Cipher This

Category A: Competitive

New Mexico High School
Supercomputing Challenge
Final Report
April 4, 2001

029
Highland

Team Members
Dawud Shakir
Candice Woodard
Yousuf Shakir
Robert Jones

Teacher
J. K. Raloff

Project Mentor
D. Downs

Table of Content

1. Executive Summary

An Efficient Method For RSA Decryption

Security of information is a national security issue in addition to being of commercial importance. It is very important to test the security of encryption methods used such as the RSA encryption technique. The RSA represents a class of encryption methods called public key systems where a key, n, is publicly known. We have developed an original method of factoring the public key n, to find out the prime numbers p and q which effectively allows us to break the encryption. We show that the method is efficient when p and q are close. The number of trials needed to factor the public key is quadratic and is given by $\frac{(p-q)^2}{8\sqrt{n}}$. We wrote a computer program using the C++ language. The program can handle integers of any length, which is necessary since the RSA method usually uses huge integers larger than 100 digits.

2. The History of Cryptography

The use of encryption can be traced back to 2000 BC to the Egyptians and their use of hieroglyphics. Hieroglyphics were meant to be cryptic however, they were not intended to hide text, rather they were used for Kings because they were thought to look more regal and prestigious. Much like the Egyptians the Mayans also used a language system based on the use of pictures. These early encrypted languages served the purpose of being a basis of communication rather than a means for concealing information. Julius Caesar and his substitution cipher, the Caesar Cipher, is one of the earliest encryption's that had the sole purpose of concealing information as he used it to protect private communications between Roman legions that were scattered over Europe, Africa and the Middle East. The Caesar Cipher, as previously stated, was a substitution or additive cipher that worked by replacing each letter of the alphabet with the third letter that came after it. Similar to Caesar's Cipher, the One Time Pad developed by AT&T engineer, Gilbert Vernam, was also an additive cipher. Developed to aid the war effort, like most of the advancements in cryptology were, the One Time Pad was used as a new perfect security to replace the failed codebook system. With this system each plaintext is enciphered using an additive cipher shift, but the catch and element that made this system so secure was that for every letter the shift is different, hence the name One Time. During the nineteenth century an approach using statistics was favored. Using the frequency average of each letter of the alphabet codebreakers were able to devise a system in which they would go through a document, count the letter frequencies and then compare them to those of the alphabet. Throughout history there were dozens of advancements made. The Arabs, Greek, English and Americans all came up with their own ciphering methods that were more or less effective.

3. Introduction

RSA, named after it's inventors (Rivest, Shamir, and Adleman) is an encryption system made public in the mid seventies. Since RSA's advent the government and big businesses have used the system to encrypt some of their most sensitive information. Today with so much of the world's population owning a computer and using the internet, it was only a matter of time before the internet branched out and became the world's forum for marketing. In today's society it has become second nature for people to do shopping over the internet. There are no limits as to what type of business one can find. One can buy a car, do their banking, find clothing, purchase food, or find and purchase a home. The list has really become endless. However, the comfort of shopping at home does come at a price. Security. Home addresses, social security numbers, telephone numbers, full names, background history as well as credit card numbers are all information that a consumer can be required to send over the internet in order to make purchases. This very personal and sensitive information when in the wrong hands could mean a major dishevelment of a person's life. Lately, there have been numerous news reports of hackers breaking into the hard drive of big businesses and even the government. Security is an issue of major importance right now. With so many

businesses relying on the inability to decode RSA, it is important to test the strength of this encryption program.

4. Statement of the problem

The problem we are investigating concerns finding a method to break an RSA encryption. We assume that we know the public key n for an RSA encryption and we try to find the two prime number factors p and q such that $n = p \cdot q$. We first describe the RSA encryption method, then we explain our method of solution.

5. The RSA public-key encryptosystem

A user (say, Alice) of the RSA public-key cryptosystem selects two large primes *p* and *q,* and computes their product *n,* which is known as her *public modulus*. She also selects an integer $e > 2$ such that

$$\gcd(e, (p-1)(q-1)) = 1 \qquad (1)$$

e is known as her *encryption exponent*. Alice publishes the pair (*e, n*) as her *public keys.* She then computes *d,* or her *decryption exponent,* using the *Euclidean formula*, such that

$$de \equiv 1 \pmod{\phi(n)} \qquad (2)$$

where,
$$\phi(n) = (p-1)(q-1) \qquad (3)$$

$\phi(n)$ is referred to as Euler's totient function,

$$\phi(n) = |\{a : 0 < a \leq n \text{ and } \gcd(a, n) = 1\}|$$

which for $n = p \cdot q$ is $\phi(n) = (p-1)(q-1)$. Actually, eq. (2) can be improved slightly to

$$de \equiv 1 \pmod{\lambda(n)}$$

where,
$$\lambda(n) = \text{lcm}(p-1, q-1)$$

After solving for *d,* she retains the pair (*d, n*) as her *private keys.*

Another user (say, Bob) can send Alice an encrypted message M, where $0 \leq M < n$, by sending her the ciphertext

$$C = M^e \pmod{n} \qquad (4)$$

computed using Alice's public keys (*e, n*). When Alice receives C she can decipher it using the equation

$$M = C^d \pmod{n} \qquad (5)$$

computed using her private keys (d, n). Since no one but Alice possesses d, no one but Alice should be able to compute M from C.

A text message would first be transformed to a number before the RSA cryptosystem could be used. For example,

(1) Suppose Alice chooses three primes $p = 17$, $q = 11$, and $e = 7$. Normally, these primes would be much larger.

(2) That means $n = p \times q = (17)(11) = 187$. She then publishes her *public keys* (n, e) which are now $n = 187$ and $e = 7$.

(3) Bob wishes to send the letter 'X' securely to Alice, which is equivalent to 88 in ASCII. He looks up her *public keys* and encrypts his message

$$C = M^e \bmod n$$
$$C = 88^7 \bmod 187$$
$$C = 11$$

Thus, he sends the encrypted message C = 11 to Alice.

(4) Alice now wants to decrypt Bob's message. She proceeds by then finding her *decryption exponent* as follows

$$de = 1 \pmod{(p-1) \times (q-1)}$$
$$d \times 7 = 1 \pmod{(17-1) \times (11-1)}$$
$$d \times 7 = 1 \pmod{160}$$
$$d \times 7 = 161$$
$$d = 23$$

(d is found using the *Euclidean formula*)

(5) Decrypting Bob's message is now simple

$$M = C^d \bmod n$$
$$M = 11^{23} \bmod 187$$
$$M = 88 = X \text{ in ASCII}$$

6. Breaking the RSA cryptosystem

For an encrypted message to be safe, p, q, e, and even the message itself, must be large in order to protect against someone decrypting it. With sufficiently large values of p and q,

RSA is impregnable. For a cryptanalyst (say, Eve) to crack an encrypted message, she must first know p or q. To do this, she must first factor Alice's public key n since $n = p \times q$.

7. Previous attempts to factor *n*

There are various methods used to factor n. The obvious approach would be to divide every odd integer up to $\sqrt{n}$ into n until either p or q were found. This method is inadequate, due to the enormous time[1] it would take to factor large numbers. Therefore, cryptanalysts needed a faster, more efficient way to factor n. All current methods to factor n depend on a "condition" to be able to finish in a feasible amount of time. For example,

1.) Algorithms whose running time depends mainly on the size of n

2.) Algorithms whose running time depends mainly on the size of p and the size of q

3.) Algorithms whose running time depends on the size of $p - 1$, $p - 2$, or $p + 1$

4.) Algorithms whose running time depends on the 'closeness' of p and q

As seen later, our proposed method is based on condition four.

Since the RSA system was invented, people have been trying to find efficient methods to factor n. Some of these where,

7.1 Pollard's p – 1 Method for Factoring

Pollard's p – 1 method is a technique for splitting a given composite number n that is divisible by at least two distinct primes, using any given multiple m of p – 1, for some prime factor p of n.

1. Choose an element a at random for $\mathbf{Z} = \{1, 2, \ldots, n - 1\}$.
2. Compute $d = \gcd(a, n)$. If $d > 1$, report that d is a factor of n and halt.
3. Compute $x = a^m \bmod n$.
4. Compute $d = \gcd(x - 1, n)$. If d > 1, report that d is a factor of n and halt.
5. If $x = 1$ and m is even, set $m \leftarrow m/2$ and return to step 3.
6. Report failure to find a factor. Halt.

[1] At $n = 10^{308}$, the combined effort of a hundred million personal computers would take more then one thousand years to crack such a cipher.

7.2 The Elliptic Curve Method for Factoring

A generalization of the p – 1 method, the elliptic curve method is considered one of the most efficient methods of factoring. We will not discuss the theory of the method here, but will mention that the success of the elliptic curve method depends on the likely situation that an integer "close to" p has only "small" prime factors.

7.3 Cycling Attacks

Not involving any factorization of *n*, this method involves taking an encrypted message and cycling values for the cyphertext until the correct value is attained

$$M = C^e \bmod n$$

8. Our Method of Solution

Our proposed method utilizes the two defining equations

$$n = p \cdot q \tag{1}$$

$$\phi(n) = (p - 1)(q - 1) \tag{2}$$

where p and q are prime numbers. The number n is the public key and $\phi(n)$ is a quantity called the Totient function. Therefore using eqs. (1) and (2) we have,

$$\phi(n) = (p - 1)\left(\frac{n}{p} - 1\right)$$

$$\phi(n) = n - p - \frac{n}{p} + 1$$

$$\phi(n) \cdot p = n \cdot p - p^2 - n + 1$$

$$0 = p^2 - (n - \phi(n) + 1)p + n \tag{3}$$

Using the solution for a quadratic equation we get,

$$p, q = \frac{(n - \varphi(n) + 1) \pm \sqrt{(n - \varphi(n) + 1)^2 - 4n}}{2} \tag{4}$$

Hence, the difference between p and q is

$$| p - q | = \sqrt{(n - \varphi(n) + 1)^2 - 4n}$$

Therefore, if we set L to

$$L = \frac{1}{2}(n - \phi(n) + 1)$$

and j equal to the average difference between p and q; $j = \frac{1}{2}(p - q)$; we get

$$j^2 = \frac{1}{4}(n - \phi(n) + 1)^2 - n$$

$$j = \sqrt{L^2 - n} \qquad (5)$$

Thus, we need to solve for L such that L^2-n is a complete square. It is more convenient to write L as (m + r), where m = Int(√n). Hence, the working equation is

$$j = \sqrt{(m + r)^2 - n} \; ; \qquad r \in \{0, 1, 2 \ldots \sqrt{n}\} \qquad (6)$$

When the correct value for r is found, $(L^2 - n)$ will be a complete square.

Both p and q can now be found

$$p, q = (m + r) \pm \sqrt{(m + r)^2 - n} \qquad (7)$$

Where $m = \text{Int}(\sqrt{n})$

The maximum number of trials needed can be estimated in terms of p and q by rewriting eq. (6) as

$$n + j^2 = (m + r)^2$$

And $$r = \sqrt{n(1 + \frac{j}{n})^2 - m}$$

We may assume $\frac{j}{n} << 1$, hence we can use binomial series expansion to expand the square root so that

$$r \approx (\sqrt{n} - \text{Int}(n)) + \frac{L^2}{2\sqrt{n}}$$

Since $L = \frac{(p - q)}{2}$ and $n = p \cdot q$, we have

$$r \approx \frac{(p - q)^2}{8\sqrt{n}} \qquad (8)$$

For example, if $(p - q) \approx .001\ p$, then $j \approx .0000001\ p$. If p is a 100 digit prime number and $(p - q)$ is a 40 digit number, then $r = 1$. Which means the solution can be found in one trial in this case. In conclusion, our method is quite efficient for prime numbers p and q which differ by less then 10% in value.

To see how efficient the method is, we chose a prime number $p = 99991$ and a second variable prime number q. We calculated $n = p \cdot q$ and used n to factor the original prime numbers p and q using our method. The number of trial integers (r) needed to factor n was noted. A plot of the number of trials as a function of the difference $(p - q)$ is plotted below. The plot shows very clearly the efficiency of the method when $(p - q)$ is relatively small. In many cases requiring a single trial number. The dependence on $(p - q)$ is essentially quadratic. We have found the same behavior with huge integers of orders more than 128 digits.

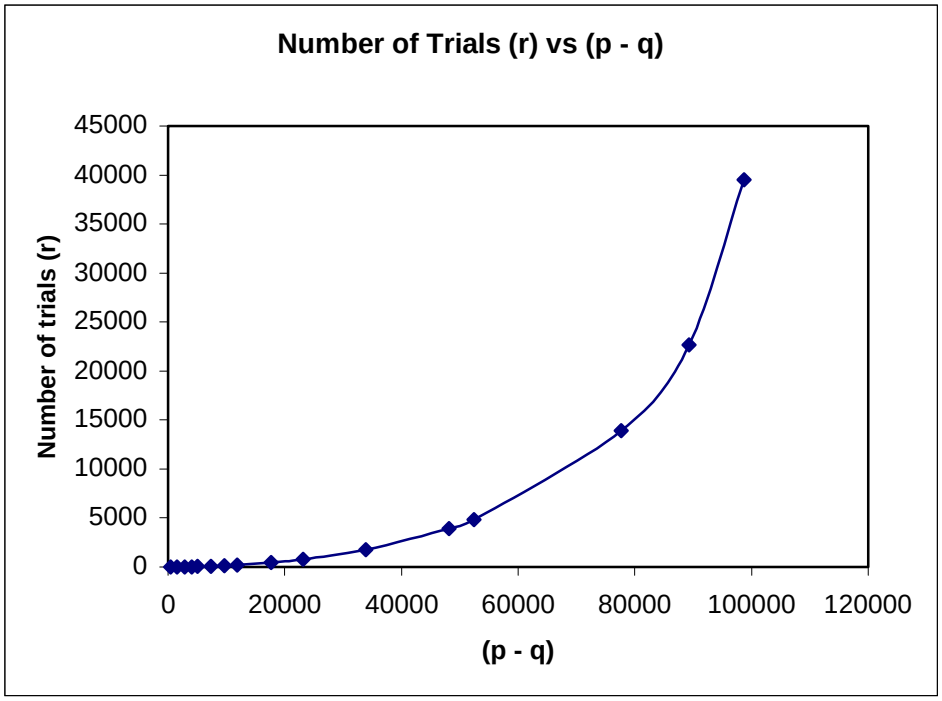

Figure (1a) The dependence of the number of trials (r) vs the difference (p - q)

In figure (1b) we plot the results of running the computer program on the supercomputer. We plot time taken by the computer as a function of (p - q). Notice that the behavior is again quadratic near the origin and becomes linear for large (p - q) .

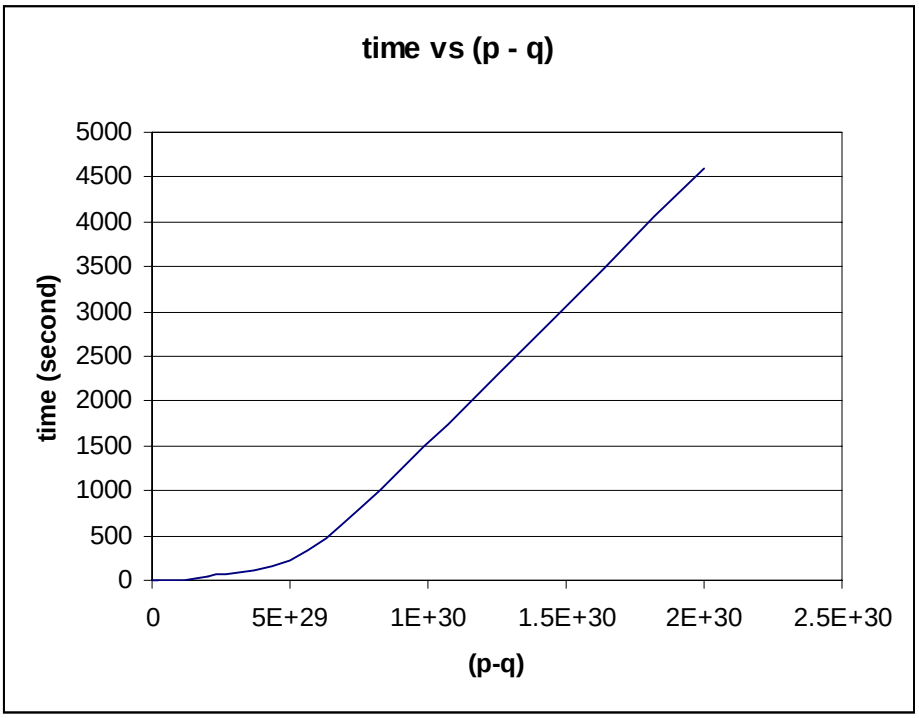

Figure (1b) Supercomputer factoring time vs (p - q)

9. Future Considerations

If given a chance to continue our project, several step could be taken to further enhance our research.

1) A plausible way to speed up our factorization method would be to choose random points for r, and scan the area at which j approaches a complete square

   Choose a random value r from a set of $Z \in \{0, 1, 2 \ldots \sqrt{n}\}$

   $$j_i = \sqrt{(m + r)^2 - n}$$

   Points at which j approaches an integer, scan that area

$$j = \sqrt{(m + r)^2 - n}\ ; \qquad r = \{j_i\text{-}100, j_i\text{-}99 \ldots j_i\text{+}99, j_i\text{+}100\}$$

2) We would like to improve on the method so that it will work on more general cases of prime numbers p and q. We noticed that the efficiency of the method can be understood graphically as follows,

$$n + j^2 = L^2$$

$$\text{where } L = \frac{p - q}{2} \quad \text{and} \quad L = \frac{p + q}{2}$$

Let us replace L7 by x and j by y and divide through by n, we get

$$\frac{x^2}{n} - \frac{y^2}{n} = 1$$

This is the equation of a hyperbola as shown in the figure (Fig. 2)

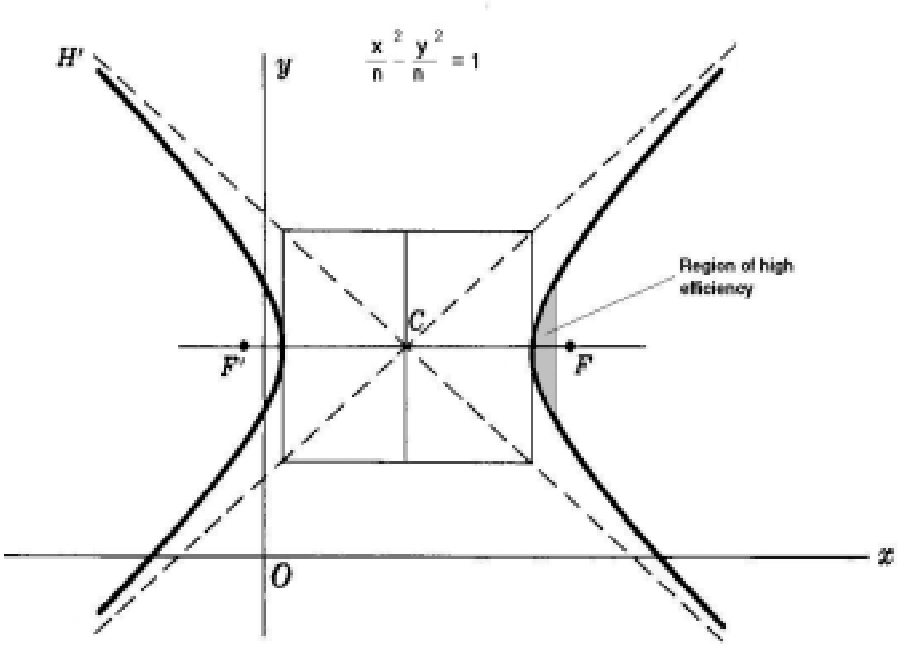

Figure 2

Notice that the slope of the curve represents the efficiency which is very high near the x-axis when y is small. As y becomes large, the slope approaches unity which means that the efficiency of the method becomes relatively small. If we could find a way to make this region of high efficiency extend over a large region or make a position variable, then we can slide the region of high efficiency at will. We hope to find a way to take advantage of this observation.

3) Compare our factoring method against other methods for factoring n such as the p – 1, elliptic curve, and quadratic sieve method (not mentioned in this report, yet currently considered the most successful algorithm).

10. Conclusion

In conclusion, we created an original factoring method which can be used to decrypt an RSA encryption by factoring the unknown prime numbers p and q from the known public key n. Perhaps our research will caution ciphers to provide a safe distance between p and q when choosing supposedly "safe" primes.
We hope to extend our method to more general cases of prime numbers. As seen in our research, utilizing a super computer greatly increased our method's efficiency. Our program overcame the language limitations imposed on the size of integers used and can handle numbers of any size. In addition to factoring n, the program also provides some useful functions to encode and decode information using the RSA method.

This project has been an excellent learning experience and we would like to thank the teachers and supervisors who helped us overcome so many problems in this challenge.

11. References

1- Douglas R. Stinson. Cryptography, Theory and Practice , CRC Press LLC 1995.
2- Simon Singh. The Code Book. Anchor Books 1999.
3- William H. Beyer. CRC Standard Mathematical Tables. CRC Press 1979.
4- Stephen Prata. C++ Primer Plus. Waite Group Press 1995.
5- Beyer, William H., CRC Standard Mathematical Tables. CRC Press 1979.
6- Prata, Stephen, C++ Primer Plus. Waite Group Press 1995.
7- Cohen, Fred, "A Short History of Cryptography". 1995.
<http://www.all.net/books/ip/chap2-1.htm>
8- "Nineteenth Century: Statistics".
<http://www.math.okstate.edu/~wrightd/crypt/cryptintro/node9.htm>

Appendix A - Catalog of classes and structures

Key

#include "key.h"

The **Key** structure holds a set of keys used in encryption and decryption

```
struct Key {
        mpz_t key1;
        mpz_t key2;
        mpz_t key3;
        bool initialized;
};
```

KeyGenerator

#include "key.h"

The **KeyGenerator** class supports initialization and destruction of a **Key** structure.

Construction — Public Members

KeyGenerator(void) Constructs a **KeyGenerator** object.

KeyGenerator(int) Constructs a **KeyGenerator** object (algorithm specifies which form of encryption to use when initializing Key structures. Potential values are KeyGenerator::UNINITIALIZED, KeyGenerator::RSA)

Other Functions — Public Members

initialize(unsigned) Creates the random seed

generateKey(unsigned) Returns a **Key** structure with random values (if max_size isn't specified, the default is 1)

generateKey(const char*, const char *, const char *) Returns a **Key** structure with specified values

destroyKey(Key) Releases memory for a **Key** structure

getInstance(int) Returns an instance of **KeyGenerator**

Cipher

#include "cipher.h"

The **Cipher** class is used to encrypt and decrypt data

Construction - Public Members

Cipher(void) Constructs a **Cipher** object

Other Functions – Public Members

getState(void) Returns the current state (encrypt or decrypt)
blockSize(void) Returns the size of a **Cipher** block
initEncrypt(Key key) Initializes this cipher for encryption
initDecrypt(Key key) Initializes this cipher for decryption
convert(mpz_t&, const char*) Converts a string to a number (base 10)
convert(mpz_t&, const char*) Converts a number to a string (base 255)
crypt(mpz_t& rop, mpz_t nData) Encrypts or Decrypts a data block and places it in rop (returns 1 on success, 0 on failure)

Appendix B - Program

```
// eve.cpp (dShakir)                    Last Update: March 29, 2001
// Demonstrates our proposed method for factoring N (see report)

#include <time.h>
#include <fstream.h>
#include "key.h"                                        // nextprime(mpz_t, const char*)

const char* N = "11000000077";

ostream& operator<<(ostream&, mpz_t);                   // create the ability to output mpzs

void main(int argc, char* argv[])
{
        ofstream output("out.dat", ios::app);
        mpz_t n, m, j, l, p, q;

        // initialize mpz
        mpz_init(j);
        mpz_init(m);
        mpz_init(l);

        mpz_init_set_str(n, N, 10);

        output << "N:" << mpz_get_str(0, 10, n) << endl;

        mpz_sqrt(m, n);                                 // m = int(n)

        mpz_init_set(j, m);                             // j = m

        mpz_sub_ui(j, j, 1);                            // start j = 0 in while loop

        clock_t start, finish;
        start = clock();

        do {
                mpz_add_ui(j, j, 1);                    // j = j + 1
                mpz_mul(l, j, j);                       // l = j ^ 2
                mpz_sub(l, l, n);                       // l = l - n

        } while(!mpz_perfect_square_p(l));

        finish = clock();

        // p or q = (j + m) +/- sqrt((j + m) ^ 2 - n)

        mpz_sqrt(l, l);                                 // l = sqrt((j + m) ^ 2 - n)

        mpz_init_set(p, j);                             // p = (j + m)
        mpz_init_set(q, j);                             // q = (j + m)

        mpz_sub(p, p, l);                               // p = p - l
        mpz_add(q, q, l);                               // q = q - l
```

```
        mpz_sub(j, j, m);                                   // j - m = total iterations

        output << "time duration (sec): " << (double)(finish - start) / CLOCKS_PER_SEC << endl;

        output << "j: " << mpz_get_str(0, 10, j) << endl;
        output << "p: " << mpz_get_str(0, 10, p) << endl;
        output << "q: " << mpz_get_str(0, 10, q) << endl;

        // release mpz
        mpz_clear(n);
        mpz_clear(l);
        mpz_clear(m);
        mpz_clear(j);
        mpz_clear(p);
        mpz_clear(q);

        output.close();
}

ostream& operator<<(ostream& os, mpz_t op)
{
      os << mpz_get_str(0, 10, op);
      return os;
}

/*
////////// Program Output (out.dat) ////////////////////

N:11000000077
time duration (sec): 470.64
j: 499895129
p: 11
q: 1000000007
*/
```

```
// bob.cpp (dShakir)                Last Update: March 29, 2001
// Demonstrates the Cipher and KeyGenerator classes
```

```
#include <iostream.h>
#include <stdio.h>
#include "cipher.h"

const char* MESSAGE = "helloalice";

ostream& operator<<(ostream&, mpz_t);                    // create the ability to output mpzs

void main(void)
{
        Cipher cipher;
        KeyGenerator generator(KeyGenerator::RSA);
        mpz_t c, n, result;

        generator.initialize(unsigned(time(NULL)));
        Key keys = generator.generateKey(9);             // generate a set of random keys

        cout << "p - " << keys.key1 << endl;
        cout << "q - " << keys.key2 << endl;
        cout << "e - " << keys.key3 << endl << endl;

        mpz_init(result);
        mpz_init(n);

        mpz_mul(n, keys.key1, keys.key2);                // find n (n = p x q)
        cout << "N - " << n << endl << endl;

        cout << "Message(text) - " << MESSAGE << endl;
        cipher.convert(c, MESSAGE);                      // convert message to ASCII
        cout << "Message(ascii) - " << c << endl << endl;

        cipher.initEncrypt(keys);                        // initialize encryption
        bool out = cipher.crypt(result, c);              // encrypt message
        cout << "Encrypted Message - " << result << endl;

        cipher.initDecrypt(keys);
        out = cipher.crypt(result, result);
        cout << "Decrypted Message - " << result << endl;

        cout << "Decrypted Message(text) - " << cipher.convert(result) << endl;

        generator.distroyKey(keys);

        mpz_clear(c);
        mpz_clear(result);
}
ostream& operator<<(ostream& os, mpz_t op)
{
        os << mpz_get_str(0, 10, op);
        return os;
}

/*
```

////////// Program Output //

p - 7767159204022160317694640711879129210331162089377414496387575659854494861762
3160808199
q - 7178045237321459295219473181898271558811008640356741998257631819809970933871
6179199447
e - 1327340545478027228044859866734844310308616980089034465285746599771806441668
01860045523

N - 5575302013194880463446083054725055770159080465377540402310639550228338898861
4734171849262780988549895301411564787678365835791051874250677974880408690701718 8
5803705333865953

Message(text) - helloalice
Message(ascii) - 77753349573733071041774

Encrypted Message - 4580811303206918120964594488173184144915144654025825861832585
53085959661612616215560937530999828938737271270374993502391064337517953618105581
07265332351487623533405487609 74
Decrypted Message - 77753349573733071041774
Decrypted Message(text) - helloalice

*/

// Key.h (dShakir) Last Updated: March 29, 2001
// Key - holds a set of keys which can be used in several forms of encryption including RSA/DES
// KeyGenerator - initializes and destroys keys. It may gives random values to a set of keys

```
#ifndef _KEY_H
#define _KEY_H

#include <gmp.h>
#include <time.h>
#include <stdlib.h>
#include <limits.h>

void mpz_nextprime(mpz_t, mpz_t);                       // find the next occuring prime number
void mpz_nextprime(mpz_t, const char*);

struct Key
{
        mpz_t key1;
        mpz_t key2;
        mpz_t key3;
        bool initialized;
};

class KeyGenerator
{
public:
        // Construction
        KeyGenerator(void);
        KeyGenerator(int);

        // Operations
        void initialize(unsigned);                                      // initialize random seed
        Key generateKey(unsigned =1);     // create a set of random keys of specified size
(max_size~size*8)
        Key generateKey(const char*, const char*, const char*);         // assign values for a set of keys
        void destroyKey(Key);                                           // release memory for a set of keys
        static KeyGenerator getInstance(int);                           // create an instance of
KeyGenerator

//      static bool isPrime(int);                                       // obsolete after gmp.h
//      unsigned generatePrime(mpz_t);                                  // obsolete after gmp.h

        // Implementation
        enum key_algorithm {
        KEY_START = 0x00,
                UNINITIALIZED = 0x00,
                RSA = 0x01,
                KEY_END = 0x01
        };

private:
        int m_nAlgorithm;                                               // specifies which algorithm to use
};

////////////////////////////////////////////////////////////////////////////////////////////////////
// KeyGenerator out-of-line functions

KeyGenerator::KeyGenerator(void)
{
```

```
	m_nAlgorithm = UNINITIALIZED;
}

KeyGenerator::KeyGenerator(int nAlgorithm)
{
	if(nAlgorithm < KEY_START || nAlgorithm > KEY_END)
		throw("NoSuchAlgorithmException");

	m_nAlgorithm = nAlgorithm;
}

// obsolete after gmp.h
// bool KeyGenerator::isPrime(int nPrime)
// {
//
//	const unsigned iterations = 3;
//
//	for(unsigned i = 0; i < iterations; i++)
//	{
//	  if(fnModule(rand()%(nPrime-1), nPrime-1, nPrime)!=1)
//	    return 0x0;
//	}
//
//	return 0x1;
//}
// obsolete after gmp.h
//unsigned KeyGenerator::generatePrime(unsigned nPrime)
//{
//	throw("PrimeGenerationException");
//
//	while(!KeyGenerator::isPrime(++nPrime)) ;
//
//	return nPrime;
//}

void KeyGenerator::initialize(unsigned seed)
{
	srand(seed);
}

Key KeyGenerator::generateKey(const char* p, const char* q, const char* e)
{
	Key key;
	mpz_init_set_str(key.key1, p, 10);
	mpz_init_set_str(key.key2, q, 10);
	mpz_init_set_str(key.key3, e, 10);
	key.initialized = 1;
	return key;
}

Key KeyGenerator::generateKey(unsigned max_size)		// max_size ~ max_size * 8
{
	if(m_nAlgorithm < KEY_START&&m_nAlgorithm > KEY_END)
		throw("NoSuchAlgorithmException");
	else if(m_nAlgorithm & UNINITIALIZED)
```

```
		throw("AlgorithmUninitializedException");

	Key key = {0, 0, 0, 0};

	if(m_nAlgorithm & RSA)
	{
		mpz_t tbprime;
		mpz_init(tbprime);

		// initialize key
		mpz_init(key.key1);
		mpz_init(key.key2);
		mpz_init(key.key3);

		// create random key values (primes)
		mpz_random(tbprime, max_size);
		mpz_nextprime(key.key1, tbprime);
		mpz_random(tbprime, max_size);
		mpz_nextprime(key.key2, tbprime);
		mpz_random(tbprime, max_size);
		mpz_nextprime(key.key3, tbprime);

		if(!mpz_cmp(key.key1, key.key2))			// p and q cannot be equal
		key = generateKey(max_size);

		pz_t product, op1, op2, result;

		mpz_init(product);
		mpz_init(op1);
		mpz_init(op2);
		mpz_init(result);

		// prove e is relatively prime to p and q if gcd(e, (p - 1)(q - 1)) = 1

		while(1)
		{
			mpz_sub_ui(op1, key.key1, 1);
			mpz_sub_ui(op2, key.key2, 1);
			mpz_mul(product, op1, op2);
			mpz_gcd(result, key.key3, product);
			if(!mpz_cmp_ui(result, 1))
				break;
			key = generateKey(max_size);
		}

		key.initialized = 0x01;

		mpz_clear(tbprime);
	}

	return key;
}

void KeyGenerator::destroyKey(Key key)
{
```

```
	mpz_clear(key.key1);
	mpz_clear(key.key2);
	mpz_clear(key.key3);
}

KeyGenerator KeyGenerator::getInstance(int nAlgorithm)
{
	if(nAlgorithm < KEY_START || nAlgorithm > KEY_END)
		throw("NoSuchAlgorithmException");

	return KeyGenerator(nAlgorithm);
}

void mpz_nextprime(mpz_t rop, const char* str)
{
	const int reps = 10;

	mpz_t op;
	mpz_init_set_str(op, str, 10);

	while(1)
	{
	if(mpz_probab_prime_p(op, reps))
		{
			mpz_set(rop, op);
			return;
		}
		mpz_add_ui(op, op, 1);
	}
}

void mpz_nextprime(mpz_t rop, mpz_t op)
{
	const int reps = 10;

	while(1)
	{
	if(mpz_probab_prime_p(op, reps))
		{
			mpz_set(rop, op);
			return;
		}
		mpz_add_ui(op, op, 1);
    }
}

#endif
// Cipher.h (dShakir)                    Last Updated: March 29, 2001
// Cipher - using a set of keys, can encrypt and decrypt a given value, using the RSA
// algorithm (see report)

#ifndef _CIPHER_H
#define _CIPHER_H

#include "project.h"
```

```
#include "key.h"
#include <gmp.h>
#include <stddef.h>

class Cipher
{
public:
	// Constructors
	Cipher(void);

	// Attributes
	int getState(void);				// returns the current state
	int blockSize(void);				// returns the size of a cipher block
	void initEncrypt(Key);				// initializes this cipher for encryption
	void initDecrypt(Key);				// initializes this cipher for decryption

	void convert(mpz_t&, const char*);	// convert a string to a number (base 10)
	char* convert(mpz_t);				// convert a number to a string (base 255)

	// Operations
	bool crypt(mpz_t&, mpz_t);			// encrypts or decrypts the specified data

	enum cipher_state{
		CIPHER_START = 0x00,
		UNINITIALIZED = 0x00,
		ENCRYPT = 0x01,
		DECRYPT = 0x02,
		CIPHER_END = 0x02
	};

private:
	// Implementation
	Key m_pKeys;
	int m_nState;
};

////////////////////////////////////////////////////////////////////////////////////////////////////
// Cipher inline functions

inline int Cipher::getState(void)
	{ return m_nState; }
inline int Cipher::blockSize(void)
	{ return sizeof(int); }


////////////////////////////////////////////////////////////////////////////////////////////////////
// Cipher out-of-line functions

Cipher::Cipher(void)
{
	m_nState = UNINITIALIZED;
}

void Cipher::initEncrypt(Key rKeys)
{
```

```
	m_pKeys = rKeys;
	m_nState = ENCRYPT;
}

void Cipher::initDecrypt(Key rKeys)
{
	m_pKeys = rKeys;
	m_nState = DECRYPT;
}

char* Cipher::convert(mpz_t op)
{
	return mpz_get_str(0, 255, op);
}

void Cipher::convert(mpz_t& rop, const char* str)
{
	mpz_t temp;
	mpz_init_set_str(temp, str, 255);
	mpz_init_set(rop, temp);
}

bool Cipher::crypt(mpz_t& nResult, mpz_t nData)
{
	mpz_t product;
	mpz_init(product);
	mpz_mul(product, m_pKeys.key1, m_pKeys.key2);			// N = p * q

	if(m_nState < CIPHER_START || m_nState > CIPHER_END)
		throw "StateOutOfBoundsException";

	if(mpz_cmp(nData, product) > 0)					// M must be < N
		return 0x00;

	if(m_nState & ENCRYPT)						// C = M^e mod N
	{
		mpz_powm(nResult, nData, m_pKeys.key3, product);
		return 0x01;
	}
	else if(m_nState & DECRYPT)					// M = C^d mod N
	{								// d is found using the
Euclidean
		mpz_t euclidean, op1, op2, product2;			// algorithm, which
essentially
		mpz_init(euclidean);					// finds the least common
multiplier
		mpz_init(op1);						// between p - 1 and q - 1
		mpz_init(op2);						// d = lcm(p - 1, q - 1)
		mpz_init(product2);
		mpz_sub_ui(op1, m_pKeys.key1, 1);
		mpz_sub_ui(op2, m_pKeys.key2, 1);
		mpz_mul(product2, op1, op2);
		mpz_euclidean(euclidean, product2, m_pKeys.key3);
		mpz_powm(nResult, nData, euclidean, product);
		return 0x01;
```

```
	}
	else
	{
		return 0x00;
	}
}

#endif
```

```
// Project.h (dShakir)		Last Updated: March 29, 2001
// Most functions in this header became obsolete after we found gmp.h (with the mpz data types)

#ifndef _PROJECT_H
#define _PROJECT_H

#include <gmp.h>
#include <math.h>

double fnModule(int, int, int);
int fnEuclidean(int, int);
void mpz_euclidean(mpz_t, mpz_t, mpz_t);

int fnEncryptInteger(int, int, int, int);
int fnDecryptInteger(int, int, int, int);
```

```
// Encrypts an integer M using the RSA algorithm
int fnEncryptInteger(int p, int q, int e, int m)                    // C = M^e mod N
{                                                                   // N = p * q
        return int(fnModule(m, e, p * q));
}

// Decrypts an integer C using the RSA algorithm
int fnDecryptInteger(int p, int q, int e, int c)                    // M = C^d mod N
{                                                                   // N = p * q
        int euclidean;                                              // (d is found using Euclid's
algorithm)

        if(!(euclidean = fnEuclidean((p-1)*(q-1), e)))
        {
                return -1;
        }

        return int(fnModule(c, euclidean, p * q));
}

// This function returns the result of x^y mod z
// Its really ingenious how this function operates.
// It utilizes the fact that you may split up y in modular math. For example,
// M^10 mod z = (M^2 mod z * M^3 mod z * M^5 mod z) mod z (2 + 3 + 5 = 10)
// What this function does is use recursion to divide y by 2 again and again,
// until y is small enough to accurately handle the math.
// (x^y mod z (x^y/2 mod z * (x^(y/3) mod z(x^y/n mod z)))) mod z
// where y/n is smaller then 10^16, the maximum before accuracy is lost.

double fnModule(int x, int y, int z)
{
        double t = 1.0, d;

        if(fmod(y, 2) != 0)                                         // checks if y is odd
        {
                t = fmod(x, z);
                y = y - 1;
        }

        if(y * log(x) > 16)                                         // log x^y > log 10^16
                d = fnModule(x, y / 2, z);
        else
                d = fmod(pow(x, y / 2), z);

        return fmod(fmod(d, z) * fmod(d, z) * t, z);                // ((d mod z)^2 * t) mod z
}

// Returns the least common multiplier (lcm) of n and e

void mpz_euclidean(mpz_t rop, mpz_t n, mpz_t e)
{
        mpz_t temp, n0, e0;
        mpz_t t0;
        mpz_t t1;
        mpz_t negone;
```

```
	mpz_init(temp);
	mpz_init_set(n0, n);
	mpz_init_set(e0, e);

	mpz_init_set_str(t0, "0", 10);
	mpz_init_set_str(t1, "1", 10);
	mpz_init_set_str(negone, “-1”, 10);

	mpz_t q, r;

	mpz_init(q);
	mpz_init(r);

	mpz_tdiv_q(q, n0, e0);                          // q = int(n0 / e0)

	mpz_mul(r, q, e0);                              // r = n0 - q * e0
	mpz_sub(r, n0, r);

	while(mpz_cmp_ui(r, 0))
	{
		// temp = t0 - q * t1;
		mpz_mul(temp, q, t1);
		mpz_sub(temp, t0, temp);

		if(mpz_cmp_ui(temp, 0) >= 0)
			mpz_mod(temp, temp, n);         // temp = (temp % n)
		if(mpz_cmp_ui(temp, 0) < 0)
		{
			mpz_mul(temp, temp, negone);    // temp = n - ((-1 * temp) % n)
			mpz_mod(temp, temp, n);
			mpz_sub(temp, n, temp);
		}

		mpz_set(t0, t1);                        // t0 = t1
		mpz_set(t1, temp);                      // t1 = temp
		mpz_set(n0, e0);                        // n0 = e0
		mpz_set(e0, r);                         // e0 = r
		mpz_tdiv_q(q, n0, e0);                  // q = int(n0 / e0)
		mpz_mul(r, q, e0);                      // r = n0 - q * e0
		mpz_sub(r, n0, r);
	}

	if(mpz_cmp_ui(e0, 1))
	{
		mpz_set_ui(rop, 0);                     // no result found
	}
	else
	{
		mpz_mod(rop, t1, n);                    // return int(t1 % n);
	}
}

#endif
```