UCRL- 88494 Rev. 1
PREPRINT

Lucifer, A Cryptographic Algorithm

Arthur Sorkin

This paper was prepared for submittal to
Cryptologia

April 1983

DISCLAIMER

Lucifer, A Cryptographic Algorithm

Arthur Sorkin

Lawrence Livermore National Laboratory

Abstract:

Lucifer, a direct predecessor of the DES algorithm, is a block-cipher having a 128 bit block size and 128 bit key length. Its general design principles and properties are described and discussed. A simple FORTRAN program is presented which implements the algorithm, providing a modern, secure cryptographic algorithm that can be used in personal computers. Lucifer is of special interest because it is in the same class of product ciphers as DES but is much simpler. Study of Lucifer may reveal cryptanalytic methods that can be applied to DES.

Keywords: Lucifer, block ciphers, DES, cryptographic algorithms, FORTRAN programs

I. Introduction

Lucifer is a high security, 128 bit key, block-cipher algorithm with a 128 bit block size. It is a direct predecessor of DES and is the same variety of product cipher, using alternating linear and non-linear transformations. Lucifer is therefore a good subject for cryptanalysis in order to discover principles that can be applied to DES.

One of the principle complaints about DES is the short, 56 bit length of the key. It is asserted [4,5,11,12] that DES could be broken by exhaustive search at a reasonable cost with today's hardware. It is also asserted that by 1990, the increased speed of the available hardware will

make DES so insecure that some form of replacement will be a necessity [4]. It is unlikely that exhaustive search will ever be a feasible technique with Lucifer because of its 128 bit key length, and it is extremely likely that any successor of DES will have a 128 bit key [4].

DES has also been criticized because some of its design principles have been kept secret at the request of NSA [11,12], allowing for the possibility that there are weaknesses that only NSA and its designers are aware of. The same degree of secrecy does not appear to have been applied to Lucifer.

Because of its relationship to DES, and its relative simplicity compared to DES, Lucifer is worthy of study. An understanding of Lucifer helps to clarify the internal operation of DES, since the principal elements of Lucifer are also present in DES, though in more complex form. Since Lucifer and DES have similar key-message statistical properties, and Lucifer is very resistent to exhaustive search because of its 128 bit key, Lucifer appears to be relatively strong cryptographically. Lucifer's relative simplicity also makes it an extremely easy algorithm to implement in software, making it a reasonable candidate for use on a personal computer.

This paper discusses the development of Lucifer, its design and hardware implementation. The relationship of Lucifer to DES is also discussed, and some suggestions for cryptanalytic study are made. An original FORTRAN implementation is presented that is suitable for use on a personal computer or in other applications.

II. The Hardware Device

Lucifer was developed at IBM Thomas J. Watson Research Laboratory in

the early 1970's [6,16] and was the subject of several U.S. patents [7,8,9,17]. The original Lucifer was a prototype cryptographic device constructed at Watson Laboratory for use in data communication. The Lucifer device was combined with the IBM 2770 Data Communications System as part of an experiment in computer security. A software algorithm provided the same cryptographic transformation in the host computer. The Lucifer device allowed the key to be loaded at the operator's option either from ROM or from magnetic cards. A mode selection switch allowed three modes of operation. One mode disabled the cryptographic function, and, therefore, cleartext was both sent and received. A second mode enabled Lucifer to receive ciphertext and decipher it; messages received in cleartext were not altered. Cleartext was always transmitted. The third mode was similar to the second, except that, in addition, all messages transmitted were enciphered.

The Lucifer device was constructed from standard TTL SSI and TTL MSI components. In all, 178 TTL modules were used, mounted on four wire-wrap boards. The state-of-the-art in LSI at the time Lucifer was constructed had an influence on the design of the device (and therefore the algorithm) in that an attempt was made to limit the complexity of the circuits. As a result, the Lucifer hardware used circular shift registers to store the key and the two halves of the message in order to simplify access to successive bytes. The APL program written by IBM that implements the algorithm in software mimics the hardware by actually shifting the key and halves of the message. The Fortran programs, discussed in Section V and shown in the Appendices, do not move the key or halves of the message; the appropriate bytes are accessed in place as needed.

III. The Algorithm

The Lucifer algorithm is a product cipher that uses alternating linear and non-linear transformations with the choice of non-linear transformation under control of the key [6]. Informally, a linear transformation, T, on a vector space, V, is one that has the linearity property: $T(ax+by) = aT(x)+bT(y)$, where x and y are vectors and a and b are scalars. A non-linear transformation is one that does not have the linearity property. Strings of bits and Boolean operations can form a vector space with bit strings vectors, individual bits scalars, bitwise EXCLUSIVE-OR vector addition, and bitwise logical AND multiplication by a scalar. For a formal treatment see [11,14]. For example, a permutation $p(x_1x_2 \ldots x_n) = x_{p(1)}x_{p(2)} \ldots x_{p(n)}$ of a string of bits is a linear transformation.

Lucifer has a block size and key size of 128 bits (16 bytes). 128 bit plaintext blocks produce 128 bit ciphertext blocks under control of the 128 bit key. Every bit of the key and every bit of the cleartext participate in the construction of every bit of the ciphertext. Tests conducted by IBM indicate that the change of a single bit in either the key or the message causes approximately half (64) of the bits in the resulting ciphertext to change [16]. The probability that a particular ciphertext bit will change appears to be very close to one-half, and the probabilities seem tc be independent for each bit of the ciphertext block.

The message block to be enciphered is divided into two halves, the upper and lower, each containing eight bytes (64 bits). The bytes of the message are initially ordered so that the rightmost byte is the highest, and the leftmost byte is the lowest. Encryption (and decryption) is divided into sixteen rounds. A block diagram of the functional units

appearing in one round is shown in Figure 1. During a round, the lower half of the message is transformed; the upper half is not changed, but its contents are used as input to the transformation. Between rounds, the upper and lower halves of the message are exchanged.

The sixteen bytes of the key and the eight bytes of each message half can be viewed as being enscribed axially on the face of three cylinders. The cylinders all rotate in the same direction. Let the initial position of byte zero on each cylinder be the origin for that cylinder with respect to rotation along its axis. The rotation of each cylinder by one step brings a new byte to the origin. The number of the new byte is one greater (modulo 16) than the number of the previous byte.

The halves of the message move in step and rotate one position after each byte is used. Byte zero returns to the origin on both cylinders after eight steps, i.e. one round. During encryption, the key rotates one step after each key byte is used, except at the end of each round, when it does not advance. Therefore, the key byte that ended the previous round begins the next one. For example, bytes zero through seven are used in the first round, and byte seven, not zero, starts the second round. For decryption, the key bytes are accessed in reverse order. The key is initially rotated to bring byte eight to the origin. Before each round and after each byte is used, the key advances one step. Thus, the first decryption round starts with byte nine and rotates past byte fifteen to end with byte zero; byte two starts the second round. Different key bytes are accessed in each round; the period of repetition is sixteen, so after the last round, the key is back in its initial position. The order in which the key bytes are accessed for each round is shown in Figure 2 and is discussed further in Section IV. The rotation of the key is not in

step with the rotation of the two halves of the message. This permits the bits of every key byte to be used in the generation of every ciphertext bit.

The first key byte accessed in each round is used as the transform-control-byte. Its number is the leftmost entry in each row (round) in Figure 2. Each of its eight bits in turn becomes the interchange-control-bit, which is used to choose which of two non-linear transformations will be applied to a byte in the upper half of the message. For both encryption and decryption, bits seven through zero of the transform-control-byte choose the transform for bytes zero through seven of the upper half of the message respectively.

The non-linear transformations contain two different non-linear substitution boxes (S-boxes), S_0 and S_1. Each S-box has four input bits and four output bits, so the input and output can represent the numbers from zero to fifteen (one hexdigit) in binary. An S-box can be considered to implement a permutation of the numbers from 0 to 15. Equivalently, it can also be viewed as a simple substitution of 4-bit quantities into 4-bit quantities. The Lucifer S-box implementation decodes the four binary bits into values from zero to fifteen, performs a fixed permutation of the values from zero to fifteen, and encodes the values from zero to fifteen back into four binary bits. While the internal S-box permutation is a linear transformation of the 4 input bits when they are considered to be binary numbers from 0 to 15, it is a <u>non-linear</u> transformation of the four input bits when they are considered to be simply a vector of bits. A block diagram of an S-box appears in Figure 3. The permutations for S_0 and S_1 are shown in Figure 4. If the interchange-control-bit is zero, then the right hexdigit of the

message byte is input to S_1 and the left hexdigit is input to S_0. If the interchange-control-bit is one, then the hexdigit inputs to the S-boxes are interchanged; the right hexdigit is input to S_0 and the left hexdigit is input to S_1. One of two non-linear transforms results. The generation of transformed bytes using the bytes of the upper half of the message as input to the key-controlled S-boxes is called confusion.

For step n in a particular round, a confused byte is generated from byte n in the upper message half. It is then bitwise XOR'ed (addition modulo 2) with the key byte that is at the origin of the key cylinder for step n in that particular round. Figure 2 shows the key byte accessed for every step of each round. This process is called key interruption, since the use of the key acts as a barrier to cryptanalysis by merging some secret information into the confused bytes. The eight bits of each resulting interrupted byte are permuted according to a fixed permutation, shown in Figures 1 and 4.

The permuted bits are then XOR'ed with eight bits of the lower part of the message. The eight bits in the lower half of the message are chosen according to the bit pattern of convolution XOR cells shown in Figures 1 and 5. The convolution XOR cells remain fixed in space with respect to the origin as the lower message cylinder rotates. Figure 5 shows the lower message cylinder and convolution cells of Figure 1 cut at the origin and unfolded. As the cylinder rotates, each permuted-interrupted byte is bitwise XOR'ed with a different eight bits of the lower half of the message. All 64 bits of the lower message half are used in each round, and each bit is used exactly once. This process is called diffusion, since the result of the transformation of one-half of the message is diffused throughout the other half of the message. Key-interruption and

diffusion can be combined because XOR is associative and commutative.

The confusion, key interruption, and diffusion (c-i-d) cycle described above forms a single round. Confusion and diffusion were first suggested as a way in which to create a cryptographically strong cipher by Shannon in his well-known paper on secrecy systems [15]. Confusion, key interruption and diffusion are also used in DES, though in a somewhat more complex manner than in Lucifer [13]. The description of DES is actually more understandable when looked upon as describing key interruption, confusion, and then diffusion.

The Lucifer c-i-d cycle is repeated sixteen times with fifteen interchanges of the upper and lower message halves. The result is the ciphertext. Deciphering is performed by repeating the c-i-d cycles in reverse order, with the key rotated eight positions before the beginning of the first decryption round. In the sixteen repetitions of the c-i-d cycle, each of the 128 key bits is used once for confusion control and eight times for key interruption, so every bit of the ciphertext depends in a very complex way upon every bit of the message and every bit of the key.

IV. Key-Byte Access Schedule

Figure 2 shows the order in which the key bytes (rows) are accessed for each round. Each entry in the table is the number of the key byte to be accessed. For encipher operations the key bytes are taken from left to right and top to bottom. For decipher operations the key bytes are taken from left to right and bottom to top. The leftmost column contains the number of the transform-control-byte. The rows show the numbers of the eight key bytes used for key-interruption in the corresponding round.

Each element of row n + 1 of the table is obtained by adding 7 modulo 16 to the corresponding element of row n. Alternatively, each element of row n is obtained by adding 9 modulo 16 (= -7 modulo 16) to the corresponding element of row n + 1. In implementing the algorithm, it is not necessary to store the key-byte access schedule in tabular form, since it merely shows the exact order in which the key bytes pass the origin during the rotation of the key, as described above.

V. FORTRAN Programs

The original IBM report describing Lucifer contains an APL implementation that emulates the hardware implementation. A FORTRAN implementation was developed for this paper because APL is very hard to read and understand, and APL is unavailable on most small and many large computers. In addition, APL can be quite inefficient. On the other hand, FORTRAN is very widely available (including on personal computers), FORTRAN has an ANSI Standard, and FORTRAN programs are usually compiled so they normally have reasonable performance. The FORTRAN implementation was developed by comparing the APL program with the hardware block diagrams and written descriptions contained in [6,16]. There were some ambiguities in the written description of the order in which the bits of the key and message were loaded and stored in the hardware. These ambiguities were resolved by reference to the APL implementation.

Appendix 1 presents a simple FORTRAN subroutine which implements Lucifer. Because very few high-level languages, including FORTRAN, are well suited for bit manipulation operations, the message and key are stored in integer arrays with one message or key bit per array element; the value of each array element must be either zero or one. The

subroutine assumes that the key and message have already been converted from input format into array format by another subroutine; no attempt is made to verify that every array element is either zero or one.

The subroutine handles the key and message halves in place, without rotating them. Instead, pointers are used to indicate which key and message bytes are to be accessed, and the pointers are moved instead. The message is stored in variable m as an 8x8x2 three dimensional array (column, row, plane). It can be equivalenced to a 128 element one dimensional array. The planes correspond to the two halves of the message. The key is stored in the variable k as a 16x8 two dimensional array (column, row). It can be equivalenced to a 128 element one dimensional array. Variables s0 and s1 contain the permutations for the S-boxes, and variable pr contains the inverse of the fixed permutation used after key interruption. If the variable d is equal to one, then the subroutine deciphers; otherwise, it enciphers. Variable jj contains the index of the message byte (row) being accessed; variable kk contains the index of the bit (column) being accessed. Variable p(0) contains the index of the lower half of the message; p(1) contains the index of the upper half of the message. Variable kc holds the array index of the key byte currently being accessed; ks holds the array index of the transform-control-byte. Lines 7600 through 10000 implement the S-boxes and interchange control. Lines 10200 through 10900 implement key interruption and diffusion. Key interruption and diffusion are combined into one operation by first permuting the confused byte and the key byte, and <u>then</u> doing the XOR's (implemented as additon modulo 2). The diffusion pattern is contained in variable o, and the convolution cell for column kk and row jj is equal to (o(kk)+jj) modulo 8). Because the subroutine

operates on the message in place, it is necessary to physically swap the contents of the upper and lower halves after the end of the sixteen rounds in order to have the halves in the correct order. Lines 10900 through 11900 implement the interchange. This final swap would not have been necessary if we had been physically swapping halves all along.

Appendix 2 presents a sample FORTRAN program which calls Lucifer. The message block is enciphered and deciphered 500 times each, so that Lucifer is invoked 1000 times. Appendix 3 shows the timing for 1000 invocations. On a VAX 11/780 computer, the time to encrypt or decrypt a 128 bit block is approximately 100 ms. On the same computer, an optimized version of the NBS DES algorithm [13] takes between 40 and 50 ms to encrypt or decrypt 64 bits. It should be possible to speed up the Lucifer subroutine by optimizing it, however, that was not done in this presentation for reasons of clarity. An optimized program would not have corresponded in an obvious way to the description and figures presented in the rest of the paper.

Appendix 4 shows a subroutine that expands input bytes into array format, and a subroutine that compresses array format back into byte format. The conversion from byte format to array format guarantees that each array element is either one or zero.

VI. Conclusion

A recent article [3] asked if there was a reasonable secure, modern cryptographic algorithm that could be easily implemented on a personal computer. The FORTRAN version of the Lucifer algorithm presented in this paper is very suitable for use on a personal computer. The advantages of FORTRAN are that it is widely available, standardized, and usually

produces programs with reasonable performance. The Lucifer FORTRAN implementation is simple and reasonably fast. These programs can be optimized for speed, though the particular techniques employed would depend upon the processor used and its architecture. The programs presented here can easily be converted into another programming language (e.g. BASIC).

Lucifer is also interesting because it is the direct predecessor of DES, but is much simpler than DES. For example, Lucifer only has two S-boxes, the minimum possible for this kind of product cipher with rotating key, and therefore, the key is used to choose between the same two non-linear transforms in every round. DES effectively has 32 S-boxes (constructed from 8 more complicated ones), and the choice of which non-linear transforms are used in each round depends upon input bits as well as key bits. Studying the properties of Lucifer should yield some insights into the cryptanalysis of product ciphers with rotating keys.

It is known that without the rotating key this type of cipher is weak [10]. Some statistical techniques have been developed that allow cryptanalysis under a known-plaintext attack of very simple ciphers using alternating S-boxes and permutations [1,2]. It is possible that these techniques might be extended to Lucifer and DES, and they need not provide a complete cryptanalysis. The statistical attack might be used in combination with exhaustive search by first reducing the set of possible keys to a practical size; exhaustive search would then be used to examine every remaining key to find the correct one. However, to-date, the only publicly known cryptanalysis of Lucifer or DES is exhaustive search of the entire key space, which is currently impractical for DES and virtually impossible for Lucifer [4,5,11,12].

Even if the combined statistical/brute-force method suggested in the previous paragraph doesn't work, understanding the simpler Lucifer problem should help us to understand the DES problem, which, in turn, might lead to cryptanalytic techniques that can be applied directly to DES. Also, understanding the ways in which Lucifer was strengthened (to arrive at DES) might aid us in understanding the (still classified) criteria used by IBM (and NSA) to design and evaluate DES. This in turn might answer some of the questions raised about the existence of hidden weaknesses in DES.

VII. Acknowledgements

This work was performed under the auspices of the U. S. Department of Energy by Lawrence Livermore National Laboratory under contract number W-7405-ENG-48.

References

1. Andelman, D., "Maximum Likelihood Estimation Applied to Cryptanalysis," Doctoral Dissertation, Stanford University, Department of Electrical Engineering, December 1979.

2. Andelman, D. and J. Reed, "On the Cryptanalysis of Rotor Machines and Substitution-Permutation Networks, IEEE Transactions on Information Theory, Vol. IT-28, No. 4, pp. 578-584, July 1982.

3. Deavours, C. A., "The Black Chamber," Cryptologia, Vol. 6 No. 1, pp. 34-37, January 1982.

4. Diffie, W., "Cryptographic Technology: Fifteen Year Forcast," Advances in Cryptography - A Report on Crypto81, Allen Gersho, Editor, Department of Electrical and Computer Engineering, Report ECE 82-04, University of California, Santa Barbara, August 20, 1982.

5. Diffie, W. and M. Hellman, "Exhaustive Cryptanalysis of the NBS Data Encryption Standard," Computer, Vol. 10, No. 6, pp. 74-84, June 1977.

6. Feistel, H., "Cryptographic Coding for Data-Bank Privacy," IBM Research Report RC2827, Yorktown Heights, New York, March 18, 1970.

7. Feistel, H., "Block Cipher Cryptographic System," U.S. Patent No. 3798359, March 19,1974.

8. Feistel, H., "Step Code Ciphering System," U.S. Patent No. 3798360, March 19, 1974.

9. Feistel, H., "Centralized Verification System," U.S. Patent No. 3798605, March 19, 1974.

9. Grossman E. and B. Tuckerman, "Analysis of a Feistel-Like Cipher Weakened By Having No Rotating Key," IBM Research Report RC6375, Yorktown Heights, New York, January 31, 1977.

11. Konheim, A., Cryptography, A Primer, John Wiley and Sons, 1981. New York, New York.

12. Meyer, C. and S. Matyas, Cryptography - A New Dimension in Computer Data Security, Wiley-Interscience, New York, 1982.

13. National Bureau of Standards, Data Encryption Standard, FIPS Publication 46, Gaithersburg, Maryland.

14. Paige L. and J. D. Swift, Elements of Linear Algebra, Blaisdell Publishing Company, New York, 1965.

15. Shannon, C., "The Communications Theory of Secrecy Systems," Bell System Technical Journal, Vol. 28, pp. 656-715, 1949.

16. Smith, J. L., "The Design of Lucifer, A Cryptographic Device for Data Communications," IBM Research Report RC3326, Yorktown Heights, New York, April 15, 1971.

17. Smith, J. L., "Recirculating Block Cipher Cryptographic System," U.S. Patent No. 3796830, March 12, 1974.

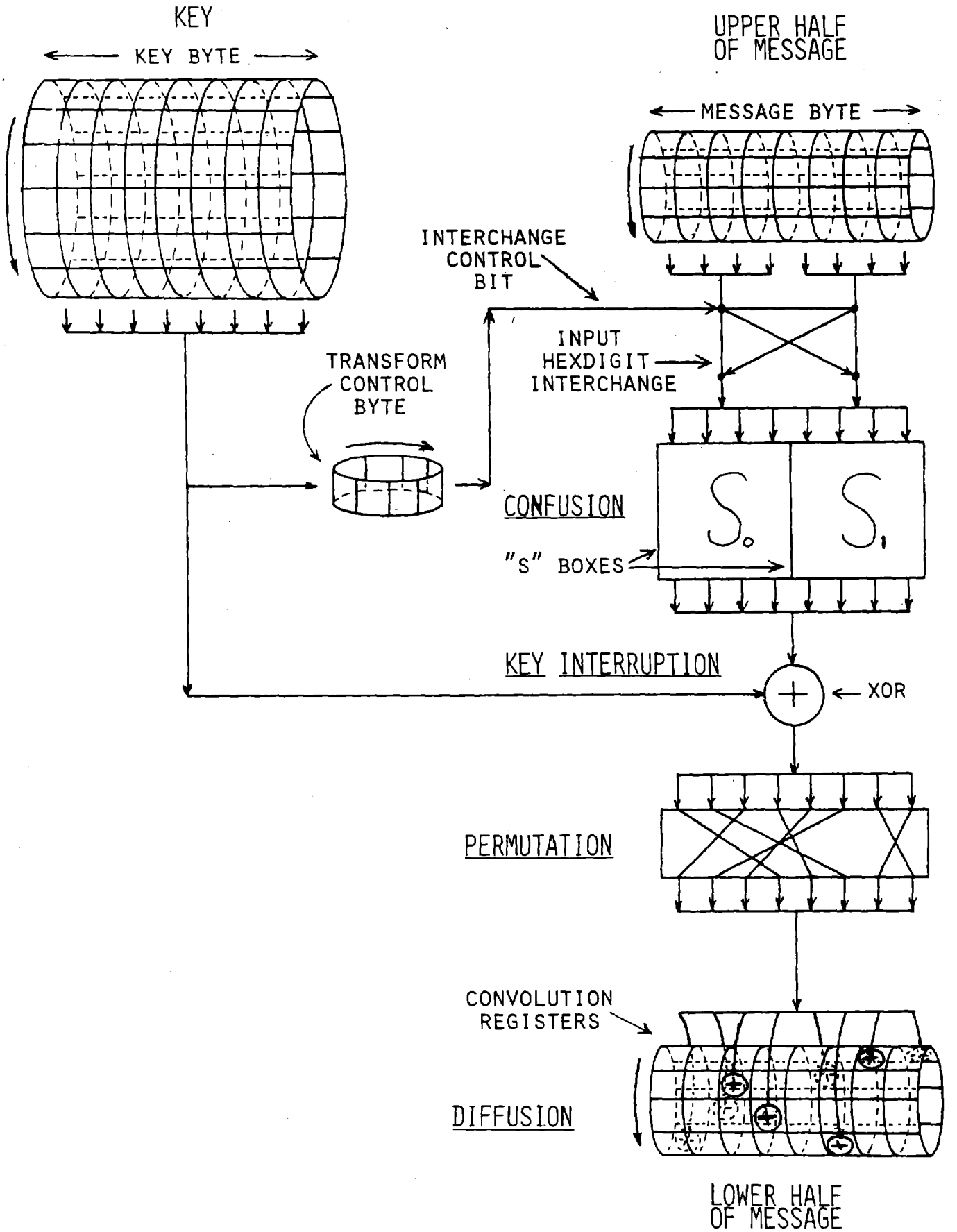

FIGURE 1
BLOCK DIAGRAM OF CID LOGIC

MESSAGE BYTE

C-I-D ROUND	0	1	2	3	4	5	6	7
1	0	1	2	3	4	5	6	7
2	7	8	9	10	11	12	13	14
3	14	15	0	1	2	3	4	5
4	5	6	7	8	9	10	11	12
5	12	13	14	15	0	1	2	3
6	3	4	5	6	7	8	9	10
7	10	11	12	13	14	15	0	1
8	1	2	3	4	5	6	7	8
9	8	9	10	11	12	13	14	15
10	15	0	1	2	3	4	5	6
11	6	7	8	9	10	11	12	13
12	13	14	15	0	1	2	3	4
13	4	5	6	7	8	9	10	11
14	11	12	13	14	15	0	1	2
15	2	3	4	5	6	7	8	9
16	9	10	11	12	13	14	15	0

FIGURE 2
KEY BYTE ACCESS SCHEDULE

S-BOX IMPLEMENTATION

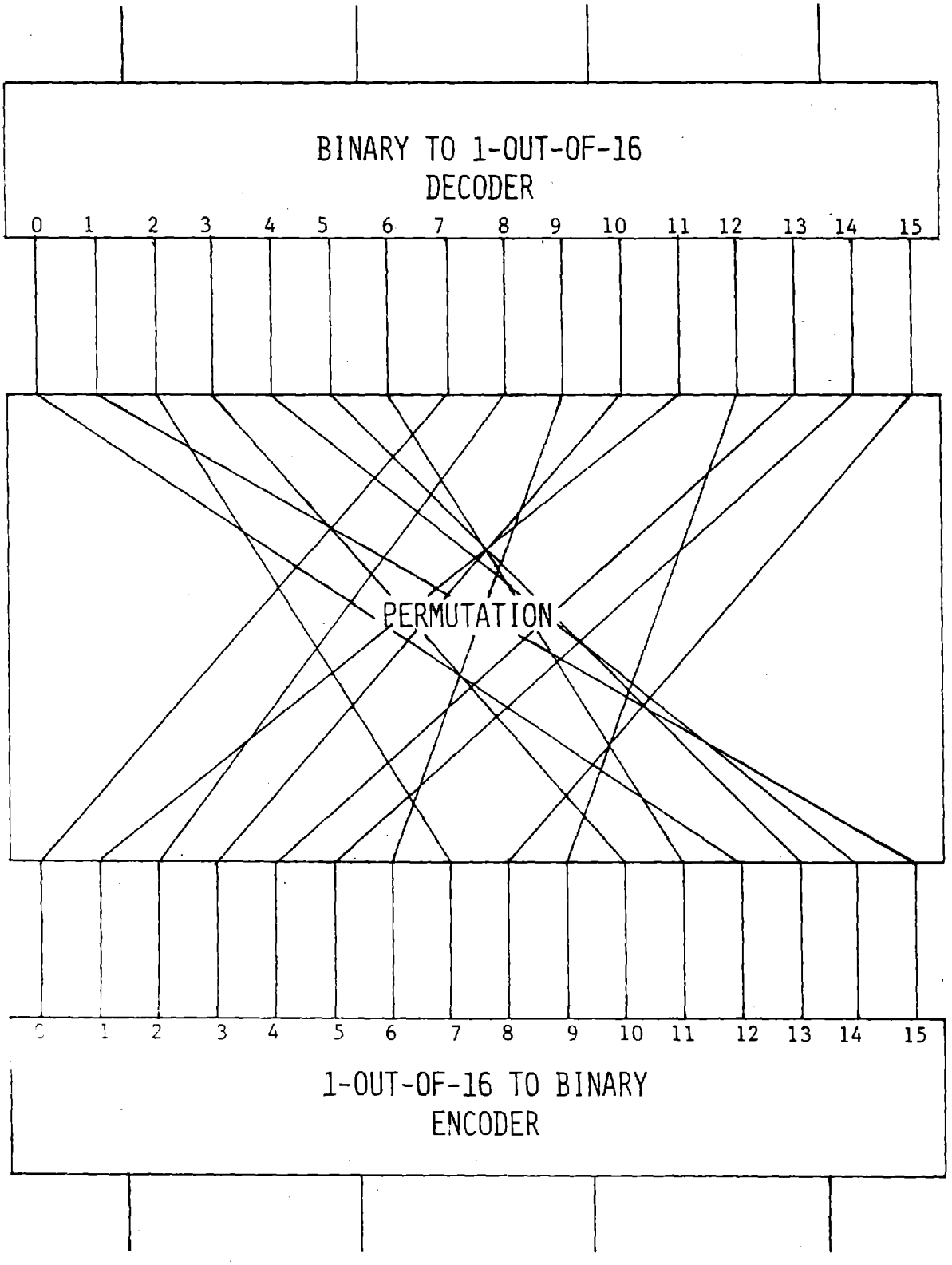

FIGURE 3

S-BOX
INTERNAL
PERMUTATIONS

S_0	S_1
0 ---- 12	0 ----- 7
1 ---- 15	1 ----- 2
2 ----- 7	2 ---- 14
3 ---- 10	3 ----- 9
4 ---- 14	4 ----- 3
5 ---- 13	5 ---- 11
6 ---- 11	6 ----- 0
7 ----- 0	7 ----- 4
8 ----- 2	8 ---- 12
9 ----- 6	9 ---- 13
10 ----- 3	10 ----- 1
11 ----- 1	11 ---- 10
12 ----- 9	12 ----- 6
13 ----- 4	13 ---- 15
14 ----- 5	14 ----- 8
15 ----- 8	15 ----- 5

FIXED
PERMUTATION

0 ---- 3
1 ---- 5
2 ---- 0
3 ---- 4
4 ---- 2
5 ---- 1
6 ---- 7
7 ---- 6

FIGURE 4
PERMUTATIONS

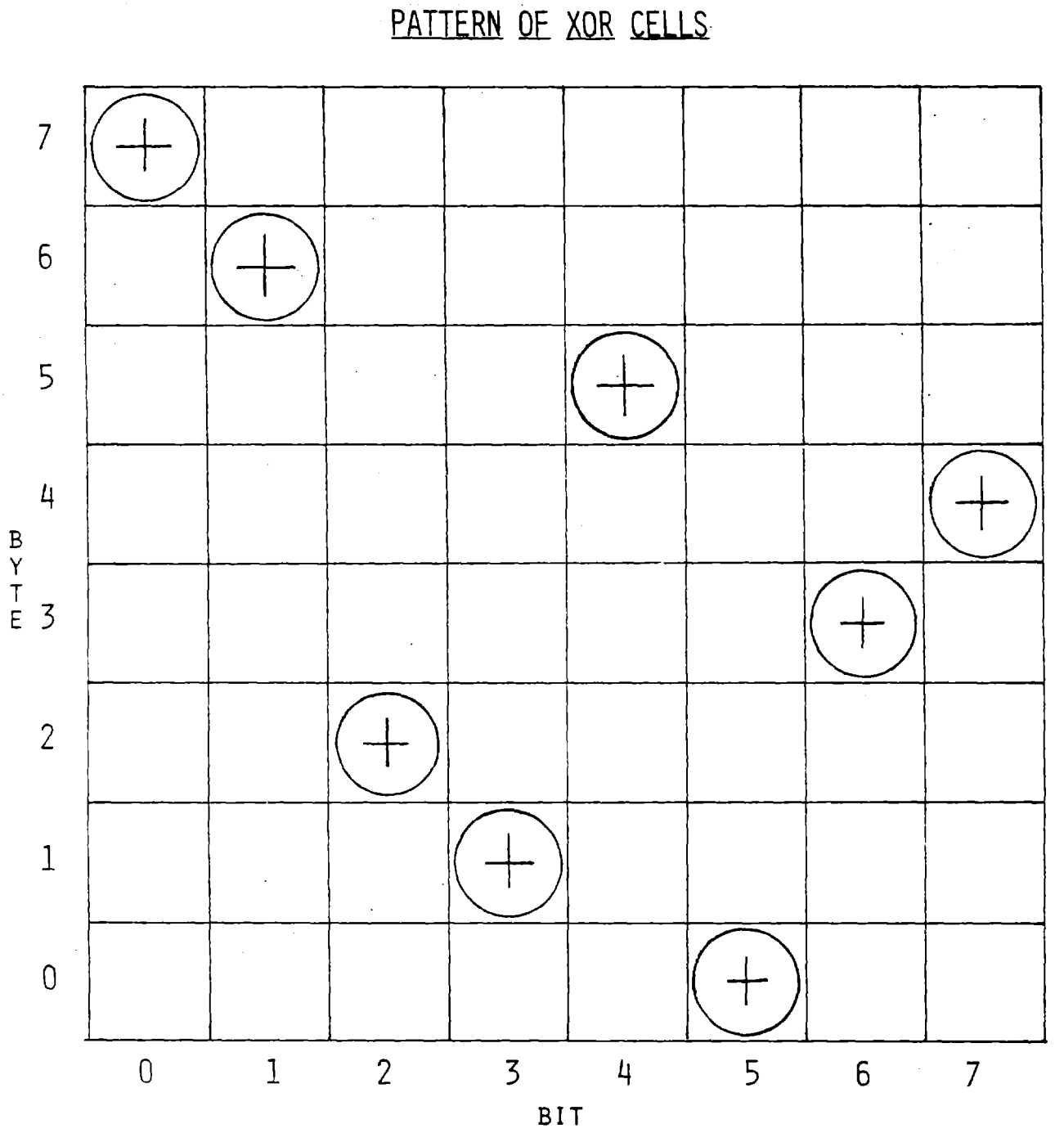

FIGURE 5
UNFOLDED CONVOLUTION REGISTERS

```
00100           subroutine lucifer(d,k,m)
00200           implicit integer(a-z)
00300           dimension m(0:7,0:7,0:1),k(0:7,0:15),o(0:7)
00400
00500     c     message block stored one bit/location.
00600     c     key stored one bit/location.
00700     c     values must be 0 or 1. this subroutine doesn't verify that
00800     c     condition for message or key.
00900
01000     c     fortran stores data with innermost subscript varying the
01100     c     fastest.therefore, we have m(column,row,plane) and
01200     c     k(column,row). the rows are the bytes of the message and
01300     c     key. the columns are the bits in the bytes. for a normal
01400     c     langage such as pl/1, we would declare m(row,column,plane)
01500     c     and k(row,column). we can equivalence a linear array of
01600     c     128 entries to the message and key because of the way
01700     c     in which they are stored.
01800
01900           dimension sw(0:7,0:7),pr(0:7),tr(0:7),c(0:1)
02000           dimension s0(0:15),s1(0:15)
02100           equivalence (c(0),h),(c(1),l)
02200
02300     c     diffusion  pattern
02400           data o/7,6,2,1,5,0,3,4/
02500
02600     c     inverse of fixed permutation
02700           data pr/2,5,4,0,3,1,7,6/
02800
02900     c     S-box permutations
03000           data s0/12,15,7,10,14,13,11,0,2,6,3,1,9,4,5,8/
03100           data s1/7,2,14,9,3,11,0,4,12,13,1,10,6,15,8,5/
03200
03300     c     the halves of the message byte selected are used as input
03400     c     to s0 and s1 to produce 4 v bits each. if k(jj,ks)=0 then
03500     c     the low order 4 bits are used with s0 and the high order 4
03600     c     bits are used with s1. if k(jj,ks)=1 then the low order
03700     c     4 bits are used with s1 and the high order 4 bits are used
03800     c     with s0.
03900
04000     c     we don't physically swap the halves of the message or rotate
04100     c     the message halves or key.  we use pointers into the arrays
04200     c     to tell which bytes are being operated on.
04300
04400     c     d=1 indicates decipher. encipher otherwise.
04500
04600     c     h0 and h1 point to the two  halves of the message.
04700     c     value 0 is the lower half and value 1 is the upper
04800
04900           h0=0
05000           h1=1
05100
05200           kc=0
05300           if (d.eq.1) kc=8
05400
05500           do 100 ii=1,16,1
*
```

```
06400    c          c-i-d cycle
06500
06600               if (d.eq.1) kc=mod(kc+1,16)
06700
06800    c          ks is the index of the transform control byte
06900               ks=kc
07000
07100               do 200 jj=0,7,1
07200
07300                 l=0
07400                 h=0
07500
07600    c            construct the integer values of the hexdigits of one byte
07700    c            of the message.
07800    c            call compress(m(0,jj,h1),c,2) is equivalent and simpler
07900    c            but was slower. c(0)=h & c(1)=l by equivalence.
08000
08100                 do 400 kk=0,3,1
08200                   l=l*2+m(7-kk,jj,h1)
08300    400          continue
08400
08500                 do 410 kk=4,7,1
08600                   h=h*2+m(7-kk,jj,h1)
08700    410          continue
08800
08900    c            controlled interchange and s-box permutation.
09000
09100                 v=(s0(l)+16*s1(h))*(1-k(jj,ks))+(s0(h)+16*s1(l))*k(jj,ks)
09200
09300    c            convert v back into bit array format.
09400    c            call expand(v,tr,2) is equivalent and simpler but
09500    c            was slower.
09600
09700                 do 500 kk=0,7,1
09800                   tr(kk)=mod(v,2)
09900                   v=v/2
10000    500          continue
10100
10200    c            key-interruption and diffusion combined.
10300    c            the k+ir term is the permuted key interruption.
10400    c            mod(0(kk)+jj,8) is the diffusion row for column kk.
10500    c            row = byte & column = bit within byte.
10600                 do 300 kk=0,7,1
10700                   m(kk,mod(o(kk)+jj,8),h0)=mod(k(pr(kk),kc)+tr(pr(kk))+
10800         1           m(kk,mod(o(kk)+jj,8),h0),2)
10900    300          continue
11000
11100                 if (jj.lt.7.or.d.eq.1) kc=mod(kc+1,16)
11200
11300    200        continue
11400
11500    c          swap values in h0 and h1 to swap halves of message.
11600               jjj=h0
11700               h0=h1
11800               h1=jjj
11900
12000    100      continue
*
```

```
13300    c        physically swap upper and lower halves of the message after
13400    c        the last round. we wouldn't have needed to do this if we
13500    c        had been swapping all along.
13600
13700             do 700 jj=0,7,1
13800               do 800 kk=0,7,1
13900                 sw(kk,jj)=m(kk,jj,0)
14000                 m(kk,jj,0)=m(kk,jj,1)
14100                 m(kk,jj,1)=sw(kk,jj)
14200    800        continue
14300    700       continue
14400
14500             return
14600             end
*
```

APPENDIX 2

```
00100    c        main program that uses Lucifer
00200             implicit integer (a-z)
00300             data handle/0/
00400             dimension k(0:7,0:15),m(0:7,0:7,0:1)
00500    c        message and key arrays are equivalenced to 128 element linear
00600    c        arrays.
00700             dimension key(0:127),message(0:127)
00800             equivalence (k(0,0),key(1)),(m(0,0,0),message(1))
00900    c        input byte arrays for reading key and message
01000    c        input is in hex digits. 128 bits = 32 hex digits = 16 bytes
01100             dimension kb(0:31),mb(0:31)
01200
01300             write(6,1003)
01400             read(5,1004) (kb(i),i=0,31)
01500
01600             write(6,1005)
01700             read(5,1006) (mb(i),i=0,31)
01800
01900             call expand(message,mb,32)
02000             call expand(key,kb,32)
02100
02200             write(6,1000) (key(i), i=0,127)
02300             write(6,1001) (message(i), i=0,127)
02400
02500             if (.not. lib$init_timer(handle)) goto 800
02600
02700             do 500 i=1,500,1
02800
02900    c        encipher
03000             d=0
03100             call lucifer(d,k,m)
03200
03300    c        decipher
03400             d=1
03500             call lucifer(d,k,m)
03600
03700    500      continue
03800             if(.not.lib$show_timer(handle)) goto 800
03900    800      continue
04000             write(6,1001) (message(i), i=0,127)
04100
04200             call compress(message,mb,32)
04300             call compress(key,kb,32)
04400             write(6,1003)
04500             write(6,1007) (kb(i),i=0,31)
04600             write(6,1005)
04700             write(6,1007) (mb(i),i=0,31)
04800
04900    1000     format(' key '/16(1x,i1))
05000    1001     format(' plain '/16(1x,i1))
05100    1002     format(' cipher '/16(1x,i1))
05200    1003     format(' key ')
05300    1004     format(32z1.1)
05400    1005     format(' plain ')
05500    1006     format(32z1.1)
05600    1007     format(1x,32z1.1)
05700             end
```

```
key
0123456789ABCDEFFEDCBA9876543210
plain
AAAAAAAAAAAAAAAABBBBBBBBBBBBBBBB
key
0 0 0 0 0 0 0 1 0 0 1 0 0 0 1 1
0 1 0 0 0 1 0 1 0 1 1 0 0 1 1 1
1 0 0 0 1 0 0 1 1 0 1 0 1 0 1 1
1 1 0 0 1 1 0 1 1 1 1 0 1 1 1 1
1 1 1 1 1 1 1 0 1 1 0 1 1 1 0 0
1 0 1 1 1 0 1 0 1 0 0 1 1 0 0 0
0 1 1 1 0 1 1 0 0 1 0 1 0 1 0 0
0 0 1 1 0 0 1 0 0 0 0 1 0 0 0 0
plain
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1
1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1
1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1
1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1
 ELAPSED: 00:01:47.51  CPU: 0:01:41.17  BUFIO: 0  DIRIO: 0  FAULTS: 0
plain
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1
1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1
1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1
1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1
key
0123456789ABCDEFFEDCBA9876543210
plain
AAAAAAAAAAAAAAAABBBBBBBBBBBBBBBB
$
```

```
00100            subroutine compress(a,b,l)
00200              implicit integer(a-z)
00300              dimension a(0:*),b(0:*)
00400
00500    c         a is the array in bit array format.
00600    c         b is the array in byte format.
00700    c         l is the length of array b in hexdigits.
00800    c         a must be 4*l.
00900
01000              do 100 i=0,l-1,1
01100                v=0
01200                do 200 j=0,3,1
01300                  v=v*2+mod(a(j+i*4),2)
01400    200         continue
01500                b(i)=v
01600    100       continue
01700
01800              return
01900            end
```

```
00100            subroutine expand(a,b,l)
00200              implicit integer (a-z)
00300              dimension a(0:*),b(0:*)
00400
00500    c         a is the array in bit array format.
00600    c         b is the array in byte format.
00700    c         l is the length of the array b in hexdigits.
00800    c         a must be 4*l long.
00900
01000              do 100 i=0,l-1,1
01100                v=b(i)
01200                do 200 j=0,3,1
01300                  a((3-j)+i*4)=mod(v,2)
01400                  v=v/2
01500    200         continue
01600    100       continue
01700
01800              return
01900            end
```