. I can also check that p and q are different.

Table 1 - List of primes less than 256:

2	3	5	7	11	13	17	19	23
29	31	37	41	43	47	53	59	61
67	71	73	79	83	89	97	101	103
107	109	113	127	131	137	139	149	151
157	163	167	173	179	181	191	193	197
199	211	223	227	229	233	239	241	251

Results:

Table 2 - Results for Checking p and q:

p	Prime?	q	Prime?
37	Yes	137	Yes
181	Yes	211	Yes
61	Yes	47	Yes
151	Yes	17	Yes
197	Yes	31	Yes
107	Yes	193	Yes
67	Yes	41	Yes
157	Yes	13	Yes
73	Yes	23	Yes
199	Yes	7	Yes

As you can see from *Table 2*, all the numbers generated for p and q are prime and p is never equal to q . This shows that p and q have been generated correctly and can be used to create RSA keys

6.4.1.2 Checking n and r

To check the program is generating n and r correctly I again will let the program generate the numbers and compare them to ones generated by hand. n should be calculated as $p * q$ and r as $(p - 1) * (q - 1)$.

Results:

Table 3 - Results for Checking n and r:

p	q	n	p * q	r	(p - 1) * (q - 1)	Correct?
37	137	5069	5069	4896	4896	Yes
181	211	38191	38191	37800	37800	Yes
61	47	2867	2867	2760	2760	Yes
151	17	2567	2567	2400	2400	Yes
197	31	6107	6107	5880	5880	Yes
107	193	20651	20651	20352	20352	Yes
67	41	2747	2747	2640	2640	Yes
157	13	2041	2041	1872	1872	Yes
73	23	1679	1679	1584	1584	Yes
199	7	1393	1393	1188	1188	Yes

As you can see from *Table 3*, all the numbers generated for n and r have been calculated correctly and can be used to create RSA keys.

6.4.1.3 Checking e and d

To check the program is generating e and d correctly I will let the program generate the numbers again and compare them to ones generated by hand. The greatest common denominator of e and r must be 1 and d must be the multiplicative inverse of $e \bmod r$. I will calculate these using the Extended Euclidean Algorithm.

Results:

Table 4 – Results for Checking e and d :

r	e	d	$\text{gcd}(e, r)$	inverse of $e \bmod r$	Correct?
4896	203	2243	1	2243	Yes
37800	101	24701	1	24701	Yes
2760	187	1963	1	1963	Yes
2400	17	1553	1	1553	Yes
5880	157	1573	1	1573	Yes
20352	167	4631	1	4631	Yes
2640	167	743	1	743	Yes
1872	77	389	1	389	Yes
1584	23	551	1	551	Yes
1188	103	1015	1	1015	Yes

As you can see from Table 4, all of the greatest common denominators of e and r are equal to 1; therefore e must have been worked out correctly. The numbers calculated for d match the number when worked out by hand; this also shows that they are correct.

6.4.1.4 Test Conclusion

All the tests have been successful; the numbers calculated by the computer match the ones worked out by hand. Therefore I can conclude that the algorithms I have implemented for the public and private keys generation are correct and the keys generated can be used to encrypt and decrypt numbers.

6.4.2 Testing 2 – Encryption/Decryption:

Now the testing on the keys generated has been successful I can go on to test the encryption and decryption algorithms.

I will use the program to generate a key pair. I can then use the key to work through by hand and see what a particular number encrypted should be. The encrypted number can then be decrypted again by hand and checked against the original number. The same number can then be encrypted and decrypted using the computer and the results compared to the ones worked out by hand. As I will be working the results out by hand I will decrease the key size, so the maximum values for p and q is 20, to make it manageable. I will test five different key pairs and encrypt/decrypt five different numbers with each.

Results:

Table 5 – Encryption/Decryption Test Results

Public Key	Private Key	Plaintext	Computer		By Hand		Correct?
			Encrypted	Decrypted	Encrypted	Decrypted	
203 2519	1067 11 229	123	1460	123	1460	123	Yes
		321	393	321	393	321	Yes
		456	1093	456	1093	456	Yes
		789	1040	654	1040	654	Yes
		1234	2043	1234	2043	1234	Yes
85 32111	17557 163 197	123	10673	123	10673	123	Yes
		321	27194	321	27194	321	Yes
		456	9521	456	9521	456	Yes
		789	1380	654	1380	654	Yes
		1234	10695	1234	10695	1234	Yes
151 8003	3151 151 53	123	7371	123	7371	123	Yes
		321	5153	321	5153	321	Yes
		456	7100	456	7100	456	Yes
		789	3809	654	3809	654	Yes
		1234	630	1234	630	1234	Yes
207 5497	2403 239 23	123	1274	123	1274	123	Yes
		321	3058	321	3058	321	Yes
		456	585	456	585	456	Yes
		789	3393	654	3393	654	Yes
		1234	543	1234	543	1234	Yes
247 3103	2223 107 29	123	223	123	223	123	Yes
		321	1605	321	1605	321	Yes
		456	2103	456	2103	456	Yes
		789	747	654	747	654	Yes
		1234	140	1234	140	1234	Yes

As Table 5 shows the numbers calculated by hand match the numbers calculated by the computer therefore the computer is encrypting and decrypting the values correctly.

6.4.2.1 Test Conclusion

The decrypted numbers are the same as the original plaintext proving the tests have all been successful. I can conclude that the encryption and decryption parts of the program work correctly and generate the required results.

6.5 Evaluation

The main aim of this section of the project was to understand the algorithms involved in RSA cryptography. I feel that now I have a fairly good understanding of how the three algorithms; key generation, encryption and decryption, work and I now have the knowledge to extend the code to create a secure larger number implementation of the RSA algorithms.

A secondary aim for this part of the project was to re-familiarise myself with the C programming language, I feel this has been achieved and I am confident in continuing with the programming for the rest of the project.

7 Implementation 2 – Large Numbers

Once the small number implementation had been finished and I understood the algorithms required for RSA encryption, I went on to implement them using large numbers. This would mean I could generate at least 2048 bit keys which would be secure.

7.1 Library Choice

The standard C library can only handle numbers up to 4,294,967,295 in an unsigned long integer data type. To implement the algorithms using large numbers I needed an additional C library which could handle such big numbers, after looking on the Internet I found two which would possibly be what I was looking for.

7.1.1 BigDigits Library

The BigDigits library is a C library that contains a number of multiple precision routines; these would allow me to implement the required algorithms using the large numbers needed.

After looking at the website for this library, I could not find much documentation about the different functions the library contained, so could not tell if it would suit all my needs or not. After trying to implement some of the simple parts of the RSA algorithm using this library I decided it was not very user friendly and over complex for my needs. Another problem with this library, according to the BigDigits website, is that it is not the most efficient and not cryptographically secure. This should not be too much of a problem for this project, but if I was wanting to implement a commercial RSA program then this would be a major disadvantage.

The library and further information can be found at:
<http://www.di-mgt.com.au/bigdigits.html>

7.1.2 GMP Library

The GMP library is the GNU Multiple Precision library which contains a number of functions for large number arithmetic. The only limit to the size of the numbers used in this library is the size of memory on the machine the code is running on.

This library had better documentation than the BigDigits Library and therefore it was easier to install the library. I could also tell from the documentation that it had all the functions I needed for this project and it was very clear how they worked and how to implement them.

The library and further information can be found at <http://gmplib.org/>

7.2 Function Definitions

A list of functions used (definitions taken from the GNU MP documentation):

Initialisation Functions:

- ***mpz_t integer***; – Define an integer object called *integer*.
- ***mpz_init (mpz_t integer)***; – Initialise *integer* and set the value to 0.

Arithmetic Functions:

- ***mpz_mul(mpz_t answer, mpz_t op1, mpz_t op2)***; – Set *answer* to $op1 * op2$.

- ***mpz_sub_ui(mpz_t answer, mpz_t op1, unsigned long int op2);*** – Set *answer* to $op1 - op2$.

Exponentiation Functions:

- ***mpz_powm(mpz_t answer, mpz_t base, mpz_t exp, mpz_t mod);*** – Set *answer* to $base^{exp} \bmod mod$.

Number Theoretic Functions:

- ***mpz_probab_prime_p(mpz_t n, int reps);*** – Check if n is prime, returns 2 if n is definitely prime, 1 if n is probably prime and 0 if n is defiantly not prime. The function uses the Miller–Rabin probabilistic primality tests. *reps* is the number of tests to carry out, a higher number reduces a non-prime being returned as probably prime.
- ***mpz_gcd(mpz_t answer, mpz_t op1, mpz_t op2);*** – Set *answer* to the greatest common denominator of $op1$ and $op2$.
- ***mpz_gcdext(mpz_t g, mpz_t s, mpz_t t, mpz_t a, mpz_t b);*** – Set g to the greatest common denominator of a and b . Also find s and t to satisfy the equation $as + bt = g$.

Comparison Functions:

- ***mpz_cmp(mpz_t op1, mpz_t op2);*** and ***mpz_cmp_si (mpz_t op1, unsigned long int op2);*** – Compare $op1$ and $op2$, return a positive value if $op1 > op2$, 0 if $op1 = op2$ and a negative value if $op1 < op2$.

Random Number Functions:

- ***mpz_urandomb(mpz_t answer, gmp_rand_state_t state, unsigned long int n);*** – Generate a random number between 0 and $2^n - 1$ and store in *answer*. *state* is a variable which is initialised using the Random State Initialisation Functions below.

Random State Initialisation

- ***gmp_randstate_t state;*** – Define *state* as a state variable.
- ***gmp_randseed_ui(gmp_rand_state_t state, unsigned long int seed);*** – Set an initial seed value into *state*.
- ***gmp_randinit_default(state);*** – Initialise *state* with the default algorithm (default set to the Mersenne Twister algorithm).

Input/Output Functions:

- ***gmp_printf(const char *fmt,...);*** – Print to standard output.
- ***gmp_fprintf(FILE *fp, const char *fmt,...);*** – Print to the file *fp*.
- ***gmp_scanf(const char *fmt,...);*** – Read from standard input.
- ***gmp_fscanf(FILE *fp, const char *fmt,...);*** – Read from file *fp*.

7.3 Code

Using the small number implementation code as a template I rewrote the code using the functions from the GMP library.

The final version of the large number implementation consisted of the following files:

Source Files:

- ***cipher_big.c*** – this file contained the functions to encrypt and decrypt numbers using the two key files.

- **gen_keys_big.c** – this file contained the functions used to generate the numbers for the keys and write the two key files.
- **menu_big.c** – this file contained the main program, it links the other files together and includes the function to write the menu system.

Text Files:

- **private_key.txt** – a text file containing the two numbers which make up the private key.
- **public_key.txt** – a text file containing the two numbers which make up the public key.

Other Files:

- **menu_big** – the main program file, this is a compiled version of **menu_big.c**.

Code listing of key functions can be found in *Appendix C*.

7.3.1 Menu – menu_big.c

The **menu_big.c** file was similar to **menu.c**, it contained the **main()** which displayed the menu and accepted the user input in the same way. The **open_key_big()** function displayed the two keys in the same way as **open_key()** but used the GMP functions **gmp_scanf()** and **gmp_printf()** instead of standard C ones.

7.3.2 Key Generation – gen_keys_big.c

The values for the key files were worked out as described below then written to the two key files, **public_key.txt** and **private_key.txt** using **gmp_fprintf()**.

7.3.2.1 Generate p and q

To generate the two prime numbers, a random number was generated using the **mpz_urandomb()** function then tested if it was a prime number using **mpz_probab_prime()**. A second prime number was then generated in the same way and checked that it was different from the first.

7.3.2.2 Calculating n and r

As n is calculated by multiplying p and q the **mpz_mul()** function was used to multiply the two numbers. r is $(p-1) * (q-1)$ so the **mpz_sub_ui()** was used to subtract 1 from p and q then the **mpz_mul()** function used to multiply the answers.

7.3.2.3 Calculating e

A random number was generated again using **mpz_urandomb()**. The greatest common denominator of this number and r was then calculated using **mpz_gcd()**. This process was repeated until the generated number and r had a greatest common denominator of 1. This was checked using the **mpz_cmp_si()** function.

7.3.2.4 Calculating d

The **gmp_gcdext()** function with the parameters e and r was used to run the Extended Euclidean Algorithm and calculate d .

7.4 Encryption/Decryption – cipher_big.c

cipher_big.c contains the two functions **encrypt_big()** and **decrypt_big()** which get called from **menu_big.c**, These functions read the numbers stored in the relevant key file and get an input from the user of either the plaintext or ciphertext, this is

done using the `gmp_fscanf()`, `gmp_printf()` and `gmp_scanf()` functions. The encryption/decryption algorithm is then used on these values using the `mpz_powm()` function.

7.5 Testing

7.5.1 Key Generation, Encryption and Decryption

As the keys generated in this implementation are so large, it is not practical to work through any of the algorithms by hand. Instead I will test this implementation by generating different key pairs and using them to encrypt and decrypt different numbers. If the decrypted number matches the original one then the key generation, encryption and decryption must have been successful.

I will generate five 1024 bit key pairs and encrypt and decrypt five different numbers with each. *Table 6* shows the plaintexts used for encrypting, *Table 7* shows the keys which were used.

Table 6 – Plaintexts Used for Testing

	Plaintext
1	1234567890
2	9876543210
3	123
4	1234567899876543210123456789987654321012345678998765432101234567899876543210
5	1234567899876543210123456789987654321012345678998765432101234567899876543210 1234567899876543210123456789987654321012345678998765432101234567899876543210 1234567899876543210123456789987654321012345678998765432101234567899876543210

Table 7 – Keys used for Testing

	Public Key	Private Key
1	28447387842614708445606379578304754089234318252024 26791204160290508170802835346861055309779518175147 70324417321046482153454550943120993625453467061515 2225 85901542317544644977308053961195364004178369462584 44442648868811583829119510145882314898720789741313 96415187296324921555069800346572023403007985849378 45826121481710366064094446668254021701379560054293 94269292865546238407977425923223978638154163410671 91570653962945303793329666405611911053889123979370 5651803	23537313402530884266144522458567925332151479335704 37976286905449956389097188068572110623015883338833 60298425697563646812055104707806797926945081095361 10852403459399673543437755533831442104346880760146 10938687028737215803285943291609795193776649838467 82551193469999256516942643410423146442120511282082 902105 85901542317544644977308053961195364004178369462584 44442648868811583829119510145882314898720789741313 96415187296324921555069800346572023403007985849378 45826121481710366064094446668254021701379560054293 94269292865546238407977425923223978638154163410671 91570653962945303793329666405611911053889123979370 5651803
2	10053986556483130130981513837905546434567082711757 42375437076683369478396973899782470680105158257840 90243922989882667158136521348666236616046703498808 29197 15657309273019004644211649654572841478703429931826 98885646071477350406607191888714226412172097146464 72160698422297811205702994662511209938364872800273 8594789257980615458704270343926459201017132687533 09822804407189542540764638527665779197303148312447 48482065996089440586456885589893858347275808369727 1642119	41120311045606940000831480404467257502988836888498 68546892860624372905939649688594754978772651760498 77425243965941666849830774725059922184899097668869 38404654662798704941952443611564736544073466753211 05291166514151116768612868400743849027846822047337 75423294194534014104888556679651968421094427082844 521173 15657309273019004644211649654572841478703429931826 98885646071477350406607191888714226412172097146464 72160698422297811205702994662511209938364872800273 8594789257980615458704270343926459201017132687533 09822804407189542540764638527665779197303148312447 48482065996089440586456885589893858347275808369727 1642119
3	11668524429044200325863284858030030219761542041067 87120830663691604248357593139749615809830435895707 24456176557261136324596865889585811684487340957520 85293 35900056174147545271024684110387512586031447607989 93872937737233483481860299483731743079805309543257 38934397641845593765184269718172020055447973645562 49077936750752814213320741119646660484544171411530 77997999306131746416256263029761657466557339755967 05953235309145860415636063229897000143166837399827 79414307	14914501920908439621136050170287911824299205053594 11675631772264766671296875008093997469998629600004 35436508420042127355766748233312714965359305675210 19689230023802288379008739048782551847185880746707 83853113309388496266405542724508549842849343690352 65505564155855434854814874144047356919559319254631 9296837 35900056174147545271024684110387512586031447607989 93872937737233483481860299483731743079805309543257 38934397641845593765184269718172020055447973645562 49077936750752814213320741119646660484544171411530 77997999306131746416256263029761657466557339755967 05953235309145860415636063229897000143166837399827 79414307
4	12561714138367112533189317848058884130015998267994 61503221499470787857833407182147955470851889656211 18180708577801324991722826643458255888014797444604 88153 34599756926766898009276185620281435552993260720077 72486524408771348749584082410133747830973643196799 81252927137123288925355009380025794609700398533208 54708947858771733618286318768629768997328106743074 24797702513839217680393374240763507048445783758569 49909609003328362450746943061118808658001942404630 8256529	12420752323298297434562394041503903125630611931075 37374170144833409584007301055626239365487614093294 00383000715050130058704221375258630851143305076386 59154753309198049929732561528748353164842322204258 77221374860561508038406969067978177172117369991758 20919695645419180358628712127770673299310589462352 3489317 34599756926766898009276185620281435552993260720077 72486524408771348749584082410133747830973643196799 81252927137123288925355009380025794609700398533208 54708947858771733618286318768629768997328106743074 24797702513839217680393374240763507048445783758569 49909609003328362450746943061118808658001942404630 8256529
5	54754894544918474279595884533579352258302394615475 61518396565832529878228529852356382926853257531297 98713888322044054393403308381340903327302711173875 195 80126620555891326169488798763381718756777166883112 69489333784396088972251431012577947713096630904654 89356450888104689282212325218374945964236884447752 12211055817474731367928102031878849474918996723524 41953059466385607453613638189207762094719633351127 96946038221375953872345090364380199521993943869827 6594541	17398251189467504150693309628978590859901413617956 44774204870313047573594950040139191774236497884357 76807421474420597995624275332817044186774362785619 40570389726041463068942116512367222362596243330231 27969139085740096812502039433914805386057960634592 34905537062000161170929238811846505486474532779729 9684371 80126620555891326169488798763381718756777166883112 69489333784396088972251431012577947713096630904654 89356450888104689282212325218374945964236884447752 12211055817474731367928102031878849474918996723524 41953059466385607453613638189207762094719633351127 96946038221375953872345090364380199521993943869827 6594541

7.5.1.1 Results

Table 8 – Large Number Implementation Test Results

Key Pair	Plaintext	Result
1	1	Correct
1	2	Correct
1	3	Correct
1	4	Correct
1	5	Correct
2	1	Correct
2	2	Correct
2	3	Correct
2	4	Correct
2	5	Correct
3	1	Correct
3	2	Correct
3	3	Correct
3	4	Correct
3	5	Correct
4	1	Correct
4	2	Correct
4	3	Correct
4	4	Correct
4	5	Correct
5	1	Correct
5	2	Correct
5	3	Correct
5	4	Correct
5	5	Correct

Table 8 shows a summary of the results of the test, full results can be found in Appendix B. The results show all the tests have succeeded.

7.5.1.2 Test Conclusion

The encrypted numbers are the same as the original therefore the tests are shown to be successful. I can conclude that the three parts of the program, key generation, encryption and decryption work correctly and generate the required results.

7.5.2 Key Length

It is important that the key length being generated is the length expected, therefore I will also check the program can generate keys of 2048 bits. I will generate five 2048 bit key pairs and check the lengths of the keys.

7.5.2.1 Results

Table 9- Key Bit Length Test Results

Key Pair	Bit Length
1	2048
2	2046
3	2047
4	2045
5	2048

The key pairs generated can be found in Appendix B. Table 9 shows the bit lengths of each of these keys.

7.5.2.2 Test Conclusion

Two out of the five test generated keys which were 2048, the others were less than this. This would be due to the fact that any leading zeros in the numbers generated in the keys would be ignored as the numbers are treated as integers not binary strings.

7.6 Evaluation

The main aim for this section of the project was to generate 2048 bit keys and then use them to encrypt and decrypt large numbers. The program I have implemented can generate keys of the desired length and encrypt numbers that are smaller than n for a particular key. During this implementation I have gained an understanding of some of the functions contained in the Gnu Multiple Precision library and have used them to help with the implementation.

8 Implementation 3 – File Encryption

Once the large number implementation had been finished and I had an understanding of the library functions from the GMP Library, I could extend the implementation to include file encryption and decryption for simple text files.

8.1 Function Definitions

In addition to the functions listed previously in the Large Number section, the following were also used:

Initialisation Functions:

- `size_t size;` – Define a size object called `size`.

Assignment Functions:

- `mpz_set_ui(mpz_t answer, unsigned long int op);` – Set `answer` to `op`.

Conversion Functions:

- `unsigned long int mpz_get_ui(mpz_t op);` – Return the value of `op` as an `unsigned integer`.

Arithmetic Functions:

- `mpz_add(mpz_t answer, mpz_t op1, mpz_t op2);` and `mpz_add_ui(mpz_t answer, mpz_t op1, unsigned long integer op2);` – Set `answer` to `op1 + op2`.
- `mpz_sub(mpz_t answer, mpz_t op1, mpz_t op2);` – Set `answer` to `op1 - op2`.
- `mpz_mul_ui(mpz_t answer, mpz_t op1, unsigned long int op2);` – Set `answer` to `op1 * op2`

Division Functions:

- `mpz_cdiv_q_ui(mpz_t q, mpz_t n, unsigned long int d);` – Set `q` to the quotient of `n / r`.
- `mpz_cdiv_r_ui(mpz_t r, mpz_t n, unsigned long int d);` – Set `r` to the remainder of `n / r`.

Miscellaneous Functions:

- `size_t mpz_sizeinbase(mpz_t op, int base);` – Return the size of `op` measured in the number of digits in the given `base`.

8.2 Code

Using the large number implementation code as a template I rewrote the code so that I could encrypt and decrypt files. The menu and key generation sections of the code did not need changing at all, only the encryption and decryption sections.

The final version of the file encryption implementation consisted of the following files:

Source Files:

- ***cipher_file.c*** – this file contained the functions to encrypt and decrypt files using the two key files.
- ***gen_keys_file.c*** – this file was the same as ***gen_keys_big.c*** it contained the functions used to generate the numbers for the keys and write the two key files.
- ***menu_file.c*** – this file was the same as ***menu_big.c*** it is a c source file containing the source code for the main program, it links the other files together and includes the function to write the menu system.

Text Files:

- ***private_key.txt*** – a text file containing the two numbers which make up the private key.
- ***public_key.txt*** – a text file containing the two numbers which make up the public key.
- ***input.txt*** – a text file containing the input text to be encrypted.
- ***output.txt*** – the encrypted version of ***input.txt***.
- ***decrypted.txt*** – the decrypted version of ***output.txt***.

Other Files:

- ***menu_file*** – the main program file, this is a compiled version of ***menu_file.c***.

Code listing of key functions can be found in *Appendix C*.

8.2.1 Menu – *menu_file.c*

The ***menu_file.c*** file was the same as ***menu_big.c***, the only difference was the name of the functions included to ensure the correct ones get called.

8.2.2 Key Generation – *gen_keys_file.c*

The ***gen_keys_file.c*** file was the same as ***gen_keys_big.c***, again the only difference was the name of the functions included.

8.3 Encryption/Decryption – *cipher_file.c*

cipher_file.c contains the two functions ***encrypt_file()*** which encrypts a text file and ***decrypt_file()*** which decrypts the output from ***encrypt_file()***. Both these functions get called from ***menu_big.c***.

8.3.1 *encrypt_file()*

This function reads the two key values from ***public_key.txt***, calculates the length of the key in bytes. The number of letters in ***input.txt*** is calculated, encrypted and stored in ***output.txt***. Then blocks of letters are read from ***input.txt*** (the number of bytes in the key determine the number of letters in a block, one letter for each byte in the key). The ASCII value of these letters is then combined into one value which is encrypted and written to ***output.txt***. This process is repeated with all the blocks of

that length in *input.txt*. The last block is padded with 0's if needed to make a whole block.

8.3.2 decrypt_file()

This function reads the two key values from *private_key.txt*, calculates the length of the key in bytes. The number of letters in the original plaintext is calculated by decrypting the first number from *output.txt*. Each number in *output.txt* is then taken in turn, decrypted and split back into blocks of characters the same size as the key. These blocks are then written out to *decrypted.txt*. The final block has the 0's used as padding removed before being written.

8.4 Testing

8.4.1 Key Generation

As the files for key generation are the same as the ones implemented and tested in the large numbers section there is no need to test them again for this section.

8.4.2 Encryption and Decryption

There are two types of test I will run to check the encryption and decryption algorithms. The first test will be to check the program can encrypt and decrypt all the alphanumeric characters and symbols present on the keyboard. The second test will be to ensure the program can handle files of different lengths. To run the test I will use the program to encrypt and decrypt the different files, I will then use the Linux *diff* command on the input file and output file to check that they are the same.

8.4.2.1 Character Test

I will create a file containing all the alphanumeric characters and symbols on a standard keyboard. I can then use this file as the input for my program, encrypt it and decrypt the ciphertext produced. I will check the decrypted version against the original version to ensure they are the same.

I will use the following characters repeated a couple of times in a file:

```
"1234567890-="!"~£$%^&*()_+~`qwertyuiop[]  
asdfghjkl;'#  
|zxcvbnm,./QWERTYUIOP{}  
  
ASDFGHJKL:@~  
|ZXCVBNM<>?"
```

This includes all the numbers, letters and symbols as well as return and tab.

Results

The input file and decrypted file were identical, therefore the program encrypted and decrypted the file successfully.

8.4.2.2 File Size Test

To ensure the program can handle files of different sizes I will create various different sized files, encrypt them and decrypt the ciphertext. The files will be created with a certain number of letters ranging from 1,000 to 1,000,000.

Results

Table 10 – File Size Test Results

Number of Letters	Result
1,000	Pass
5,000	Pass
10,000	Pass
50,000	Pass
100,000	Pass
500,000	Pass
1,000,000	Pass

Table 10 shows all the tests have been successful; the input file matched the decrypted file.

8.4.2.3 Test Conclusion

As all the tests have been successful, I can conclude that my implementation of the decryption and encryption algorithms works. It can encrypt and decrypt simple text files containing up to 1 million standard alphanumeric character or symbols,

8.5 Evaluation

The main aim of this section was to create a program to encrypt and decrypt simple text files. The program I have implemented does this and can handle large text files made from any standard alphanumeric or symbolic characters present on the keyboard.

The way this program reads the input text file, as a block of ASCII characters rather than a block of binary data, results in the size of the encrypted file being roughly 2.4 times larger than the plaintext file. This size difference between the plaintext and ciphertext files increases if very small plaintext files are used.

If more time were available, giving the user the ability to specify the key length from the user interface would be favourable to having it compiled directly into the code. It would also be useful if the user could change the names of the input and output files used in the program.

9 Performance Analysis

The performance and efficiency of the code will be measured by calculating the time it takes to generate keys and use them to encrypt and decrypt files of different sizes. The way I will measure the time will be by using the Linux *time* command, because of this I will use a slightly modified version of the code that does not make use of the menu system and requires no user input once running. These tests will be run five times and the average taken.

9.1 Key Generation

I will generate ten 2048 bit key pairs and take the average time taken to generate them.

9.1.1 Results

Table 11 – Key Generation Times

Time (S)	
2.499	
3.233	
2.258	
2.211	
7.962	
3.789	
2.698	
2.117	
3.319	
4.552	
Average	2.474

9.2 Encryption/Decryption

I will encrypt and decrypt three different sized files, five times each and measure the average time taken for each file size. The file sizes I will use are 1,000, 500,000 and 1,000,000 characters.

9.2.1 Results

Table 12 – Encryption/Decryption Times

No. Characters	Time (S)	
	Encryption	Decryption
1,000	0.184	0.356
	0.183	0.354
	0.179	0.346
	0.177	0.343
	0.177	0.341
Average	0.180	0.348
500,000	65.849	129.542
	65.822	129.421
	65.762	129.113
	65.902	129.110
	65.796	129.006
Average	65.826	129.238
1,000,000	131.031	258.516
	131.268	258.472
	130.899	258.487
	131.075	258.781
	131.184	158.632
Average	131.091	238.578

9.3 Evaluation

Table 11 shows the times taken to generate 2048 bit keys, the times vary between just over 2 seconds to just less than 8 seconds. This is because when generating key pairs the program has to generate two prime numbers. It does this by

generating a random number and testing if it is prime, the first random number may be prime, or it might have to generate quite a few numbers before finding a prime number.

Table 11 shows the time taken to encrypt and decrypt various sized files. The 1000 character file encrypted and decrypted very quickly, this would be good for users wanting to encrypt a short email for example, as they would not have to wait long for it to encrypt. The decryption time is roughly twice the encryption time. This is because more arithmetic has to be done to decrypt the numbers, than encrypt them. When the number of characters doubles from 500,000 to 1,000,000 the time also doubles, showing the time is proportional to the number of letters. This relationship can be used to estimate the time it would take to encrypt or decrypt files of different sizes. It takes about 2 minutes to encrypt a 1,000,000 character file and about 4 minutes to decrypt it. This is quite a long time but it is still reasonable as a 1,000,000 character file is very large and is unlikely to have to be encrypted very often.

10 Evaluation & Conclusion

10.1 Project Achievements

I feel the project has met all the original aims and objectives set out at the beginning. The three sections implemented have been successful; this can be seen by looking at the test results for each section.

My implementation is able to generate a key pair and encrypt a 1000 character text file in just a few seconds which is reasonable. One of the main uses of RSA is to encrypt a key for a symmetric algorithm. A 4096 bit symmetric key would generate a 512 character file so would not take long to encrypt.

10.2 Problems Encountered

I had wanted to evaluate my implementations against existing ones, but was unable to find any programs on the Internet that would allow me to do this. The programs I did find were mainly windows based and had a complex graphical user interface. As my implementation was Linux based it meant the comparisons of results would be unreliable as the computers they would be run on would have different hardware specifications. The graphical user interface would also add additional overhead to the program and not being able to run the program on a command line would mean there is no easy way to accurately time the programs.

To produce a faster implementation, I would want to try to get the program to read the plaintext file as blocks of binary data rather than blocks of ASCII letters. This would also help to decrease the size of the ciphertext files.

10.3 Personal achievements

After doing this project I feel I have a greater understanding of cryptography in general as well as the RSA algorithm. I have also gained time management and organisational skills.

By implementing the small number section of the project without using library functions, I gained knowledge of the RSA algorithms and the mathematics behind them. It also allowed me to re-familiarise myself with the C programming language. The large number implementation allowed me get a greater understanding of how C

libraries work and how they can be used to add additional functions to the standard ones included in C. The final file encryption section allowed me to further my knowledge in both C and the GMP Library.

10.4 Final Conclusion

Overall this project has been a success; the programs implemented meet the objectives set out in the beginning and the tests run on the programs have all passed. The programs I have implemented can generate key pairs of up to 8152 bits in length. Being able to generate keys this long should mean they stay secure for a few years as it is infeasible to factor them.

While doing this project I have learnt a lot about cryptographic methods used in computing. I have also improved my knowledge of the C programming language.

11 Further Work

Possible suggestions for extensions to the work covered in this project are listed below:

11.1 Binary File Encryption

As mentioned earlier, the file size of the ciphertext version of the file is over twice the size of the original plaintext file. This is not ideal as file encryption is commonly used for transmitting data across networks and a large file size means longer transmission time.

One possible extension to this project would be to extend the file encryption implementation to encrypt files using blocks of binary data rather than blocks of letters, this would help reduce the size of the file after encryption.

11.2 Chinese Remainder Theorem

The Chinese Remainder Theorem is a mathematical theorem which can be used to decrypt the ciphertext back into the plaintext. This method is more efficient than the square and multiply method implemented in this project.

11.3 Open PGP

Another extension to the project would be to look at the compatibility of Open PGP keys with the encryption and decryption sections of my implementation. If they were compatible it would mean existing keys generated using Open PGP software could be used to encrypt and decrypt files using the algorithms implemented here, and vice versa; the keys generated here could be used with Open PGP implementations of the encryption and decryption algorithms.

12 Acknowledgements

Maths and project guidance: Ana Salagean
Proof Reading: Kelly Ralph, Anne Jones

13 References

13.1 Websites

Webopedia

What is Cryptography? – A Word Definition From the Webopedia Computer Dictionary

<http://www.webopedia.com/TERM/c/cryptography.html>

Last modified: 16th March 2004

Accessed: March–April 2007

Wikipedia

RSA – Wikipedia, the free encyclopaedia

<http://en.wikipedia.org/wiki/Rsa>

Last modified: 1st May 2007

Accessed: March–May 2007

13.2 Books

William Stallings *“Cryptography and Network Security Principles and Practices”*
Fourth Edition (Prentice Hall 2006)

13.3 Papers

RSA Laboratories *“PKCS #1 v2.1: RSA Cryptography Standard”* (June 14, 2002)

Can be found at: <http://www.rsa.com/rsalabs/node.asp?id=2125>

13.4 Libraries

DI Management Services *“BigDigits multiple-precision arithmetic source code”*,
Version 2.1

Downloaded: December 2006

Can be found at: <http://www.di-mgt.com.au/bigdigits.html>

GNU *“The GNU MP Bignum Library”*, Version 4.2.1

Downloaded: December 2006

Can be found at: <http://gmplib.org/>

14 Bibliography

14.1 Books

Richard A. Mollin “*RSA Public-Key Cryptography*” (Chapman & Hall 2002)

Dan Gookin “*C For Dummies*” Second Edition (Wiley Publishing Inc. 2004)

Peter Prinz & Tony Crawford “*C In A Nutshell*” (O’Reilly 2006)

Bradley L. Jones & Peter Aitken “*Sams Teach Yourself C in 21 Days*” Sixth Edition (Sams 2003)

14.2 Websites

Data encryption: information from Answers.com

<http://www.answers.com/cryptography>

Last modified: ?

Accessed: March–May 2007

RSA Algorithm

http://www.di-mgt.com.au/rsa_alg.html

Last modified: 2nd October 2006

Accessed: March–May 2007

RSA Security – Crypto FAQ

<http://www.rsasecurity.com/rsalabs/node.asp?id=2152>

Last modified: ?

Accessed: March–May 2007

Wikipedia – Symmetric-Key Algorithm

http://en.wikipedia.org/wiki/Symmetric-key_algorithm

Wikipedia – Various RSA and Cryptography Information

http://en.wikipedia.org/wiki/Main_Page

Last modified: ?

Accessed: March–May 2007

14.3 Papers

Rajorshi Biswas “*A Fast Implementation Of The RSA Algorithm Using The GNU MP Library*” (2003)

Can be found at: <http://www.cs.ucr.edu/~anirban/pubs.html>

15 Appendix A - Gantt Chart

16 Appendix B - Testing

Table 13 – Large Number Testing

Table 14 – Large Number Testing

Table 15 – Large Number Testing

Table 16 – Large Number Testing

Table 17 – Large Number Testing

Table 18 – Large Number Testing

17 Appendix C – Key C Source Files

17.1 Small Number Implementation – menu.c

```
// menu.c By Stephen Jones
// Provides the menu interface and links to the other source files

// Include standard headers and source files
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include "gen_keys.c"
#include "cipher.c"

// define the functions
int gen_keys(void);
int encrypt(void);
int decrypt(void);
int open_key(int file);

int main (void)
{
    int option;

    //display the menu
    printf("=====\n");
    printf("=                               =\n");
    printf("=  RSA Cryptography System  =\n");
    printf("=                               =\n");
    printf("=====\n");
    printf("          Main Menu          \n\n");
    printf("Please make your selection:\n\n");
    printf("  1 - Generate Keys\n");
    printf("  2 - Encrypt a number\n");
    printf("  3 - Decrypt a number\n");
    printf("  4 - View Private Key\n");
    printf("  5 - View Public Key\n");
    printf("  6 - Quit\n\n");

    //get choice off user
    scanf("%d", &option);

    //run the required function
    switch(option)
    {
        case 1:
            gen_keys();
            break;
        case 2:
            encrypt();
            break;
    }
}
```

```
    case 3:
        decrypt();
        break;
    case 4:
        printf("\n\nPrivate Key:\n");
        open_key(1);
        break;
    case 5:
        printf("\n\nPublic Key:\n");
        open_key(2);
        break;
    case 6:
        exit(1);
    default:
        printf("Unknown Key\n");
        printf("Please make another selection\n\n");
        break;
}
main();
return 0;
}

// the function to display a key file
int open_key(int file)
{
    unsigned int exponent, modulus;
    FILE *fptr;
    fptr = NULL;

    if (file == 1)
    {
        fptr = fopen("private_key.txt", "r");
    }
    if (file == 2)
    {
        fptr = fopen("public_key.txt", "r");
    }

    if (fptr != NULL) // check if file exists
    {
        fscanf(fptr, "%d %d", &exponent, &modulus);
        printf("%d %d \n\n", exponent, modulus);
        fflush(fptr);
        fclose(fptr);
    }
    else // if file doesn't exist print error
    {
        printf("ERROR: KEY FILE NOT FOUND\n\n");
    }
    return 0;
}
```

17.2 Small Number Implementation – gen_keys.c

```
// gen_keys.c By Stephen Jones
// Provides the key generation functions

//MAX and MIN values for random numbers
unsigned int MAX=256;
unsigned int MIN=1;

//function to calculate the multiplicative inverse of e mod r, using the
extended Euclidean algorithm
int gencongre1 (unsigned int e, unsigned int r)
{
    long temp, u, v, quotient, previous_u, previous_v;
    u = 0;
    previous_u = 1;
    v = 1;
    previous_v = 0;
    while (r != 0) //keep calculating until it reaches 0
    {
        temp = r; //store the old value
        quotient = e / r; //quotient of e and r
        r = e % r; //calculate the new value, e mod r
        e = temp; //restore the old value

        temp = u; //store the old value
        u = previous_u - quotient * u; //calculate the new u using the previous
one and quotient
        previous_u = temp; //restore the old value

        temp = v; //store the old value
        v = previous_v - quotient * v; //calculate the new v using the previous
one and exponent
        previous_v = temp; //restore the old value
    }
    return (previous_u);
}

//function to calculate the greatest common denominator of two numbers,
using the Euclidean algorithm
int gcd(unsigned int number, unsigned int r)
{
    unsigned int temp;
    while (r != 0 ) //keep calculating until it reaches 0
    {
        temp = r; //store the old value
        r = number % r; //calculate the new value, number mod r
        number = temp; //restore the old value
    }
    return number;
}
```

```
//function to generate a number coprime to r called from gen_keys
int gencoprime(unsigned int r)
{
    unsigned int number;
    do
    {
        do //Generate a random number between MIN and MAX
        {
            number = rand() % MAX;
        } while (number < 3);

    } while (gcd(number, r) != 1); //keep generating numbers until the gcd of
the generated number and r is 1
    return (number);
}

//function to generate prime numbers, called from gen_keys
int genprime (void)
{
    unsigned int modulus, modulus_squared, prime, number, result;
    prime = 0; //Assume the number is not prime
    result = 0;

    while (prime == 0) //While it is not prime
    {
        modulus = 2; //Can start at 2 dividing by 1 not needed
        modulus_squared = 4;

        do //Generate a random number between MIN and MAX
        {
            number=rand() % MAX;
        } while (number < MIN);

        while (modulus_squared < number) //Only need to check up to the square
root of the number
        {
            result = number % modulus; //The result is the remainder after
dividing the number by the current modulus
            if (result == 0) //If the remainder is 0
            {
                prime = 0; //The number is not prime
                break;
            }
            modulus_squared = modulus * modulus;
            if (modulus == 2) //If the current modulus is 2
            {
                modulus++; //Add one to make next modulus 3
            }
            else
            {
                modulus++; //Otherwise add 2 to keep modulus odd
                modulus++;
            }
        }
    }
}
```

```
    }
}

    if (result != 0) //If the result is 0 when divided by all numbers up to
the square root of the number
    {
        prime = 1; //Number must be prime
    }
}
return (number);
}

//key generation function called from menu.c
int gen_keys (void)
{
    unsigned int p, q, e, n, r;
    long d;
    int error;
    srand(time(NULL)); //randomise the random number generator
    error = 1;

    do //generate two different prime numbers
    {
        p = genprime();
        q = genprime();
    } while (q == p);

    n = p * q; //n is the product of p and q
    r = (p - 1) * (q - 1); //r is the totient of p and q
    e = gencoprime(r); //e is coprime to r
    d = gencongrrel(e, r); //d is the multiplicative inverse of e mod r

    if (d <= 0)
    {
        d = d + r; //d must be positive so calculate d + r to get d mod r
    }

    //Write the key files
    FILE *fptr;

    fptr = fopen("private_key.txt", "w");
    if (fptr != NULL) //check the file was opened correctly
    {
        fprintf(fptr, "%ld %d\n", d, n); //write the values
        fflush(fptr);
        fclose(fptr);
        error = 0;
    }

    fptr = fopen("public_key.txt", "w");
    if ((fptr != NULL) || (error == 0)) //check the file was opened correctly
    {
```

```
    fprintf(fptr,"%d %d\n", e, n); //write the values out
    fflush(fptr);
    fclose(fptr);
    error = 0;
}
else
{
    error = 1;
}

if (error == 0)
{
    printf("\nKEY GENERATION SUCCESSFULL\n\n\n"); //if there wasn't an
error
}
else
{
    printf("\nERROR: KEY GENERATION UNSUCCESSFULL\n\n\n"); //if there was
an error
    return(1);
}
return(0);
}
```

17.3 Small Number Implementation – cipher.c

```
// cipher.c By Stephen Jones
// Provides the encryption and decryption functions

//cipher function called from either encrypt or decrypt
int cipher(unsigned int exponent, unsigned int text, unsigned int modulus)
{
    unsigned long result;
    result = 1;

    //use the square and multiply algorithm to calculate the result
    while (exponent > 0)
    {
        if ((exponent & 1) == 1) // if the least significant bit of exponent is
a 1
        {
            result = (result * text) % modulus; // multiply result so far by text
and use modulus to keep result small
        }
        exponent >>= 1; // shift the exponent right by 1, losing the most
significant bit (0110 becomes 011)
        text = (text * text) % modulus; //square the text and use modulus to
keep small
    }
    return result;
}
```

```
//encrypt function called from menu.c
int encrypt (void)
{
    unsigned int plaintext, exponent, modulus, result;

    //Open public_key.txt and read out the two values stored in it
    FILE *fptr;
    fptr = fopen("public_key.txt","r");
    if (fptr != NULL) //check if file exists
    {
        fscanf(fptr,"%d %d", &exponent, &modulus); //read the values
        fflush(fptr);
        fclose(fptr);

        //Get plaintext off user it must be less than the modulus
        do
        {
            printf("Enter Plaintext, must be a positive number less than the
modulus n: %d:\n", modulus);
            scanf("%d",&plaintext);
        } while (plaintext >= modulus);

        //encrypt the plaintext using the cipher function
        result = cipher(exponent, plaintext, modulus);

        //print out the result for the user
        printf("Encrypted message is %d\n", result);
    }
    else //if the file doesn't exist print the error
    {
        printf("\nERROR: ENCRYPTION UNSUCCESSFULL\n\n\n"); //if there was an
error
        return(1);
    }
    return 0;
}

//decrypt function called from menu.c
int decrypt (void)
{
    unsigned int ciphertext, exponent, modulus, result;

    //Open private_key.txt and read out the two values stored in it
    FILE *fptr;
    fptr = fopen("private_key.txt","r");

    if (fptr != NULL)//check the file exists
    {
        fscanf(fptr,"%d %d", &exponent, &modulus); //read values
        fflush(fptr);
        fclose(fptr);
    }
}
```

```
//Get ciphertext of user
printf("Enter ciphertext:\n");
scanf("%d",&ciphertext);

//decrypt the message
result = cipher(exponent, ciphertext, modulus);

//print the result for the user
printf("Decrypted message is %d\n", result);
}
else //if the file doesn't exist print the error
{
printf("\nERROR: DECRYPTION UNSUCCESSFULL\n\n\n"); //if there was an
error
return(1);
}
return 0;
}
```

17.4 Large Number Implementation – gen_keys_big.c

```
// gen_keys_big.c By Stephen Jones
// Provides the key generation functions

//MAX and MIN values for random numbers
int SIZE = 512; //half bit size of key

//key generation function called from menu.c
int gen_keys_big (void)
{
mpz_t p, q, n, gcd, e, nm1, pm1, qm1, r, g, s, t;
int seed, prime_count1, prime_count2, error;
gmp_randstate_t state;

mpz_init (p);
mpz_init (q);
mpz_init (nm1);
mpz_init (pm1);
mpz_init (qm1);
mpz_init (n);
mpz_init (r);
mpz_init (e);
mpz_init (gcd);
mpz_init (g);
mpz_init (s);
mpz_init (t);

error = 0;
srand(time(NULL));
gmp_randinit_default(state);
seed = rand();
gmp_randseed_ui(state, seed);
```

```

do
{
do //generate two different prime numbers
{
mpz_urandomb (p, state, SIZE); //generate a random number up to SIZE
prime_count1 = mpz_probab_prime_p(p, 100); // check if probably prime
} while (prime_count1 == 0); //wont be prime if the return value is 0

do
{
do
{
mpz_urandomb (q, state, SIZE);
prime_count2 = mpz_probab_prime_p(q, 100);
} while (prime_count2 == 0);
} while (q == p); //make sure not equal

mpz_mul(n, p, q); //multiply p and q to get n
mpz_sub_ui(nm1, n, 1); // subtract 1 from n
mpz_sub_ui(pm1, p, 1); // subtract 1 from p
mpz_sub_ui(qm1, q, 1); // subtract 1 from q
mpz_mul(r, pm1, qm1); //multiply pm1 and qm1 to get r

do
{
do
{
mpz_urandomb (e, state, SIZE); //generate a random number
} while ((mpz_cmp_si(e, 3) < 0) || (mpz_cmp(e, nm1) > 0)); //make
sure 3 < e < n-1
mpz_gcd(gcd, e, r); // find the gcd of e and r
} while (mpz_cmp_si(gcd, 1) != 0); // continue until the gcd is 1
mpz_gcdext (g, s, t, e, r); // find the multiplicative inverse of e
} while (mpz_cmp_si (s, 1) < 0);

//Write the key files
FILE *fptr;
fptr = fopen("private_key.txt","w");
if (fptr != NULL) //check the file was opened correctly
{
gmp_fprintf(fptr,"%Zd %Zd\n", s, n); //write the values
fflush(fptr);
fclose(fptr);
error = 0;
}

fptr = fopen("public_key.txt","w");
if ((fptr != NULL) || (error == 0)) //check the file was opened correctly
{
gmp_fprintf(fptr,"%Zd %Zd\n", e, n); //write the values out
fflush(fptr);
fclose(fptr);
}

```

```
    error = 0;
}
else
{
    error = 1;
}
if (error == 0)
{
    printf("\nKEY GENERATION SUCCESSFULL\n\n\n"); //if there wasn't an
error
}
else
{
    printf("\nERROR: KEY GENERATION UNSUCCESSFULL\n\n\n"); //if there was
an error
    return(1);
}
return(0);
}
```

17.5 Large Number Implementation – cipher_big.c

// cipher_big.c By Stephen Jones

// Provides the encryption and decryption functions

//encrypt function called from menu_big.c

```
int encrypt_big (void)
{
    int error;
    mpz_t exponent, modulus, result, plaintext;

    mpz_init (exponent);
    mpz_init (result);
    mpz_init (plaintext);
    mpz_init (modulus);
    error = 0;

    //Open private_key.txt and read out the two values stored in it
    FILE *fptr;

    fptr = fopen("public_key.txt","r");
    if (fptr != NULL) //check if file exists
    {
        gmp_fscanf(fptr,"%Zd %Zd", &exponent, &modulus); //read the values in
it
        fflush(fptr);
        fclose(fptr);

        //Get plaintext off user it must be less than the modulus
        do
        {
            gmp_printf("Enter Plaintext, must be a positive number less than the
modulus n: %Zd:\n", modulus);
```

```
    gmp_scanf("%Zd",&plaintext);
} while (mpz_cmp(plaintext, modulus) >= 0 );

//encrypt the plaintext using the cipher function
mpz_powm(result, plaintext, exponent, modulus);

//print out the result for the user
gmp_printf("Encrypted message is %Zd\n", result);
}
else //if the file doesn't exist print the error
{
    printf("\nERROR: ENCRYPTION UNSUCCESSFULL\n\n\n"); //if there was an
error
    return(1);
}
return 0;
}

//decrypt function called from menu.c
int decrypt_big (void)
{
    mpz_t exponent, modulus, result, ciphertext;

    mpz_init (exponent);
    mpz_init (result);
    mpz_init (ciphertext);
    mpz_init (modulus);

    //Open private_key.txt and read out the two values stored in it
    FILE *fptr;
    fptr = fopen("private_key.txt","r");

    if (fptr != NULL)//check the file exists
    {
        gmp_fscanf(fptr,"%Zd %Zd", &exponent, &modulus); //read values
        fflush(fptr);
        fclose(fptr);

        //Get ciphertext of user
        printf("Enter ciphertext:\n");
        gmp_scanf("%Zd",&ciphertext);

        //decrypt the message
        mpz_powm(result, ciphertext, exponent, modulus);

        //print the result for the user
        gmp_printf("Decrypted message is %Zd\n", result);
    }
    else //if the file doesn't exist print the error
    {
        printf("\nERROR: DECRYPTION UNSUCCESSFULL\n\n\n"); //if there was an
error
```

```

    return(1);
}
return 0;
}

```

17.6 File Implementation – cipher_file.c

```

// cipher_file.c By Stephen Jones
// Provides the encryption and decryption functions

//encrypt function called from menu_file.c
int encrypt_file (void)
{
    unsigned long int counter, counter2, max, keysize, keydiff, error;
    int filechar;
    mpz_t exponent, modulus, result, plaintext, total, temp;
    mpz_init (exponent);
    mpz_init (result);
    mpz_init (plaintext);
    mpz_init (modulus);
    mpz_init(total);
    mpz_init(temp);
    size_t keylength;

    FILE *infile = fopen("input.txt", "rb");
    FILE *keyfile = fopen("public_key.txt", "r");
    FILE *outfile = fopen("output.txt", "w");
    error = 0;
    max = 0;
    keysize = 0;
    counter = 0;

    if (keyfile != NULL) //check if file exists
    {
        gmp_fscanf(keyfile, "%Zd %Zd", &exponent, &modulus); //read the values
in it
        fflush(keyfile);
        fclose(keyfile);

        keylength = mpz_sizeinbase(modulus, 2);
        keysize = keylength / 8;

        while (fgetc(infile) != EOF)
        {
            max++;
        }
        keydiff = keysize - (max % keysize);
        mpz_t buffer[max + keydiff];

        for (counter = 0; counter < max + keydiff; counter++)
        {
            mpz_init(buffer[counter]);
        }
    }
}

```

```
infile = fopen("input.txt", "rb");

//write initial number, filesize
mpz_set_ui(plaintext, max);
mpz_powm(result, plaintext, exponent, modulus);
if (outfile != NULL)
{
    gmp_fprintf(outfile, "%Zd\n", result);
}
else
{
    error = 1;
}

if (infile == NULL)
{
    error = 1;
}
for (counter = 0; counter < max; counter++)
{
    filechar = fgetc(infile);
    if (filechar == EOF)
    {
        error = 1;
    }
    mpz_set_ui(buffer[counter], filechar);
}

counter = 0;
counter2 = 0;
mpz_set_ui(plaintext, 0);

while (counter < max)
{
    mpz_set_ui(plaintext, 0);

    for (counter2 = 0; counter2 < keysize; counter2++)
    {
        mpz_mul_ui(temp, plaintext, 256);
        mpz_add(plaintext, temp, buffer[counter2 + counter]);
    }

    //encrypt the plaintext using the cipher function
    mpz_powm(result, plaintext, exponent, modulus);
    if (outfile != NULL)
    {
        gmp_fprintf(outfile, "%Zd\n", result);
    }
    else
    {
        error = 1;
    }
}
```

```
    }
    counter = counter + keysize;
}
fflush(infile);
fclose(infile);
fflush(outfile);
fclose(outfile);
}
else //if the file doesn't exist print the error
{
    error = 1;
}

if (error == 0)
{
    printf("\nENCRYPTION SUCCESSFUL\n\n\n"); //if there wasn't an error
}
else
{
    printf("\nERROR: ENCRYPTION UNSUCCESSFUL\n\n\n"); //if there was an
error
    return(1);
}
return(0);
}

//decrypt function called from menu.c
int decrypt_file (void)
{
    unsigned int filesize, character[65000], counter, counter2, counter3,
keydiff, keysize, diff, error;
    mpz_t exponent, modulus, ciphertext, temp, letter[65000], result[65000];
    char line [ 65000 ];
    size_t keylength;

    counter = 1;
    error = 0;
    mpz_init(temp);
    mpz_init (exponent);
    mpz_init (ciphertext);
    mpz_init (modulus);

    FILE *keyfile;
    FILE *outfile;
    FILE *decryptfile;
    keyfile = fopen("private_key.txt", "r");
    outfile = fopen ("output.txt", "r" );
    decryptfile = fopen("decryptedfile.txt", "w");

    //initialise arrays
    for (counter = 0; counter < 65000; counter++)
    {
```

```
mpz_init(letter[counter]);
mpz_init(result[counter]);
character[counter] = 0;
line[counter] = 0;
}

//Open private_key.txt and read out the two values stored in it
if (keyfile != NULL)//check the file exists
{
    gmp_fscanf(keyfile,"%Zd %Zd", &exponent, &modulus); //read values
    fflush(keyfile);
    fclose(keyfile);

    //calculate the key length, the number of letters encrypted in each
section
    keylength = mpz_sizeinbase(modulus, 2);
    keysize = keylength / 8;
    keydiff = keysize - 1;

    if ( outfile != NULL )
    {
        //work out the size of the original file
        fgets ( line, sizeof line, outfile );
        gmp_sscanf(line, "%Zd", &ciphertext);
        mpz_powm(result[keydiff], ciphertext, exponent, modulus);
        counter3 = 0;
        filesize = mpz_get_ui(result[keydiff]);

        while ( fgets ( line, sizeof line, outfile ) != NULL )
        {
            //read each line of the file and decrypt the letters into the
arrays
            gmp_sscanf(line, "%Zd", &ciphertext);
            mpz_powm(result[keydiff], ciphertext, exponent, modulus);
            mpz_cdiv_r_ui(temp, result[keydiff], 256);
            mpz_add_ui(letter[keydiff], temp, 256);
            character[keydiff] = mpz_get_ui(letter[keydiff]);
            counter3 = counter3 + keysize;

            for (counter = 0; counter < keysize; counter++)
            {
                mpz_sub(temp, result[keydiff - counter], letter[keydiff -
counter]);
                mpz_cdiv_q_ui(result[keydiff - counter - 1], temp, 256);
                mpz_cdiv_r_ui(temp, result[keydiff - counter - 1], 256);
                mpz_add_ui(letter[keydiff - counter - 1], temp, 256);
                character[keydiff - counter] = mpz_get_ui(letter[keydiff -
counter]);
            }

            //print the decrypted letters to the output file
            if (counter3 < filesize)
```

```
{
  for (counter2 = 0; counter2 < keysize; counter2++)
  {
    gmp_fprintf(decryptfile, "%c", character[counter2]);
  }
}
else
{
  diff = counter3 - filesize;
  for (counter2 = 0; counter2 < (keysize - diff - 1); counter2++)
  {
    gmp_fprintf(decryptfile, "%c", character[counter2]);
  }
}
}
fflush(outfile);
fclose(outfile);
}
else
{
  error = 1;
}
gmp_fprintf(decryptfile, "\n");
}
else
{
  error = 1;
}
fflush(decryptfile);
fclose(decryptfile);

if (error == 0)
{
  printf("\nDECRYPTION SUCCESSFUL\n\n\n"); //if there wasn't an error
}
else
{
  printf("\nERROR: DECRYPTION UNSUCCESSFUL\n\n\n"); //if there was an
error
  return(1);
}
return(0);
}
```