A FAST IMPLEMENTATION OF THE RSA ALGORITHM USING THE GNU MP LIBRARY

Rajorshi Biswas
Shibdas Bandyopadhyay
Anirban Banerjee
IIIT-Calcutta

ABSTRACT

Organizations in both public and private sectors have become increasingly dependent on electronic data processing. Protecting these important data is of utmost concern to the organizations and cryptography is one of the primary ways to do the job. Public Key Cryptography is used to protect digital data going through an insecure channel from one place to another. RSA algorithm is extensively used in the popular implementations of Public Key Infrastructures. In this paper, we have done a efficient implementation of RSA algorithm using gmp library from GNU. We have also analyzed the changes in the performance of the algorithm by changing the number of characters we are encoding together (we termed this procedure as bitwise incremental RSA).

Key Words

Cryptography, RSA, PKI, GMP.

1. INTRODUCTION

Data communication is an important aspect of our living. So, protection of data from misuse is essential. A cryptosystem defines a pair of data transformations called encryption and decryption. Encryption is applied to the plain text i.e. the data to be communicated to produce cipher text i.e. encrypted data using encryption key. Decryption uses the decryption key to convert cipher text to plain text i.e. the original data. Now, if the encryption key and the decryption key is the same or one can be derived from the other then it is said to be symmetric cryptography. This type of cryptosystem can be easily broken if the key used to encrypt or decrypt can be found. To improve the protection mechanism Public Key Cryptosystem was introduced in 1976 by Whitfield Diffe and Martin Hellman of Stanford University. It uses a pair of related keys one for

encryption and other for decryption. One key, which is called the private key, is kept secret and other one known as public key is disclosed.

The message is encrypted with public key and can only be decrypted by using the private key. So, the encrypted message cannot be decrypted by anyone who knows the public key and thus secure communication is possible. RSA (named after its authors – Rivest, Shamir and Adleman) is the most popular public key algorithm. In relies on the factorization problem of mathematics that indicates that given a very large number it is quite impossible in today's aspect to find two prime numbers whose product is the given number. As we increase the number the possibility for factoring the number decreases.

So, we need very large numbers for a good Public Key Cryptosystem. GNU has an excellent library called GMP that can handle numbers of arbitrary precision. We have used this library to implement RSA algorithm. As we have shown in this paper number of bits encrypted together using a public key has significant impact on the decryption time and the strength of the cryptosystem.

2. REVIEW OF EXISTING LITERATURE

Authentication protocols and their implications are discussed in [1]. Computing inverse of a shared secret modulus, which involves mathematical formulation of RSA, is discussed in [2]. Application of hash function in the field of cryptography is discussed in [3]. The strength of RSA algorithm is discussed in [4]. A survey of fast exponentiation method is done in [5]. Cryptosystem for sensor networks is studied in [6]. Security proofs for various digital signature scheme is studied in [7]. Multiparty authentication services and key agreement protocols are discussed in [8]. Various fast RSA implementations are described in [9]. An efficient implementation of RSA is discussed in [10]. The basic RSA algorithms and other cryptography related issues are discussed in [11].

3. SCOPE OF PRESENT WORK

Our work in this paper is focused primarily on the implementation of RSA. For efficient implementation we have used the GMP library, we have explored the behaviour

and feasibility of the algorithm with the change of various input parameters, and finally a user interface is developed to provide an application of our analysis.

4. REVIEW OF THE RSA ALGORITHM

4.1 Introduction

The RSA public key cryptosystem was invented by R. Rivest, A. Shamir and L. Adleman. The RSA cryptosystem is based on the dramatic difference between the ease of finding large primes and the difficulty of factoring the product of two large prime numbers (the integer factorization problem). This section gives a brief overview of the RSA algorithm for encrypting and decrypting messages.

4.2 Key generation

For the RSA cryptosystem, we first start off by generating two large prime numbers, 'p' and 'q', of about the same size in bits. Next, compute 'n' where $n = pq$, and 'x' such that, $x = (p-1)(q-1)$. We select a small odd integer less than x, which is relatively prime to it i.e. $\gcd(e,x) = 1$. Finally we find out the unique multiplicative inverse of e modulo x, and name it 'd'. In other words, $ed = 1 \pmod{x}$, and of course, $1 < d < x$. Now, the public key is the pair (e,n) and the private key is d.

4.3 RSA Encryption

Suppose Bob wishes to send a message (say 'm') to Alice. To encrypt the message using the RSA encryption scheme, Bob must obtain Alice's public key pair (e,n). The message to send must now be encrypted using this pair (e,n). However, the message 'm' must be represented as an integer in the interval [0,n-1]. To encrypt it, Bob simply computes the number 'c' where $c = m^e \bmod n$. Bob sends the ciphertext c to Alice.

4.4 RSA Decryption

To decrypt the ciphertext c, Alice needs to use her own private key d (the decryption exponent) and the modulus n. Simply computing the value of $c^d \bmod n$ yields back the decypted message (m).

Any article treating the RSA algorithm in considerable depth proves the correctness of the decryption algorithm. And such texts also offer considerable insights into the various security issues related to the scheme. Our primary focus is on a simple yet flexible implementation of the RSA cryptosystem that may be of practical value.

5. OUR IMPLEMENTATION OF THE RSA ALGORITHM

5.1 Introduction

We have implemented the RSA cryptosystem in two forms : a console mode implementation, as well as a user friendly GUI implementation. We focus on the console mode implementation here, and leave the GUI implementation for a later section of this report. The console application uses a 1024 bit modulus RSA implementation, which is adequate for non-critical applications. By a simple modification of the source code, higher bit-strengths may be easily achieved, albeit with a slight performance hit.

5.2 Handling large integers and the GMP library

Any practical implementation of the RSA cryptosystem would involve working with large integers (in our case, of 1024 bits or more in size). One way of dealing with this requirement would be to write our own library that handles all the required functions. While this would indeed make our application independent of any other third-party library, we refrained from doing so due to mainly two considerations. First, the speed of our implementation would not match the speed of the libraries available for such purposes. Second, it would probably be not as secure as some available open-source libraries are.

There were several libraries to consider for our application. We narrowed the choice to three libraries: the BigInteger library (Java), the GNU MP Arbitrary Precision library (C/C++), and the OpenSSL crypto library (C/C++). Of these, the GMP library (i.e. the GNU MP library) seemed to suit our needs the most.

The GMP library aims to provide the fastest possible arithmetic for applications that need a higher precision than the ones directly supported under C/C++ by using

highly optimized assembly code. Further the GMP library is a cross-platform library, implying that our application should work across platforms with minimal modifications, provided it is linked with the GMP library for the appropriate platform. We have used the facilities offered by the GMP library heavily throughout our application. The key generation, encryption and decryption routines all use the integer handling functions offered by this library.

5.3 Application overview

In this subsection, we present the basic structure of the console mode RSA implementation. The program is meant for use on a per user basis where each user's home directory stores files containing the private and public keys for the particular user. The application stores the private and public keys for a user in the files $HOME/.rsaprivate and $HOME/.rsapublic respectively.

At the very beginning the program searches for the existence of the aforementioned files and reads in the values of the private and public keys. If they are not present (as when the application is run for the first time), the program proceeds to generate the keys and writes them to the files.

Following this, the user is presented with a menu, asking him whether he would like to encrypt a file, decrypt an encrypted file, or quit the application. If the user chooses to encrypt a file, he is asked to enter the path to the file containing the recipient's public keys as well as the number of characters to encrypt at a time (this is justified later). If decryption is chosen, the path to the encrypted file is requested and the program subsequently decrypts the file to the standard output.

5.4 RSA key generation

The generation of the RSA keys is of paramount importance to the whole application. The application maintains a constant named 'BITSTRENGTH' which is the size of the RSA modulus (n) in bits. According to this setting, two character arrays to contain the digits of the primes p and q are declared. A simple loop through all the digits

of this array initializes the array with a random string of bits. We have used C's inbuilt random number generation routines to generate the bits of the string. The random generator is seeded using the srand() routine by the return value of the function time(), which returns the time since the epoch (00:00:00 UTC, January 1, 1970), measured in seconds. Key generation at the same return value of time() is avoided by sleep(), which delays the execution by one second, thus ensuring that the random numbers are never repeated.

At the end of this process, we have strings containing binary representations of the numbers p and q, but they are not prime yet. To achieve that, two gmp integers are first initialized with the contents of these strings. Then, the function mpz_nextprime() is called, which changes p and q to the next possible primes. This function uses a probabilistic algorithm to identify primes. According to the gmp manual, it is adequate for practical purposes and the chance of a composite passing is extremely small.

Now that we have the two 512-bit primes p and q, calculating the values of n (=pq) and x (=(p- 1)*(q-1)) is a simple matter of invoking mpz_mul() with the proper arguments. Next, to determine the value of 'e', we started with a value of 65537, and incremented it by two each time until the condition gcd(e,x) = 1 is satisfied (which, incidentally, is almost always satisfied by the number 65537 itself).

Now there exists a procedure in the gmp library with the prototype int mpz_invert(mpz_t ROP, mpz_t OP1, mpz_t OP2) which computes the multiplicative inverse of OP1 modulo OP2 and puts the result in ROP. Using this function helps us avoid writing our own routine based on the Extended Euclidean Algorithm (as this function executes extremely fast). In this way, we obtain the value of d, which completes our quest for the RSA keys. Finally the keys (d,e,n) are stored in two files .rsapublic and .rsaprivate, both in the user's home directory.

It is to be noted that the entire application can use a higher (or lower) bitstrength of the RSA modulus n by simply changing the constant BITSTRENGTH at the very beginning of the source-code.

5.5 RSA encryption

As the application is executed, the existence of the key files are checked and if they do not exist, the RSA keys are generated. Following this, the user is presented with a menu, which enables him to choose to encrypt, decrypt or quit.

First, the path to the file containing the public key of the recipient is requested from the user. However the user is also given the option of using his own public keys (in which case, only he can decrypt the message). Using the values of e and n read from the relevant file containing the public keys, the message is encrypted.

A critical requirement for the proper functioning of the RSA algorithm is that the message m must be represented as an integer in the range [0,n-1] (where n is, as usual, the RSA modulus). Our application converts text messages to integers by using a simple mapping of every character to its ASCII code. But encrypting only one character at a time is not only expensive in terms of the time required to encrypt and decrypt, but also in terms of security. This is because the encrypted integers would then only be from a small finite set (containing a maximum of as many integers as the number of ASCII characters). Hence, we ask the user the number of characters to encrypt at a time. For an RSA encryption scheme with the modulus size of 1024 bits, we have seen that about 100 characters can be encrypted at once. Lesser number of characters cause encryption and (especially) decryption to take significantly longer, whereas higher number of characters often violate the condition that the message m must lie in the interval [0,n-1].

The entire file to encrypt is processed as a group of strings each containing the specified number of characters (except possibly the last such string). Each character in such a string is converted to its 3- character wide ASCII code, and the entire resulting numeric string is our message m. Encrypting it is achieved by computing m ^ e mod n.

There is a gmp routine specifically for such a computation having the prototype void mpz_powm (mpz_t ROP, mpz_t BASE, mpz_t EXP, mpz_t MOD) which sets ROP to (BASE raised to EXP) modulo MOD. Thus invoking moz_powm(c,m,e,n) stores the encrypted partial message in the integer c. This is written each time to the file to encrypt to until the whole message has been processed. This completes the encryption process.

5.6 RSA decryption

If, in the menu, the user chooses to decrypt an encrypted file, the decryption routine is invoked. The operation of this routine is really quite straightforward. From the file to decrypt (the path to which is input from the user), the function processes each encrypted integer. It does so by computing the value of c ^ d mod n by invoking gmp_powm(m,c,d,n) and stores the decrypted part in m. Of course, the values of d and n are read in beforehand.

Here m however contains the integer representation of the message i.e. it is a string of numbers where each 3 character sequence signifies the ASCII code of a particular character. An inverse mapping to the relevant character is carried out and the message, now as was in the original file is displayed on the standard output. The decryption process is over once all the integers (ciphertext) in the encrypted file are processed in the described manner.

This completes a discussion of our console mode implementation of the RSA algorithm. This public key infrastructure has been tested in a multi-user scenario successfully. The following section analyses the time required for the various functions of our implementation, varying quite a few parameters.

6. TIME ANALYSIS

6.1 Timings for 1024-bit RSA (without compiler optimization)

All the times recorded below have been measured on a 733 MHz Pentium class processor, using the time measurement functions offered by the C library on a GNU/Linux platform (kernel 2.4.20).

Key generation : 0.465994 seconds (averaged over 5 samples)

The following times were recorded while encrypting/decrypting a file with exactly 10,000 characters:

Number of characters	Encryption time	Decryption time
1	5.068153	254.349886
25	0.219304	11.367492
50	0.128547	6.293279
75	0.096291	4.419277
100	0.078851	3.365591

(All times per 10,000 characters)

6.2 Timings for RSA for varying bit strengths

By varying the constant representing the bit-strength, RSA moduli of other sizes may be used quite easily. The following times were recorded using the same input file (10,000 characters):

Bit Strength	Chars at once	Key Generation	Encryption	Decryption
512	50	0.057984	0.054362	0.903180
768	75	0.194653	0.065302	2.098904
1024	100	0.465994	0.078851	3.365591
1280	125	0.657473	0.089712	4.437929
1536	150	1.613467	0.105439	5.804798
1792	175	2.057411	0.116585	7.430849
2048	200	4.052181	0.126361	9.001286

(All times per 10,000 characters)

While the 512-bit RSA is definitely the fastest among the ones shown, it is not the most secure, providing marginal security from a concerted attack. The slowest (2048-bit RSA) should be used in critical situations since it offers the maximum resistance to attacks. In our opinion the 1024-bit modulus is a good balance between speed and security.

7. A GRAPHICAL USER INTERFACE TO THE RSA CRYPTOSYSTEM

7.1 Introduction

This section describes the GUI version of our implementation of the RSA algorithm. While the concepts and libraries used are essentially the same, we briefly describe the implementation with special regard to the considerations that were unique to the graphical version. Finally we present the reader some screenshots of the application.

7.2 Implementation

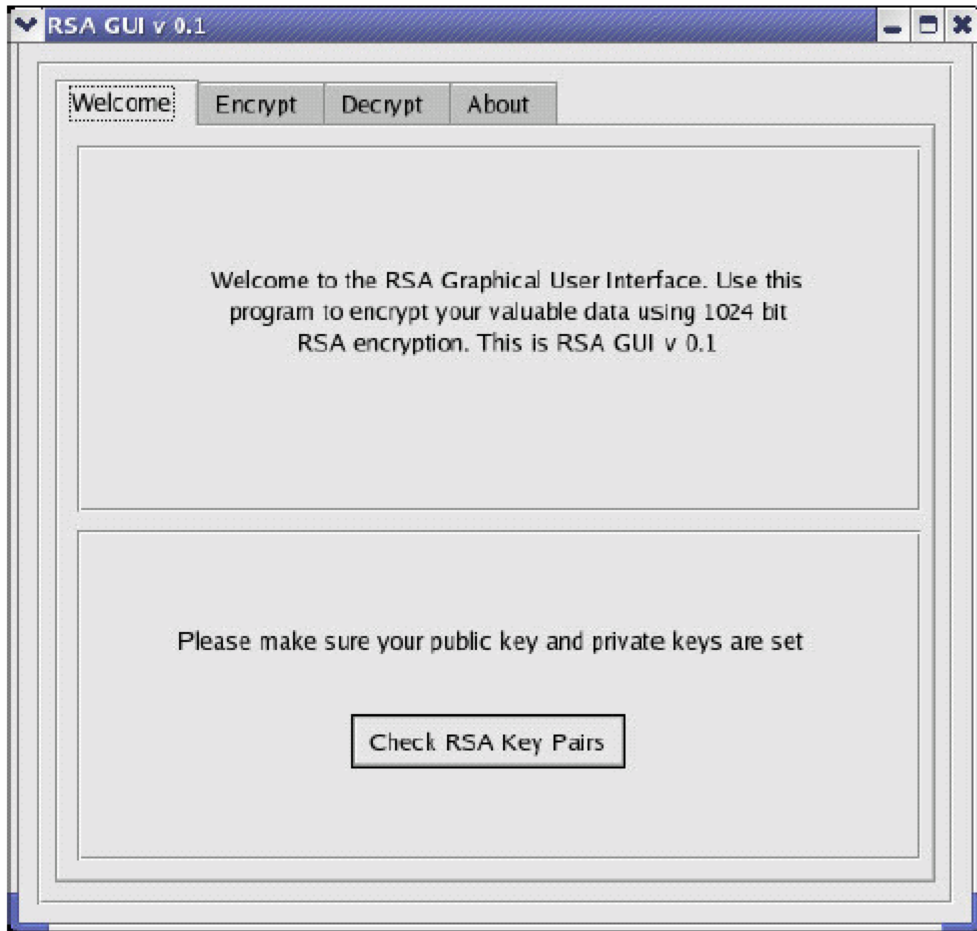
The GUI application was developed using the KDE/Qt libraries on Red-Hat Linux version 8.0. We used KDevelop 2.1 as our integrated development environment.

Our application consists of three C++ classes, of which the class named RSA is the most important. It provides slots (signal handlers) for encrypting files, decrypting files, mailing the encrypted file to another user, loading the values of the RSA keys from the key-files, saving encrypted/decrypted files and so on. One notable difference as far as features are concerned is that the GUI application does not ask the user the number of characters to enccrypt together.

The performance of the GUI version is similar to the console mode, since the underlying algorithms are the same. But as any program running on an X-server does require considerable amount of memory, the speed might be somewhat slower than the console version on a system without adequate memory.

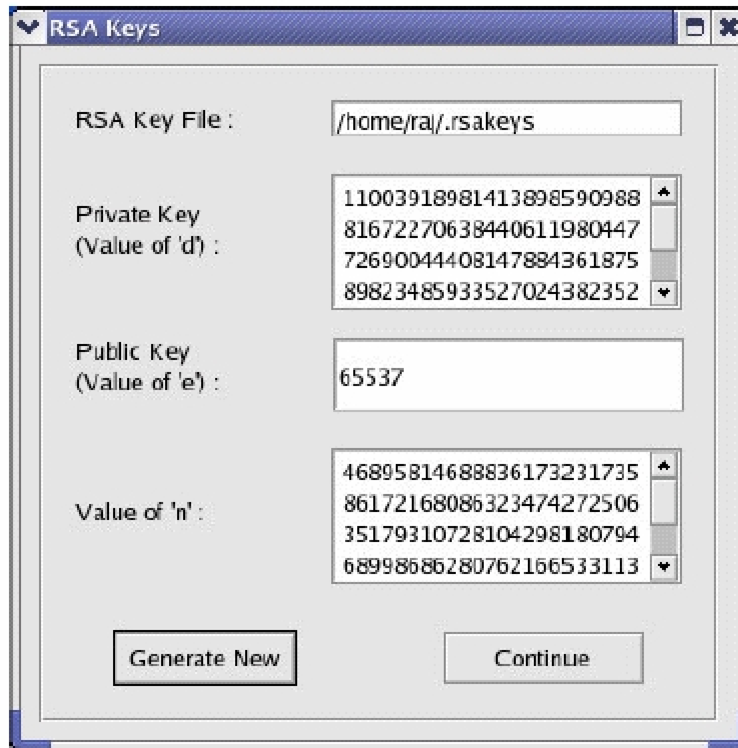
7.3 Screenshots of the application.

The following screen shows the main window of our RSA GUI:

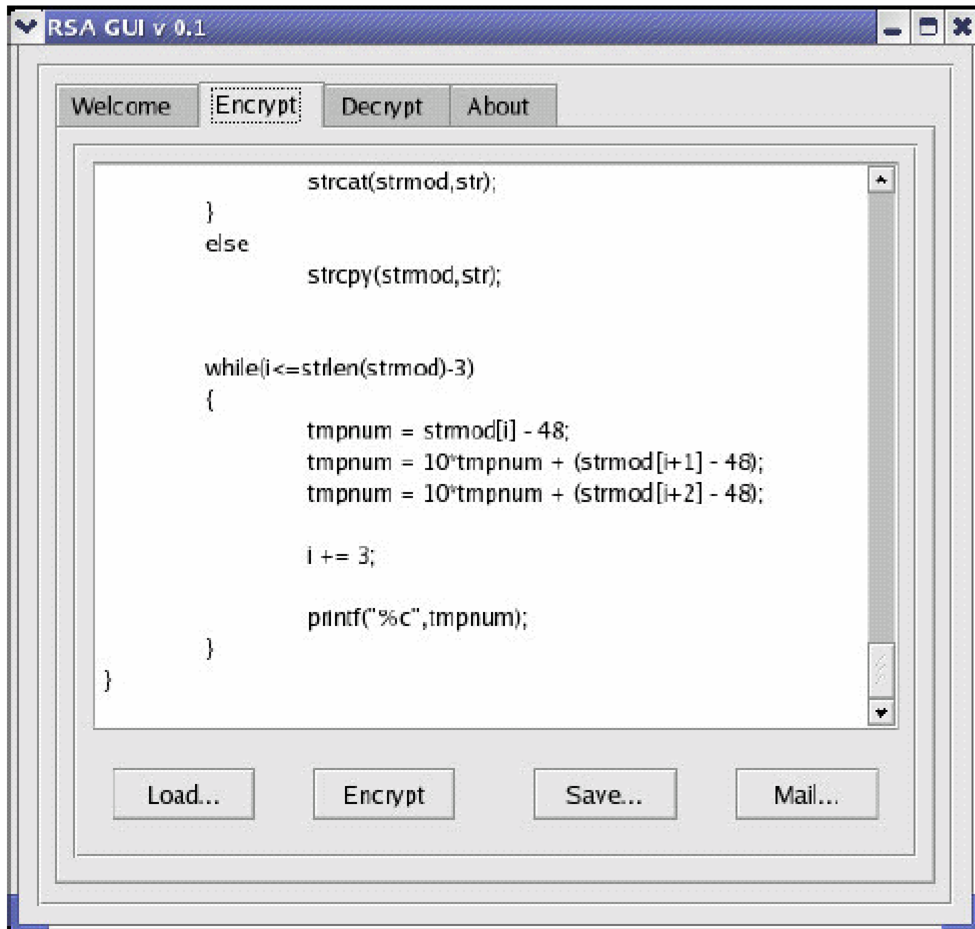


When the user clicks on the “Check RSA Key Pairs” button, if the key files do not exist (as in the first time), the following message is shown:

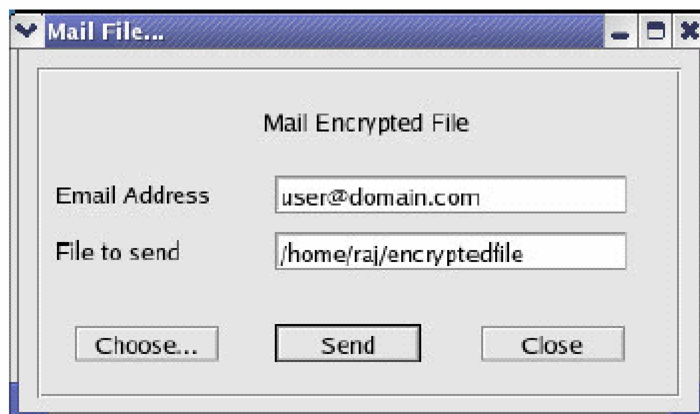
The user is taken to the key generation dialog, whose functions are pretty self-explanatory. This dialog is shown next:



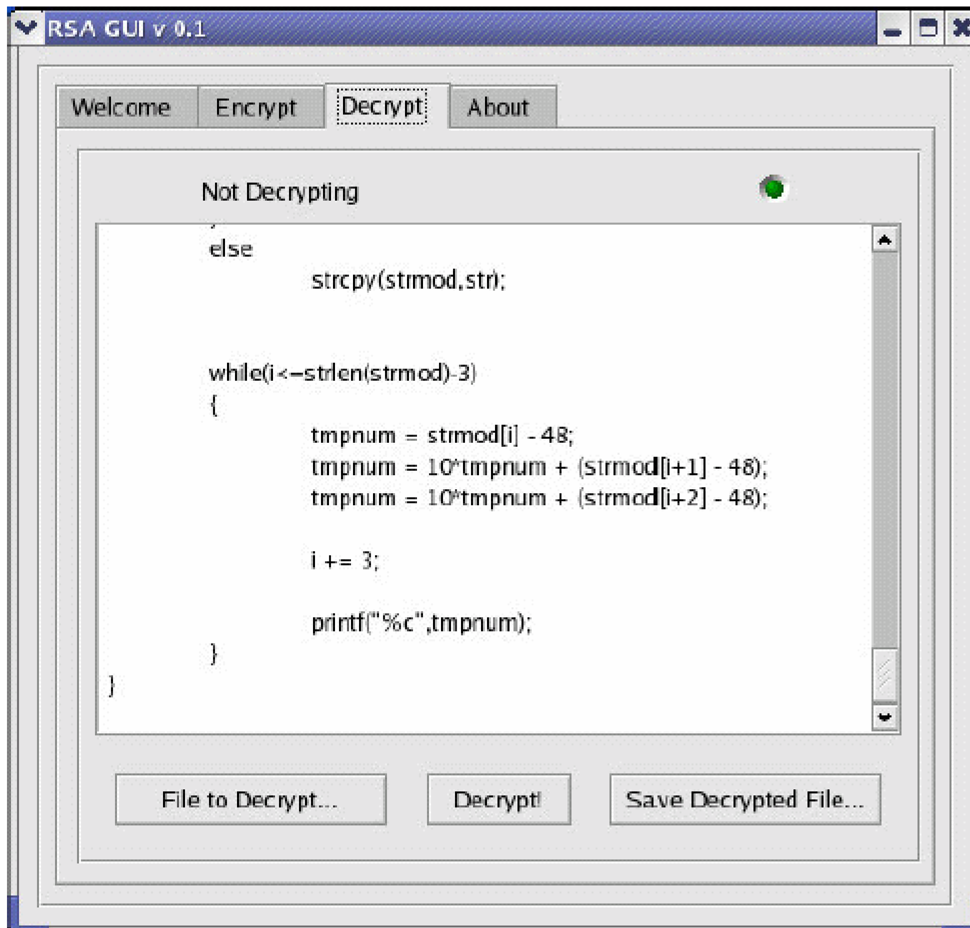
The following diagram shows the “Encrypt” tab of the main window:



The encrypted file can be mailed to another user by clicking on the “Mail…” button:



And finally, here is the similar looking "Decrypt" tab:



The decrypted file may be saved using the "Save Decrypted File…" button, which opens up a file - dialog for saving the decrypted file.

8. CONCLUSION

In this paper an efficient implementation of RSA is shown by using various functions of the GMP library. Feasibility analysis is done by comparing the time taken for encryption and decryption. It shows that when we increase the number of bits of information to be encrypted together the total time including encryption and decryption

steadily decreases. It must always be kept in mind that the integer representation of the message to be encrypted should lie within the range specified by the modulus (that is, m lies in the range [0,n-1]), which poses a limitation on the maximum number of characters that can be encrypted at a single time.

9. REFERENCES

[1] Paul Syversion and Illiano Cervesato, *The logic of authentication protocols*, FOSAD'00, Bertinoro, Italy, 2000.

[2] Dario Catalano, Rosario Gennaro and Shai Halevi, *Computing inverse over a shared secret modulus*, IBM T. J. Watson Research center, NY, USA, 1999.

[3] Don coppersmith, Markus Jakobsson, *Almost optimal hash sequence traversal*, RSA Laboratories, NY, 2001.

[4] Elichiro Fujisaki, Tatsuaki Okamoto, David Pointcheval and Jacques Stern, *RSA-OAEP is secure under the RSA assumption*, Journal of Cryptology, 2002.

[5] Daniel M. Gordon, *A survey of fast exponentiation methods*, Journal of algorithms, 27, 1998, 126-146.

[6] Adrian Perrig, Robet Szewczyk, Victor Wen, David Culler and J.D. Tygar, *SPINS: Security protocols for sensor networks*, Mobile Computing and Networking, Rome, Italy, 2001.

[7] David Pointcheval and Jacques Stern, *Security proofs for signature schemes*, EUROCRYPT '96, Zaragoza, Spain, 1996.

[8] Giuseppe Ateniese, Michael Steiner, and Gene Tsudik, *New multiparty authentication services and key agreement protocols*, IEEE Journal of Selected Areas in Communication, 18(4), 2000.

[9] Cetin Kaya Koc, *High speed RSA implementation*, RSA Laboratories, CA, 1994.

[10] Anand Krishnamurthy, Yiyan Tang, Cathy Xu and Yuke Wang, *An efficient implementation of multi-prime RSA on DSP processor*, University of Texas, Texas, USA,2002.

[11] A. Menezes, P. Van Oorschot, S. Vanstone, *Handbook of Applied Cryptography,* CRC Press, 1996 (www.cacr.math.uwaterloo.ca/hac)