Algorithms Explained By Ankit Fadia ankit@bol.net.in <mailto:ankit@bol.net.in>

Encryption has become a part and parcel of our lives and we have accepted the fact that data is going to encrypted and decrypted at various stages. However, there is not a single encryption algorithm followed everywhere. There are a number of algorithms existing, and I feel there is a need to understand how they work. So this text explains a number of popular encryption algorithms and makes you look at them as mathematical formulas.

Data Encryption Standard or DES

The U.S government in 1977 adopted the Data Encryption Standard (DES) algorithm. According to it's developer the DES algorithm is:

" It is a block cipher system which transforms 64-bit data blocks under a 56-bit secret key under a 56-bit secret key, by means of permutation and substitution."

Now, this tutorial will guide you through the various steps of the DES encryption algorithm making you more confident in dealing with DES encryption.

The following is a step by step guide to the DES algorithm, which was originally written by Matthew Fischer and has been edited by me-:

1.) Firstly, we need to process the key.

1.1 Get a 64-bit key from the user. (Every 8th bit is considered a parity bit. For a key to have correct parity, each byte should contain an odd number of "1" bits.)

1.2 Calculate the key schedule.

1.2.1 Perform the following permutation on the 64-bit key. (The parity bits are discarded, reducing the key to 56 bits. Bit 1 of the permuted block is bit 57 of the original key, bit 2 is bit 49, and so on

with bit
56 being bit 4 of the original key.)

Permuted Choice 1 (PC-1)

57	49	41	33	25	17	9
1	58	50	42	34	26	18
10	2	59	51	43	35	27
19	11	3	60	52	44	36
63	55	47	39	31	23	15
7	62	54	46	38	30	22
14	6	61	53	45	37	29
21	13	5	28	20	12	4

1.2.2 Split the permuted key into two halves. The first 28 bits are called C[0] and the last 28 bits are called D[0].

1.2.3 Calculate the 16 subkeys. Start with i = 1.

1.2.3.1 Perform one or two circular left shifts on both C[i-1] and D[i-1] to get C[i] and D[i], respectively. The number of shifts per iteration are given in the table below.

Iteration #	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Left Shifts	1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

1.2.3.2 Permute the concatenation C[i]D[i] as indicated below. This will yield K[i], which is 48 bits long.

Permuted Choice 2 (PC-2)

14	17	11	24	1	5
3	28	15	6	21	10
23	19	12	4	26	8
16	7	27	20	13	2
41	52	31	37	47	55
30	40	51	45	33	48
44	49	39	56	34	53

46 42 50 36 29 32

1.2.3.3 Loop back to 1.2.3.1 until K[16] has been calculated.

2 Process a 64-bit data block.

2.1 Get a 64-bit data block. If the block is shorter than 64 bits, it should be padded as appropriate for the application.

2.2 Perform the following permutation on the data block.

Initial Permutation (IP)

58 50 42 34 26 18 10 2
60 52 44 36 28 20 12 4
62 54 46 38 30 22 14 6
64 56 48 40 32 24 16 8
57 49 41 33 25 17 9 1
59 51 43 35 27 19 11 3
61 53 45 37 29 21 13 5
63 55 47 39 31 23 15 7

2.3 Split the block into two halves. The first 32 bits are called L[0], and the last 32 bits are called R[0].

2.4 Apply the 16 subkeys to the data block. Start with i = 1.

2.4.1 Expand the 32-bit R[i-1] into 48 bits according to the bit-selection function below.

Expansion (E)

32 1 2 3 4 5
4 5 6 7 8 9
8 9 10 11 12 13
12 13 14 15 16 17
16 17 18 19 20 21
20 21 22 23 24 25

24 25 26 27 28 29
28 29 30 31 32 1

2.4.2 Exclusive-or E(R[i-1]) with K[i].

2.4.3 Break E(R[i-1]) xor K[i] into eight 6-bit blocks. Bits 1-6 are B[1], bits 7-12 are B[2], and so on with bits 43-48 being B[8].

2.4.4 Substitute the values found in the S-boxes for all B[j]. Start with j = 1. All values in the S-boxes should be considered 4 bits wide.

2.4.4.1 Take the 1st and 6th bits of B[j] together as a 2-bit value (call it m) indicating the row in S[j] to look in for the substitution.

2.4.4.2 Take the 2nd through 5th bits of B[j] together as a 4-bit value (call it n) indicating the column in S[j] to find the substitution.

2.4.4.3 Replace B[j] with S[j][m][n].

Substitution Box 1 (S[1])

14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

S[2]

15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9

S[3]

10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7

1 10 13 0 6 9 8 7 4 15 14 3 11 5 2 12

S[4]

7 13 14 3 0 6 9 10 1 2 8 5 11 12 4 15
13 8 11 5 6 15 0 3 4 7 2 12 1 10 14 9
10 6 9 0 12 11 7 13 15 1 3 14 5 2 8 4
3 15 0 6 10 1 13 8 9 4 5 11 12 7 2 14

S[5]

2 12 4 1 7 10 11 6 8 5 3 15 13 0 14 9
14 11 2 12 4 7 13 1 5 0 15 10 3 9 8 6
4 2 1 11 10 13 7 8 15 9 12 5 6 3 0 14
11 8 12 7 1 14 2 13 6 15 0 9 10 4 5 3

S[6]

12 1 10 15 9 2 6 8 0 13 3 4 14 7 5 11
10 15 4 2 7 12 9 5 6 1 13 14 0 11 3 8
9 14 15 5 2 8 12 3 7 0 4 10 1 13 11 6
4 3 2 12 9 5 15 10 11 14 1 7 6 0 8 13

S[7]

4 11 2 14 15 0 8 13 3 12 9 7 5 10 6 1
13 0 11 7 4 9 1 10 14 3 5 12 2 15 8 6
1 4 11 13 12 3 7 14 10 15 6 8 0 5 9 2
6 11 13 8 1 4 10 7 9 5 0 15 14 2 3 12

S[8]

13 2 8 4 6 15 11 1 10 9 3 14 5 0 12 7
1 15 13 8 10 3 7 4 12 5 6 11 0 14 9 2
7 11 4 1 9 12 14 2 0 6 10 13 15 3 5 8
2 1 14 7 4 10 8 13 15 12 9 0 3 5 6 11

2.4.4.4 Loop back to 2.4.4.1 until all 8 blocks have been replaced.

2.4.5 Permute the concatenation of B[1] through B[8] as indicated below.

Permutation P

16	7	20	21
29	12	28	17
1	15	23	26
5	18	31	10
2	8	24	14
32	27	3	9
19	13	30	6
22	11	4	25

2.4.6 Exclusive-or the resulting value with L[i-1]. Thus, all together, your R[i] = L[i-1] xor P(S[1](B[1])...S[8](B[8])), where B[j] is a 6-bit block of E(R[i-1]) xor K[i]. (The function for R[i] is written as, R[i] = L[i-1] xor f(R[i-1], K[i]).)

2.4.7 L[i] = R[i-1].

2.4.8 Loop back to 2.4.1 until K[16] has been applied.

2.5 Perform the following permutation on the block R[16]L[16].

Final Permutation (IP-1)**

40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

This has been a description of how to use the DES algorithm to encrypt one 64-bit block. To decrypt, use the same process, but just use the keys K[i] in reverse order. That is, instead of applying K[1] for the first
iteration, apply K[16], and then K[15] for the second, on down to K[1].

Summaries:

```
Key schedule:
 C[0]D[0] = PC1(key)
 for 1 <= i <= 16
  C[i] = LS[i](C[i-1])
  D[i] = LS[i](D[i-1])
  K[i] = PC2(C[i]D[i])

Encipherment:
 L[0]R[0] = IP(plain block)
 for 1 <= i <= 16
  L[i] = R[i-1]
  R[i] = L[i-1] xor f(R[i-1], K[i])
 cipher block = FP(R[16]L[16])

Decipherment:
 R[16]L[16] = IP(cipher block)
 for 1 <= i <= 16
  R[i-1] = L[i]
  L[i-1] = R[i] xor f(L[i], K[i])
 plain block = FP(L[0]R[0])
```

To encrypt or decrypt more than 64 bits there are four official modes (defined in FIPS PUB 81). One is to go through the above-described process for each block in succession. This is called Electronic Codebook (ECB) mode. A stronger method is to exclusive-or each plaintext block with the preceding ciphertext block prior to encryption. (The first block is exclusive-or'ed with a secret 64-bit initialization vector (IV).) This is called Cipher Block Chaining (CBC) mode. The other two modes are Output Feedback (OFB) and Cipher Feedback (CFB).

When it comes to padding the data block, there are several options. One is to simply append zeros. Two suggested by FIPS PUB 81 are, if the data is binary data, fill up the block with bits that are the opposite of the last bit of data, or, if the data is ASCII data, fill up the block with random bytes and put the ASCII character for the number of pad bytes in the last byte of the block. Another technique is to pad the block with random bytes and in the last 3 bits store the original number of data bytes.

The DES algorithm can also be used to calculate checksums up to 64 bits long (see FIPS PUB 113). If the number of data bits to be check summed is not a multiple of 64, the last data block should be padded with zeros. If the data is ASCII data, the first bit of each byte should be set to 0. The data is then encrypted in CBC mode with IV = 0. The leftmost n bits (where 16 <= n <= 64, and n is a multiple of 8) of the final ciphertext block are an n-bit checksum.

Wow, that was one heck of a paper on DES. That would be all you need to implement DES. Well, if you still have not understood how the DES algorithm is implemented, then I suggest you go through the following C program:

```
#include <stdio.h>
static int keyout[17][48];
void des_init(),lshift(),cypher(),des_encrypt(),des_descrypt();
void des_init(unsigned char *key){
unsigned char c[28],d[28];
static int pc1[56] = {57,49,41,33,25,17,9,
01,58,50,42,34,26,18,
10,02,59,51,43,35,27,
19,11,03,60,52,44,36,
63,55,47,39,31,23,15,
07,62,54,46,38,30,22,
14,06,61,53,45,37,29,
21,13,05,28,20,12,04};
static int pc2[48] = {14,17,11,24,1,5,
3,28,15,6,21,10,
23,19,12,4,26,8,
```

```
16,7,27,20,13,2,
41,52,31,37,47,55,
30,40,51,45,33,48,
44,49,39,56,34,53,
46,42,50,36,29,32};
static int nls[17] = {
0,1,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1};
static int cd[56],keyb[64];
static int cnt,n=0;
register int i,j;
for(i=0;i<8;i++) /*Read in key*/
for(j=0;j<8;j++) keyb[n++]=(key[i]>>j&0x01);
for(i=0;i<56;i++) /*Permuted choice 1*/
cd[i]=keyb[pc1[1]-1];
for(i=0;i<28;i++){
c[i]=cd[i];
d[i]=cd[i+28];
}
for(cnt=1;cnt<=16;cnt++){
for(i=0;i<nls[cnt];i++){
lshift(c); lshift(d);
}
for(i=0;i<28;i++){
cd[i]=c[i];
cd[i+28]=d[i];
}
for(i=0;i<48;i++) /*Permuted Choice 2*/
keyout[cnt][i]=cd[pc2[i]-1];
}
}
static void lshift(unsigned char shft[]){
register int temp,i;
temp=shft[0];
for(i=0;i<27;i++) shft[i]=shft[i+1];
shft[27]=temp;
}
static void cypher(int *r, int cnt, int *fout){
static int expand[48],b[8][6],sout[8],pin[48];
```

```
register int i,j;
static int n,row,col,scnt;
static int p[32]={
16,7,20,21,29,12,28,17,1,15,23,26,
5,18,31,10,2,8,24,14,32,27,3,9,
19,13,30,6,22,11,4,25};
static int e[48] = {32,1,2,3,4,5,
4,5,6,7,8,9,
8,9,10,11,12,13,
12,13,14,15,16,17,
16,17,18,19,20,21,
20,21,22,23,24,25,
24,25,26,27,28,29,
28,29,30,31,32,1};
static char s[8][64] = {
14,4,13,1,2,15,11,8,3,10,6,12,5,9,0,7, /*s1*/
0,15,7,4,14,2,13,1,10,6,12,11,9,5,3,8,
4,1,14,8,13,6,2,11,15,12,9,7,3,10,5,0,
15,12,8,2,4,9,1,7,5,11,3,14,10,0,6,13,
15,1,8,14,6,11,3,4,9,7,2,13,12,0,5,10, /*s2*/
3,13,4,7,15,2,8,14,12,0,1,10,6,9,11,5,
0,14,7,11,10,4,13,1,5,8,12,6,9,3,2,15,
13,8,10,1,3,15,4,2,11,6,7,12,0,5,14,9,
10,0,9,14,6,3,15,5,1,13,12,7,11,4,2,8, /*s3*/
13,7,0,9,3,4,6,10,2,8,5,14,12,11,15,1,
13,6,4,9,8,15,3,0,11,1,2,12,5,10,14,7,
1,10,13,0,6,9,8,7,4,15,14,3,11,5,2,12,
7,13,14,3,0,6,9,10,1,2,8,5,11,12,4,15,/*s4*/
13,8,11,5,6,15,0,3,4,7,2,12,1,10,14,9,
10,6,9,0,12,11,7,13,15,1,3,14,5,2,8,4,
3,15,0,6,10,1,13,8,9,4,5,11,12,7,2,14,
2,12,4,1,7,10,11,6,8,5,3,15,13,0,14,9,/*s5*/
14,11,2,12,4,7,13,1,5,0,15,10,3,9,8,6,
4,2,1,11,10,13,7,8,15,9,12,5,6,3,0,14,
11,8,12,7,1,14,2,13,6,15,0,9,10,4,5,3,
12,1,10,15,9,2,6,8,0,13,3,4,14,7,5,11, /*s6*/
10,15,4,2,7,12,9,5,6,1,13,14,0,11,3,8,
9,14,15,5,2,8,12,3,7,0,4,10,1,13,11,6,
```

```
4,3,2,12,9,5,15,10,11,14,1,7,6,0,8,13,
4,11,2,14,15,0,8,13,3,12,9,7,5,10,6,1,/*s7*/
13,0,11,7,4,9,1,10,14,3,5,12,2,15,8,6,
1,4,11,13,12,3,7,14,10,15,6,8,0,5,9,2,
6,11,13,8,1,4,10,7,9,5,0,15,14,2,3,12,
13,2,8,4,6,15,11,1,10,9,3,14,5,0,12,7, /*s8*/
1,15,13,8,10,3,7,4,12,5,6,11,0,14,9,2,
7,11,4,1,9,12,14,2,0,6,10,13,15,3,5,8,
2,1,14,7,4,10,8,13,15,12,9,0,3,5,6,11
};
for(i=0;i<48;i++) expand[i]=r[e[i]-1]; /*Expansion Function*/
for(i=n=0;i<8;i++) {
for(j=0;j<6;j++,n++) b[i][j]=expand[n]^keyout[cnt][n];
}
/*Selection functions*/
for(scnt=n=0;scnt<8;scnt++){
row=(b[scnt][0]<<1)+b[scnt][5];
col=(b[scnt][1]<<3)+(b[scnt][2]<<2)+(b[scnt][3]<<1)+b[scnt][4];
sout[scnt]=s[scnt][(row<<4)+col];
for(i=3;i>=0;i--){
pin[n]=sout[scnt]>>i;
sout[scnt]=sout[scnt]-(pin[n++]<<i);
}
}
for(i=0;i<32;i++) fout[i]=pin[p[i]-1]; /*Permutation Function*/
}
static int p[64] = {58,50,42,34,26,18,10,2,
60,52,44,36,28,20,12,4,
62,54,46,38,30,22,14,6,
64,56,48,40,32,24,16,8,
5 = {58,50,42,34,26,18,10,2,
60,52,44,36,28,20,12,4,
62,54,46,38,30,22,14,6,
64,56,48,40,32,24,16,8,
57,49,41,33,25,17,9,1,
59,51,43,35,27,19,11,3,
61,53,45,37,29,21,13,5,
63,55,47,39,31,23,15,7};
```

```
static int invp[64]={
40, 8,48,16,56,24,64,32,39, 7,47,15,55,23,63,31,
38, 6,46,14,54,22,62,30,37, 5,45,13,53,21,61,29,
36, 4,44,12,52,20,60,28,35, 3,43,11,51,19,59,27,
34, 2,42,10,50,18,58,26,33, 1,41, 9,49,17,57,25};
void des_encrypt(unsigned char *input){
static unsigned char out[64];
static int inputb[64],lr[64],l[32],r[32];
static int fn[32];
static int cnt,n;
register int i,j;
for(i=n=0;i<8;i++)
for(j=0;j<8;j++) inputb[n++]=(input[i]>>j&0x01);
for(i=0;i<64;i++){ /*Initial Permutation*/
lr[i]=inputb[p[i]-1];
if(i<32) l[i]=lr[i];
else r[i-32]=lr[i];
}
for(cnt=1;cnt<=16;cnt++){ /*Main encryption loop*/
cypher(r,cnt,fn);
for(i=0;i<32;i++){
j=r[i];
r[i]=l[i]^fn[i];
l[i]=j;
}
}
for(i=0;i<32;i++){
lr[i]=r[i];
lr[i+32]=l[i];
}
for(i=0;i<64;i++) out[i]=lr[invp[i]-1]; /*Inverse IP*/
for(i=1;i<=8;i++)
for(j=1;j<=8;j++) input[i-1]=(input[i-1]<<1)|out[i*8-j];
}
void des_decrypt(unsigned char *input){
static unsigned char out[64];
static int inputb[64],lr[64],l[32],r[32];
static int fn[32];
```

```
static int cnt,rtemp,n;
register int i,j;
for(i=n=0;i<8;i++)
for(j=0;j<8;j++) inputb[n++]=(input[i]>>j&0x01);
for(i=0;i<64;i++){ /*Initial Permutation*/
lr[i]=inputb[p[i]-1];
if(i<32) l[i]=lr[i];
else r[i-32]=lr[i];
}
for(cnt=16;cnt>0;cnt--){ /*Main decryption loop*/
cypher(r,cnt,fn);
for(i=0;i<32;i++){
rtemp=r[i];
if(l[i]==1 && fn[i]==1) r[i]=0;
else r[i]=(l[i] || fn[i]);
l[i]=rtemp;
}
}
for(i=0;i<32;i++){
lr[i]=r[i];
lr[i+32]=l[i];
}
for(i=0;i<64;i++) out[i]=lr[invp[i]-1]; /*Inverse IP*/
for(i=1;i<=8;i++)
for(j=1;j<=8;j++) input[i-1]=(input[i-1]<<1) | out[i*8-j];
}
int main(int argc, char *argv[]){
unsigned char *key;
unsigned char data[8];
int n;
FILE *in;
FILE *out;
if (argc!=4) {
printf("\r\nUsage: des [e][d] <source file> <destination file>\r\n");
return 1;
}
key=(unsigned char*)getpass("Enter Key:");
des_init(key);
```

```
if((in=fopen(argv[2],"rb"))==NULL){
fprintf(stderr,"\r\nCould not open input file: %s",argv[2]);
return 2;
}
if((out=fopen(argv[3],"wb"))==NULL){
fprintf(stderr,"\r\nCould not open output file: %s",argv[3]);
return 3;
}
if(argv[1][0]=='e'){
while ((n=fread(data,1,8,in)) >0){
des_encrypt(data);
printf("data enctyted");
if(fwrite(data,1,8,out) < 8){
fprintf(stderr,"\r\nError writing to output file\r\n");
return(3);
}
}
}
if(argv[1][0]=='d'){
while ((n=fread(data,1,8,in)) >0){
des_decrypt(data);
if(fwrite(data,1,8,out) < 8){
fprintf(stderr,"\r\nError writing to output file\r\n");
return(3);
}
}
}
fclose(in); fclose(out);
return 0;
}ntf(stderr,"\r\nError writing to output file\r\n");
return(3);
}
}
}
fclose(in); fclose(out);
return 0;
}
```

The RSA Encryption Algorithm

RSA is one of the most popular encryption algorithms around. It was invented in 1977 by three MIT scientists; Ronald Rivest, Adi Shamir and Leonard Adelman. This algorithm uses very very large prime numbers to generate public and private keys. For more details about the implementation of the RSA algorithm, read on.

The entire RSA algorithm is based on the concept of factoring. Factoring is very easy to calculate i.e. an algorithm based on factoring can easily be carried out, however, the strength of an algorithm based on factoring lies in the fact that factoring is quite difficult to reverse. So, encrypting data using a factoring algorithm can be done without much of a problem, however to decrypt or crack an algorithm using a factoring algorithm is not that easy.

Say you have two numbers x and y. Now it is relatively easier to find out the final numbers when they are multiplied by a number N, however on the other hand, if only N is know to us, then it is not easy to calculate x and y. On top of this in RSA, N is very large number. This makes the calculation all the more difficult. RSA uses factors of around 150 digits, making it not only impossible for manual calculation but also making sure that the RSA encryption cannot be cracked within a feasible amount of time.

In case of the encryption process, as RSA is a block encryption algorithm, the entire data is broken into blocks and each block is treated as a sequence of bits, with the number of digits being just a little less than N. Each block is considered as a single digit, and multiplied 'e' number of times by itself, [In the case of PGP, e is normally 17]. The result thus obtained, is divided by N and the remainder obtained is the final encrypted message.

In case of the decryption process, the recipient, makes use of another special number; 'k' where (ke-1) is divisible by (p-1)(q-1). The value k is chosen such that multiplying the encrypted message k times by itself and then dividing by N, gives the original message as the remainder. So basically to find out k, p and q should be known.

e and N constitute the public key which can be freely distributed, while k forms the private key, which should be kept a secret.

Note: In this case, e and k and symmetric.

To understand the working of the RSA algorithm, study the following Perl program which implements it.

#!/usr/local/bin/perl -s-- #export-a-crypto-system sig, RSA in 4 lines

PERL:

```
#
# -d (decrypt)
# or -e (encrypt)
#
# $k is exponent, $n is modulus; $k and $n in hex
#
# use of -s was contributed by Jeff Friedl, a cool perl hacker
#
# the $e-$d (grok that? awesome hack by Jeff also) checks for -d or
-e:
#
# when perl -s sets $x for -x so that means $d is set for -d, $e for
-e
# if they are both set 1-1 = 0 so it fails if neither are set it fails
# and if either one is set we're ok! This is to get around using | ,
# as | has higher precedence than & things group wrongly.
#
$e-$d&(($k,$n)=@ARGV)==2||die"$0 -d|-e key mod <in >out\n";
#
# $v will be the digits of output per block, $w the digits of input per
block.
# If encrypting need to reduce $w so input is guaranteed to be less
than
# modulus; for decrypting reduce $v to match.
#
# blocks are based on modulus size in hex digits rounded up to nearest
even
# length (~1&1+length$n) so that things will unpack properly
#
$v=$w=1+length$n&~1;
$v-=$d*2:$w-=$e*2;
#
# Make $_ be the exponent $k as a binary bit string
#
# Add a leading 0 to make length of $k be even so that it will fill
# Bytes when packed as 2 digits per byte
#
```

```
$_=unpack('B*',pack('H*',1&length$k?"0$k":$k));
#
# strip leading 0's from $_
#
s/^0+//;
#
# Turn every 0 into "d*ln%", every 1 into "d*ln%lm*ln%". These are dc
codes
# which construct an exponentiation algorithm for that exponent.
# "d*ln%" is duplicate, square, load n, modulus; e.g. square the number
# on the stack, mod n. "d*ln%lm*ln%" does this then, load m, multiply,
# load n, modulus; e.g. then multiply by m mod n. This is the square
and
# multiply algorithm for modular exponentiation.
#
# (Kudos to Hal for shortened this one by 4 chars)
#
s/1/0lM*ln%/g;
s/0/d*ln%/g;
#
# Encryption/decryption loop. Read $w/2 bytes of data to $m.
#
while(read(STDIN,$m,$w/2)){
#
# Turn data into equivalent hex digits in $m
#
$m=unpack("H$w",$m);
#
# Run dc: 16 bit radix for input and output; $m into dc register "M";
# $n into dc register "n"; execute $_, the exponentiation program
above.
# "\U...\E" forces upper case on the hex digits as dc requires.
# Put the result in $e.
#
$a=`echo 16oOi\U$m SM$n\Esn1$_ p|dc`;
#
# Pad the result with leading 0's to $v digits, pack to raw data and
output.
```

```
#
print pack("H$v",'0'x($v+1-length$a).$a);
}
```

Blowfish

Blowfish is yet another popular encryption algorithm which is based on:
The following C program demonstrates the implementation of the
Blowfish encryption algorithm:

```
/**********************blowfish.h*********************/
/* $Id: blowfish.h,v 1.3 1995/01/23 12:38:02 pr Exp pr $*/
#define MAXKEYBYTES 56 /* 448 bits */
#define bf_N 16
#define noErr 0
#define DATAERROR -1
#define KEYBYTES 8
#define subkeyfilename "Blowfish.dat"
#define UWORD_32bits unsigned long
#define UWORD_16bits unsigned short
#define UBYTE_08bits unsigned char
/* choose a byte order for your hardware */
/* ABCD - big endian - motorola */
#ifdef ORDER_ABCD
union aword {
UWORD_32bits word;
UBYTE_08bits byte [4];
struct {
unsigned int byte0:8;
unsigned int byte1:8;
unsigned int byte2:8;
unsigned int byte3:8;
} w;
};
#endif /* ORDER_ABCD */
/* DCBA - little endian - intel */
#ifdef ORDER_DCBA
union aword {
UWORD_32bits word;
UBYTE_08bits byte [4];
struct {
```

```
unsigned int byte3:8;
unsigned int byte2:8;
unsigned int byte1:8;
unsigned int byte0:8;
} w;
};
#endif /* ORDER_DCBA */
/* BADC - vax */
#ifdef ORDER_BADC
union aword {
UWORD_32bits word;
UBYTE_08bits byte [4];
struct {
unsigned int byte1:8;
unsigned int byte0:8;
unsigned int byte3:8;
unsigned int byte2:8;
} w;
};
#endif /* ORDER_BADC */
short opensubkeyfile(void);
unsigned long F(unsigned long x);
void Blowfish_encipher(unsigned long *xl, unsigned long *xr);
void Blowfish_decipher(unsigned long *xl, unsigned long *xr);
short InitializeBlowfish(unsigned char key[], short keybytes);
/**********************blowfish.c**********************/
/* TODO: test with zero length key */
/* TODO: test with a through z as key and plain text */
/* TODO: make this byte order independent */
#include <stdio.h> /* used for debugging */
#ifdef MACINTOSH
#include <Types.h> /* FIXME: do we need this? */
#endif
#include "blowfish.h"
#include "bf_tab.h" /* P-box P-array, S-box */
#define S(x,i) (bf_S[i][x.w.byte##i])
#define bf_F(x) (((S(x,0) + S(x,1)) ^ S(x,2)) + S(x,3))
#define ROUND(a,b,n) (a.word ^= bf_F(b) ^ bf_P[n])
```

```
inline
void Blowfish_encipher(UWORD_32bits *xl, UWORD_32bits *xr)
{
union aword Xl;
union aword Xr;
Xl.word = *xl;
Xr.word = *xr;
Xl.word ^= bf_P[0];
ROUND (Xr, Xl, 1); ROUND (Xl, Xr, 2);
ROUND (Xr, Xl, 3); ROUND (Xl, Xr, 4);
ROUND (Xr, Xl, 5); ROUND (Xl, Xr, 6);
ROUND (Xr, Xl, 7); ROUND (Xl, Xr, 8);
ROUND (Xr, Xl, 9); ROUND (Xl, Xr, 10);
ROUND (Xr, Xl, 11); ROUND (Xl, Xr, 12);
ROUND (Xr, Xl, 13); ROUND (Xl, Xr, 14);
ROUND (Xr, Xl, 15); ROUND (Xl, Xr, 16);
Xr.word ^= bf_P[17];
*xr = Xl.word;
*xl = Xr.word;
}
void Blowfish_decipher(UWORD_32bits *xl, UWORD_32bits *xr)
{
union aword Xl;
union aword Xr;
Xl = *xl;
Xr = *xr;
Xl.word ^= bf_P[17];
ROUND (Xr, Xl, 16); ROUND (Xl, Xr, 15);
ROUND (Xr, Xl, 14); ROUND (Xl, Xr, 13);
ROUND (Xr, Xl, 12); ROUND (Xl, Xr, 11);
ROUND (Xr, Xl, 10); ROUND (Xl, Xr, 9);
ROUND (Xr, Xl, 8); ROUND (Xl, Xr, 7);
ROUND (Xr, Xl, 6); ROUND (Xl, Xr, 5);
ROUND (Xr, Xl, 4); ROUND (Xl, Xr, 3);
ROUND (Xr, Xl, 2); ROUND (Xl, Xr, 1);
Xr.word ^= bf_P[0];
*xl = Xr.word;
*xr = Xl.word;
```

```
}
/* FIXME: Blowfish_Initialize() ??? */
short InitializeBlowfish(UBYTE_08bits key[], short keybytes)
{
short i; /* FIXME: unsigned int, char? */
short j; /* FIXME: unsigned int, char? */
UWORD_32bits data;
UWORD_32bits datal;
UWORD_32bits datar;
union aword temp;
/* fprintf (stderr, "0x%x 0x%x ", bf_P[0], bf_P[1]); /* DEBUG */
/* fprintf (stderr, "%d %d\n", bf_P[0], bf_P[1]); /* DEBUG */
j = 0;
for (i = 0; i < bf_N + 2; ++i) {
temp.word = 0;
temp.w.byte0 = key[j];
temp.w.byte1 = key[(j+1)%keybytes];
temp.w.byte2 = key[(j+2)%keybytes];
temp.w.byte3 = key[(j+3)%keybytes];
data = temp.word;
bf_P[i] = bf_P[i] ^ data;
j = (j + 4) % keybytes;
}
datal = 0x00000000;
datar = 0x00000000;
for (i = 0; i < bf_N + 2; i += 2) {
Blowfish_encipher(&datal, &datar);
bf_P[i] = datal;
bf_P[i + 1] = datar;
}
for (i = 0; i < 4; ++i) {
for (j = 0; j < 256; j += 2) {
Blowfish_encipher(&datal, &datar);
bf_S[i][j] = datal;
bf_S[i][j + 1] = datar;
}
}
return 0;
```

```
}
=============== bf_tab.h ==============
/* bf_tab.h: Blowfish P-box and S-box tables */
static UWORD_32bits bf_P[bf_N + 2] = {
0x243f6a88, 0x85a308d3, 0x13198a2e, 0x03707344,
0xa4093822, 0x299f31d0, 0x082efa98, 0xec4e6c89,
0x452821e6, 0x38d01377, 0xbe5466cf, 0x34e90c6c,
0xc0ac29b7, 0xc97c50dd, 0x3f84d5b5, 0xb5470917,
0x9216d5d9, 0x8979fb1b,
};
static UWORD_32bits bf_S[4][256] = {
0xd1310ba6, 0x98dfb5ac, 0x2ffd72db, 0xd01adfb7,
0xb8e1afed, 0x6a267e96, 0xba7c9045, 0xf12c7f99,
0x24a19947, 0xb3916cf7, 0x0801f2e2, 0x858efc16,
0x636920d8, 0x71574e69, 0xa458fea3, 0xf4933d7e,
0x0d95748f, 0x728eb658, 0x718bcd58, 0x82154aee,
0x7b54a41d, 0xc25a59b5, 0x9c30d539, 0x2af26013,
0xc5d1b023, 0x286085f0, 0xca417918, 0xb8db38ef,
0x8e79dcb0, 0x603a180e, 0x6c9e0e8b, 0xb01e8a3e,
0xd71577c1, 0xbd314b27, 0x78af2fda, 0x55605c60,
0xe65525f3, 0xaa55ab94, 0x57489862, 0x63e81440,
0x55ca396a, 0x2aab10b6, 0xb4cc5c34, 0x1141e8ce,
0xa15486af, 0x7c72e993, 0xb3ee1411, 0x636fbc2a,
0x2ba9c55d, 0x741831f6, 0xce5c3e16, 0x9b87931e,
0xafd6ba33, 0x6c24cf5c, 0x7a325381, 0x28958677,
0x3b8f4898, 0x6b4bb9af, 0xc4bfe81b, 0x66282193,
0x61d809cc, 0xfb21a991, 0x487cac60, 0x5dec8032,
0xef845d5d, 0xe98575b1, 0xdc262302, 0xeb651b88,
0x23893e81, 0xd396acc5, 0x0f6d6ff3, 0x83f44239,
0x2e0b4482, 0xa4842004, 0x69c8f04a, 0x9e1f9b5e,
0x21c66842, 0xf6e96c9a, 0x670c9c61, 0xabd388f0,
0x6a51a0d2, 0xd8542f68, 0x960fa728, 0xab5133a3,
0x6eef0b6c, 0x137a3be4, 0xba3bf050, 0x7efb2a98,
0xa1f1651d, 0x39af0176, 0x66ca593e, 0x82430e88,
0x8cee8619, 0x456f9fb4, 0x7d84a5c3, 0x3b8b5ebe,
0xe06f75d8, 0x85c12073, 0x401a449f, 0x56c16aa6,
0x4ed3aa62, 0x363f7706, 0x1bfedf72, 0x429b023d,
0x37d0d724, 0xd00a1248, 0xdb0fead3, 0x49f1c09b,
```

0x075372c9, 0x80991b7b, 0x25d479d8, 0xf6e8def7,
0xe3fe501a, 0xb6794c3b, 0x976ce0bd, 0x04c006ba,
0xc1a94fb6, 0x409f60c4, 0x5e5c9ec2, 0x196a2463,
0x68fb6faf, 0x3e6c53b5, 0x1339b2eb, 0x3b52ec6f,
0x6dfc511f, 0x9b30952c, 0xcc814544, 0xaf5ebd09,
0xbee3d004, 0xde334afd, 0x660f2807, 0x192e4bb3,
0xc0cba857, 0x45c8740f, 0xd20b5f39, 0xb9d3fbdb,
0x5579c0bd, 0x1a60320a, 0xd6a100c6, 0x402c7279,
0x679f25fe, 0xfb1fa3cc, 0x8ea5e9f8, 0xdb3222f8,
0x3c7516df, 0xfd616b15, 0x2f501ec8, 0xad0552ab,
0x323db5fa, 0xfd238760, 0x53317b48, 0x3e00df82,
0x9e5c57bb, 0xca6f8ca0, 0x1a87562e, 0xdf1769db,
0xd542a8f6, 0x287effc3, 0xac6732c6, 0x8c4f5573,
0x695b27b0, 0xbbca58c8, 0xe1ffa35d, 0xb8f011a0,
0x10fa3d98, 0xfd2183b8, 0x4afcb56c, 0x2dd1d35b,
0x9a53e479, 0xb6f84565, 0xd28e49bc, 0x4bfb9790,
0xe1ddf2da, 0xa4cb7e33, 0x62fb1341, 0xcee4c6e8,
0xef20cada, 0x36774c01, 0xd07e9efe, 0x2bf11fb4,
0x95dbda4d, 0xae909198, 0xeaad8e71, 0x6b93d5a0,
0xd08ed1d0, 0xafc725e0, 0x8e3c5b2f, 0x8e7594b7,
0x8ff6e2fb, 0xf2122b64, 0x8888b812, 0x900df01c,
0x4fad5ea0, 0x688fc31c, 0xd1cff191, 0xb3a8c1ad,
0x2f2f2218, 0xbe0e1777, 0xea752dfe, 0x8b021fa1,
0xe5a0cc0f, 0xb56f74e8, 0x18acf3d6, 0xce89e299,
0xb4a84fe0, 0xfd13e0b7, 0x7cc43b81, 0xd2ada8d9,
0x165fa266, 0x80957705, 0x93cc7314, 0x211a1477,
0xe6ad2065, 0x77b5fa86, 0xc75442f5, 0xfb9d35cf,
0xebcdaf0c, 0x7b3e89a0, 0xd6411bd3, 0xae1e7e49,
0x00250e2d, 0x2071b35e, 0x226800bb, 0x57b8e0af,
0x2464369b, 0xf009b91e, 0x5563911d, 0x59dfa6aa,
0x78c14389, 0xd95a537f, 0x207d5ba2, 0x02e5b9c5,
0x83260376, 0x6295cfa9, 0x11c81968, 0x4e734a41,
0xb3472dca, 0x7b14a94a, 0x1b510052, 0x9a532915,
0xd60f573f, 0xbc9bc6e4, 0x2b60a476, 0x81e67400,
0x08ba6fb5, 0x571be91f, 0xf296ec6b, 0x2a0dd915,
0xb6636521, 0xe7b9f9b6, 0xff34052e, 0xc5855664,
0x53b02d5d, 0xa99f8fa1, 0x08ba4799, 0x6e85076a,
0x4b7a70e9, 0xb5b32944, 0xdb75092e, 0xc4192623,

0xad6ea6b0, 0x49a7df7d, 0x9cee60b8, 0x8fedb266,
0xecaa8c71, 0x699a17ff, 0x5664526c, 0xc2b19ee1,
0x193602a5, 0x75094c29, 0xa0591340, 0xe4183a3e,
0x3f54989a, 0x5b429d65, 0x6b8fe4d6, 0x99f73fd6,
0xa1d29c07, 0xefe830f5, 0x4d2d38e6, 0xf0255dc1,
0x4cdd2086, 0x8470eb26, 0x6382e9c6, 0x021ecc5e,
0x09686b3f, 0x3ebaefc9, 0x3c971814, 0x6b6a70a1,
0x687f3584, 0x52a0e286, 0xb79c5305, 0xaa500737,
0x3e07841c, 0x7fdeae5c, 0x8e7d44ec, 0x5716f2b8,
0xb03ada37, 0xf0500c0d, 0xf01c1f04, 0x0200b3ff,
0xae0cf51a, 0x3cb574b2, 0x25837a58, 0xdc0921bd,
0xd19113f9, 0x7ca92ff6, 0x94324773, 0x22f54701,
0x3ae5e581, 0x37c2dadc, 0xc8b57634, 0x9af3dda7,
0xa9446146, 0x0fd0030e, 0xecc8c73e, 0xa4751e41,
0xe238cd99, 0x3bea0e2f, 0x3280bba1, 0x183eb331,
0x4e548b38, 0x4f6db908, 0x6f420d03, 0xf60a04bf,
0x2cb81290, 0x24977c79, 0x5679b072, 0xbcaf89af,
0xde9a771f, 0xd9930810, 0xb38bae12, 0xdccf3f2e,
0x5512721f, 0x2e6b7124, 0x501adde6, 0x9f84cd87,
0x7a584718, 0x7408da17, 0xbc9f9abc, 0xe94b7d8c,
0xec7aec3a, 0xdb851dfa, 0x63094366, 0xc464c3d2,
0xef1c1847, 0x3215d908, 0xdd433b37, 0x24c2ba16,
0x12a14d43, 0x2a65c451, 0x50940002, 0x133ae4dd,
0x71dff89e, 0x10314e55, 0x81ac77d6, 0x5f11199b,
0x043556f1, 0xd7a3c76b, 0x3c11183b, 0x5924a509,
0xf28fe6ed, 0x97f1fbfa, 0x9ebabf2c, 0x1e153c6e,
0x86e34570, 0xeae96fb1, 0x860e5e0a, 0x5a3e2ab3,
0x771fe71c, 0x4e3d06fa, 0x2965dcb9, 0x99e71d0f,
0x803e89d6, 0x5266c825, 0x2e4cc978, 0x9c10b36a,
0xc6150eba, 0x94e2ea78, 0xa5fc3c53, 0x1e0a2df4,
0xf2f74ea7, 0x361d2b3d, 0x1939260f, 0x19c27960,
0x5223a708, 0xf71312b6, 0xebadfe6e, 0xeac31f66,
0xe3bc4595, 0xa67bc883, 0xb17f37d1, 0x018cff28,
0xc332ddef, 0xbe6c5aa5, 0x65582185, 0x68ab9802,
0xeecea50f, 0xdb2f953b, 0x2aef7dad, 0x5b6e2f84,
0x1521b628, 0x29076170, 0xecdd4775, 0x619f1510,
0x13cca830, 0xeb61bd96, 0x0334fe1e, 0xaa0363cf,
0xb5735c90, 0x4c70a239, 0xd59e9e0b, 0xcbaade14,

0xeecc86bc, 0x60622ca7, 0x9cab5cab, 0xb2f3846e,
0x648b1eaf, 0x19bdf0ca, 0xa02369b9, 0x655abb50,
0x40685a32, 0x3c2ab4b3, 0x319ee9d5, 0xc021b8f7,
0x9b540b19, 0x875fa099, 0x95f7997e, 0x623d7da8,
0xf837889a, 0x97e32d77, 0x11ed935f, 0x16681281,
0x0e358829, 0xc7e61fd6, 0x96dedfa1, 0x7858ba99,
0x57f584a5, 0x1b227263, 0x9b83c3ff, 0x1ac24696,
0xcdb30aeb, 0x532e3054, 0x8fd948e4, 0x6dbc3128,
0x58ebf2ef, 0x34c6ffea, 0xfe28ed61, 0xee7c3c73,
0x5d4a14d9, 0xe864b7e3, 0x42105d14, 0x203e13e0,
0x45eee2b6, 0xa3aaabea, 0xdb6c4f15, 0xfacb4fd0,
0xc742f442, 0xef6abbb5, 0x654f3b1d, 0x41cd2105,
0xd81e799e, 0x86854dc7, 0xe44b476a, 0x3d816250,
0xcf62a1f2, 0x5b8d2646, 0xfc8883a0, 0xc1c7b6a3,
0x7f1524c3, 0x69cb7492, 0x47848a0b, 0x5692b285,
0x095bbf00, 0xad19489d, 0x1462b174, 0x23820e00,
0x58428d2a, 0x0c55f5ea, 0x1dadf43e, 0x233f7061,
0x3372f092, 0x8d937e41, 0xd65fecf1, 0x6c223bdb,
0x7cde3759, 0xcbee7460, 0x4085f2a7, 0xce77326e,
0xa6078084, 0x19f8509e, 0xe8efd855, 0x61d99735,
0xa969a7aa, 0xc50c06c2, 0x5a04abfc, 0x800bcadc,
0x9e447a2e, 0xc3453484, 0xfdd56705, 0x0e1e9ec9,
0xdb73dbd3, 0x105588cd, 0x675fda79, 0xe3674340,
0xc5c43465, 0x713e38d8, 0x3d28f89e, 0xf16dff20,
0x153e21e7, 0x8fb03d4a, 0xe6e39f2b, 0xdb83adf7,
0xe93d5a68, 0x948140f7, 0xf64c261c, 0x94692934,
0x411520f7, 0x7602d4f7, 0xbcf46b2e, 0xd4a20068,
0xd4082471, 0x3320f46a, 0x43b7d4b7, 0x500061af,
0x1e39f62e, 0x97244546, 0x14214f74, 0xbf8b8840,
0x4d95fc1d, 0x96b591af, 0x70f4ddd3, 0x66a02f45,
0xbfbc09ec, 0x03bd9785, 0x7fac6dd0, 0x31cb8504,
0x96eb27b3, 0x55fd3941, 0xda2547e6, 0xabca0a9a,
0x28507825, 0x530429f4, 0x0a2c86da, 0xe9b66dfb,
0x68dc1462, 0xd7486900, 0x680ec0a4, 0x27a18dee,
0x4f3ffea2, 0xe887ad8c, 0xb58ce006, 0x7af4d6b6,
0xaace1e7c, 0xd3375fec, 0xce78a399, 0x406b2a42,
0x20fe9e35, 0xd9f385b9, 0xee39d7ab, 0x3b124e8b,
0x1dc9faf7, 0x4b6d1856, 0x26a36631, 0xeae397b2,

0x3a6efa74, 0xdd5b4332, 0x6841e7f7, 0xca7820fb,
0xfb0af54e, 0xd8feb397, 0x454056ac, 0xba489527,
0x55533a3a, 0x20838d87, 0xfe6ba9b7, 0xd096954b,
0x55a867bc, 0xa1159a58, 0xcca92963, 0x99e1db33,
0xa62a4a56, 0x3f3125f9, 0x5ef47e1c, 0x9029317c,
0xfdf8e802, 0x04272f70, 0x80bb155c, 0x05282ce3,
0x95c11548, 0xe4c66d22, 0x48c1133f, 0xc70f86dc,
0x07f9c9ee, 0x41041f0f, 0x404779a4, 0x5d886e17,
0x325f51eb, 0xd59bc0d1, 0xf2bcc18f, 0x41113564,
0x257b7834, 0x602a9c60, 0xdff8e8a3, 0x1f636c1b,
0x0e12b4c2, 0x02e1329e, 0xaf664fd1, 0xcad18115,
0x6b2395e0, 0x333e92e1, 0x3b240b62, 0xeebeb922,
0x85b2a20e, 0xe6ba0d99, 0xde720c8c, 0x2da2f728,
0xd0127845, 0x95b794fd, 0x647d0862, 0xe7ccf5f0,
0x5449a36f, 0x877d48fa, 0xc39dfd27, 0xf33e8d1e,
0x0a476341, 0x992eff74, 0x3a6f6eab, 0xf4f8fd37,
0xa812dc60, 0xa1ebddf8, 0x991be14c, 0xdb6e6b0d,
0xc67b5510, 0x6d672c37, 0x2765d43b, 0xdcd0e804,
0xf1290dc7, 0xcc00ffa3, 0xb5390f92, 0x690fed0b,
0x667b9ffb, 0xcedb7d9c, 0xa091cf0b, 0xd9155ea3,
0xbb132f88, 0x515bad24, 0x7b9479bf, 0x763bd6eb,
0x37392eb3, 0xcc115979, 0x8026e297, 0xf42e312d,
0x6842ada7, 0xc66a2b3b, 0x12754ccc, 0x782ef11c,
0x6a124237, 0xb79251e7, 0x06a1bbe6, 0x4bfb6350,
0x1a6b1018, 0x11caedfa, 0x3d25bdd8, 0xe2e1c3c9,
0x44421659, 0x0a121386, 0xd90cec6e, 0xd5abea2a,
0x64af674e, 0xda86a85f, 0xbebfe988, 0x64e4c3fe,
0x9dbc8057, 0xf0f7c086, 0x60787bf8, 0x6003604d,
0xd1fd8346, 0xf6381fb0, 0x7745ae04, 0xd736fccc,
0x83426b33, 0xf01eab71, 0xb0804187, 0x3c005e5f,
0x77a057be, 0xbde8ae24, 0x55464299, 0xbf582e61,
0x4e58f48f, 0xf2ddfda2, 0xf474ef38, 0x8789bdc2,
0x5366f9c3, 0xc8b38e74, 0xb475f255, 0x46fcd9b9,
0x7aeb2661, 0x8b1ddf84, 0x846a0e79, 0x915f95e2,
0x466e598e, 0x20b45770, 0x8cd55591, 0xc902de4c,
0xb90bace1, 0xbb8205d0, 0x11a86248, 0x7574a99e,
0xb77f19b6, 0xe0a9dc09, 0x662d09a1, 0xc4324633,
0xe85a1f02, 0x09f0be8c, 0x4a99a025, 0x1d6efe10,

0x1ab93d1d, 0x0ba5a4df, 0xa186f20f, 0x2868f169,
0xdcb7da83, 0x573906fe, 0xa1e2ce9b, 0x4fcd7f52,
0x50115e01, 0xa70683fa, 0xa002b5c4, 0x0de6d027,
0x9af88c27, 0x773f8641, 0xc3604c06, 0x61a806b5,
0xf0177a28, 0xc0f586e0, 0x006058aa, 0x30dc7d62,
0x11e69ed7, 0x2338ea63, 0x53c2dd94, 0xc2c21634,
0xbbcbee56, 0x90bcb6de, 0xebfc7da1, 0xce591d76,
0x6f05e409, 0x4b7c0188, 0x39720a3d, 0x7c927c24,
0x86e3725f, 0x724d9db9, 0x1ac15bb4, 0xd39eb8fc,
0xed545578, 0x08fca5b5, 0xd83d7cd3, 0x4dad0fc4,
0x1e50ef5e, 0xb161e6f8, 0xa28514d9, 0x6c51133c,
0x6fd5c7e7, 0x56e14ec4, 0x362abfce, 0xddc6c837,
0xd79a3234, 0x92638212, 0x670efa8e, 0x406000e0,
0x3a39ce37, 0xd3faf5cf, 0xabc27737, 0x5ac52d1b,
0x5cb0679e, 0x4fa33742, 0xd3822740, 0x99bc9bbe,
0xd5118e9d, 0xbf0f7315, 0xd62d1c7e, 0xc700c47b,
0xb78c1b6b, 0x21a19045, 0xb26eb1be, 0x6a366eb4,
0x5748ab2f, 0xbc946e79, 0xc6a376d2, 0x6549c2c8,
0x530ff8ee, 0x468dde7d, 0xd5730a1d, 0x4cd04dc6,
0x2939bbdb, 0xa9ba4650, 0xac9526e8, 0xbe5ee304,
0xa1fad5f0, 0x6a2d519a, 0x63ef8ce2, 0x9a86ee22,
0xc089c2b8, 0x43242ef6, 0xa51e03aa, 0x9cf2d0a4,
0x83c061ba, 0x9be96a4d, 0x8fe51550, 0xba645bd6,
0x2826a2f9, 0xa73a3ae1, 0x4ba99586, 0xef5562e9,
0xc72fefd3, 0xf752f7da, 0x3f046f69, 0x77fa0a59,
0x80e4a915, 0x87b08601, 0x9b09e6ad, 0x3b3ee593,
0xe990fd5a, 0x9e34d797, 0x2cf0b7d9, 0x022b8b51,
0x96d5ac3a, 0x017da67d, 0xd1cf3ed6, 0x7c7d2d28,
0x1f9f25cf, 0xadf2b89b, 0x5ad6b472, 0x5a88f54c,
0xe029ac71, 0xe019a5e6, 0x47b0acfd, 0xed93fa9b,
0xe8d3c48d, 0x283b57cc, 0xf8d56629, 0x79132e28,
0x785f0191, 0xed756055, 0xf7960e44, 0xe3d35e8c,
0x15056dd4, 0x88f46dba, 0x03a16125, 0x0564f0bd,
0xc3eb9e15, 0x3c9057a2, 0x97271aec, 0xa93a072a,
0x1b3f6d9b, 0x1e6321f5, 0xf59c66fb, 0x26dcf319,
0x7533d928, 0xb155fdf5, 0x03563482, 0x8aba3cbb,
0x28517711, 0xc20ad9f8, 0xabcc5167, 0xccad925f,
0x4de81751, 0x3830dc8e, 0x379d5862, 0x9320f991,

0xea7a90c2, 0xfb3e7bce, 0x5121ce64, 0x774fbe32,
0xa8b6e37e, 0xc3293d46, 0x48de5369, 0x6413e680,
0xa2ae0810, 0xdd6db224, 0x69852dfd, 0x09072166,
0xb39a460a, 0x6445c0dd, 0x586cdecf, 0x1c20c8ae,
0x5bbef7dd, 0x1b588d40, 0xccd2017f, 0x6bb4e3bb,
0xdda26a7e, 0x3a59ff45, 0x3e350a44, 0xbcb4cdd5,
0x72eacea8, 0xfa6484bb, 0x8d6612ae, 0xbf3c6f47,
0xd29be463, 0x542f5d9e, 0xaec2771b, 0xf64e6370,
0x740e0d8d, 0xe75b1357, 0xf8721671, 0xaf537d5d,
0x4040cb08, 0x4eb4e2cc, 0x34d2466a, 0x0115af84,
0xe1b00428, 0x95983a1d, 0x06b89fb4, 0xce6ea048,
0x6f3f3b82, 0x3520ab82, 0x011a1d4b, 0x277227f8,
0x611560b1, 0xe7933fdc, 0xbb3a792b, 0x344525bd,
0xa08839e1, 0x51ce794b, 0x2f32c9b7, 0xa01fbac9,
0xe01cc87e, 0xbcc7d1f6, 0xcf0111c3, 0xa1e8aac7,
0x1a908749, 0xd44fbd9a, 0xd0dadecb, 0xd50ada38,
0x0339c32a, 0xc6913667, 0x8df9317c, 0xe0b12b4f,
0xf79e59b7, 0x43f5bb3a, 0xf2d519ff, 0x27d9459c,
0xbf97222c, 0x15e6fc2a, 0x0f91fc71, 0x9b941525,
0xfae59361, 0xceb69ceb, 0xc2a86459, 0x12baa8d1,
0xb6c1075e, 0xe3056a0c, 0x10d25065, 0xcb03a442,
0xe0ec6e0e, 0x1698db3b, 0x4c98a0be, 0x3278e964,
0x9f1f9532, 0xe0d392df, 0xd3a0342b, 0x8971f21e,
0x1b0a7441, 0x4ba3348c, 0xc5be7120, 0xc37632d8,
0xdf359f8d, 0x9b992f2e, 0xe60b6f47, 0x0fe3f11d,
0xe54cda54, 0x1edad891, 0xce6279cf, 0xcd3e7e6f,
0x1618b166, 0xfd2c1d05, 0x848fd2c5, 0xf6fb2299,
0xf523f357, 0xa6327623, 0x93a83531, 0x56cccd02,
0xacf08162, 0x5a75ebb5, 0x6e163697, 0x88d273cc,
0xde966292, 0x81b949d0, 0x4c50901b, 0x71c65614,
0xe6c6c7bd, 0x327a140a, 0x45e1d006, 0xc3f27b9a,
0xc9aa53fd, 0x62a80f00, 0xbb25bfe2, 0x35bdd2f6,
0x71126905, 0xb2040222, 0xb6cbcf7c, 0xcd769c2b,
0x53113ec0, 0x1640e3d3, 0x38abbd60, 0x2547adf0,
0xba38209c, 0xf746ce76, 0x77afa1c5, 0x20756060,
0x85cbfe4e, 0x8ae88dd8, 0x7aaaf9b0, 0x4cf9aa7e,
0x1948c25c, 0x02fb8a8c, 0x01c36ae4, 0xd6ebe1f9,
0x90d4f869, 0xa65cdea0, 0x3f09252d, 0xc208e69f,

0xb74e6132, 0xce77e25b, 0x578fdfe3, 0x3ac372e6,
};
********** TEST VECTORS **********
This is a test vector.
Plaintext is "BLOWFISH".
The key is "abcdefghijklmnopqrstuvwxyz".
#define PL 0x424c4f57l
#define PR 0x46495348l
#define CL 0x324ed0fel
#define CR 0xf413a203l
static char keey[]="abcdefghijklmnopqrstuvwxyz";
This is another test vector.
The key is "Who is John Galt?"
#define PL 0xfedcba98l
#define PR 0x76543210l
#define CL 0xcc91732bl
#define CR 0x8022f684l

Well, that is all for this edition, more in the next update. Till then bye,
Ankit Fadia
ankit@bol.net.in <mailto:ankit@bol.net.in>