USAISEC Technical Report ASQB-OSO-S-TR-92-04

A 24-bit Encryption Algorithm for Linking Protection (Johnson Algorithm)

Eric E. Johnson

March 1992

A 24-Bit Encryption Algorithm for Linking Protection

Eric E. Johnson
New Mexico State University

Abstract

This report describes an unclassified algorithm for the encryption and decryption of 24-bit words which was specifically developed for use in scrambling the transmissions of the high-frequency radio automatic link establishment protocol specified in MIL-STD-188-141A/FED-STD-1045.

Introduction

The purpose of this report is to present an algorithm for encrypting and decrypting 24-bit quantities which may be used to provide some protection against low-level cryptographic threats. It is not intended for use in conventional cryptographic applications (i.e., COMSEC).

The application for which this algorithm was developed is the scrambling of 24-bit automatic link establishment (ALE) words used by high-frequency (HF) radios adhering to MIL-STD-188-141A/FED-STD-1045 [1]. This scrambling is intended to provide "linking protection" by preventing radios which do not have the current key in use by a protected radio from linking with that radio.

In terms of the ISO Open Systems Interconnection reference model [2], linking protection is employed just below the ALE protocol within the data link layer, as shown in Figure 1. The scrambler is designed to maximally distort transmissions which are not encrypted under the current key so that they appear meaningless to the ALE protocol module, and therefore do not interact with it.

The linking protection mechanism is intended to counter two types of attacks: a passive playback attack (e.g., using a tape recorder to capture modem tones and play them back to cause unknown mischief), and active cryptanalysis directed to recovery of the current key. The general linking protection procedure [3] includes time of day (TOD) and frequency inputs to counter playback attacks: identical plaintext ALE words encrypted under the same key will produce different results at different times or on different frequencies.

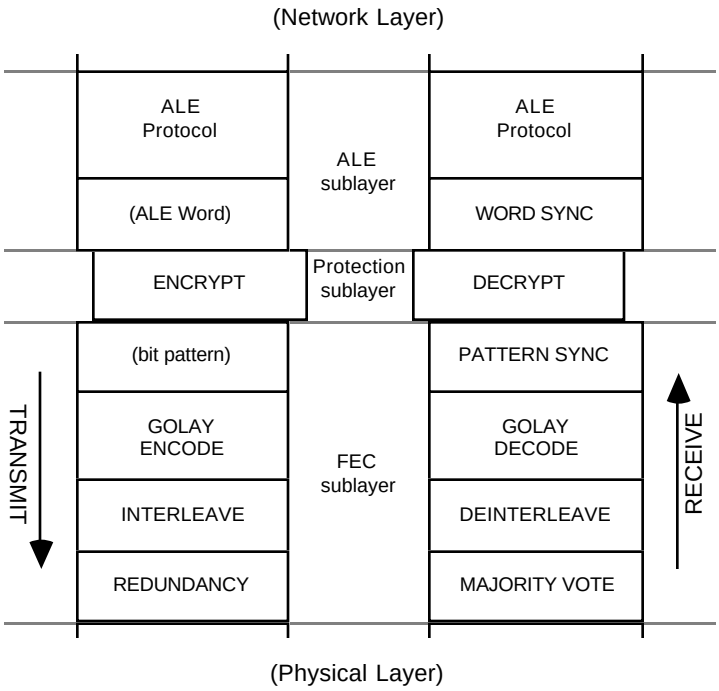

Figure 1: Conceptual Model of MIL-STD-188-141A Data Link Layer

Active attacks on the cryptographic algorithm itself must be countered by the strength of the algorithm and the size of its key. The algorithm specified in this report is not intended to withstand a determined attack by an adversary with unlimited resources, but it should resist attack by individuals with limited computing resources. It satisfies the basic requirement of a "good" algorithm that each ciphertext bit depend upon all bits of the plaintext and all bits of the key, although the degree of dependence is uneven. The key length need only be chosen to render an exhaustive search of the key space unfruitful within the useful life of the key, which is the lesser of the key change interval or the time to recover the key through other cryptanalytic means than exhaustion of the key space.

The Lattice Algorithm

A schematic representation of the algorithm is shown in Figure 2. The algorithm operates on each of the three bytes of the 24-bit word individually. At each step, each byte is exclusive-ored with one or both of the other data bytes, a byte of key, and a byte of TOD, and the result is then translated using a 256x8 bit substitution table ("S-box").

Mathematically, the encryption algorithm works as follows:

1. Let f(•) be an invertible function mapping {0..255} -> {0..255}.

2. Let V be a vector of key variable bytes and S be a vector of TOD/frequency "seed" bytes. Starting with the first byte in each of V and S, perform several "rounds" of the sequence in 4 below, using the next byte from V and S (modulo their lengths) each time a reference to V[ ] and S[ ] is made.

3. Let A be the most significant of the three-byte input to each round of encryption, B be the middle byte, and C be the least significant byte, and A', B', and C' be the corresponding output bytes of each round.

4. Then for each round,
   A' := f(A + B + V[ ] + S[ ])
   C' := f(C + B + V[ ] + S[ ])
   B' := f(A' + B + C' + V[ ] + S[ ])

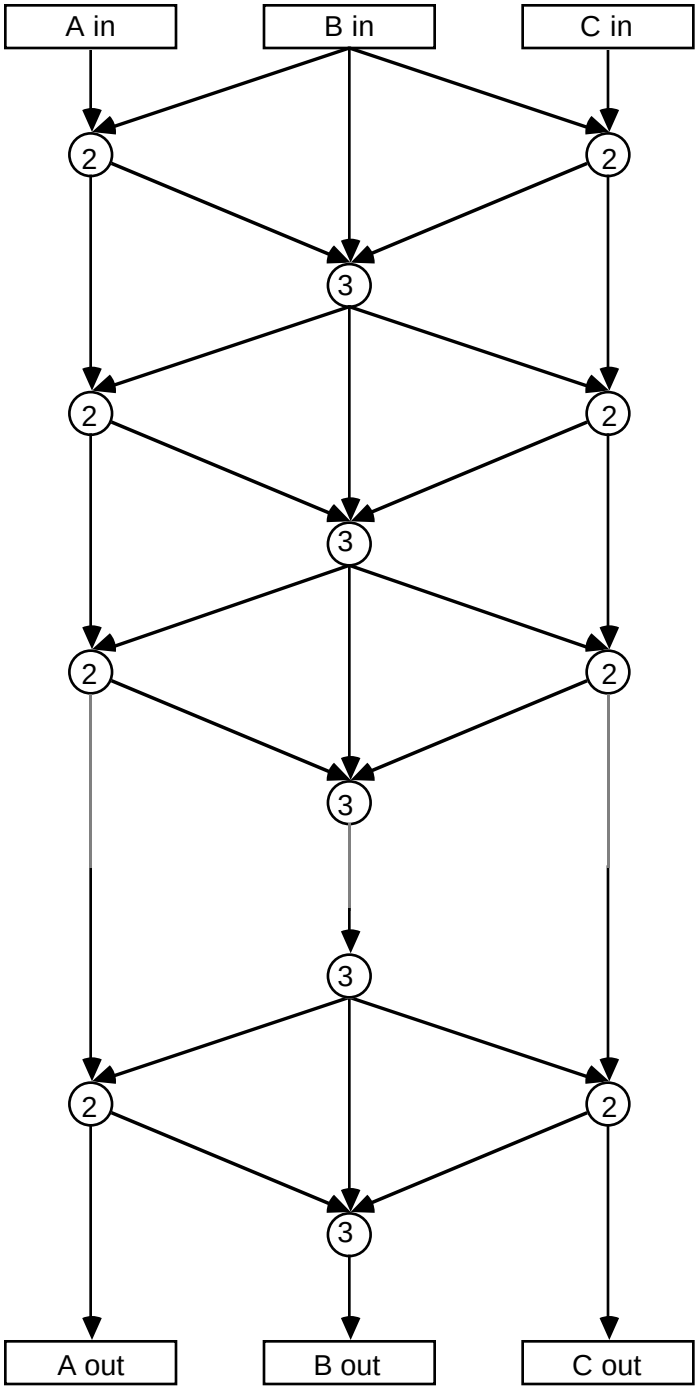

Figure 2: Lattice Algorithm Schematic Diagram (Encryption)

The decryption algorithm simply inverts the encryption algorithm. Note that the starting point in the V and S vectors must be pre-computed, based upon the number of rounds performed, and that the bytes are used in reverse order.

1. Let g(•) be the inverse of the f(•) used for encryption.
2. Starting with the last elements of the V and S vectors used in encryption, perform the same number of rounds of the following decryption steps as were used in encryption, working backward through the V and S vectors.
3. B := g(B') + A' + C' + V[ ] + S[ ]
   C := g(C') + B + V[ ] + S[ ]
   A := g(A') + B + V[ ] + S[ ]

Performance Analysis

In this section we present the results of simulations of the lattice algorithm, showing (a) the average number of ciphertext bits changed for single-bit changes in the plaintext, and the average number of decrypted plaintext bits changed for (b) single-bit changes in the ciphertext, (c) single-bit changes in the key variable, and (d) single-bit changes in the seed, all versus the number of rounds used in encryption.

The results were obtained as follows: a random 56-bit key was generated and used for all of the simulations. The seed chosen represents a date of 16 May, a time of 15:57:34, word = 0, and frequency = 1.755 MHz. The ALE word "<TO> SAM" was encrypted using these inputs and the translation tables shown in Appendix A, and the result was stored for comparison. For part (a), the plaintext bits were inverted one at a time and the resulting bit errors in the resulting ciphertext were noted and averaged over the 24 trials. A similar procedure was followed for parts (b) through (d), counting bit errors in decrypted plaintext. The maximum and minimum number of bit errors for each trial are also shown in the plots on the next two pages.

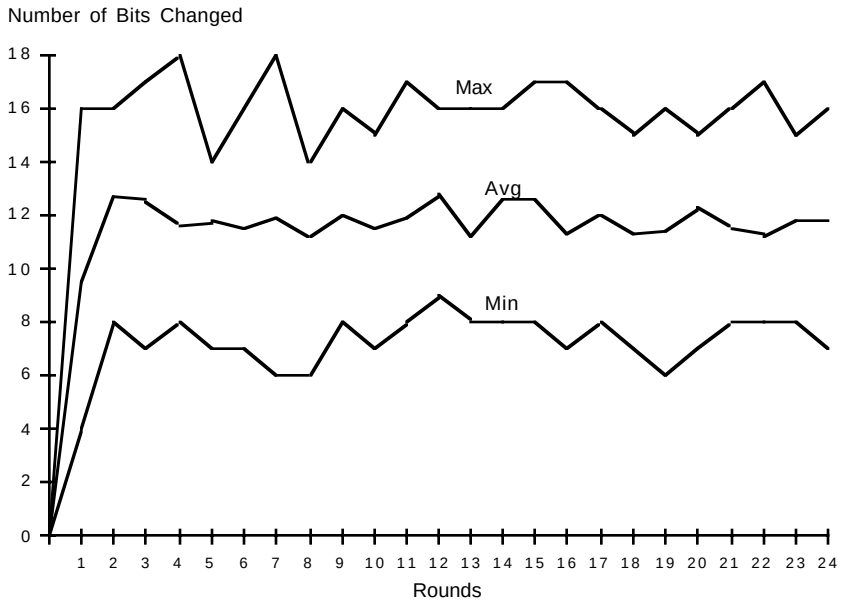

Figure 3: Effect on Ciphertext of Single-Bit Changes in Plaintext

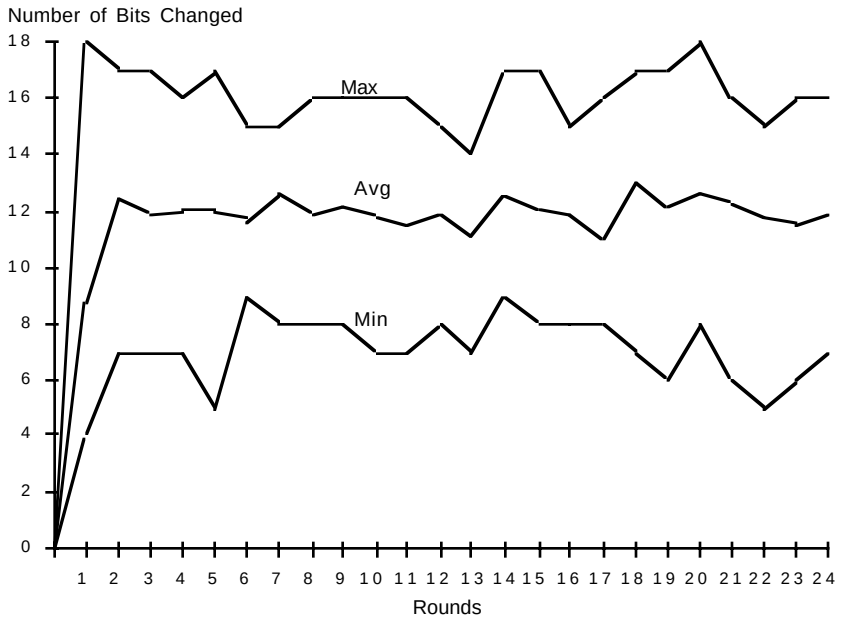

Figure 4: Effect on Plaintext of Single-Bit Changes in Ciphertext

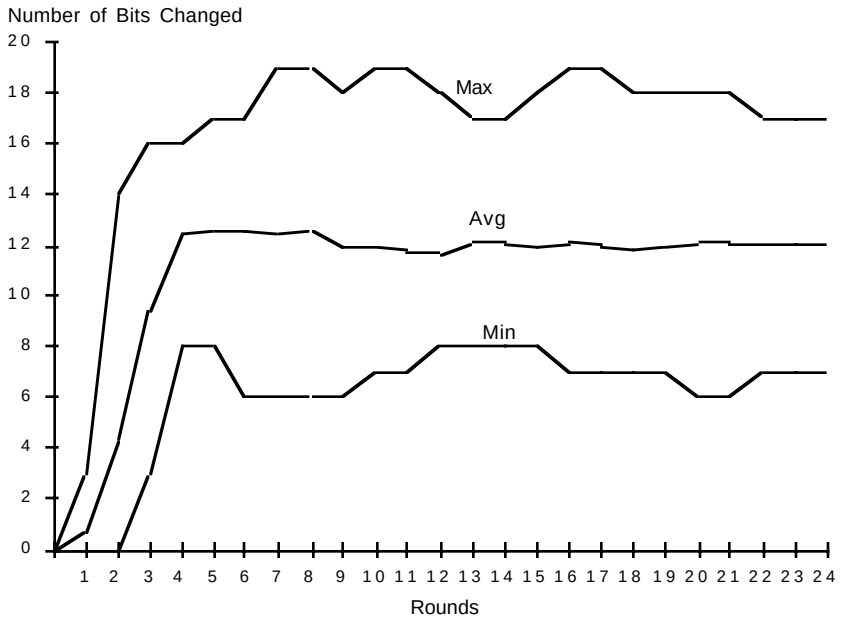

Figure 5: Effect on Ciphertext of Single-Bit Changes in Key

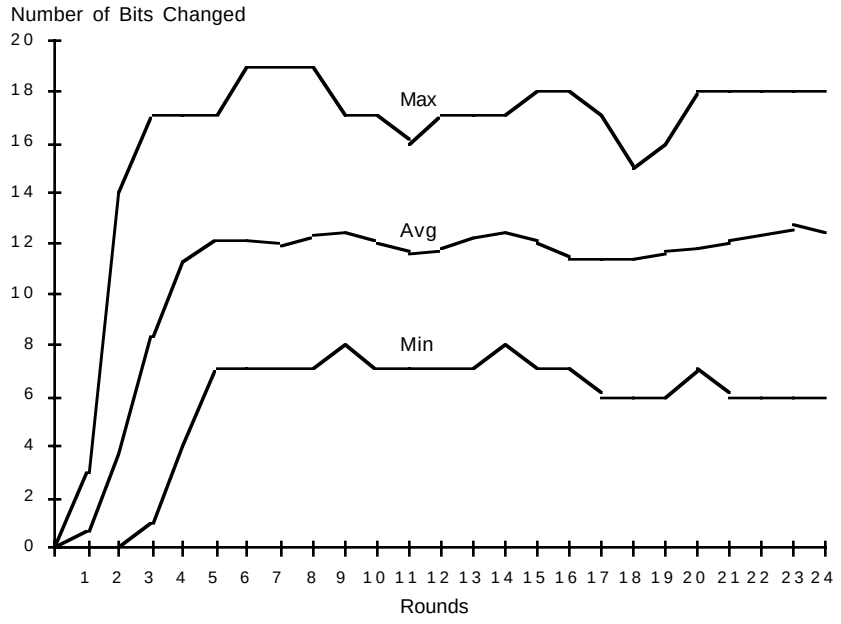

Figure 6: Effect on Plaintext of Single-Bit Changes in Seed

From this rather small series of simulations, it appears that the algorithm achieves its goal of thoroughly scrambling decrypted plaintext given single-bit errors in ciphertext, key, or seed, when run for at least five rounds. To account for possible anomalies due to choice of key and input ALE word (which were not varied here), eight rounds shall be employed when this algorithm is used for linking protection.

Examples

Appendix A contains a set of examples of the use of this algorithm for HF ALE, as described in the Linking Protection Implementation Guide [3]. The seed used reflects a date of 16 May, a time of 15:57:34, and a frequency of 1755 KHz. Three ALE words are encrypted and then decrypted: the word <TO> SAM is encrypted using word number (**w**) = 0, then again using **w** = 1, as specified in [3] for the first words of a call. The word <THIS IS> JOE is then processed using **w** = 2. In each case, the 24-bit results of each round (step) of the encryption process are given.

References

1. **MIL-STD-188-141A**, *Interoperability and Performance Standards for Medium and High Frequency Radio Equipment*, U.S. Army Information Systems Engineering Command, 1988.

2. **ISO 7498-1984**, *Information Processing Systems - Open Systems Interconnection - Basic Reference Model*, American National Standards Institute, 1984.

3. Johnson, E.E., "High Frequency Radio Linking Protection Implementation Guide," New Mexico State University, March 1992.

Appendix A — Examples

Key variable = c2284a1ce7be2f

Number of rounds = 8

seed = 543bd88000017550 (w=0)

Encrypt 54e0cd (<TO> SAM)

Step	A	B	C
0	54	E0	CD
1	D0	72	1D
2	1D	48	3C
3	41	DB	0C
4	98	7C	6D
5	39	10	3D
6	13	AA	E4
7	FC	82	27
8	C0	D7	05

Result: C0D705

Decrypt C0D705

Step	A	B	C
0	C0	D7	05
1	FC	82	27
2	13	AA	E4
3	39	10	3D
4	98	7C	6D
5	41	DB	0C
6	1D	48	3C
7	D0	72	1D
8	54	E0	CD

Result: 54E0CD

seed = 543bd88040017550 (w=1)

Encrypt 54E0CD (<TO> SAM)

Step	A	B	C
0	54	E0	CD
1	D0	72	1D
2	1D	3D	EF
3	E1	F8	6B
4	11	A0	A2
5	6E	32	A0
6	B0	B4	E2
7	CF	CB	11
8	70	84	34

Result: 708434

Decrypt 708434

Step	A	B	C
0	70	84	34
1	CF	CB	11
2	B0	B4	E2
3	6E	32	A0
4	11	A0	A2
5	E1	F8	6B
6	1D	3D	EF
7	D0	72	1D
8	54	E0	CD

Result: 54E0CD

seed = 543bd88080017550 (w=2)

Encrypt b2a7c5 (<TIS> JOE)

Step	A	B	C
0	B2	A7	C5
1	59	47	E6
2	91	BF	83
3	D1	B8	E8
4	53	ED	A9
5	F4	55	9E
6	32	25	FA
7	DD	5D	15
8	28	ED	4A

Result: 28ED4A

Decrypt 28ED4A

Step	A	B	C
0	28	ED	4A
1	DD	5D	15
2	32	25	FA
3	F4	55	9E
4	53	ED	A9
5	D1	B8	E8
6	91	BF	83
7	59	47	E6
8	B2	A7	C5

Result: B2A7C5

Appendix B — Translation Tables

The 256 -> 256 mapping tables for use in the algorithm are given below. To use these tables, use the most significant 4 bits of the input byte to select a row in the table, and the least significant 4 bits to select a column. The output byte is contained at the selected location.

Encryption table f(•)

9c	f2	14	c1	8e	cb	b2	65	97	7a	60	17	92	F9	78	41
07	4c	67	6d	66	4a	30	7d	53	9d	b5	bc	c3	ca	f1	04
03	ec	d0	38	B0	ed	ad	c4	dd	56	42	bd	a0	de	1b	81
55	44	5a	e4	50	DC	43	63	09	5c	74	cf	0e	ab	1d	3d
6b	02	5d	28	e7	c6	ee	b4	d9	7c	19	3e	5e	6c	d6	6e
2a	13	a5	08	b9	2d	BB	a2	d4	96	39	e0	ba	d7	82	33
0d	5f	26	16	fe	22	af	00	11	c8	9e	88	8b	a1	7b	87
27	E6	c7	94	d1	5b	9b	f0	9f	db	e1	8d	d2	1f	6a	90
f4	18	91	59	01	b1	FC	34	3c	37	47	29	e2	64	69	24
0a	2f	73	71	a9	84	8c	a8	a3	3b	E3	E9	58	80	a7	D3
b7	c2	1c	95	1e	4d	4f	4E	fb	76	fd	99	c5	C9	e8	2e
8a	df	f5	49	f3	6f	8f	e5	EB	F6	25	d5	31	c0	57	72
aa	46	68	0b	93	89	83	70	ef	a4	85	f8	0f	b3	AC	10
62	cc	61	40	f7	fa	52	7f	ff	32	45	20	79	ce	ea	be
cd	15	21	23	D8	b6	0c	3f	54	1A	bf	98	48	3a	75	77
2b	ae	36	da	7e	86	35	51	05	12	b8	a6	9a	2C	06	4b

Decryption table g(•)

67	84	41	20	1f	f8	fe	10	53	38	90	c3	e6	60	3c	cc
cf	68	f9	51	02	e1	63	0b	81	4a	E9	2e	a2	3e	a4	7d
db	e2	65	E3	8f	ba	62	70	43	8b	50	f0	Fd	55	af	91
16	bc	D9	5f	87	F6	F2	89	23	5a	ed	99	88	3f	4b	e7
d3	0f	2a	36	31	da	c1	8a	ec	b3	15	ff	11	a5	A7	a6
34	f7	d6	18	e8	30	29	BE	9c	83	32	75	39	42	4c	61
0a	d2	d0	37	8d	07	14	12	c2	8e	7e	40	4d	13	4f	b5
c7	93	bf	92	3a	EE	a9	ef	0e	dc	09	6e	49	17	f4	d7
9d	2f	5e	c6	95	ca	F5	6f	6b	c5	b0	6c	96	7b	04	b6
7F	82	0c	c4	73	a3	59	08	EB	ab	fc	76	00	19	6a	78
2c	6d	57	98	c9	52	fb	9e	97	94	c0	3d	CE	26	f1	66
24	85	06	cd	47	1a	e5	a0	fa	54	5c	56	1b	2b	df	ea
bd	03	a1	1c	27	ac	45	72	69	AD	1d	05	d1	e0	dd	3b
22	74	7c	9F	58	bb	4e	5d	E4	48	f3	79	35	28	2d	b1
5b	7a	8c	9A	33	b7	71	44	ae	9B	de	B8	21	25	46	c8
77	1e	01	b4	80	b2	B9	d4	cb	0D	d5	a8	86	aa	64	d8

Appendix C — Program Listing for Lattice Algorithm

```
/*  lattice.c  eej  11/11/89              */
/*  encrypt and decrypt routines for      */
/*  24 bit words using lattice algorithm  */

long e2(x, y)
long x, y;
{
    long temp;

    temp = f[(x ^ y ^ v[vptr] ^ s[sptr]) & 0x0ff];
    vptr = (++vptr % vsize);
    sptr = (++sptr % ssize);
    return(temp);
}

long e3(x, y, z)
long x, y, z;
{
    long temp;

    temp = f[ x ^ y ^ z ^ v[vptr] ^ s[sptr] ];
    vptr = (++vptr % vsize);
    sptr = (++sptr % ssize);
    return(temp);
}

void e_step(x, y, z)
long *x, *y, *z;
{
    *x = e2(*x, *y);
    *z = e2(*z, *y);
    *y = e3(*x, *y, *z);
}

long encrypt(w, rounds)
long w, rounds;
{
    long a, b, c;
    long i;

    vptr = sptr = 0;
    a = (w & 0x0ff0000) >> 16;  /* most sig byte  */
    b = (w & 0x000ff00) >> 8;   /* middle byte    */
    c = (w & 0x00000ff);        /* least sig byte */

    for (i=0; i < rounds; i++)
      e_step(&a, &b, &c);

    w = ((a & 0x0ff)<<16) | ((b & 0x0ff)<<8) | (c & 0x0ff);
    return(w);
}
```

```
long d2(x, y)
long x, y;
{
    long temp;

    vptr = vptr ? (--vptr % vsize) : vsize-1;
    sptr = sptr ? (--sptr % ssize) : ssize-1;
    temp = (g[x] ^ y ^ v[vptr] ^ s[sptr]);
    return(temp);
}

long d3(x, y, z)
long x, y, z;
{
    long temp;

    vptr = vptr ? (--vptr % vsize) : vsize-1;
    sptr = sptr ? (--sptr % ssize) : ssize-1;
    temp = (g[y] ^ x ^ z ^ v[vptr] ^ s[sptr]);
    return(temp);
}

void d_step(x, y, z)
long *x, *y, *z;
{
    *y = d3(*x, *y, *z);
    *z = d2(*z, *y);
    *x = d2(*x, *y);
}

long decrypt(w, rounds)
long w, rounds;
{
    long a, b, c;
    int i;
    a = (w & 0x0ff0000) >> 16;  /* most sig byte  */
    b = (w & 0x000ff00) >> 8;   /* middle byte    */
    c = (w & 0x00000ff);        /* least sig byte */

    vptr = (3 * rounds) % vsize; /* set pointers to positions */
    sptr = (3 * rounds) % ssize; /* at end of encryption      */

    for (i=0; i < rounds; i++)
      d_step(&a, &b, &c);

    w = ((a & 0x0ff)<<16) | ((b & 0x0ff)<<8) | (c & 0x0ff);
    return(w);
}
```

```
/*  crypto24.c      eej     11/11/89            */
/*  uses translation tables (256 -> 256) stored  */
/*  externally in files f256 and g256.          */

#include <stdio.h>

#define vsize 7
#define ssize 8

int v[vsize],        /* variable */
    s[ssize],        /*  seed    */
    f[256],          /* encrypt table */
    g[256];          /* decrypt table */

int vptr, sptr;       /* pointers into tables */

#include "lattice.c"

int asc2hex(a)
int a;
{
    switch (a) {
      case '0': return (0);
      case '1': return (1);
      case '2': return (2);
      case '3': return (3);
      case '4': return (4);
      case '5': return (5);
      case '6': return (6);
      case '7': return (7);
      case '8': return (8);
      case '9': return (9);
      case 'A': return (10);
      case 'B': return (11);
      case 'C': return (12);
      case 'D': return (13);
      case 'E': return (14);
      case 'F': return (15);
      case 'a': return (10);
      case 'b': return (11);
      case 'c': return (12);
      case 'd': return (13);
      case 'e': return (14);
      case 'f': return (15);
      default: exit();
    }
}

int pack(a, b)
char a, b;
{
    return ((asc2hex(a) << 4) | (asc2hex(b)));
}
```

```
void getvar(v)
int *v;
{
    int i;
    char temp[80];

    printf("Enter variable (hex):  ");
    scanf("%s", temp);
    for (i = 0; i < vsize<<1; i += 2) {
        if (temp[i] && temp[i+1]) v[i>>1] = pack(temp[i], temp[i+1]);
        else for (; i < vsize<<1; i += 2) v[i>>1] = 0;
    }
}

void getseed(s)
int *s;
{
    int i, j;
    char temp[80];

    printf("Enter seed (hex):  ");
    scanf("%s", temp);
    for (i = 0; i < ssize<<1; i += 2) {
        if (temp[i] && temp[i+1]) s[i>>1] = pack(temp[i], temp[i+1]);
        else for (; i < ssize<<1; i += 2) s[i>>1] = 0;
    }
}

main()
{
    long pt, ct, rounds;
    FILE *fd;
    int i;

/*****     read in f table for encryption     *****/
    if ((fd = fopen("f256","r")) == NULL) exit(-1);
    for (i=0; i<256; i++)
        fscanf(fd, "%x", &f[i]);
    fclose(fd);

/*****     read in g table for decryption     *****/
    if ((fd = fopen("g256","r")) == NULL) exit(-2);
    for (i=0; i<256; i++)
        fscanf(fd, "%x", &g[i]);
    fclose(fd);

    printf("Enter number of rounds:  ");
    scanf("%ld", &rounds);
    getvar(v);
    getseed(s);
    do {
        printf("Enter word to encrypt (hex):  ");
        scanf("%6lx", &pt);
        printf("Encrypted:  %lx\n", ct = encrypt(pt, rounds));
        printf("Decrypted:  %lx\n", decrypt(ct, rounds));
    } while (pt != 0);
}
```