The MacGuffin Block Cipher Algorithm

Matt Blaze
AT&T Bell Laboratories
101 Crawfords Corner Road, Holmdel, NJ 07733 USA
mab@research.att.com

Bruce Schneier
Counterpane Systems
730 Fair Oaks Avenue, Oak Park, IL 70302 USA
schneier@chinet.com

Abstract. This paper introduces MacGuffin, a 64 bit "codebook" block cipher. Many of its characteristics (block size, application domain, performance and implementation structure) are similar to those of the U.S. Data Encryption Standard (DES). It is based on a Feistel network, in which the cleartext is split into two sides with one side repeatedly modified according to a keyed function of the other. Previous block ciphers of this design, such as DES, operate on equal length sides. MacGuffin is unusual in that it is based on a *generalized unbalanced Feistel network (GUFN)* in which each round of the cipher modifies only 16 bits according to a function of the other 48. We describe the general characteristics of MacGuffin architecture and implementation and give a complete specification for the 32-round, 128-bit key version of the cipher.

1 Introduction

Feistel ciphers [1] operate by alternately encrypting the bits in one "side" of their input based on a keyed non-linear function of the bits in the other. This is done repeatedly, for a fixed number of "rounds". It is believed that, when iterated over sufficiently many rounds, even relatively simple non-linear functions can provide high security. Traditionally, such ciphers split their input block evenly about the middle; a 64 bit cipher would operate on two 32 bit internal blocks, swapping the "left" (the *target block*) and "right" (the *control block*) sides with each round. Several important block ciphers, including DES [3], are built upon this structure. We say these ciphers are based on *balanced Feistel networks (BFNs)*, since both sides are of equal length.

This paper describes a block cipher, called *MacGuffin,* that is based on a variant of this structure, the *generalized unbalanced Feistel network (GUFN)*, in which the target and control blocks need not be of equal length[1]. GUFNs, especially those in which the target block is smaller than the control block, appear to have a number of attractive properties for cipher design, particularly

[1] Several cryptographic hash functions, such as MD5 [6] and SHA [5], employ an unbalanced structure similar in some respects to a GUFN.

with respect to the design of the non-linear function. The principles underlying GUFNs are discussed in [7].

As its name suggests, MacGuffin is intended primarily as a catalyst for discussion and analysis. We believe it may also prove a practical, high security block cipher suitable for general use as an alternative to DES. It operates on 64 bit blocks of data, with an internal structure containing a 16 bit target block and a 48 bit control block ("48 on 16", in the notation of [7]). In principle, almost any length key and any number of rounds may be used, although we specify 32 rounds and a 128 bit key as "standard".

2 Architecture

We have been conservative in most aspects of MacGuffin's design, isolating most of its novel features to those parts of the design related to its unbalanced structure. As such, much of our design is adapted directly from DES. We hope that the many similarities between DES and MacGuffin will invite analysis of their differences.

Basically, the input cipherblock is partitioned into four 16 bit words, from left to right. In each round, the three rightmost words comprise the control block and are bitwise exclusive-ORed (XORed) with three words derived from the key. These 48 bits are then split eight ways according to a fixed permutation to provide input to eight functions of six bits (the "S-boxes"), each producing two bits of output. The 16 S-box output bits are then XORed, according to another fixed permutation, with the bits in the leftmost (target) word. Finally, the leftmost word is rotated into the rightmost position. The cipher can be reversed by a similar process, with the key derived bits applied in reverse order.

2.1 Design Principles

Because each round operates on only half as many bits as in a BFN (16 as opposed to 32), we use 32 rounds, twice as many as in DES, in our standard version. Because there are twice as many rounds, however, there are also a total of twice as many key bits XORed with the control blocks. These bits are obtained from the 128 bit key with the key expansion function described in the next section.

We adapt our S-boxes directly from those of DES. The eight DES S-boxes each produce four bits of output. Since we require only two bits from each (for a total of 16 bits), we use only the "outer" two output bits from each S-box.

In each round, each control block bit is XORed with one derived key bit and provides one input to exactly one S-box. There is no "expansion" permutation, since the number of control bits equals the number of S-box inputs. The control bits are mapped $1:1$ to S-box inputs according to a fixed permutation. This permutation was designed so that each S-box receives two of its six inputs from each of the three registers in the control block.

S-box outputs are distributed across the 16 target bits. No S-box output goes to a bit position that is used as a direct input to itself in the next four rounds.

Observe that each of the three control registers contains bits produced in a different round of the cipher, and that each encrypted bit provides input to three different S-boxes (in the next three rounds), before it is encrypted again.

The cipher is designed for implementation in either hardware or software. Permutations were chosen to minimize the number of shift and mask operations and to allow time/memory optimizations in a software implementation.

3 Algorithm Description

3.1 Data Structures and Notation

We use the following notation:

$\oplus$ represents a 16 bit bitwise exclusive-OR (XOR) operation.
$\leftarrow$ is the conventional assignment operator, except as noted below.
$w, x, y, z \leftarrow i$ copies the data from 64 bit interface i, from lowest to highest bit position, into 16 bit registers w, x, y and z, respectively.
$i \leftarrow w, x, y, z$ copies the bits from 16 bit registers w, x, y and z, respectively into interface i, from lowest to highest bit position.
$s, t, u, v \leftarrow w, x, y, z$ copies w, x, y and z to s, t, u and v, respectively, in parallel (e.g., $x, y \leftarrow y, x$ swaps x and y).
$w \Leftarrow F(x, y, z)$ selects, according to a fixed permutation, bits from x, y and z as input to function F, storing the function output in bits of w, selected according to a fixed permutation.

The cipher employs the following internal structures:

$I_{0...63}, O_{0...63}$ are the 64 bit external input and output interfaces.
left, a, b, c, t are 16 bit registers on which all cryptographic operations are performed. r_0 represents the least significant bit of r, r_{15} the most significant.
$k_{0...127}$ is a 128 bit secret key parameter.
$K[0...31, 0...2]$ is a 32×3 table of 16 bit words containing an expansion of k, as explained below.

3.2 S-boxes and Permutations

Nonlinearity in the encryption and key setup processes is provided primarily through eight functions, or "S-boxes", denoted $S_1 ... S_8$, each taking six bits of input selected from the a, b and c registers and producing two bits of output (which are XORed into the *left* register).

Inputs to each S-box are selected uniquely from the a, b and c registers, as specified in Table 1. (In this table, input bit 0 is the least significant bit.) Outputs from each S-box are distributed across the 16 bit target block as specified in Table 2. Each S-box is defined as a 64×2 bit mapping of input values to outputs, as given in Table 3.

S-box	Input Bit					
	0	1	2	3	4	5
S_1	a_2	a_5	b_6	b_9	c_{11}	c_{13}
S_2	a_1	a_4	b_7	b_{10}	c_8	c_{14}
S_3	a_3	a_6	b_8	b_{13}	c_0	c_{15}
S_4	a_{12}	a_{14}	b_1	b_2	c_4	c_{10}
S_5	a_0	a_{10}	b_3	b_{14}	c_6	c_{12}
S_6	a_7	a_8	b_{12}	b_{15}	c_1	c_5
S_7	a_9	a_{15}	b_5	b_{11}	c_2	c_7
S_8	a_{11}	a_{13}	b_0	b_4	c_3	c_9

Table 1. S-Box Input Permutation

S-box	Output Bit	
	0	1
S_1	t_0	t_1
S_2	t_2	t_3
S_3	t_4	t_5
S_4	t_6	t_7
S_5	t_8	t_9
S_6	t_{10}	t_{11}
S_7	t_{12}	t_{13}
S_8	t_{14}	t_{15}

Table 2. S-Box Output Permutation

3.3 Key Setup

Each round of the cipher uses the secret key parameter to perturb the S-boxes by bitwise XOR against the S-box inputs. Each round thus requires 48 key bits. To convert the 128 bit k parameter to a sequence of 48 bit values for each round (the K table), MacGuffin uses an iterated version of its own block encryption function. See Figure 1.

3.4 Block Encryption

Block encryption is defined in Figure 2.

3.5 Block Decryption

Block decryption is similar to block encryption, and is defined in Figure 3.

$K \leftarrow 0$
$left, a, b, c \leftarrow k_{0\ldots63}$
for $h = 0$ **to** 31 **do**
    for $i = 0$ **to** 31 **do**
        for $j = 1$ **to** 8 **do**
            $t \Leftarrow S_j(a \oplus K[i,0], b \oplus K[i,1], c \oplus K[i,2])$
        $left \leftarrow left \oplus t$
        $left, a, b, c \leftarrow a, b, c, left$
    $K[h,0] \leftarrow left$
    $K[h,1] \leftarrow a$
    $K[h,2] \leftarrow b$
$left, a, b, c \leftarrow k_{64\ldots127}$
for $h = 0$ **to** 31 **do**
    for $i = 0$ **to** 31 **do**
        for $j = 1$ **to** 8 **do**
            $t \Leftarrow S_j(a \oplus K[i,0], b \oplus K[i,1], c \oplus K[i,2])$
        $left \leftarrow left \oplus t$
        $left, a, b, c \leftarrow a, b, c, left$
    $K[h,0] \leftarrow K[h,0] \oplus left$
    $K[h,1] \leftarrow K[h,1] \oplus a$
    $K[h,2] \leftarrow K[h,2] \oplus b$

Fig. 1. MacGuffin Key Setup

$left, a, b, c \leftarrow I$
for $i = 0$ **to** 31 **do**
    for $j = 1$ **to** 8 **do**
        $t \Leftarrow S_j(a \oplus K[i,0], b \oplus K[i,1], c \oplus K[i,2])$
    $left \leftarrow left \oplus t$
    $left, a, b, c \leftarrow a, b, c, left$
$O \leftarrow left, a, b, c$

Fig. 2. MacGuffin Block Encryption

$c, left, a, b \leftarrow I$
for $i = 31$ **downto** 0 **do**
    for $j = 1$ **to** 8 **do**
        $t \Leftarrow S_j(a \oplus K[i,0], b \oplus K[i,1], c \oplus K[i,2])$
    $left \leftarrow left \oplus t$
    $left, a, b, c \leftarrow c, left, a, b$
$O \leftarrow left, a, b, c$

Fig. 3. MacGuffin Block Decryption

S_1

2	0	0	3	3	1	1	0	0	2	3	0	3	3	2	1	1	2	2	0	0	2	2	3	1	3	3	1	0	1	1	2
0	3	1	2	2	2	2	0	3	0	0	3	0	1	3	1	3	1	2	3	3	1	1	2	1	2	2	0	1	0	0	3

S_2

3	1	1	3	2	0	2	1	0	3	3	0	1	2	0	2	3	2	1	0	0	1	3	2	2	0	0	3	1	3	2	1
0	3	2	2	1	2	3	1	2	1	0	3	3	0	1	0	1	3	2	0	2	1	0	2	3	0	1	1	0	2	3	3

S_3

2	3	0	1	3	0	2	3	0	1	1	0	3	0	1	2	1	0	3	2	2	1	1	2	3	2	0	3	0	3	2	1
3	1	0	2	0	3	3	0	2	0	3	3	1	2	0	1	3	0	1	3	0	2	2	1	1	3	2	1	2	0	1	2

S_4

1	3	3	2	2	3	1	1	0	0	0	3	3	0	2	1	1	0	0	1	2	0	1	2	3	1	2	2	0	2	3	3
2	1	0	3	3	0	0	0	2	2	3	1	1	3	3	2	3	3	1	0	1	1	2	3	1	2	0	1	2	0	0	2

S_5

0	2	2	3	0	0	1	2	1	0	2	1	3	3	0	1	2	1	1	0	1	3	3	2	3	1	0	3	2	2	3	0
0	3	0	2	1	2	3	1	2	1	3	2	1	0	2	3	3	0	3	3	2	0	1	3	0	2	1	0	0	1	2	1

S_6

2	2	1	3	2	0	3	0	3	1	0	2	0	3	2	1	0	0	3	1	1	3	0	2	2	0	1	3	1	1	3	2
3	0	2	1	3	0	1	2	0	3	2	1	2	3	1	2	1	3	0	2	0	1	2	1	1	0	3	0	3	2	0	3

S_7

0	3	3	0	0	3	2	1	3	0	0	3	2	1	3	2	1	2	2	1	3	1	1	2	1	0	2	3	0	2	1	0
1	0	0	3	3	3	3	2	2	1	1	0	1	2	2	1	2	3	3	1	0	0	2	3	0	2	1	0	3	1	0	2

S_8

3	1	0	3	2	3	0	2	0	2	3	1	3	1	1	0	2	2	3	1	1	0	2	3	1	0	0	2	2	3	1	0
1	0	3	1	0	2	1	1	3	0	2	2	2	2	0	3	0	3	0	2	2	3	3	0	3	1	1	1	1	0	2	3

Table 3. MacGuffin S-Boxes

4 Implementation, Performance and Applications

Feistel ciphers, with their many permutation operations and table lookups, are particularly well suited to hardware implementation. Because permutations in hardware are "free" (they are implemented with simple connections), and because S-box lookups can occur in parallel, each round can be implemented with conventional modern hardware in two clock cycles.

Software implementations of Feistel ciphers on general-purpose computers are typically much slower than their hardware counterparts, since the S-boxes must be evaluated in sequence and bit permutations must be simulated with

shifts, ANDs, ORs and other operators. Depending on the specific permutations and S-box structures, however, many of these operations can be made faster with table lookups and by combining several operations into one.

The permutations in MacGuffin have been designed explicitly to permit software optimization. First, the six inputs to each S-box are from different bits from each of the a, b and c registers, allowing the three registers to be masked and ORed together (without individual shifting) for a single lookup for each S-box. Furthermore, for each S-box there is a unique "mate" S-box with which it shares no common inputs. This allows the eight S-boxes to be "paired off" and looked up two at a time with a single 2^{16} entry table containing the combined outputs of both S-boxes. (The pairs are S_1S_2, S_3S_4, S_5S_7 and S_6S_8).

An optimized software implementation (given in the Appendix) of 32 round MacGuffin runs at close to the speed of optimized 16 round DES in software. An implementation on a 486/66 processor has a bandwidth of about 1.5Mbps; a reasonable DES implementation [2] on the same processor runs at 2.1Mbps.

The MacGuffin interface is similar to that of DES (except for the larger keyspace). It can be used with the standard "FIPS-81" modes of operation[4]. Note that key setup is an explicitly time consuming process. This is intended to discourage exhaustive search of poorly chosen keys. In an implementation where rapid selection among many keys is required (such as a packet-based network security protocol) the 1536 bit expanded key may be passed directly as the cryptovariable.

Experiments with MacGuffin are detailed in [7].

While we believe the GUFN structure is superior to the conventional BFN cipher structure, much more discussion and analysis is required before we can recommend its use for protecting sensitive data. We encourage attacks against MacGuffin in particular and the GUFN structure in general.

References

1. H. Feistel. Cryptography and Computer Privacy. *Scientific American,* May 1973.
2. J. Lacy, D.P. Mitchell, and W.M. Schell. CryptoLib: Cryptography in Software. *Proceedings of USENIX Security Symposium IV,* October 1993.
3. National Bureau of Standards. Data Encryption Standard, *Federal Information Processing Standards Publication 46*, US Government Printing Office, Washington, D.C., 1977.
4. National Bureau of Standards. Data Encryption Standard Modes of Operation, *Federal Information Processing Standards Publication 81,* US Government Printing Office, Washington, D.C., 1980.
5. National Institute for Standards and Technology. Secure Hash Standard. *Federal Information Processing Standard Publication 180,* US Government Printing Office, April 1993.
6. R. Rivest. The MD5 Message Digest Algorithm. *RFC 1321,* IETF, April 1992.
7. B. Schneier and M. Blaze. Unbalanced Feistel Network Block Ciphers. *To appear,* 1994.

Appendix: Optimized C Language Implementation

```
/*
 * MacGuffin Cipher
 * 10/3/94 - Matt Blaze
 * (fast, unrolled version)
 */

#define ROUNDS 32
#define KSIZE (ROUNDS*3)

/* expanded key structure */
typedef struct mcg_key {
  unsigned short val[KSIZE];
} mcg_key;

#define TSIZE (1<<16)

/* the 8 s-boxes, expanded to put the output bits in the right
 * places.  note that these are the des s-boxes (in left-right,
 * not canonical, order), but with only the "outer" two output
 * bits. */
unsigned short sboxes[8][64] = {
/* 0 (S1) */
 {0x0002, 0x0000, 0x0000, 0x0003, 0x0003, 0x0001, 0x0001, 0x0000,
  0x0000, 0x0002, 0x0003, 0x0000, 0x0003, 0x0003, 0x0002, 0x0001,
  0x0001, 0x0002, 0x0002, 0x0000, 0x0000, 0x0002, 0x0002, 0x0003,
  0x0001, 0x0003, 0x0003, 0x0001, 0x0000, 0x0001, 0x0001, 0x0002,
  0x0000, 0x0003, 0x0001, 0x0002, 0x0002, 0x0002, 0x0002, 0x0000,
  0x0003, 0x0000, 0x0000, 0x0003, 0x0000, 0x0001, 0x0003, 0x0001,
  0x0003, 0x0001, 0x0002, 0x0003, 0x0003, 0x0001, 0x0001, 0x0002,
  0x0001, 0x0002, 0x0002, 0x0000, 0x0001, 0x0000, 0x0000, 0x0003},
/* 1 (S2) */
 {0x000c, 0x0004, 0x0004, 0x000c, 0x0008, 0x0000, 0x0008, 0x0004,
  0x0000, 0x000c, 0x000c, 0x0000, 0x0004, 0x0008, 0x0000, 0x0008,
  0x000c, 0x0008, 0x0004, 0x0000, 0x0000, 0x0004, 0x000c, 0x0008,
  0x0008, 0x0000, 0x0000, 0x000c, 0x0004, 0x000c, 0x0008, 0x0004,
  0x0000, 0x000c, 0x0008, 0x0008, 0x0004, 0x0008, 0x000c, 0x0004,
  0x0008, 0x0004, 0x0000, 0x000c, 0x000c, 0x0000, 0x0004, 0x0000,
  0x0004, 0x000c, 0x0008, 0x0000, 0x0008, 0x0004, 0x0000, 0x0008,
  0x000c, 0x0000, 0x0004, 0x0004, 0x0000, 0x0008, 0x000c, 0x000c},
/* 2 (S3) */
 {0x0020, 0x0030, 0x0000, 0x0010, 0x0030, 0x0000, 0x0020, 0x0030,
  0x0000, 0x0010, 0x0010, 0x0000, 0x0030, 0x0000, 0x0010, 0x0020,
  0x0010, 0x0000, 0x0030, 0x0020, 0x0020, 0x0010, 0x0010, 0x0020,
  0x0030, 0x0020, 0x0000, 0x0030, 0x0000, 0x0030, 0x0020, 0x0010,
```

```
  0x0030, 0x0010, 0x0000, 0x0020, 0x0000, 0x0030, 0x0030, 0x0000,
  0x0020, 0x0000, 0x0030, 0x0030, 0x0010, 0x0020, 0x0000, 0x0010,
  0x0030, 0x0000, 0x0010, 0x0030, 0x0000, 0x0020, 0x0020, 0x0010,
  0x0010, 0x0030, 0x0020, 0x0010, 0x0020, 0x0000, 0x0010, 0x0020},
/* 3 (S4) */
 {0x0040, 0x00c0, 0x00c0, 0x0080, 0x0080, 0x00c0, 0x0040, 0x0040,
  0x0000, 0x0000, 0x0000, 0x00c0, 0x00c0, 0x0000, 0x0080, 0x0040,
  0x0040, 0x0000, 0x0000, 0x0040, 0x0080, 0x0000, 0x0040, 0x0080,
  0x00c0, 0x0040, 0x0080, 0x0080, 0x0000, 0x0080, 0x00c0, 0x00c0,
  0x0080, 0x0040, 0x0000, 0x00c0, 0x00c0, 0x0000, 0x0000, 0x0000,
  0x0080, 0x0080, 0x00c0, 0x0040, 0x0040, 0x00c0, 0x00c0, 0x0080,
  0x00c0, 0x00c0, 0x0040, 0x0000, 0x0040, 0x0040, 0x0080, 0x00c0,
  0x0040, 0x0080, 0x0000, 0x0040, 0x0080, 0x0000, 0x0000, 0x0080},
/* 4 (S5) */
 {0x0000, 0x0200, 0x0200, 0x0300, 0x0000, 0x0000, 0x0100, 0x0200,
  0x0100, 0x0000, 0x0200, 0x0100, 0x0300, 0x0300, 0x0000, 0x0100,
  0x0200, 0x0100, 0x0100, 0x0000, 0x0100, 0x0300, 0x0300, 0x0200,
  0x0300, 0x0100, 0x0000, 0x0300, 0x0200, 0x0200, 0x0300, 0x0000,
  0x0000, 0x0300, 0x0000, 0x0200, 0x0100, 0x0200, 0x0300, 0x0100,
  0x0200, 0x0100, 0x0300, 0x0200, 0x0100, 0x0000, 0x0200, 0x0300,
  0x0300, 0x0000, 0x0300, 0x0300, 0x0200, 0x0000, 0x0100, 0x0300,
  0x0000, 0x0200, 0x0100, 0x0000, 0x0000, 0x0100, 0x0200, 0x0100},
/* 5 (S6) */
 {0x0800, 0x0800, 0x0400, 0x0c00, 0x0800, 0x0000, 0x0c00, 0x0000,
  0x0c00, 0x0400, 0x0000, 0x0800, 0x0000, 0x0c00, 0x0800, 0x0400,
  0x0000, 0x0000, 0x0c00, 0x0400, 0x0400, 0x0c00, 0x0000, 0x0800,
  0x0800, 0x0000, 0x0400, 0x0c00, 0x0400, 0x0400, 0x0c00, 0x0800,
  0x0c00, 0x0000, 0x0800, 0x0400, 0x0c00, 0x0000, 0x0400, 0x0800,
  0x0000, 0x0c00, 0x0800, 0x0400, 0x0800, 0x0c00, 0x0400, 0x0800,
  0x0400, 0x0c00, 0x0000, 0x0800, 0x0000, 0x0400, 0x0800, 0x0400,
  0x0400, 0x0000, 0x0c00, 0x0000, 0x0c00, 0x0800, 0x0000, 0x0c00},
/* 6 (S7) */
 {0x0000, 0x3000, 0x3000, 0x0000, 0x0000, 0x3000, 0x2000, 0x1000,
  0x3000, 0x0000, 0x0000, 0x3000, 0x2000, 0x1000, 0x3000, 0x2000,
  0x1000, 0x2000, 0x2000, 0x1000, 0x3000, 0x1000, 0x1000, 0x2000,
  0x1000, 0x0000, 0x2000, 0x3000, 0x0000, 0x2000, 0x1000, 0x0000,
  0x1000, 0x0000, 0x0000, 0x3000, 0x3000, 0x3000, 0x3000, 0x2000,
  0x2000, 0x1000, 0x1000, 0x0000, 0x1000, 0x2000, 0x2000, 0x1000,
  0x2000, 0x3000, 0x3000, 0x1000, 0x0000, 0x0000, 0x2000, 0x3000,
  0x0000, 0x2000, 0x1000, 0x0000, 0x3000, 0x1000, 0x0000, 0x2000},
/* 7 (S8) */
 {0xc000, 0x4000, 0x0000, 0xc000, 0x8000, 0xc000, 0x0000, 0x8000,
  0x0000, 0x8000, 0xc000, 0x4000, 0xc000, 0x4000, 0x4000, 0x0000,
  0x8000, 0x8000, 0xc000, 0x4000, 0x4000, 0x0000, 0x8000, 0xc000,
  0x4000, 0x0000, 0x0000, 0x8000, 0x8000, 0xc000, 0x4000, 0x0000,
```

```
  0x4000, 0x0000, 0xc000, 0x4000, 0x0000, 0x8000, 0x4000, 0x4000,
  0xc000, 0x0000, 0x8000, 0x8000, 0x8000, 0x8000, 0x0000, 0xc000,
  0x0000, 0xc000, 0x0000, 0x8000, 0x8000, 0xc000, 0xc000, 0x0000,
  0xc000, 0x4000, 0x4000, 0x4000, 0x4000, 0x0000, 0x8000, 0xc000}
};

/* table of s-box outputs, expanded for 16 bit input.
 * this one table includes all 8 sboxes - just mask off
 * the output bits not in use. */
unsigned short stable[TSIZE];

/* we exploit two features of the s-box input & output perms -
 * first, each s-box uses as input two different bits from each
 * of the three registers in the right side, and, second,
 * for each s-box there is another-sbox with no common input bits
 * between them. therefore we can lookup two s-box outputs in one
 * probe of the table.  just mask off the approprate input bits
 * in the table below for each of the three registers and OR
 * together for the table lookup index.
 * these masks are also available below in #defines, for better
 * lookup speed in unrolled loops. */
unsigned short lookupmasks[4][3] = {
  /* a   ,   b   ,   c    */
  {0x0036, 0x06c0, 0x6900},  /* s1+s2 */
  {0x5048, 0x2106, 0x8411},  /* s3+s4 */
  {0x8601, 0x4828, 0x10c4},  /* s5+s7 */
  {0x2980, 0x9011, 0x022a}}; /* s6+s8 */

/* this table contains the corresponding output masks for the
 * lookup procedure mentioned above.
 * (similarly available below in #defines). */
unsigned short outputmasks[4] = {
  0x000f /*s1+s2*/, 0x00f0 /*s3+s4*/,
  0x3300 /*s5+s7*/, 0xcc00 /*s6+s8*/};

/* input and output lookup masks (see above) */
/* s1+s2 */
#define IN00  0x0036
#define IN01  0x06c0
#define IN02  0x6900
#define OUT0  0x000f
/* s3+s4 */
#define IN10  0x5048
#define IN11  0x2106
#define IN12  0x8411
```

```
#define OUT1  0x00f0
/* s5+s7 */
#define IN20  0x8601
#define IN21  0x4828
#define IN22  0x10c4
#define OUT2  0x3300
/* s6+s8 */
#define IN30  0x2980
#define IN31  0x9011
#define IN32  0x022a
#define OUT3  0xcc00

/*
 * initialize the macguffin s-box tables.
 * this takes a while, but is only done once.
 */
mcg_init()
{
  unsigned int i,j,k;
  int b;
  /*
   * input permutation for the 8 s-boxes.
   * each row entry is a bit position from
   * one of the three right hand registers,
   * as follows:
   *   a,a,b,b,c,c
   */
  static int sbits[8][6] = {
    {2,5,6,9,11,13},  {1,4,7,10,8,14},
    {3,6,8,13,0,15},  {12,14,1,2,4,10},
    {0,10,3,14,6,12}, {7,8,12,15,1,5},
    {9,15,5,11,2,7},  {11,13,0,4,3,9}};

  for (i=0; i<TSIZE; i++) {
    stable[i]=0;
    for (j=0; j<8; j++)
      stable[i] |=
        sboxes[j][((i>>sbits[j][0])&1)
            |(((i>>sbits[j][1])&1)<<1)
            |(((i>>sbits[j][2])&1)<<2)
            |(((i>>sbits[j][3])&1)<<3)
            |(((i>>sbits[j][4])&1)<<4)
            |(((i>>sbits[j][5])&1)<<5)];
  }
}
```

```
/*
 * expand key to ek
 */
mcg_keyset(key,ek)
     unsigned char *key;
     mcg_key *ek;
{
  int i,j;
  unsigned char k[2][8];

  mcg_init();
  bcopy(&key[0],k[0],8);
  bcopy(&key[8],k[1],8);
  for (i=0; i<KSIZE; i++)
    ek->val[i]=0;
  for (i=0; i<2; i++)
    for (j=0; j<32; j++) {
      mcg_block_encrypt(k[i],ek);
      ek->val[j*3] ^= k[i][0] | (k[i][1]<<8);
      ek->val[j*3+1] ^= k[i][2] | (k[i][3]<<8);
      ek->val[j*3+2] ^= k[i][4] | (k[i][5]<<8);
    }
}

/*
 * codebook encrypt one block with given expanded key
 */
mcg_block_encrypt(blk,key)
     unsigned char *blk;
     mcg_key *key;
{
  unsigned short r0, r1, r2, r3, a, b, c;
  int i;
  unsigned short *ek;

  /* copy cleartext into local words */
  r0=blk[0]|(blk[1]<<8);
  r1=blk[2]|(blk[3]<<8);
  r2=blk[4]|(blk[5]<<8);
  r3=blk[6]|(blk[7]<<8);

  ek = &(key->val[0]);
  /* round loop, unrolled 4x */
  for (i=0; i<(ROUNDS/4); i++) {
```

```
    a = r1 ^ *(ek++);  b = r2 ^ *(ek++);  c = r3 ^ *(ek++);
    r0 ^=((OUT0 & stable[(a & IN00)|(b & IN01)|(c & IN02)])
        | (OUT1 & stable[(a & IN10)|(b & IN11)|(c & IN12)])
        | (OUT2 & stable[(a & IN20)|(b & IN21)|(c & IN22)])
        | (OUT3 & stable[(a & IN30)|(b & IN31)|(c & IN32)]));
    a = r2 ^ *(ek++);  b = r3 ^ *(ek++);  c = r0 ^ *(ek++);
    r1 ^=((OUT0 & stable[(a & IN00)|(b & IN01)|(c & IN02)])
        | (OUT1 & stable[(a & IN10)|(b & IN11)|(c & IN12)])
        | (OUT2 & stable[(a & IN20)|(b & IN21)|(c & IN22)])
        | (OUT3 & stable[(a & IN30)|(b & IN31)|(c & IN32)]));
    a = r3 ^ *(ek++);  b = r0 ^ *(ek++);  c = r1 ^ *(ek++);
    r2 ^=((OUT0 & stable[(a & IN00)|(b & IN01)|(c & IN02)])
        | (OUT1 & stable[(a & IN10)|(b & IN11)|(c & IN12)])
        | (OUT2 & stable[(a & IN20)|(b & IN21)|(c & IN22)])
        | (OUT3 & stable[(a & IN30)|(b & IN31)|(c & IN32)]));
    a = r0 ^ *(ek++);  b = r1 ^ *(ek++);  c = r2 ^ *(ek++);
    r3 ^=((OUT0 & stable[(a & IN00)|(b & IN01)|(c & IN02)])
        | (OUT1 & stable[(a & IN10)|(b & IN11)|(c & IN12)])
        | (OUT2 & stable[(a & IN20)|(b & IN21)|(c & IN22)])
        | (OUT3 & stable[(a & IN30)|(b & IN31)|(c & IN32)]));
  }
  /* copy 4 encrypted words back to output */
  blk[0] = r0;  blk[1] = r0>>8;
  blk[2] = r1;  blk[3] = r1>>8;
  blk[4] = r2;  blk[5] = r2>>8;
  blk[6] = r3;  blk[7] = r3>>8;
}

/*
 * codebook decrypt one block with given expanded key
 */
mcg_block_decrypt(blk,key)
     unsigned char *blk;
     mcg_key *key;
{
  unsigned short r0, r1, r2, r3, a, b, c;
  int i;
  unsigned short *ek;

  /* copy ciphertext to 4 local words */
  r0=blk[0]|(blk[1]<<8);
  r1=blk[2]|(blk[3]<<8);
  r2=blk[4]|(blk[5]<<8);
  r3=blk[6]|(blk[7]<<8);
```

```
  ek = &(key->val[KSIZE]);
  /* round loop, unrolled 4x */
  for (i=0; i<(ROUNDS/4); ++i) {
    c = r2 ^ *(--ek);  b = r1 ^ *(--ek);  a = r0 ^ *(--ek);
    r3 ^=((OUT0 & stable[(a & IN00)|(b & IN01)|(c & IN02)])
        | (OUT1 & stable[(a & IN10)|(b & IN11)|(c & IN12)])
        | (OUT2 & stable[(a & IN20)|(b & IN21)|(c & IN22)])
        | (OUT3 & stable[(a & IN30)|(b & IN31)|(c & IN32)]));
    c = r1 ^ *(--ek);  b = r0 ^ *(--ek);  a = r3 ^ *(--ek);
    r2 ^=((OUT0 & stable[(a & IN00)|(b & IN01)|(c & IN02)])
        | (OUT1 & stable[(a & IN10)|(b & IN11)|(c & IN12)])
        | (OUT2 & stable[(a & IN20)|(b & IN21)|(c & IN22)])
        | (OUT3 & stable[(a & IN30)|(b & IN31)|(c & IN32)]));
    c = r0 ^ *(--ek);  b = r3 ^ *(--ek);  a = r2 ^ *(--ek);
    r1 ^=((OUT0 & stable[(a & IN00)|(b & IN01)|(c & IN02)])
        | (OUT1 & stable[(a & IN10)|(b & IN11)|(c & IN12)])
        | (OUT2 & stable[(a & IN20)|(b & IN21)|(c & IN22)])
        | (OUT3 & stable[(a & IN30)|(b & IN31)|(c & IN32)]));
    c = r3 ^ *(--ek);  b = r2 ^ *(--ek);  a = r1 ^ *(--ek);
    r0 ^=((OUT0 & stable[(a & IN00)|(b & IN01)|(c & IN02)])
        | (OUT1 & stable[(a & IN10)|(b & IN11)|(c & IN12)])
        | (OUT2 & stable[(a & IN20)|(b & IN21)|(c & IN22)])
        | (OUT3 & stable[(a & IN30)|(b & IN31)|(c & IN32)]));
  }
  /* copy decrypted bits back to output */
  blk[0] = r0;  blk[1] = r0>>8;
  blk[2] = r1;  blk[3] = r1>>8;
  blk[4] = r2;  blk[5] = r2>>8;
  blk[6] = r3;  blk[7] = r3>>8;
}
```