MIT/LCS/TM-50

AN ENCIPHERING MODULE
FOR
MULTICS

G. Gordon Benedict

July 1974

MAC TECHNICAL MEMORANDUM 50

AN ENCIPHERING MODULE FOR MULTICS

G. Gordon Benedict

July, 1974

This research was performed in the Computer Systems Research Division of Project MAC, an M.I.T. Interdepartmental Laboratory, and was sponsored in part by the Advanced Research Projects Agency (ARPA) of the Department of Defense under ARPA Order No. 2095 which was monitored by Office of Naval Research Contract No. N00014-70-A-0362-0006; in part by the Air Force Information Systems Technology Applications Office and by ARPA under ARPA Order No. 2641; and in part by Honeywell Information Systems, Inc.

This Technical Memorandum reproduces a June, 1974, M.I.T. Electrical Engineering Department S. B. Thesis of the same title.

ABSTRACT

Recently IBM Corporation has declassified an algorithm for encryption usable for computer-to-computer or computer-to-terminal communications. Their algorithm was implemented in a hardware device called Lucifer. A software implementation of Lucifer for Multics is described. A proof of the algorithm's reversibility for deciphering is provided. A special hand-coded (assembly language) version of Lucifer is described whose goal is to attain performance as close as possible to that of the hardware device. Performance measurements of this program are given. Questions addressed are: How complex is it to implement an algorithm in software designed primarily for digital hardware? Can such a program perform well enough for use in the I/O system of a large time-sharing system?

Author: G. Gordon Benedict

Thesis Supervisor: Prof. Jerome H. Saltzer

CONTENTS

FIGURES

TABLES

OVERVIEW

This thesis examines the enciphering algorithm recently released by IBM, Lucifer. This algorithm is described as a hardware mechanism in "The Design of Lucifer, a Cryptographic Device for Data Communications", by J. Lynn Smith; this was the primary source document.

A proof of Lucifer's reversibility is given, that it will in fact correctly decipher its previously-output ciphertext when provided with the same key used for enciphering. Two software implementations are described and their performance measured.

This paper is divided into five sections and four appendices. "Introduction to Enciphering" briefly explains the uses of enciphering in computer-to-computer and computer-to-terminal communication as a security enhancement. "Enciphering Algorithms and Lucifer in Particular" lists some criteria for a good computer-oriented cipher. The general operation of Lucifer is depicted without much detail. Sufficient detail is however given for understanding of "A Simple Proof of Lucifer's Reversibility". This section provides an informal proof that Lucifer works in that it correctly deciphers its own ciphertext. "The Multics Software Implementation" demonstrates how to use the enciphering programs. The final section, "Timing and Conclusions", presents performance measurements of a PL/I and a Multics assembly language version of Lucifer. Appendix A, "Operation of the Lucifer Hardware", details the operation of the hardware device described by Smith. Appendix B, "The PL/I Implementation", details a software version in the PL/I language designed to simulate closely the Lucifer hardware in its operation and be readable and exportable. Appendix C, "The Assembly Language Implementation", details a version of Lucifer optimized for execution time. For those readers unfamiliar with the Multics hardware, "An Introduction to Multics Assembler" briefly explains those features of the Honeywell model 6180 processor used by Lucifer.

INTRODUCTION TO ENCIPHERING

Much attention has been paid recently to computer and data security. Computer security consists of regulating the use of computer facilities to only those people or those tasks authorized to use them. This has been attempted by such mechanisms as passwords, protection rings, and privileged instructions. Data security is becoming more important with the advent of government and corporate personal-data files. This problem is magnified if the computer system is available to many users via telecommunications. Given the above facilities for regulating computer facility use, access control is one mechanism that is available for preventing unauthorized access to data files. However, this mechanism fails when data is transmitted over telephone lines, radio links, or physical (mail or courier) shipments. Such communications are easily tapped without the legitimite user's knowledge, except for the case of a courier. Even more insidious than the traditional reading of sensitive data is the insertion of spurious data designed to confuse or misdirect the operation of a system. One mechanism for minimizing this problem is enciphering that data, which protects the data itself rather than the medium of transmitting the data.

Enciphering is a process whereby transformations are made on the message (cleartext), usually on a bit or character level. If the algorithm is known the cipher may be breakable by analyzing the ciphertext, particularly if sample cleartext for some of the ciphertext is available. Since an enciphering algorithm must be reversible to be useful, a key known by both the message originator and the intended receiver is also used. Thus if the key is intercepted or deduced the cipher is now cracked. The essence of successful cryptology is in devising an enciphering algorithm which is not possible to crack in the time-span of the message's usefulness, and in keeping the key secret.

Enciphering helps in preventing insertion of spurious data to confuse a computer, as well as preventing reading of secret data. This is because a random message inserted onto the communication link will probably decipher to unrecognizable garbage. The algorithm implemented in this paper is so constructed that if one bit is changed in a legitimate enciphered message, the deciphered text will almost certainly be unrecognizable. This prevents the form of interference wherein a saboteur records (taps) the ciphertext, changes some bits randomly without even understanding the message, and inserts the text onto the telephone lines. Unrecognizable text can usually be rejected by the computer. There still remains the problem of the saboteur who records the ciphertext and replays it unchanged later. This can be extremely damaging to unrepeatable or irreversible processes. A method of avoiding this problem is message chaining, whereby a part of the previous data exchange is enciphered in this data exchange, as a verification field. Thus the same message replayed tomorrow would contain an out-of-date verification field and be rejected. The operation of such a system is discussed at length in Smith's paper.

Enciphering can also be used for computer-to-terminal communications. The terminal would contain a hardware deciphering module; the algorithm described here was designed with this purpose in mind. The user could have his key on a magnetic card, or he could type it in on the terminal. The computer would contain a central file of all users' keys and a software or hardware version of the enciphering module.

Enciphering can add some security to online files against the possibility of random hardware or software failures or physical stealing of backup tapes, disk packs, etc. Enciphering in this application merely adds another dimension of security.

This paper details an enciphering algorithm developed by Feistel and Smith of IBM for computer-to-terminal communications. A software version has been prepared, intended to be used as part of the input/output software or the network interface of Multics. A command to encipher and decipher online segments has also been written. A proof of the algorithm's reversibility is also given; this was hinted at but not proved in the Smith and Feistel papers.

ENCIPHERING ALGORITHMS AND LUCIFER IN PARTICULAR

There are several desiderata in the design of an enciphering algorithm. One is needed which is easily implemented in hardware, yet would provide a great measure of security against cryptanalysts -- especially against those armed with computers of their own.

Many traditional algorithms have operated by performing one-for-one character substitutions based on the key. For example, the "Vignere-Vernam" ciphers use a square array of characters. To encipher, each character of cleartext is used as a column index into this array; the character of the key corresponding to this character of cleartext (i.e., the _n_th character of the key corresponds with the _n_th character of cleartext) is used as a row index. The character at the intersection is the corresponding ciphertext character. The key is repeated as many times as necessary to exhaust all characters of cleartext. The square array can contain essentially any characters. These ciphers' weakness arise from the key repitition and the simple substitution of a very short message element (a character). Such ciphers are subject to frequency analysis, particularly if a sample of cleartext is available. This oversimplified account is drawn from "Cryptology, the Computer, and Data Privacy" by M. B. Girdansky.

The algorithm developed by Smith and Feistel uses the traditional enciphering mechanisms of substitution of strings and modulo arithmetic on strings. However, by repeated cycles, essentially a substitution is performed on not small characters but 128-bit blocks. Thus such methods as frequency analysis require computation time on the order of the lifetime of the universe.

This algorithm, called Lucifer, has the added advantages of simple hardware implementation with shift-registers and easy reversibility. A general description of the algorithm follows and then a proof of its reversibility.

The basic transformations used are one-to-one mappings and exclusive-ors (mod-2 addition). The input is divided into equal-sized blocks; each block is processed completely independently of the others. The following description refers to one block only. It is thus desirable from a cryptographic point of view to use as large a block size as possible, since the more bits which affect a given bit of ciphertext, the harder will be the job of the cryptanalyst. As mentioned before, a basic weakness in many ciphers is the small block size.

A block is broken into the top half and the bottom half. Without changing the bottom half, it is broken into easily manipulable units called bytes. Each byte undergoes one of two one-to-one transformations depending upon a bit of the key. This collection of transformed bytes is referred to as confused bytes, and the operation is referred to as confusion. Next, each bit of the confused bytes is modulo-2 summed with a different bit of the key. This operation is referred to as interruption. Now these bytes are modulo-2 summed with the top half of the cleartext, the block previously unused. This is called diffusion. The two halves are swapped; this operation is called interchange. Sixteen such cycles occur. One complete confusion-interruption-diffusion cycle is called a CID cycle. The schedule for accessing key bits is so arranged that every key bit is used for both controlling the confusion transformation and for interruption. The interchange operation occurs on every cycle except the last.

Figure 1: Flowchart

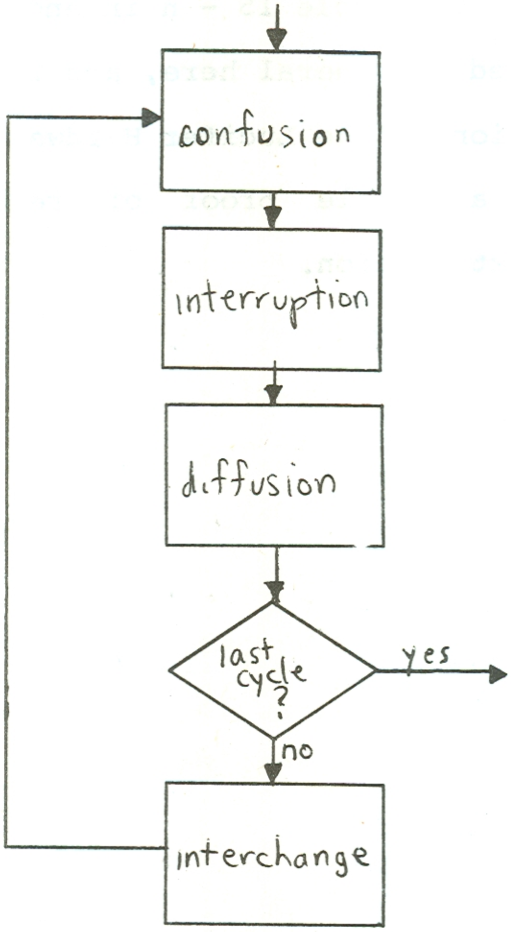
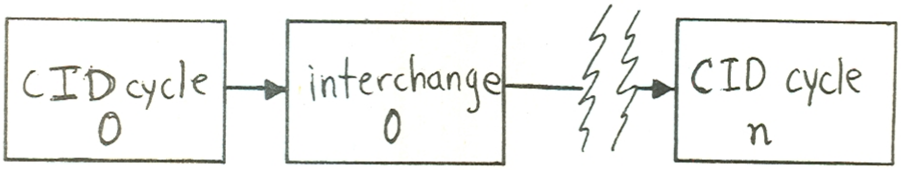


Figure 1 shows a flowchart of the operation. Thus the algorithm consists of:

Figure 2: Block Diagram



The only difference between enciphering and deciphering is the order in which the key bits are accessed. Within CID cycle n during deciphering, key bits are accessed in the same order as in CID cycle 15 - n in enciphering. These operations, explained in general here, are fully detailed in Appendix A - Operation of the Lucifer Hardware.

This leads to a simple proof of reversibility, as explained in the next section.

A PROOF OF LUCIFER'S REVERSIBILITY

Assume there are n + 1 CID cycles and thus n interchanges. Call output of the CID cycle n - 1 M0 || M1 (where M0 is the first half of the message, M1 is the second half). Call the output of cycle n C0 || C1. The double vertical bar represents concatenation. M0 || M1 is transformed in the following manner by cycle n, which is the last cycle (the first is numbered 0). Confusion: A transformation T (M1) is applied. Which transformation depends on a bit of the key (one for each byte of M1) but since the same key bits will be accessed for the same byte positions during deciphering the specific transformations selected is irrelevent, as long as they are all one-to-one. Interruption: T (M1) is exclusive-ored with specific key bits KI. Diffusion: T (M1) + KI is exclusive-ored with the top half. The total message is thus T (M1) + KI + M0 || M1. Remember that on cycle n no interchange occurs. On deciphering, this output will be fed into decipher cycle 0, which is the same as encipher cycle n. Since this cycle is exactly the same as the last encipher cycle, confusion and interruption will generate T (M1) + KI just as before. When this is exclusive-ored with the top half consisting of T (M1) + KI + M0 the original M0 will be regenerated.

Since the interchange before encipher cycle n occurs after decipher cycle 0, the output from the interchange will also match. Thus the entire n - 1 interchange and n CID for encipher is equivalent to the 0 CID and 0 interchange. Thus these cycles can now be effectively stripped off; the same proof is applied to a Lucifer consisting of n CID cycles and n - 1 interchanges. Eventually a Lucifer of one CID cycle and zero interchanges remain; this has already been demonstrated above to be reversible.

In the actual specific operation of Lucifer, the diffusion operation does not consist of a simple exclusive-or; instead the bits are permuted in a fixed fashion before diffusion. This does not affect the reversibility, since the ciphertext will undergo the same permutation and thus each cycle will regenerate the input of the corresponding encipher cycle. However, this permutation is necessary for the cipher to be difficult to break. It ensures that small differences, say a one-bit change, in a given message block will propagate throughout all the bits of that block of ciphertext. Each bit of cleartext potentially affects every bit of ciphertext, within a 128-bit block.

THE MULTICS SOFTWARE IMPLEMENTATION

Two programs were written as implementations of the IBM hardware versions of Lucifer. One is a straightforward PL/I program which manipulates the bits in essentially the same fashion the hardware does. The other is a Multics assembly language program optimized for speed of execution. Details and listings of each may be found in the appendices. Instructions on using them are given here.

First, a key must be supplied. This is done by calling the set_key entry:

```
          declare  lucifer_$set_key entry (bit (128));
          call lucifer_$set_key (key);
```

This entry saves the key in internal static. This key will be used for all future enciphering and deciphering until set_key is called again.

To encipher:

```
          declare  lucifer_$encipher entry (dimension (*)
bit (128), dimension (*) bit (128), fixed binary precision
(35));
          call lucifer_$encipher (cleartext, ciphertext,
code);
```

The packed bit array, cleartext, is enciphered and deposited in the equal-sized array ciphertext. The code argument will be set to zero unless the dimensions of cleartext and ciphertext do not agree, in which case code will be set to one and the enciphering not performed. The ciphertext and cleartext may be the same variable.

To decipher:

```
call lucifer_$decipher (ciphertext, cleartext, code);
```

This entry is declared the same as encipher, and its operation is similar.

One problem with this implementation is that Lucifer requires a 128-bit block to encipher each 128-bit block of the cleartext. If the cleartext is not a multiple of 128 bits the last block could be padded with zeroes, but the output ciphertext corresponding to this block cannot be truncated. If it is information will be lost and it will not be deciphered correctly. This is because on decipher the truncated block will be padded to 128 bits (with zeroes, presumably) which is not identical to the original output of encipher before truncation. Therefore the primitive subroutines lucifer_$encipher and lucifer_$decipher require data to be passed in 128-bit blocks.

To make this more palatable to Multics users (to whom data tends to come in multiples of 9-bit characters or 36-bit words anyway) a command has been written to translate an entire segment. To set the key, type:

```
set_key -key-
```

where -key- will be padded or truncated to 128 bits and is an octal string.

To encipher a segment, type:

```
encipher -cleartext-  -ciphertext-
```

The segment whose relative pathname is -cleartext- will be enciphered. If the optional argument -ciphertext- is not given the original segment will be overwritten; otherwise the ciphertext will be written onto the segment named -ciphertext-.

The input will be padded to a mod 128 bit length with zeroes, and the output segment will be equal in length. Note that no additional pages can ever be required by this padding, since a page is 36*1024 bits long, a multiple of 128.

To decipher, type:

```
decipher -ciphertext-  -cleartext-
```

This command operates in the same way as encipher. Since the ciphertext segment must be a multiple of 128 bits long, exactly as produced by encipher, the output deciphered text will be exactly as long. This is because decipher has no way of knowing how long the original was. This can damage standard object segments which have significant words expected to be found at the end of the segment. Note that a better version of this command would encipher the original cleartext length into the ciphertext segment.

TIMING MEASUREMENTS AND CONCLUSIONS

One of the important questions addressed by this paper is "Is it possible to take an algorithm designed for easy hardware implementation and efficiently translate it to software?". Performance measurements by Feistel show that the Lucifer hardware module enciphered a 128-bit block in about 165 microseconds. A version written in 360 assembly language for the 360/67 required about 9 milliseconds. The current Multics hardware, the Honeywell model 6180, executes instructions at approximately the same rate as the IBM 360/67. The PL/I version, as expected, was extremely slow and required 10.4 seconds to encipher 72 blocks of 128 bits each, or 144 milliseconds/block. The assembly language version required .4 seconds/72 blocks, or 5.5 milliseconds/block. Multiplying by ten the number of blocks passed to lucifer_ did not substantially reduce the time/block, suggesting that 5.5 milliseconds represents real computation and not overhead. Since Multics characters are nine bits long, Lucifer requires 5.5 * (9/128) = 390 microseconds per character enciphered. Currently the Multics I/O system requires about 100 microseconds per character for its processing; thus if Lucifer were used for all I/O a severe performance degradation could occur. However this speed probably suffices for the occasional use to which it might be put.

There are some possibilities for further speed-up of the assembly language version; this is discussed in Appendix C.

APPENDIX A - OPERATION OF THE LUCIFER HARDWARE

This appendix explains the details of the operation of Lucifer as it was originally designed, as a hardware device. This material is drawn from J. Lynn Smith's "The Design of Lucifer, a Cryptographic Device for Data Communications".

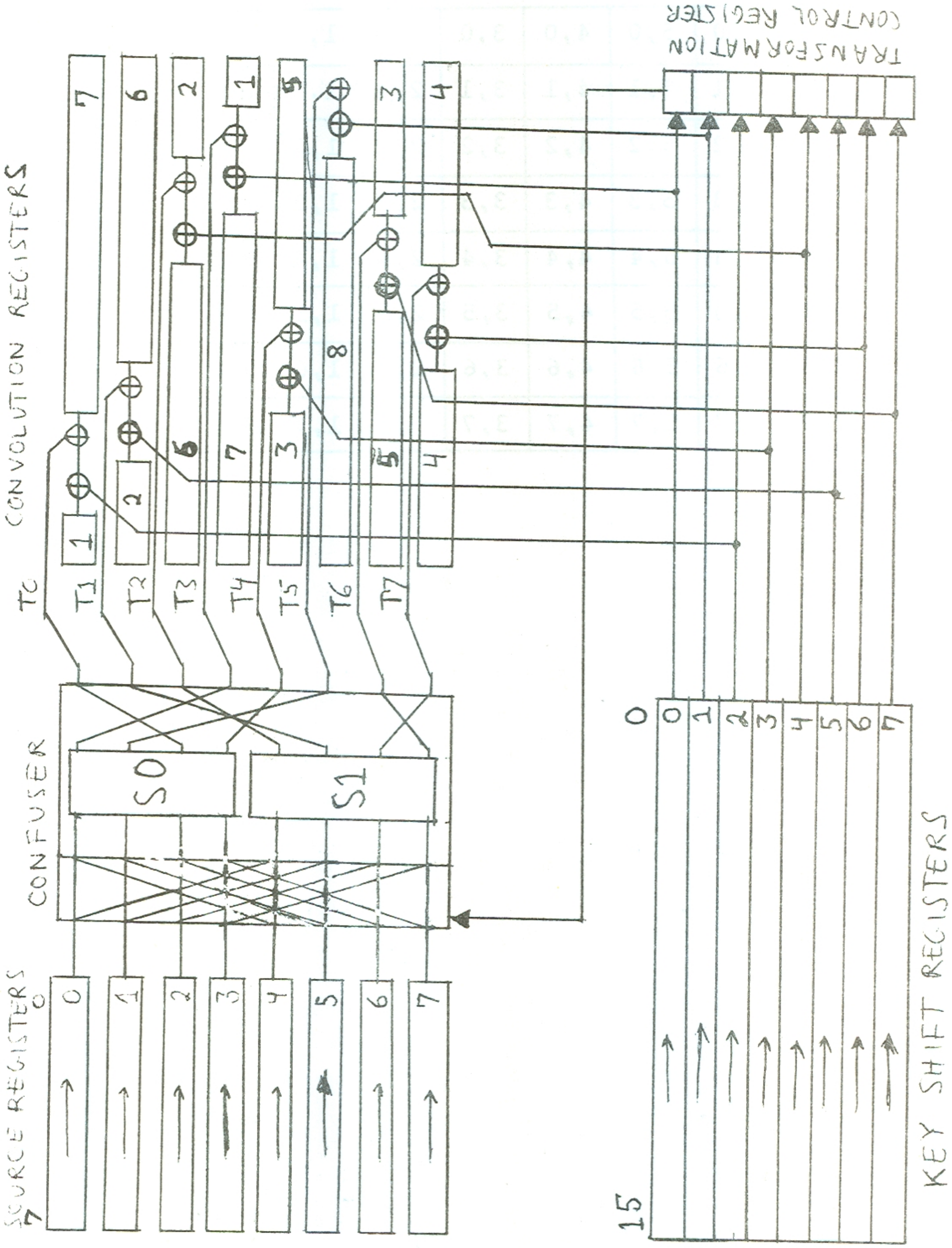
A copy of the PL/I program which implements the algorithm, duplicating very closely the exact bit flows within the hardware, is shown and explained in Appendix B.

Several cautions must be made in reading the hardware diagram given in figure 4. Individual bits of a given byte are arrayed vertically across registers; bytes are numbered right-to-left, bits of a byte top-to-bottom. Thus each vertical column below represents one byte of eight bits. Therefore if the bytes are adjacent (0, 1, 2...etc) the storage order in memory (in a two-dimensional array) is according to the ordered pairs in each bit position shown below.

Figure 3: Bit Addresses in Registers

7	6	5	4	3	2	1	0	byte / bit
7,0	6,0	5,0	4,0	3,0	2,0	1,0	0,0	0
7,1	6,1	5,1	4,1	3,1	2,1	1,1	0,1	1
7,2	6,2	5,2	4,2	3,2	2,2	1,2	0,2	2
7,3	6,3	5,3	4,3	3,3	2,3	1,3	0,3	3
7,4	6,4	5,4	4,4	3,4	2,4	1,4	0,4	4
7,5	6,5	5,5	4,5	3,5	2,5	1,5	0,5	5
7,6	6,6	5,6	4,6	3,6	2,6	1,6	0,6	6
7,7	6,7	5,7	4,7	3,7	2,7	1,7	0,7	7

Figure 4: Hardware Schematic



Note also that the author assumed that high-order bits are transmitted first; the Smith paper does not specify this. Thus bits are first loaded into position 0 of the convolution registers (top half), then position 1, 2 etc. on to position 0 of the source registers (bottom half).

Each of the registers shown is connected as a circular shift-register. In addition, bits can be shifted from the convolution registers to the source registers and back for the interchange operation.

A complete enciphering or deciphering operation for one 128-bit block consists of sixteen confusion-interruption-diffusion (CID) cycles, with an interchange cycle in between each CID cycle for a total of 15 interchange cycles.

At the start of a CID cycle, byte 0 of the key is copied into the transformation-control register. This register will supply eight bits for controlling the confusion operation; each bit will correspond with one byte of the source registers.

A CID cycle consists of eight shifts of the source, convolution, and transformation-control register (TCR). The TCR shifts vertically upward; other registers rotate horizontally, byte n going to byte mod (n - 1, 8).

An individual shift of a CID cycle occurs as follows. Byte 0 is taken from the source registers. It flows into the confusion box along with bit 0 of the TCR. A one-to-one transformation is applied to this byte, according to the bit from the TCR. The output from the confusion box is an eight-bit confused byte. Each bit of the confused byte is exclusive-ored with some bit of the convolution registers; note that no two bit positions are in the same byte. Each of these result bits is exclusive-ored with some bit of the rightmost byte of the key; this constitutes the interruption function. The result of this operation is stored in the bit position of the convolution registers to the right of the pair of exclusive-or gates. Note that diffusion occurs before interruption, but this is immaterial since mod 2 addition is commutative. As the result bit is stored in the convolution registers, the convolution registers, source registers, and TCR undergo a shift. Thus the bit that previously was to the right of the exclusive-or gates in the convolution registers is not destroyed; it is shifted right, and the result of diffusion occupies its old position.

These shifts are executed eight times for each CID cycle. In addition, during each shift the 16-byte key registers each rotate right one position with one exception: during the last shift of each CID cycle the key register is not rotated during encipher; during decipher the key registers rotate two positions after the last shift. Thus seven key shifts occur per CID cycle on encipher and nine key shifts occur per CID cycle on decipher. This, coupled with an initial shift of nine positions before processing any blocks, constitutes the only difference between enciphering and deciphering.

When eight shifts of one CID cycle are complete, the source registers will be back to their original position. The convolution registers are also restored except that each of its 64 bits has been exclusive-ored with exactly one key bit exclusive-ored with exactly one source bit. This is guaranteed by the placing of the gates in a different byte position for each bit of the confused byte. The key registers have been rotated either seven times (for encipher) or nine times (for decipher). The TCR has yielded all its bits. An interchange cycle now occurs, unless this is the last CID cycle. This consists of connecting positions 0 and 7 of the source registers with positions 7 and 0 of the convolution registers, respectively; eight shifts now occur. This merely swaps the contents of the registers.

Now the next CID cycle begins. A new key byte is fetched into the TCR. On CID cycle 1 this will be byte 7 for encipher and byte 2 for decipher of the original key.

It is important that the key bits be accessed in the reverse order (between CID cycles) when deciphering as compared to enciphering, but in the same order within each CID cycle. This is to ensure reversibility, as explained earlier. In addition, for cryptographic strength each bit of the key should be accessed an equal number of times:

eight times for interruption and once for transformation control of one byte of the source registers. The following method of accessing key bytes was thus devised. If there is to be an encipher, the key is initialized by loading it into the key registers. If a decipher is to be performed, the key registers are then rotated so that the first CID cycle will use bytes 9 to 0 rather than 0 to 7. After each CID cycle there will be no key shifts on encipher, but there will be two shifts during decipher. This will cause the key bytes to be accessed as shown in table 1.

Table 1: Key Byte Access Schedule

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

The byte of the key used for transformation control is in the left-hand column. Note that the decipher schedule is the same as the encipher schedule read upsidedown, but within a CID cycle, read horizontally, bytes are accessed in the same order. Also note that the key registers will be so positioned after sixteen CID cycles ready for the next block: in byte 0 for encipher, byte 9 for decipher.

The exact nature of the confusion operation has not been explained yet. It is not important particularly what it is, as long as it is one-to-one and sufficiently random. It works as follows. Each byte to be confused (from the source registers) is split into two four-bit halves. If the key bit from the TCR for this byte is 1, the two halves are exchanged; otherwise no operation is performed. Next, each four-bit half undergoes a one-to-one mapping. The method in hardware used decoders, encoders, and permuted wires, but effectively a table look-up was done to associate with each of the sixteen bit combinations a unique four-bit replacement. The two mappings for the two halves are different; the one for the top half is called S0 and the one for the bottom half is S1. Finally an 8-bit byte is generated by permuting the eight wires from these two mapping networks. The result of this entire confusion operation (and the way it is done in the software versions) is to consider the key bit concatenated with the source byte as a nine-bit index into a 512 element table. Each element is an eight-bit confused byte. This is explained in Appendix B, the PL/I implementation.

Table 2: Four-bit Permutations

input	S0	S1
0000	1100	0111
0001	1111	0010
0010	0111	1110
0011	1010	1001
0100	1110	0011
0101	1101	1011
0110	1011	0000
0111	0000	0100
1000	0010	1100
1001	0110	1101
1010	0011	0001
1011	0001	1010
1100	1001	0110
1101	0100	1111
1110	0101	1000
1111	1000	0101

APPENDIX B - THE PL/I IMPLEMENTATION

The PL/I implementation is very similar to the hardware design. However, instead of rotating data toward the low address end of each register, index values into fixed arrays are decremented and wrapped around to the high order end. Note very carefully that each byte shown in the hardware diagram, those bits arrayed vertically, are rows of two-dimensional arrays. Thus if a conventional PL/I array is printed it will appear transposed as compared to the map of the registers. For consistency within this document all arrays will be transposed from the conventional order so that they appear identical to the hardware bit orderings.

Instead of doing 15 interchanges (unlike most other operations, a real movement of data occurs on interchange) 16 are done. This last interchange is undone by copying the source registers first into the result block followed by the convolution registers. This is to avoid checking within the loop for the special case of the last execution. Similarly rather than skipping a key-shift cycle on encipher and performing an extra one on decipher each CID cycle, eight increments of the key index interruption_row are always performed. After a CID cycle is complete, a fixup variable either_one_or_minus_one is added modulo 16 to interruption_row; this variable is -1 for encipher and 1 for decipher.

The program operates as follows. It copies the first half of a given 128-bit block into the convolution_registers; the second half is copied into source_registers. The interchange_index loop counts the CID-interchange cycles, sixteen in number. Within that loop a CID cycle is performed by assigning interruption_row to ks_row; interruption_row shows which byte of the key will next be used for interruption, ks_row shows which byte will be used for transformation control. This assignment is the equivalent of copying the next byte of the key into the TCR at the start of a CID cycle. Now the data_row loops eight times, once for each byte in source_registers. The entire confusion operation is implemented by a 512 byte table; the first half for key bit = 0, the second half for key bit = 1. Thus the confused byte is found by indexing this table with the key bit identified by ks_row and data_row concatenated with the source byte identified by data_row. Now convolution_index loops eight times, once for each bit in the confused byte. Note that this is all done in parallel in the hardware version and in the assembly language version described in Appendix C. Each bit of the confused byte must be exclusive-ored with some bit of the key byte identified by interruption_row. Just as the key interruption wires were permuted in the hardware, so key_table tells which bit of that key byte is supplied for each bit of the confused byte. This interrupted bit is now exclusive-ored with some bit of the convolution registers. The register in which the bit lies which will be diffused (the one to the right of the exclusive-or gates) is the one corresponding to the source register from which the interrupted bit was derived. The number of this register, the column in the PL/I sense (although it is horizontal on the diagrams) is therefore convolution_index. The byte in which this bit lies is given by a table, convolution_table. These positions rotate right around the registers, one position for each shift of the CID cycle, once for each incrementing of data_row. Therefore the correct convolution_table entry for this bit of the interrupted byte must be mod-8 summed with data_row; this supplies the byte or row number of the target bit.

After this byte is complete, interruption_row is incremented mod 16 to simulate rotating the key registers once to the right. Now data_row is incremented to have the effect of rotating the source, convolution, and transformation-control registers.

After the eight loops of data_row, interruption_row must be readjusted to simulate only seven key shifts on encipher but nine shifts on decipher. As explained before, a fixup variable either_one_or_minus_one is mod 16 added to interruption_row; this fixup variable is set at the entry points. The two entry points also set the initial interruption_row, either 0 for encipher or 9 for decipher.

After sixteen loops of interchange_index, sixteen

CID-interchange pairs have been performed. The block is now copied into the result field; the source registers are copied first to undo the effect of the extra interchange cycle.

```
/*************************************************************************************************
 *                                                                                               *
 *                          Copyright (c) 1974, Massachusetts Institute of Technology            *
 *                          and Honeywell Information Systems, Inc.                              *
 *                                                                                               *
 *************************************************************************************************/

/* This module implements the Lucifer enciphering algorithm as developed by IBM.
   Initially coded by C. Gordon Benedict 04/26/74 at the Computer Systems Research division of Project MAC */

set_key:
          procedure (a_key);  /* this entry used to tell Lucifer what key to use */

declare   a_key parameter bit (128);     /* key user has */
declare   key bit (8) dimension (0 : 15) internal static;

          do data_row = 0 to 15;         /* iterate thru columns of key */
               do ks_row = 0 to 7;       /* iterate thru rows of key */
                    substr (key (data_row), ks_row + 1, 1) =           /* transpose */
                         substr (a_key, 16 * ks_row + data_row + 1, 1);
               end;
          end;
          return;

/* Declarations for enciphering and deciphering entries follow */

declare   (addr,
           bool,
           dim,
           fixed,
           mod,
           string,
           substr) builtin;


declare   (source_registers,  /* the source registers (bottom half) */
               convolution_registers)   /* convolution registers (top half) */
                    dimension (0 : 7) bit (8) unaligned;

declare   text_position fixed binary precision (24, 0);      /* bits of input string processed so far */
declare   (interchange_index, /* counts interchange cycles (0 - 15) */
           data_row,          /* what row of source or convolution register now munging */
           ks_row,            /* what row of key now using for transformation control */
           convolution_index, /* which bit of confused byte (during one CID) convolving now */
           convolution_row,   /* which row of convolution registers contains XOR gate (hardware hack) */
           interruption_row,  /* row of key used for interruption-diffusion */
           either_one_or_minus_one)      /* -1 for encipher, 1 for decipher */
                    fixed binary;

declare   confused_byte bit (8);         /* output of confuser (1 byte) */
declare   temp_register bit (64);        /* used merely for swapping source and convolution registers */

declare   convolution_table dimension (0 : 7)     /* which bit positions to mung in convolution registers */
```

```
             initial (7, 6, 2, 1, 5, 0, 3, 4) static internal fixed binary precision (3);
declare   key_table dimension (0 : 7)   /* gives permutation of key bits used for interruption */
             initial (2, 5, 4, 0, 3, 1, 7, 6) internal static fixed binary precision (3);

%include confusion_table;


encipher:          /* enciphering entry */
          entry (a_in, a_out, a_code);

declare   (a_in,               /* cleartext (ciphertext for decipher) */
           a_out) dimension (*) bit (128) parameter;         /* ciphertext (cleartext for decipher) */
declare   (a_in_ovly based (addr (a_in)),
           a_out_ovly based (addr (a_out))) bit (message_length) unaligned;
declare   message_length fixed binary precision (24);
declare   a_code fixed binary precision (35);      /* status code */

          either_one_or_minus_one = -1; /* amount to add after a CID cycle to
                                           interruption_row, because encipher resuses last byte */
          interruption_row = 0;          /* first byte of key to use is byte 0 */
          goto join;                     /* common code */

decipher:          /* deciphering entry -- note ciphertext is first arg */
          entry (a_in, a_out, a_code);

          either_one_or_minus_one = 1;  /* skip a byte of key when deciphering for each CID cycle */

          interruption_row = 9;          /* first byte of key to use when deciphering */

join:                        /* common section */
          message_length = dim (a_in, 1) * 128;    /* number of bits in input */
          if dim (a_out, 1) * 128 ¬= message_length then do;         /* barf at this */
               a_code = 1;
               return;
          end;

/* main loop follows. this consists of separately and independently processing each 128-bit
   block of input text (may be clear- or  cipher-text).  each block is processed by
  16 interchange cycles interspersed with 16 CIP (confusion-interruption-diffusion) cycles.
   For more details see IBM papers and my thesis. */

          do text_position = 0 by 128 while (text_position < message_length); /* each block */

               string (convolution_registers) = substr (a_in_ovly, text_position + 1, 64);
               string (source_registers) = substr (a_in_ovly, text_position + 65, 64);
               do interchange_index = 0 by 1 to 15;            /* 16 interchange cycles */

                    ks_row = interruption_row;     /* transformation control is first byte of key
                                                      used  for interruption in this CID cycle */
                    do data_row = 0 to 7;          /* process 8 bytes of input each CID cycle */

                         confused_byte =           /* look up in table to get confusion */
                              confusion_table (fixed (substr (key (ks_row), data_row +1, 1) ||
```

```
                              source_registers (data_row), 9, 0));

                         do convolution_index = 0 to 7;     /* convolve each bit of confused byte */

                              convolution_row =    /* for each cycle
                                                       convolution positions rotate around registers */
                                   mod (convolution_table (convolution_index) + data_row, 8);
                              substr (convolution_registers (convolution_row), convolution_index +1, 1) =
                                   bool (substr (key (interruption_row),
                                   key_table (convolution_index) + 1, 1),
                                   bool (substr (confused_byte, convolution_index +1, 1),
                                   substr (convolution_registers (convolution_row),
                                   convolution_index +1, 1), "0110"b), "0110"b);

                         end;

                         interruption_row   =         /* add 1 for next key byte with wraparound */
                              mod   (interruption_row + 1, 16);
                    end;

                    interruption_row =  /* on encipher, go back 1 byte, decipher, skip 1 */
                         mod (interruption_row + either_one_or_minus_one, 16);

/* swap source and convolution registers */

                    string (temp_register) = string (source_registers (*));
                    string  (source_registers (*)) = string (convolution_registers (*));
                    string (convolution_registers (*)) = string (temp_register);
               end;

               substr (a_out_ovly, text_position + 1, 64) = string (source_registers);
               substr (a_out_ovly, text_position + 65, 64) = string (convolution_registers);
          end;
          a_code = 0;
          return;
end set_key;
```

```
/* INCLUDE FILE confusion_table.incl.pl1
   This implements the confusion operation of Lucifer.
   It should only be used by lucifer.pl1          */

declare   confusion_table initial (
          "01010111"b, "11011111"b, "11001111"b, "11010011"b, "11010111"b, "01011111"b, "11011011"b, "01000011"b,
          "11000011"b, "11000111"b, "11001011"b, "01001011"b, "01011011"b, "01000111"b, "01001111"b, "01010011"b,
          "00010101"b, "10011101"b, "10001101"b, "10010001"b, "10010101"b, "00011101"b, "10011001"b, "00000001"b,
          "10000001"b, "10000101"b, "10001001"b, "00001001"b, "00011001"b, "00000101"b, "00001101"b, "00010001"b,
          "01110101"b, "11111101"b, "11101101"b, "11110001"b, "11110101"b, "01111101"b, "11111001"b, "01100001"b,
          "11100001"b, "11100101"b, "11101001"b, "01101001"b, "01111001"b, "01100101"b, "01101101"b, "01110001"b,
          "00110110"b, "10111110"b, "10101110"b, "10110010"b, "10110110"b, "00111110"b, "10111010"b, "00100010"b,
          "10100010"b, "10100110"b, "10101010"b, "00101010"b, "00111010"b, "00100110"b, "00101110"b, "00110010"b,
          "00010111"b, "10011111"b, "10001111"b, "10010011"b, "10010111"b, "00011111"b, "10011011"b, "00000011"b,
          "10000011"b, "10000111"b, "10001011"b, "00001011"b, "00011011"b, "00000111"b, "00001111"b, "00010011"b,
          "00110111"b, "10111111"b, "10101111"b, "10110011"b, "10110111"b, "00111111"b, "10111011"b, "00100011"b,
          "10100011"b, "10100111"b, "10101011"b, "00101011"b, "00111011"b, "00100111"b, "00101111"b, "00110011"b,
          "00010100"b, "10011100"b, "10001100"b, "10010000"b, "10010100"b, "00011100"b, "10011000"b, "00000000"b,
          "10000000"b, "10000100"b, "10001000"b, "00001000"b, "00011000"b, "00000100"b, "00001100"b, "00010000"b,
          "01010100"b, "11011100"b, "11001100"b, "11010000"b, "11010100"b, "01011100"b, "11011000"b, "01000000"b,
          "11000000"b, "11000100"b, "11001000"b, "01001000"b, "01011000"b, "01000100"b, "01001100"b, "01010000"b,
          "01110100"b, "11111100"b, "11101100"b, "11110000"b, "11110100"b, "01111100"b, "11111000"b, "01100000"b,
          "11100000"b, "11100100"b, "11101000"b, "01101000"b, "01111000"b, "01100100"b, "01101100"b, "01110000"b,
          "01110110"b, "11111110"b, "11101110"b, "11110010"b, "11110110"b, "01111110"b, "11111010"b, "01100010"b,
          "11100010"b, "11100110"b, "11101010"b, "01101010"b, "01111010"b, "01100110"b, "01101110"b, "01110010"b,
          "00010110"b, "10011110"b, "10001110"b, "10010010"b, "10010110"b, "00011110"b, "10011010"b, "00000010"b,
          "10000010"b, "10000110"b, "10001010"b, "00001010"b, "00011010"b, "00000110"b, "00001110"b, "00010010"b,
          "00110101"b, "10111101"b, "10101101"b, "10110001"b, "10110101"b, "00111101"b, "10111001"b, "00100001"b,
          "10100001"b, "10100101"b, "10101001"b, "00101001"b, "00111001"b, "00100101"b, "00101101"b, "00110001"b,
          "01010101"b, "11011101"b, "11001101"b, "11010001"b, "11010101"b, "01011101"b, "11011001"b, "01000001"b,
          "11000001"b, "11000101"b, "11001001"b, "01001001"b, "01011001"b, "01000101"b, "01001101"b, "01010001"b,
          "01110111"b, "11111111"b, "11101111"b, "11110011"b, "11110111"b, "01111111"b, "11111011"b, "01100011"b,
          "11100011"b, "11100111"b, "11101011"b, "01101011"b, "01111011"b, "01100111"b, "01101111"b, "01110011"b,
          "00110100"b, "10111100"b, "10101100"b, "10110000"b, "10110100"b, "00111100"b, "10111000"b, "00100000"b,
          "10100000"b, "10100100"b, "10101000"b, "00101000"b, "00111000"b, "00100100"b, "00101100"b, "00110000"b,
          "01010110"b, "11011110"b, "11001110"b, "11010010"b, "11010110"b, "01011110"b, "11011010"b, "01000010"b,
          "11000010"b, "11000110"b, "11001010"b, "01001010"b, "01011010"b, "01000110"b, "01001110"b, "01010010"b,
          "01010111"b, "11011111"b, "11001111"b, "11010011"b, "11010111"b, "01011111"b, "11011011"b, "01000011"b,
          "11000011"b, "11000111"b, "11001011"b, "01001011"b, "01011011"b, "01000111"b, "01001111"b, "01010011"b,
          "00010101"b, "10011101"b, "10001101"b, "10010001"b, "10010101"b, "00011101"b, "10011001"b, "00000001"b,
          "10000001"b, "10000101"b, "10001001"b, "00001001"b, "00011001"b, "00000101"b, "00001101"b, "00010001"b,
          "01110101"b, "11111101"b, "11101101"b, "11110001"b, "11110101"b, "01111101"b, "11111001"b, "01100001"b,
          "11100001"b, "11100101"b, "11101001"b, "01101001"b, "01111001"b, "01100101"b, "01101101"b, "01110001"b,
          "00110110"b, "10111110"b, "10101110"b, "10110010"b, "10110110"b, "00111110"b, "10111010"b, "00100010"b,
          "10100010"b, "10100110"b, "10101010"b, "00101010"b, "00111010"b, "00100110"b, "00101110"b, "00110010"b,
          "00010111"b, "10011111"b, "10001111"b, "10010011"b, "10010111"b, "00011111"b, "10011011"b, "00000011"b,
          "10000011"b, "10000111"b, "10001011"b, "00001011"b, "00011011"b, "00000111"b, "00001111"b, "00010011"b,
          "00110111"b, "10111111"b, "10101111"b, "10110011"b, "10110111"b, "00111111"b, "10111011"b, "00100011"b,
          "10100011"b, "10100111"b, "10101011"b, "00101011"b, "00111011"b, "00100111"b, "00101111"b, "00110011"b,
          "00010100"b, "10011100"b, "10001100"b, "10010000"b, "10010100"b, "00011100"b, "10011000"b, "00000000"b,
          "10000000"b, "10000100"b, "10001000"b, "00001000"b, "00011000"b, "00000100"b, "00001100"b, "00010000"b,
          "01010100"b, "11011100"b, "11001100"b, "11010000"b, "11010100"b, "01011100"b, "11011000"b, "01000000"b,
          "11000000"b, "11000100"b, "11001000"b, "01001000"b, "01011000"b, "01000100"b, "01001100"b, "01010000"b,
```

```
        "01110100"b, "11111100"b, "11101100"b, "11110000"b, "11110100"b, "01111100"b, "11111000"b, "01100000"b,
        "11100000"b, "11100100"b, "11101000"b, "01101000"b, "01111000"b, "01100100"b, "01101100"b, "01110000"b,
        "01110110"b, "11111110"b, "11101110"b, "11110010"b, "11110110"b, "01111110"b, "11111010"b, "01100010"b,
        "11100010"b, "11100110"b, "11101010"b, "01101010"b, "01111010"b, "01100110"b, "01101110"b, "01110010"b,
        "00010110"b, "10011110"b, "10001110"b, "10010010"b, "10010110"b, "00011110"b, "10011010"b, "00000010"b,
        "10000010"b, "10000110"b, "10001010"b, "00001010"b, "00011010"b, "00000110"b, "00001110"b, "00010010"b,
        "00110101"b, "10111101"b, "10101101"b, "10110001"b, "10110101"b, "00111101"b, "10111001"b, "00100001"b,
        "10100001"b, "10100101"b, "10101001"b, "00101001"b, "00111001"b, "00100101"b, "00101101"b, "00110001"b,
        "01010101"b, "11011101"b, "11001101"b, "11010001"b, "11010101"b, "01011101"b, "11011001"b, "01000001"b,
        "11000001"b, "11000101"b, "11001001"b, "01001001"b, "01011001"b, "01000101"b, "01001101"b, "01010001"b,
        "01110111"b, "11111111"b, "11101111"b, "11110011"b, "11110111"b, "01111111"b, "11111011"b, "01100011"b,
        "11100011"b, "11100111"b, "11101011"b, "01101011"b, "01111011"b, "01100111"b, "01101111"b, "01110011"b,
        "00110100"b, "10111100"b, "10101100"b, "10110000"b, "10110100"b, "00111100"b, "10111000"b, "00100000"b,
        "10100000"b, "10100100"b, "10101000"b, "00101000"b, "00111000"b, "00100100"b, "00101100"b, "00110000"b,
        "01010110"b, "11011110"b, "11001110"b, "11010010"b, "11010110"b, "01011110"b, "11011010"b, "01000010"b,
        "11000010"b, "11000110"b, "11001010"b, "01001010"b, "01011010"b, "01000110"b, "01001110"b, "01010010"b
                ) bit (8) unaligned dimension (0 : 511) internal static;

/* END INCLUDE FILE confusion_table.incl.pl1 */
```

APPENDIX C - THE ASSEMBLY LANGUAGE IMPLEMENTATION

The basic philosophy of the Multics assembly language version of Lucifer was to produce a program which could encipher or decipher at the highest speed. This does not contribute to the readibility of the program; therefore this explanation is quite detailed. If the reader is unfamiliar with Multics assembly language, a short introduction is given in Appendix D.

The set_key entry does more than store the key in internal static. During ciphering the key is used in two places: transformation control and interruption. For reasons explained later, each purpose requires the key to be in a different format for optimal operation. To avoid key manipulation during ciphering, set_key stores the key in two variables, key and exploded_key.

In exploded_key each bit of the key is given its own nine-bit byte. The high-order bit of each byte contains the key bit; the low order eight bits are zero. This key is for transformation control. In the diagram below showing the storage assignment, the ordered pair in each byte position gives the byte of the key number and the bit within the byte. As in the hardware diagrams adjacent bits of a byte are arrayed vertically, although it is more conventional to show memory words horizontally. Thus each byte of the key requires two words; thirty-two words for 128 bits.

Figure 5: Exploded Key Bit Assignment

30	28	26	24	22	20	18	16	14	12	10	8	6	4	2	0	word
120	112	104	96	88	80	72	64	56	48	40	32	24	16	8	0	byte
121	113	105	97	89	81	73	65	57	49	41	33	25	17	9	1	
122	114	106	98	90	82	74	66	58	50	42	34	26	18	10	2	
123	115	107	99	91	83	75	67	59	51	43	35	27	19	11	3	
124	116	108	100	92	84	76	68	60	52	44	36	28	20	12	4	
125	117	109	101	93	85	77	69	61	53	45	37	29	21	13	5	
126	118	110	102	94	86	78	70	62	54	46	38	30	22	14	6	
127	119	111	103	95	87	79	71	63	55	47	39	31	23	15	7	

For interruption, the key bits within a key byte are not accessed in the same order as the confused byte's bits, 0, 1, 2...7. Rather they are accessed 2, 5, 4, 0, 3, 1, 7, 6 as given in key_table of the PL/I program or as shown by the wiring of the hardware. To avoid the use of such a table and lookup time during ciphering, the key bytes are presorted by set_key. Each 8-bit byte of the key is stored in the high order part of a Multics 9-bit byte, the remaining bit being zero. Thus the storage assignment is as shown in the diagram below.

Figure 6: Key Bit Assignment

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

Words 0 and 1 are copied into words 4 and 5. This is to permit directly addressing eight bytes starting at any byte between 0 and 15 without programming a complicated wraparound routine.

The basic idea underlying this program is to process all 64 bits of the source and convolution registers at once, each CID cycle. In order to do this, the key bits must be so arranged that each of its bits lies in the bit position corresponding to that of the source register bit with which it will be exclusive-ored during interruption. This explains the rearranging above.

When the encipher entry is called, it sets interruption_row (held in index register 2) to zero as in the PL/I program. Since an entire CID cycle is done in parallel, interruption_row will never be incremented along the horizontal line of the key byte access schedule given earlier. Instead it will be incremented each CID cycle to assume the values given in the schedule's left-hand column. Examining the schedule it can be seen that interruption_row should thus be incremented by 7 for encipher and -7 for decipher, modulo 16. Thus each entry also sets the variable either_7_or_minus_7 to the appropriate value. This is added to x2 mod 16 each CID cycle.

After the argument extents are calculated and pointers to the strings fetched (bp -> input string, bb -> output string), the main loop is entered.

As in the PL/I program, the first 64 bits of each 128-bit block are placed into convolution_registers, the next 64 into source_registers. As with the key, each 8-bit byte is placed in the high order eight bits of a Multics 9-bit byte. This unpacking is accomplished by unpack_loop. This loop depends on the fact that the assembler will assign source_registers a location after convolution_registers because it is declared afterward. The low order (high address) bytes are unpacked first.

Once this is complete, sixteen CID-interchange pairs are executed.

First, the convolution registers are prepared for the diffusion operation. Referring to the hardware diagram, one can see that each bit of a confused, interrupted byte (vertically arrayed) corresponds to a different byte but the same bit (i.e., horizontal register) of the convolution registers. As seen in the PL/I program, if a source register bit has address [i, j] (byte i, bit j) the convolution register bit corresponding to it is [mod (i + convolution_table [j], 8), j]

where convolution_table is [7, 6, 2, 1, 5, 0, 3, 4]. Instead of looping through each bit as the PL/I program does, the convolution registers are rotated so the bit positions for diffusions line up, corresponding with those of the source registers.

Since the horizontal registers are the bits to rotate, the bits to rotate are not adjacent. Thus the bit addresses within the two-word convolution_registers of each bit before rotation is as follows:

Figure 7: Convolution Registers

7	6	5	4	3	2	1	0	byte bit
63	54	45	36	27	18	9	0	0
64	55	46	37	28	19	10	1	1
65	56	47	38	29	20	11	2	2
66	57	48	39	30	21	12	3	3
67	58	49	40	31	22	13	4	4
68	59	50	41	32	23	14	5	5
69	60	51	42	33	24	15	6	6
70	61	52	43	34	25	16	7	7

Notice that bits 8, 17, 26... 71 do not appear assigned on the matrix. This is due to the unpacking of each 8-bit byte to a 9-bit byte. The unassigned offsets are those of the pad bits. The purpose of this rotation is to align all the exclusive-or positions on the right edge of the matrix. Looking at the hardware schematic, the desired position of each bit is as follows:

Figure 8: Postrotation Convolution Registers

7	6	5	4	3	2	1	0	byte bit
6,0	5,0	4,0	3,0	2,0	1,0	0,0	7,0	0
5,1	4,1	3,1	2,1	1,1	0,1	7,1	6,1	1
1,2	0,2	7,2	6,2	5,2	4,2	3,2	2,2	2
0,3	7,3	6,3	5,3	4,3	3,3	2,3	1,3	3
4,4	3,4	2,4	1,4	0,4	7,4	6,4	5,4	4
7,5	6,5	5,5	4,5	3,5	2,5	1,5	0,5	5
2,6	1,6	0,6	7,6	6,6	5,6	4,6	3,6	6
3,7	2,7	1,7	0,7	7,7	6,7	5,7	4,7	7

This rotation is accomplished as follows. Row 0 (bits 0, 9, 18... 63) must be rotated right on the diagram (left in the AQ register as it happens) seven positions or 63 bits. Row 1 (bits 1, 10, 19... 64) must be rotated 6 positions or 54 bits, etc. An array of masks, and_masks, has been prepared with a 1-bit in each bit position for a given register. They are ordered according to the number of positions of rotation needed. Since register 5 needs no rotation (because the exclusive-or gate is already in byte 0), the mask for it occurs first. It consists of four zeroes, a one, eight zeroes, a one, eight zeroes... Thus, when convolution_registers is loaded into the AQ register and is ANDed with this mask, only bits 5, 14, 23... 68 will remain. This register is rotated 0 bits left and then ORed into a previously zeroed doubleword, named "normalized".

Next, register 3 must be rotated left one position or nine bits. Thus the second mask has a one in bit 3 and a one every nine bits thereafter. After ANDing the convolution_registers with this mask only bits 3, 12, 21... 66 remain. The AQ is rotated left nine bits, and ORed into "normalized".

There is a pointer to and_masks called and_masks_ptr. It is referenced by using the add-delta (AD) type indirect reference. When an indirect reference is made through this word, after completion of the specified operation the contents of the delta field (here 2) will be added to the address field. Thus the next time the AQ is ANDed the next doubleword mask will be used. Similarly an AD word controls the shift count. The first time through the loop the AQ must be shifted zero bits so the address field of this word contains zero. After every indirect reference the address field will be incremented by the delta field, here nine. Thus the rotate counts will be 0, 9, 18... 63. In addition this word is used to control the number of times the loop will execute. After an add-delta reference is made the tally field of the word is decremented by one; if it reaches zero the tally runout indicator is set. This tally field is set to eight before beginning the loop. Thus the loop will iterate eight times, due to the transfer-tally-runout-flag off instruction at the end.

After preparing the convolution registers, the confusion operation is performed on the source registers. This is done by loading the source registers into the AQ and shifting right one bit position. Now each 8-bit byte appears right justified in each Multics 9-bit byte of the AQ. The AQ is now ORed with some doubleword of exploded_key. Each bit of exploded_key occupies the high order bit of a 9-bit byte; thus each bit to be used for transformation control now resides to the left of the corresponding byte of the source.

The doubleword of exploded_key to use for transformation control is equal to the byte of the key addressed by interruption_row. This is because each byte of the key uses a doubleword of exploded_key, and because interruption_row (in x2) always addresses the first byte of the key to use for interruption this CID cycle which is also the byte to use for transformation control. Since even the doubleword instructions address in word indexes, interruption_row must be doubled. This is done by adding it in twice, once in the epplb instruction and once in the oraq instruction itself.

The AQ is stored and translated by the mvt instruction. The confusion_table used here is identical to the one in the PL/I program, except that each 8-bit result byte is as usual left justified within a 9-bit byte.

These confused bytes are now interrupted by exclusive-oring with the eight bytes of the key addressed by interruption_row. Diffusion is obtained by exclusive-oring with the prerotated convolution registers stored in "normalized".

The interchange operation must, as well as swapping the source and convolution (now stored in "normalized"), unrotate the convolution registers to undo the effect of lining up the exclusive-or gates described above. This is done via a very similar loop to rotate_loop. A subtract-delta modifier references through and_masks_ptr. Since this modifier subtracts delta before indirecting the masks will be used in the reverse order. The shift counts needed are shown below; the add-delta word for shifting again supplies loop control.

Table 3: Convolution Register Rotation Counts

Row	Previous Rotation	Post-Rotation
5	0	72
3	9	63
2	18	54
6	27	45
7	36	36
4	45	27
1	53	18
0	63	9

The register accesses and rotate counts for the prerotating should be read down; for postrotation the table should be read up.

After sixteen CID-interchange pairs, one more interchange has been done than desired. This is undone by swapping the two registers. The bytes are now packed into the result field.

Some possibilities still exist for speeding up this program. The two loops controlled by tally words only loop eight times; they could be exploded into eight copies. Since the address of and_masks and the rotate counts would in each copy be known at compile time no indirect words would be needed. In addition the loop control instruction ttf would be eliminated. Counting ttf as two memory accesses and each of the tally references as one, four memory accesses could be saved each rotation. Since eight are required in the loop, and there are two loops, 64 memory accesses would be saved. Eight more would be saved by eliminating the tally word setup instructions at the beginning of each loop, for a total of 72. Since there are sixteen CID cycles a total of 72 times 16 = 1152 memory cycles might be saved. This may total as much as a millisecond, thus saving about twenty percent of the cipher time for a given block. This demonstrates how sensitive a program's performance can be to minor changes in coding style. Other experiments are suggested, such as completely rewriting the program with all arrays transposed (so that the bits of a byte are not stored sequentially), or eliminating the padding bit on each byte.

```
"                    Copyright (c) 1974 by Massachusetts Institute of Technology and
"                    Honeywell Information Systems, Inc.

" This program is a special version of Lucifer designed to run very quickly.
" Few programs could compete with this for obscurity.
"          Coded May 1, 1974 ' ' G. Gordon Benedict
"           at the Computer Systems Research division of Project MAC

          entry     set_key,encipher,decipher
          equ       move,3
          equ       a_in,2
          equ       a_out,4
          equ       a_code,6
          equ       a_in_desc,8
          equ       a_out_desc,10
          temp      text_length,text_position,either_7_or_minus_7,shift_word
          tempd     convolution,source,confused_bytes,normalized
          temp      initial_value

encipher:
          push
          eax2      0                   initial interruption row
          eax7      7                   go forward 7 bytes in key after each CID cycle
          stx7      either_7_or_minus_7
          tra       join-*,ic

decipher:
          push
          eax2      9                   initial interruption row (ninth byte of key)
          eax7      -7                  start each CID cycle with interruption row 7
          stx7      either_7_or_minus_7 more than last for later

join:
          stx2      initial_value       termination condition after 16 CID cycles
          eax0      0                   assume no display ptr in arg list
          lxl7      ap|0                get code which tells us if assumption is operative
          cmpx7     8,du                is there a display ptr
          tnz       2,ic                no
          eax0      2                   yes, put length of this ptr in x0 so we will skip it

          eppbp     ap|a_in_desc,0*     get ptr to descriptor
          ldq       bp|2                hbound (a_in)...
          sbq       bp|1                - lbound (a_in)...
          adq       1,dl                + 1 = dim (a_in, 1)
          qls       7                   * 128 = length in bits of whole array
          stq       text_length
          eppbp     ap|a_out_desc,0*    get ptr to descriptor
          ldq       bp|2                hbound (a_out)...
          sbq       bp|1                - lbound (a_out)...
          adq       1,dl                + 1 = dim (a_out, 1)
          qls       7                   * 128 = length in bits of whole array
          cmpq      text_length
          tnz       no_length_match-*,ic         error, both must be same
```

```
        epphp     apla_in,*              get ptr to input arg
        epphb     apla_out,*             get ptr to output arg
```

```
" begin main loop processing. read in each
" 128-bit block and encrypt separately.

          stz       text_position       zero processed so far
text_loop:
          ldq       text_position       get amount processed so far
          cmpq      text_length         see if handled all in string
          tpl       return_now-*,ic     if so, return
" unpack next 128-bit block such that each
" 8-bit byte occupies the high order 8
" bits of a Multics 9-bit block.
" this makes manipulation by FIS instructions convenient.
          adq       15*8,dl             get position of last 8-bit byte in this block
          lda       15*9,dl             get offset to last 9-bit block in registers
unpack_loop:
          csl       (pr,ql),(pr,al),bool(move),fill(0)
          desch     bp|0,8              move an 8-bit byte...
          desch     convolution,9             ...to a 9-bit byte and stick on a "0"b

          sbq       8,dl                go to next lower 8-bit byte
          sba       9,dl                same for target
          tpl       unpack_loop-*,ic    continue until 16 bytes are unpacked,
"                                       8 in source, 8 in convolution
" now do 15 interchange and 16 CIP cycles.
interchange_loop:
          fld       0,dl                zero A0 (kludge)
          staq      normalized          make zero for oring
          lda       =o001011,dl         tally = 8, initial value = 0, delta = 9
          sta       shift_word          AR word for shifting (increments 9 each time)

rotate_loop:
          ldaq      convolution         get entire convolution regs (bits 0 - 63)
          anaq      lp|and_masks_ptr,ad clear all but columns 5, then 0, 1, 4, 7, 6, 2, 3
          llr       shift_word,ad       shift first by 0, then 9, then 18...etc.
          orsa      normalized          put in first word's bits
          orsq      normalized+1        now 2nd word
          ttf       rotate_loop-*,ic    do 8 times (see tally)
" now have in normalized a copy of convolution
" registers with each column so rotated
" that all the XOR gates are aligned on the right hand edge. now confuse source
          epplb     lp|exploded_key,x2  when x2 is added to this addr,
"                                       will have addr of key words
          ldaq      source              get source reg
          lrl       1                   put 0 at left edge of each byte instead of right
          oraq      lb|0,x2             put each bit of ks-row key in high order bit of source byte
          staq      confused_bytes

          mvt       (pr),(pr)           translate via table (confusion)
          desc9a    confused_bytes,8
          desc9a    confused_bytes,8
          arg       confusion_table+3-*,ic
```

```
          ldaq      confused_bytes
          mlr       (pr,x2),(pr)          get row of key used for interruption
          desc9a    lp|key,8
          desc9a    confused_bytes,8

          eraq      confused_bytes        interruption

          ersa      normalized            diffusion
          ersq      normalized+1          2nd word
"now do interchange cycle.

          ldaq      source
          staq      convolution           one half of work
          fld       0,dl                  zero out source for oring in
          staq      source

          lda       =o000011001011        tally = 8, delta = 9, initial value = 9
          sta       shift_word            put back tally of 8
unrotate_loop:
          ldaq      normalized            get diffused convolution registers
          anaq      lp|and_masks_ptr,sd          and out all but that column to be rotated
          llr       shift_word,ad         shift by appropriate amount
          orsa      source                put into source
          orsq      source+1              2nd word
          ttf       unrotate_loop-*,ic

          adx2      either_7_or_minus_7 go forward or backward thru key
          anx2      =o17,du               mod 16

          cmpx2     initial_value         back to where we started this block
          tnz       interchange_loop-*,ic

" done with this 128-bit block. recompact and store
          ldaq      source                exchange source and convolution
          staq      normalized
          ldaq      convolution
          staq      source
          ldaq      normalized
          staq      convolution
          ldq       text_position         go to next 128-bit block
          adq       128,dl
          stq       text_position
          lda       9*15,dl               16 9-bit bytes to pack
pack_loop:
          sha       8,dl                  go to next lower byte
          csl       (pr,al),(pr,al),bool(move),fill(0)
          desch     convolution,9
          desch     bb|0,8

          sha       9,dl                  go to next lower 9-bit bytes
          tpl       pack_loop-*,ic
          tra       text_loop-*,ic        go to next 128-bit block
```

```
no_length_match:                         "bomb, lengths of input and output not same
        ldq     1,dl                     code to return
        stq     ap|a_code,*
        return

return_now:
        stz     ap|a_code,*
        return


" set_key entry, to set the key for subsequent calls to lucifer.
set_key:
        epbpb   ap|2,*                   get addr of 128-bit string which is key
" explode the key and transpose it,
" so each bit occupies the first bit of a 9-bit byte
        eax0    0                        first bit of key
        eax1    0                        first byte of exploded key

explode_loop:
        csl     (pr,x0),(pr,x1),bool(move),fill(0)
        desc9b  bp|0,1                   move one bit of key...
        desc9b  lp|exploded_key,9        ...to the top bit of a 9-bit byte

        eax1    9,x1                     next time use next byte of exploded_key
        eax0    16,x0                    take next column entry, 16 bits away
        cmpx0   128,du                   see if done with this column
        tmi     explode_loop-*,ic        done
" just finished one column of 8 bits. now do next column, starting one bit away
        eax0    -127,x0                  put us back 127 bits, offset 1 from previous beginning
        cmpx0   16,du                    if 16, we have swept thru all bits (16 = 127 + 16 - 127)
        tmi     explode_loop-*,ic

" now explode each 8-bit permuted block to a 9-bit row
        eax0    0                        first column of key
        eax1    0
        eax2    0
permutation_loop:
        eax3    0,x0                     copy column of key
        adx3    permutation_table,x2             get specific bit number
        csl     (pr,x3),(pr,x1),bool(move),fill(0)
        desc9b  bp|0,1
        desc9b  lp|key,2                 pad with a 0 bit (only counts at end of loop)

        eax1    1,x1                     go to next bit of key result
        eax2    1,x2                     next permutation_table entry
        cmpx2   8,du                     done with this loop
        tmi     permutation_loop-*,ic
" did one 8-bit block. skip last zero bit
        eax1    1,x1
        eax0    1,x0
        cmpx0   16,du
        tmi     permutation_loop-*-1,ic
```

```
" duplicate first 8 rows of key at end to prevent wraparound problems
          ldaq      lp|key
          staq      lp|key+4
" set up the initial tally word used for running down and-masks
          eaa       lp|and_masks
          orsa      lp|and_masks_ptr

          short_return
permutation_table:            "gives permutations of key columns used for interruption
          arg       16*2
          arg       16*5
          arg       16*4
          arg       16*0
          arg       16*3
          arg       16*1
          arg       16*7
          arg       16*6
```

```
confusion_table:
          include   confusion_table


          use       linkage_section
          even
          bss       key,6
          bss       exploded_key,32
and_masks_ptr:
          dec       2         delta of 2
          even                "need on even word boundary
and_masks:
          vfd       6/1,9/1,9/1,9/1,9/1,9/1,9/1,9/1
          vfd       4/1,9/1,9/1,9/1,9/1,9/1,9/1,9/1
          vfd       3/1,9/1,9/1,9/1,9/1,9/1,9/1,9/1
          vfd       7/1,9/1,9/1,9/1,9/1,9/1,9/1,9/1
          vfd       8/1,9/1,9/1,9/1,9/1,9/1,9/1,9/1
          vfd       5/1,9/1,9/1,9/1,9/1,9/1,9/1,9/1
          vfd       2/1,9/1,9/1,9/1,9/1,9/1,9/1,9/1
          vfd       1/1,9/1,9/1,9/1,9/1,9/1,9/1,9/1
          join      /link/linkage_section
          end
```

```
" INCLUDE FILE confusion_table.incl.alm
" This implements the confusion operation for Lucifer
" It should only be called from lucifer_.alm

        vfd     9o/256,9o/676,9o/636,9o/646,9o/656,9o/276,9o/666,9o/206
        vfd     9o/606,9o/616,9o/626,9o/226,9o/266,9o/216,9o/236,9o/246
        vfd     9o/052,9o/472,9o/432,9o/442,9o/452,9o/072,9o/462,9o/002
        vfd     9o/402,9o/412,9o/422,9o/022,9o/062,9o/012,9o/032,9o/042
        vfd     9o/352,9o/772,9o/732,9o/742,9o/752,9o/372,9o/762,9o/302
        vfd     9o/702,9o/712,9o/722,9o/322,9o/362,9o/312,9o/332,9o/342
        vfd     9o/154,9o/574,9o/534,9o/544,9o/554,9o/174,9o/564,9o/104
        vfd     9o/504,9o/514,9o/524,9o/124,9o/164,9o/114,9o/134,9o/144
        vfd     9o/056,9o/476,9o/436,9o/446,9o/456,9o/076,9o/466,9o/006
        vfd     9o/406,9o/416,9o/426,9o/026,9o/066,9o/016,9o/036,9o/046
        vfd     9o/156,9o/576,9o/536,9o/546,9o/556,9o/176,9o/566,9o/106
        vfd     9o/506,9o/516,9o/526,9o/126,9o/166,9o/116,9o/136,9o/146
        vfd     9o/050,9o/470,9o/430,9o/440,9o/450,9o/070,9o/460,9o/000
        vfd     9o/400,9o/410,9o/420,9o/020,9o/060,9o/010,9o/030,9o/040
        vfd     9o/250,9o/670,9o/630,9o/640,9o/650,9o/270,9o/660,9o/200
        vfd     9o/600,9o/610,9o/620,9o/220,9o/260,9o/210,9o/230,9o/240
        vfd     9o/350,9o/770,9o/730,9o/740,9o/750,9o/370,9o/760,9o/300
        vfd     9o/700,9o/710,9o/720,9o/320,9o/360,9o/310,9o/330,9o/340
        vfd     9o/354,9o/774,9o/734,9o/744,9o/754,9o/374,9o/764,9o/304
        vfd     9o/704,9o/714,9o/724,9o/324,9o/364,9o/314,9o/334,9o/344
        vfd     9o/054,9o/474,9o/434,9o/444,9o/454,9o/074,9o/464,9o/004
        vfd     9o/404,9o/414,9o/424,9o/024,9o/064,9o/014,9o/034,9o/044
        vfd     9o/152,9o/572,9o/532,9o/542,9o/552,9o/172,9o/562,9o/102
        vfd     9o/502,9o/512,9o/522,9o/122,9o/162,9o/112,9o/132,9o/142
        vfd     9o/252,9o/672,9o/632,9o/642,9o/652,9o/272,9o/662,9o/202
        vfd     9o/602,9o/612,9o/622,9o/222,9o/262,9o/212,9o/232,9o/242
        vfd     9o/356,9o/776,9o/736,9o/746,9o/756,9o/376,9o/766,9o/306
        vfd     9o/706,9o/716,9o/726,9o/326,9o/366,9o/316,9o/336,9o/346
        vfd     9o/150,9o/570,9o/530,9o/540,9o/550,9o/170,9o/560,9o/100
        vfd     9o/500,9o/510,9o/520,9o/120,9o/160,9o/110,9o/130,9o/140
        vfd     9o/254,9o/674,9o/634,9o/644,9o/654,9o/274,9o/664,9o/204
        vfd     9o/604,9o/614,9o/624,9o/224,9o/264,9o/214,9o/234,9o/244
        vfd     9o/256,9o/676,9o/636,9o/646,9o/656,9o/276,9o/666,9o/206
        vfd     9o/606,9o/616,9o/626,9o/226,9o/266,9o/216,9o/236,9o/246
        vfd     9o/052,9o/472,9o/432,9o/442,9o/452,9o/072,9o/462,9o/002
        vfd     9o/402,9o/412,9o/422,9o/022,9o/062,9o/012,9o/032,9o/042
        vfd     9o/352,9o/772,9o/732,9o/742,9o/752,9o/372,9o/762,9o/302
        vfd     9o/702,9o/712,9o/722,9o/322,9o/362,9o/312,9o/332,9o/342
        vfd     9o/154,9o/574,9o/534,9o/544,9o/554,9o/174,9o/564,9o/104
        vfd     9o/504,9o/514,9o/524,9o/124,9o/164,9o/114,9o/134,9o/144
        vfd     9o/056,9o/476,9o/436,9o/446,9o/456,9o/076,9o/466,9o/006
        vfd     9o/406,9o/416,9o/426,9o/026,9o/066,9o/016,9o/036,9o/046
        vfd     9o/156,9o/576,9o/536,9o/546,9o/556,9o/176,9o/566,9o/106
        vfd     9o/506,9o/516,9o/526,9o/126,9o/166,9o/116,9o/136,9o/146
        vfd     9o/050,9o/470,9o/430,9o/440,9o/450,9o/070,9o/460,9o/000
        vfd     9o/400,9o/410,9o/420,9o/020,9o/060,9o/010,9o/030,9o/040
        vfd     9o/250,9o/670,9o/630,9o/640,9o/650,9o/270,9o/660,9o/200
        vfd     9o/600,9o/610,9o/620,9o/220,9o/260,9o/210,9o/230,9o/240
        vfd     9o/350,9o/770,9o/730,9o/740,9o/750,9o/370,9o/760,9o/300
```

```
        vfd     9o/700,9o/710,9o/720,9o/320,9o/360,9o/310,9o/330,9o/340
        vfd     9o/354,9o/774,9o/734,9o/744,9o/754,9o/374,9o/764,9o/304
        vfd     9o/704,9o/714,9o/724,9o/324,9o/364,9o/314,9o/334,9o/344
        vfd     9o/054,9o/474,9o/434,9o/444,9o/454,9o/074,9o/464,9o/004
        vfd     9o/404,9o/414,9o/424,9o/024,9o/064,9o/014,9o/034,9o/044
        vfd     9o/152,9o/572,9o/532,9o/542,9o/552,9o/172,9o/562,9o/102
        vfd     9o/502,9o/512,9o/522,9o/122,9o/162,9o/112,9o/132,9o/142
        vfd     9o/252,9o/672,9o/632,9o/642,9o/652,9o/272,9o/662,9o/202
        vfd     9o/602,9o/612,9o/622,9o/222,9o/262,9o/212,9o/232,9o/242
        vfd     9o/356,9o/776,9o/736,9o/746,9o/756,9o/376,9o/766,9o/306
        vfd     9o/706,9o/716,9o/726,9o/326,9o/366,9o/316,9o/336,9o/346
        vfd     9o/150,9o/570,9o/530,9o/540,9o/550,9o/170,9o/560,9o/100
        vfd     9o/500,9o/510,9o/520,9o/120,9o/160,9o/110,9o/130,9o/140
        vfd     9o/254,9o/674,9o/634,9o/644,9o/654,9o/274,9o/664,9o/204
        vfd     9o/604,9o/614,9o/624,9o/224,9o/264,9o/214,9o/234,9o/244

" END INCLUDE FILE confusion_table.incl.alm
```

APPENDIX D - INTRODUCTION TO MULTICS ASSEMBLER

This section is intended to be a quick introduction to the Honeywell model 6180 processor for those who are unfamiliar with its machine language.

The 6180 is a word-addressed machine with a 36-bit word; it also possesses some very powerful bit string and character string handling instructions. There are two major arithmetic registers of 36 bits each, the accumulator (A) and the quotient (Q) registers. These may be coupled to form a double length register, the AQ. Instructions ending in A, Q, or AQ operate on the corresponding registers.

There are in addition eight index registers of eighteen bits each. Instructions ending in xN where N is an octal digit operate on these registers. Most index register instructions take a storage operand in the top half of a word, except for sxlN (store xN in lower half) and lxlN (load index N from lower half).

There exist eight pointer registers for generating segment number - word number pairs. These registers contain a character offset and a bit offset from the addressed word for the use of character string and bit string instructions. The names of these registers (in numeric address order) are ap, ab, bp, bb, lp, lb, sp and sb. The ap points to a procedure's argument list. The lp points to the procedure's linkage section where internal static variables are kept, such as the key. The sp points at the stack frame, in which automatic variables are kept. Variables declared in a "temp" or "tempd" pseudoop are placed in the stack frame by the assembler and are given one or two words each respectively. A temp variable may also be given a subscript in which case it will be assigned that many words. Declaration in a temp or tempd implies an sp reference. The other pointer registers are used for spare registers; for example, the bp points at the input string and the bb points at the output string.

A sample instruction would be

```
        ldq       lp|foo
```

This instruction will load the Q register with the internal static (because of the lp reference) variable foo.

```
        adq       15*8,dl
```

will add 120 to the Q register. The dl address modifier causes the address field to act like a memory operand, padded on the left with zeroes. The du modifier pads on the right with zeroes.

The following strange-looking multiword instructions are the special character string and bit string instructions; this one performs boolean operations on bit strings. Here a simple move is indicated.

```
        csl       (pr,ql),(pr,al),fill(0),bool(move)
        descb     bp|0,8
        descb     convolution,9
```

will move eight bits from the address bp|0+ql to a 9-bit field (padding with a zero bit) at convolution (plus implicit sp reference) + al. The offset modifiers ql and al refer to the bottom of the Q and A.

```
mvt      (pr),(pr)
desc9a   confused_bytes,8
desc9a   confused_bytes,8
arg      confusion_table+3-*,ic
```

will translate the eight 9-bit bytes at confused_bytes (first argument) according to the table at confusion_table (third argument) and deposit the resultant eight 9-bit bytes in confused_bytes (second argument). The lookup is done by treating each character as an index into the table.

A list of most of the instructions used in Lucifer and their meaning follows.

ada, q, xN	add to A, Q, xN
ana, q, xN	and to A, Q, xN
anaq	and to AQ (two words)
arg	zero opcode (used for mvt table and constants)
cmpa, q, xN	compare A, Q, xN
csl	combine bit strings left (three word instruction)
descb	a pseudoop which generates a bit string descriptor for a csl

	instruction.
desc9a	generates a 9-bit character descriptor
eaa, xN	effective address to A (top half), xN
eppN	effective pointer to pointer register N
era, q, aq, xN	exclusive or A, Q, AQ, xN
ersa, ersq	exclusive or A, Q to storage
lda, q, aq	load A, Q, AQ
llr	long (AQ) left rotate
lls	long (AQ) left shift
lrl	long (AQ) right logical shift
lxlN	load xN from lower half
mlr	move character string left to right (three word instruction)
mvt	move with translation (four word instruction)
ora, q, aq	OR A, Q, AQ
orsa, q	OR A, Q to storage
qls	Q left shift
sba, q, xN	subtract A, Q, xN
sta, q, aq	store A, Q, AQ
stxN	store xN
stz	store zero
tmi	transfer on minus
tnz	transfer on not zero
tpl	transfer on plus (including zero)

tra	unconditional transfer
ttf	transfer tally-runout flag off

Address modifiers appear after a comma in an address field. For example

```
        ldq       bp|0,x2
```

causes indexing by x2.

xN	index by index register N
*	indirect
*xN or *N	indirect then index (i.e., add index register to address in indirect word).
xN* or N*	index then indirect

As well as xN index modification, the following can be used whenever xN appears above:

au	top of A
al	bottom of A
qu	top of Q
ql	bottom of Q
ic	instruction counter
du	direct to upper
dl	direct to lower

The indirect and tally modifiers add-delta (AD) and subtract-delta (SD) take an indirect word. Add-delta causes, after the instruction is executed on the operand pointed to by the address field (bits 0 - 17; the operand lies in the same segment as the AD word), the delta (rightmost six bits) to be added to the address field. The tally (bits 18 to 29) is decremented by one. If the tally reaches zero the tally-runout indicator is set, but no fault occurs. Subtract-delta, before executing the instruction, subtracts the delta from the address field and increments the tally by one.

BIBLIOGRAPHY

1. Girdansky, M. B. "Cryptology, The Computer, and Data Privacy," Computers and Automation, April, 1972, pp. 12-19.

2. Smith, J. L., "The Design of Lucifer, a Cryptographic Device for Data Communications," IBM Research Report RC 3326, April 15, 1971.

3. Honeywell Information Systems, Inc. Honeywell 645 Processor Manual.

Related material:

4. Smith, J. L., Notz, W. A., and Osseck, P. R., "An Experimental Application of Cryptography to a Remotely Accessed Data System," IBM Research Report RC 3508, August 18, 1971. (Also Proc ACM 25th Nat Conf., August, 1972, pp. 282-297.)

5. Feistel, H., "Cryptographic Coding for Databank Privacy," IBM Research Report RC 2827, March 18, 1970.

6. Feistel, H., Notz, W. A., and Smith, J. L., "Cryptographic Techniques for Machine to Machine Data Communications," IBM Research Report RC 3663, December 27, 1971.

MIT/LCS/TM-50

AN ENCIPHERING MODULE FOR MULTICS

G. Gordon Benedict

July 1974

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

PUBLICATIONS TR/TM FORM

Title of Thesis or Report:

An Enciphering Module for Multics

Author(s):

G. Gordon Benedict

No. Assigned:

MAC TM-50

Technical Report:

Technical Memoranda: ✓

If Thesis, type:

S.B. Thesis (June 1974)

Department:

EE Dept. Systems Research - Division II

25 copies
to
Prof. Saltzer

Signature of MAC Group Leader

7/10/74

Date

PUBLICATIONS DISTRIBUTION
PROJECT MAC, ROOM 417A
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
545 TECHNOLOGY SQUARE
CAMBRIDGE, MA 02139

253-5894

July 1974

We have recently issued Project MAC Technical Memoradum 50:

An Enciphering Module for Multics

Benedict, G. Gordon (This Technical Memorandum reproduces a June 1974, M.I.T. Electrical Engineering Department S.B. Thesis of the same title)

AD 782-658

ABSTRACT

Recently IBM Corporation has declassified an algorithm for encryption usable for computer-to-computer or computer-to-terminal communications. Their algorithm was implemented in a hardware device called Lucifer. A software implementation of Lucifer for Multics is described. A proof of the algorithm's reversibility for deciphering is provided. A special hand-coded (assembly language) version of Lucifer is described whose goal is to attain performance as close as possible to that of the hardware device. Performance measurements of this program are given. Questions addressed are: How complex is it to impelment an algorithm in software designed primarily for digital hardware? Can such a program perform well enough for use in the I/O system of a large time-sharing system?