**PES University, Bangalore**

(Established under Karnataka Act No. 16 of 2013)

**Department of Computer Science & Engineering**

**SESSION : JAN- MAY,2019-20**

**DAA PROJECT**

**CRYPTOGRAPHIC LIBRARY**

|  |  |
| --- | --- |
| Name: MONISH REDDY BS | Name: GOUTHAM R |
| SRN: PES1201701666 | SRN: PES1201701025 |
| Section: H | Section: H |

**Cryptographic Libraries**

**RSA-Algorithm:**

This report describes in general the design and working of the Rivest-Shamir-Adleman Algorithm commonly known by the name RSA algorithm. It is a public key cryptosystem that makes use of a pair of keys, namely the public and private key.

It is an asymmetric cryptographic algorithm. Asymmetric actually means that it works on two different keys i.e. Public Key and Private Key. As the name describes that the Public Key is given to everyone and Private Key is kept private. The RSA cryptosystem, named after its invertors R. Rivest, A. Shamir, and L. Adleman, is the most widely used public-key cryptosystem. It may be used to provide both secrecy and digital signatures and its security is based on the intractability of the integer factorization problem.

The existence of cryptography from this early a time, proves that there was a need for data security from a very long time.

It is based on a very simple number-theoretical idea, and yet it has been able to resist all cryptanalytic attacks. The idea is a clever use of the fact that, while it is easy to multiply two large primes, it is extremely difficult to factorize their product. Thus, the product can be publicized and used as the encryption key. The primes themselves cannot be recovered from the product and are used for decryption.

**Working of RSA-Algorithm:**

Public and Private Keys:

· Take two large prime numbers p and q(of the order of a few hundred bits).

· Compute their product n. Also compute the Euler function phi(n) = (p 1)(q 1)

· Choose a large random number d (d > 1) such that (d; phi(n)) = 1 (i.e, d and phi(n) are relatively prime).

· Compute the number e, 1 < e < phi(n) such that ed = 1(mod phi(n)) (i.e, ed 1 is divisible by (p 1)(q 1)).

An introduction to some terminology will be appropriate at this juncture

Terminology used are:

n : Modulus or Key

d : Private or Decryption Exponent

e : Public or Encryption Exponent

(n; e) : Public Key

(n; d) : Private Key

p; q; phi(n); d : form the Secret Trapdoor (p and q may be kept with the private key or destroyed.

**Important Features of RSA:**

While studying the working of the RSA system, we need to note the following: Encryption and authentication takes place without sharing of private keys: each person uses only other people's public keys and his/her own private key. Anyone can send an encrypted message or verify a signed message, using only public keys, but only someone in possession of correct private keys can decrypt or sign a message. Modular Exponentiation: The computation of (a^r ; mod n) is done using a method that is faster than repeatedly multiplying a by itself. We use squaring. After each squaring, reduction modulo n is done. So we never encounter numbers greater than n^2 . Thus (a^r ; mod n) can be computed in O(log r) time.

**Example:**

The working of the RSA system will become clearer with the help of an example:

p = 5, q = 11, n = 55, phi(n) = (p 1)(q 1) = 40, e = 7, d = 23.

**To calculate (8^7 ; mod 55):**

j (8^2^j ; mod 55)

0 8

1 9

2 26

7 = 1112 (8^7; mod 55) = ((26(9.8))mod 55) = 2

This contrived example proves that public-key cryptosystems never work for small plain-text spaces. A cryptanalyst can construct a complete decryption table by encrypting all possible plain-texts and rearranging them in alphabetic order.

**Tests for Primality:**

An efficient algorithm for the problem PRIMALITY(n) is necessary for RSA cryptosystem design. It is not known whether the problem is in P. However, stochastic algorithms with a low probability of failure are quite acceptable. Such a stochastic algorithm works in most cases as follows. Consider a compositeness test C(m). If an integer m passes the test, it is definitely composite. If m fails the test, m may be prime. The likelihood of m being prime increases with the number of compositeness tests it fails.

Code to test whether the selected number is a prime or not:

int prime(long int pr){

int i;

j = sqrt(pr);

for(i = 2; i <= j; i++){

if(pr % i == 0)

return 0;

}

return 1;

}

**Implementation Code for RSA-Algorithm:**

#include<stdio.h>

#include<stdlib.h>

#include<math.h>

#include<string.h>

int x, y, n, t, i, flag;

long int e[50], d[50], temp[50], j, m[50], en[50];

char msg[100];

int prime(long int);

void encryption\_key();

long int cd(long int);

void encrypt();

void decrypt();

int main()

{

printf(" Enter 1st prime number : ");

scanf("%d", &x);

flag = prime(x);

if(flag == 0)

{

printf("\nINVALID INPUT\n");

exit(0);

}

printf("Enter 2nd prime number : ");

scanf("%d", &y);

flag = prime(y);

if(flag == 0 || x == y)

{

printf("\nINVALID INPUT\n");

exit(0);

}

printf("Enter message to encrypt : ");

scanf("%s",msg);

for(i = 0; msg[i] != NULL; i++)

m[i] = msg[i];

n = x \* y;

t = (x-1) \* (y-1);

encryption\_key();

printf("Possible values of 'e' and 'd' are : \n");

for(i = 0; i < j-1; i++)

printf("\n[+] %ld\t%ld", e[i], d[i]);

encrypt();

decrypt();

return 0;

}

int prime(long int pr)

{

int i;

j = sqrt(pr);

for(i = 2; i <= j; i++)

{

if(pr % i == 0)

return 0;

}

return 1;

}

//function to generate encryption key

void encryption\_key()

{

int k;

k = 0;

for(i = 2; i < t; i++)

{

if(t % i == 0)

continue;

flag = prime(i);

if(flag == 1 && i != x && i != y)

{

e[k] = i;

flag = cd(e[k]);

if(flag > 0)

{

d[k] = flag;

k++;

}

if(k == 99)

break;

}

}

}

long int cd(long int a)

{

long int k = 1;

while(1)

{

k = k + t;

if(k % a == 0)

return(k / a);

}

}

//function to encrypt the message

void encrypt()

{

long int pt, ct, key = e[0], k, len;

i = 0;

len = strlen(msg);

while(i != len)

{

pt = m[i];

pt = pt - 96;

k = 1;

for(j = 0; j < key; j++)

{

k = k \* pt;

k = k % n;

}

temp[i] = k;

ct = k + 96;

en[i] = ct;

i++;

}

en[i] = -1;

printf("Encrypted message is : ");

for(i = 0; en[i] != -1; i++)

printf("%c", en[i]);

}

//function to decrypt the message

void decrypt()

{

long int pt, ct, key = d[0], k;

i = 0;

while(en[i] != -1)

{

ct = temp[i];

k = 1;

for(j = 0; j < key; j++)

{

k = k \* ct;

k = k % n;

}

pt = k + 96;

m[i] = pt;

i++;

}

m[i] = -1;

printf("Decypted message : ");

for(i = 0; m[i] != -1; i++)

printf("%c", m[i]);

printf("\n");

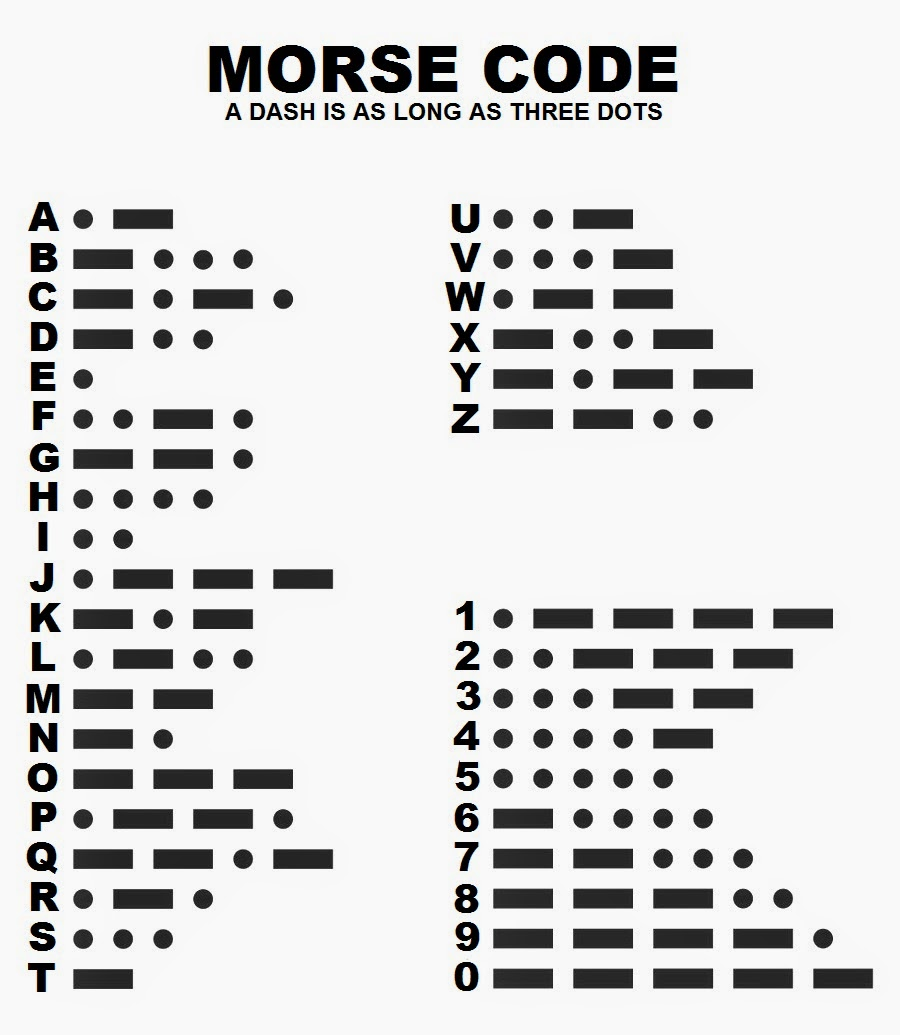
}

**Morse Code Algorithm:**

**Morse code** is a method of transmitting text information as a series of on-off tones, lights, or clicks that can be directly understood by a skilled listener or observer without special equipment. It is named for Samuel F. B. Morse, an inventor of the telegraph.

The algorithm is very simple. Every character in the English language is substituted by a series of ‘dots’ and ‘dashes’ or sometimes just singular ‘dot’ or ‘dash’ and vice versa.

Every text string is converted into the series of dots and dashes. For this every character is converted into its Morse code and appended in encoded message. Here we have copied space as it is. And we have not considered numbers, only alphabets. **Morse code** is a character encoding scheme used in telecommunication that encodes text characters as standardized sequences of two different signal durations called dots and dashes or dits and dahs.



The above image displays the corresponding morse codes for each English alphabets. Using these codes we can encrypt the message which to be sent and can be easily decrypted using these codes.

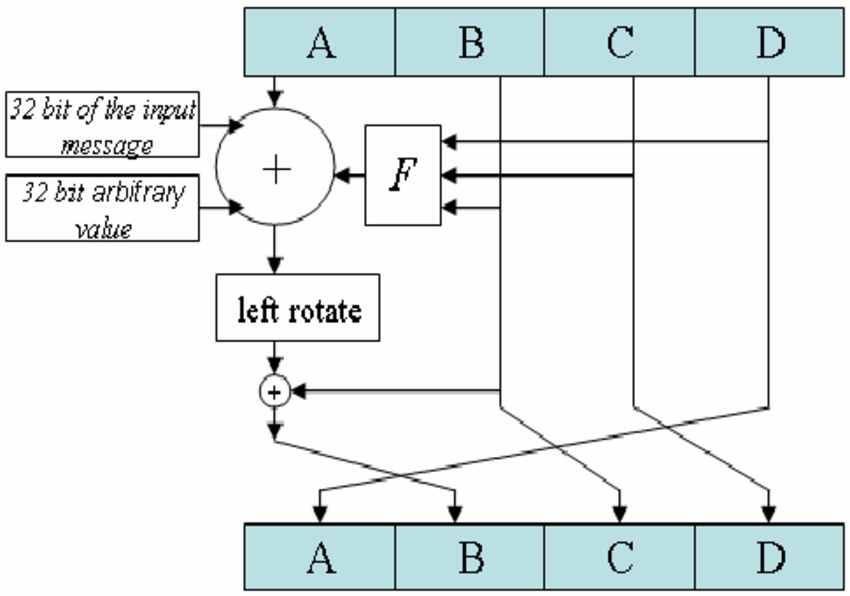
**MD5 ALGORITHM**

The MD5 message-digest algorithm is a widely used hash function producing a 128-bit hash value. Although MD5 was initially designed to be used as a cryptographic hash function, it has been found to suffer from extensive vulnerabilities. It can still be used as a checksum to verify data integrity, but only against unintentional corruption. It remains suitable for other non-cryptographic purposes, for example for determining the partition for a particular key in a partitioned database.

MD5 processes a variable-length message into a fixed-length output of 128 bits. The input message is broken up into chunks of 512-bit blocks (sixteen 32-bit words); the message is [padded](https://en.wikipedia.org/wiki/Padding_(cryptography)) so that its length is divisible by 512. The padding works as follows: first a single bit, 1, is appended to the end of the message. This is followed by as many zeros as are required to bring the length of the message up to 64 bits fewer than a multiple of 512. The remaining bits are filled up with 64 bits representing the length of the original message, modulo 264.

The main MD5 algorithm operates on a 128-bit state, divided into four 32-bit words, denoted *A*, *B*, *C*, and *D*. These are initialized to certain fixed constants. The main algorithm then uses each 512-bit message block in turn to modify the state. The processing of a message block consists of four similar stages, termed *rounds*; each round is composed of 16 similar operations based on a non-linear function *F*, modular addition, and left rotation. Figure 1 illustrates one operation within a round. There are four possible functions; a different one is used in each round:

*F ( B , C , D ) = ( B ∧ C ) ∨ ( ¬ B ∧ D ) G ( B , C , D ) = ( B ∧ D ) ∨ ( C ∧ ¬ D ) H ( B , C , D ) = B ⊕ C ⊕ D I ( B , C , D ) = C ⊕ ( B ∨ ¬ D )*

**Code:**

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <math.h>

typedef union uwb {

unsigned w;

unsigned char b[4];

} MD5union;

typedef unsigned DigestArray[4];

unsigned func0( unsigned abcd[] ){

return ( abcd[1] & abcd[2]) | (~abcd[1] & abcd[3]);}

unsigned func1( unsigned abcd[] ){

return ( abcd[3] & abcd[1]) | (~abcd[3] & abcd[2]);}

unsigned func2( unsigned abcd[] ){

return abcd[1] ^ abcd[2] ^ abcd[3];}

unsigned func3( unsigned abcd[] ){

return abcd[2] ^ (abcd[1] |~ abcd[3]);}

typedef unsigned (\*DgstFctn)(unsigned a[]);

/\*Use binary integer part of the sines of integers (Radians) as constants\*/

/\*or\*/

/\* Hardcode the below table but i want generic code that why coded it

using calcuate table function

k[ 0.. 3] := { 0xd76aa478, 0xe8c7b756, 0x242070db, 0xc1bdceee }

k[ 4.. 7] := { 0xf57c0faf, 0x4787c62a, 0xa8304613, 0xfd469501 }

k[ 8..11] := { 0x698098d8, 0x8b44f7af, 0xffff5bb1, 0x895cd7be }

k[12..15] := { 0x6b901122, 0xfd987193, 0xa679438e, 0x49b40821 }

k[16..19] := { 0xf61e2562, 0xc040b340, 0x265e5a51, 0xe9b6c7aa }

k[20..23] := { 0xd62f105d, 0x02441453, 0xd8a1e681, 0xe7d3fbc8 }

k[24..27] := { 0x21e1cde6, 0xc33707d6, 0xf4d50d87, 0x455a14ed }

k[28..31] := { 0xa9e3e905, 0xfcefa3f8, 0x676f02d9, 0x8d2a4c8a }

k[32..35] := { 0xfffa3942, 0x8771f681, 0x6d9d6122, 0xfde5380c }

k[36..39] := { 0xa4beea44, 0x4bdecfa9, 0xf6bb4b60, 0xbebfbc70 }

k[40..43] := { 0x289b7ec6, 0xeaa127fa, 0xd4ef3085, 0x04881d05 }

k[44..47] := { 0xd9d4d039, 0xe6db99e5, 0x1fa27cf8, 0xc4ac5665 }

k[48..51] := { 0xf4292244, 0x432aff97, 0xab9423a7, 0xfc93a039 }

k[52..55] := { 0x655b59c3, 0x8f0ccc92, 0xffeff47d, 0x85845dd1 }

k[56..59] := { 0x6fa87e4f, 0xfe2ce6e0, 0xa3014314, 0x4e0811a1 }

k[60..63] := { 0xf7537e82, 0xbd3af235, 0x2ad7d2bb, 0xeb86d391 }\*/

unsigned \*calctable(unsigned \*k){

double s, pwr;

int i;

pwr = pow(2, 32);

for (i = 0; i < 64; i++) {

s = fabs(sin(1 + i));

k[i] = (unsigned)(s \* pwr);

}

return k;

}

/\*Rotate Left r by N bits

or

We can directly hardcode below table but as i explained above we are opting

generic code so shifting the bit manually.

r[ 0..15] := {7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22}

r[16..31] := {5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20}

r[32..47] := {4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23}

r[48..63] := {6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21}

\*/

unsigned rol(unsigned r, short N ){

unsigned mask1 = (1<<N) - 1;

return ((r>>(32-N)) & mask1) | ((r<<N) & ~mask1);

}

unsigned \*md5(const char \*msg, int mlen){

/\*Initialize Digest Array as A , B, C, D \*/

static DigestArray h0 = { 0x67452301, 0xEFCDAB89, 0x98BADCFE, 0x10325476 };

static DgstFctn ff[] = { &func0, &func1, &func2, &func3 };

static short M[] = { 1, 5, 3, 7 };

static short O[] = { 0, 1, 5, 0 };

static short rot0[] = { 7,12,17,22};

static short rot1[] = { 5, 9,14,20};

static short rot2[] = { 4,11,16,23};

static short rot3[] = { 6,10,15,21};

static short \*rots[] = {rot0, rot1, rot2, rot3 };

static unsigned kspace[64];

static unsigned \*k;

static DigestArray h;

DigestArray abcd;

DgstFctn fctn;

short m, o, g;

unsigned f;

short \*rotn;

union {

unsigned w[16];

char b[64];

}mm;

int os = 0;

int grp, grps, q, p;

unsigned char \*msg2;

if (k == NULL) k = calctable(kspace);

for (q = 0; q < 4; q++) h[q] = h0[q]; // initialize

{

grps = 1 + (mlen+8)/64;

msg2 = malloc( 64\*grps);

memcpy( msg2, msg, mlen);

msg2[mlen] = (unsigned char)0x80;

q = mlen + 1;

while (q < 64\*grps){ msg2[q] = 0; q++ ; }

{

MD5union u;

u.w = 8\*mlen;

q -= 8;

memcpy(msg2+q, &u.w, 4 );

}

}

for (grp=0; grp<grps; grp++){

memcpy( mm.b, msg2+os, 64);

for(q = 0; q < 4; q++) abcd[q] = h[q];

for (p = 0; p < 4; p++) {

fctn = ff[p];

rotn = rots[p];

m = M[p]; o = O[p];

for (q = 0; q < 16; q++) {

g = (m\*q + o) % 16;

f = abcd[1] + rol( abcd[0]+ fctn(abcd) + k[q+16\*p] + mm.w[g], rotn[q%4]);

abcd[0] = abcd[3];

abcd[3] = abcd[2];

abcd[2] = abcd[1];

abcd[1] = f;

}

}

for (p = 0; p < 4; p++)

h[p] += abcd[p];

os += 64;

}

return h;

}

int main( int argc, char \*argv[] )

{

int j,k;

const char \*msg = argv[1];

if(argv[1] == "-h" || argv[1] == "--help" || argc!=2){

printf("Usage : ./a.out <Your string for encryption>\n");

printf(" -h, --help\tDisplay this help and exit\n");;

}

else{

//printf("\n\t MD5 ENCRYPTION \n\n");

//printf("Input String to be Encrypted using MD5 : %s",msg);

unsigned \*d = md5(msg, strlen(msg));

MD5union u;

printf("[+] Encrypted text : ");

for (j = 0; j < 4; j++){

u.w = d[j];

for (k = 0; k < 4; k++) printf("%02x", u.b[k]);

}

printf("\n\n");

//printf("MD5 Encyption Successfully Completed!!!\n\n");

}

return 0;

}

**CONCLUSION**

We have studied the RSA public-key cryptosystem. We have delved into the basis for its working, its strength, and its ease of understanding and use. We have dealt mainly from the point of view of legal users. RSA is the most popular public-key cryptosystem available today. Its popularity stems from the fact that it can be used for both encryption and authentication, and that it has been around for many years and has successfully withstood much scrutiny. The security of RSA is related to the assumption that factoring is difficult. An easy factoring method or some other feasible attack would break RSA. RSA is built into current operating systems by Microsoft, Apple, Sun, and Novell. In hardware, RSA can be found in secure telephones, on Ethernet network cards, and on smart cards. In addition, RSA is incorporated into all of the major protocolss for secure Internet communications. The estimated installed base of RSA encryption engines is around 20 million, making it by far the most widely used public-key cryptosystem in the world.