BIDIRECTIONAL HEURISTIC SEARCH AND SPECTRAL S-BOX SIMPLIFICATION FOR THE CRYPTANALYSIS OF THE NBS DATA ENCRYPTION STANDARD

by

Eric Alexander Gullichsen
B.Sc.(Hons.), University of Manitoba, 1981

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

DEPARTMENT OF COMPUTER SCIENCE

We accept this thesis as conforming

to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

February 1983

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
1956 Main Mall
Vancouver, Canada
V6T 1Y3

Date March 3, 1983.

ABSTRACT

Details of the National Bureau of Standards Data Encryption Standard (DES) are examined, and the strength of the cryptosystem found to lie in its substitution box (S-box) components. An unsuccessful attempt is made to discover symmetries in the S-box functions under permutation and/or complementation of variables.

The problem of cryptanalyzing DES is then shown to be equivalent to a problem of tree search. Techniques which can reduce the number of tree nodes which need be visited to effect a cryptanalysis are investigated. The linearization of the S-box functions by coefficient translations in the Hadamard spectral domain is found to be highly effective in reducing search tree size. For a bidirectional tree search which employs the linearized S-boxes, the number of nodes which need be visited to cryptanalyze DES is shown to be on the order of the key space size. The use of an AND/OR search tree structure with key bit constraints stored at the leaves ensures that each node need be visited only once.

Given that the work involved in visiting a node is less than that required for a key trial, this key search method represents an improvement over the cryptanalytic technique of exhaustive key search.

Thesis Supervisor .....

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

The author wishes to thank the following people, without whom the generation of this thesis would have been impossible: Dr. R.G. Stanton and NSERC for remuneration in various forms; Dr. Paul Gilmore, for accepting my thesis quickly, and my advisor Dr. Cyril Leung, for not accepting my thesis quickly; Dr. D.M. Miller, my surrogate thesis advisor at the University of Manitoba, for many invaluable and esoteric ideas about the manipulation and minimization of Boolean functions; Dr. Hugh Williams for wry psychological encouragement and advice. Finally, I wish to acknowledge Micaela, who helped in her own ways.

Chapter I

INTRODUCTION

In January of 1977, the National Bureau of Standards (NBS) of the United States of America proposed a data encryption standard (DES) which they recommended be adopted for the purposes of cryptographic protection of commercial and non-military governmental data [26]. The standard is designed to be implemented in hardware, and may be employed for the purposes of both privacy and the authentication of messages [3,4].

An integral part of the specification of any cryptosystem is some indication of the nature of the security threat which the system is designed to successfully resist. The types of attack to which any system may be exposed are usually divided into three categories [3,4]. The least potent of these is the "ciphertext only" attack, in which the cryptanalyst has in his possession only encrypted data, with no direct knowledge concerning the plaintext. Cryptosystems unable to resist such an attack are very feeble, and not in modern use. The "plaintext" attack is more difficult for a cryptosystem to resist. Here, the cryptanalyst has knowledge of the blocks of plaintext which correspond to the blocks of encrypted text. In this case, only the encryption key, K,

remains to be discovered. Finally, the most powerful of all attacks to which any given cryptosystem may be exposed is the "chosen plaintext attack", in which the cryptanalyst has possession of corresponding blocks of plaintext (P) and ciphertext (C), as in the case of the plaintext attack, and furthermore that the P are selected by the cryptanalyst.

Although situations of known or chosen plaintext attack on a cryptosystem may tend to arise fairly infrequently in the real world, DES was designed to resist even such an attack. In fact, NBS claims that

> "... no technique other than trying all possible keys using known input and output for DES will guarantee finding the chosen key." [26]

NBS continues its discussion of the security of DES and indicates that there exist a very large number of possible keys of 56 bits (about 7×10^{16}) as used in DES, in order to assert that the security of the system is adequate and will continue to be so, given the current state of computer technology, with the standard to be reviewed in five years.

In the author's view such superficial reasoning is potentially dangerous, and appears tantamount to asserting that a simple substitution cipher as applied to a natural language is quite secure, since there are 26! possible keys which must be tried, to guarantee breaking the system. To put forth such a naive claim indicates that one is either overlooking or purposefully ignoring a wide range of factors which may indeed be of assistance in breaking the cipher

system, such as an underlying statistical structure to the language being encrypted [21], or a feasible means of algorithmically or heuristically "inverting" the encryption algorithm in order to solve for the key K from the given P-C pairs. The former may be of interest with respect to the cryptanalysis of text of known structure encrypted with the DES algorithm. However, this thesis will deal principally with particular heuristic techniques for the purposes of an "inversion" of the DES encryption algorithm.

It is realized that at present there exist no good theoretical tools for proving the impossibility of breaking a given practical cryptosystem, and that the demonstration of the security of any such system is usually provided by the inability of expert cryptanalysts to perform a successful cryptanalysis. Indeed, with the exception of encryptions based on the Vernam system or its variants, no cryptosystems are theoretically secure, but are simply difficult to break given the best known algorithms for performing various tasks [21].

In the majority of instances, the most explicit quantification of security of a cryptosystem which can be provided is to indicate that breaking the system will be "at least as hard as" some task assumed to be of substantial time complexity as a function of instance length. For instance, the difficulty of breaking the well-known public key cryptosystem of Rivest et. al. [20] is assumed to be at least as hard

as factoring a very large number chosen for use in the system.

With respect to the above discussion, NBS may now be criticized on at least two accounts. Firstly, NBS has refused to provide any theoretical justification for the supposed security of DES, and has only indicated that about 17 man-years of effort were expended in the certification of the standard and that the system is thus secure. Neither NBS, nor the National Security Agency (NSA) which participated in this certification process, have released any details of the study which apparently indicated the strength of the system. Similarly, no explanation for the structure of the encryption algorithm has been offered. Such a noticeable omission of information has led some authors [2,5] to speculate that DES has concealed within it some "trap-door" information which would allow those in possession of such details (i.e., the NSA) to break the system with relative ease.

Secondly, various attempts at cryptanalysis of DES have indicated that the NBS claim that all possible keys must be tried to ensure breaking the system is exaggerated. Hellman et. al. [5] discovered a symmetry under complementation of P, C, and K which results in a 50% time saving in cryptanalysis over exhaustive key search. Elsewhere [6], Hellman also describes how after initial exhaustive cryptanalysis, DES-like systems may be broken for subsequent P-C pairs in

time on the order of the square root of the key space size. Other authors [2] have suggested that it is technologically feasible to construct a special-purpose machine with a million distinct processor elements which would be capable of solving for K from any P-C pair in less than 24 hours.

Hopefully, the above serves to indicate that the DES system may not be as secure for many applications as either NBS or the NSA would prefer that people believe.

Chapter II

S-BOX COMPLEXITY: STRENGTH OF DES

It will be useful for both purposes of familiarization of the reader with some details of DES, and to indicate in which areas of the algorithm the strengths of the system lie, to examine the encryption algorithm with some precision. Full details of the algorithm are publically available in the appropriate NBS documents [26].

In its most common mode of operation, DES serves as an "electronic code book", encrypting 64-bit blocks P to form 64-bit blocks C, using 56 bits of a 64-bit key K. For a given K, DES may be thought of as a one-to-one mapping of a 64-dimensional vector space over GF(2) into itself. To aid in assuring the security of the cryptosystem this mapping should be highly non-linear. Examination of the internal structure of the DES algorithm indicates that it follows Shannon's advice of alternating layers of permutation and substitution, in order to respectively provide diffusion and confusion [4,21].

Following an initial permutation, the P block is subjected to 16 rounds of an encryption process, where each round consists basically of a substitution of bits followed by a simple permutation, and is preceeded by an XORing of bits of

K selected in a permuted fashion as a function of layer with a permutation of the current bits in the developing ciphertext. At each layer of encryption, this operation is performed on only the rightmost 32 bits of the developing ciphertext, but the right and left halves of this block are transposed at each level. After these 16 layers of encryption, the resulting block is subjected to another application of the initial permutation, inverted, to yield C. In the case of a known plaintext attack, as is our assumption, the applications of the publically-known initial permutation add no difficulty to the cryptanalytic task.

It may clearly be seen at this point that the strength of DES rests in the process of substitution of bits at each round, as performed by the S-boxes. All other operations in the encryption procedure: the XORing, the permutation, and the expansion, are linear in binary arithmetic. Were the substitutions performed by the S-boxes also linear, the entire encryption procedure would be linear, and

$$C=AP + BK$$

for C, P and K considered as binary vectors. In such a case, chosen plaintext cryptanalysis would be equivalent to performing the inversion of a 56x56 binary matrix, as from the above equation:

$$BK = C - AP$$
$$K = B^{-1} (C - AP)$$

It is thus trivial to calculate K when P and C are known; the A and B matrices come from the encryption algorithm. It has been shown [5] that the S-boxes as employed in DES are neither linear nor affine (a case which would yield almost as simple a cryptanalytic procedure), although there exists speculation as to whether or not the S-boxes conceal less overt trap-door information such as parity.

That it is this non-linearity of the S-boxes which leads to extensive difficulties in a "search-tree" exhaustive approach to cryptanalysis may easily be perceived when the search for K from a known P-C pair is represented graphically. (Figure 1, Exhaustive Tree Search) The search for K may be represented as an AND/OR tree traversed in a top-down fashion. At level 1 in the tree, the values for C in all 64 bit positions are known. Level 0 is represented by an AND node, as all of its successors must be true, as a result of precise knowledge of all bits of C. At the leaves of the tree, the values of P in all bit positions are similarly known. This search procedure is presented more formally in Chapter VII.

When one considers the first[1] branch to this tree, it may be seen that there are two ways to make the 64th bit of the C block 1.[2] One such possibility is that the positionally corresponding bit of L15 and that of f(R15,K16) are both 0.

---

[1] employing a standard preorder tree traversal

[2] or 0, without loss of generality

As a consequence of the structure of the S-boxes, there are 32 ways in which this condition may be satisfied, if the S-boxes are utilized in the manner in which they are presented in the DES literature [26]. Of the s↑6=64 possible input configurations for any S-box, precisely half of these (32) result in the value of some specified S-box output being 1; the other 32 cause the S-box output to have a value of 0. Thus, when the process of encryption is inverted, it may be seen that any of 32 possible inputs can have caused some specific S-box output to have a certain value.

Due to this very high tree branching factor, an exhaustive tree search for K would be no more efficacious than exhaustive cryptanalysis by trying all possible K's and determining their correctness by use of DES in the forward direction. In fact, there would be far more nodes in such a search tree than there are possible keys of 56 bits.

The fact that the 32 conditions which lead to the same f output are disjunctive makes it similarly impossible to discover K by heuristically pruning mutually incompatible but necessarily conjunctive conditions from the same level of the tree. All of the input conditions leading to the S-boxes production of a 1 output in the given bit position would have to be inconsistent with that of another AND path before the subtree growing at such a point could be disregarded. The search for such conditions throughout the entire tree to heuristically guide search for K would be more costly than brute-force exhaustive key trials.

Chapter III

AN INVESTIGATION OF S-BOX GROUP PROPERTIES

3.1 THEORY: PERMUTATION AND COMPLEMENTATION OPERATORS

In an attempt to discover potential regularities within the structure of the DES S-boxes, a method devised by McCluskey [12] was employed to ascertain whether or not any of the Boolean functions represented by the actual S-boxes employed in the DES system possess any properties of group invariance. As is described below, it is convenient to interpret each of of the 8 S-boxes as a set of 4 Boolean functions of 6 variables. Each of the 8x4=32 outputs of the bank of S-boxes is a different function of 6 variables. It is our concern here to discover which, if any, of these functions are invariant under the permutation and/or complementation of input variables. The set of all permutation and complementation operators forms a mathematical group, hence the term "group invariance" of a function.

As they are represented in the DES algorithm in tabular form, the S-boxes are exceedingly difficult to work with: conventional Boolean algebra provides no formalisms for dealing with such structures. Consequently, each S-box was interpreted as 4 Boolean functions, each of 6 independent variables, one such function for each of the S-box outputs.

With little difficulty, it is possible to obtain a Boolean function whose value is equivalent to that of a specified output bit of any desired S-box. This function is in the form of a logical sum of elementary product terms (p-terms) where each such p-term is a logical product of the values of the 6 input variables to the chosen S-box.

By examining a binary representation of the contents of an S-box, it is possible to discover for which 32 of the 2^6 =64 possible input configurations to the S-box a chosen output bit will be on. The row and column index of the S-box entry with a 1-bit in the chosen output position are used to determine which input configuration is responsible for the selection of this entry causing the 1 output. Each of the 32 entries for which the desired output bit will be on will correspond to a p-term in the Boolean function form for that S-box - output bit pair: a 0 in a position of the binary representation of a S-box entry corresponds to a complemented literal in the p-term in the corresponding position, whereas a 1 corresponds to an uncomplemented literal. For instance, if S-box 1 output 1 is on for the configurations of input variables: 000000, 000001, ... , 111110 then the p-term Boolean function for that S-box and output may be written as:

X1'X2'X3'X4'X5'X6' + X1'X2'X3'X4'X5'X6 + ... + X1X2X3X4X5X6'

Such functions may be obtained for each of the 4 outputs of each of the 8 S-boxes, and may be represented as 32x6 bi-

nary matrices. Each row in such a representation will correspond to a single conjunctive p-term. Following the terminology of McCluskey [12], these matrices will be referred to as transmission matrices, T.

What is of interest with respect to S-box structure is the group invariance (or lack thereof) of these Boolean functions. Discovery of the group properties with which we concern ourselves at this point is equivalent to the determination of whether or not there exist any permutation and/or complementation operations which leave the functions unchanged when these operations are applied to the input variables. If any such group properties are discovered within the S-boxes, the symmetries they represent may be used to reduce the size of the search space involved in the search for the encryption key when cryptanalyzing instances of the application of DES. The discovery of any symmetries in the S-box functions will make it possible to represent these functions in a more compact form, and hence reduce the branching factor in the search for K. Clearly, it is of great interest to be able to discover any possible means of reducing the size of the inevitably large search tree formed to uncover the key used to encrypt known plain and ciphertext blocks.

As a simple example of how a discovery of functional invariance under the permutation of input variables allows a more succinct representation of a function, consider the function:

$$f(X1,X2) = X1X2 + X1'X2 + X1X2'$$

If it is known that f is symmetric in X1 and X2, then

$$f(X2,X1) = X2X1 + X2'X1 + X2X1' = f(X1,X2)$$

and it may be concluded that $f(X1,X2) = X1 + X2$.

We shall use McCluskey's notation of SiT to represent some permutation of the Boolean transmission function T where the i subscript represents the specific permutation. NjT shall be used to represent a complementation of the input variables of T which correspond to a 1 in the binary representation of the subscript j. For example,
Si(X1 X2 X3 X4 X5 X6) = (X2 X1 X3 X4 X6 X5) for i=213465
and
Nj(X1 X2 X3 X4 X5 X6)=(X1 X2 X3' X4 X5' X6') for j=001011
We are interested in determining, for each T, the values of i and j such that SiNjT=T.

Even for our application, which involves a relatively small number, n=6, of independent variables in the T functions, an exhaustive search for group invariants may easily be shown to be intractable. There are n! possible Si operators, and $2\uparrow n$ possible Nj operators, hence $n!2^n$ possible SiNj operators. When the T functions involve 6 independent variables, this means there exist $6!2^6 = 46080$ possible SiNj operators. A brute-force determination of the invariance of T under these operators would involve operating on T with each of the operators, and then determining if there is some

row permutation of the binary matrix which represents SiNjT which would make SiNjT identical to T's matrix representation. (If an SiNj operator leaves T unchanged, then the only possible effect of applying the operator to T is to change the order of the rows of T, analogous to changing the order of the disjunction of conjunct terms in the elementary p-term expression). As our T functions involve 32 rows each, up to $32!=2.6\times10^{35}$ permutations of SiNjT might have to be tested for each of the 46080 SiNj operators. To circumvent such blatant computational intractability, some consideration of the characteristics of the specific T functions is required.

For an Si operator to have no effect, the columns of T exchanged by the Si must have equal numbers of 1's, as permuting the rows cannot vary the total number of 1's in any column. For an Nj operator to have no effect, either the single primed column of T must have an equal number of 1's and 0's, or else there must exist two primed columns, where the first has as many 1's as the second has 0's. Following McCluskey, if one transforms T into a standard matrix D, the SiNj operators leaving T invariant may be determined directly from the Si operator which leave D invariant.[3] (SiD=D). D is formed from T by priming all columns with more 1's than 0's.

---

[3] Actually, which leave invariant either D or any D' formed by priming suitable combinations of columns of T with equal numbers of 1's and 0's. This consideration of Ni operators will be deferred until later.

As D has columns with at least as many 0's as 1's, one need only consider permutations of columns with equal numbers of 0's. For this reason, D is partitioned into column partitions where each column present in any given partition has the same number of 0's. Thus, one need only consider Si operators which switch columns within the same column partitions. Rows may also be partitioned in an identical fashion: Only rows from within the same row partition may be permuted to identify SiD with D.

It can be reasoned that this process of partitioning should be further carried out on the submatrices of D formed by the initial partitioning. As McCluskey indicates:

> "In general, only rows which have the same weight in each submatrix can be interchanged. Priming columns of the same partition does not change the weight of the rows in the corresponding submatrices" [12: p.1448]

The partitioning process is carried out recursively on the submatrices formed by prior partitionings of D until a matrix results in which each row and column of each submatrix has the same number of 0's. Assuming that the partitions are relatively small, even an exhaustive approach to the determination of which row and column permutations leave D unchanged should be tractable, as only permutations involving rows or columns from within the same partition need be considered. That is, for each possible permutation of each of the column partitions in the fully-partitioned D, permutations of row partitions are applied to restore D to its original form. If the column partitions of D are very

small, the number of possible column permutations is drastically limited. In the trivial case where each column is in a partition by itself, where submatrices have different column weights for all columns of D, it can be concluded that no Si exists such that SiD=D.

After D has been fully partitioned and the Si permutations which leave D invariant have been discovered, we must consider the D'. Recall that as defined, our D may possess some columns with an equal number of 0's and 1's. In this eventuality, we must form a set of possible special standard matrices D', by priming certain combinations of the columns of D which have this equal number of 0's and 1's. These primings will determine the j superscripts of possible Nj operators. if we form D'=NjD, and SiD'=D' as determined by the partitioning and column and row permutations of D', we can deduce the SiNjT=T represented by this invariance.

Not all possible combinations of primings of these columns of D need be considered; special characteristics of D will permit the <u>a priori</u> elimination of some D'. If some row of D is all 0 (or 1) and after priming, D'=NjD does not also have a row of all 0 (or 1) we know there cannot exist an Si such that SiNjD=D. No amount of column switching can allow us to form a row of all 0 (or 1) if such a row does not already exist in D'.

After the elimination of some potential D' in this manner, we form the D' and partition them recursively as was

done for the D matrix, to form submatrices of the D' with the property that each row (column) of each submatrix has the same number of 0's. If any of these D' matrices has the same partitioning as D, permutations of columns within column partitions are examined to determine if any such column permutation, followed by a row permutation of rows within the same partition(s) can restore the SiD' to D'. If such permutations exist, we have determined SiNj operators such that SiNjD=D. (Nj is the priming of D to form D'). From the initial primings used to transform the transmission matrix T into the standard matrix D, the SiNjT=T invariances may be directly determined.

3.2 IMPLEMENTATION FOR DES

The language first chosen for the implementation of this algorithm, and some subsequent experimentation with DES was APL, due to its pseudo-parallel array processing capabilities, its power with respect to both Boolean and matrix manipulations, and its interactive nature. The code for all APL functions referred to in this chapter may be found in Appendix A. Unfortunately, it was discovered that the computational cost overhead incurred by the fact that APL is an interpreted language limits its applicability to problems of a relatively small size. For later programs involving computations of a combinatorially large nature, the compiled language PL/I was employed.

The partitioning procedure described in the preceeding section was implemented as a recursive APL routine, PARTITION. The routine is passed a standard matrix, as defined earlier, with rows and column both permuted in order of increasing number of 1-bits. The initial partitioning of rows and columns is determined by examining at which points the next row (column) has more 1's than the preceeding row (column). Such positions indicate partition points in the standard matrix.

This initial partitioning is discovered by means of call to the routine INITPARTIT. Given a binary representation of a standard matrix, the routine INITPARTIT returns a 2xN integer matrix of pointers into the standard matrix. Each pointer indicates a partition point of the standard matrix: a point before which the matrix should be divided to form a submatrix. The first row of the matrix of pointers returned from the INITPARTIT routine refers to divisions between rows of the standard matrix, while the second row refers to column divisions.

After having called INITPARTIT to determine the initial partitioning of the standard matrix as determined by where the number of 1's in rows and columns changes, PARTITION calls the recursive routine PARTITCALL with both the initial partitioning and the standard matrix as arguments.

It is this PARTITCALL routine which may be considered the central routine in the partitioning system. The routine is

passed both the standard matrix, and the partition points which divide that matrix into the first level of submatrices. Employing two nested loops, PARTITCALL iterates through all of the initial submatrices of the standard matrix by columns. For each of these submatrices, INITPARTIT is called to obtain the initial partitioning of that submatrix, and PARTITCALL is recursively invoked to further partition the submatrix. PARTITCALL returns a matrix of pointers containing all partition points discovered for either the initial matrix with which it was invoked, or any recursively-discovered submatrices of that matrix.

During the debugging of this system of routines, it was discovered that the above recursion was insufficient as implemented to discover the complete partitioning of a binary matrix. (Where as defined earlier, a completely partitioned matrix is one in which all rows (columns) within any partition have the same number of 1's). The reason for this was that partitions made in one submatrix at a specific level of recursion are not known to other submatrices at the same level of recursion during their partitioning.

Consider, for instance, the following case which acutally occured during the partitioning of the standard matrix representing the Boolean function for S-box 1, output 1. Suppose that 2 has already been discovered as a column partition point for the top level matrix as a result of the partitioning of some submatrix occuring higher in the same

column as the submatrix currently being processed. That is, a partition should exist between columns 1 and 2 of the standard matrix. Suppose also that the submatrix now occuring lower in the column is:

011110
101011

Application of the recursive partitioning routine to this submatrix would result in the following partitioning:

01	1	1	1	0
10	0	1	1	1

Each of the submatrices formed as a result of this partitioning indeed satisfies the property of each row (column) having an equal number of 1's. However, as a division exists in the top level matrix of which this is a submatrix as a result of an earlier partitioning of another submatrix, the top level matrix may not be fully partitioned, even after all submatrices have been partitioned in this manner. The division existing between the first and second columns implies that within the illustrated submatrix, two 2x1 submatrices exist which do not have an equal number of 1's in their rows.

As a consequence of this ignorance of each submatrix concerning the partitioning of the other submatrices at its same level, to fully partition the standard matrix it does not suffice to simply call PARTITCALL once. Consequently, the PARTITION routine calls PARTITCALL iteratively. On the first call to PARTITCALL, the partitioning of the standard

matrix supplied is that returned by INITPARTIT. Subsequently, PARTITCALL is called with the initial partitioning set to be the complete partitioning as returned from the previous call to PARTITCALL. In this manner, the partitionings of each submatrix are made known to other submatrices at the same level. With reference to our example, the fact that a partition exists between the first and second columns is known globally when PARTITCALL is iteratively reinvoked from PARTITION, so the two 2x1 submatrices with unequal numbers of 1's in their rows would be further partitioned during this call. The process of partitioning terminates when no further partitionings are discovered as a result of repeated calls to PARTITCALL.

Another routine, PRINT-PARTIT, was devised to display the partitioning of a matrix. When called with a matrix and its partition points as arguments, the matrix is printed with spaces between its submatrix components.

An illustration of the operation of these partitioning routines may be seen in Figure 2, Complete Partitioning of a Matrix. A 9x6 binary matrix is partitioned, and the resulting partitioning displayed by call to the PRINT-PARTIT routine. This matrix is the same as that used by McCluskey [12: p.1447]. From this example, it may clearly be seen that each row (column) of the fully partitioned matrix has an equal number of 1's.

3.3 APPLICATION TO DES

As the discussion of the algorithm for detecting group symmetries in Boolean functions indicated, after a standard matrix has been formed, only columns from within the same column partition may be permuted, if the matrix is to be restored by means of row permutations. Thus, the first step towards the discovery of possible symmetries in the transmission functions which represent the DES S-boxes is to fully partition the standard matrices for such transmission functions.

For this purpose a driver routine, PARTITION-ALL, was implemented, to call PARTITION with standard matrices representative of the transmission functions for each of the 32 possible S-box - output pairs. This driver routine forms all of these transmission matrices T from S-box data, and puts T into standard form by priming all columns which contain more 1's than 0's. The rows and columns of each standard matrix are then permuted in order of increasing 1's.

The purpose of this routine was to obtain some approximate idea of how small the column partitions of the standard matrices would be, to determine the tractability of an exhaustive approach to the permutations of columns within the same column partitions. As usual, the numbers indicative of the partition points are positions before which the matrix should be divided.

As may be seen in the table of output from this routine, somewhat surprising results were obtained. All 32 matrices partition so that there is only one row and one column in each submatrix; submatrices are all 1x1 in size. Such a structure implies that for these functions, no SiNj exist such that SiNjT=T. There is no possible way to complement and/or permute the inputs to any S-box and leave the S-box functionally invariant. This approach to the discovery of S-box symmetry is consequently of no use in reducing the search space size during cryptanalysis.

Subsequent discussion of the problem of symmetry detection with Dr. D.M. Miller led to the idea of the use of Rademacher-Walsh spectral techniques for the detection of any partial two or multi-variable symmetries which may be present in the Boolean functions for the S-boxes [14,17]. Although it is not possible to permute and/or complement any S-box inputs and leave the function of any S-box invariant, it may be possible that for some S-boxes such symmetries as:

$$f(0,1,\ldots) = f(1,0,\ldots)$$

do exist. The existence of even such partial symmetries in the S-boxes could allow us to reduce the size of the seach tree for the encryption key. The application of such techniques to the S-box functions has not as yet been pursued, and remains as an interesting problem for future research.

Chapter IV

CRYPTANALYSIS BY S-BOX APPROXIMATION

Given that the core of the problem of cryptanalysis of DES rests in the complexity of the S-boxes, it was decided that one potentially successful means of attack of DES could conceivably be through approximation of the S-boxes. P-term expressions for the Boolean functions embodied in the S-boxes have already been obtained as a result of the analysis of the preceeding chapter. Other advantages may result from obtaining some sort of "minimal" sum-of-products expression for each S-box output, as a function of the 6 input variables or their complements. Such a representation may be amenable to making some as-yet unnoticed S-box structure more apparent. A more compact expression for the action of the S-boxes should reduce the effective branching factor of the search tree, and should also assist in increasing the tractability of the operation of pruning this search tree. For these reasons, we shall wish to obtain sum-of-product expressions for each output of every S-box which contain the minimal number of Boolean literals required to express the function which represents that S-box - output pair.

More specifically, having the S-boxes in such a form may permit the Boolean functions performed by the S-boxes to be

approximated in such a manner as to allow a valuable trade-off between the accuracy of the approximated S-boxes and the cost of the solution for K from P and C in such an approximate system. Suppose for instance that some approximation to the S-boxes as could be achieved by considering the sum of only the 3 most significant conjunctive terms in the sum-of-products expression for each S-box - output pair yields a system which may be simply inverted, and it is possible to tractably solve for K, from P and C.[4]

Unfortunately, if the functions which represent the S-boxes are only approximate, and are thus not correct for all possible input configurations, the K obtained from search with a P-C pair may also be incorrect. If the K we obtain as a solution has a probability of being correct of only 1/n, then on average we must solve for K using n/2 P-C pairs before the resulting K:P->C under the "real" DES algorithm. However, if the time required to discover a K in the approximate DES system is a factor of more than n/2 less than that required when accurate S-box functions are employed (weighted by the cost of the n/2 encryption trials to ascertain if the K is correct), then this cryptanalytic technique should be of some merit.

---

[4] The solution need not be analytical, but may well involve heuristic search of a tree simplified in the sense of it having a reduced branching factor resulting from the use of the simplified S-box expressions.

In summary, we wish to approximate the Boolean functions which represent the S-boxes, and search for K from P-C pairs, an unlimited amount of which are avaialble to us. As the S-box functions are only approximate, such K may be incorrect. Any potentially-correct K discovered may be quickly verified by seeing if it does map P to C. We continue to produce a potential K by using the search procedure with with different P-C pairs until a correct K is produced.

Generally, it would be of interest to examine the characteristics of the tradeoff between the degree of S-box approximation, and the time required for cryptanalysis. As one continues to simplify the S-box approximating expression in some regular fashion, the encryption employing these expressions will continue to lose accuracy, although an analytical or search-tree solution for K should become more simple. Exactly how optimal a simplification may be achieved, from the perspective of cryptanalytic cost, is another topic for future research. This thesis will be more concerned with the reduction of the size and complexity of the functions represented by the S-boxes without any compromise in their accuracy.

Chapter V

QUINE-MCCLUSKEY MINIMIZATION OF S-BOXES

5.1 QUINE-MCCLUSKEY: IMPLEMENTATION

Several classical methods exist in the field of digital logic design for the minimization of Boolean functions. These include the Karnaugh map method and the Quine-McCluskey (QM) procedure [13], this latter procedure being more suited to computerized implementation. Both are designed to minimize the Boolean function in question as a sum-of-products expression. The Reed-McClennan technique is occasionally employed as an alternate procedure, to minimally express any given Boolean function by means of XOR operations. While such a minimization algorithm may be of use as the S-boxes could be as heavily XOR-oriented as the remainder of the DES algorithm, the reliance of the Reed-McClennan algorithm on highly topological methods makes it clumsy to implement. Chapter VII will be concerned with alternatives to the Reed-McClennan technique for the extraction of XORs.

The QM algorithm may be seen to have two distinct phases: that of discovering all prime implicants, and that of forming non-redundant sums. A prime implicant of an n-variable Boolean function f is a product term B consisting of m (which is not greater than n) literals, such that B->f, but

that any B' formed by deleting a literal from B no longer implies f. A literal is a Boolean variable or the complement of a Boolean variable. It is not of substantial interest to discuss the details of the QM procedure here, as the algorithm referred to may be found in McCluskey's paper [13], or in any standard textbook concerned with digital logic design [16].

An implementation of the QM algorithm was devised and applied to the S-boxes of the DES system. The code for the routines referred to in this section may be found in Appendix B. As previously mentioned, it is possible to view the bank of S-boxes in the DES encryption algorithm as 32 separate Boolean functions, each of 6 independent variables. Both these functions and their complements were minimized by the QM procedure. The requirement for minimal forms for the complements of the S-box functions, i.e. minimal forms to describe the input conditions for which an output of the S-bank is 0, is elaborated upon in Chapter VIII. The requirement is connected to the fundamental asymmetry in the number of ways a sum-of-products form and the corresponding product-of-sum form generated by DeMorgan complementation may be instantiated to produce a functional output of 1.

The routine ANALYZE calls QM iteratively for each of the 32 possible S-box - output pair combinations. QM calls the function PRIMIMP in order to determine the prime implicant terms for the various inputs. Complemented and uncomple-

mented forms of the S-boxes themselves are represented as a global 3-dimensional matrix consisting of a lamination of the tables as supplied in the DES literature [26]. The function BINARY converts the decimal representation of the S-boxes to binary, and the ON function, also called from ANALYZE, returns the 32x6 matrix of p-term inputs for which the specified output of the specified S-box is on, i.e. 1.

By means of the ANALYZE procedure, global tables of essential and alternative products of input literals for each of the 32 S-box - output pairs were constructed. At this point, exact expressions for the S-boxes had still not been discovered. The essential terms are those which must be included; the table of alternatives indicates what options are available in selecting the remainder. For the purposes of visual inspection to detect some overt S-box structure, these tables were printed by the DUMP-SP routine. One such table may be seen in Figure 3, Essential and Alternative Sum-of-Products Terms.

Each row of the table of terms corresponds to a single p-term, where the p-terms are in a notation known as "cube" notation[5] as developed by Roth [10]. In this notation, the presence of a 1 in some position of a p-term indicates that the corresponding literal is to remain uncomplemented; a 0 indicates complementation is to occur. The presence of an X

---

[5] Named for the topological interpretation in which each variable of an n-variable function corresponds to a vertex of an n-dimensional cube.

indicates that the value is a "don't care", and the input variable may be ignored. The rather sparse occurence of don't care inputs is remarkable. In the alternatives table, the integer to the right of the term indicates the class to which that product term belongs, and only one term from each class need be chosen from the set of all choices, when combined with the essential terms, to form a fully accurate representation of the S-box.

In order to verify that the QM minimizations performed are accurate, the PROB-CORR function was used to check that the minimal Boolean expression for each S-box returns the same value for every possible input configuration as does the real tabular S-box. This function was also later used to determine the probability of correctness of some approximate forms of the S-boxes.

5.2 SELECTION OF ALTERNATIVE TERMS BY REDUCE

After the Quine-McCluskey minimization procedure had been applied to the Boolean functions represented by the S-box tables, it was necessary to select one conjunct term from each of the classes of implicationally equivalent terms, in order to be able to represent the output of each of the S-boxes in a closed form as a sum-of-products expression.

After some reflection, it became apparent that there existed techniques for the selection of such a class representative which were in some ways superior to simply picking

the positionally first member of each class as the conjunctive term representative of that class, a simpleminded strategy first followed by SELECT-SP in forming the sum-of-products expression used by PROB-CORR to verify the correctness of the minimization as mentioned in the preceeding section. Referring to Figure 3 again, one may notice that there exist terms which appear in more than one class. Certainly, the selection of such a term as a class representative would eliminate the need for selecting any member of the other class(es) in which it appears, hence reducing the number of terms in the sum-of-products expression without losing any accuracy. Such an ability is clearly advantageous.

The APL routine REDUCE employs this strategy in a relaxation-like fashion, to potentially reduce the number of classes prior to the selection of terms as class representatives. A comparison of each alternative term with all other alternative terms is performed to detect the existence of identical terms in different classes. Beginning with the term which most often appears inter-class, as the same term could appear in more than two distinct classes, and iterative repeating the process on the alternative terms which remain after this reduction, the REDUCE routine serves to constrain potential choices of alternatives.

As the above technique cannot serve to completely constrain the choice of class representative terms except in

the most radical of circumstances, where all classes contain some term also occuring in another class, a case which never occurs in DES, the problem remains of choosing the "best" representative of each class from the remaining terms. The REDUCE routine heuristically and somewhat arbitrarily selects as class representative the term which contains the most don't care (X) values. It was hoped that such a selection criterion could serve to simplify future operations which involve the sum-of-products expressions.

It is clear that the heuristic of selecting class representatives having the most don't care inputs may not be optimal for reasons which pertain to the applications of our minimal functions. As the minimal sum-of-products expressions are to be used in the tree search procedure for K, it will be desirable to have expressions for the S-boxes with a maximal number of terms in common, to allow pruning across subtrees during search tree growth. For this reason, the interactions between the functions which represent the S-boxes may prove to be of significance. Thus, it may eventually be necessary or advantageous to perform a simultaneous minimization of all functions for the entire bank of S-boxes.

The sum-of-product terms resulting from the application of REDUCE to the tables of essential and alternative terms are stored in a global 4-dimensional matrix, SPTERMS, whose space and plane coordinates respectively represent S-box and output bit choices. The sum-of-product terms produced by

REDUCE may be seen in Table 1, where these p-terms are in cube notation. The minimal sum-of-product expressions for the S-box outputs contain between 14 and 23 terms depending on the function. This may be seen as a significant reduction from the 32 terms involved in the elementary p-term expression for each function.

5.3 INDIVIDUAL TERM CONTRIBUTION: RANK-TERMS

After having obtained such a minimal expression for each S-box and output pair, it was desired to ascertain which of the terms of each expression was most important, i.e. which of the terms contributed the most to the probability of correctness of the expression. One may speak of each term as possessing an associated probability of correctness value, Pcorr, which indicates the probability that the single term produces the same output for each possible input configuration as does the S-box in whose approximation the term exists.

Knowledge of this quantity was originally considered necessary in order to rank the terms in importance, to allow the selection of the "best" n terms to approximate the exact sum-of-products form. A scheme of greater sophistication was actually employed for this purpose, as elaborated in the succeeding chapter.

To accomplish this estimation of the importance of individual conjunctive terms, the RANK-TERMS procedure was de-

vised. This program calculates the contribution of each term to the correctness of the expression, and accordingly reorders the terms in SPTERMS for each S-box and output. The routine CONTRIB calculates the "contribution" of a single term, by assuming that the S-box is represented by the single conjunctive term which the routine receives as an argument and calculating the percentage of the 64 possible inputs to that S-box which produce the correct output bit. For most terms, the calculated contribution value is slightly above .5, i.e. the term in question "turns on" for several input configurations.

RANK-TERMS iterates over all S-box - output pairs, and calls CONTRIB to ascertain the contribution of each term. The terms in SPTERMS are ranked in decreasing order of contribution. Table 1 displays the ranked terms, together with their associated Pcorr values.

Chapter VI

SELECTION OF THE BEST SUM-OF-PRODUCT TERMS

6.1 COMBINATORIALLY EXHAUSTIVE BEST-SET DISCOVERY

When one approximates something, it is often useful to have a precise quantitative measure of the qualitative "goodness" of the approximation, where "goodness" must be accurately defined. In our situation which pertains to the formation of sum-of-product expressions to approximate the output of an S-box, the precision of the expressions may easily be quantified. The Pcorr of a sum-of-products expression is defined as the fraction of the number of inputs for which the result of the expression has the same value as the output of the S-box which it approximates. Notice that these Pcorr will always lie in the range {.5,1}. In the trivial case where the approximation consists of 0 terms, the approximate expression's output will always be 0, as will that of the S-box for half of the input configurations. An increase in the number of terms in the approximation can only increase the Pcorr. As the conjunct terms were obtained from a QM minimization of the S-box output values, the approximating expression can never return a 1, when the actual S-box output should be 0.

The principal question to be answered at this point in our discussion is: Given the chance to select n sum-of-product terms to approximate an S-box output, what terms should be chosen to guarantee the best possible approximation? This raises a related consideration, the answer to which is not evident a priori: Denote by Bn a set of n conjunct terms which have a correctness value at least as high as any other possible selection of n terms. If one is only permitted to change the approximating expression by the monotonic addition of terms, will such an expression always be a most correct approximation? That is, will there always be a set Bn+1 such that this Bn+1 contains a Bn, for all n?

To attempt to address the preceeding two questions, a system of PL/I routines was written, as may be seen in Appendix C. Although the author finds PL/I to be a primitive and clumsy computer language to use, it was chosen for its reasonably high speed of execution. Combinatorially large problems tend to be computationally intractable in APL, due to APL's interpreted nature. Two small APL routines, DUMPTERMS and DUMPONS, were written to create PL/I-accessable datasets which contain both the complete sum-of-products representation for each S-box - output pair, and the vectors of input variable values for which the corresponding S-box outputs were on.

The output which resulted from running the routines on the p-terms discovered by Quine-McCluskey minimization of

S-box 1, output 1 may be seen in Appendix C, following the program listing. Page xxx lists this minimal set of p-terms, together with a list of the 32 input configurations for which this particular S-box function returns a 1.

The PL/I system operates in three distinct steps. First, the 23x32 binary "contribution table" is formed, as may be seen on page 188. Each row of this table corresponds to a particular p-term; the corresponding p-terms may be seen to the left of the table. The binary entries in a row of the table indicate if that p-term is "on" for a particular input configuration which causes the function to be 1. There are 32 columns in this table, as for each S-box output there are precisely 32 input configurations which cause that output to be 1. In this sense, the matrix indicates the contribution of each term. Specifically, a particular term is on for a given input configuration if an ORing of the don't care input positions of the term with the XOR of the complemented (0) positions of the term with the input string yields a string of all 1's. The rows of CONTRIB correspond to terms. As mentioned at the end of section 5.2, an exact sum-of-products expression for any given S-box - output pair requires at most 23 conjunctive terms, which explains the dimensioning of the CONTRIB matrix. Within a row of CONTRIB, 1's indicate for which of the 32 inputs the term is on in the sense indicated above.

Next, the COVER table is created, using CONTRIB. COVER is an Mx23 binary matrix, formed to contain an indication of which of the conjunct terms to select to get the best approximation to the S-box, given that one is restricted to selecting precisely k=1,2,...,n of these terms. A portion of this (rather lengthy) table appears from pages 189 to 193. It is in the creation of COVER that most of the computational complexity of the system resides. In order to determine the best selection of k terms, all choices of k terms selected from a set of n possible terms must be examined. In iteratively determining the best selection of k terms as k ranges form 1 to n terms, a total of 2↑n selections must be considered. As n may be as large as 23, this computation is far from trivial.

Due to the associated expense, only the terms for the first output of the first S-box were searched in this manner, prior to developing a more sophisticated technique for discovering best sets. To perform the analysis for output 1 of S-box 1 required 788 seconds of execution time, on the AMDAHL 470/V8 at U.B.C. The improved algorithm of the next section subsequently reduced this execution time by a factor of more than 40.

The cover table is partitioned into n sets of best-sets, where n is the number of p-terms in the S-box approximation (n=17 in the example for S-box 1, output 1). The partitioning of the table is indicated in the output by blank lines.

The cover table analysis table at the top of page 189 indicates the number of best-sets in each partition. For instance, there are 2 ways to select the best set B1, which contains 1 p-terms; only 1 way to select B2; 12 ways to select B3; and so forth. For each of the n sets of these best-sets, a Pcorr value may be seen in the cover table analysis section of the output produced by the PL/I routine. For a particular best-set k, this value denotes the correctness of the approximation if any set from that set of best sets is chosen as the approximation to the S-box. That is, the Pcorr value denotes what fraction of the possible inputs are mapped to the correct output value. It is trivial to notice that as k increases, the associated correctness value strictly increases.[6] For ease of readability, the dump of the COVER table shows the table partitioned into the different sets of best-sets. It is clear that within a row of this table a value of 1 indicates that the term in the corresponding bit position is to be chosen as part of the approximation. (The 23 columns in the table correspond to the 23 p-terms in an S-box function).

The creation of the COVER table served to answer the principal question posed at the beginning of this section. A best approximation to an S-box, where the approximation is restricted to possessing k terms, is found by selecting any element from the kth set of best sets in the COVER table.

---

[6] If it did not, this would imply that our QM minimization procedure was faulty, and had produced redundant terms.

The correctness value for such an approximation is known.

The last purpose of the system of routines is to perform a search of the cover table to see if there exists a way to select one member of each set of best sets, such that the set of terms indicated by each selection is a subset of the terms indicated by the selection from the next set of sets. The search was implemented as a standard top-down search of the COVER table, with backtracking on failure. As may be seen on page 195 of the output, this search was successful for S-box 1, output 1.

This term selection table is a cumulative record of the sum-of-product terms chosen at each level. One new 1 appears in each successive row of the table; its position corresponds to the one new term added to the previous terms to form the new, more correct, approximation. The corresponding correctness values for these successive selections are repeated in this table. An alternative representation of this data is provided below the table, for the sake of convenience, with the terms listed in order of decreasing value. That is, to form a best approximation to the S-box output using k terms, one should select the first k terms of this list.

It should be apparent at this point why the RANK-TERMS procedure was used to put the terms in decreasing order of their individual contribution to S-box correctness. By having the best terms appear first, considerable backtracking

during the search of the COVER table was avoided. As this table was of substantial size, the rather minimal effort involved in ranking the terms individually was deemed worthwhile.

6.2 A HIERARCHICAL APPROACH TO BEST SET DISCOVERY

Due to the computational expense inherent in the combinatorially exhaustive approach to the discovery of the "best sets" which was described in the preceeding section, a more sophisticated technique was later devised in order to reduce the expense of this operation within reasonable bounds.

The 788 seconds of CPU time required for the best set determination for output 1 of S-box 1 was considered to be indicative of the infeasibility of using such an exhaustive-search program on all of the 32 S-box - output pairs. This particular S-box - output pair has a complete approximation containing 17 terms, hence 2^{17} selections are involved in a complete exhaustive search for best sets of size 1...n. To apply such an approach to an S-box output whose approximation contains 23 terms would require 2^{23} comparisons. This is a factor of 64 times more than that required for output 1 of S-box 1, and could be expected to require almost 800 hours of CPU time, to perform exhaustively.

For S-box 1, output 1, it was empirically demonstrated by use of the exhaustive search PL/I system that the property of monotonic addition of sum-of-product terms did hold,

i.e., it was true that there existed a Bi+1 which contained a Bi for all i=1,2,...,n-1, where Bi is a best set formed by the disjunction of i sum-of-product terms.

That such a property is not necessarily true for every possible collection of sum-of-product terms may be illustrated by a simple counter-example. Suppose that the following 3 bit strings represent how these terms "contribute" to the coverage of some hypothetical situation involving $2^6 = 64$ possible input configurations:

1) 101011

2) 110001

3) 001110

The first bit string connotes that the result will be on if input bits 1,3,5, and 6 are on and input bits 2 and 4 are off. One may see that the only possible B1 consists of a set containing term 1 alone. Term 1 has 4 bits on, which is the largest number of any term. The only possible B2, however, is formed as the union of terms B2 and B3 and has all 6 bits on. There are no other possible selections for B1 or B2 which could allow B1 B2 ; all other choices of two terms from the given set of terms have strictly less than 6 bits on.

Accordingly, given that one is willing to slightly compromise the optimality of the approximations obtained for a given S-box's output, but that one insists that such an "optimal" approximation involving n sum-of-product terms be a

subset of some "optimal" approximation of n+1 terms, a computationally tractable algorithm to discover such "quasi-best sets", denoted Bi', has been devised and implemented. In the cases where the sum-of-product terms and the real S-box are such that a monotonic addition of terms to form the Bi sets is possible, as was the case for the first output of S-box 1, then our algorithm will return Bi'=Bi for each quasi-best set Bi'.

This algorithm used to form the Bi' is as follows: Clearly the B1' are those single sum-of-product conjunctive terms which are "on" for the largest number of input configurations for which the specific output of the S-box under consideration is also on. These B1', which are always the same as the actual best sets B1, are thus easily determined by counting the bits on in the bit string which represents the contribution of the term to the correctness of the output. Note that there may be more than one B1'; more than one term may have a maximal number of bits on.

Given the Bi', form the Bi+1' as follows. For each Bi', add to the set of terms comprising this Bi' the one new sum-of-products term which has a contribution bit string which is on for the most input configurations for which the disjunction of terms in the Bi' are off, to form a <u>potential</u> Bi+1'. Note that more than one such potential Bi+1' may be formed for each Bi'. If the total number of input configurations for which this potential Bi+1' is on is as great as

that for any potential Bi+1' formed from any other Bi', then the potential quasi-best set Bi+1' is to be retained as an actual Bi+1'.

This process is repeated until some Bn' is on for all of the input configurations for which the real S-box is. (I.e., until a Bn' has a contribution bit string all of whose bits are on). At such a point, one may trace back through the addition of terms which resulted in the formation of that Bn', in order to uncover the sequence of quasi-best sets: B1',B2',...,Bn'. As a consequence of the manner in which the Bi' were produced, the property of monotonic addition of terms holds for this sequence. With some good fortune, the probability of correctness values of approximations to the S-box formed by the disjunction of the sum-of product terms in such quasi-best sets should not be significantly lower than the corresponding values for the real best sets, which could only be discovered through combinatorially exhaustive search.

The preceeding algorithm may be viewed as the breadth-first construction of, and subsequent trace-back through, an n-ary tree. The tree will possess as many levels as there are sum-of-product terms required for a perfect (Pcorr=1) approximation to the given S-box output. During tree growth, the open nodes of this tree at any level i will correspond to nodes from which the development in parallel of all of the quasi-best sets Bi+1' may occur. Cutoffs will occur at

level i for branches leading to potential Bi+1' which are superseded by the discovery of other potential Bi+1' with better Pcorr values.

The algorithm was implemented as a system of PL/I procedures compiled by the IBM PL/I Optimizing compiler. (See Appendix D). It is worthy of mention that these procedures managed to discover the quasi-best sets for S-box 1, output 1 (for which the property of monotonic addition happened to hold) in less than 33 seconds of CPU time, on the University of Manitoba AMDAHL 470/V7. This represents an improvement in speed over the exhaustive search algorithm by a factor of more than 40.

6.3 N-ARY TREE IMPLEMENTATION

The n-ary tree formed for each S-box - output pair to discover the quasi-best sets contains data nodes with a variable number of pointers. Each node at level i represents a quasi-best set Bi' and contains: the number of the sum-of-products term added to form this Bi' from its father Bi-1', a pointer to this father node, a field (ORMASK) containing a 32-bit string with 1's in positions corresponding to S-box input configurations for which the Bi' approximation is on, and a pointer to a linked list of child pointers. (See Figure 4, Representation of the Quasi-Best Set Search Tree).

It is required that one maintains a linked list of child pointers for each data node, as each such Bi' node may have up to n-i children, given that there are n sum-of-product terms in the complete approximation to the output under consideration. Clearly, such a case would only occur when the addition of any new term to Bi' to form Bi+1' would result in an equally good approximation to the S-box. For reasons of efficient memory utilization such a variability of branching factor implies the use of linked lists to contain each node's child pointers.

For each of the 32 S-box - output pairs, processing begins as it did for the combinatorially exhaustive best set search: by the formation of the binary contribution matrix (CONTRIB) which indicates, for each term, for which of the 32 input configurations where the real S-box is on the term is also on. In this fashion, each term may be represented as a bit string where a 1 in position k indicates that an approximation containing this term will be on for the kth input for which the real S-box should be on.

The top level of the search tree is then formed, from terms whose corresponding contribution vector has a maximal number of 1's. As was mentioned in the preceeding section, such terms must comprise the best sets B1. The top pointer to this first tree level actually points to a linked list of link nodes, each of whose son pointer fields point to the respective data nodes actually containing the sum-of-product

term numbers and other associated fields. During tree growth, a vector (OPEN) of pointers is maintained. The elements of this vector point to tree nodes from which further growth is possible. All top level nodes are initially placed in this OPEN vector.

The following process then iterates to generate the tree in a breadth-first fashion, and continues until a complete sequence B1',B2',...,Bn' of quasi-best sets has been formed. Within this iteration to produce new tree levels, the algorithm loops through all nodes in the OPEN vector in order to produce a new OPEN vector which is to be used in the generation of the next tree level.

For each node in the OPEN vector, it is determined what would be the best term to add to the Bi' in that node to form the potential Bi+1' with a maximal contribution to the correctness of this new approximation. It is possible that there may be more than one such term which could be added to generate approximations with the same degree of correctness. If the new potential Bi+1' is a better approximation to the S-box than any other potential Bi+1' yet formed at this level i, all of these other potential Bi+1' are discarded and are superseded by this newly-formed potential Bi+1'. Even if this supersedence does not occur, if the new Bi+1' is as good or better than other potential Bi+1' yet discovered, it might be added to the tree as a child of the current Bi' being examined.

Whether or not the algorithm is to add this new potential quasi-best set to the tree depends on whether or not there has yet been added to the tree a quasi-best set which contains the same terms as this new quasi-best set. Once terms have been added to the tree at the end of some path, one term being added at each tree level, the order of addition of terms is irrelevant. Neglect of this fact will lead to redundancy of sets in the tree, and vast associated computational expense. Consider, for instance, the case where there exist two B1, sum-of-product terms #1 and #2, both of which have contribution vectors with more 1's than those of any other terms, and where all 1's of the respective vectors are in different positions. Such a case actually occurs for S-box 1, output 1, where terms #6 and #7 both have 4 1's. One potential B2' consists of terms #1 and #2. However, to form another potential B2' consisting of term #2 then term #1, by expanding from level 1 the node containing term #2 is not a reasonable operation to perform. (See Figure 5, Permutation Cutoff).

To avoid these permutations of order of addition of terms when following different paths in traversing the tree, prior to adding a new child to the tree at level i+1 the algorithm examines the new OPEN vector containing pointers to the nodes added so far at this new level, to see if the contribution vector of this new potential Bi+1' is the same as that for a term already added. If this is the case, one may

be assured that the addition of this new potential Bi+1' is redundant, and represents a quasi-best set already added to the tree. One need not even follow back-links to ascertain that precisely the same terms are included in some different permutation in another path to level i+1 if the ORMASK vector is redundant at this level. This is so, as since all sum-of-product terms are unique, to arrive at the same ORMASK at the same level of the tree, one must have used the same terms to form that ORMASK, unless the addition of one or more terms had no effect at all on the ORMASK. The latter condition is impossible, as if this were the case, the term causing an increase in the number of 1's in the ORMASK would not have been added to the tree. In summary, if the ORMASK of the potential quasi-best set to be added is not redundant at the current level of tree expansion, the new node added to form this set is added to the tree.

After all nodes in the current OPEN vector have been processed, any child nodes which remain as potential quasi-best sets for level i+1 are indeed the quasi-best sets for that level as there are no better sets. The new OPEN vector of pointers to the level i+1 nodes becomes the OPEN vector, and the process iterates until some Bn' covers all input for which the real S-box is on.

When such a Bn' is produced, one can follow its father links back to tree level 1, to trace-back and recover the sequence of quasi-best sets B1',B2',...,Bn'. As has been

stated, because of the fashion in which these sets are formed, that Bi' Bi+1' is assured.

There exists another feature of the implementation of this search-tree algorithm which contributes to its efficiency with respect to both memory and processor utilization. This feature is tantamount to a depth-first component of the tree search which is activated under certain specific conditions.

In general, the n-ary tree is generated in a breath-first fashion. That is, all nodes are generated at some level i, prior to any of the level i+1 nodes being formed. This is essential, as we wish to be guaranteed that the quasi-best set sequence we ultimately discover is optimal in the sense defined earlier for such quasi-best sets. If the tree was not generated breadth-first in parallel, the satisfaction of this condition would entail a rather complex backtracking operation. Some path would be produced depth-first to level n, and exhaustive backtrack exploration of every junction at every level would be required, to ensure that some other complete path from the root to level n did not contain some Bi' at level i which was a better quasi-best set than that discovered on the former path. Such a depth-first expansion would generate the complete tree, as we have just seen to be required to assure the optimality of the sequence B1',B2',...,Bn', and would also involve the additional computational expense associated with the backtracking operations.

Nothwithstanding the preceeding argument, there is indeed a requirement for some depth-first component in our search algorithm. Initial implementations of the search which neglected to consider this feature expanded a vast number of nodes at the lower levels (large i) of the tree for certain S-box - output pairs. Analysis of the factors which produced such an undesirable situation resulted in the inclusion of the following modifications to the basic search algorithm.

At some point in the breadth-first generation of the search tree, all nodes at level i+1 will be such that they represent an approximation to the actual S-box which is correct for exactly one more S-box input than any of the nodes in the preceeding tree level i. The crux of the matter is that since the tree is formed in a best-first fashion, with the algorithm adding at <u>earlier</u> tree levels the terms whose contributions add <u>more</u> 1's to the ORMASK contribution of the Bi', once a time is reached where only one more 1 bit is added to the approximation of the preceeding tree level, no later term choice may subsequently add more than 1 bit per level. As each sum-of-product term must contribute something to the correctness of the overall approximation in order to have been returned by the Quine-McCluskey minimization routines, it must be that all as-yet-unused terms will add <u>exactly</u> one bit to the correctness of the approximating expression. Thus, at this point, the algorithm need no

longer expand the tree breadth-first, but can penetrate immediately in a depth-first manner to level n, adding as-yet-unused terms in an arbitrary order to any node in the OPEN vector to form the desired B1',B2',...,Bn' sequence.

More explicitly, once it has been discovered at level k that only terms "adding" one bit of correctness to the approximating expression remain unused, only one path in the n-ary tree from level k+1 to level n need be formed in order to ensure that the nodes on the resulting path form a sequence of optimal quasi-best sets.

Applied to the DES S-box functions, this set of routines was capable of discovering these quasi-best set sequences in reasonable time. As mentioned at the end of Chapter IV however, the principal direction of this thesis has been to work with accurate expressions for the S-boxes, as the sophisticated techniques described in the following chapter permitted the size of the S-box functions to be reduced sufficiently to allow a special type of search to be (marginally) tractable, without the need to use approximations to the S-box functions. The potential applications of approximations to the S-box functions as have been produced by the routines of this chapter remains a topic for future research.

Chapter VII

SPECTRAL DOMAIN S-BOX ANALYSIS

7.1 ORTHOGONAL TRANSFORMATIONS TO THE SPECTRAL DOMAIN: THEORY

Our previous attempts at minimization of the Boolean functions embodied in the S-boxes have not proven very useful. One conventional technique of Boolean minimization, the Quine-McCluskey method, has allowed us to reduce the number of p-terms in accurate sum-of-products expressions for the S-boxes and their complementation from 32, to between 14 and 23, depending on the S-box - output pair. While this does represent a significant decrease in the effective branching factor of the search tree for K when compared with that of 32 if the elementary p-term expressions for the S-boxes were employed, as may be seen from the probability of correctness values in the table on page 145, nearly all of these minimal p-terms must be retained in an S-box approximation, for the approximation to have a high (>.9) probability of correctness value. As a result of this indication that the use of approximations to the S-boxes may not have many applications towards cryptanalysis, we turn instead to methods which are capable of minimizing certain classes of Boolean functions more effectively than can the Quine-McCluskey procedure.

Discussions with an expert in the field of digital logic design led to experimentation with a number of more recently-developed Boolean function manipulation and minimization techniques, as are employed in the field of logic design for hardware applications. Specifically, such techniques refer to the use of suitable transforms to permit the manipulation of Boolean functions in a "spectral" domain, analogous to the use of the Fourier transform to allow manipulation of real functions in a frequency domain. For a more comprehensive treatment of the subject than can be included in this thesis, the reader is referred to several excellent recent works on the subject by Hurst [7,8] and Karpovsky [10].

Consider a Boolean function of n variables defined by a vector of 2↑n bits, which represent the output values of the function for each of the 2↑n possible input configuations. Let us term this the specification vector, Fs. Knowledge of any particular bit of Fs does not decrease the entropy, in an information-theoretic sense, of any other bit of Fs, unless other a priori knowledge of some characteristics of the function is avaliable. We wish to represent Fs in some other domain in which any correlation between the outputs of the Boolean function and its inputs will be more evident. That specific members of a set of elements of Fs have the same value is often indicative of some structure of the function, yet this structure is not made explicit by the Fs vector representation. As a dramatic illustration of this

fact, consider the following specification vector for a function of 4 variables.

Fs=(0 1 1 0 1 0 0 1 1 0 0 1 0 1 1 0)

Notice that Fs contains 2^4=16 entries, one for each possible input configuration 0000 to 1111 for the function. In this representation, it is not at all clear that the function is actually

$f(X1,X2,X3) = X1 \oplus X2 \oplus X3$

In fact, this high degree of structure is quite obscured by its representation as the vector Fs. While topological methods such as the Karnaugh map technique [16] are capable of making some types of structure more explicit, these methods are difficult to program.

If the function were represented in a different domain, the detection of any symmetries possessed by the function may be more easily accomplished. There exist several well-known techniques which permit the mapping of any Fs to this alternative spectral domain, where these transformations may be accomplished by means of a matrix multiplication [7,8]. These transformations will be invertible; the information content of the specification vector is preserved when mapping from one domain to the other. The representation of Boolean functions in the spectral domain will use numbers not confined to the range {0,1}.

Square 2↑n x 2↑n orthogonal matrices with entries +1 and -1 are used to indicate the mappings between the Boolean and

spectral domains. The "spectral domain" refers simply to a domain in which different basis functions are used to represent any desired function, just as sine and cosine basis functions of varying frequency are used to represent real functions in the Fourier domain. In this Boolean spectral domain, the basis functions employed will be XOR functions of various variables, the functions being specified in the rows of the transform matrix, T [8]. The mapping from the conventional functional Boolean domain to the spectral domain can be defined by this orthogonal mapping matrix T. (T^{-1} is the inverse mapping, from the spectral to the Boolean domain. T^{-1} will always exist, and $T^{-1} = T'$, as T is orthogonal).

For the matrix multiplication to preserve the information content of the specification vector, the mapping to the spectral domain is actually a mapping of a modified specification vector with entries from a different range. The vector F is actually mapped, where the entries fi of F correspond to the entries fsi of Fs through the equation:

$$fi = 1 - 2fsi \quad \text{for all } i = 1 \ldots n$$

Thus, +1 corresponds to the usual Boolean 0, while -1 corresponds to the usual Boolean 1. This new representation is required, as the presence of 0's in the mapped vector would result in the matrix multiplication procedure (over the real field) causing a loss of information, as the multiplication of 0 with either 0 or 1 returns the same result.

Mapping with this new specification vector F results in a spectrum R of the function whose specification vector is Fs. Mathematically:

$$R=TF$$

$$\text{and } F=T^{-1}R$$

The entries of the vector R, ri for i=0...2↑(n-1), are termed the spectral coefficients of the function.

These ri are commonly interpreted as the coefficients of the correlation between the outputs Fs of the Boolean function, and XOR's of various combinations of the input variables [7]. In order to explain what is meant by this, consider the coefficient r5 of a 4 variable function, f. As 5 is 0101 in binary, the r5 coefficient gives the correlation of Fs with X2⊕X4, where X2 and X4 are the 2nd and 4th input variables to the function. (These variables corrspond to the 1s in the binary representation of the coefficient number). By "correlation" is meant the number of times the output of the basis function X2⊕X4 equals the output of f, minus the number of times it differs. Clearly, if it happens that f=X2⊕X4, then r5=2^4, its maximum possible value. Thus, each ri gives some aspect of global information about the entire function.

There exist many possible variants of orthogonal transformations T which are nonetheless all the same, independent of row permutations. These bear different names, as they were developed independently, and include the: Hadamard,

Walsh-Kacmaz and Rademacher-Walsh transforms. We shall employ the former, defined recursively [8] as:

$$To=[1]$$

$$Tn=\begin{pmatrix} Tn-1 & Tn-1 \\ Tn-1 & -Tn-1 \end{pmatrix}$$

With this choice of T, we may divide the ri into three disjoint sets on the basis of the quantity whose correlation with the Boolean function specification vector Fs they represent. Each spectral coefficient is classified into one of the sets on the basis of its order, where the order of a spectral coefficient ri, denoted ||ri||, is defined as the number of 1s in the binary number representation of its subscript i.

The zero-ordered coefficient, ro, provides only a measure of the number of +1's and -1's in F. Each of the ri for which the binary representation of i has exactly one 1 (there are n such ri) are termed the primary spectral coefficients, and measure the correlation of each of the independent Boolean variables xi, for i=1...n, with the Fs vector of function outputs. The remaining (2↑n)-n+1 spectral coefficients constitute the secondary spectral coefficients. These represent the correlation of fs with all other possible XOR combinations of the input variables. For an example in which n=6, r7=r000111 measures the correlation of Fs with the function X4⊕X5⊕X6. Negative values for any spectral

coefficient indicate correlation with the complement of the Boolean function.

Computationally, there exist techniques to calculate the spectral coefficients R which are less expensive than an ordinary matrix multiplication. The calculation of R by multiplying T by F entails $((2\uparrow n)\uparrow 3)$ multiplication operations for an n variable function, as the size of T is $(2\uparrow n)x(2\uparrow n)$. Various "fast" transforms are well-known in the literature on the subject [7,8], and operate in $O(n2\uparrow n)$ time. However, such fast transforms will not be employed in the following analysis of the DES S-box spectra, as for this case where n=6, the very simple matrix multiplication procedure is of adequate speed, as only 64^3 multiplications are required.

7.2 S-BOX COMPLEXITY IN THE SPECTRAL DOMAIN

One of our concerns involves the complexity of the Boolean functions which represent the action of the S-boxes. The complexity of AND/OR circuitry required to realize the effect of the S-boxes corresponds directly to the degree of branching in the search tree at the point where the action of the S-box is functionally inverted during search for the encryption key, K. In order to be able to effectively minimize the complexity of any Boolean function, it is necessary to define a metric capable of usefully measuring this complexity. Algorithmic procedures may then be devised to minimize the value of this complexity function.

One classically useful metric for Boolean function complexity, employed in the conventional Boolean domain, has been found to be a count of the number of adjacent pairs of input variable assignments for which the function f has the same output values for both assignments [16]. Geometrically, this corresponds to a count of pairs of adjacent 1's and 0's in the Karnaugh map representation of the function. The higher the value of such a complexity metric for a function, the more easily that function may be synthesized using conventional AND and OR gates.

It has recently been shown [9] that the same metric may easily be computed in the spectral domain from the inner product of the square of the spectrum R, and the corresponding orders of the spectral coefficients in R. Following Hurst et. al. [9] we shall use the spectral domain complexity estimator:

$$C(f) = n2^{n} - \frac{1}{2^{n-2}} \sum_{v=0}^{2^{n}-1} ||v||\ rv^{2}$$

where the order of the vth spectral coefficient rv is $||v||$.

If the spectral coefficients of some function f are dominated by the primary coefficients, i.e. the largest magnitude coefficients are among the primary coefficients, then C(f) will be high and the function may be represented as a sum-of-product expression with few terms. Otherwise, if the largest magnitude coefficients are among the secondary coefficients, C(f) will be low and the function may only be represented by a more complex sum-of-products expression.

Considered informally, this latter situation will tend to occur when the function is heavily "XOR-oriented", and may be more easily realized by means of XOR operations than by AND and OR operations. As shall be described, in such a case the use of certain spectral translation operations to shift the largest magnitude spectral coefficients into the primary range may be advantageous. The C(f) complexity estimator will prove to be of use in measuring the degree of simplification of the function f effected by such translations.

7.3 SPECTRAL TRANSLATIONS

It is possible to manipulate any Boolean function in the spectral domain so as to maximize the value of the C(f) complexity metric for that function. Groups of spectral translation operations are performed on the rv vector in order to permute this vector in such a manner as to shift the largest magnitude spectral coefficients into positions of primary coefficients. The "core" function f' which remains after all translations have been performed and the translated function is mapped back to the conventional domain by applying T^{-1} to its permuted rv vector is conjectured [10] to be of maximum possible simplicity. Classical minimization techniques, such as the Quine-McCluskey method, may then be applied to the core function to produce a minimal sum-of-products form.

Following Karpovsky [10: p.69], the required spectral translations may be seen to be of the form:

$$f(x1,x2,\ldots,xn) = f'(x1,x2,\ldots,xi-1,xi\oplus xj,xi+1,\ldots,xn),$$
$$i,j,i\neq j, \in \{1,n\}$$

Inputs to the function f are replaced by XORs of inputs, to form the translated function, f'. By a repeated application of such translations, the complexity present in any function can be factored out into a tree of XOR gates through which inputs to the simplified core function are conditioned. Translations of this nature are performed until all of the n primary spectral coefficient positions of the n-variable function are occupied by the coefficients of the largest magnitude.

In practice, the tree of XORs which condition the inputs to the function f' may be represented by a basis matrix B of 1's and 0's through which any input to f' must be multiplied under GF(2) to simulate the effect of the XORing on the functional inputs. Schematically, the following situation exists:

(a) Before translation:

$$\text{input x vector} \xrightarrow{f} \text{output bit}$$

(b) After translation:

$$\text{input x vector} \xrightarrow{B} \text{modified inputs} \xrightarrow{f'} \text{output bit}$$

where B is the matrix through which the x inputs are multiplied, and f' is the simplified function which results from a conventional minimization of the translated function as returned to the functional domain

Any input vector x is mapped to the same output value by either the original function f, or by the combined action of the B matrix and the simplified f'. The technique of spectral translation is exemplified in the following section.

7.4 IMPLEMENTATION FOR DES

APL routines were devised to follow Karpovsky's algorithm [10] to perform these translations for the production of the mapping B and simplified f', and were applied to the 32 functions which represent the DES S-boxes. The routines discussed in this section may be found in Appendix E.

The routine SPECTRUM maps a function's specification vector to the spectral domain, by application of the appropriately-dimensioned Hadamard transform matrix, T. FUNC applies the inverse mapping T^{-1} to a vector of spectral coefficients to return the representation to the conventional functional domain. Both of these routines call the function TRANS, which recursively builds the required orthogonal Hadamard transform matrix. The function COMPLEXITY applies the formula for computation of the C(f) complexity metric to a vector of spectral coefficients.

After the spectrum of an S-box function has been formed by application of SPECTRUM to the Fs specification vector for that function, the BASIS routine is called with the spectral coefficients, to determine the translations required to maximize the primary coefficients. These transla-

tions are represented in the form of a matrix BAS, which when transposed and inverted under GF(2) will serve to indicate how the rv vector must be permuted. That is, the mapping matrix B referred to in the previous section is simplify the transpose of the inverse of the matrix BAS, whose creation shall now be discussed. For a complete understanding of the procedure, the interested reader is referred to Karpovsky [10].

In BASIS, the largest coefficient in rv is discovered, and its corresponding position v, represented as a binary number, is added as a new row of the initially-empty BAS matrix. Elements in rv whose position corresponds to any possible linear (XOR) combinations of rows which already exist in BAS are then deleted, to remove them from further consideration as large-magnitude coefficients. After BAS has acquired n rows, at which time all entries in rv should have been zeroed by the above process as all linear XOR combinations of the rows of the complete BAS must span then entire spectral space, it is transposed and inverted under GF(2) to form the basis matrix for the permutation of rv. The matrix BAS is the mapping B discussed in the preceeding section through which inputs to the new f' must be mapped.

For the purposes of illustration of this techniques, let us consider the function f of 4 variables:

f = X1X2' + X1'X2 + X3X4' + X3'X4 + X1X2X3'X4'

Inspection reveals that this function is heavily XOR-oriented. In fact

f = (X1⊕X2) + (X3⊕X4) + X1X2X3'X4'

The specification vector Fs for f is found to be

Fs = (0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 0)

and multiplication into the 16x16 transform matrix T produces a spectrum

R = (13 1 1 -3 -1 -1 -1 -1 -1 -1 -1 -1 -3 1 1 -3)

When the BASIS function is invoked to produce a basis matrix for the translation of these coefficients ri, the following operations occur, in accordance with the preceeding description of the translation algorithm. The largest element in R (not considering r0) is found in r3. Thus, the row (0 0 1 1), for "3" in binary is catenated as a new row of the (originally empty) basis matrix. All linear (XOR) conbinations of rows in the basis are formed, and positions of R corresponding to these combinations are zeroed. As in this case the basis has only one row so far, only position 3 of R is zeroed.

The largest element of this new R is then located at r12. As a result, the new row (1 1 0 0) is added to the basis. Positions 3 and 12 of R are zeroed as a result of considering combinations of the vectors in the basis taken one-at-a-time. Position 15 of R is also zeroed as a result of combinations of the rows of the basis considered 2-at-a-time.[7]

---

[7] As (0 0 1 1)⊕(1 1 0 0)=(1 1 1 1), or 15.

The largest element of this modified R is now in r1 (only 1s remain unzeroed in R). The basis vector (0 0 0 1) is added, and positions of R corresponding to all possible XOR combinations of the 3 rows in the basis are zeroed. The process is repeated once more to add (0 1 0 0) to the basis matrix, at which point the entire basis has been formed (and all elements of R are 0).

MAXPRIM is a routine which accepts the basis matrix and the coefficients rv, and returns rv permuted by the mapping implied by the basis. In MAXPRIM, the 2|n possible input configurations for f are mapped through the basis, and the resulting sequence of configurations taken to define a permutation of the original inputs. This permutation, when applied to rv, produces the vector of spectral coefficients for the simplified f', where all of the largest magnitude coefficients occupy primary positions.

To continue our earlier 4-variable example, the required permutation of the spectrum R according to the basis matrix formed by the BASIS routine is

R' = (13 -1 1 -1 -3 -1 1 -1 -3 -1 1 -1 -3 -1 1 -1)

Mapping this permuted spectrum through the inverse Hadamard transform T^{-1} yields a specification vector Fs' for the new "core" function f'

Fs' = (0 1 0 0 1 1 1 1 1 1 1 1 1 1 1 1)

As may be simply discerned from a Karnaugh map, this corresponds to a function

f' = X3 + X4 + X1X2'

which is evidently far more minimal than was the original f. The complexity of the function f attributable to XORs has been removed.

The inverse (over GF(2)) of the transpose of the basis matrix may be found to be

$$B = \begin{pmatrix} 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

Any input multiplied through this matrix and then subjected to f' will be found to have a value identical in all cases to that of the same input subjected to the original f.

Results of the application of these simplification routines to the DES S-box functions are remarkable. Table 2 presents the C(f) complexity measure for each of the 32 functions both before and after the simplification by means of spectral translations. The average complexity prior to translation was 139. Following translation, the core functions exhibited much greater simplicity, with an average measure of 247. This means that in a Karnaugh map representation for the S-box functions, there are on average approximately twice the number of adjacent cells containing the same value as there were prior to the spectral translation procedure. This has a great impact on the simplicity of minimizations of the new "core" functions by conventional Quine-McCluskey techniques.

The translated f' were returned to the functional domain and subjected to the Quine-McCluskey minimization. The functions that resulted had between 9 and 13 p-terms per function. These are far fewer than the 14 to 23 p-terms of the Quine-McCluskey minimized S-box functions (Chapter V). The overall average number of literals per p-term decreased following the spectral simplification to 3.57, from a value of 4.96 prior to simplification. The ramifications of this substantial improvement in simplicity for the key search is discussed in Chapter IX.

Chapter VIII

UNIDIRECTIONAL CRYPTANALYTIC SEARCH

It has been stated that our aim is to functionally "invert" the DES encryption algorithm, so as to be able to determine the values of all bit positions of the encryption key K used as a mapping between the known corresponding pairs of plaintext P and ciphertext C. This particular chosen plaintext attack may conveniently be viewed as a problem of search: Given the constraints imposed by the bits of P and C and the details of the DES algorithm, a search may be conducted to determine the assignments to the bit positions of K which satisfy these constraints.

As is typical for problems of this nature, a search tree may be constructed and traversed in the course of the determination of the K bit values. A tree is defined to be a loop-free directed graph with a distinguished node of indegree 0 (the root). A subtree is any subset of the nodes of a tree which themselves form a tree. A node is said to be "satisfied" if the assignments to bits of K which are required by the complete development of the subtree from that

---

[8] By "key bit hypothesis" is meant an assignment of a value from {0,1} to a specific bit position of K which is not contradicted by any other assignments to key bits required so far during the development of the search tree.

node are compatible with current key bit hypotheses[8] for K.[9]

The search tree will contain both "and" and "or" nodes, where the former is the root[10] of a subtree, all of whose immediate children must be satisfied, and the latter is the root of a subtree, any of whose immediate children must be satisfied for the node itself to be satisified [17]. Whereas all children of an "and" node will be developed breadth-first in parallel, only one child of any "or" node will be present in the search tree at any given time. A contradiction in key bit hypotheses will cause a recursive backtrack to the most recent "or" node for which alternative children still remain, and cause the selection of such an alternative branch at that node.

Within our particular search tree, constraints will reside at both the top level of the tree and at the leaves.[11] This occurs as both the image and preimage of the K mapping are known, in accordance with the assumptions of a chosen plaintext attack. Inversion of the action of this K mapping as driven by the DES algorithm comprises the body of the

---

[9] If no hypothetical assignments for relevant key bits yet exist, any assignments engendered by the development of the subtree for a node by selection of the first disjunctive possibility at each choice point will seem correct, and will remain as hypotheses until the development of a different node leads to contradictory hypothetical assignments for some bit of the key.

[10] The root of a tree is the node with indegree 0, i.e. the node with no edges entering.

[11] Leaves of a tree are nodes with outdegree 0, i.e. nodes from which no edges emanate.

search tree. At the top of the tree, the value of each bit in the ciphertext block C is known. At the leaves, the bits in the corresponding plaintext block P are known. In working back through the encryption procedure from round 15 to round 0 to ascertain how the particular bits of C came to have their respective values, various hypotheses concerning the values of the bits of K will be generated.

Until the search procedure is completed, these key bit assignments will only be hypothetical and may be contradicted by further search tree development. The presence of OR nodes in the search tree will allow backtracking throughout the tree to permit alternatives in tree development which could lead to different key bit assignments, in the event that contradictions occur in key hypotheses during tree growth. For an illustration of one variant of the AND/OR search tree discussed in this chapter, see Figure 6, Partial Search Tree for 2-Round DES.

As Figure 6 embodies all of the significant features of a tree search, we shall attempt to precisely explain its characteristics. At the root are situated the (known) 64 bits of the ciphertext C. In the tree, nodes with arcs below them are "and" nodes. All subtrees which descend from an "and" node must be satisfied, for the "and" node to be satisifed. For this reason, in an implementation of this tree search, the expansion of subtrees from an "and" node will occur in a breadth-first manner. There is no advantage to delaying the

construction of these subtrees, as all must be expanded at some time.

Nodes without arcs are "or" nodes. "Or" nodes are satisfied if any of their children are satisfied. For this reason, "or" nodes will be expanded in a depth-first manner in an implementation of this search. There is no need to expand in several directions simultaneously, when expansion of a single subtree suffices. Alternative "or" paths are expanded only if the search down one "or" path fails, and backtrack to the "or" node necessitates the selection of an alternative path.

An expansion of any node in this search tree constitutes an "inversion" of some aspect of the encryption procedure. Various points in the search tree of Figure 6 are numbered with the circled digits (1) through (6), to aid the following description of the process of formation of this tree.

At (1), it can be seen that the leftmost bit of C, that is, bit position 2 of the L block at encryption round 2, has a value of 1. It follows that bit position 32 of the R block at round 1 must also have had the value 1, as Ln=Rn-1 by the first of the two DES equations [26].

It is the second of the two DES equations [26]

$$Rn = Ln-1 \oplus f(Rn-1,Kn)$$

which accounts for the expansion of the subtree from (2). From (2) it may be reasoned that either L at round 0 position 32 was 1 and bit 32 of the output of the f function op-

erating on R0 and K1 was 0; or else that the L was 0 and the f function was 1. This is because the XOR operator indicates non-equivalence. Assuming the former of the two possibilities (since (2) was an "or" node, these two disjunctive possibilites for subtrees from (2) are explored one-at-a-time) the node marked (3) is to be expanded.

If the output from the f function at position 32 was 0, this implies that the value of S-box 6, output 1 was 0, as 32 mapped back through the inverse of the P permutation of DES is 21, and the 21st output of the bank of S-boxes refers to this particular S-box - output pair.

Knowledge concerning this value of the output thus constrains what the inputs to S-box 6 could have been. In Figure 6, it is reasoned that if this output of the S-box was 0, then the DeMorgan complement of our minimal expression for the Boolean function which represents S-box 6 output 1 must have been 1. While this is true, such reasoning can lead to the expansion of more nodes than are necessary, as a minimization of the <u>complements</u> of the S-box functions would permit an "or" of "ands" to extend from (3), instead of the "and" of "ors" which Figure 6 depicts. Nevertheless, as Figure 6 appears, at (4) we must construct "or" subtrees to permit x1=0, x2=0, and x3=1, where the xi for i=1,2,...,6 are the 6 input variables to S-box 6.

At (5) we consider the problem of how to make a given S-box input have a certain value. From the specification of

the DES algorithm [26], it may be seen that inputs to the S-boxes are formed by an XOR of bits of K with E-permuted bits of R from the previous level. Utilizing details of the key selection algorithm, and mapping the position of the first input to the S-box 6 through the inverse of the E permutation, it may be concluded that the quantities XORed to produce the first input to S-box 6 are key bit 5 and bit position 16 of the R block at round 0.

At (6) it is realized that as the R block is at encryption round 0, it corresponds to a bit of plaintext, whose value is known by our attack assumptions. As discussed later, if either our assignment of a value to the bit of K, or the discovery of the value of the bit of R by this process of inversion constitutes a contradiction with respect to what has already been discovered, backtrack occurs, to explore different disjunctive paths. This procedure is explained further in section 8.1.

It should be noted that for reasons of computational tractability, we shall be considering a 2-round DES encryption algorithm throughout this chapter. Techniques which serve to reduce cryptanalytic time for such a simplification of the DES algorithm may be applied to the actual 16-round DES algorithm to achieve a similar time saving. Such a simplification of DES by a reduction of the number of encryption rounds is frequently employed to permit inexpensive experimentation with cryptanalysis [5]. In addition, the IP

and IP^{-1} permutations are not being considered in our model, as they are of no cryptanalytic significance under the conditions of a known plaintext attack.

Exhaustive key search is a technique in which all possible values of K are used to map the plaintext block P to see if the expected C is the image of the mapping. The desirability of a search tree cryptanalytic approach relative to this brute-force procedure of exhaustive key trials may be seen to be a function of the degree of simplicity of the S-box representations as they are embodied in the DES f function. Indeed, the choice of S-box representation is the only variable parameter of the search. The DES equations:

$$Ln = Rn\text{-}1$$
and
$$Rn = Ln\text{-}1 \oplus f(Rn\text{-}1, Kn)$$

lead to constant branching factors in the search tree of 1 and 4,[12] respectively. The branching factor of a tree is defined to be the average outdegree of nodes in the tree. Similarly, the branching factors caused by the E and P permutations, the KS key schedule function, and the XORing of the KS output with an R block to form the S-box inputs are all constant. There is no apparent way to reduce their associated branching factors by any alternative representations of the operations they embody. Consequently, reduction of the search tree size must be effected by the compact repre-

[12] For an XOR of two terms to have the value 1 either the first term must have the value 1 and the second the value 0, or vice versa, which implies that up to 4 paths may have to be expanded to satisfy a node. Similar reasoning applies for an XOR which must have the value 0.

sentation of the S-boxes, perhaps coupled with the clever use of heuristics to accomplish tree pruning prior to or during development, or to guide tree traversal. For instance, should the elementary p-term expressions for the S-boxes as discussed in Chapter 3 be employed as the representation of the S-boxes during the search for K, the search tree would possess a branching factor of 32x6=192 at each point in the search where the action of the function f operating on some arguments needed to be inverted. This is the case, as when the S-boxes are represented in the elementary p-term form, the value of any given output of a specific S-box is specified by a disjunction of 32 terms, each of which is a conjunct of 6 literals.

A simple calculation will show that in this event, the complete search tree will possess far more nodes than there are trials in an exhaustive key search[13] (2^{56}). From Figure 6 it may be seen that the following expression constitutes an upper bound on the number of nodes in the search tree.

$$64 \times \sum_{i=1}^{15} (2 \text{ t } 1)^{i}$$

where: t=maximum number of conjunctive terms in any s-box representation
l=maximum number of literals in any conjunctive term

[13] The validity of comparing number of key trials to tree size is discussed in the first section of the next chapter.

There are 64 subtrees at the first level of the tree, all of which have the same worst-case maximum possible number of nodes. At each of the subsequent 15 tree levels, each node has at most (2 t 1) children, where the 2 constant arises from the XOR, and the t and 1 variables from the sizes of the minimal expressions discovered for the S-box functions.

For the elementary p-term S-box expressions, t=32 and 1=6, so the above expression has the value of $1.2 \times 10^{45} >> 2^{56}$, the key space size.

It should however be borne in mind that in practice, such a complete search tree would never be grown. Some subtrees may be pruned on the basis of mutual incompatibility of necessarily conjunctive conditions within those subtrees. As well, all n subtrees from an OR node are only developed if the first n-1 subtrees cannot be satisfied, a condition which is highly unlikely to occur for all OR nodes in the entire search tree.

Due to such factors, it is difficult to analytically predict how concise the S-box expressions must be for a search tree approach to cryptanalysis to be superior to exhaustive key search, although a "worst case" analysis of the situation is attempted later.

8.1 SEARCH STRATEGY

The top-down search[14] for K will be performed in a manner which utilizes a combination of breadth-first and depth-first search strategies [17]. Since for a 2-round DES core memory constraints on the number of nodes in the tree are not significant, and as all branches from AND nodes must be followed eventually, a breadth-first parallel-expansion discipline is followed at AND nodes. At OR nodes, which may be considered "choice points" in the search, the expansion is depth-first, as if any branch from an OR node is satisified, so is the OR node. It is not reasonable to expand all alternative branches simultaneously, when the satisfaction of only one branch is required. Backtracking to these choice points upon failure at lower tree levels will allow us to attempt the satisfaction of other alternatives should this be required. That the conjunctive terms in the sum-of-products expressions for the S-boxes were ordered by their contribution to expression correctness, as discussed in Chapter 6, adds a heuristic element to this search. At all choice points, the search tree is expanded in a "best first" fashion. As an implementation consideration, father pointers are maintained in each node, to facilitate the backtracking to the most recently expanded OR node in the event of the failure to satisfy some subtree.

---

[14] A top-down search is a search which commences at the root (top) of a tree, and proceeds downwards towards the leaves.

That is, if some choice as manifested by the expansion of the tree from some specific branch of an OR node necessitates that a certain bit position of K be assigned a value when it already possess a different value, then either our most recent choice to follow this particular branch from the OR node, or the choice which had previously resulted in the assignment to the key bit, is incorrect. The two choices are mutually incompatible. Recursive backtracking techniques can be used to undo the most recent choices first.

If no disjunctive alternative at the most recent choice point (OR node) may be expanded without the result being an inconsistency of key bits, then an earlier choice must be in error, and backtracking must occur to re-make this earlier choice. If accurate representations of the S-boxes are employed, then there must exist some selection of OR node paths which permit a unique assignment to all key bits. If this is not possible, an error must be present in either the search procedure or the correspondence between the P and C blocks. However, if the search is employing approximations to the S-boxes, the procedure may backtrack all the way to the root of the search tree, a condition which indicates that another P-C pair should be used in the search. If this arises, the use of approximations to the S-boxes has led to irreconcilable inconsistencies in what the key bits must have been.

In summary, there are three types of conditions whose occurence leads to backtracking in the search:

1. If the search has reached the bottom level of the tree through some path, which implies that the entire process of encryption has been inverted for some bit of the plaintext, the value of the actual known plaintext at the corresponding bit position must agree with this bit of the round 0 encryption block uncovered by the search. As such bits of "plaintext" are produced by the search, if they fail to agree with the corresponding position in the known plaintext, backtrack must occur, to follow disjunctive paths in the search other than those which led to this erroneous development.
2. Secondly, if during the inversion of an encryption operation which set some position in the vector of inputs to an S-bank to a certain value, it is required that a bit of K be assigned a certain value and it is the case that this position of K has already been assigned a different value, then backtrack must occur to remake earlier erroneous path-selection decisions.
3. Finally, if it happens that no more alternative expansions exist for a node which must be expanded in a new way as a result of the search backtracking to that node, the backtracking is invoked recursively, to backtrack yet higher in the tree.

It should be noted that with our 2-round simplified DES, this search procedure may not establish the values of all 56 bits of K, as not all of these key bits are produced by the permutations and selections of K provided by the first three cycles of the KS algorithm. Consequently, the assignments made to bits of K which are not used in our simplification of the DES algorithm are arbitrary.

At this point in the discussion, it should be noted that the use of a bidirectional search [17,18] to discover K seems intuitively appealing, as the cryptanalytic task involves constraints at both the root and leaf nodes, and as the encryption and decryption algorithms are identical. It should be possible to drive the search backwards from the ciphertext towards the plaintext as the above discussion has described while at the same time, one is driving forwards from the known plaintext. For a known plaintext cryptanalysis of a full 16-round DES, one would search 8 rounds forwards from the plaintext, and 8 rounds backwards from the ciphertext and join the two search trees in the middle. The existence of constraints at both ends of the search tree, and the symmetry of encryption and decryption make this approach possible. A more formal argument which indicates why the bidirectional scheme is viable may be seen in the next chapter.

8.2 NODES IN THE SEARCH TREE

The approach employed in our search requires the existence of 3 major distinct types of data nodes in the search tree, in addition to a descriptor type of node which is associated with every other type of node and which contains information common to all node types. Each type of data node possesses a unique structure, is expanded differently from the other types of nodes, and requires a different response if backtracking reaches the node. This heterogeneity of nodes is necessitated by the fact that the DES encryption algorithm involves a number of different operations in each round. A thorough discussion of these types of nodes and their characteristics follows. (See Figure 7, Nodes in the Cryptanalytic Search Tree).

8.2.1 Descriptor Node: SUPER

Two factors necessitate the inclusion of a descriptor node structure associated with the actual data portion of each of the 3 major types of data nodes in the search tree. Firstly, such a structure, known as the based-storage structure SUPER in the PL/I routines which perform the search, permits data common to all types of nodes to be factored out of these nodes. This ability serves to simplify the structure of the actual data nodes.

More importantly, the PL/I language does not permit pointer reference to based-storage data, unless the symbolic

name of the based variable is known. This in turn implies that during the traversal of an existing tree structure, as occurs during recursive backtrack, one must know what type of node one has linked to, before it is possible to access the link fields in that node. As we are dealing with a tree structure with heterogeneous nodes which may be linked in a wide variety of ways, it is crucial to be able to link to a node without knowing its type a priori. A field in SUPER which contains the type of the data node for which the SUPER node is a descriptor permits traversal in this fashion.

The SUPER node contains 6 fields. As mentioned, there is a single character type field, which allows the search routine to determine for which of the 4 types of data nodes it is a descriptor. A position field contains an integer in the range {1,32} to indicate which bit of the current ciphertext block is represented by the value in the data node. Similarly, a level field contains an integer from {1,16} to indicate at which level in the 16-round encryption procedure the block in which the bit occurs is contained. A father pointer points to the SUPER node associated with the immediate ancestor of the current node in the tree. There is a pointer to the actual data node which is described by the SUPER node. Finally, there is a pointer to the node on the queue of nodes which are "open" for further development which points to the SUPER node. If it is the case that the SUPER node has already been expanded and is not still a tem-

porary leaf of the expanding tree, this OPENQ pointer is null. The need for such a pointer is discussed in the section pertaining to the BACKTRACK routine.

8.2.2 Data Node: RNODE

The RNODE structure permits representation of the knowledge that at some specific level in the encryption process a specific bit position in an R-block possessed a certain bit value.[15]

The position and level information for the RNODE resides in its descriptor. In the actual RNODE, there exist fields to represent the bit value of the RNODE, a count field from {0,2} to indicate how many times the node has previously been expanded, and two child pointers. Where these child pointers must point may be deduced through the examination of the second of the DES encryption equations:

$$R_n = L_{n-1} \oplus f(R_{n-1}, K_n)$$
$$= R_{n-2} \oplus f(R_{n-1}, K_n) \qquad \text{(see footnote)}$$

If the RNODE being expanded has a value 1, then its two children (the RNODE at level n-2 and the FNODE) must have respective values 1 and 0, or 0 and 1, as a result of the XOR. These 2 possibilites are disjunctive. The first is de-

---

[15] No analogous node to represent knowledge about L-blocks is required. As Ln=Rn-1 by the DES algorithm, it is known immediately that instead of representing some bit position p of an L-block at level l of encryption which possesses a value v, the RNODE which must have been the predecessor to the LNODE during encryption may immediately be created. This RNODE will represent bit position p at level l-1 and have the value v.

veloped the first time the RNODE is expanded (when its count field is 0), and the second is developed as an alternative should backtrack ever reach the RNODE, as a result of the occurence of some key bit contradiction.

Otherwise, should the RNODE being expanded have the value 0, its two children must both have the value 0, or both have the value 1. The count field is maintained in the RNODE only to allow the search algorithm to determine its state of expansion, should backtracking require other possibilites for the RNODE to be developed.

8.2.3 Data Node: FNODE

Should it be desired to employ minimized forms for only the uncomplemented S-box functions, i.e. to characterize input variable configurations which result in a specified output of the S-bank having a value of 1, then the expansion of nodes which embody knowledge concerning the value of the output of the DES f function at some encryption level and bit position would be intrinsically asymmetric, for nodes of differing value. Given that QM-minimized expressions for only the uncomplemented S-box functions are available, forms for the corresponding complemented S-box functions could be produced by the DeMorgan complementation [16] of the positive sum-of-products forms. While the process of complementation itself is trivial, the complexity required in the search procedure to handle the asymmetry introduced by the use of product-of-sum forms is significant.

While an FNODE whose value is captured by a sum-of-products expression may be satisfied by alternatively attempting to satisfy each of the conjunctive terms, an FNODE whose value is expressed in a product-of-sum form has far more potentially-satisfying input variable configurations. To satisfy the latter form, one X literal from each of the disjunctive terms need be satisfied.

Consider as a simplified example the case where an S-box function f has the following minimized form:

f = x1 x2 x3' x4' x5' x6 + x2 x3 x4 x6'

then by DeMorgan complementation:

f' = (x1' + x2' + x3 + x4 + x5 + x6')(x2' + x3' + x4' + x6)

To attempt to satisfy the former uncomplemented form, at most 2 disjunctive possibilities will have to be explored, each of which is a conjunct of a number of X literals. The number of potentially-satisfying literal instantiations for the latter complemented form of the f function may be seen to be a product of the number of literals in disjunction in each conjunctive term, i.e. 6x4=24. Specifically, each line of the following table provides an instantiation of X literals which causes f'=1:

```
x1=0  x2=0
x1=0  x3=0
x1=0  x4=0
x1=0  x6=1
----------
x2=0  x2=0
x2=0  x3=0
x2=0  x4=0
x2=0  x6=1
----------
x3=1  x2=0
  .     .
```

Although it is possible to eliminate certain instantiation configurations a priori, as, for instance, the possibilty that x3=1 (satisfying the first conjunctive term) and x3'=1 (satsifying the second such term) cannot be realized simultaneously, there are still exponentially many more ways to potentially satisfy the DeMorgan complementation of a positive S-box function than there are to satisfy a sum-of-products form. As a consequence of this, despite earlier experimentation with a form of the search procedure which actually complemented the QM-minimized expressions for the positive S-box functions, the complemented S-box functions were themselves minimized by the QM procedure, and these sum-of-products forms for the f' used in the search.

Recall that the sum-of-product expression for an S-box, or its complement, as obtained by the Quine-McCluskey minimization technique, consists of the disjunction of up to 23 p-terms, each of which is a conjunct of up to 6 literals. (The literals constitute the input to the S-box). For an FNODE at some point in the search to have a value of 1, any of the p-terms in the appropriate sum-of-products expression for the uncomplemented S-box function must be "on". Such a term will be "on" only if all literals in the term possess the appropriate values. Similarily, for an FNODE to have the value 0 at some point, any of the p-terms in the appropriate sum-of-products expression for f' must be "on", as a

result of all literals in this p-term possessing appropriate values.

Consequently, a based structure known as an FNODE is used in the search tree to represent knowledge that at some encryption level, some f function must have a specified Boolean output value. The FNODE structure contains an integer count field from {1,23} which indicates the number of the p-term which the search currently assumes is responsible for turning the f function "on". There is a value bit, which indicates whether the FNODE is to represent an S-bank output with a value of 0 or 1. This value determines whether the QM-minimized forms for the complemented or uncomplemented S-box function, respectively, are to be used. Also in the FNODE are 6 pointer fields which contain links to the appropriate XNODEs which possess the values required to turn the p-term on.[16] This expansion paradigm introduces a heuristic component into the search, as the p-terms occur in the sum-of-products expression in best-first order, and this is the order in which they are expanded.

Should backtrack occur to an FNODE, its count field is examined, and the next p-term in the appropriate sum-of-products expression for this function (or its complement, depending upon the value of the value bit within the FNODE) is assumed to be the term responsible for making the f func-

---

[16] If the p-term considered has some "don't care" values in some literal positions, the corresponding pointers in the FNODE will be null.

tion output 1 (or 0). If no p-term in the appropriate sum-of-products expression can have a value of 1 without the consequences resulting in key bit contradictions, backtrack continues past the FNODE to previous nodes in the search tree.

8.2.4 Data Node: XNODE

The structure of the XNODE type is very similar to that of the RNODE, as the values of a particular position of an R-block and of an X variable are both formed as a result of an XOR operation. In this discussion, inputs to the S-boxes are referred to as variables by the name of X. Such XNODEs contain 3 fields.

As for RNODES, there is a count field which contains an integer from {0,2} to record the number of times the XNODE has previously been expanded. There is also a value field to contain a bit indicating whether the X variable is to have the value 0 or 1. The third field contained in an XNODE is a pointer to the RNODE of appropriate level, value, and position which caused the production of the particular X value during encryption.

An X variable at some particular position and level obtains its value during the encryption procedure by means of application of the following DES formula:

$$X = KS(p,n) \oplus Rn-1$$

The first term represents the output of the DES key schedule function at level n, position p. If the value of the XNODE being developed is 1, then the two possible disjunctive cases are that the KS output and the particular position of the R-block were respectively either 1 and 0 or 0 and 1. As in the case of the RNODE, should the value of the XNODE have been 0, both the KS output and R-block bit would have to possess the same value, for the XNODE to be satisfied. (Both 0 or both 1).

Only one child pointer extends from an XNODE, as the requirement that a key bit possess a certain value does not imply any further tree development. If it is known that the output of the KS function must have a certain value in a certain position at a certain level, a key bit hypothesis may be immediately formed and posted in the global variable containing the developing key. The key schedule function is inverted to determine which bit of K is produced in position p at the given round of encryption, and this bit of K is assigned the appropriate value. Should backtrack occur to the XNODE, the key bit hypothesis must be deleted.

8.3 THE PL/I PROCEDURE: SEARCH

The implementation of the search strategy described earlier was carried out in PL/I. The actual code for the routines to be discussed may be seen in Appendix F. Although the strategy employed for the purposes of unidirectional

search for K has already been discussed at some length, certain features of the PL/I implementation are noteworthy. In particular, attention will be paid to the techniques used for node expansion and backtracking during the search.

Tree development is controlled by a queue of nodes which are "open" for further development, where the characteristic common to all such open nodes is that they are not yet satisfied. Their subtrees require further development. Whereas a similar queue maintained for the purpose of breadth-first expansion of the n-ary tree to discover the best sum-of-product terms (as discussed in Chapter VI) was implemented using an array of pointers to open nodes, here a linked list of pointers is maintained. Tree development is accomplished by expanding the subsequent open node on this queue, deleting it from the queue of open nodes, and then moving on to expand the next open node. This procedure constitutes the mainline of the search procedure, and continues until the open queue is empty.

The reason that a linked-list implementation was chosen for the open queue involves the need to be able to choose whether the expansion is to be depth-first or breadth-first. An array implementation may be seen to make the ability to support the former expansion capability prohibitively expensive. As the tree development is accomplished by expanding nodes in the open queue successively, to grow the tree in a breadth-first manner one adds new nodes to the end of

the queue, where they will be expanded after all other nodes in the queue. Adding new nodes to the queue in a position immediately following the node currently being expanded means that these new nodes will be expanded next, before others in the queue, and that the expansion of the tree will occur in a depth-first fashion. In an array implementation of the open queue, the insertion of entries in the middle of the queue would entail the "shuffling" of elements, and a great associated computational expense.

When a node pointed to by an element on this open queue is to be expanded, the EXPAND routine is called, from the mainline. It selects and invokes one of the routines: R_EXPAND, F_EXPAND, or X_EXPAND, depending on the value of the type field of the node being expanded, for R_NODEs, FNODEs, and XNODEs, respectively.

8.3.1 The R EXPAND procedure

The routine R_EXPAND commences by checking the encryption level of the RNODE to be expanded. If it is the case that the RNODE is from encryption round 0 or -1[17] the bottom of the tree has been reached, and the value of the RNODE may be compared with the value required for such a node, as bits in the R block at levels 0 and -1 correspond to bits of the known plaintext. Should the RNODE have the correct value,

[17] No L nodes are explicitly represented in the tree, but their presence is accounted for by RNODEs of the preceeding level. Hence an RNODE at level -1 corresponds to a level 0 L node.

R_EXPAND returns without adding any new nodes to the open queue. However, if the value of this RNODE as produced by the inversion of the encryption process is incorrect, the BACKTRACK procedure is called to remake choices earlier in the tree which led to the production of this erroneous RNODE.

If round 0 of the encryption has not yet been reached, the count field within the RNODE is examined. This field contains a value from {0,2} to indicate how many times the node has already been expanded. There are only 2 possible ways to expand any RNODE, corresponding to the 2 possible inputs to an XOR function which can cause the function to attain a specified value. If the count field is 2, no further expansion possibilities remain for the RNODE, and BACKTRACK is called.

If the count field is other than 2, appropriate left and right child nodes are created and added to the end of the OPEN queue, after any existing subtrees have been destroyed and their ramifications removed. The left child is always another RNODE of level 2 less than the RNODE being expanded.[18] The value of this child RNODE is assigned as 1 upon the first expansion of the father RNODE, and 0 for the subsequent expansion. The right child is an FNODE of value 0 or 1, depending on both the value of the parent RNODE and the number of times which the parent has been expanded. To be

[18] This RNODE represents the virtual LNODE at level 1 less than the RNODE being expanded.

specific, an FNODE of value 1 is the right child if and only if the parent RNODE has a count field which is equivalent to the number of times the RNODE has previously been expanded. (Both 0 or both 1). Otherwise, an FNODE of value 0 is created. Before leaving the R_EXPAND rouitne, the count field of the RNODE being expanded is incremented.

8.3.2 The F_EXPAND procedure

The F_EXPAND routine is invoked by the EXPAND driver to expand an FNODE. As mentioned, all simultaneously disjunctive paths from a particular node in the tree exist only virtually: The FNODE structure at any given time has children which consist of the set of XNODEs engendered by the assumption that a single particular p-term in the sum-of-products expression for the S-box in question will cause the satisfaction of the FNODE.

When it is necessary to expand an FNODE, it is first determined through the examination of the value bit within the FNODE whether the sum-of-products form for the complemented or uncomplemented S-box function is to be employed in the expansion. The procedure then ascertains whether there yet remain any p-terms in the appropriate S-box expression corresponding to the S-bank output under consideration, the consequences of which have not yet been explored. To accomplish this, the count field maintained within the FNODE is compared with the number of p-terms in the associated sum-

of-products expression. This field is incremented each time an expansion of the FNODE is performed. As no Quine-McCluskey minimized S-box expressions contains more than 23 p-terms, this count field has a value from {0,23}. Alternatively stated, satisfaction of any FNODE can be attempted at most 23 times before backtrack continues higher in the tree. If all p-terms have been exhausted, BACKTRACK is called. In such a case, no possibilities remain to expand the FNODE without the consequences of the expansion causing some contradiction.

If untried p-terms still remain in the appropriate sum-of-products expression, children corresponding to the X literals in this next p-term are created and added to the OPEN queue. In practice, the existing children of the FNODE are replaced only if the value of the X literal they represent differs in value from the corresponding X literal in the new p-term. If the values do not differ, the old XNODE child is retained along with the subtree of which it is the root.

8.3.3 The X EXPAND procedure

The expansion of an XNODE is somewhat analogous to that of an RNODE, as both of these structures are formed as a result of an XOR operation. To expand an XNODE, its count field is first examined. If this field has reached 2, BACKTRACK is called, as no possibilities for expansion remain.

XNODES have only one child in the search tree, the other component of the XOR producing an XNODE is a bit of K. If the number of times the XNODE has been expanded is equivalent to its value, as in the expansion of an RNODE, an RNODE of value 0 is created as a child at the preceeding round of encryption, and is added to the OPEN queue. An RNODE with value 0 is created if the equivalence does not occur.

Based on the count field, a hypothesis for the value of the bit of K is produced. The position of the bit within K is determined through knowledge of the key schedule permutation. Beore posting this hypothesis for the bit of K, the current hypotheses are examined to ensure that a contradictory hypothesis for the same bit does not exist. If such a hypothesis already exists, BACKTRACK is called to resolve the contradiction. If no hypothesis yet exists for the bit under consideration, the new hypothesis is posted and X_EXPAND returns, after having incremented the count field within the XNODE.

8.3.4 The BACKTRACK procedure

The BACKTRACK routine has been mentioned extensively in the preceeding discussion, although its characteristics have not yet been examined. It is invoked when no disjunctive alternatives for expansion remain for the node whose expansion is currently being attempted by the tree search procedure. All possible expansions have led to a contradiction, either

of key bit hypothesis, or between what has been produced as plaintext by inversion of the encryption process and what the plaintext is known to be.

If such a condition arises, it must be the case that an erroneous choice of some disjunctive path to follow has occured earlier in the tree, and such a choice must be remade.

BACKTRACK first deletes the current node. This is accomplished by setting to null any pointer in the father of the node which points to that node, as well as removing from the OPEN queue any references to the node.

The need for the latter action may be seen from the following example. Consider a subtree of the search tree, consisting of an RNODE at encryption round 2, and its 2 children: a round 0 RNODE, and an FNODE at level 1. Suppose that the RNODE at level 0 is the node currently being developed and that the FNODE is the next on the OPEN queue. Should the check with the known plaintext indicate that the value of the level 0 RNODE is incorrect, backtrack occurs to re-expand its father, the level 2 RNODE. This re-expansion will cause the creation of new left and right children. In particular, if the level 2 RNODE may be re-expanded, a new FNODE child with a value different from the previous right child is produced. Clearly, the old FNODE must be removed from the OPEN queue to prevent such a node, which now does not belong in the tree, from ever being developed.

It is for this reason that the OPENQ field exists in each SUPER node. This field conatins a pointer to the queue node which points to the SUPER node. Essentially, the OPEN queue references are backlinked to permit the immediate location of a particular queue node for deletion. If such a field were not provided, it would be necessary to search the OPEN queue linearly for any references to the node being deleted, each time any open node was to be removed from the tree. As the OPEN queue may be as long as the maximum width of the search tree, such a procedure would be computationally wasteful. Clearly, the OPENQ field points to nodes on the OPEN queue only for nodes which are still open for further expansion. The field is set to null when a node is expanded by the EXPAND routine. After these references are deleted, BACKTRACK calls EXPAND for the father of the deleted node, to attempt an alternative expansion of this earlier node.

It should be noted that this process is recursive, and will continue the expansion and backtrack until all 64 of the round 16 root nodes are satisfied, and all positions of the results of our encryption inversion agree in value with what the plaintext P is known to be, with no contradictions in what the bits of K must have been. At such a point, the encryption key K has been uncovered.

8.4 APPLICATION TO A 2-ROUND DES

For the purposes of testing these unidirectional key search procedures, a set of APL routines were written to perform 2-round DES encryption of randomly-chosen plaintext bits under a randomly-chosen key, and store the resulting P-C pairs in files accessable to the search routines. These APL routines may be found in Appendix G.

Experimentation with the PL/1 search routines quickly demonstrated the intractability of a unidirectional search approach to the discovery of the encryption key, even for a DES algorithm of only 2 rounds. Computer time limits of up to 30 minutes on the University of Manitoba AMDAHL 470/V8 were exceeded during the course of execution of the search, and further experimentation with such searches had to be curtailed due to the computational expense. The search tree itself was constructed down to the plaintext leaf level fairly rapidly, but the process of backtracking to remake choices in the tree to get the known plaintext to agree with what had been generated as a result of the posted key bit hypotheses continued in all trials until the processing time limit imposed upon the program had been exceeded.

Examination of the causes of the failure of this approach to cryptanalysis led to a direction of further research differing in two components: An analysis of the use of processing time by the search procedure showed that a considerable amount of time was wasted by both the dynamic allocation and

freeing of storage from the heap by PL/1 during execution, and by the continual paging of (4k byte) segments of the large (384k) search tree during the search. While the former problem could have been avoided through exploitation of the realization that the tree, in a somewhat altered representation space, is static and thus all required memory may be pre-allocated and need never be freed, the latter seems regrettably inavoidable due to the scattered distribution of key bit use throughout the encryption. The regularity of structure of the properly-viewed search tree will be further discussed in the following chapter.

Aside from these implementation considerations, upon proper reflection it must be concluded that the unidirectional search approach is destined to be computationally intractable, as it is essentially a "generate and test" approach. In the algorithm presented, not all of the available data is effectively used to guide the search. While the ciphertext is employed to determine the growth of the tree, the known plaintext is used in an inefficient manner, to simply verify that the tree has grown properly, and to cause backtrack for the purposes of correction, if it has not.

Algorithms of this type are known in the field of computer science as the "British Museum" approach for their exhaustive and somewhat foolhardy nature. Our unidirectional approach is analogous to employing exhaustive bottom-up generation of all possible facts in order to prove a theorem,

instead of using the goal, i.e. the theorem to be proven, to guide the search in a top-down goal-directed fashion.

What seems required is an ability to effectively use all available knowledge, both P and C at once, to guide the key search and thus effectively limit the search time. The next chapter discusses such a bidirectional approach from a conceptual viewpoint as a problem of search, and then identifies the problem of the search of a tree of fixed structure with that of the symbolic solution to a set of Boolean equations. After an unsuccessful attempt to program methods to symbolically simplify the Boolean equations which constrain the values of the bits of K which maps between a given P-C pair, a new type of AND/OR tree search is developed. This new search benefits from all of the minimizations performed on the S-box functions, and has the additional advantage of not requiring any backtrack.

Chapter IX

KEY SEARCHES OF GREATER SOPHISTICATION

9.1 COMPUTATIONAL COMPLEXITY AND BIDIRECTIONAL SEARCH

Certain properties of the DES encryption procedure make it vulnerable to an attack by the methods of bidirectional search, under the assumption of a known plaintext attack. It has been mentioned that the knowledge possessed by the cryptanalyst under this assumption is situated at both the root and the leaves of the cryptanalytic search tree.

The unidirectional search of the previous chapter attempted to invert the encryption process through the construction of a search tree beginning at the (known) bits of ciphertext. Hypotheses concerning the values of key bits and bits in earlier blocks of developing ciphertext were forwarded, based on knowledge of the values of bits in blocks formed later in encryption. The known plaintext bits were reached after all levels of encryption had been inverted. Backtrack to attempt alternative disjunctive paths in tree development occured when either plaintext bits did not agree in value with what the bit values should have been on the basis of how the encryption was inverted, or when required assignments of key bit values were contradictory.

In addition, during this search, a single key hypothesis was maintained in a variable globally available for all subtree expansions to update. In retrospect, the "thrashing" behavior of the unidirectional search may be seen to result at least partially from the design of the search algorithm which required such a unique hypothesis to be maintained. In the AND/OR search which is discussed in section 9.4, this requirement is removed to allow the search to proceed by building an expression tree top-down, and then traverse the tree in a bottom-up fashion, without the need for any backtracking.

It is possible to obtain an estimate of the efficacy of this search procedure by comparing the worst case number of trials in an exhaustive key trial approach to DES cryptanalysis (2^{56} or on the order of 10^{17}) to the greatest possible number of nodes which would have to be developed in the tree during the search approach to encryption key discovery. This comparison is not altogether reasonable, as exhaustive key trials would probably be performed in special-purpose hardware [2], while the search techniques discussed would at first be implemented in software. However, it would not be impossible to build a hardware device to perform the operations involved in the search procedure. As a 16-round DES encryption requires far more basic operations than the expansion and traversal of a single search tree node, the identification of these heterogeneous quantities errs to-

wards the conservative. It is crucial to the success of the attack methods presented here that one accept this idea that it is reasonable to compare key space size with number of tree nodes.

A simple argument will quickly demonstrate that the use of a unidirectional search for cryptanalysis as attempted in the preceeding chapter is destined to failure, even apart from considerations of backtracking. (I.e., even if each node in the tree is visited only once, a unidirectional search will fail). For such a search to be useful, the required branching factor engendered by the S-boxes is so small so as to be unattainable by means of known Boolean minimization techniques.

The top level of the search tree contains 64 nodes, one for each bit of ciphertext. In proceeding from one level of the search tree to the next, each node which represents a bit in a block of developing ciphertext can require the inversion of <u>two</u> instances of the function f. If some bit position of some R-block has a value of 1, then the values of the same bit position of the f function which operates on the R block of the preceeding level and that of the L block at the preceeding level must differ, as an XOR indicates non-equivalence. (See Figure 6, position (2)). would be investigated, if the search ever backtracked to the node. As the (Quine McCluskey) representation for any S-box is a p-term expression of at most 23 p-terms, each of which is a

conjunct of at most 6 literals, the expansion of an FNODE once in the search, can engender a branching factor of up to 23x6 in the search tree. As two such expansions may be required per node per level of the search tree (if the first expansion fails), the worst case number of nodes in the unidirectional search tree using QM S-box representation for the inversion of a 16-round DES will be approximately:

$$64 \times \sum_{i=1}^{15} (2 \times 23 \times 6)^{i} = 2.6 \times 10^{38} >> 10^{17}$$

It may also be calculated that for the number of nodes in the search tree to be less than the number of possible keys, again under the highly conservative assumption that all possible disjunctive paths in the search tree must be followed, the branching factor caused by the S-boxes must be less than 5. Even the spectral minimization techniques do not allow such a small expansion factor to be achieved.

Nevertheless, the use of bidirectional search techniques [18,19] may be shown to upper-bound the number of nodes in the search tree, even under worst case assumptions, to a number which is on the order of the key space size. As encryption and decryption procedures are virtually identical in the DES algorithm, it is then possible to expand the tree "backwards" from the known plaintext towards the ciphertext, at the same time as one is expanding the tree forwards, from the ciphertext. Key bit hypotheses are generated during the

course of the expansion in both directions, and standard recursive backtrack occurs within either search (or within both searches) if mutually incompatible key bit hypotheses are generated either within one "half" of the search tree, or if some incompatible hypotheses exist when the union is taken of the two generated sets of hypotheses. The matching of the two halves of the tree between rounds 7 and 8 is not difficult. However, it does imply that to implement the bidirectional search for a full 16-round DES, enough memory capacity is avaliable to store the entirety of the middle (widest) layer of the tree. This layer contains $64(2x13x5)^7$ nodes.

It is not difficult to show how the use of such a bidirectional search reduces the number of nodes in the tree. What the approach achieves may be seen schematically in Figure 8, Bidirectional Search Tree. The search is driven forwards from both the top and bottom simultaneously, and meets in the middle of the tree. It is only possible to do this because the encryption and decryption procedures are the same, and the attack assumptions include knowledge of the plaintext.

The search now consists of two symmetric searches, each of which continues for 7 levels beyond the first level at which the 64 nodes representing the known bits of plain or cipher text are established. A total of 8 levels are searched in each half of the search. Using the same QM S-

box approximations as seen in the preceeding calculation, the worst case number of nodes in the bidirectional search tree is found to be:

$$2 \times 64 \times \sum_{i=1}^{7} (2 \times 23 \times 6)^{i} = 1.6 \times 10^{19}$$

Should this search employ the spectrally-minimized S-box representations, even in the worst case the total number of nodes expanded in the search tree would be on the order of the key space size:

$$2 \times 64 \times \sum_{i=1}^{7} (2 \times 13 \times 5)^{i} = 8.9 \times 10^{16}$$

It is important to realize that all of the above rough calculations have assumed worst case conditions for the search. That is, it has been assumed that all possible disjunctive alternatives of all OR nodes must be expanded in the course of the search, and furthermore that all such expansions entail in all places the maximum number of branches. In fact, only one S-box has as many as 13 p-terms in its spectrally minimized form. The mean number of p-term is about 11. In addition, the fact that p-terms are ordered in the sum-of-products expressions in a "best first" manner lends a heuristic component to the search; the alternatives

most likely to succeed[19] are chosen for expansion first.

It may be possible to quantitatively estimate the advantages of the bidirectional search technique by slightly relaxing the worst case assumptions. If the expected branching factor of the search tree is based upon the average number of literals per p-term (3.57) instead of the maximum possible number of such literals (5), the expected number of nodes in the search tree decreases to 7.68×10^{15}. This is a factor of 10 less than the key space size. The unidirectional search of Chapter 8 was never actually implemented in the bidirectional manner described so far in this section. A realization that the structure of the search tree would also be uniform, regardless of the particular P-C pair cryptanalyzed, led to the digression of the next section, in which the cryptanalytic problem is shown to be equivalent to the simplification of a set of Boolean equations. While experimentation with this technique was limited by the its requirement for very large amounts of memory, this next view of the cryptanalytic problem was of use, in that it eventually led to the development of a more sophisticated type of search, in which backtracking is unnecessary.

---

[19] This is somewhat simpleminded, as this considers only the p-terms which "turn on" the S-box functions by themselves, and ignores interactions with other branches in the tree.

9.2 DIGRESSION: SEARCH AS THE SOLUTION OF BOOLEAN EQUATIONS

During the course of development of the unidirectional search procedures, it was realized that should it be desired to invert the action of the DES encryption in making the output of some f function 0 at some round and position, to expand the product-of-sums expressions resulting from DeMorgan inversion of the QM-minimized S-box functions would make the search intractable. As a consequence, complements of the S-boxes were minimized in order to possess sum-of-products expressions for use in such circumstances.[20]

What was only realized later was that the use of such minimizations of the complements of the S-boxes during the inversion of encryption implied that the structure of the resulting search tree was always the same, independent of what P-C pair was used.[21] RNODEs always have another RNODE (of level 2 less than their father) as a left child, and an FNODE as a right child. FNODEs have up to 6 XNODE children, depending on the number of "don't care" positions in the conjunct term currently being expanded. Each XNODE has an RNODE as an only child. Expansion of an XNODE also causes the posting of a key bit hypothesis as a side-effect. (See

[20] Quite clearly, a sum-of-products expression of the general form arising for DES may be potentially satisfied in far fewer ways than the corresponding product-of-sums expression resulting from its complementation, due to the presence of more disjuncts in the latter.

[21] This is not entirely accurate. An FNODE in the final state of the tree may have less than 6 XNODE children, if the corresponding p-term contains "don't-care" values.

Figure 9, 2-Round Search Tree of Uniform Structure).

As the morphology of the tree would thus be constant, regardless of the actual P-C pair for which K was being discovered, there was no longer any need to allocate subtrees dynamically through the use of the PL/I ALLOC and FREE functions. The entire tree structure could be pre-allocated, and the fields in its nodes simply filled in to reflect the tree contents at any moment. In addition to the resultant time saving, memory space could also be saved as a result of this realization, as pointers are only really needed to refer to a child when the location of the child cannot be determined when the father is created. With the entire tree structure known a priori, pointer locations could be determined by address calculation, and not explicitly stored with the nodes.

An even more important revelation which followed from this discovery is that when the form of the tree is predetermined, the entire tree structure can be collapsed into a set of equations, where the key bits are the unknowns, and the plain and ciphertext bits are constants. In such a representation, the search tree would be present only virtually, implicit in the expression-tree structure of the equations. The problem of bidirectional key search is isomorphic to that of solving for the key bits in the equations implied by the search tree structure.

Furthermore, the process of cryptanalysis by such a method could be partitioned neatly into a (lengthy) symbolic

precomputation procedure, followed by a (fast) application of the results of this precomputation for the cryptanalysis of specific P-C pairs. In the precomputation phase, the constraint equations which contain the key bit variables and the constants for P and C as symbolic constants would be simplified through the application of algebraic transformations. Such a computation would only ever have to be done once. After these symbolically-simplified equations are produced, to discover K for any specific P-C pair, one would substitute the known values for P and C for the symbolic constants, then simplify the equations further, to discover the actual values for the bits of K.

The idea of such a separation of components of a cryptanalytic process has been proposed elsewhere [6]. This method of cryptanalysis would also benefit from all of the search space size reductions which arise from the functional and spectral S-box minimizations discussed in Chapters V and VII.

Specifically, the approach to be considered here is as follows: From the details of the DES encryption algorithm together with the minimized representations for the S-boxes, a set of 64 equations involving ORs, ANDs, and NOTs with the 56 key bits as unknowns and the 64 positions of P and 64 positions of C as symbolic constants may be formulated. It should be observed that for an n-round DES, these equations have 2n+1 AND and OR levels, where the number of levels

possessed by a Boolean equation may be defined as the number of levels in the correponding n-ary[22] expression tree.

The symbolic simplification of this set of 64 equations, one for each bit of C, by means of the application of theorems of Boolean algebra or some other similar simplification technique would constitute the precomputation phase of the cryptanalysis.

Simplification, in this context, involves the "flattening" of the implicit expression tree into a 2-level sum-of-products form, with the removal of redundant terms, and will be discussed more thoroughly in the following section. The fashion in which this simplification should be carried out in order that it not require undue amounts of memory or processing time is not at all evident. One of the central problems which must be addressed is that the tradeoff which exists between the advantages of moving negation "inwards" in the expressions, and those of applying theorems (such as the absorption or consensus theorems [16]) to reduce the number of terms in the expression.

If negation is moved inwards by application of the DeMorgan theorems in a careless manner, the fact that a complex subexpression and its complement exist together in conjunc-

---

[22] OR and AND operators will not be restricted to 2 arguments, but may operate on n arguments. This convention will permit an expression a+b+c to be represented in one expression tree level, as "+" operating on 3 arguments. Were we forced to consider "+" as a binary predicate, 2 tree levels would be required, to represent the expression as a+(b+c) or (a+b)+c.

tion or disjunction may be overlooked, a situation which wastes memory space, and eventually processing time as well. However, the detection of instances in which theorems such as the absorption theorem are applicable seems to require complex and computationally expensive pattern-matching procedures. In what sequence to apply these simplification techniques is not evident.

It has been mentioned that the preprocessing phase of the cryptanalysis consists of the symbolic simplification of the set of 64 equations relating the variables in K and the constants in the known P to each of the 64 bits of C, respectively. After these equations have been simplified, to discover K for any known P-C pair, the Boolean constants 1 and 0 are substituted in the equations for the symbolic P and C constants, and each of the 64 equations is further simplified as much as possible.

Finally, as the 64 equations themselves must necessarily be satisfiable together, they are put into conjunction, and this conjunct further simplified, by the same simplification process. What must result, for a 16-round DES, is the unique conjunction of key bits and complements of key bits which equals 1.[23] This yields the encryption key, K.

---

[23] For a DES of less than 16 rounds, possibly a disjunction of such conjunctions may result. This corresponds to the situation where more than one key performs a specific P->C mapping in a 2-round DES. It is unknown if this can occur in a full 16-round DES.

9.3 SYMBOLIC SIMPLIFICATION METHODS

A number of potential methods were examined, for the simplification of the sets of Boolean equations which are seen to represent in a new form the cryptanalytic search tree of the preceeding chapter. As it has been realized that the expression tree is of fixed structure, it is possible to calculate the size of such a tree for an n-round DES. The number of leaf nodes in such a tree is the same as the number of literal terms in the (unsimplified) expression it represents, and will vary as a function of the quality of the S-box minimization employed.

9.3.1 Expression Size

It is of some interest to determine the theoretical maximum size of the 64 expressions which result from the flattening of the search tree, as their size will determine the applicability of various simplification techniques. The number of literals which will be present in any expression is identical to the number of leaf nodes in the corresponding expression tree, as any Boolean expression is simply an expression tree with the structure obscured.

With reference to Figure 9, the number of literals in unsimplified expressions for a 2-round DES may be seen to be:

$2(1+ (23)(6)(2(1+ (23)(6)))) = 148326$

for each of the 32 subtrees from the leftmost 32 bits of C, using the conventionally QM-minimized S-boxes, and:

$2(1+(23)(6)) = 396$

for the 32 subtrees from the rightmost 32 bits of C.

9.3.2 Problems of Simplification

In view of the large size of the equations involved in the problem, the simplification methods to be employed must be carefully chosen and implemented to be compuatationally tractable, even for a 2-round simulation.

Immediately, a Quine-McCluskey simplification may be seen to be inappropriate. The QM method requires that all prime implicants for the function be generated, and the resultant combinatorial explosion renders this impossible in our 56-variable case. For QM to be used, the expressions to be simplified must first be multiplied out into a 2-level sum-of-products form prior to simplification. While such a form is the ultimate goal form for our equations, a feasible algorithm should not expand any expression prior to exhausting all possibilities to reduce its size, due to the already unwieldy size of these expressions.

Very little information exists in the literature concerning the automated symbolic simplification of Boolean expressions. A manual application of the theorems of Boolean algebra to an expression will theoretically result in its reduction to some form of maximal simplicity, if these mathematical operations are carried out in the correct order. One problem with the automation of such a procedure is that the correct order in which to apply the algebraic theorems is often not at all clear.

The only attempt known to the author at an algorithmic specification of how such a simplification might proceed is that of Zissos [25]. Zissos presents a somewhat vague "research algorithm" for the symbolic minimization of Boolean expressions. This algorithm, while not producing a minimal form in all circumstances, has the advantage that it does not entail expansion of the original expression. Its disadvantage is that the algorithm presented is somewhat obtuse, and does not seem well suited to computerized implementation.

In order to carry out the required Boolean simplifications for the reduction of the equations embodied in the search tree, an admittedly ad hoc system of PROLOG (PROgramming in LOGic) routines was devised and implemented under UNIX on a DEC PDP 11-45 minicomputer.

PROLOG is a very high-level theorem proving language, with capabilites of automatic recursive backtracking and pattern-directed procedure invocation. In PROLOG, control is decoupled from the logic of a program, is built-in to the PROLOG interpreter, and so need not be explicitly specified. A user of PROLOG simply provides truths to the interpreter in the form of Horn clause[24] axioms and implications, in a notation similar to that of first-order logic [23]. The theorem whose proof is requested is specified in a similar form, perhaps with some unbound variables, the values of

[24] A Horn clause is a conjunct of logical predicates which involve no negations.

which are determined in the course of satisfaction of the theorem.

The linear-input resolution theorem-proving methods of Robinson [21] are then used by PROLOG to affirm the theorem by the syntactic manipulation of the facts to derive what is known as the "empty clause." A more detailed description of the operation of the PROLOG interpreter would serve little purpose here, and the reader is directed to the literature on the subject [23,24].

9.3.3 A PROLOG Symbolic Simplifier

For the purposes of exploring the possibility of cryptanalysis by means of solving the Boolean equations relating P,C,and K for the unknowns K, a set of PROLOG axioms and implications were developed which were capable of simplifying arbitrary multi-level Boolean expressions involving ANDs, ORs, and NOTs into a 2-level sum-of-products form. Appendix H lists these routines, and a small example of their application appears on page 239.

The simplification system relies heavily on the pattern-matching capabilities built-in to the PROLOG interpreter to determine the applicability of various simplification theorems. The feature of PROLOG which permits the user to define new operators was employed to permit the PROLOG interpreter to parse well-formed Boolean formulae which contain tildes (for negation), ampersands (for conjunction) and backslashes (for disjunction).

To simplify an expression, the predicate "simplify" iteratively calls the "simp" predicate, until no further simplifying transformations may be effected. The "simp" predicate attempts to apply some simplifying transformation directly to the expression. If this is not possible, it recusively attempts to do the same for subexpressions of the original expression, until the literals of the expression are reached.

These simplifying transformations are a PROLOG encapsulation of the pertinent rules of Boolean Algebra, and are represented by the "s" and "s2" predicates which are activated by the "simp" predicate. The first two "s" predicates represent the trivial case of the recursive simplification of expressions, a bottoming-out on literal atoms which can be simplified no further.

If these are not satisfied, the next "s" predicate checks whether the expression to be simplified is a one-level expression, i.e. whether it consists of a conjunct of disjuncts of literals. This check was added late in the development of the simplification system, as a measure to prevent the excessive use of stack space by the PROLOG interpreter in recursing down to the atomic level for all formulae. The literals in the expression are sorted alphabetically by a quicksort, and scanned to effect the transformations:

a&¬a -> 0 a&a -> a a|a -> a a|¬a -> 1

Should the expression not be of a single level, the next "s" predicate checks for the applicability of the involution law, and the next two attempt to apply the 2 DeMorgan laws to move negation inwards. The last "s" predicate activates the "s2" transformation predicates to match, 2-at-a-time, the top level terms in the expression to be simplified against the forms of the remaining simplification theorems. These "s2" simplification predicates include the theorems of: idempotency, complementarity, distributivity ((a|b)&(a|c)->a|(b&c)), absorption, consensus, and the other distributivity theorem (a&(b|c)->a&b | a&c). The system attempts to apply this latter distributivity theorem to multiply conjuncts only if all other simplification theorems fail to be applicable, as it is this last theorem which can make expressions larger.

It is unfortunate that despite their sophisticated nature, these simplification predicates failed to be of much use in the cryptanalysis of DES. The fact that the PROLOG interpreter tends to be very inefficient in its utilization of memory space restricted the application of this elegant simplification system to problems of a small "toy" nature. Despite careful use of the PROLOG cut operator (!) to remove choice points to which backtrack should never return and thereby save stack space, the very small core memory space available on the PDP 11-45 upon which the system was implemented (approximately 256k bytes) made it impossible to ap-

ply the system to equations whose size was even a factor of 100 smaller than those involved in a 2-round DES.

As a consequence of this regrettable fact, the PROLOG system was abandoned in favour of a more sophisticated search technique, implemented in a conventional language. The idea of the formulation of Boolean equations which constrain the values which the bits of K are free to assume, as presented above, is central to this new search technique.

9.4 A MODIFIED, KNOWLEDGE-INTENSIVE KEY SEARCH

Another version of a key-search procedure was developed to once again attempt to empirically demonstrate the feasibility of the attack method which involves the S-box minimizations which have been discussed. This approach utilized some of the features of the unidirectional search of the preceeding chapter, combined with a better use of the available knowledge concerning P and C, to limit the search tree size, and thereby decrease the time required for such a search to within the limit of computer time available. A listing of the routines discussed in this section may be found in Appendix I.

Viewed from a high level of abstraction, the decryption procedure involves two phases. An n-ary Boolean expression tree containing AND and OR nodes is consrtucted for each of the 64 bits of C, to represent the Boolean algebraic combination of bits of K required to produce the known value for

each of the bits. This tree construction is performed in a top-down fashion from knowledge of value of the bit of C being considered, the particular known plaintext block P, and the details of the DES encryption algorithm.

The expression tree is then evaluated bottom-up, where in the course of the evaluation the leaves contain the restriction on K currently required, in a sum-of-products form represented as a matrix containing values '1', '0', and 'X'. Each row of such a matrix corresponds to a single p-term, in which a '1' corresponds to the presence of an uncomplemented variable, '0' to the presence of a complemented variable, and an 'X' to a "don't care", or the absence of a variable. This representation has been discussed earlier in section 5.1, and will henceforth be referred to as cube notation. ORs are evaulated by performing a specific type of "union" of the key restrictions, while ANDs are evaluated by an "intersection".

The single sum-of-products expression which results from such a traversal of the expression tree represents the disjunctive alternatives which constrain the values of the bits of K which perform the required P->C mapping under the DES algorithm. As mentioned in section 9.2, the final sum-of-products expression will consist of a single p-term if and only if a K which maps P->C for the given P-C pair analyzed is unique. With a DES of only 2 encryption rounds, it is possible that many different K could perform the required mapping.

9.4.1 AND/OR Expression Tree Formation

Given the values of the bits in a plaintext block P, and the value of a particular bit in some known position of C, it is possible to construct an AND/OR expression tree at the leaves of which reside the necessary constraints on a key K so that K:P->C. If 64 such expression trees are constructed, one for each bit position in C, and the key constraints implied by each are ANDed, the key K required to map the 64 bit quantity P to the 64 bit quantity C is uncovered. A description of the technique whereby each of these 64 trees may be produced through knowledge of the details of the encryption process follows.

Consider some bit position of the ciphertext, such as the last bit, 64. Assuming our usual model of a 2-round DES with no IP or IP^{-1} permutations, and QM-minimized representation of the S-boxes and their complements, this is the 32nd bit of the R block at encryption round 2, which we shall denote $R2^{32}$. Using the DES equation relating R at some round n to L and R at round n-1, we see that:

$$\begin{aligned} R2^{32} &= L1^{32} \oplus f^{32}(R1,K2) \\ &= R0^{32} \oplus f^{32}(R1,K2) \end{aligned} \qquad \text{-----(1)}$$

by the other DES equation, where f^{32} denotes the 32nd bit of output from the f function as described in Figure 2 of Fips Publication 46 [26]. Rearranging equation (1) yields:

$$f^{32}(R1,K2) = R2^{32} \oplus R0^{32} \qquad \text{-----(2)}$$

where the right hand side of this equation is known, as $R0^{32}$ is just the 32nd bit of the R block of the plaintext P.

To determine which inputs X to the bank of S-boxes fix the output of the f function, the inverse of the DES P permutation[25] and the structure of the bank of S-boxes must be considered. Bit position 32 mapped through the inverse of the P permutation yields 25, which indicates that output 25 of the bank of S-boxes is involved. This is the 1st output of S-box 7. The value of this S-box function may be seen to be controlled by inputs X^{36} to X^{41} of the bank of S-boxes.

The (known) bit value 1 or 0 of the right hand side of equation (2) determines whether the minimal representation for S-box 7 output 1 of the representation for its complement should be employed, respectively. Whichever should be used, the first 2 levels (OR, then AND) of the AND/OR tree to be constructed can now be established. For the purposes of this illustrative example, assume that the right hand side of equation (2) has the value 1. The QM-minimized representation of S-box 7 output 1 (uncomplemented) begins (in cube notation):

```
010100
011101
111100
X10000
   .
   .
   .
```

This implies that:

$X2^{36}{}'X2^{37}X2^{38}{}'X2^{39}X2^{40}{}'X2^{41}{}'$ +

---

[25] Do not confuse the permutation P of the outputs of the S-boxes with the 64 bit block of plaintext P. When P the permutation is meant, the word "permutation" will always be specified.

$$X2^{36}{}'X2^{37}X2^{38}X2^{39}X2^{40}{}'X2^{41} \quad +$$
$$X2^{36}X2^{37}X2^{38}X2^{39}X2^{40}{}'X2^{41}{}' \quad +$$
$$X2^{37}X2^{38}{}'X2^{39}{}'X2^{40}{}'X2^{41}{}' + \ldots + \ldots \quad \text{-----(3)}$$

This expression may be represented as an AND/OR tree of 2 levels, with the X literals currently at the leaves of the tree. (Figure 10 (b)).

Examination of DES to ascertain how the value of an input to the S-boxes, such as $X2^{36}{}'$ is determined , in order to further expand the X variables currently at the leaves of the developing tree reveals that such Xs are the result of the XOR of a particular bit of R1 with a bit of K. Application of the inverse of the DES E permutation shows that $R1^{21}$ is XORed with the bit of K produced by the key schedule generator in round 2, position 36 (which happens to be K^{7}), to produce $X2^{36}{}'$, i.e. key bit 7. Thus, $X2^{36}{}'=1$ implies:

$$(R1^{21} \oplus K^{7})' = 1$$
$$\text{or} \quad R1^{21}K^{7} + R1^{21}{}'K^{7}{}' = 1 \qquad \text{-----(4)}$$

Equation (4) allows the construction of another 2 levels of the AND/OR expression tree. The $X2^{36}{}'$ variable currently occupying a leaf of the developing tree is replaced by the OR of two ANDs, with $R1^{21}$ and $R1^{21}{}'$ as new variables now at the leaves of the tree, and K^{7} as a key bit constraint at what will remain a leaf of the tree (Figure 10 (c)).

$R1^{21}$ and the other R1 variables which appear at the leaves of the tree as a result of expanding other X variables are then themselves expanded by the same method as was used to expand the original R2 variable.

This process of tree expansion terminates after all X1 variables, i.e. representations of inputs to the S-boxes at encryption round 1, are expanded. To clearly perceive why this is so, suppose $X1^{36}$ is at a leaf of the expanding AND/OR tree. As $X1^{36}=1$ implies:

$$R0^{21} \oplus K^{7} = 1$$

and as the values of all positions of R0 are known (R0 is simply the right half of the block of plaintext P), the value of K^{7} is thus fixed, and the variable $X1^{36}$ may be replaced by the key bit constraint which fixes forever $K^{7}=0$. Through the application of such an expansion procedure, all leaves of the AND/OR search tree are eventually made to contain key bit constraints, and a tree as in Figure 10 (d) is formed.

9.4.1.1 Implementation of the Tree Formation Algorithm

The above procedure for producing the AND/OR expression tree corresponding to a particular bit of C was implemented as a recursive APL procedure, BUILDSUB. (Appendix I, page 240). As Figure 10(d) illustrates, the search tree for a bit of C encrypted through 2 DES rounds has only 7 levels, and so should be built in a depth-first, as opposed to a breadth-first fashion, as such a tree may have a branching factor as high as 23 in some places.[26]

---

[26] Recall that S-box approximations have up to 23 disjunctive p-terms.

In order to maximize the speed of execution, the recursive tree building routine was written as a single procedure instead of as a set of mutually-recursive modules. The BUILDSUB routine is passed a character type code to indicate what structure is being expanded, as well as the round of encryption at which the structure occurs (2 or 1, for our simplified DES), and the position in the parallel vectors representing the tree at which the structure is to be placed. Upon entry into the routine, a branch is taken to the section of the program corresponding to the type of structure being expanded: output from the S-bank, input to an S-box, etc.

APL was chosen as the language for these tree routines, as earlier experience had demonstrated that the capacity to perform interactive debugging of programs which deal with large and complex tree structures was invaluable. The fact that APL lacks a built-in capacity for the indirect referencing of memory locations (pointer types) did not cause any problems in the implementation of the routines to handle the AND/OR tree construction and traversal, as these algorithms require no subsequent modifications to the initial structure of these trees.

The tree nodes are represented in APL across corresponding fields of a set of parallel vectors. When BUILDSUB creates a tree in a depth-first top-down recursive manner, gaps are left in the required places in the vectors, to be filled in, later in the recursion.

For example, suppose an AND node is to be created, to point to 6 children corresponding to non-don't-care positions in some p-term. The next free slot in the set of parallel vectors is located, and the global "free" pointer for these vectors incremented by 6 positions. Depth-first recursion then occurs to expand the first of the non-don't-care literals in what was formerly the first free slot. Eventually, when this recursion returns, a loop continues, to expand the second literal in the slot immediately after what was originally the free slot. The automatic stacking of local variables permits the former "free" location to be retained, while the depth-first recursion is building parts of the tree at lower levels.

Although the above description indicates that the entire AND/OR tree is built in memory prior to its traversal, there is no reason why the tree could not be traversed depth-first, while it is being built. Such an implementation, while conceptually more complex than the method presented above, would have the advantage of requiring far less memory space.

The results of executing BUILDSUB to build the AND/OR expression tree which captures the required key bit constraints on bit 1 of the ciphertext C which results from encrypting a plaintext block which consisted entirely of 0-bits under a key of 56 0-bits is somewhat remarkable. The routine required just over 4 minutes of CPU time on an Am-

dahl 470/V8 using the IBM program IKJETF01 for batch APL execution, and produced an AND/OR tree containing 20082 nodes, 16382 of which were leaves (which contain single key bit hypotheses). Although no statistical analysis of the situation was attempted, due to the extremely complex non-uniform discrete distribution nature of the problem, the size of this tree was significantly less than the maximum possible size of 148326 nodes (section 9.3.1) engendered by the worst-case branching factor of 23. Further trials of search tree formation has indicated that an AND/OR tree size on the order of 20000 nodes is typical for a 2-round DES, when using Quine-McCluskey-minimized S-boxes. This means that in practice, search tree sizes on the order of one seventh of the theoretical worst-case maximum size may be expected. If it is granted that key trial effort is comparable to that needed for the expansion and traversal of a tree node, the cryptanalytic method of tree search presented here is faster than exhaustive key search.

The processing time requirements for this tree construction rendered it impossible to construct trees for each of the 64 bits of C, using these APL routines. PL/I procedures are currently being devised, for these purposes.

9.4.2 AND/OR Expression Tree Traversal

Once a tree as in Figure 10 (d) has been constructed, its recursive traversal to simplify the embodied constraint expression on K is a trivial matter. As the tree may be seen to possess only 7 levels for a 2-round DES (although it is quite wide), a standard depth-first recursive traversal for the purposes of evaluating the key constraints described in the tree is computationally tractable. As the tree is evaluated, key constraints expressed in cube notation must be ORed and ANDed together. Efficient algorithms for performing such logical operations on 2-level sum-of-product forms were developed, and are presented in the following sections.

The APL routine TRAVERSE constitutes a conventional implementation of the technique of depth-first tree traversal. The routine is passed a single argument: the location in the parallel vectors at which the node to be expanded is located. If the node represents a key bit constraint, i.e. the node is a leaf of the tree, the constraint is returned as the result. Otherwise, if the node is an AND or OR node, recursion occurs to apply the required logical operation to all children of the node.

An example of the traversal of a small (63 node) AND/OR tree (printed out by the PP function) may be seen at the end of Appendix I, page 249.

9.4.2.1 OR-merging of Sum-of-Products Expressions

An OR-merging of a pair of Boolean expressions A and B in sum-of-products form is performed by considering all p-terms in either expression as a single set, and removing from this set those p-terms which are redundantly represented.[27] A p-term is said to be redundant if removing it from the cover C of the switching function in which it is a p-term does not alter C's status as a cover of the function. The algorithm assumes that no p-terms in either A or B are redundant to begin with. Specifically, within either of the expressions to be merged, no 2 p-terms exist which differ only by one having X(s) (don't care(s)) where the other has 1s or 0s. For example, it would be impossible for p-terms X01 and 101 to both exist in one expression; the latter is redundant, as b'c+ab'c = b'c.

When the 2 sets of p-terms are unioned to form the OR of expressions A and B, redundant p-terms must be removed from the union, to make the resulting sum-of-products expression minimal. To accomplish this, a special representation of the p-terms was discovered, and employed together with the algorithm of Mott [15] for discovering consensus terms. We call some p-term c the consensus term for terms a and b if c->a|b.

---

[27] A special case for "don't cares" also requires the modification of some p-terms which are retained during the merge.

Mott's algorithm for consensus terms may be employed to OR two Boolean expressions in the following manner. The set of all p-terms in either expression is a (non-minimal) expression for the required OR. Form all possible consensus terms obtainable from pairs of p-terms in either function, and replace one or both of these p-terms with any consensus term which covers it/them. This process of attempting to form consensus terms is repeated for all pairs of p-terms selected such that one p-term comes from each of the 2 expressions to be ORed, and continues until no more consensus terms can be formed.

By using a representation for p-terms different from the cube notation discussed earlier, both processing time and memory space may be conserved. Since any variable in a p-term is in one of 3 different states ('1','0', or 'X'), it is wasteful to use a character representation (1 byte) to store such variables. Two bits suffice to distinguish between 4 states, and if the assignment of bit configurations to such variable states is done cleverly, substantial computational time savings may be achieved when ORing and ANDing expressions in this notation. Such time savings result from the ability of a computer based on a k bit wide processor to OR or AND together 2 strings of k bits in a single processor cycle.

Let '0' be represented by the pair of bits (0,1), '1' by (1,0), and 'X' by (0,0). Any p-term of k variables may then

be represented as 2 parallel bit strings of length k: n1 to represent the bits occuring first in each pair, and n2 to represent the bits occuring second. As an example of this scheme, the p-term x='010X1' may be represented by 2 bit strings of length 5, x n1=01001 and x n2=10100.

To form the OR of expressions A and B, compare each p-term ai in A with each p-term bi in B, where each these p-terms is represented as a pair of bit strings n1 and n2 as described above. Initially, all p-terms in either A or B are considered to be terms present in the OR, but terms are removed or modified during the process of the pairwise comparison by means of the following algorithm to be applied for each pair ai,bi:

1. Form the potential consensus term c where the string n1 for c is produced by ORing the corresponding bits in the n1 strings for ai and bi, and the n2 string for c is formed as an OR of the n2 strings. That is:

   c n1 = ai n1 | bi n1

   c n2 = ai n2 | bi n2

2. If c is identical to ai in all bit positions in both strings n1 and n2,[28] then ai may be deleted from the OR as ai is redundant with respect to bi; bi covers ai. The symmetric situation exists if c<=>bi. This can only occur when the differences between the p-terms ai and bi are such that ai has X's wherever bi

---

[28] We shall write this as c<=>ai.

has literals. In such a case, ai covers bi.

Consider ai='0XX1'

bi='0101'

Using our new representation,

ai n1=0001 ai n2=1000

bi n1=0101 bi n2=1010

so c n1=0101 c n2=1010

and we see that c<=>b1; ai covers bi.

3. Otherwise, check to see if c is a consensus term. If the bitwise AND of c n1 with c n2 contains exactly one 1-bit, then c is a consensus term. The examination of a bitstring to determine if that bitstring has precisely one 1-bit may be performed in a computationally efficient manner by seeing if (c n1 & c n2) & ((c n1 & c n2)-1) = 0. For example, let:

ai='010'

bi='000'

Then in the new notation:

ai n1=010 ai n2=101

bi n1=000 bi n2=111

so:

c n1=010 c n2=111

c n1 & c n2 = 010

As this string has precisely one 1-bit, c is a consensus term. If c is not a consensus term, it is

known immediately that both ai and bi must be retained so far in the OR-merge.

4. If c has been discovered to be a consensus term, it must be determined which of the p-terms ai and bi (if any) this consensus term covers. The bitstring (c n1 & c n2) is subtracted from both c n1 and c n2 to "turn off" bits related to the consensus variable. When performed on the above example, c is left as c n1=000 c n2=101.

5. The original operation of ORing n1 and n2 bit strings (as described in step 1) is then applied again between this modified c and both the original ai and the original bi to form 2 new pairs of strings, ai' and bi' respectively.

   (a) If ai'<=>ai and bi'<=>bi, then remove ai and replace bi with c. This situation arises when the p-terms ai and bi differ in exactly one bit position, and neither p-term is 'X' in that position.

   (b) Else if ai'<≠>ai and bi'<=>bi, then replace bi with c.

   (c) Else if ai'<=>ai and bi'<≠>bi then replace ai with c.

   (d) Otherwise, retain both ai and bi.

The iterative application of this algorithm between all pairs of p-terms ai and bi produces a minimal OR of the expressions A and B. An APL implementation of this algorithm may be found in Appendix I, under the name 'OR'.

9.4.2.2 AND-merging of Sum-of-Products Expressions

In order to be able to AND together Boolean expressions in sum-of-products form, it is necessary to be able to AND p-terms. It is convenient that the bit-string representation for p-terms developed in section 9.4.2.1 above permits a computationally-efficient method for this ANDing. To AND p-terms ai and bi, we form:

```
c n1 = ai n1  |  bi n1
c n2 = ai n2  |  bi n2
```

by a bitwise OR as is done in step 1 of the OR algorithm. If there is a 1-bit in any bit position of the bitstring (c n1 & c n2), then the AND of p-terms ai and bi is null, as in some position ai (in cube notation) has a '1' where in the same position bi has a '0' (or vice versa), and as for any x, x & x'=0. Otherwise, c represents the new p-term resulting from the AND of ai and bi, and is to be retained.

In ANDing two sum-of-products expressions, the above procedure to AND a pair of p-terms is employed together with the previously-discussed OR algorithm, in the following manner. The AND of expressions A and B is the AND of p-term a1 with the expression B, ORed with the and of a2 with B, and so forth. Algebraically:

AB=

(a1+a2+...+an)B =

a1B+a2B+...+anB

The AND of a term ai with an expression B may be seen similarly to be the AND of ai with b1, ORed with the AND of ai with b2, etcetera. That is:

aiB = ai(b1+b2+...+bn) = aib1+aib2+...+aibn

As a consequence of this, the AND procedure calls the OR procedure repeatedly. The APL implementation of this algorithm may be seen in Appendix I, under the function name 'AND'.

Chapter X

CONCLUSIONS

The DES cryptographic system has been investigated, and the strength of the cipher found to lie in the S-box components. The S-boxes were examined for the existence of structural symmetries by the methods of McCluskey [12], and none were discovered.

The principal direction of this thesis has been to map the cryptanalytic problem into a domain for which powerful algorithmic and heuristic methods exist. Specifically, the discovery of the bits of K under known plaintext attack assumptions has been viewed as a problem in search, for which conventional search trees may be constructed and traversed. The worst-case size of a bidirectionally-searched AND/OR tree which stored key hypotheses at the leaves to avoid backtrack was shown theoretically in section 9.1 to be on the order of the key space size. The results of experimentation with such trees for a 2-round DES model (section 9.4.1.1) has indicated that in practice, tree sizes on the order of one seventh of the worst case theoretical maximum can be expected. If it is accepted that the amount of effort to expand a node is less than that involved in a key trial in exhaustive key search, and given that sufficient memory

capacity exists to store the mating halves of the bidirectional search tree, the cryptanalytic technique of bidirectional tree search represents an improvement over that of exhaustive key search.

In the course of developing a tractable search procedure for K, compact representations for the functions embodied by the S-boxes have been developed. At first, conventional functional-domain logical minimization techniques such as the Quine-McCluskey procedure [13,16] were applied to the S-boxes. The minimized S-box functions which resulted were theoretically shown to permit a small (and not large enough) reduction in the search tree size. More sophisticated spectral-domain minimization techniques [7,8,9,10] were then programmed and applied to the S-boxes, and these resulted in a far greater degree of simplification of the S-box functions.

Several attempts to demonstrate the usefulness of such minimal S-box representations in limiting the time required to uncover K through the upper-bounding of the search tree branching factor in a 2-round model of DES were programmed, in a variety of computer languages.

The first of these, a unidirectional key search procedure written in PL/I, failed to operate in tractable time on even a 2-round DES. Even when an initial oversight concerning the expansion of a subtree to represent the output of the DES f function with a value of 0 was corrected, the program

still could not discover K within reasonable computer time limits.

It was eventually discovered that the failure of this initial approach to key search could be attributed to two independent aspects of the search procedure. Firstly, even the linearization of the S-boxes did not sufficiently reduce the search tree branching factor to make a unidirectional key search tractable. Secondly, the maintenance of a single globally-posted hypothesis for K which was modified as contradictions in key bit values arose anywhere in the tree caused an excessive backtrack "thrashing" behavior in the search procedure.

It was demonstrated that the former problem could be overcome by searching the tree bidirectionally to greatly reduce the number of nodes which would have to be expanded. The latter problem was avoided by utilizing a more sophisticated search tree structure, the AND/OR tree. With key constraints stored locally at the tree leaves, backtracking was avoided as each node was visited only once as the tree was traversed.

After a theoretical investigation of the potential advantages of a bidirectional search procedure to expand the search tree in two directions, from both P and C, simultaneously, a realization of the uniform structure of the search tree led to an ability to formulate the cryptanalytic problem as a set of Boolean equations to be solved. Essential-

ly, the search tree could be "flattened", to reduce the problem of the discovery of K to that of the symbolic simplification of Boolean equations.

Although this approach initially appeared attractive due to its mathematical flavour, it was only useful insofar as it led to the development of the AND/OR tree methods. The connection between these two methods may be clearly perceived if one views an AND/OR tree as a pre-parsed Boolean expression containing only the Boolean constants 1 and 0 (no variables). In retrospect, it may be seen as foolhardy to flatten a tree and remove its structure in order to represent it mathematically, when the structure must be recovered, in order to symbolically simplify the expressions represented by the tree.

Before this was realized, a series of PROLOG routines were written to symbolically apply rules of Boolean algebra to simplify Boolean expressions which contain variables, ANDs, ORs, and NOTs. Initially, it was believed that the constraints on bits of K required for a bit of C to have the appropriate value, when formulated as a Boolean expression, could be reduced by this PROLOG system to produce K. Unfortunately, the very limited non-virtual memory of the DEC machine on which the routines were implemented led to a failure of this approach to provide any useful results. In view of the recursive implementation of PROLOG, a vast amount of memory would be required in order to simplify the key constraint expressions for even a 2-round DES.

From ideas about key constraint made evident by the PROLOG approach, a modification of the original search procedure was produced, written in APL. This approach generated an AND/OR expression tree for the key constraint expressions mentioned above, and then traversed this tree in a recursive top-down fashion, maintaining key hypotheses locally at the leaves as the tree was being traversed. These hypotheses were OR-merged or AND-merged by fast algorithms at appropriate nodes, to accomplish the evaluation of the search tree and uncover K.

In a number of trials, this method (employed unidirectionally) managed to discover the key K used for encryption with a 2-round DES. This AND/OR tree search benefits from the reduction in branching factor resulting from the use of the linearized S-boxes, and also could be performed in a bi-directional fashion, although this was never programmed. Even more significantly, experimentation with this search procedure indicated that such trees tend to possess approximately one seventh of the worst-case maximum possible number of nodes.

At least in the case of a 2-round model, the potential vulnerability of DES to methods of key search combined with appropriate S-box representations has been empirically demonstrated. Work continues towards the development of more computationally-efficient routines written in lower-level languages to experiment more fully with 2-round DES decryptions.

It is recommended that the DES algorithm be strengthened in one or more of the following simple ways, to reduce its susceptibility to the attacks outlined in this thesis. Increasing the number of layers of encryption in the algorithm by even a few should make the search tree sufficiently large to render key search as intractable as exhaustive key trials, as the tree grows exponentially in the number of layers of encryption, and it is only marginally small enough in its current form to permit our search methods to be applicable. A more complex use of the bits of K in the course of encryption would also present difficulties to the attack presented in this thesis. For instance, a concatenation of bits of K with developing L and R blocks at each level could serve to confound our method of cryptanalysis, by introducing far more complex constraint conditions on K. Finally, as other researchers in the area have indicated [2], increasing the length of K would also make DES more resistant to cryptanalytic attack. However, this would only cause the tree size to grow linearly with the increase in K size, whereas the addition of further rounds of encryption engenders an exponential growth in search tree size.

REFERENCES

1. Coppersmith, D. & Grossman, E. Generators for Certain Alternating Groups with Applications to Cryptography. SIAM J. Appl. Math. 29, #4, Dec. 1975, pp.624-627.

2. Diffie, W. & Hellman, M. Exhaustive Cryptanalysis of the NBS DES Computer #10, June 1977.

3. Diffie, W. & Hellman, M. New Directions in Cryptography IEEE Trans. on Info. Th. IT-22 #6, Nov. 1976, pp.644-654.

4. Feistel, H. Cryptography and Computer Privacy Scientific American, Vol. 228, May 1973, pp. 15-23.

5. Hellman, M. et. al. Results of an Initial Attempt to Cryptanalyze the NBS DES Tech. report SEL 76-042, Stanford University, 1976.

6. Hellman, M. A Cryptanalytic Time-Memory Tradeoff IEEE Trans. on Info. Th., 1980.

7. Hurst, S.L. (ed.) Conference: Recent Developments in Digital Logic Design Conference Proceedings, University of Bath, Claverton Down, Bath, Sept. 1977, pp.1.0-2.19.

8. Hurst, S.L. Logical Processing of Digital Signals Crane Russak, New York, 1978.

9. Hurst, S.L., Miller, D.M. & Muzio, J.C. Spectral Method of Boolean Function Complexity Electronics Letters, Vol. 18, #13, June, 1982. pp. 572-573.

10. Karpovsky, M.G. Finite Orthogonal Series in the Design of Digital Devices John Wiley & Son, New York, 1976.

11. Knuth, D.E. The Art of Computer Programming: Sorting and Searching Addison-Wesley, Mass., 1969.

12. McCluskey, E. J. Determination of Group Invariance or Total Symmetry of a Boolean Function BSTJ Vol. 35, #5, Nov. 1956, pp. 1445-1453.

13. McCluskey, E. J. Minimization of Boolean Functions BSTJ Vol. 35, #5, Nov. 1956, pp. 1417-1444.

14. Miller, D. M. & Muzio, J.C. Detection of Symmetries in Totally Specified or Partially Specified Combinational Functions Computers and Digital Techniques, Vol. 2, #5, Oct. 1979 pp.203-209.

15. Mott, T.H. Determination of the Irredundant Normal Forms of a Truth Function by Iterated Consensus of the Prime Implicants IRE Transactions on Electronic Computers, June, 1960, pp.245-252.

16. Mowle, F. J. A Systematic Approach to Digital Logic Design Addison-Wesley, New York, 1976.

17. Muzio, J.C., Miller, D.M. & Hurst, S.L. Multi-variable Symmetries and Their Detection Unpublished.

18. Nilsson, N.J. Problem-Solving Methods in Artificial Intelligence McGraw Hill, New York, 1971.

19. Pohl, I. Bidirectional and Heuristic Search in Path Problems Stanford Linear Accelerator Center Report, #104, May, 1969.

20. Rivest, R. L. et. al. A Method for Obtaining Digital Signatures and Public Key Cryptosystems CACM Vol. 21, #2, Feb. 1978, pp. 120-126.

21. Robinson, J.A. A Machine-Oriented Logic Based on the Resolution Principle JACM Vol. 12, #1, January 1965, pp. 23-41.

22. Shannon, C. E. Communication Theory of Secrecy Systems BSTJ Vol. 28, Oct. 1949, pp. 656-715.

23. Van Emden, M.H. & Kowalski, R.A. The Semantics of Predicate Logic as a Programming Language JACM Vol. 23, 1976, pp.733-742.

24. Warren, D. Pereira, L.M., & Pereira, F., PROLOG- The Language and its Implementation Compared with LISP SIGPLAN Notices (ACM), Vol. 12, #8.

25. Zissos, D. & Duncan, F.G. Boolean Minimization British Computer Journal, Vol. 16, #2, 1972, pp. 174-179.

26. Federal Information Processing Standards Publication. Announcing the Data Encryption Standard FIPS PUB 46, Jan. 1977.

TABLE 1 - MINIMAL SUM OF PRODUCT TERMS FOR EACH S-BOX AND OUTPUT.
- terms are 'ranked' by weight of contribution to correct s-box output

S-BOX NUMBER 1
****

-- output 1--		-- output 2--		-- output 3--		-- output 4--	
TERM	CONTR.	TERM	CONTR.	TERM	CONTR.	TERM	CONTR.
X1X010	0.562	X00X00	0.562	0010XX	0.562	00X10X	0.562
X1X111	0.562	0X010X	0.562	0100XX	0.562	X1X000	0.562
000X00	0.531	1X000X	0.562	X10X11	0.562	0110XX	0.562
X00011	0.531	10X00X	0.562	010X00	0.531	11XX00	0.562
X11001	0.531	01X110	0.531	10X100	0.531	110X0X	0.562
10X011	0.531	01X011	0.531	X11101	0.531	X011X1	0.562
001X01	0.531	1111OX	0.531	11011X	0.531	0001X0	0.531
001X10	0.531	1X1111	0.531	00X000	0.531	0X1010	0.531
00X100	0.531	00X010	0.531	00X011	0.531	1X1000	0.531
010X01	0.531	000X11	0.531	001X00	0.531	1X0001	0.531
010X10	0.531	001X01	0.531	10X111	0.531	011X10	0.531
100X01	0.531	X00011	0.531	101X10	0.531	1101X0	0.531
1001X0	0.531	1010X0	0.531	11X000	0.531	1100X1	0.531
X01110	0.531	110X10	0.531	1101X1	0.531	10111X	0.531
110X00	0.531	11011X	0.531	11100X	0.531	101X11	0.531
11001X	0.531	011000	0.516	1X1010	0.531	1X1111	0.531
101000	0.516			000101	0.516	0X1101	0.531
				100001	0.516	000011	0.516
				011110	0.516	100010	0.516
						010111	0.516

TERMS IN OUTPUT 1 = 17
OUTPUT 2 = 16
OUTPUT 3 = 19
OUTPUT 4 = 20

S-BOX NUMBER 2
****

-- output 1--		-- output 2--		-- output 3--		-- output 4--	
TERM	CONTR.	TERM	CONTR.	TERM	CONTR.	TERM	CONTR.
X00011	0.531	0X100X	0.562	00X00X	0.562	X0X001	0.562
001X01	0.531	00X11X	0.562	0010XX	0.562	10X1X0	0.562
1X0010	0.531	1101XX	0.562	X0100X	0.562	X11X11	0.562
101X00	0.531	00X000	0.531	X01X11	0.562	0X00X0	0.562
100X01	0.531	010X10	0.531	001X00	0.531	111X1X	0.562
0011X1	0.531	011X00	0.531	100X10	0.531	00X010	0.531
10X011	0.531	000X11	0.531	0X0111	0.531	010X10	0.531
000X00	0.531	0001X1	0.531	1100X1	0.531	000X11	0.531
0X0110	0.531	01X001	0.531	X11110	0.531	0X1100	0.531
01X000	0.531	10X010	0.531	111X10	0.531	01X101	0.531
X10001	0.531	110X00	0.531	00011X	0.531	X00111	0.531
X10111	0.531	0X1111	0.531	011X01	0.531	1010X1	0.531
X11101	0.531	11X110	0.531	10010X	0.531	11X000	0.531
X11110	0.531	10X100	0.531	11X101	0.531	110X01	0.531
X00110	0.531	1X1011	0.531	1X0110	0.531		
X11000	0.531	1X1101	0.531	1111X0	0.531		
11111X	0.531	110X11	0.531	010010	0.516		
001010	0.516	100001	0.516	010100	0.516		
110100	0.516						
011011	0.516						

TERMS IN OUTPUT 1 = 20
OUTPUT 2 = 18
OUTPUT 3 = 18
OUTPUT 4 = 14

S-BOX NUMBER 3
****

-- output 1--		-- output 2--		-- output 3--		-- output 4--	
TERM	CONTR.	TERM	CONTR.	TERM	CONTR.	TERM	CONTR.
00000X	0.531	0101XX	0.562	0X1X00	0.562	X100X0	0.562
00X100	0.531	001X00	0.531	XX1100	0.562	00X100	0.531
0X0100	0.531	0000X1	0.531	0X110X	0.562	00X001	0.531
1X0000	0.531	01X010	0.531	11X10X	0.562	X01100	0.531
01001X	0.531	00X011	0.531	00X000	0.531	010X10	0.531
01100X	0.531	10X010	0.531	0X0110	0.531	000X11	0.531
00X111	0.531	011X01	0.531	X00011	0.531	100X01	0.531
011X01	0.531	111X00	0.531	110X00	0.531	01X101	0.531
X11001	0.531	1X1111	0.531	X01111	0.531	11100X	0.531
1X0110	0.531	00X110	0.531	X01001	0.531	101X11	0.531
1X1010	0.531	00110X	0.531	X01010	0.531	110X11	0.531
10X101	0.531	100X00	0.531	X10111	0.531	001X10	0.531
1X0101	0.531	1X0101	0.531	10X010	0.531	011X11	0.531
0001X0	0.531	1100X1	0.531	1X0011	0.531	1X0000	0.531
0X0111	0.531	11X110	0.531	1X1001	0.531	X01010	0.531
01X011	0.531	11X011	0.531	111X10	0.531	X11011	0.531
10X011	0.531	101001	0.516	010001	0.516	100110	0.516
10X000	0.531			011011	0.516	111110	0.516
X10011	0.531						
011110	0.516						
111100	0.516						
111111	0.516						

TERMS IN OUTPUT 1 = 22
OUTPUT 2 = 17
OUTPUT 3 = 18
OUTPUT 4 = 18

S-BOX NUMBER 4
****

-- output 1--		-- output 2--		-- output 3--		-- output 4--	
TERM	CONTR.	TERM	CONTR.	TERM	CONTR.	TERM	CONTR.
X1X111	0.562	X1X110	0.562	001X1X	0.562	X1X000	0.562
011X1X	0.562	000X00	0.531	1000XX	0.562	110X0X	0.562
00001X	0.531	11X000	0.531	01001X	0.531	X01100	0.531
0X0100	0.531	01110X	0.531	0X1110	0.531	01100X	0.531
0110X0	0.531	10110X	0.531	000X00	0.531	1100X0	0.531
X00011	0.531	00X001	0.531	0X0101	0.531	10110X	0.531
0111X1	0.531	00X010	0.531	0001X0	0.531	1X1011	0.531
11011X	0.531	0X0111	0.531	X01001	0.531	1101X1	0.531
000X01	0.531	00101X	0.531	X11011	0.531	0000X0	0.531
0X1011	0.531	0100X1	0.531	X11101	0.531	000X01	0.531
0011X0	0.531	0X1010	0.531	1X0111	0.531	0X0110	0.531
100X00	0.531	10001X	0.531	1X1010	0.531	00X111	0.531
1010X0	0.531	X00111	0.531	110X00	0.531	001X11	0.531
101X01	0.531	101X00	0.531	1101X0	0.531	01111X	0.531
10111X	0.531	1011X0	0.531	11X111	0.531	1000X1	0.531
11000X	0.531	11X011	0.531	011000	0.516	1X0100	0.531
1X1001	0.531	11100X	0.531	101100	0.516	101X10	0.531
111100	0.516	11111X	0.531			010011	0.516
		110101	0.516				

TERMS IN OUTPUT 1 = 18
OUTPUT 2 = 19
OUTPUT 3 = 17
OUTPUT 4 = 18

S-BOX NUMBER 5
****

-- output 1--		-- output 2--		-- output 3--		-- output 4--	
TERM	CONTR.	TERM	CONTR.	TERM	CONTR.	TERM	CONTR.
X000X1	0.562	0X0010	0.531	001XX0	0.562	XX0110	0.562
0X110X	0.562	1X0000	0.531	0101XX	0.562	X10X10	0.562
101X1X	0.562	011X00	0.531	11X00X	0.562	110X1X	0.562
1010X0	0.531	001X01	0.531	X01X00	0.562	1XX111	0.562
110X00	0.531	010X01	0.531	100X10	0.531	00X011	0.531
100X01	0.531	X00111	0.531	X01011	0.531	1100X0	0.531
10X110	0.531	10101X	0.531	X11001	0.531	10X001	0.531
00X010	0.531	1001X1	0.531	10011X	0.531	01X110	0.531
0X0111	0.531	11X110	0.531	10110X	0.531	1X1010	0.531
01X000	0.531	00X001	0.531	1100X1	0.531	1111X1	0.531
0101X1	0.531	0X1000	0.531	00000X	0.531	001X00	0.531
01X110	0.531	0010X1	0.531	0000X1	0.531	0011X1	0.531
11001X	0.531	010X10	0.531	000X01	0.531	010X01	0.531
X10111	0.531	10X111	0.531	0X1100	0.531	0101X0	0.531
X11110	0.531	1100X1	0.531	X11111	0.531	01100X	0.531
011011	0.516	110X00	0.531	X00001	0.531	0X1011	0.531
111001	0.516	X11000	0.531	111X10	0.531	10X100	0.531
		1X1011	0.531				
		000100	0.516				
		001110	0.516				
		101100	0.516				
		011111	0.516				
		111101	0.516				

TERMS IN OUTPUT 1 = 17
OUTPUT 2 = 23
OUTPUT 3 = 17
OUTPUT 4 = 17

S-BOX NUMBER 6
****

-- output 1--		-- output 2--		-- output 3--		-- output 4--	
TERM	CONTR.	TERM	CONTR.	TERM	CONTR.	TERM	CONTR.
1X110X	0.562	00X011	0.531	00X100	0.531	01X01X	0.562
X00X00	0.562	0X0101	0.531	0X0100	0.531	01X10X	0.562
1X11X1	0.562	0010X1	0.531	00X001	0.531	X01X01	0.562
0X1000	0.531	010X01	0.531	0X1010	0.531	00100X	0.531
X01101	0.531	100X10	0.531	0110X0	0.531	0011X1	0.531
0101X1	0.531	110X00	0.531	10001X	0.531	10X110	0.531
1X1010	0.531	001X11	0.531	11000X	0.531	11X111	0.531
101X01	0.531	X01011	0.531	01101X	0.531	000X10	0.531
1100X1	0.531	10110X	0.531	11X001	0.531	X00011	0.531
10X111	0.531	110X11	0.531	1111X0	0.531	011X10	0.531
0000X1	0.531	0X0110	0.531	000X11	0.531	100X00	0.531
00X110	0.531	0X1100	0.531	0001X0	0.531	10X011	0.531
0X1011	0.531	01X010	0.531	0X0001	0.531	11X000	0.531
01111X	0.531	01011X	0.531	X10111	0.531	110X01	0.531
1000X0	0.531	0110X0	0.531	011X10	0.531	X11010	0.531
010010	0.516	1001X0	0.531	10010X	0.531	X11100	0.531
110110	0.516	10011X	0.531	1011X1	0.531		
		111X10	0.531	1X1110	0.531		
		11111X	0.531	1X0011	0.531		
		000000	0.516	11X110	0.531		
		100001	0.516	101000	0.516		
		111001	0.516	011101	0.516		

TERMS IN OUTPUT 1 = 17
OUTPUT 2 = 22
OUTPUT 3 = 22
OUTPUT 4 = 16

S-BOX NUMBER 7
****

-- output 1--		-- output 2--		-- output 3--		-- output 4--	
TERM	CONTR.	TERM	CONTR.	TERM	CONTR.	TERM	CONTR.
X0X110	0.562	00X00X	0.562	0001XX	0.562	0X1000	0.531
1001XX	0.562	X0X110	0.562	01X0X1	0.562	000X01	0.531
1X011X	0.562	0X011X	0.562	X110X1	0.562	100X00	0.531
X01000	0.531	X1X111	0.562	1011XX	0.562	0001X1	0.531
100X11	0.531	011X00	0.531	000X10	0.531	00X101	0.531
10X101	0.531	100X01	0.531	0X0110	0.531	1001X0	0.531
11X001	0.531	11X010	0.531	1100X0	0.531	0X1110	0.531
X10111	0.531	011X11	0.531	1000X1	0.531	0X1011	0.531
11X111	0.531	101X11	0.531	01101X	0.531	1X1010	0.531
000X01	0.531	010X01	0.531	101X10	0.531	1X0011	0.531
0X0010	0.531	010X10	0.531	1X1110	0.531	1100X1	0.531
001X11	0.531	100X10	0.531	0X1111	0.531	11110X	0.531
001X00	0.531	101X00	0.531	01000X	0.531	110X11	0.531
0X0001	0.531	110X11	0.531	1X0100	0.531	010X00	0.531
01101X	0.531	110100	0.516	1X1101	0.531	01X011	0.531
1100X0	0.531	111001	0.516	001000	0.516	01010X	0.531
010100	0.516			011100	0.516	0101X0	0.531
011101	0.516			110111	0.516	X00101	0.531
111100	0.516					10X100	0.531
						11X010	0.531
						000010	0.516
						101001	0.516
						101111	0.516

TERMS IN OUTPUT 1 = 19
OUTPUT 2 = 16
OUTPUT 3 = 18
OUTPUT 4 = 23

S-BOX NUMBER 8
****

-- output 1--		-- output 2--		-- output 3--		-- output 4--	
TERM	CONTR.	TERM	CONTR.	TERM	CONTR.	TERM	CONTR.
X01010	0.531	X11X00	0.562	00X01X	0.562	111OXX	0.562
000X11	0.531	X100X1	0.562	001X0X	0.562	100X1X	0.562
10X010	0.531	X1X011	0.562	0101XX	0.562	01001X	0.531
1X1000	0.531	0X0110	0.531	01X11X	0.562	X11000	0.531
01110X	0.531	X01010	0.531	010X00	0.531	0X1110	0.531
1100X1	0.531	0X0011	0.531	X10100	0.531	0X1101	0.531
000X00	0.531	11110X	0.531	X01011	0.531	010X11	0.531
X00101	0.531	00X000	0.531	1011X0	0.531	1X0110	0.531
X01100	0.531	00X101	0.531	1001X1	0.531	111X00	0.531
0100X0	0.531	X01111	0.531	0X1011	0.531	110X01	0.531
01000X	0.531	0X0101	0.531	1000X0	0.531	00000X	0.531
01011X	0.531	01X110	0.531	1X0001	0.531	00X011	0.531
101X11	0.531	100X00	0.531	11X010	0.531	000X01	0.531
1011OX	0.531	1001X1	0.531	11100X	0.531	001X10	0.531
1101X0	0.531	10X100	0.531	1111X1	0.531	0011X0	0.531
11010X	0.531	110X10	0.531			10X000	0.531
11111X	0.531	101001	0.516			1X1111	0.531
001001	0.516					010100	0.516
011011	0.516						

TERMS IN OUTPUT 1 = 19
OUTPUT 2 = 17
OUTPUT 3 = 15
OUTPUT 4 = 18

=>

TABLE 2. C(F) METRIC FOR S-BOXES BEFORE & AFTER TRANSLATION

S-BOX #	OUTPUT # 1 bef, aft	2 bef, aft	3 bef, aft	4 bef, aft
(1)	148 252	156 244	144 248	136 260
(2)	120 252	132 264	144 232	168 252
(3)	112 260	128 252	152 232	128 236
(4)	148 248	148 252	148 252	148 248
(5)	148 248	112 244	152 244	152 224
(6)	132 252	120 244	128 236	156 256
(7)	136 244	148 268	136 240	108 252
(8)	140 228	144 252	152 244	140 244

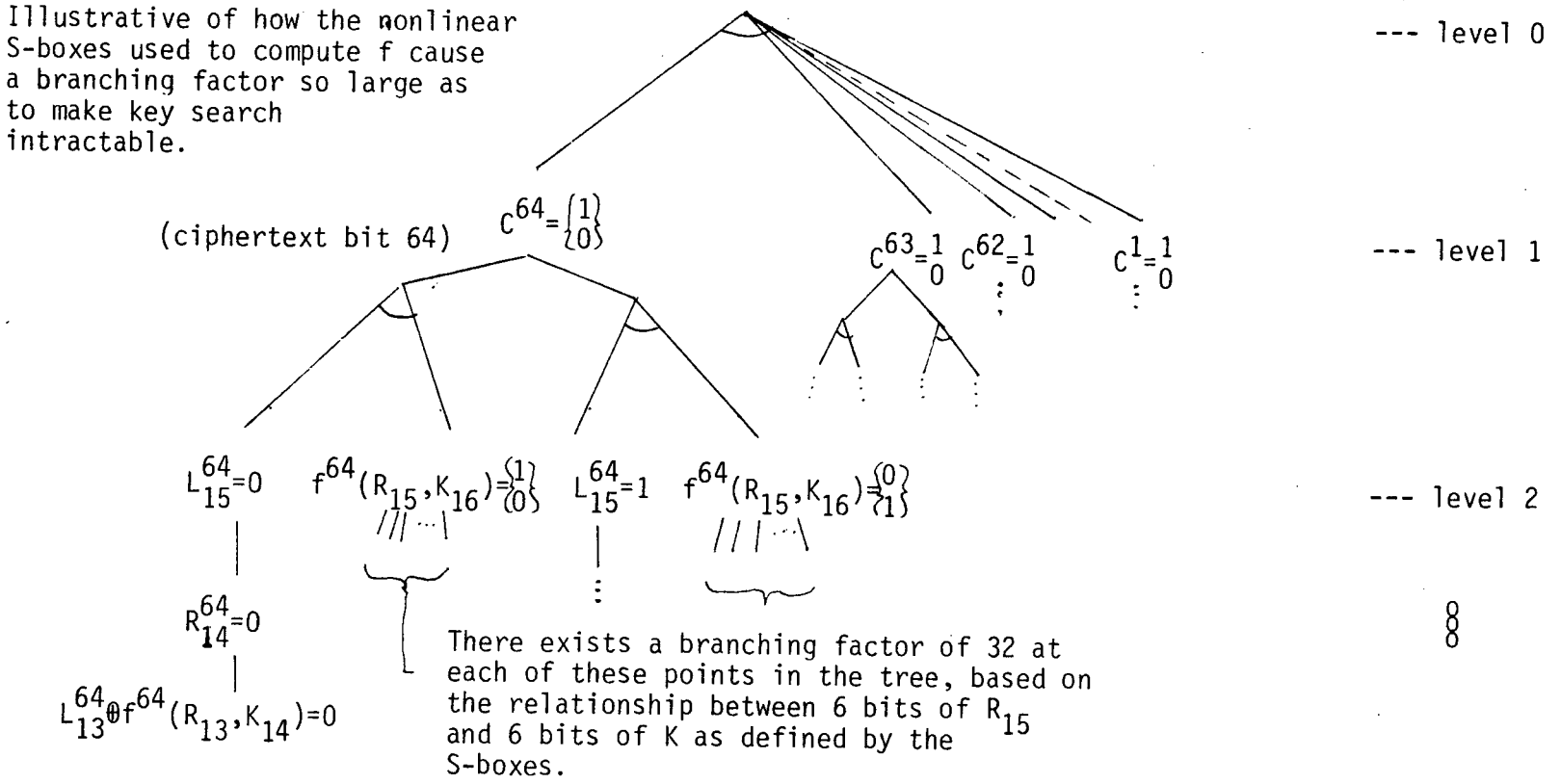
AVERAGE COMPLEXITY BEFORE TRANSLATION: 139

AVERAGE COMPLEXITY AFTER TRANSLATION: 247

$$C(f) = n2^n - \frac{1}{2^{n-2}} \left[\sum_{v=0}^{2^n-1} ||v|| \, r_v^2 \right]$$

C(f) is a spectral-domain measure of function complexity equal to the classical functional-domain complexity measure which counts the number of topologically adjacent pairs of assignments for which f takes the same value for both assignments in the pair.

FIGURE 1 Exhaustive Tree Search Using No S-Box Reduction



Notational Notes: - superscripts indicate bit position in block.
- subscripts indicate encryption round $0 \leq r \leq 16$
- after selection from $\{a\}\{b\}$ made at level 1 node, select same position for subtree.

FIGURE 2 Complete Partitioning of a Matrix

- an example of the partitioning of a matrix, from McCluskey [7 p.1447]

The matrix X:

```
0 0 0 1 0 0
0 0 1 0 0 0
1 0 0 0 0 0
0 0 0 1 0 1
0 0 0 1 1 0
0 0 1 0 0 1
0 0 1 0 1 0
1 1 0 0 0 0
0 1 1 1 1 1
```

- transcribed results of running the APL routine PARTITION on X:

Row and column slice points:
1 3 4 8 9
1 2 3 5

Partitioned matrix:

```
  0 | 0 | 01 | 00
  0 | 0 | 10 | 00
3 ------------------
  1 | 0 | 00 | 00
4 ------------------
  0 | 0 | 01 | 01
  0 | 0 | 01 | 10
  0 | 0 | 10 | 01
  0 | 0 | 10 | 10
8 ------------------
  1 | 1 | 00 | 00
9 ------------------
  0 | 1 | 11 | 11
    2   3    5
```

FIGURE 3 Essential and Alternative Sum-of-Product Terms

FOR S-BOX 1 OUTPUT 3:

Essential:

000101
100001
011110
010X00 — i.e.: $\overline{X}_1 X_2 \overline{X}_3 X_5 \overline{X}_6$
10X100
X11101 X_4 is a "don't care"
11011X
0010XX
0100XX
X10X11

Alternative:

00X000	1
0X0000	1
00X011	4
0X0011	4
001X00	8
X01100	8
(10X111)	25
1X0111	25
X01010	28
101X10	28
1X1010	28
101X10	31
1011X0	31
10111X	31
10111X	32
(10X111)	32
X10000	33
11X000	33
1101X1	37
11X101	37
11X000	38
1110X0	38
11100X	38
11100X	39
111X01	39
1X1010	40
1110X0	40

Note inter-class occurences of the same p-term.

Example of one of the many alternative "classes". One member of each must be chosen as a p-term in the minimal representation.

FIGURE 4 Representation of Quasi-Best Set Search Tree

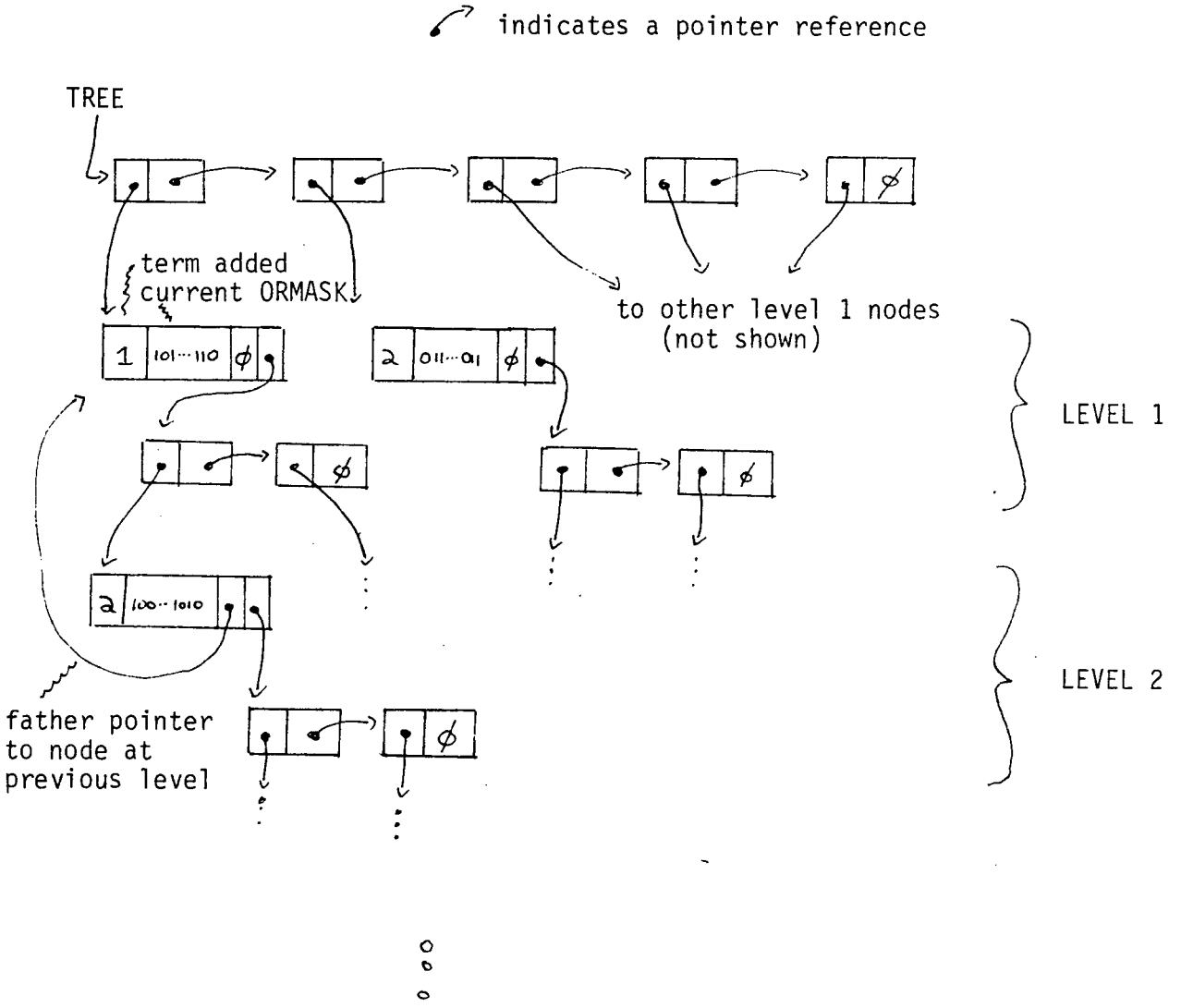


FIGURE 5 Permutation Cutoff During N-ary Tree Expansion

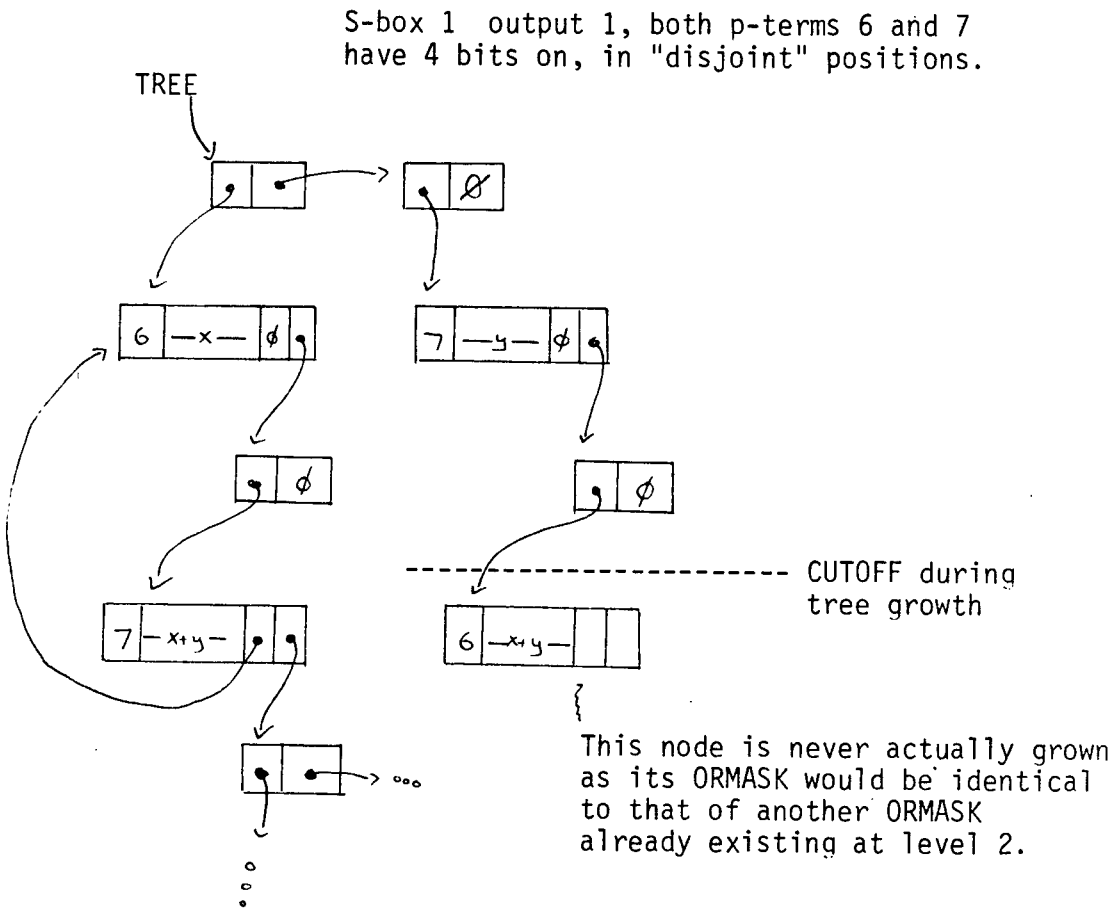
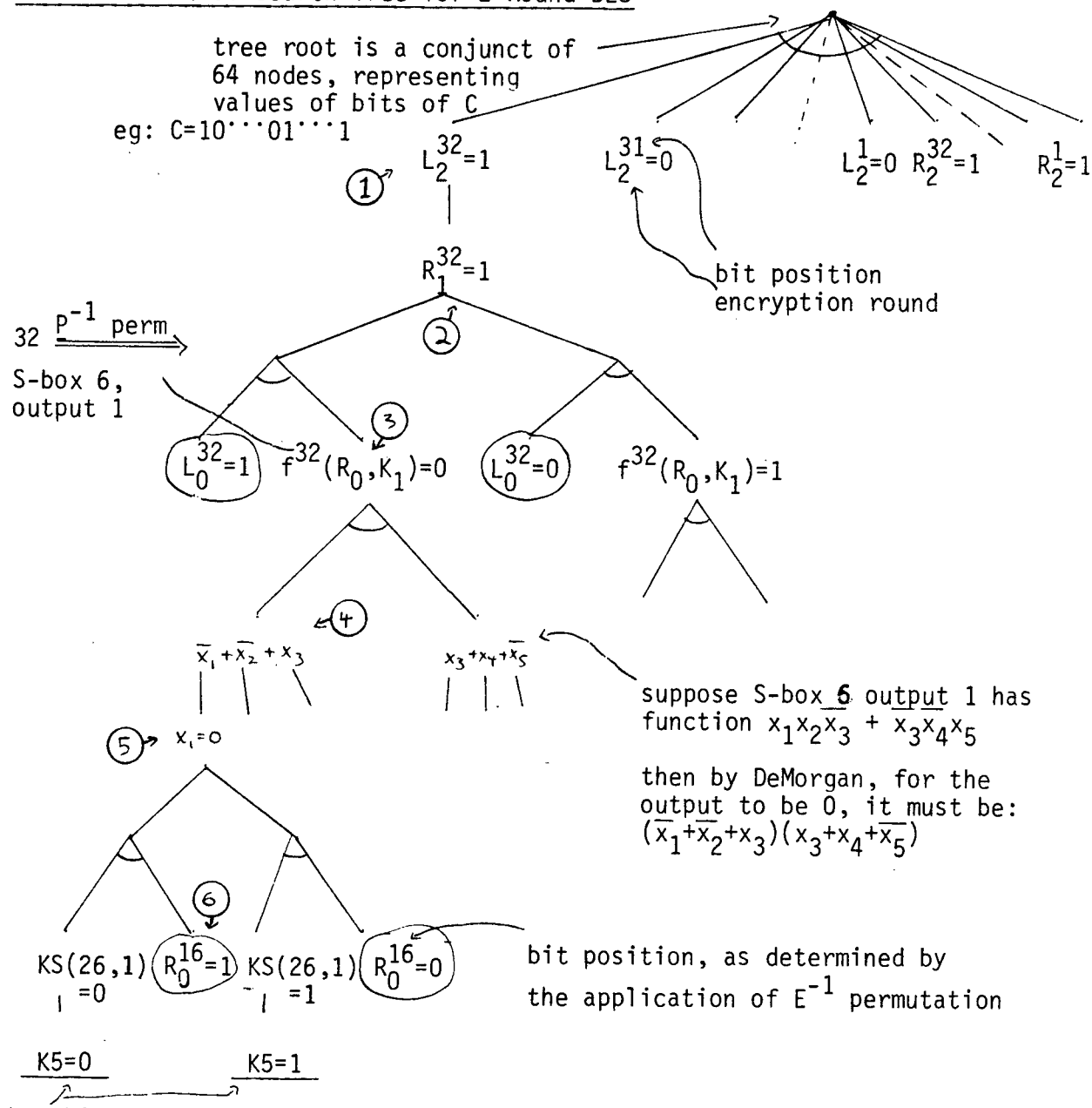


FIGURE 6 Partial Search Tree for 2-Round DES



NOTES :

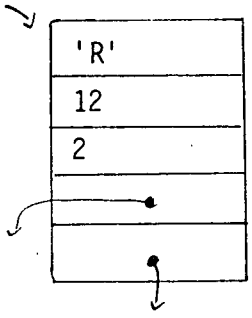
1) AND nodes labelled with arcs.

2) OR paths exist simultaneoulsy only conceptually. Alternatives developed iff backtrack.

3) Circled nodes are search tree leaves, which may cause backtrack.

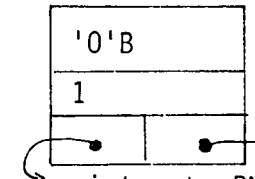
FIGURE 7 Nodes in the Cryptanalytic Search Tree

1) SUPER descriptor node



- type code. this is a descriptor for a type 'R' node
- position from {1,32} in R block
- round ε{1,16} in encryption
- father pointer
- pointer to RNODE

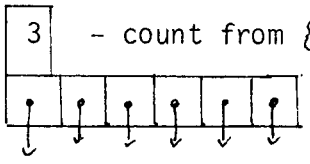
2) Node of type 'RNODE'



- value of RNODE from {0,1}
- count. i.e., # of times node expanded from {0,2}

pointer to RNODE 2 levels before
pointer to an FNODE1 or an FNODE0

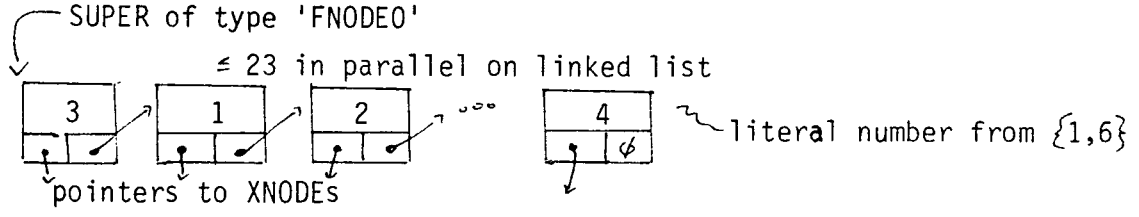
3) Node of type 'FNODE1'



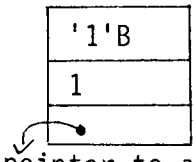
- count from {1,23} which conjunct term in sum-of-product representation is currently used

to 6 XNODES, expanded in parallel

4) Chain of type 'FNODE0' nodes



5) Node of type 'XNODE'

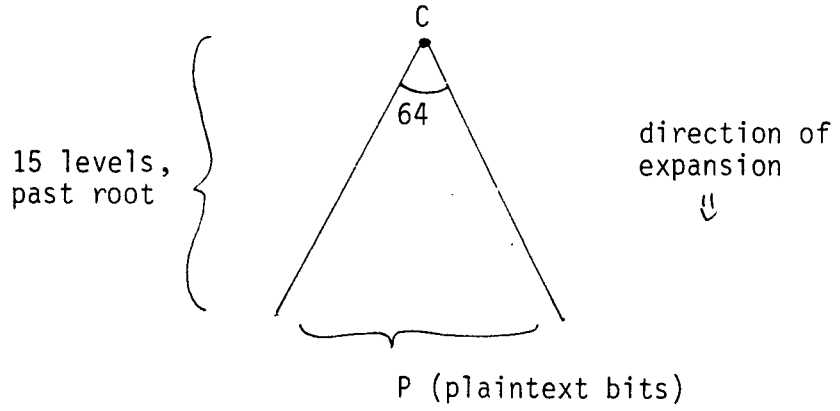


- value of XNODE from {0,1}
- count. i.e., # of times node expanded from {0,2}

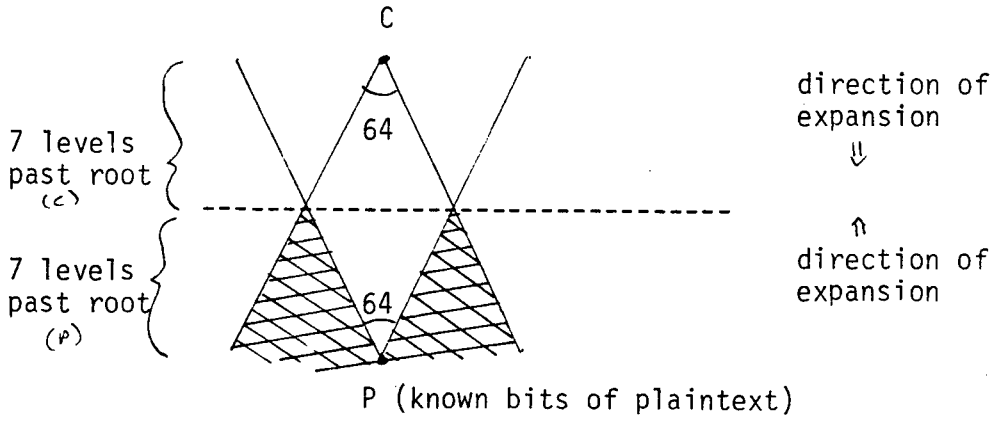
pointer to an RNODE. (a virtual 2nd pointer affects key bit hypothesis)

FIGURE 8 Bidirectional Search Tree

(a) unidirectional search tree



(b) bidirectional search tree



- cross-hatched area is an area existing in the unidirectional search tree, which need never be developed in the bidirectional tree.
- bidirectional search techniques make linear what is exponential.

FIGURE 9 2-Round Search Tree of Uniform Structure

note: bit positions (superscripts) not explicitly shown

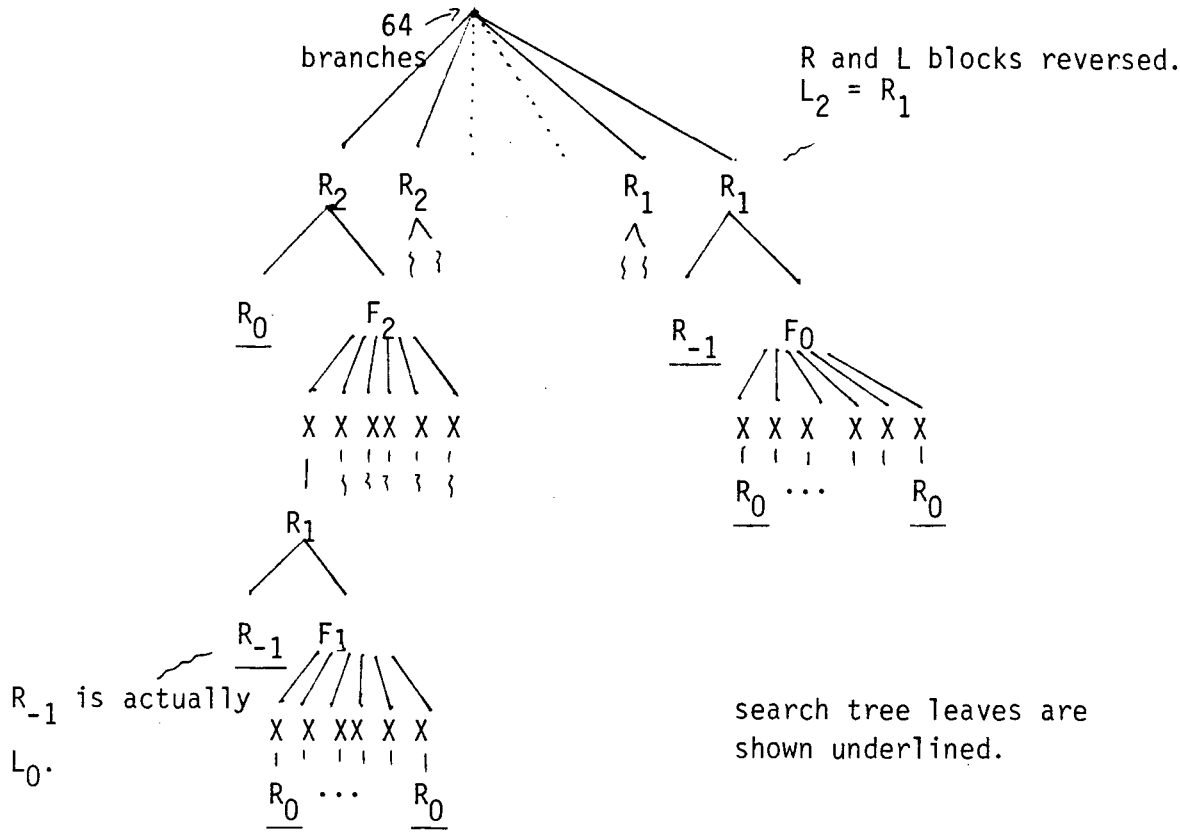
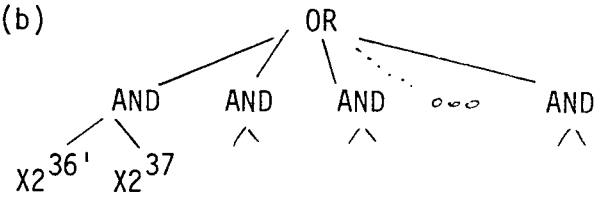


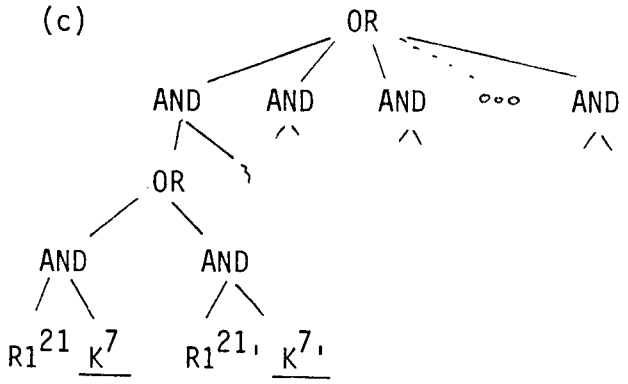
FIGURE 10 Stages in the Development of the AND/OR Search Tree

(a) $R2^{32}$

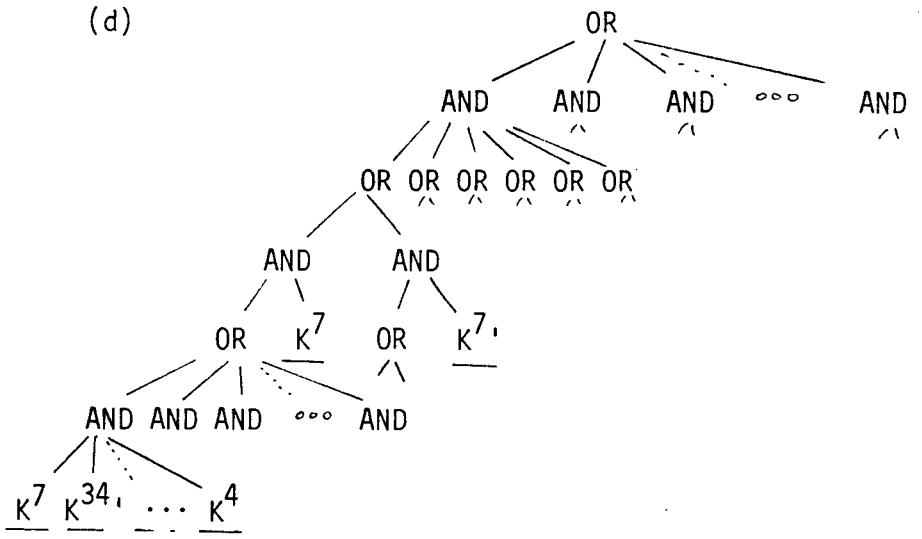
(b)



(c)



(d)



APPENDIX A

APL CODE FOR THE DETECTION OF GROUP INVARIANCE OF A BOOLEAN FUNCTION BY MCCLUSKEY ALGORITHM.

```
      ∇  Z←BINARY DECBOX
[1]   ⍝ TO CONVERT AN SBOX INTO BINARY FORM
[2]   Z← 3 1 2 ⍉ 2 2 2 2 ⍴DECBOX
      ∇

      ∇  Z←DUPKILL V
[]    Z←((⍳⍴V)=V⍳V)/V
      ∇

      ∇  Z←INITPARTIT M;RWT;CWT;RSLICE;CSLICE
[ 1]  ⍝ TO INITIALLY PARTITION A MATRIX INTO SUBMATRICES
[ 2]  RWT←+/M
[ 3]  CWT←+⌿M
[ 4]  →(1=⍴RWT)/ROWVEC
[ 5]  RSLICE←((1,≠/ ¯1 0 ↓((¯1+⍳⍴RWT)∘.=(2⍴⍴RWT)⍴RWT)[;⍳2])/⍳⍴RWT),
1+1↑⍴M
[ 6]  →JOIN1
[ 7]  ROWVEC:RSLICE←1,1+1↑⍴M
[ 8]  JOIN1:→(1=⍴CWT)/COLVEC
[ 9]  CSLICE←((1,≠/ ¯1 0 ↓((¯1+⍳⍴CWT)∘.=(2⍴⍴CWT)⍴CWT)[;⍳2])/⍳⍴CWT),
1+¯1↑⍴M
[10]  →JOIN2
[11]  COLVEC:CSLICE←1,1+¯1↑⍴M
[12]  JOIN2:Z←((1+⌈/⍴M)↑RSLICE),[0.5](1+⌈/⍴M)↑CSLICE
      ∇

      ∇  Z←IOTA X
[1]   ⍝ FOR ENUMERATION OF INCLUSIVELY BOUNDED INTEGER LIST
[2]   →(X[2]≤X[1])/SCAL
[3]   Z←X[1]+¯1+⍳1+|-/X
[4]   →0
[5]   SCAL:Z←,X[1]
      ∇

      ∇  Z←BITPOS ON S
[1]   ⍝ GIVEN AN SBOX AND OUTPUT BIT POSITION, RETURN THE 32×6
[2]   ⍝ MATRIX OF BINARY INPUTS FOR WHICH THE OUTPUT BIT IS 1
[3]   Z←(16 16)⍴Z←¯1+(,Z)/⍳×/⍴Z←S[;;BITPOS]
[4]   Z← 3 1 2 ⍉ 2 2 2 2 ⍴Z
[5]   Z←Z[1;;3],Z[2;;],Z[1;;4]
      ∇
```

```
      ∇ Z←SLICES PARTITCALL M;RSLICE;CSLICE;NEWRSL;NEWCSL;RCTR;C
CTR;SUB;P
[ 1]    ⍝
[ 2]    ⍝ RECURSIVE FUNCTION TO FURTHER PARTITION A MATRIX M,
[ 3]    ⍝ GIVEN A CURRENT STATE OF PARTITIONING, SLICES.
[ 4]    ⍝ SLICES IS A MATRIX WITH 2 ROWS, WHERE FIRST ROW CONTAINS
[ 5]    ⍝ POINTERS BEFORE WHICH ROW SLICES OCCUR, SECOND ROW SAME F
OR COLS.
[ 6]    ⍝
[ 7]    →(∧/ 1 1 =⍴M)/TRIV
[ 8]    RSLICE←(SLICES[1;]≠0)/SLICES[1;]
[ 9]    CSLICE←(SLICES[2;]≠0)/SLICES[2;]
[10]    MAX←(⍴RSLICE)⌈⍴CSLICE
[11]    Z← 0 1 ↑((MAX↑RSLICE),[0.5] MAX↑CSLICE), 2 1 ⍴0
[12]    →(∧/∧/(2 2 ↑Z)=⍉(1+⍴M),[0.5] 0 0)/TRIV
[13]    ⍝
[14]    ⍝ ITERATE THROUGH EACH SUBMTX, RECURSIVELY SLICING
[15]    NEWRSL←NEWCSL←⍳0
[16]    CCTR←1
[17]    CLOOP:RCTR←1
[18]    RLOOP:SUB←M[IOTA 0 ¯1 +RSLICE[RCTR+ 0 1];IOTA 0 ¯1 +CSLICE[
CCTR+ 0 1]]
[19]    RECURSE:P←(INITPARTIT SUB) PARTITCALL SUB
[20]    NEWRSL←NEWRSL,¯1+RSLICE[RCTR]+P[1;]
[21]    NEWCSL←NEWCSL,¯1+CSLICE[CCTR]+P[2;]
[22]    →((RCTR←RCTR+1)<⍴RSLICE)/RLOOP
[23]    →((CCTR←CCTR+1)<⍴CSLICE)/CLOOP
[24]    ⍝
[25]    ⍝ IF NOT AT TOP LEVEL, JUST RETURN NEWRSL, NEWCSL
[26]    →(~∨/RECURSE=⎕LC)/TOPLEVEL
[27]    Z←(MAX↑RSLICE),[0.5](MAX←(⍴RSLICE)⌈⍴CSLICE)↑CSLICE
[28]    Z← 0 ¯1 ↓ 0 1 ↓Z
[29]    →0
[30]    TOPLEVEL:
[31]    NEWRSL←(⌈/⍴M)↑(NEWRSL≤1↑⍴M)/NEWRSL←NEWRSL[⍋NEWRSL←DUPKILL N
EWRSL,RSLICE]
[32]    NEWCSL←(⌈/⍴M)↑(NEWCSL≤¯1↑⍴M)/NEWCSL←NEWCSL[⍋NEWCSL←DUPKILL
NEWCSL,CSLICE]
[33]    Z←NEWRSL,[0.5] NEWCSL
[34]    →0
[35]    TRIV:Z← 2 0 ⍴0
      ∇
```

```
      ∇ Z←PARTITION M;OLDZ
[ 1]  ⍝
[ 2]  ⍝ TO FULLY PARTITION A MATRIX, M.
[ 3]  ⍝ DIVIDE INTO ROWS AND COLUMNS SUCH THAT ALL ROWS/COLS IN A
 SUBMATRIX
[ 4]  ⍝ HAVE AN EQUAL NUMBER OF 1'S.
[ 5]  ⍝ RETURN PARTITION POINTS AS A LIST OF POINTERS INTO M BEFO
RE WHICH
[ 6]  ⍝ DIVISIONS SHOULD OCCUR
[ 7]  ⍝
[ 8]  OLDZ←(INITPARTIT M) PARTITCALL M
[ 9]  LOOP:→(∧/∧/OLDZ=Z←(OLDZ, 2 1 ⍴1+⍴M) PARTITCALL M)/0
[10]  OLDZ←Z
[11]  →LOOP
      ∇


      ∇ PARTITION∆ALL;BOX;BIT;STD;TR;P
[ 1]  ⍝
[ 2]  ⍝ TO PARTITION STANDARD TRANSMISSION MATRICES REPRESENTING
[ 3]  ⍝ ELEMENTARY PRODUCT TERM BOOLEAN FNS FOR EACH OF THE 32
[ 4]  ⍝ S-BOX - OUTPUT PAIRS:
[ 5]  ⍝
[ 6]  BOX←1
[ 7]  BOXLOOP:BIT←1
[ 8]  BITLOOP:TR←BIT ON BINARY SBOX[BOX;;]
[ 9]  ⍝ FORM STANDARD MATRIX:
[10]  STD←TR≠(⍴TR)⍴FLIP←(0.5×1↑⍴TR)<+⌿TR
[11]  ⍝RANK SO 1'S INCREASE IN DIRECTIONS → AND ↓
[12]  STD←STD[⍋+/STD;]
[13]  STD←STD[;⍋+⌿STD]
[14]  ⍝ PARTITION THIS MATRIX:
[15]  P←PARTITION STD
[16]  ''
[17]  'FOR S-BOX ',(⍕BOX),' BIT: ',⍕BIT
[18]  'ROW SLICES: ', 3 0 ⍕(P[1;]>0)/P[1;]
[19]  'COL SLICES: ', 3 0 ⍕(P[2;]>0)/P[2;]
[20]  ((1↑⍴TR)=+/P[1;]>0)/'NO POSSIBLE ROW PERMUTATIONS'
[21]  ((¯1↑⍴TR)=+/P[2;]>0)/'NO POSSIBLE COL PERMUTATIONS'
[22]  →((BIT←BIT+1)≤4)/BITLOOP
[23]  →((BOX←BOX+1)≤8)/BOXLOOP
      ∇
```

```
      ∇  Z←PARTIT PRINTΔPARTIT M;R;C;EX
[ 1]  ⍝ GIVEN PARTITION POINTS FOR A MATRIX M,
[ 2]  ⍝ PRINT THE MATRIX IN PARTITIONED FORM.
[ 3]  ⍝ LEAVE BLANK ROWS/COLUMNS BETWEEN SUBMATRICES
[ 4]  ⍝
[ 5]  PARTIT
[ 6]  Z←(M[1;]≠' ')/M←⍉M
[ 7]  C←(PARTIT[2;]≠0)/PARTIT[2;]
[ 8]  EX←((⍴C)+¯1↑⍴Z)⍴1
[ 9]  EX[¯1+C+⍳⍴C]←0
[10]  Z←EX\Z
[11]  R←(PARTIT[1;]≠0)/PARTIT[1;]
[12]  EX←((⍴R)+1↑⍴Z)⍴1
[13]  EX[¯1+R+⍳⍴R]←0
[14]  Z←EX⌿Z
      ∇


      ∇  Z←RAND
[1]   ⍝
[2]   ⍝ TO GENERATE RANDOM 32×6 BINARY MATRICES IN STANDARD FORM
[3]   ⍝
[4]   Z←¯1+? 32 6 ⍴2
[5]   ⍝ PUT INTO STD FORM AND RANK :
[6]   Z←Z≠(⍴Z)⍴(0.5×1↑⍴Z)<+⌿Z
[7]   Z←Z[⍋+/Z;]
[8]   Z←Z[;⍋+⌿Z]
      ∇
```

```
      ∇ SYM
[ 1]  ⍝ TR←1 ON BINARY SBOX[1;;]
[ 2]  STD←TR≠(⍴TR)⍴FLIP←(0.5×1↑⍴TR)<+⌿TR
[ 3]  EQUAL←(0.5×1↑⍴STD)=+⌿STD
[ 4]  ⍝ EXCHG ROWS SO NO. 1'S INCREASES ↓ AND →
[ 5]  STD←STD[⍒+/STD;]
[ 6]  STD←STD[;PERM←⍒+⌿STD]
[ 7]  PRIMING←EQUAL\ ¯1 0 ↓⍉((+/EQUAL)⍴2)⊤⍳2*+/EQUAL
[ 8]  ⍝ PRIMING IS MTX WHOSE ROWS INDICATE POSSIBLE WAYS COLS
[ 9]  ⍝ OF STD CAN BE PRIMED
[10]  ⍝
[11]  ⍝ PARTITION THE STD MATX
[12]  PART←PARTITION STD
[13]  ⍝ ELIMINATE SOME OF THE POSSIBLE PRIMING OPERATIONS:
[14]  ⍝ IF SOME ROW IS ALL 0/1 AND NO ROW AFTER PRIMING IS 0/1,E
[15]  ⍝ THE PRIMING IS NOT POSSIBLE
[16]  CONST←0
[17]  ELIM:KEEPPRIMROW←(1↑⍴PRIMING)⍴1
[18]  K←1
[19]  REM1:KEEPPRIMROW[K]←~(∨/∧⌿CONST=STD)∧~∨/∧⌿CONST=STD≠(⍴STD)⍴
PRIMING[K;]
[20]  →((K←K+1)≤1↑⍴PRIMING)/REM1
[21]  PRIMING←KEEPPRIMROW⌿PRIMING
[22]  →((CONST←CONST+1)≤1)/ELIM
[23]  ⍝
[24]  ⍝ ELIMINATE PRIMINGS IF PRIMED MTX DOES NOT PARTITION AS
[25]  ⍝ DOES THE STD MTX.
[26]  ⍝ SAVE TO COLUMN PERMUTATIONS REQD FOR 1'S →
[27]  SAVEPERMS←(⍴STD)⍴0
[28]  K←1
[29]  KEEPPRIMROW←(1↑⍴PRIMING)⍴1
[30]  SAVEMTX←(0,⍴STD)⍴0
[31]  REM2:POSSM←STD≠(⍴STD)⍴PRIMING[K;]
[32]  POSSM←POSSM[⍒+/POSSM;]
[33]  POSSM←POSSM[;SAVEPERMS[K;]←⍒+⌿POSSM]
[34]  SAVEMTX←SAVEMTX,[1] POSSM
[35]  KEEPPRIMROW[K]←∧/∧/PART=PARTITION POSSM
[36]  →((K←K+1)≤1↑⍴PRIMING)/REM2
[37]  PRIMING←KEEPPRIMROW⌿PRIMING
[38]  SAVEPERMS←KEEPPRIMROW⌿SAVEPERMS
[39]  ⍝ FOR STD + EACH RETAINED PRIMING, CHECK VAR PERM:
[40]  SI←PART PERMUTE STD
[41]  PRINTΔINVARIANT⍳¯1↑⍴SAVEPERM
[42]  ⍝
[43]  K←1
[44]  GOLOOP:SI←PART PERMUTE SAVEMTX[K;;]
[45]  PRINTΔINVARIANT SAVEPERM[K;]
[46]  →((K←K+1)≤1↑⍴PRIMING)/GOLOOP
      ∇
```

APPENDIX B

APL CODE FOR QUINE-MCCLUSKEY MINIMIZATION OF S-BOXES.

```
      ∇  Z←ALT BOXINP
[1]   ⍝ GIVEN BOX NUMBER AND OUTPUT, RETURN TABLE OF ALTERNATE SP'
S
[2]   IND←BOXINP[2]+4×¯1+BOXINP[1]
[3]   Z←(~∧/' '=Z)⌿Z←ALLALT[IOTA,ALLPTR[IND+ 0 1 ;2];]
      ∇


      ∇  ANALYZE;SBOXCNT;OUTBIT
[ 1]  ⍝
[ 2]  ⍝ TO CALC AND SAVE THE ESSENTIAL AND ALTERNATIVE SP TERMS
[ 3]  ⍝ FOR EACH SBOX AND OUTPUT, IN INDEXED MATRICES ALLESS
[ 4]  ⍝ AND ALLALT, RESPECTIVELLY
[ 5]  ALLALT← 0 10 ⍴' '
[ 6]  ALLESS← 0 6 ⍴' '
[ 7]  ALLPTR← 1 2 ⍴ 1 1
[ 8]  SBOXCNT←1
[ 9]  BOXLOOP:OUTBIT←1
[10]  BITLOOP:QM PRIMIMP OUTBIT ON BINARY SBOX[SBOXCNT;;]
[11]  ALLESS←ALLESS,[1] ES
[12]  ALLALT←ALLALT,[1]((1↑⍴AL),10)↑AL
[13]  ALLESS←ALLESS,[1] 6⍴' '
[14]  ALLALT←ALLALT,[1] 10⍴' '
[15]  ALLPTR←ALLPTR,[1](1↑⍴ALLESS),1↑⍴ALLALT
[16]  →((OUTBIT←OUTBIT+1)≤4)/BITLOOP
[17]  →((SBOXCNT←SBOXCNT+1)≤8)/BOXLOOP
      ∇


      ∇  Z←BINARY DECBOX
[1]   ⍝ TO CONVERT AN SBOX INTO BINARY FORM
[2]   Z← 3 1 2 ⍉ 2 2 2 2 ⊤DECBOX
      ∇


      ∇  Z←ONETERM CONTRIB BOXOUT;SP;ONFOR;OFFOR
[1]   ⍝ TO DETERMINE CONTRIBUTION TO OVERALL BOX OF ONE SP TERM
[2]   ⍝
[3]   SP←(32 6)⍴ONETERM
[4]   ONFOR←BOXOUT[2] ON BINARY SBOX[BOXOUT[1];;]
[5]   OFFOR←⍉(∧⌿~ONFOR∧.=ALL)/ALL←(6⍴2)⊤¯1+⍳64
[6]   Z←+/∧/(SP='X')∨(SP='0')≠ONFOR
[7]   Z←Z++/~∧/(SP≠'X')∧(SP='0')≠OFFOR
[8]   Z←Z÷64
      ∇
```

```
        ∇ DUMPONS;BOX;BIT;TERM;ONTERMS;T
[ 1]    DUM←100 ⎕SVO 'TSO'
[ 2]    TSO←'ALLOC DA(DES.ONFOR) OLD FILE(ONFOR)'
[ 3]    OUT←'ONFOR(APL)'
[ 4]    CTL←'ONFOR(CTL)'
[ 5]    111 ⎕SVO 2 3 ⍴'OUTCTL'
[ 6]    DUM←OUT
[ 7]    BOX←1
[ 8]    BOXLOOP:BIT←1
[ 9]    ONTERMS←BIT ON BINARY SBOX[BOX;;]
[10]    BITLOOP:TERM←1
[11]    TERMLOOP:OUT←(T≠' ')/T←⍕ONTERMS[TERM;]
[12]    →((TERM←TERM+1)≤32)/TERMLOOP
[13]    →((BIT←BIT+1)≤4)/BITLOOP
[14]    →((BOX←BOX+1)≤8)/BOXLOOP
[15]    DUM←⎕SVR 'OUT'
[16]    TSO←'FREE F(ONFOR)'
        ∇


        ∇ Z←DUPKILL MAT;T
[1]     ⍝ REMOVES ANY DUPLICATE ROWS FROM MAT
[2]     Z←(KILL←∧/(MAT∨.≠⍉MAT)∨T∘.≤T←⍳1↑⍴MAT)⌿MAT
        ∇


        ∇ Z←ESS BOXINP;IND
[1]     ⍝ GIVEN BOX NUMBER AND OUTPUT, RETURNS TABLE OF ESSENTIAL SP
'S
[2]     IND←BOXINP[2]+4×¯1+BOXINP[1]
[3]     Z←(~∧/' '=Z)⌿Z←ALLESS[IOTA,ALLPTR[IND+ 0 1 ;1];]
        ∇


        ∇ Z←MATRIX FINDCOORDS SUBSTRING;MATCH;COORD
[1]     ⍝ FIND MTX OF COORDS, ONE ROW FOR EACH OCCURENCE OF SUBSTRIN
G
[2]     ⍝ IN THE ROWS OF MATRIX
[3]     COORD←∧⌿(⍉(⍉(⍉1↑⍴MATCH)⍴⍳⎕IO-1⍴,SUBSTRING)⌽MATCH←(,SUBSTRING)
∘.=MATRIX
[4]     Z←⎕IO+⍉(⍴COORD)⊤⍳⎕IO-(,COORD)/⍳⍴,COORD
        ∇
```

```
      ∇  Z←IOTA X
[1]   ⍝ FOR ENUMERATION OF INCLUSIVELY BOUNDED INTEGER LIST
[2]   Z←X[1]+¯1+⍳1+|-/X
      ∇


      ∇  Z←PI MC2 PIT;E;A;M;V;C;NC;NZ
[1]   ⍝ THIS FUNCTION FINDS A MINIMAL COVER FROM A PRIME IMPLICAN
T TABLE.
[2]   Z←(0,N)⍴0
[3]   L1:Z←Z,[1](E←PIT∨.∧1=+/[1] PIT)/[1] PI
[4]   →(~∨/E)/PET
[5]   →(0=×/⍴PIT←(~∨/[1] E/[1] PIT)/(~E)/[1] PIT)/0
[6]   PI←(~E)/[1] PI
[7]   PIT←(~∨/[1] A∧~((⍳M)∘.≥⍳M←¯1↑⍴PIT)∧A∧⍉A←(~⍉PIT)∧.∨PIT)/PIT
[8]   PIT←(V←∨/PIT)/[1] PIT
[9]   PI←V/[1] PI
[10]  V←~∨/A∧~((⍳M)∘.≥⍳M←1↑⍴PIT)∧A∧⍉A←(V∘.≥V←+/2>PI)∧(~PIT)∧.∨⍉PI
T
[11]  PIT←V/[1] PIT
[12]  →L1,0⍴PI←V/[1] PI
[13]  PET:V←((,PIT[;1↑⍋+/[1] PIT])/⍳M←1↑⍴PIT),0⍴C←100
[14]  SL:→(C≤NC←+/+/2>E←PI MC2 PIT,V[1]=⍳M)/EL
[15]  C←NC,0⍴NZ←E
[16]  EL:→(0≠⍴V←1↓V)/SL
[17]  Z←Z,[1] NZ
      ∇


      ∇  Z←BITPOS ON S
[1]   ⍝ GIVEN AN SBOX AND OUTPUT BIT POSITION, RETURN THE 32×6
[2]   ⍝ MATRIX OF BINARY INPUTS FOR WHICH THE OUTPUT BIT IS 1
[3]   Z←(16 16)⊤Z←¯1+(,Z)/⍳×/⍴Z←S[;;BITPOS]
[4]   Z← 3 1 2 ⍉ 2 2 2 2 ⊤Z
[5]   Z←Z[1;;3],Z[2;;],Z[1;;4]
      ∇
```

```
      ∇  PIM T2;C;D;NXT;V;T
[1]   ⍝ TAB IS TABLE OF MINTERMS IN BINARY
[2]   ⍝ N IS NUMBER OF VARIABLES IN FN, PASSED GLOBALLY
[3]   ⍝ THIS FUNCTION FINDS THE PRIME IMPLICANTS OF A TABLE OF MIN
TERMS.
[4]   PI←(0,N)⍴0,0⍴T←T2
[5]   L1:NXT←(0,N)⍴0,V←(1↑⍴T)⍴0
[6]   L2:A←(~∨/((⍳D)∘.>⍳D←1↑⍴A)∧A∧.=⍉A)/[1] A←(2×C/[1] D)⌈(C←1=+/D
←T≠(⍴T)⍴T[1;])/[1] T
[7]   NXT←NXT,[1](~∨/[1] NXT∧.=⍉A)/[1] A
[8]   →(0≠⍴V←1↓V∨C,(0⍴T← 1 0 ↓T),0⍴PI←PI,[1](((~1↑V)∧0=+/C),N)⍴T[1
;])/L2
[9]   →(0≠1↑⍴T←NXT)/L1
[0]   PIT←⍉((~T2)∧.∨⍉PI≠0)∧T2∧.∨⍉PI≠1
      ∇
```

```
      ∇  Z←PRIMIMP IN;T;BND;K;P1;P2;CMATCH;NOPREVMATCH;NEWIN
[ 1]  ⍝
[ 2]  ⍝ FOR QUINE+MCCLUSKEY MINIMIZATION OF BOOLEAN FUNCTIONS:
[ 3]  ⍝ RETURNS PRIME IMPLICANTS OF GIVEN N×6 INPUT MATRIX
[ 4]  Z←(0,¯1↑⍴IN)⍴0
[ 5]  ⍝ NUMON IS VECTOR PARALLEL TO IN MTX, INDICATING NUMBER OF
[ 6]  ⍝ BITS ON IN IN ENTRIES, IN SORTED ⍋ IN BITS ON
[ 7]  ⍝ BND IS MTX INDICATING DIVISIONS BETWEEN K AND K+1 BITS ON
      ⋄
[ 8]  ITER:IN←IN[T←⍋NUMON←+/IN=1;]
[ 9]  T←(NUMON≠1⌽NUMON)/⍳⍴BND←NUMON←NUMON[T]
[10]  BND←(2,⍴BND)⍴1
[11]  →(0=⍴T)/JUST1CLASS
[12]  BND←⍪(1,1+¯1↑T),[0.5] T
[13]  JUST1CLASS:NEWIN←(0,¯1↑⍴IN)⍴0
[14]  K←1
[15]  ⍝
[16]  NOPREVMATCH←(⍴IOTA,BND[1;])⍴1
[17]  MATCHLOOP:P1←IN[IOTA,BND[K;];]
[18]  P2←IN[IOTA,BND[K+1;];]
[19]  MATCH←1=P1+.≠⍪P2
[20]  ⍝ ANY ROW OF P1 W/ NO MATCHES IS A PRIME IMPLICANT:
[21]  Z←Z,[1](NOPREVMATCH∧~∨/MATCH)⌿P1
[22]  C←MATCH FINDCOORDS 1
[23]  NOPREVMATCH←~∨⌿MATCH
[24]  NEWIN←NEWIN,[1](¯1 ¯1 0 ¯1 1)[3+P1[C[;1];]+P2[C[;2];]]
[25]  →((K←K+1)<1↑⍴BND)/MATCHLOOP
[26]  ⍝ ADD UNMATCHED ELTS OF LAST P2 TO RESULT
[27]  Z←Z,[1] NOPREVMATCH⌿P2
[28]  →(1<1↑⍴IN←NEWIN)/ITER
[29]  ⍝ ADD LAST POSSIBLE IMPLICANT FROM P2:
[30]  Z←Z,[1] P2[1↑⍴P2;]
[31]  ⍝ REMOVE POTENTIAL DUPLICATION:
[32]  Z←DUPKILL Z
      ∇


      ∇  Z←SP PROB∆CORR ONFOR
[1]   Z←0
[2]   K←1
[3]   LOOP:Z←Z+∨/∧/(SP='X')∨(SP='0')≠(⍴SP)⍴ONFOR[K;]
[4]   →((K←K+1)≤1↑⍴ONFOR)/LOOP
[5]   Z←Z÷1↑⍴ONFOR
      ∇
```

```
      ∇  Z←QM MIN;N
[1]   ⍝ QM MINIMIZATION, GIVEN MINTERM NUMBERS IN DECIMAL
[2]   PIM⌹((N←⌊1+2⊛MIN[1↑⍒MIN])⍴2)⊤MIN
[3]   Z←'01X'[1+PI MC2 PIT]
      ∇


      ∇  REDUCE;SBOXCNT;OUTBIT
[ 1]  ⍝
[ 2]  ⍝ TO FORM THE GLOBAL 4D MAT SPTERMS (8×4×30×6), WHICH GIVES
[ 3]  ⍝ MINIMAL SP FORMS FOR EACH SBOX AND OUTPUT BIT, USING REDU
CE∆ALTS
[ 4]  ⍝
[ 5]  SPTERMS← 8 4 30 6 ⍴' '
[ 6]  SBOXCNT←1
[ 7]  BOXLOOP:OUTBIT←1
[ 8]  BITLOOP:SPTERMS[SBOXCNT;OUTBIT;;]← 30 6 ↑REDUCE∆ALTS SBOXCN
T,OUTBIT
[ 9]  →((OUTBIT←OUTBIT+1)≤4)/BITLOOP
[10]  →((SBOXCNT←SBOXCNT+1)≤8)/BOXLOOP
[11]  SPTERMS←(8 4 ,(+/∨≠∨/∨≠SPTERMS≠' '),6)↑SPTERMS
      ∇
```

```
      ∇ Z←REDUCE∆ALTS BOXOUT;A;T;CLASS;TAB;FREQ;MOSTFREQ;KILL;KI
LLPOS;CLASS;GRP
[ 1]    ⍝
[ 2]    ⍝ TO PICK ALTERNATIVES FOR ACCURATE SP EXPRESSION FOR SBOX
[ 3]    ⍝ 1) CHOOSE TERMS MOST FREQUENTLY APPEARING INTER-CLASS
[ 4]    ⍝ 2) PICK CLASS REP. AS REMAINING TERM WITH MOST DC'S
[ 5]    ⍝
[ 6]    CLASS←⍳⍴ 0 7 ↓A←ALT BOXOUT
[ 7]    TAB←T∧.=⍉T←A[⍳1↑⍴A;⍳6]
[ 8]    REDUCE∆LOOP:→(1=MOSTFREQ←⌈/FREQ←+/TAB)/OUT
[ 9]    KILLPOS←(T∧.=⍉T[FREQ⍳MOSTFREQ;])/⍳1↑⍴T
[10]    KILL←(~CLASS∈CLASS[KILLPOS])∨(⍳⍴CLASS)=1↑KILLPOS
[11]    TAB←KILL/KILL⌿TAB
[12]    T←KILL⌿T
[13]    CLASS←KILL/CLASS
[14]    →REDUCE∆LOOP
[15]    OUT: ⍝ SELECT REPRESENTATIVE FROM EACH CLASS
[16]    Z←ESS BOXOUT
[17]    SELECT∆LOOP:NUMDCS←+/'X'=GRP←(X←CLASS=1↑CLASS)⌿T
[18]    Z←Z,[1] GRP[NUMDCS⍳⌈/NUMDCS;]
[19]    T←(~X)⌿T
[20]    CLASS←(~X)/CLASS
[21]    →(0≠⍴CLASS)/SELECT∆LOOP
        ∇


      ∇ SELECT∆SP BOXOUT;AL;ES
[1]     ⍝
[2]     ⍝ TO SELECT THE FIRST PRECISE SP EXPRESSION FOR GIVEN SBOX
[3]     ⍝ AND OUTPUT FROM QM MINIMIZED EXPRESSION
[4]     ES←ESS BOXOUT
[5]     AL←ALT BOXOUT
[6]     0/DUPKILL 0 7 ↓AL
[7]     SP←ES,[1] KILL⌿AL[⍳1↑⍴AL;⍳6]
[8]     SP←(~∧/' '=SP)⌿SP
        ∇
```

```
     ∇ Z←SELECT∆ALTS;T;CLASS;TAB;FREQ;MOSTFREQ;KILL;KILLPOS;CLASS;GRP
[ 1]   ⍝
[ 2]   ⍝ TO PICK ALTERNATIVES FOR ACCURATE SP EXPRESSION FOR SBOX
[ 3]   ⍝ 1) CHOOSE TERMS MOST FREQUENTLY APPEARING INTER-CLASS
[ 4]   ⍝ 2) PICK CLASS REP. AS REMAINING TERM WITH MOST DC'S
[ 5]   ⍝
[ 6]   →(0=1↑⍴AL)/NOALTS
[ 7]   CLASS←⍳⍪0 7↓AL
[ 8]   TAB←T∧.=⍉T←AL[⍳1↑⍴AL;⍳6]
[ 9]   REDUCE∆LOOP:→(1=MOSTFREQ←⌈/FREQ←+/TAB)/OUT
[10]   KILLPOS←(T∧.=⍉T[FREQ⍳MOSTFREQ;])/⍳1↑⍴T
[11]   KILL←(~CLASS∈CLASS[KILLPOS])∨(⍳⍴CLASS)=1↑KILLPOS
[12]   TAB←KILL/KILL≠TAB
[13]   T←KILL≠T
[14]   CLASS←KILL/CLASS
[15]   →REDUCE∆LOOP
[16]   OUT: ⍝ SELECT REPRESENTATIVE FROM EACH CLASS
[17]   Z←ES
[18]   SELECT∆LOOP:NUMDCS←+/'X'=GRP←(X←CLASS=1↑CLASS)≠T
[19]   Z←Z,[1] GRP[NUMDCS⍳⌈/NUMDCS;]
[20]   T←(~X)≠T
[21]   CLASS←(~X)/CLASS
[22]   →(0≠⍴CLASS)/SELECT∆LOOP
[23]   'NUMBER OF P-TERMS: ',▼1↑⍴Z
[24]   →0
[25]   NOALTS:Z←ES
       ∇
```

```
      ∇  SPDUMP;BOX;BIT;TERM
[ 1]  DUM←100 ⎕SVO 'TSO'
[ 2]  TSO←'ALLOC DA(DES.SPTERMS) OLD FILE(SPF)'
[ 3]  OUT←'SPF(APL)'
[ 4]  CTL←'SPF(CTL)'
[ 5]  111 ⎕SVO 2 3 ⍴'OUTCTL'
[ 6]  DUM←OUT
[ 7]  BOX←1
[ 8]  BOXLOOP:BIT←1
[ 9]  BITLOOP:TERM←1
[10]  TERMLOOP:OUT←SPTERMS[BOX;BIT;TERM;]
[11]  →((TERM←TERM+1)≤23)/TERMLOOP
[12]  →((BIT←BIT+1)≤4)/BITLOOP
[13]  →((BOX←BOX+1)≤8)/BOXLOOP
[14]  DUM←⎕SVR 'OUT'
[15]  TSO←'FREE F(SPF)'
      ∇
```

END OF APPENDIX

APPENDIX C

PL/I CODE AND OUTPUT FOR COMBINATORIALLY-EXHAUSTIVE APPROACH TO BEST-SET DISCOVERY.

```
/* BEST-TERM SELECTION FOR S-BOX APPROXIMATION  |  E.GULLICHSEN  */
/*****************************************************************/
/*                                                               */
/* CONTRIB PROC OPERATES IN 3 PHASES:                            */
/*   1) FORM ARRAY CONTRIB(23,32), INDICATING HOW EACH TERM,     */
/*      CONSISTING OF A CONJUNCT OF S-BOX INPUTS 'COVERS' THE 32 */
/*      TERMS FOR WHICH THE SBOX SHOULD BE ON.                   */
/*   2) USING PROC CHOOSE TO TRY ALL COMBINATIONS OF S.P. TERMS, */
/*      BUILD THE ARRAY COVER(5000,23) CONTAINING INDICATION OF  */
/*      WHICH S.P. TERMS TO TAKE, TO GET BEST APPROX. TO S-BOX, IF*/
/*      RESTRICTED TO 1,2,...,23 TERMS                           */
/*   3) SEARCH THE COVER TABLE, TO SEE IF THE SET OF 'BEST N'    */
/*      TERMS IS A SUBSET OF BEST N-1 TERMS                      */
/*****************************************************************/

CONTRIB: PROC OPTIONS(MAIN);
  DCL (NUM_TERMS,TERMCNT,INPCNT,LEVEL,DIFF,NEXTERM) FIXED BIN(15);
  DCL TERMS(23,6) CHAR(1); /* S.P. TERMS AS READ FROM FILE */
  DCL ONFOR(32,6) BIT(1);  /* INPUTS FOR WHICH S-BOX SHOULD BE ON */
  DCL COVER(5000,23) BIT(1);
  DCL COVERPROB(23) FIXED BIN(15);
  DCL (COVERPTS,SEARCHPTS)(24) FIXED BIN(15);
  DCL CURR(23) BIT(1);
  DCL XOR_RES(6) BIT(1);
  DCL AND RETURNS(BIT(1)); /* ANDING ROUTINE */
  DCL ZERO6(6) CHAR(1) INIT((6) (1) '0');
  DCL XSTR6(6) CHAR(1) INIT((6) (1) 'X');
  DCL (GO,FAIL) BIT(1); /* LOOP FLAGS */
  DCL CUMUL_TERMS(23) BIT(1);

  COVER(*,*)='0'B;
  /* TITLE */
  PUT SKIP FILE(SPRINT)
   EDIT('BEST-TERM SELECTION FOR S-BOX APPROXIMATION',(43)'*')
   (2 (A,SKIP));

  /* READ IN STUFF FOR S1, OUTPUT 1 FROM FILES */
  CALL READIN;

CONTRIB_CALC: BEGIN;
    /* CREATE TABLE OF CONTRIBUTIONS, INDICATING WHICH TERMS ARE
       ON FOR WHICH INPUTS    */

    DCL CONTRIB(NUM_TERMS,32) BIT(1);

    DO INPCNT=1 TO 32;
       DO TERMCNT=1 TO NUM_TERMS;

          /* SINGLE S.P. TERM IS A CONJUNCT, HENCE 'ALL'.
             FLIP BITS WHICH CORR. TO A 0 IN THE TERM;
             OR WITH 1'S FOR DC'S (X'S) IN TERMS  */

          CALL XOR(ONFOR(INPCNT,*) , (ZERO6 = TERMS(TERMCNT,*))
                    , XOR_RES);

          CONTRIB(TERMCNT,INPCNT)=
           AND( (XSTR6 = TERMS(TERMCNT,*)) | XOR_RES);

       END;
    END;
```

```
/* DUMP THE CONTRIBUTION TABLE */
PUT PAGE FILE(SPRINT) LIST('    *** CONTRIBUTION TABLE ***');
PUT SKIP(2) FILE(SPRINT) EDIT('TERM IS ON FOR INPUT','SP TERM',
  (I DO I=1 TO 32),(7)'*',(96)'*')
  (COL(40),A,SKIP,A,X(2),32 F(3),SKIP,A,X(2),A);
PUT SKIP FILE(SPRINT) EDIT((TERMS(I,*),CONTRIB(I,*)
          DO I=1 TO NUM_TERMS))(6 A(1),X(5),32 B(3),SKIP);

    /* CREATE COVER TABLE */

    COVERPTS(1)=1;
    PUT PAGE FILE(SPRINT) LIST('    *** COVER TABLE ANALYSIS ***');
    PUT SKIP(2) FILE(SPRINT) EDIT('# SPTERMS','# OF BEST SETS',
      'CORRECTNESS',(9)'*',(14)'*',(11)'*')
                          (2 (A,COL(15),A,COL(35),A,SKIP));

    DO TERMCNT=1 TO NUM_TERMS;

COVER_CALC: BEGIN;
       DCL (NEW,OLD)(TERMCNT) FIXED BIN(15); /* SELECTION INDICES */
       DCL NUMON              FIXED BIN(15);
       DCL MOST  INIT(0)      FIXED BIN(15); /* MAX OF ABOVE */
       DCL SAVE(3000,TERMCNT) FIXED BIN(15);
       DCL SAVEPTR INIT(1)    FIXED BIN(15);
       DCL TEMPBITS(32)       BIT(1);

       /* SETUP OLD FOR 1ST CALL TO CHOOSE */
       DO I=1 TO TERMCNT; NEW(I)=I; END;

       DO WHILE(NEW(1) ¬= 0);
          /* CALC SUM OF OR OF TERMS CHOSEN */
             TEMPBITS(*)='0'B;
             DO I=1 TO TERMCNT;
                TEMPBITS(*)=TEMPBITS(*) | CONTRIB(NEW(I),*);
             END;

             NUMON=0;
             DO I=1 TO 32;
                IF TEMPBITS(I) THEN NUMON=NUMON+1;
             END;

             IF NUMON > MOST THEN DO;
                MOST=NUMON;
                SAVE(1,*)=NEW(*);
                SAVEPTR=2;
             END;
             ELSE IF NUMON = MOST THEN DO;
                SAVE(SAVEPTR,*)=NEW(*);
                SAVEPTR=SAVEPTR+1;
             END;
         OLD=NEW;
          CALL CHOOSE(NUM_TERMS,TERMCNT,OLD,NEW);
       END;

/* PUT THE BEST COMBINATIONS (AS SAVED IN THE SAVE ARRAY) INTO
   THE COVER TABLE */


     COVERPROB(TERMCNT)=MOST;
```

```
      DO I=0 TO SAVEPTR-2;
         DO J=1 TO TERMCNT;

            COVER(COVERPTS(TERMCNT)+I,SAVE(I+1,J)) = '1'B;
         END;
      END;

      PUT SKIP FILE(SPRINT) EDIT(TERMCNT,SAVEPTR-1,.5+MOST/64.)
                        (X(2),F(4),COL(15),F(5),COL(38),F(7,3));

      COVERPTS(TERMCNT+1)=COVERPTS(TERMCNT)+SAVEPTR-1;
  END COVER_CALC;
 END;

 /* SEARCH COVER TABLE, TO DET. MONOTONICITY */

PUT SKIP(3) FILE(SPRINT) LIST('     *** COVER TABLE ***');
J=2;
DO I=1 TO COVERPTS(NUM_TERMS+1)-1;
  PUT SKIP FILE(SPRINT) EDIT(I,')',COVER(I,*))
                        (F(5),A,X(2),23 B(1));
  /* SKIP LINES BETWEEN 'GROUPS' */
  IF COVERPTS(J)-1=I THEN DO;
     PUT SKIP FILE(SPRINT);
     J=J+1;
  END;
END;

 SEARCHPTS(*)=COVERPTS(*);
 LEVEL=1;
 FAIL='0'B;
 PUT PAGE FILE(SPRINT) EDIT('COVER TABLE SEARCH',(18)'*')
                          (2 (A,SKIP));
 DO WHILE(LEVEL < NUM_TERMS & ¬FAIL);
   PUT SKIP(2) FILE(SPRINT) EDIT('CONSIDERING ROW ',SEARCHPTS(LEVEL),
                                ' AT LEVEL ',LEVEL)(A,F(5),A,F(3));
    CURR(*)=COVER(SEARCHPTS(LEVEL),*);
    LEVEL=LEVEL+1;
    GO='1'B;

    DO I=COVERPTS(LEVEL) TO COVERPTS(LEVEL+1)-1 WHILE(GO);
       /* IF DIFF BETWEEN CURR AND COVER(I,*) EXACTLY ON BIT, THEN
          WE HAVE GOT THE TERM NEEDED IN THIS NEXT LEVEL: */
       DIFF=0;
       DO J=1 TO NUM_TERMS;
          IF CURR(J) ¬= COVER(I,J) THEN DIFF=DIFF+1;
       END;
       IF DIFF = 1 THEN GO='0'B;/* QUIT, WE HAVE MATCH AT LEVEL */
    END;

    /* IF NONE FOUND AT THIS LEVEL, THEN BACKTRACK */
    IF GO THEN DO;
  PUT SKIP FILE(SPRINT) EDIT('NO MATCH FOR CURR AT LEVEL ',LEVEL)
                           (A,F(3));
       LEVEL=LEVEL-1;
       SEARCHPTS(LEVEL)=SEARCHPTS(LEVEL)+1;
       IF SEARCHPTS(LEVEL)=COVERPTS(LEVEL+1) THEN DO;
     PUT SKIP(2) FILE(SPRINT) EDIT('** LEVEL ',LEVEL,' EXHAUSTED')
                                 (A,F(3),A);
         DO I=LEVEL TO 1 BY -1 WHILE(SEARCHPTS(I)+1 >= COVERPTS(I+1));
```

```
                SEARCHPTS(I)=COVERPTS(I);
            END;
            LEVEL=I+1;
        PUT SKIP FILE(SPRINT) EDIT('BACKTRACK TO LEVEL ',LEVEL)(A,F(3));
            SEARCHPTS(LEVEL)=SEARCHPTS(LEVEL)+1;
        /* IF FAILED ALL THE WAY BACK TO 1ST LEVEL */
            IF LEVEL=1 & SEARCHPTS(LEVEL) = COVERPTS(LEVEL+1) THEN
                    DO;

                    FAIL='1'B;
  PUT PAGE FILE(SPRINT) LIST('*** SEARCH FAILED ***');
  PUT FILE(SPRINT) DATA(SEARCHPTS,COVERPTS,CURR,COVERPROB);
  PUT SKIP(3) FILE(SPRINT) LIST('COVER TABLE');
  DO I=1 TO COVERPTS(NUM_TERMS);
    PUT SKIP FILE(SPRINT) EDIT(I,COVER(I,*))(F(6),X(2),23 B(1));
  END;
                    END;

        END;
    END;

    /* ELSE CURR DID MATCH */
    ELSE DO;
         SEARCHPTS(LEVEL)=I-1;
         PUT SKIP FILE(SPRINT) EDIT('CURR MATCHES WITH COVER TERM ',
             I-1,' AT LEVEL ',LEVEL)(A,F(5),A,F(3));

         END;
END;

/* SEARCH WAS SUCCESSFUL, PRINT THE RANKED TERMS */
IF ¬FAIL THEN DO;
    PUT PAGE FILE(SPRINT) LIST('*** SEARCH SUCCESSFUL ***');
    PUT SKIP(2) FILE(SPRINT) EDIT('# TERMS INCLUDED',
        'TERM SELECTION','CORRECTNESS',(16)'*',(14)'*',(11)'*')
        (2 (A,COL(22),A,COL(45),A,SKIP));
    DO I=1 TO NUM_TERMS;
     PUT SKIP FILE(SPRINT) EDIT(I,COVER(SEARCHPTS(I),*),
          .5+COVERPROB(I)/64.)
          (F(8),COL(18),23 B(1),COL(48),F(7,3));
    END;

    PUT SKIP(2) FILE(SPRINT)
    EDIT('TERMS IN THE ORDER OF THEIR ADDITION:')(A);
    CUMUL_TERMS(*)='0'B;/* ACCUM BIT POSNS FROM COVER */
    DO I=1 TO NUM_TERMS;
       NEXTERM=0;
       DO J=1 TO 23 WHILE(NEXTERM=0);
          IF CUMUL_TERMS(J)='0'B & COVER(SEARCHPTS(I),J)='1'B
                        THEN NEXTERM=J;
       END;
       PUT SKIP FILE(SPRINT) EDIT(TERMS(NEXTERM,*))(X(8),6 A(1));
       CUMUL_TERMS(*)=CUMUL_TERMS(*) | COVER(SEARCHPTS(I),*);
    END;

END;

END CONTRIB_CALC;
```

```
/*****************************************************/
/* XOR PROC, TO  EXCLUSIVE-OR 2 BINARY VECTORS       */
/*****************************************************/

XOR: PROC(A,B,RES);

  DCL (A,B,RES)(*) BIT(1);
  RES=(A|B)&(¬(A&B));
END XOR;

/*****************************************************/
/* AND, RETURNS 1 IFF ALL ALTS IN ARGUMENT ARRAY ARE 1 */
/*****************************************************/

AND: PROC(BOOL_VEC) RETURNS(BIT(1));

  DCL (RES,BOOL_VEC(*)) BIT(1);
  DCL I FIXED BIN(15);
  RES='1'B;
  DO I=1 TO HBOUND(BOOL_VEC,1) WHILE(RES);
     IF ¬BOOL_VEC(I) THEN RES='0'B;
  END;
  RETURN(RES);
END AND;

/*********************************************************************/
/*                                                                   */
/* INPUT ROUTINE, TO READ THE S.P. TERMS FOR A GIVEN S-BOX, OUTPUT,  */
/* AND THE 32 INPUTS FOR WHICH THE OUTPUT IS 1, FROM SEQUENTIAL      */
/* MTS FILES "SP" AND "ON", REPECTIVELY                              */
/*                                                                   */
/*********************************************************************/

READIN:PROC;
  DCL (SPFILE,ONFILE) FILE STREAM;
  CALL ATTACH('SPFILE=SP');
  CALL ATTACH('ONFILE=ON');
  OPEN FILE(ONFILE),FILE(SPFILE);

  NUM_TERMS=0;
  DO I=1 TO 23; /* MAX # S.P. TERMS FOR ANY SBOX, OUTPUT IS 23 */
     GET FILE(SPFILE) EDIT(TERMS(I,*))(6 A(1),SKIP);
     IF TERMS(I,1) ¬= ' ' THEN NUM_TERMS=NUM_TERMS+1;
  END;
  GET FILE(ONFILE) EDIT(((ONFOR(I,J) DO J=1 TO 6) DO I=1 TO 32))
                        (6 B(1),SKIP);

  PUT SKIP FILE(SPRINT) LIST('SP TERMS:');
  PUT SKIP(2) FILE(SPRINT) EDIT((TERMS(I,*) DO I=1 TO NUM_TERMS))
                               (6 A(1),SKIP);
  PUT SKIP(3) FILE(SPRINT) LIST('OUTPUT SHOULD BE ON FOR INPUTS:');
  PUT SKIP(2) FILE(SPRINT) EDIT(ONFOR)(6 B(1),SKIP);

  CLOSE FILE(ONFILE);
  CLOSE FILE(SPFILE);
  END READIN;

/*****************************************************************/
/*                                                               */
```

```
/* THE CHOOSE ROUTINE RETURNS A NEW COMBINATION OF R ITEMS      */
/* CHOSEN FROM A COLLECTION OF N, GIVEN THE PREVIOUS COMB,      */
/* OLD. THE NEW COMBINATION RETURNED IS A VECTOR OF FIXED BIN   */
/* QUANTITIES. E.G., N=5 R=3 OLD=1 2 4, NEW=>1 2 5              */
/*                                                              */
/****************************************************************/

 CHOOSE: PROC(N,R,OLD,NEW);
  DCL(N,R,I,J) FIXED BIN(15);
  DCL(OLD,NEW)(*) FIXED BIN(15);
  DCL GO BIT(1);
  NEW=OLD;
  GO='1'B;
  DO I=R TO 1 BY -1 WHILE(GO);
     /* IF ANY POSN AT ITS MAX, INCREASE PREV POSN */
     IF OLD(I) ¬= N-R+I THEN DO;
        GO='0'B;
        NEW(I)=NEW(I)+1;
        DO J=I+1 TO R;
           NEW(J)=NEW(J-1)+1;
        END;
     END;
  END;
  IF GO THEN NEW(1)=0; /* IF NO MORE COMBS */
 END CHOOSE;

END CONTRIB;
```

BEST-TERM SELECTION FOR S-BOX APPROXIMATION

SP TERMS: for S-box 1, Output 1

X1X010
X1X111
000X00
X00011
X11001
10X011
001X01
001X10
00X100
010X01
010X10
100X01
10Q1XO
X01110
110X00
11001X
101000

OUTPUT SHOULD BE ON FOR INPUTS:

000000
000100
001010
001100
001110
010010
010110
011010
000011
001001
001101
010001
010101
010111
011001
011111
100100
100110
101000
101110
110000
110010
110100
111010
100001
100011
100101
101011
110011
110111
111001
111111

*** CONTRIBUTION TABLE ***

SP TERM	TERM IS ON FOR INPUT 1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
X1X010	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0
X1X111	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1
000X00	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
X00011	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
X11001	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
10X011	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0
001X01	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
001X10	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
00X100	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
010X01	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
010X10	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
100X01	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0
1001X0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
X01110	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
110X00	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0
11001X	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0
101000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0

*** COVER TABLE ANALYSIS ***

# SPTERMS	# OF BEST SETS	CORRECTNESS
1	2	0.563
2	1	0.625
3	12	0.656
4	63	0.688
5	190	0.719
6	363	0.750
7	456	0.781
8	377	0.813
9	198	0.844
10	60	0.875
11	8	0.906
12	36	0.922
13	66	0.938
14	63	0.953
15	33	0.969
16	9	0.984
17	1	1.000

*** COVER TABLE ***

1) 10000000000000000000000
2) 01000000000000000000000

3) 11000000000000000000000

4) 11100000000000000000000
5) 11010000000000000000000
6) 11001000000000000000000
7) 11000100000000000000000
8) 11000010000000000000000
9) 11000001000000000000000
10) 11000000100000000000000
11) 11000000010000000000000
12) 11000000000100000000000
13) 11000000000010000000000
14) 11000000000001000000000
15) 11000000000000100000000

16) 11110000000000000000000
17) 11101000000000000000000
18) 11100100000000000000000
19) 11100010000000000000000
20) 11100001000000000000000
21) 11100000100000000000000
22) 11100000000100000000000
23) 11100000000010000000000
24) 11100000000001000000000
25) 11100000000000100000000
26) 11011000000000000000000
27) 11010010000000000000000
28) 11010001000000000000000
29) 11010000100000000000000
30) 11010000010000000000000
31) 11010000000100000000000
32) 11010000000010000000000
33) 11010000000001000000000

34) 1101000000000001000000000
35) 1100111000000000000000000
36) 1100101000000000000000000
37) 1100100100000000000000000
38) 1100100010000000000000000
39) 1100100001000000000000000
40) 1100100000010000000000000
41) 1100100000000100000000000
42) 1100100000000010000000000
43) 1100100000000001000000000
44) 1100011000000000000000000
45) 1100010100000000000000000
46) 1100010010000000000000000
47) 1100010001000000000000000
48) 1100010000001000000000000
49) 1100010000000100000000000
50) 1100010000000010000000000
51) 1100010000000001000000000
52) 1100001100000000000000000
53) 1100001010000000000000000
54) 1100001001000000000000000
55) 1100001000010000000000000
56) 1100001000001000000000000
57) 1100001000000100000000000
58) 1100001000000010000000000
59) 1100000110000000000000000
60) 1100000101000000000000000
61) 1100000100010000000000000
62) 1100000100001000000000000
63) 1100000100000010000000000
64) 1100000011000000000000000
65) 1100000010010000000000000
66) 1100000010001000000000000
67) 1100000010000100000000000
68) 1100000010000010000000000
69) 1100000001010000000000000
70) 1100000001001000000000000
71) 1100000001000100000000000
72) 1100000001000010000000000
73) 1100000000011000000000000
74) 1100000000010100000000000
75) 1100000000010010000000000
76) 1100000000001100000000000
77) 1100000000001010000000000
78) 1100000000000110000000000

79) 1111100000000000000000000
80) 1111001000000000000000000
81) 1111000100000000000000000
82) 1111000001000000000000000
83) 1111000000010000000000000
84) 1111000000001000000000000
85) 1111000000000100000000000
86) 1111000000000010000000000
87) 1110110000000000000000000
88) 1110101000000000000000000
89) 1110100100000000000000000
90) 1110100001000000000000000
91) 1110100000010000000000000
92) 1110100000001000000000000

1766) 11001110110111101000000

1767) 11111111110110100000000
1768) 11111111011110100000000
1769) 11111111010111100000000
1770) 11111111010110110000000
1771) 11111111010110101000000
1772) 11111110110111100000000
1773) 11111110011111100000000
1774) 11111110010111110000000
1775) 11111110010111101000000
1776) 11111011111110100000000
1777) 11111011110111100000000
1778) 11111011110110110000000
1779) 11111011110110101000000
1780) 11111011011111100000000
1781) 11111011011110110000000
1782) 11111011011110101000000
1783) 11111011010111110000000
1784) 11111011010111101000000
1785) 11111011010110111000000
1786) 11111010111111100000000
1787) 11111010110111110000000
1788) 11111010110111101000000
1789) 11111010011111110000000
1790) 11111010011111101000000
1791) 11111010010111111000000
1792) 11101111111110100000000
1793) 11101111110111100000000
1794) 11101111110110110000000
1795) 11101111110110101000000
1796) 11101111011111100000000
1797) 11101111011110110000000
1798) 11101111011110101000000
1799) 11101111010111110000000
1800) 11101111010111101000000
1801) 11101111010110111000000
1802) 11101110111111100000000
1803) 11101110110111110000000
1804) 11101110110111101000000
1805) 11101110011111110000000
1806) 11101110011111101000000
1807) 11101110010111111000000
1808) 11011111111110100000000
1809) 11011111110111100000000
1810) 11011111110110110000000
1811) 11011111110110101000000
1812) 11011110111111100000000
1813) 11011110110111110000000
1814) 11011110110111101000000
1815) 11011011111111100000000
1816) 11011011111110110000000
1817) 11011011111110101000000
1818) 11011011110111110000000
1819) 11011011110111101000000
1820) 11011011110110111000000
1821) 11011010111111110000000
1822) 11011010111111101000000
1823) 11011010110111111000000
1824) 11001111111111100000000

1825) 11001111111110110000000
1826) 11001111111110101000000
1827) 11001111110111110000000
1828) 11001111110111101000000
1829) 11001111110110111000000
1830) 11001110111111110000000
1831) 11001110111111101000000
1832) 11001110110111111000000

1833) 11111111111110100000000
1834) 11111111110111100000000
1835) 11111111110110110000000
1836) 11111111110110101000000
1837) 11111111011111100000000
1838) 11111111011110110000000
1839) 11111111011110101000000
1840) 11111111010111110000000
1841) 11111111010111101000000
1842) 11111111010110111000000
1843) 11111110111111100000000
1844) 11111110110111110000000
1845) 11111110110111101000000
1846) 11111110011111110000000
1847) 11111110011111101000000
1848) 11111100101111111000000
1849) 11111011111111100000000
1850) 11111011111110110000000
1851) 11111011111110101000000
1852) 11111011110111110000000
1853) 11111011110111101000000
1854) 11111011110110111000000
1855) 11111011011111110000000
1856) 11111011011111101000000
1857) 11111011011110111000000
1858) 11111010101111111000000
1859) 11111010111111110000000
1860) 11111010111111101000000
1861) 11111010110111111000000
1862) 11111010011111111000000
1863) 11101111111111100000000
1864) 11101111111110110000000
1865) 11101111111110101000000
1866) 11101111110111110000000
1867) 11101111110111101000000
1868) 11101111110110111000000
1869) 11101111011111110000000
1870) 11101111011111101000000
1871) 11101111011110111000000
1872) 11101111010111111000000
1873) 11101110111111110000000
1874) 11101110111111101000000
1875) 11101110110111111000000
1876) 11101110011111111000000
1877) 11011111111111100000000
1878) 11011111111111010110000000
1879) 11011111111111010101000000
1880) 11011111110111110000000
1881) 11011111110111101000000
1882) 11011111110110111000000
1883) 11011110111111110000000

1884) 11011110111111101000000
1885) 11011110110111111000000
1886) 11011011111111110000000
1887) 11011011111111101000000
1888) 11011011111110111000000
1889) 11011011110111111000000
1890) 11011010111111111000000
1891) 11001111111111110000000
1892) 11001111111111101000000
1893) 11001111111110111000000
1894) 11001111110111111000000
1895) 11001110111111111000000

1896) 11111111111111100000000
1897) 11111111111110110000000
1898) 11111111111110101000000
1899) 11111111110111110000000
1900) 11111111110111101000000
1901) 11111111110110111000000
1902) 11111111011111110000000
1903) 11111111011111101000000
1904) 11111111011110111000000
1905) 11111111010111111000000
1906) 11111110111111110000000
1907) 11111110111111101000000
1908) 11111110110111111000000
1909) 11111110011111111000000
1910) 11111011111111110000000
1911) 11111011111111101000000
1912) 11111011111110111000000
1913) 11111011110111111000000
1914) 11111011011111111000000
1915) 11111010111111111000000
1916) 11101111111111110000000
1917) 11101111111111101000000
1918) 11101111111110111000000
1919) 11101111110111111000000
1920) 11101111011111111000000
1921) 11101110111111111000000
1922) 11011111111111110000000
1923) 11011111111111101000000
1924) 11011111111110111000000
1925) 11011111110111111000000
1926) 11011110111111111000000
1927) 11011011111111111000000
1928) 11001111111111111000000

1929) 11111111111111110000000
1930) 11111111111111101000000
1931) 11111111111110111000000
1932) 11111111110111111000000
1933) 11111111011111111000000
1934) 11111110111111111000000
1935) 11111011111111111000000
1936) 11101111111111111000000
1937) 11011111111111111000000

1938) 11111111111111111000000

```
COVER TABLE SEARCH
******************

CONSIDERING ROW      1 AT LEVEL    1
CURR MATCHES WITH COVER TERM      3 AT LEVEL    2

CONSIDERING ROW      3 AT LEVEL    2
CURR MATCHES WITH COVER TERM      4 AT LEVEL    3

CONSIDERING ROW      4 AT LEVEL    3
CURR MATCHES WITH COVER TERM     16 AT LEVEL    4

CONSIDERING ROW     16 AT LEVEL    4
CURR MATCHES WITH COVER TERM     79 AT LEVEL    5

CONSIDERING ROW     79 AT LEVEL    5
CURR MATCHES WITH COVER TERM    269 AT LEVEL    6

CONSIDERING ROW    269 AT LEVEL    6
CURR MATCHES WITH COVER TERM    632 AT LEVEL    7

CONSIDERING ROW    632 AT LEVEL    7
CURR MATCHES WITH COVER TERM   1088 AT LEVEL    8

CONSIDERING ROW   1088 AT LEVEL    8
CURR MATCHES WITH COVER TERM   1465 AT LEVEL    9

CONSIDERING ROW   1465 AT LEVEL    9
CURR MATCHES WITH COVER TERM   1663 AT LEVEL   10

CONSIDERING ROW   1663 AT LEVEL   10
CURR MATCHES WITH COVER TERM   1723 AT LEVEL   11

CONSIDERING ROW   1723 AT LEVEL   11
CURR MATCHES WITH COVER TERM   1731 AT LEVEL   12

CONSIDERING ROW   1731 AT LEVEL   12
CURR MATCHES WITH COVER TERM   1767 AT LEVEL   13

CONSIDERING ROW   1767 AT LEVEL   13
CURR MATCHES WITH COVER TERM   1833 AT LEVEL   14

CONSIDERING ROW   1833 AT LEVEL   14
CURR MATCHES WITH COVER TERM   1896 AT LEVEL   15

CONSIDERING ROW   1896 AT LEVEL   15
CURR MATCHES WITH COVER TERM   1929 AT LEVEL   16

CONSIDERING ROW   1929 AT LEVEL   16
CURR MATCHES WITH COVER TERM   1938 AT LEVEL   17
```

*** SEARCH SUCCESSFUL ***

# TERMS INCLUDED	TERM SELECTION	CORRECTNESS
1	10000000000000000000000	0.563
2	11000000000000000000000	0.625
3	11100000000000000000000	0.656
4	11110000000000000000000	0.688
5	11111000000000000000000	0.719
6	11111010000000000000000	0.750
7	11111011000000000000000	0.781
8	11111011010000000000000	0.813
9	11111011010100000000000	0.844
10	11111011010110000000000	0.875
11	11111011010110100000000	0.906
12	11111111010110100000000	0.922
13	11111111110110100000000	0.938
14	11111111111110100000000	0.953
15	11111111111111100000000	0.969
16	11111111111111110000000	0.984
17	11111111111111111000000	1.000

TERMS IN THE ORDER OF THEIR ADDITION:

X1X010
X1X111
000X00
X00011
X11001
001X01
001X10
010X01
100X01
1001X0
110X00
10X011
00X100
010X10
X01110
11001X
101000

APPENDIX D

PL/I CODE FOR N-ARY TREE APPROACH TO BEST-SET DISCOVERY.

```
=>
//DES JOB ',,,T=10M,L=10,IO=20,R=1024K','ERIC GULLICHSEN',CLASS=1
// EXEC PL1OCLG,SIZE=1024K
//PL1.SYSIN DD *
 /* DISCOVERY OF SETS OF BEST TERMS FOR S-BOX APPROXIMATION */

 (NOSTRG,NOSUBRG): BEST: PROC OPTIONS(MAIN);

 /*****************************************************************/
 /*                                                               */
 /* FOR EACH OF THE 4 OUTPUTS FOR EACH OF THE 8 S-BOXES,          */
 /* WHAT IS THE *BEST* S.P. APPROXIMATION USING N TERMS TO        */
 /* THE REAL S-BOX.                                               */
 /* IS THERE ALWAYS A BEST SET OF N TERMS WHICH IS A SUBSET       */
 /* OF A BEST SET OF N+1 TERMS??                                  */
 /*                                                               */
 /* ALGORITHM: FORM THE COVER TABLE, INDICATING     WHICH OF     */
 /* THE 32 INPUTS FOR WHICH THE OUTPUT SHOULD BE ON THE OUTPUT    */
 /* IS INDEED ON FOR A SINGLE GIVEN TERM.                         */
 /* THEN SEARCH THIS TABLE USING PARALLEL TREES, TO ATTEMPT       */
 /* (FOR EACH OUTPUT OF EACH S-BOX) TO FORM A SEQUENCE OF         */
 /* BEST SETS OF SIZE 1...N (N<=23 FOR ALL OUTPUTS) TO PROVE      */
 /* THAT A BEST SET EXISTS AT ALL SIZES 1...N UNDER THE           */
 /* PROPERTY OF MONOTONIC ADDITION OF TERMS.                      */
 /*****************************************************************/

  DCL (SBOX,OUTBIT) FIXED BIN(15); /* SBOX & BIT COUNT */
  DCL (SPFILE,ONFILE) FILE STREAM INPUT;
  DCL SYSPRINT FILE STREAM OUTPUT PRINT;

  DCL (NUMTERMS, /* # S.P. TERMS IN APPROX */
       INPCNT, /* IDX 1..32 FOR ON-INPUTS */
       TERMCNT) FIXED BIN(15); /* IDX 1..N <=23 FOR TERMS */

  DCL (I,J) FIXED BIN(15); /* LOOP COUNTERS */
  DCL SRCH_EXIST FIXED BIN(15);

  DCL ONFOR(32,6) BIT(1); /* INPUTS WHERE SBOX IS ON */
  DCL TERMS(23,6) CHAR(1); /* INPUTS WHERE SBOX IS ON */
  DCL CONTRIB(23) BIT(32) ALIGNED; /* CONTRIBUTION TABLE */

  DCL MASK BIT(32);
  DCL COMPMASK BIT(32) ALIGNED;
  DCL SAVEMASK(23) BIT(32) ALIGNED;
  DCL MASKVEC(32) BIT(1) DEFINED MASK;
  DCL TERMUSED(23) BIT(1);

  DCL NUMON(23) FIXED BIN(15);
  DCL (MAXON,BITSON,BESTON,OLDBESTON) FIXED BIN(15);
```

```
PAUSE:
      DCL SAVEBITSON(23) FIXED BIN(15);

      /* VECTOR OF PTRS TO NODES STILL AVAIL FOR EXPANSION */
      DCL (OPEN,NEWOPEN)(20000) PTR;
      DCL (OPENHI,NEWOPENHI) FIXED BIN(15);

      DCL ZERO6(6) CHAR(1) INIT((6) (1)  '0');
      DCL XSTR6(6) CHAR(1) INIT((6) (1) 'X');
      DCL XORES(6) BIT(1);
      DCL TEMPBIT BIT(1);

      /* ORDER IN WHICH TO MONOTONICALLY ADD TERMS */
      DCL ORDER(8,4,23) FIXED BIN(15);
      DCL CUMON(23)  FIXED BIN(15); /* # BITSON IN ORDER */
      DCL PRTMASK(23) BIT(32) ALIGNED;

      DCL (P,TOP,CURR,NEW,PREV,FINI) PTR;

      /* N-ARY TREE NODE STRUCTURES:  */
      /* LINK NODE STRUCTURE FOR LIST OF POINTERS */
      DCL 1 LKNODE BASED,
            2 SON PTR,
            2 LINK PTR;

      /* NODE STRUCTURE TO CONTAIN A TERM */
      DCL 1 NODE BASED,
            2 TERM FIXED BIN(15), /* TERM # */
            2 ORMASK BIT(32), /* CUMUL OUTPUTS COVERED */
            2 FATHER PTR, /* TO FATHER NODE */
            2 LINK PTR;

      DCL (GO,FIRST,NOTFND) BIT(1); /* FLAGS */

      DCL (SUBSTR,NULL,SUM,EMPTY,ALL,HBOUND,FLOAT) BUILTIN;

      /* AREA IN WHICH TO BUILD THE TREE */
      DCL TREE AREA(512000);
     1
      ON AREA BEGIN;
        PUT SKIP(2) FILE(SYSPRINT) LIST('*** AREA OVERFLOW ***');
        STOP;
      END;
      OPEN FILE(SPFILE) TITLE('SPFILE');
      OPEN FILE(ONFILE) TITLE('ONFILE');

      /* LOOP FOR EACH OF THE 4 OUTPUTS OF EACH SBOX */
      DO SBOX=1 TO 1;
      DO OUTBIT=1 TO 4;
```

```
PAUSE:

        CALL READIN; /* READ IN SPTERMS AND ONFOR */
        /* FOR TABLE OF CONTRIBUTIONS, INDICATING WHICH TERMS
        ARE ON FOR WHICH INPUT */
        NUMON(*)=0;
        DO INPCNT=1 TO 32;
           DO TERMCNT=1 TO NUMTERMS;
              CALL XOR(ONFOR(INPCNT,*) , (ZERO6=TERMS(TERMCNT,*)) , XORES
);
              TEMPBIT=AND( (XSTR6=TERMS(TERMCNT,*)) | XORES);
              SUBSTR(CONTRIB(TERMCNT),INPCNT,1)=TEMPBIT;
              IF TEMPBIT THEN NUMON(TERMCNT)=NUMON(TERMCNT)+1;
           END;
        END;

        /* DUMP THE CONTRIBUTION TABLE */
        PUT PAGE FILE(SYSPRINT) LIST('    *** CONTRIBUTION TABLE***');
        PUT SKIP(2) FILE(SYSPRINT) EDIT('SP TERM','INPUT COVER',(7)'*',
          (32)'*')(A,COL(23),A,SKIP,A,COL(12),A);
        PUT SKIP FILE(SYSPRINT) EDIT((TERMS(I,*),CONTRIB(I),NUMON(I)
          DO I=1 TO NUMTERMS))(6 A(1),X(5),B(32),F(6),SKIP);

        /* FORM TOP LEVEL OF TREE FROM SINGLE TERMS WITH MOST 1*S */
        OPENHI=1; /* SET HI PTR FOR OPEN NODES */
        MAXON=LARGEST(NUMON);
        FIRST='1'B;
        DO I=1 TO NUMTERMS;
           IF MAXON=NUMON(I) THEN DO;
              ALLOC LKNODE SET(CURR) IN(TREE);
              IF FIRST THEN DO;
                 FIRST='0'B;
                 PREV,TOP=CURR;
              END;
              ALLOC NODE SET(P) IN(TREE);
              P->NODE.TERM=I;
              P->NODE.FATHER=NULL; /* NO FATHER FOR TOP LEVEL */
              P->NODE.ORMASK=CONTRIB(I);
              CURR->LKNODE.SON=P;
              OPEN(OPENHI)=P; /* PUT INTO OPEN VECTOR */
              OPENHI=OPENHI+1;
              PREV->LKNODE.LINK=CURR;
              PREV=CURR;
           END;
        END;
        CURR->LKNODE.LINK=NULL; /* LAST LINK POINTER SET NULL */

       1/* PROCESS ALL NODES IN OPEN VECTOR TO GET TO NEXT LEVEL IN TREE
*/
```

```
PAUSE:

        OLDBESTON=0;
        GO='1'B;
        DO WHILE(GO);
           BESTON=0;
           DO I=1 TO OPENHI-1; /* FOR ALL NODES IN OPEN VECTOR */
              FIRST='1'B;
              /* FIND OUT HOW MANY ON IN ORMASK,
                 GIVEN NEXT BEST TERM CHOICE */
              COMPMASK=OPEN(I)->NODE.ORMASK;
              MAXON=0;
              DO J=1 TO NUMTERMS;
                 SAVEMASK(J),MASK=COMPMASK|CONTRIB(J);
                 SAVEBITSON(J),BITSON=SUM(MASKVEC);
                 IF BITSON > MAXON THEN MAXON=BITSON;
              END;

              /* IF BETTER THAN ANYTHING YET ON NEWOPEN, RESET NEWOPEN *
/
              IF MAXON > BESTON THEN DO;
                 BESTON=MAXON;
                 NEWOPENHI=1;
              END;

     1        /* IF BETTER OR AS GOOD, ADD TO NEWOPEN LIST */
              IF MAXON >= BESTON THEN DO;
                 PREV=OPEN(I);
                 DO J=1 TO NUMTERMS;
                    IF SAVEBITSON(J) = MAXON THEN DO;
       /* EXAMINE NEWOPEN POINTER VECTOR TO DETERMINE IF
          MASK TO BE ADDED HAS ALREADY BEEN ADDED AT THIS LEVEL
          IF NOT, ADD IT TO THE TREE */
                       NOTFND='1'B;
                       DO SRCH_EXIST=1 TO NEWOPENHI-1 WHILE(NOTFND);
                          IF SAVEMASK(J)=NEWOPEN(SRCH_EXIST)->NODE.ORMAS
K
                                THEN NOTFND='0'B;
                       END;
                       /* ADD IT, IF NOT FOUND */
                       IF NOTFND THEN DO;
                          ALLOC LKNODE SET(CURR) IN(TREE);
                          /* PREV MAY PT TO NODE OR LKNODE: */
                          IF FIRST THEN DO;
                             FIRST='0'B;
                             PREV->NODE.LINK=CURR;
                          END;
                          ELSE PREV->LKNODE.LINK=CURR;
                          PREV=CURR;
```

```
PAUSE:
                        ALLOC NODE SET(P) IN(TREE);
                        CURR->LKNODE.SON=P;
                        P->NODE.TERM=J;
                        P->NODE.ORMASK=SAVEMASK(J);
                        P->NODE.FATHER=OPEN(I);
                        /* SEE IF DONE YET */
                        IF SAVEBITSON(J)=32 THEN DO;
                           GO='0'B;
                           FINI=P; /* LAST NODE FOR TRACEBACK */
                        END;
                        /* INSERT INTO NEWOPEN LIST */
                        NEWOPEN(NEWOPENHI)=P;
                        NEWOPENHI=NEWOPENHI+1;
                     END;
                  END; /* OF SAVEBITSON IF */
               END; /* OF J FORLOOP */
               CURR->LKNODE.LINK=NULL;
            END;
         END;

         /* TRANSFER NEWOPEN TO OPEN */
         DO I=1 TO NEWOPENHI-1;
            OPEN(I)=NEWOPEN(I);
         END;
         OPENHI=NEWOPENHI;
1
          /* DEPTH-FIRST SHORTCUT:
            IF ONLY ONE BIT ADDED TO ANY ORMASK DURING THIS ITERATION
            TO GENERATE NEW TREE LEVEL, WE MAY IMMEDIATELY PENETRATE
            DIST TO END OF TREE, ADDING ANY TERM NOT YET ON A BEST
            PATH */

            IF OLDBESTON+1 = BESTON THEN DO;
               GO='0'B; /*STOP THE SEARCH */
               CURR=NEWOPEN(1);
               /* FORM MASK TO TERMS USED IN BEST PATH SO FAR */
               TERMUSED(*)='0'B;
               DO WHILE(CURR ~= NULL);
                  TERMUSED(CURR->NODE.TERM)='1'B;
                  CURR=CURR->NODE.FATHER;
               END;

               /* FORM A DIST PATH TO LEVEL N */
               COMPMASK=NEWOPEN(1)->NODE.ORMASK;
               CURR=NEWOPEN(1);
               DO J=1 TO NUMTERMS;
                  IF ~TERMUSED(J) THEN DO;
                     ALLOC NODE SET(P) IN(TREE);
```

```
PAUSE:
                        P->NODE.TERM=J;
                        P->NODE.ORMASK=COMPMASK|CONTRIB(J);
                        COMPMASK=P->NODE.ORMASK;
                        P->NODE.FATHER=CURR;
                        CURR=P;
                     END;
                  END;
                  FINI=CURR; /* SET LAST POINTER */
               END;

               ELSE OLDBESTON=BESTON; /* ELSE CONTINUE TREE BUILDING */

         END; /* WHILE LOOP FOR PROCESSING TREE */

       1 /* WHILE LOOP TERMINATED AS SOME ORMASK WAS ALL 111...1
         TRACEBACK FROM FINI BY FATHER LINKS */

         CURR=FINI;
         DO I=NUMTERMS TO 1 BY -1;
            ORDER(SBOX,OUTBIT,I)=CURR->NODE.TERM;
            PRTMASK(I),MASK=CURR->NODE.ORMASK; /* SAVE THE COVER */
            CUMON(I)=SUM(MASKVEC);
            CURR=CURR->NODE.FATHER;
         END;

         /* PRINT TERMS IN ORDER OF ADDITION, TOGETHER WITH THE
            VALUES INDICATING PROBABILITY OF CORRECTNESS */
         PUT PAGE FILE(SYSPRINT) EDIT('# TERMS  INCLUDED','INPUT COVER',
           'CORRECTNESS',(16)'*',(14)'*',(11)'*')
           (2 (A,COL(25),A,COL(55),A,SKIP));
         DO I=1 TO NUMTERMS;
            PUT SKIP FILE(SYSPRINT) EDIT(ORDER(SBOX,OUTBIT,I),PRTMASK(I),
              .5+FLOAT(CUMON(I),6)/64.)(F(8),COL(21),B(32),COL(56),F(7,3)
);
         END;
         PUT SKIP(2) FILE(SYSPRINT) LIST('+++ END OF TABLE +++');

         TREE=EMPTY; /* FREE ENTIRE TREE BY EMPTYING AREA */


         END; /* OUTBIT LOOP */
         END; /* SBOX   LOOP */

         CLOSE FILE(ONFILE),FILE(SPFILE);

       1
        /*****************************************************************/
        /*                                                               */
```

```
PAUSE:
        /* INPUT ROUTINE, TO READ THE S.P. TERMS  AND THE SET OF 32     */
        /* INPUT VALUES FOR WHICH THE OUTPUT SHOULD BE ON, FOR THE      */
        /* S-BOX AND OUTPUT PAIRS, SEQUENTIALLY.                        */
        /*                                                              */
        /****************************************************************/

        READIN: PROC;
         NUMTERMS=0; /* # OF TERMS IN THIS APPROX */
         DO I=1 TO 23;
            GET FILE(SPFILE) EDIT(TERMS(I,*))(6 A(1),SKIP);
            IF TERMS(I,1) ~= ' ' THEN NUMTERMS=NUMTERMS+1;
         END;

         GET FILE(ONFILE) EDIT(((ONFOR(I,J) DO J=1 TO 6) DO I=1 TO 32))
           (6 B(1),SKIP);

         /* ECHO THE INPUT */
         PUT PAGE FILE(SYSPRINT) EDIT('FOR S-BOX ',SBOX,' OUTPUT BIT ',OU
TBIT)
          (2 (A,F(3)));
         PUT SKIP(3) FILE(SYSPRINT) LIST('SUM-OF-PRODUCT TERMS:');
         PUT SKIP(2) FILE(SYSPRINT) EDIT((TERMS(I,*) DO I=1 TO NUMTERMS))
           (6 A(1),SKIP);
         PUT SKIP(3) FILE(SYSPRINT) LIST('OUTPUT SHOULD BE ON FOR INPUTS:
');
         PUT SKIP(2) FILE(SYSPRINT) EDIT(ONFOR)(6 B(1),SKIP);
        END READIN;
        /* FOR TABLE OF CONTRIBUTIONS, INDICATING WHICH TERMS

       1
        /****************************************************************
*/
        /* AND RETURNS 1 IFF ALL ALTS IN ARGUMENT ARRAY ARE 1
*/
        /****************************************************************
*/

        AND: PROC(BOOLVEC) RETURNS(BIT(1));
         DCL (RES,BOOLVEC(*)) BIT(1);
         DCL I FIXED BIN(15);
         RES='1'B;
         DO I=1 TO HBOUND(BOOLVEC,1) WHILE(RES);
            IF ~BOOLVEC(I) THEN RES='0'B;
         END;
         RETURN(RES);
        END AND;

        /****************************************************************
```

```
PAUSE:
**/
        /* RETURN LARGEST ELEMENT IN A FIXED BIN VECTOR
 */
        /*********************************************************************
**/

        LARGEST: PROC(FBVEC) RETURNS(FIXED BIN(15));
         DCL (I,BIG,FBVEC(*)) FIXED BIN(15);
         BIG=FBVEC(1);
         DO I=2 TO HBOUND(FBVEC,1);
            IF FBVEC(I) > BIG THEN BIG=FBVEC(I);
         END;
         RETURN(BIG);
        END LARGEST;

        /*********************************************************************/
        /* XOR PROC TO XOR 2 BINARY VECTORS                                  */
        /*********************************************************************/

        XOR:PROC(A,B,RES);
          DCL (A,B,RES)(*) BIT(1);
          RES=(A|B)&(~(A&B));
        END XOR;

        END BEST;
       //GO.SPFILE DD DSN=GULLICH.DES.SPTERMS,DISP=SHR
       //GO.ONFILE DD DSN=GULLICH.DES.ONFOR,DISP=SHR
```

APPENDIX E

APL ROUTINES FOR BOOLEAN MINIMIZATION BY SPECTRAL TRANSLATION

```
      ∇ B←BASIS S;BIG;POS;N;K
[ 1]  ⍝
[ 2]  ⍝ GIVEN VECTOR S OF SPECTRAL COEFFICIENTS, RETURN BASIS MTX
. B
[ 3]  ⍝ INDICATING REQUIRED SPECTRAL TRANSLATIONS TO MAXIMIZE PRI
MARY
[ 4]  ⍝ COEFFICIENTS
[ 5]  B←(0,N←2⊛⍴S)⍴0
[ 6]  S←|S
[ 7]  S[1]←0
[ 8]  ⍝
[ 9]  ⍝ LOOP UNTIL WE HAVE N BASIS VECTORS
[10]  ⍝
[11]  LOOP:BIG←⌈/S
[12]  S[POS←S⍳BIG]←0
[13]  B←B,[1] POS←(N⍴2)⊤¯1+POS
[14]  ⍝
[15]  ⍝ REMOVE ALL LINEAR (XOR) COMBS OF BASIS VECTORS
[16]  ⍝ FROM FURTHER POSSIBLE CONSIDERATION
[17]  K←1
[18]  REM:S[1+2⊥⍉0≠⌿B[K COMB 1↑⍴B;]]←0
[19]  →((K←K+1)≤1↑⍴B)/REM
[20]  →(N≠1↑⍴B)/LOOP
[21]  ⍝ TRANSPOSE AND INVERT TO GET BASIS
[22]  B←INVERSE⍉B
      ∇


      ∇ R←M COMB N
[1]   →(M=1,N)/L1,R1
[2]   R←1+(0,(M-1) COMB N-1),[1] M COMB N-1
[3]   →0
[4]   L1:R←(⍳N)∘.×⍳1
[5]   →0
[6]   R1:R←(⍳1)∘.×⍳N
      ∇


      ∇ Z←COMPLEXITY SPECTRUM;N;ORDER
[1]   N←2⊛⍴SPECTRUM
[2]   ORDER←+⌿(6⍴2)⊤¯1+⍳2*6
[3]   Z←(N×2*N)-(÷2*N-2)×ORDER+.×SPECTRUM*2
      ∇
```

```
      ∇  F←FUNC R;P
[1]   ⍝
[2]   ⍝ GIVEN SPECTRAL COEFFS, RETURN CORR. MINTERMS NUMBERS
[3]   ⍝
[4]   F←(1÷⍴R)×(TRANS 2⊛⍴R)+.×R
[5]   F←F/⍨1+⍳⍴R
      ∇


      ∇  Z←HAD N;K
[1]   Z← 1 1 ⍴1
[2]   K←0
[3]   LOOP:→(K=N)/0
[4]   Z←(Z,Z),[1] Z,-Z
[5]   K←K+1
[6]   →LOOP
      ∇
```

```
      ∇  R←INVERSE MAT
[ 1]  ⍝ THIS FUNCTION WILL TAKE ANY MATRIX AND RETURN :
[ 2]  ⍝ 1) THE INVERSE OF THE MATRIX IF IT EXISTS
[ 3]  ⍝  2) A 0 MATRIX OTHERWISE
[ 4]  ⍝ CHECK 3 EXIT CONDITIONS :
[ 5]  ⍝ 1) MATRIX=0
[ 6]  ⍝ 2) MATRIX IS NOT SQUARE
[ 7]  ⍝ 3) MATRIX IS SCALAR OR NOTHING
[ 8]  →(~((∧/∧/⎕CT>|MAT)∨((1↑⍴MAT)≠(¯1↑⍴MAT))∨(0=⍴⍴MAT)))/START
[ 9]  ⍝ RETURN THE 0×0 MATRIX
[10]  R← 0 0 ⍴0
[11]  →0
[12]  ⍝ NOW WE KNOW THAT MAT IS SQUARE AND ≠0
[13]  ⍝ SAVE DIMENSIONS
[14]  START:N←1↑⍴MAT
[15]  ⍝ CATENATE IDENTITY TO MAT
[16]  MAT←MAT,((⍳N)∘.=⍳N)
[17]  ⍝ REDUCE MAT TO REDUCED ROW ECHELON FORM, WHICH IN A SQUARE
 MATRIX
[18]  ⍝ IS EQUIVALENT TO TRIANGULATION.
[19]  MAT←N REDROWECH MAT
[20]  ⍝ CHECK THAT THERE ARE NO 0 ROWS IN FIRST N COLUMNS
[21]  ⍝ I.E. RANK(MAT)=N
[22]  ⍝ FOR NON-SINGULARITY
[23]  →(N=MATRANK((N,N)↑MAT))/OKAY
[24]  ⍝ ELSE RETURN THE 0×0 MATRIX
[25]  R← 0 0 ⍴0
[26]  →0
[27]  OKAY:R←(0,N)↓MAT
      ∇


      ∇  R←MATRANK MAT
[1]   ⍝ THIS FUNCTION WILL DETERMINE THE RANK OF A GIVEN MATRIX
[2]   ⍝ PUT MATRIX IN ROW ECHELON FORM
[3]   MAT←(¯1↑⍴MAT) ROWECH MAT
[4]   R←+/∨/(⎕CT<MAT)
[5]   →0
      ∇
```

```
      ∇  NEW∆S←BAS MAXPRIM S
[1]   ⍝ GIVEN A BASIS MTX. AND VECTOR OF SPECTRAL COEFFS. S,
[2]   ⍝ PERMUTE THE COEFFICIENTS TO MAXIMIZE PRIMARY ONES
[3]   NEW∆S←S[⍋1+2⊥BAS≠.∧(6⍴2)⍉¯1+⍳64]
      ∇


      ∇  R←N REDROWECH MAT
[ 1]  ⍝ THIS FUNCTION WILL REDUCE ANY GIVEN MATRIX TO A
[ 2]  ⍝ REDUCED ROW ECHELON FORM.
[ 3]  ⍝ REDUCE MATRIX TO ROWECH FIRST
[ 4]  MAT←N ROWECH MAT
[ 5]  ⍝ CHECK 3 CONDITIONS TO EXIT :
[ 6]  ⍝ 1) VECTOR
[ 7]  ⍝ 2) MATRIX=0
[ 8]  ⍝ 3) SCALAR OR NOTHING
[ 9]  ⍝
[10]  →(~((1=⍴⍴MAT)∨(∧/∧/⎕CT>|MAT)∨(0=⍴⍴MAT)))/ROWCHK
[11]  R←MAT
[12]  →0
[13]  ⍝ CHECK LAST ROW : IF =0 RECURSE ON SMALLER
[14]  ⍝ MATRIX OF M-1 ROWS, N COLUMNS
[15]  ROWCHK:→(∧/⎕CT>|MAT[1↑⍴MAT;⍳N])/RECURSE
[16]  ⍝ ELSE START REDUCING THIS ROW
[17]  ⍝ ALL THE ELEMENTS ABOVE THE 1S
[18]  ⍝  FIND COLUMN NUMBER WHERE FIRST NON ZERO ELEMENT IS.
[19]  NZC←(∨⌿(⎕CT<|MAT[1↑⍴MAT;⍳N]))⍳1
[20]  ⍝ ZERO OUT ELEMENT ABOVE THIS 1 JUST FOUND
[21]  TEMP←(((¯1+1↑⍴MAT),(¯1↑⍴MAT))⍴,((-MAT[⍳¯1+1↑⍴MAT;NZC])∘.×MA
T[1↑⍴MAT;]))
[22]  MAT[⍳¯1+1↑⍴MAT;]←2|MAT[⍳¯1+1↑⍴MAT;]+TEMP
[23]  ⍝ RECURSE ON SMALLER MATRIX
[24]  RECURSE:R←(N REDROWECH(((¯1+1↑⍴MAT),(¯1↑⍴MAT))↑MAT)),[1](MA
T[1↑⍴MAT;])
[25]  ⍝
      ∇
```

```
        ∇  R←N ROWECH MAT
[ 1]    ⍝ THIS FUNCTION WILL ACCEPT ANY MATRIX AND PUT THE FIRST
[ 2]    ⍝ N COLUMNS IN ROW-ECHELON FORM.
[ 3]    ⍝
[ 4]    ⍝ CHECK 2 EXIT CONDITIONS :
[ 5]    ⍝ 1) MATRIX=0
[ 6]    →(~∧/∧/⎕CT>|(((1↑⍴MAT),N)↑MAT))/NEXT
[ 7]    R←MAT
[ 8]    →0
[ 9]    ⍝ 2) MAT IS A SCALAR OR NOTHING AT ALL
[10]    NEXT:→(~0=⍴⍴MAT)/START
[11]    R←MAT
[12]    →0
[13]    ⍝
[14]    ⍝ CHECK THAT THE NUMBER OF COLUMNS PROCESSED IS ≤N
[15]    START:→(N=0)/0
[16]    ⍝
[17]    ⍝ FIND FIRST NON-ZERO ELEMENT :
[18]    ⍝    NZC - NON ZERO COLUMN INDEX
[19]    ⍝    NZR - NON ZERO ROW INDEX
[20]    NZC←(∨≠⎕CT<|(((1↑⍴MAT),N)↑MAT))⍳1
[21]    NZR←(⎕CT<|MAT[;NZC])⍳1
[22]    ⍝ SWITCH TO PUT ELEMENT A[NZR,NZC] INTO A[1,NZC]
[23]    MAT[1,NZR;]←MAT[NZR,1;]
[24]    ⍝ MAKE A[1,NZC]=1
[25]    MAT[1;]←MAT[1;]÷MAT[1;NZC]
[26]    ⍝ MAKE COL=NZC ALL ZEROES UNDER A[1,NZC]
[27]    MAT[1+⍳¯1+1↑⍴MAT;]←2|MAT[1+⍳¯1+1↑⍴MAT;]+(-MAT[1+⍳¯1+1↑⍴MAT;
NZC])∘.×MAT[1;]
[28]    ⍝ RECURSE ON SMALLER MATRIX
[29]    R←MAT[1;],[1](0,((N-1) ROWECH(1 1 ↓MAT)))
[30]    ⍝
        ∇
```

```
      ∇ SPECTRAL△MIN;BOX;BIT;S;NEWS;F;BAS;SPFORM
[ 1]  COMP△OLD←COMP△NEW←⍳0
[ 2]  NTERMS△OLD←NTERMS△NEW←⍳0
[ 3]  NDCOLD←NDCNEW←0
[ 4]  BOX←1
[ 5]  BOXLOOP:BIT←1
[ 6]  BITLOOP:''
[ 7]  'FOR S-BOX ',(⍕BOX),' OUTPUT ',⍕BIT
[ 8]  '----------------'
[ 9]  ''
[10]  'MINTERMS:'
[11]  (BIT OUTPUT SBOX[BOX;;])/⍳64
[12]  ''
[13]  'QM MINIMIZATION:'
[14]  QM PRIMIMP BIT ON BINARY SBOX[BOX;;]
[15]  ⎕←SPFORM←SELECT△ALTS
[16]  NTERMS△OLD←NTERMS△OLD,1↑⍴SPFORM
[17]  NDCOLD←NDCOLD++/+/'X'=SPFORM
[18]  ''
[19]  ''
[20]  'SPECTRUM:'
[21]  ⎕←S←SPECTRUM BOX,BIT
[22]  COMP△OLD←COMP△OLD,COMPLEXITY S
[23]  'COMPLEXITY: ',⍕1↑COMP△OLD
[24]  ''
[25]  ''
[26]  'SPECTRAL TRANSLATION BASIS (⍒⍉⍒)'
[27]  ⎕←BAS←BASIS S
[28]  ''
[29]  ''
[30]  'TRANSLATED SPECTRUM:'
[31]  ⎕←NEWS←BAS MAXPRIM S
[32]  COMP△NEW←COMP△NEW,COMPLEXITY NEWS
[33]  'COMPLEXITY: ',⍕1↑COMP△NEW
[34]  ''
[35]  'MINTERMS FOR TRANSLATED FUNCTION:'
[36]  ⎕←F←FUNC NEWS
[37]  ''
[38]  'QM MINIMIZATION FOR TRANSLATED FUNCTION:'
[39]  QM PRIMIMP⍒⍉(6⍴2)⊤F
[40]  ⎕←SPFORM←SELECT△ALTS
[41]  NTERMS△NEW←NTERMS△NEW,1↑⍴SPFORM
[42]  NDCNEW←NDCNEW++/+/'X'=SPFORM
[43]  5↓⍴⎕TC[2]
[44]  ⍝
[45]  →((BIT←BIT+1)≤4)/BITLOOP
[46]  →((BOX←BOX+1)≤8)/BOXLOOP
[47]  'NUMBER OF P-TERMS PER FUNCTION BEFORE:'
```

```
[48]   NTERMSΔOLD
[49]   'AVG. NUMBER OF DC PER P-TERM ',▼(+/NTERMSΔOLD)÷NDCOLD
[50]   ''
[51]   'NUMBER OF P-TERMS PER FUNCTION AFTER:'
[52]   NTERMSΔNEW
[53]   'AVG. NUMBER OF DC PER P-TERM ',▼(+/NTERMSΔNEW)÷NDCNEW
       ∇


       ∇  R←SPECTRUM BOXOUT;F
[1]    ⍝
[2]    ⍝ GIVEN S-BOX AND OUTPUT BIT, RETURN THE 64 SPECTRAL COEFFS
[3]    R←(TRANS 2⊛⍴F)+.×F←BOXOUT[2] OUTPUT SBOX[BOXOUT[1];;]
       ∇


       ∇  Z←TRANS N;Q
[1]    Z←(1,1)⍴1
[2]    →(N<1)/0
[3]    Q←TRANS N-1
[4]    Z←(Q,Q),[1](Q,-Q)
       ∇


       ∇  Z←XOR VEC
[1]    ⍝ RETURN XOR OF THE BITS IN VECTOR VEC
[2]    Z←0≠2|+/VEC
       ∇


       ∇  Z←B XORMAP INPS;R;C
[1]    ⍝ TO MAP N×6 MTX OF INPUTS THROUGH TREE OF XORS
[2]    ⍝ AS REPRESENTED BY TRANSLATION MATRIX B
[3]    Z←(⍴INPS)⍴0
[4]    R←1
[5]    RLOOP:C←1
[6]    CLOOP:Z[R;C]←XOR B[C;]/INPS[R;]
[7]    →((C←C+1)≤6)/CLOOP
[8]    →((R←R+1)≤1↑⍴INPS)/RLOOP
       ∇
```

END OF APPENDIX

APPENDIX F

PL/I ROUTINES FOR UNIDIRECTIONAL KEY SEARCH

```
=>
      //DES JOB ',,,T=5M,L=15,IO=20,R=768K','ERIC GULLICHSEN',CLASS=1
      // EXEC PL1OCG,SIZE=768K
      //PL1.SYSIN DD *
       /* SEARCH TREE APPROACH TO GIVEN PLAINTEXT DES ATTACK */
       /* RECURSIVE BACKTRACK SEARCH TREE TO DISCOVER K FROM KNOWN P-C
          PAIRS. B1ST EXPANSION OF NECESSARILY CONJUNCTIVE CONDITIONS,
          VIRTUAL D1ST EXPANSION AT CHOICE POINTS WITH FATHER POINTERS T
O
          ENABLE BACKTRACKING ON FAILURE. 6 TYPES OF NODES, TO
          REPRESENT THE VARIOUS STRUCTURES IN THE DES ENCRYPTION A*/

        SEARCH: PROC OPTIONS(MAIN) REORDER;

       DCL TREE AREA(384000);     /* AREA FOR TREE GROWTH */

       /* FILES */
       DCL SYSPRINT FILE STREAM OUTPUT PRINT;
       DCL (SPFILE,SPCFILE,KSFILE,PCFILE) FILE STREAM INPUT;

       /* Q OF POINTERS TO "OPEN" NODES I.E. THOSE REQ. EXPANSION */
       DCL (OPEN,OPENEND) PTR;     /* TO START AND END OF Q */
       DCL (OPENCURR,OPENPREV) PTR;
       DCL 1 OPENNODE BASED,
            2 NODE PTR,
            2 LINK PTR;

       /* DES BLOCKS */
       DCL KEY(64) CHAR(1) INIT((64) (1) ' '); /* K TO DISCOVER IN SEARC
H */
       DCL (PTEXT,CTEXT)(64) BIT(1);            /* KNOWN P-C PAIR */

       DCL NUMNODES FIXED BIN(31) INIT(0); /* # OF NODES EXPANDED */

       DCL SPTERMS(32,23,6) CHAR(1); /* S.P. APPROX. TO S-BOXES */
       DCL SPCTERMS(32,23,6) CHAR(1); /* S.P. APPROX. TO S-BOX COMPL */
       DCL MAXPTERMS FIXED BIN(15) INIT(0); /* MAX PTERMS IN ANY SP FORM
 */
       DCL KEYPERM(16,48) FIXED BIN(15);    /* KEY BIT SELECTION INDICES
 */

       /* INVERSES OF PERMUTATIONS IN DES ALGO */
       DCL E_PERM_INV(48) FIXED BIN(15)
        INIT(32,1,2,3,4,5,4,5,6,7,8,9,8,9,10,11,12,13,12,13,14,15,16,17,
             16,17,18,19,20,21,20,21,22,23,24,25,24,25,26,27,28,29,
             28,29,30,31,32,1);
       DCL P_PERM_INV(32) FIXED BIN(15)
        INIT(16,7,20,21,29,12,28,17,1,15,23,26,5,18,31,10,2,8,24,14,
             32,27,3,9,19,13,30,6,22,11,4,25);
```

```
PAUSE:

        DCL (NULL,TRUNC,REPEAT,MAX) BUILTIN;

        /* TREEDUMP */ DCL TOP(64) PTR; /* PTRS TO TREE TOP LEVEL */

        /* HOW MANY DES ROUNDS WERE USED TO GENERATE CTEXT? */
        DCL NUM_ENCRY_RNDS FIXED BIN(15) INIT(2);
       1
        /* DECLARATIONS OF TYPES OF NODES IN SEARCH TREE */

        /* 'SUPER' NODE IS A SET OF DESCRIPTOR FIELDS ASSOCIATED WITH
           EACH NODE IN THE SEARCH TREE. CONTENTS INCLUDE A TYPE CODE,
           POINTER TO ACTUAL NODE, AND FIELDS COMMON TO ALL TYPES */

        DCL 1 SUPER BASED ,
              2 TYPE CHAR(1),        /* NODE TYPE */
              2 POS FIXED BIN(15),   /* POSITION [1,32] IN BLOCK */
              2 LVL FIXED BIN(15),   /* LEVEL IN ENCRYPTION */
              2 FATHER PTR,          /* FATHER POINTER */
              2 OPENQ PTR,           /* -> TO OPENNODE */
              2 NODE PTR;            /* -> TO ACTUAL  NODE */


        DCL 1 RNODE BASED ,         /* TYPE 'R' NODE */
              2 VAL BIT(1),        /* VALUE OF NODE FROM [0,1] */
              2 COUNT FIXED BIN(15) INIT(0), /* # TIMES EXPANDED [0,2] */
              2 LPTR PTR INIT(NULL),
              2 RPTR PTR INIT(NULL);


        DCL 1 FNODE BASED ,       /* TYPE 'F' NODE */
              2 VAL BIT(1),
              2 TERMNUM FIXED BIN(15) INIT(0), /* # OF SP TERM CONSIDERED
*/
              2 XPTR(6) PTR INIT((6) NULL);     /* ->'S TO X NODES OF SP TE
RM */



        DCL 1 XNODE BASED ,            /* TYPE 'X' NODE */
              2 VAL BIT(1),
              2 COUNT FIXED BIN(15) INIT(0),
              2 RPTR PTR INIT(NULL);


        ON ERROR BEGIN;
           PUT FLOW; /* FOR CHECKOUT */
           CALL TREEDUMP(TOP); /* DUMP ENTIRE SEARCH TREE */
```

```
PAUSE:
          STOP;
        END;
       1
        /* ------------- MAINLINE ------------------------------- */

        CALL SETUP;  /* PERFORM READS AND CREATE TREE TOP LVL */
        OPENCURR=OPEN;
        DO WHILE(OPENCURR^=NULL);  /* PROCESS Q WHILE IT IS NOT EMPTY */
           CALL EXPAND(OPENCURR->OPENNODE.NODE);  /* EXPAND CURR NODE */
           OPENCURR=OPENCURR->OPENNODE.LINK;      /* LOOK AT NEXT NODE */

           FREE OPEN->OPENNODE; /* DESTROY PROCESSED NODE POINTER */
           OPEN=OPENCURR; /* MOVE START POINTER ALONG */
        END;

        /* PRINT RESULTS */
        PUT SKIP(2) FILE(SYSPRINT) EDIT('TOTAL # OF NODES EXPANDED ',NUMN
ODES)
                                        (A,F(9));
        PUT SKIP(2) FILE(SYSPRINT) EDIT('ENCRYPTION KEY DISCOVERED:',KEY,
           (64)'.') (A,SKIP,X(10),64 A,SKIP,X(10),A);

        CALL TREEDUMP(TOP); /* PRINT COMPLETED SEARCH TREE */

         /* ------------------------------------------------- */
       1
        /* SELECTION FUNCTION, TO CALL PROPER EXPANSION ROUTINE, BASED ON
           THE TYPE OF THE NODE TO BE EXPANDED.
           INPUTS: CURR = POINTER TO SUPER NODE FOR NODE TO EXPAND  */

        EXPAND: PROC(CURR) RECURSIVE;
         DCL CURR PTR;

         /* DEBUG */ CALL DUMP(CURR);

         CURR->SUPER.OPENQ=NULL; /* REMOVE ITS REF. TO Q NODE */

         SELECT(CURR->SUPER.TYPE);
            WHEN('R') CALL R_EXPAND(CURR);
            WHEN('F') CALL F_EXPAND(CURR);
            WHEN('X') CALL X_EXPAND(CURR);
         END;
        END EXPAND;
       1
        /* DEBUG */ DUMP: PROC(CURR); /* DUMP SUPER DATA ABOUT NODE CURR)
 */
        DCL CURR PTR;
        DCL COUNT FIXED BIN(31) STATIC INIT(0); /* # NODES EXPANDED */
```

```
PAUSE:
        IF CURR=NULL THEN PUT SKIP FILE(SYSPRINT) LIST('--- NULL ---');
        ELSE DO;
            PUT SKIP FILE(SYSPRINT) EDIT(COUNT,')','TYPE:',CURR->SUPER.TY
PE,
             ' POS: ',CURR->SUPER.POS,' LVL: ',CURR->SUPER.LVL)
              (F(6),A,X(5),A,X(1),A,A,F(4),A,F(5));
            COUNT=COUNT+1;
         END;
        END DUMP;
       1
        /* TREEDUMP: A ROUTINE TO PRINT THE ENTIRE SEARCH TREE, BY
           RECURSIVE INORDER TRAVERSAL. USED FOR DIAGNOSTIC PURPOSES
           ON ERROR                */

        TREEDUMP: PROC(TOP);
         DCL TOP(64) PTR; /* POINTERS TO TREE TOP LEVEL */
         DCL (K,LEVEL) FIXED BIN(15);
         DCL SIZE FIXED BIN(31);  /* # OF NODES IN TREE */

         SIZE=0;
         PUT PAGE FILE(SYSPRINT) EDIT('TREEDUMP',(8)'*')(2 (X(50),A,SKIP)
);

         PUT SKIP(2) FILE(SYSPRINT) EDIT('ENCRYPTION KEY: ',KEY,(64)'-')
                                        (65 A,SKIP,X(16),A);

         DO K=1 TO 64; /*FOR ALL TOP LEVEL NODES IN TREE */
            LEVEL=0;
            PUT SKIP(3) FILE(SYSPRINT) EDIT('FROM CIPHERTEXT BIT: ',
              K,(23)'-')(A,F(2),SKIP,A);
            CALL NODEPRT(TOP(K));
         END;

         PUT SKIP(2) FILE(SYSPRINT) LIST('@@@@@ TOTAL # OF NODES IN TREE:
',
            SIZE);
       1
        /* NODEPRT: PRINT NODE POINTED AT BY P, THEN RECURSIVELY
           EXPAND THE SUBTREE FROM P      */

        NODEPRT: PROC(P) RECURSIVE;
         DCL (P,Q,ACTUAL) PTR;
         DCL I FIXED BIN(15);

         IF P=NULL THEN RETURN;    /* TRIVIAL CASE */
         LEVEL=LEVEL+1;  /* ARE PROCESSING ONE LEVEL DOWN IN TREE */
         SIZE=SIZE+1;     /* INCREMENT TREE SIZE COUNTER */
         PUT SKIP FILE(SYSPRINT) EDIT(LEVEL,REPEAT(' ',3*LEVEL),P->SUPER.
```

```
PAUSE:
TYPE,
          '  LVL:',P->SUPER.LVL,' POS:',P->SUPER.POS)(F(3),3 A,F(2),A,F
(2));

        ACTUAL=P->SUPER.NODE;
        SELECT(P->SUPER.TYPE);

           WHEN('R') DO;
              PUT SKIP(0) FILE(SYSPRINT) EDIT(REPEAT(' ',25+3*LEVEL),
               ' VAL:',ACTUAL->RNODE.VAL,' COUNT:',ACTUAL->RNODE.COUNT)
               (2 A,B(1),A,F(2));
              CALL NODEPRT(ACTUAL->RNODE.LPTR);  /* RECURSE ON CHILDS */
              CALL NODEPRT(ACTUAL->RNODE.RPTR);
           END;

           WHEN('F') DO;
              PUT SKIP(0) FILE(SYSPRINT) EDIT(REPEAT(' ',25+3*LEVEL),
               'TERM NUMBER: ',ACTUAL->FNODE.TERMNUM,
               ' VALUE: ',ACTUAL->FNODE.VAL)(2 A,F(3),A,B(1));
              DO I=1 TO 6;
                 IF ACTUAL->FNODE.XPTR(I) ^= NULL THEN
                    CALL NODEPRT(ACTUAL->FNODE.XPTR(I));
              END;
           END;

           WHEN('X') DO;
              PUT SKIP(0) FILE(SYSPRINT) EDIT(REPEAT(' ',41+3*LEVEL),
                 'VALUE: ',ACTUAL->XNODE.VAL,
                 'COUNT: ',ACTUAL->XNODE.COUNT,
                '=>KEY(',KEYPERM(P->SUPER.LVL,P->SUPER.POS),')=',
                ACTUAL->XNODE.COUNT=1)
                (A,A,B(1),X(2),A,F(3),X(2),A,F(2),A,B(1));
              CALL NODEPRT(ACTUAL->XNODE.RPTR);
           END;

           END;

           LEVEL=LEVEL-1;  /* AFTER RECURSION, POP UP 1 LVL */

        END NODEPRT;
        END TREEDUMP;
      1
        /* BACKTRACK: ATTEMPT EXPANSION OF OTHER DISJUNCTIVE ALTERNATIVES
           OF THE FATHER OF THE CURRENT NODE. CALLED WHEN A CONTRACTICTIO
N
           IN KEY HYPOTHESES ARISES.
           N.B. THE _EXPAND ROTUINES MUST THEMSELVES CALL BACKTRACK TO
           BACKTRACK HIGHER IN THE TREE IF THEY HAVE NO FURTHER ALTERNATI
```

```
PAUSE:
VE
          POSSIBILITIES */

        BACKTRACK: PROC(CURR) RECURSIVE;
         DCL (CURR,DAD) PTR;
         DAD=CURR->SUPER.FATHER;
         IF DAD=NULL THEN DO;
            PUT SKIP(2) FILE(SYSPRINT) EDIT('+++ ERROR +++',
             'HAVE BACKTRACKED PAST ROOT NODE OF TREE, AT ROOT BIT:')
             (A,SKIP,A);
            CALL DUMP(CURR);
            STOP;
         END;

         CALL DELETE((CURR));    /* ? */
         /*ZOOM*/ PUT SKIP FILE(SYSPRINT) LIST('BACKTRACK TO REEXPAND:');
         CALL EXPAND(DAD);
        END BACKTRACK;
       1
        /* CREATE: GIVEN NODE TYPE CHARACTER CODE, CREATE SUCH A NODE
           (BOTH THE SUPER AND DATA COMPONENT) AND RETURN A POINTER TO IT

           INPUT: NODE_TYPE = 1 CHAR CODE.
           OUTPUT: P = PTR TO SUPER COMPONENT OF NEW NODE */

        CREATE: PROC(NODE_TYPE) RETURNS(PTR);
         DCL NODE_TYPE CHAR(1);
         DCL (P,Q) PTR;

         ON AREA BEGIN;  /* IF NO SPACE LEFT FOR TREE */
            PUT SKIP(2) FILE(SYSPRINT) LIST('*** OVERFLOW IN TREE ***',
             ' TOTAL # OF NODES ALLOCATED: ',NUMNODES);
            CALL TREEDUMP(TOP);
            STOP;
         END;

          ALLOC SUPER IN(TREE) SET(P);
          NUMNODES=NUMNODES+1; /* INCR GLOBAL MAX NODE CTR */
          P->SUPER.TYPE=NODE_TYPE; /* SET NODE TYPE */

          /* ALLOCATE DATA NODE OF PROPER TYPE */
          SELECT(NODE_TYPE);
             WHEN('R') ALLOC RNODE IN(TREE) SET(Q);
             WHEN('F') ALLOC FNODE IN(TREE) SET(Q);
             WHEN('X') ALLOC XNODE IN(TREE) SET(Q);
          END;

          P->SUPER.NODE=Q;
```

```
PAUSE:
          RETURN(P);

        END CREATE;
      1
       /* TO CREATE A NODE OF TYPE R AND INITIALIZE THE FIELDS/
          INPUTS: POS = POSITION [1,32] IN BLOCK
                  LVL = LEVEL [0,3] IN ENCRYPTION
                  VAL = VALUE [0,1] OF RNODE
          OUTPUT:  P  = PTR TO SUPER OF THE NEW NODE */

        CREATE_RNODE: PROC(POS,LVL,VAL) RETURNS(PTR);
         DCL (POS,LVL) FIXED BIN(15);
         DCL VAL BIT(1);
         DCL P PTR;

         P=CREATE('R');  /* CREATE THE NODE */
         /* FILL IN THE FIELDS */
         P->SUPER.POS=POS;
         P->SUPER.LVL=LVL;
         P->SUPER.NODE->RNODE.VAL=VAL;
         RETURN(P);

        END CREATE_RNODE;
      1
       /* SETUP CAUSES DATA TO BE READ IN FROM 3 FILES AND BUILDS THE
          TOP LEVEL OF THE SEARCH TREE FROM RNODES BASED ON KNOWN CIPHER
 */

       SETUP: PROC;
        DCL K FIXED BIN(15);
        DCL P PTR;

        CALL READIN; /* READIN 3 FILES */

        /* SET UP THE TOP LEVEL OF THE SEARCH TREE, ADD NODES TO OPEN */
        ALLOC OPENNODE SET(OPEN);
        OPENPREV,OPENCURR=OPEN;

        DO K=1 TO 64;
           IF K <= 32 THEN P=CREATE_RNODE(K , NUM_ENCRY_RNDS , CTEXT(K))
;
                      ELSE P=CREATE_RNODE(K-32 , NUM_ENCRY_RNDS-1 , CTEX
T(K));
           /* TREEDUMP */ TOP(K)=P;  /* SET TOP LEVEL PTR */
           P->SUPER.FATHER=NULL; /* TOP LEVEL NODES ARE FATHERLESS */
           OPENCURR->OPENNODE.NODE=P;
           OPENPREV->OPENNODE.LINK=OPENCURR; /* CHAIN TO OPEN LIST */
           OPENPREV=OPENCURR;
```

```
PAUSE:
          ALLOC OPENNODE SET(OPENCURR);
        END;

        FREE OPENCURR->OPENNODE;
        OPENPREV->OPENNODE.LINK=NULL;
        OPENEND=OPENPREV;
      1
       /* READIN: TO READIN DATA FROM 3 FILES:
          1) S.P. APPROXIMATIONS FOR S-BOXES FROM SPFILE.
          2) SCHEDULE OF KEY INDICES BY LEVEL FROM KSFILE.
          3) P-C PAIR FROM PCFILE.
          ALL ARE PLACED IN GLOBAL VARIABLES */

       READIN: PROC;
        DCL (K,OUTPUTS,MAX) FIXED BIN(15);

        OPEN FILE(SPFILE),FILE(KSFILE),FILE(PCFILE),FILE(SPCFILE);
        MAX=0;
        /* READIN S.P. TERMS FOR S-BOX FNS UNCOMPLEMENTED */
        DO OUTPUTS=1 TO 32;
           DO K=1 TO 23;
              GET FILE(SPFILE) EDIT(SPTERMS(OUTPUTS,K,*))(6 A(1),SKIP);
              IF SPTERMS(OUTPUTS,K,1) ^= ' ' THEN MAX=K;
           END;
           IF MAX>MAXPTERMS THEN MAXPTERMS=MAX;
        END;

        /* READIN S.P. TERMS FOR S-BOX FNS UNCOMPLEMENTED */
        DO OUTPUTS=1 TO 32;
           DO K=1 TO 23;
              GET FILE(SPCFILE) EDIT(SPCTERMS(OUTPUTS,K,*))(6 A(1),SKIP)
;
              IF SPCTERMS(OUTPUTS,K,1) ^= ' ' THEN MAX=K;
           END;
           IF MAX>MAXPTERMS THEN MAXPTERMS=MAX;
        END;

        /* READIN KEY SCHEDULE */
        GET FILE(KSFILE) EDIT(KEYPERM)(48 F(3),SKIP);

        /* READIN (1ST) P-C PAIR */
        GET FILE(PCFILE) EDIT(PTEXT,CTEXT)(64 B(1),SKIP);
        PUT FILE(SYSPRINT) EDIT('DES KEY SEARCH',(14)'*')(2 (X(40),A,SKI
P));
        PUT SKIP(2) FILE(SYSPRINT) EDIT('PLAINTEXT:  ',PTEXT,
         'CIPHERTEXT: ',CTEXT)(2 (A,64 B(1),SKIP));
        PUT SKIP FILE(SYSPRINT) LIST('NUMBER OF ENCRYPTION ROUNDS:',
                NUM_ENCRY_RNDS);
```

```
PAUSE:

         CLOSE FILE(KSFILE),FILE(SPFILE),FILE(PCFILE),FILE(SPCFILE);

        END READIN;

        END SETUP;
       1
        /* ADD_TO_OPEN: TO ADD A NODE WHOSE SUPER IS POINTED TO BY P TO T
HE
           END OF THE OPEN Q FOR EVENTUAL EXPANSION.
           INPUT: P = POINTER TO THE NODE'S SUPER. */

        ADD_TO_OPEN: PROC(P);
         DCL (NEW,P) PTR;

         ALLOC OPENNODE SET(NEW);
         NEW->OPENNODE.NODE=P;
         NEW->OPENNODE.LINK=NULL;
         OPENEND->OPENNODE.LINK=NEW;
         OPENEND=NEW;

         /* A FIELD IN THE NODE -> TO THE Q NODE WHICH -> IT */
         P->SUPER.OPENQ=NEW;

        END ADD_TO_OPEN;
       1
        /* DELETE: TO DELETE THE SUBTREE WHOSE ROOT IS POINTED AT BY CURR

           1) SET THE FIELD IN THE FATHER OF CURR WHICH NOW POINTS TO CUR
R
              TO BE NULL.
           2) RECURSIVELY DELETE THE SUBTREE FROM CURR */

        DELETE: PROC(CURR);

        DCL (CURR,P) PTR;
        DCL K FIXED BIN(15);

        /* BASED ON THE TYPE OF CURR NODE, DECIDE WHAT TYPE ITS FATHER
           COULD BE, AND NULL THE PROPER FIELD OF THE FATHER ACCORDINGLY
*/

        IF CURR=NULL THEN RETURN;
        /*ZOOM*/ PUT SKIP FILE(SYSPRINT) LIST('***DELETE FOR NODE:');
        /*ZOOM*/ CALL DUMP(CURR);
        SELECT(CURR->SUPER.TYPE);

           /* IF TYPE 'R', FATHER IS R OR X */
```

```
PAUSE:
        WHEN('R') IF CURR->SUPER.FATHER->SUPER.TYPE='R'
         THEN CURR->SUPER.FATHER->SUPER.NODE->RNODE.LPTR=NULL;
         ELSE CURR->SUPER.FATHER->SUPER.NODE->XNODE.RPTR=NULL;

        /* IF TYPE 'F' FATHER IS TYPE 'R' */
        WHEN('F') CURR->SUPER.FATHER->SUPER.NODE->RNODE.RPTR=NULL;

        /* IF TYPE 'X' FATHER IS TYPE 'F'  */
        WHEN('X')
                 /* ONLY RESET TO NULL THE ONE PARTICULAR XPTR */
              DO;
                 P=CURR->SUPER.FATHER->SUPER.NODE;
                 DO K=1 TO 6 WHILE(P->FNODE.XPTR(K) ~= CURR); END;
                 P->FNODE.XPTR(K)=NULL;
              END;

      END; /* SELECT */

     CALL DELETE_SUB(CURR);  /* RECURSIVELY DELETE SUBTREE */
1
     /* DELETE_SUB: GIVEN POINTER TO A NODE, DESTROY IT AND ITS
        SUBTREE OF DESCENDANTS. CALLED FROM DELETE AFTER THE
        FATHER'S SON POINTER HAS BEEN NULLED
        INPUT: CURR = POINTER TO SUBTREE TO DESTROY    */

     DELETE_SUB: PROC(CURR) RECURSIVE;
      DCL (CURR,P,ACTUAL,PREV) PTR;
      DCL GO BIT(1);
      DCL K FIXED BIN(15);

      IF CURR=NULL THEN RETURN; /* TRIVIAL CASE */

      /* IF THE NODE TO BE DELETED IS ON THE OPEN QUEUE,
         IT MUST BE REMOVED, TO AVOID TRYING TO EXPAND A
         NODE WHICH NO LONGER EXISTS. */

     /* IF IT WAS 1ST ON Q, WILL BE KILLED IN MAINLINE ANYHOW */
     /* IF NODE IS ON THE OPEN Q, I.E. IF WE ARE DELETING AN
        UNEXPANDED NODE */
     IF CURR->SUPER.OPENQ ~= NULL THEN DO;
        /*ZOOM*/ K=0;
        PREV=OPENCURR;
        DO P=OPENCURR REPEAT P->OPENNODE.LINK
           WHILE(P ~= CURR->SUPER.OPENQ);
           PREV=P;
           /*ZOOM*/ K=K+1;
        END;
```

```
PAUSE:
          PREV->OPENNODE.LINK=P->OPENNODE.LINK;
          FREE P->OPENNODE;
          /* ZOOM*/ PUT SKIP FILE(SYSPRINT) LIST(
          /*ZOOM*/ 'NODE REMOVED FROM OPEN, ',K,' ENTRIES EXAMINED');
       END;
     1
       /* DELETE THE NODE AND ITS CHILDREN */


       ACTUAL=CURR->SUPER.NODE;    /* ACTUAL IS PTR TO DATA PART OF NODE
*/

       SELECT(CURR->SUPER.TYPE);

          WHEN('R') DO;
             CALL DELETE_SUB(ACTUAL->RNODE.RPTR);
             CALL DELETE_SUB(ACTUAL->RNODE.LPTR);
             FREE ACTUAL->RNODE IN(TREE);
          END;

          WHEN('F') DO;
             DO K=1 TO 6;
                CALL DELETE_SUB(ACTUAL->FNODE.XPTR(K));
             END;
             FREE ACTUAL->FNODE IN(TREE);
          END;

          WHEN('X') DO;
             /* REMOVE KEY BIT HYPOTHESIS */
             KEY(KEYPERM(CURR->SUPER.LVL,CURR->SUPER.POS))=' ';
             CALL DELETE_SUB(ACTUAL->XNODE.RPTR);
             FREE ACTUAL->XNODE IN(TREE);
          END;

       END;  /* SELECT */

       FREE CURR->SUPER IN(TREE);  /* FREE THE SUPER NODE */
     END DELETE_SUB;

     END DELETE;
    1
     /* R_EXPAND: TO EXPAND AN RNODE WHOSE SUPER IS POINTER TO BY CURR

        IF THE LEVEL OF THE RNODE IS 0 OR -1, HAVE HIT BOTTOM OF SEARC
H
        TREE, AND MUST CONFIRM KEY HYPOTHSIS VS. KNOWN PLAINTEXT,
        BACKTRACK ON CONTRADICTION.
        IF RNODE HAS ALREADY BEEN EXPANDED 2 TIMES, NO DISJUNCTIVE
```

```
PAUSE:
        ALTERNATIVES REMAIN, AND WE MUST BACKTRACK.        */

      R_EXPAND: PROC(CURR) RECURSIVE;
       DCL (P,CURR,ACTUAL) PTR;
       DCL BITPOS FIXED BIN(15);

       IF CURR->SUPER.LVL <= 0 THEN DO;
          /* HAVE HIT BOTTOM OF TREE. CHECK KEY HYPOTHESIS */
          /* DECIDE WHAT PLAINTEXT BIT IS REPRESENTED.
             (NOTE R3,R1,R-1 FOR LEFT BLOCK OF CIPHERTEXT */
          BITPOS=CURR->SUPER.POS;
          IF CURR->SUPER.LVL=0 THEN BITPOS=BITPOS+32;

          /* CHECK WITH PTEXT, BACKTRAVK ON =><= */
          IF CURR->SUPER.NODE->RNODE.VAL ~= PTEXT(BITPOS) THEN
             CALL BACKTRACK(CURR);
          RETURN;
       END;

       ACTUAL=CURR->SUPER.NODE;  /* SET ACTUAL TO PT TO THE NODE */

       /* IF NO EXPANSION ALTERNATIVES REMAIN, BACKTRACK */
       IF ACTUAL->RNODE.COUNT = 2 THEN CALL BACKTRACK(CURR);

       ELSE DO;  /* NODE HAS BEEN EXPANDED 011 TIMES */

          /* CREATE NEW LEFT SUBTREE */
          IF ACTUAL->RNODE.COUNT=0
           THEN P=CREATE_RNODE(CURR->SUPER.POS,CURR->SUPER.LVL-2,'1'B);
           ELSE DO;
              /* DELETE OLD RAMIFICATIONS */
              CALL DELETE((ACTUAL->RNODE.LPTR));
              CALL DELETE((ACTUAL->RNODE.RPTR));
              P=CREATE_RNODE(CURR->SUPER.POS,CURR->SUPER.LVL-2,'0'B);
           END;

           P->SUPER.FATHER=CURR; /* CHAIN LPTR INTO TREE */
           ACTUAL->RNODE.LPTR=P;
           CALL ADD_TO_OPEN(P);

           /* ALLOC AN F NODE OF VAL AS RPTR FROM RNODE.
              VALUE OF THE FNODE DEPENDS ON THE RNODE COUNT FIELD */
           P=CREATE('F');
           IF ACTUAL->RNODE.VAL = ACTUAL->RNODE.COUNT
              THEN P->SUPER.NODE->FNODE.VAL='1'B;
              ELSE P->SUPER.NODE->FNODE.VAL='0'B;

           /* INIT FIELDS OF THE F NODE */
```

```
PAUSE:
            P->SUPER.POS=P_PERM_INV(CURR->SUPER.POS);
            P->SUPER.LVL=CURR->SUPER.LVL;
            P->SUPER.FATHER=CURR;

             /* CHAIN THE RIGHT SUBTREE TO THE NEW RNODE */
             ACTUAL->RNODE.RPTR=P;
             CALL ADD_TO_OPEN(P);
             ACTUAL->RNODE.COUNT=ACTUAL->RNODE.COUNT+1;
          END;

        END R_EXPAND;
      1
       /* F_EXPAND: TO EXPAND AN FNODE. THIS MAY BE TRIED AS MANY TIMES
          AS THERE ARE CONJUNCT TERMS IN THE SP APPROX FOR THE S-BOX
          FUNCTION CORRESPONDING TO THE POSITION OF THE FNODE  */

        F_EXPAND: PROC(CURR);
        DCL (ACTUAL,CURR,P) PTR;
        DCL (TERM,K,BITPOS) FIXED BIN(15);

        ACTUAL=CURR->SUPER.NODE;
        /* LOOK AT NEXT DISJUNCTIVE POSSIBILITY FOR THE NODE */
        TERM,ACTUAL->FNODE.TERMNUM=ACTUAL->FNODE.TERMNUM+1;

        /* ESTABLISH WHICH OF THE 32 S-BANK OUTPUTS IS BEING CONSIDERED
*/
        BITPOS=P_PERM_INV(CURR->SUPER.POS);

         /* IF THE VALUE OF THE FNODE IS 1, WE USE THE S.P.
            REPRESENTATIONS FOR THE UNCOMPLEMENTED S-BOXES */

         IF ACTUAL->FNODE.VAL='1'B THEN DO;

           /* CHECK FOR NO DISJUNCTIVE POSSIBILITIES LEFT */
           IF TERM>23 | SPTERMS(BITPOS,TERM,1)=' ' THEN CALL BACKTRACK(C
URR);

           ELSE DO K=1 TO 6;

              /* DO NOT CHANGE THE XNODE SUBTREE IF:
                 1) SPTERM AT POSITION K IS A D.C.  OR
                 2) SPTERM AT POSITION K IS THE SAME AS IT WAS IN LAST T
ERM */
              IF SPTERMS(BITPOS,TERM,K)^='X'    &
           (TERM=1 | SPTERMS(BITPOS,TERM,K)^=SPTERMS(BITPOS,MAX(1,TERM-1
),K))
                 THEN DO;
```

```
PAUSE:
                    IF ACTUAL->FNODE.XPTR(K)~=NULL THEN /*KILL XNODE SUB
TREE*/
                       CALL DELETE((ACTUAL->FNODE.XPTR(K)));
                    P=CREATE('X');
                    ACTUAL->FNODE.XPTR(K)=P;
                    P->SUPER.FATHER=CURR;
                    P->SUPER.POS=K+ 6*TRUNC((BITPOS-1)/4);
                    P->SUPER.LVL=CURR->SUPER.LVL;
                    IF SPTERMS(BITPOS,TERM,K)='1'
                       THEN P->SUPER.NODE->XNODE.VAL='1'B;
                       ELSE P->SUPER.NODE->XNODE.VAL='0'B;
                    CALL ADD_TO_OPEN(P);
                 END;

                 ELSE IF SPTERMS(BITPOS,TERM,K)='X' &
                         ACTUAL->FNODE.XPTR(K)~=NULL
                      THEN CALL DELETE((ACTUAL->FNODE.XPTR(K)));

           END;
        END;
      1
       /* ELSE THE FNODE HAS VALUE 0, AND WE USE THE S.P.
          REPRESENTATIONS FOR THE COMPLEMENTED S-BOXES */

       ELSE DO;

           /* CHECK FOR NO DISJUNCTIVE POSSIBILITIES LEFT */
           IF TERM>23 | SPCTERMS(BITPOS,TERM,1)=' ' THEN CALL BACKTRACK(
CURR);

           ELSE DO K=1 TO 6;

              /* DO NOT CHANGE THE XNODE SUBTREE IF:
                1) SPCTERM AT POSITION K IS A D.C.  OR
                2) SPCTERM AT POSITION K IS THE SAME AS IT WAS IN LAST T
ERM */
              IF SPCTERMS(BITPOS,TERM,K)~='X'    &
          (TERM=1 | SPCTERMS(BITPOS,TERM,K)~=SPCTERMS(BITPOS,MAX(1,TERM-
1),K))
                 THEN DO;

                    IF ACTUAL->FNODE.XPTR(K)~=NULL THEN /*KILL XNODE SUB
TREE*/
                       CALL DELETE((ACTUAL->FNODE.XPTR(K)));
                    P=CREATE('X');
                    ACTUAL->FNODE.XPTR(K)=P;
                    P->SUPER.FATHER=CURR;
                    P->SUPER.POS=K+ 6*TRUNC((BITPOS-1)/4);
```

```
PAUSE:
                    P->SUPER.LVL=CURR->SUPER.LVL;
                    IF SPCTERMS(BITPOS,TERM,K)='1'
                       THEN P->SUPER.NODE->XNODE.VAL='1'B;
                       ELSE P->SUPER.NODE->XNODE.VAL='0'B;
                    CALL ADD_TO_OPEN(P);
                 END;

                 ELSE IF SPCTERMS(BITPOS,TERM,K)='X' &
                         ACTUAL->FNODE.XPTR(K)^=NULL
                      THEN CALL DELETE((ACTUAL->FNODE.XPTR(K)));

           END;
        END;

      END F_EXPAND;
     1
      /* X_EXPAND: TO EXPAND AN X NODE. 2 CHOICES FROM XOR ANALOGOUS TO
         RNODE EXPANSION, WITH THE EXCEPTION THAT ONLY ONE SUBTREE
         GROWS FROM AN XNODE, THE OTHER IS IN THE FORM OF A KEY BIT
         HYPOTHESIS */

       X_EXPAND: PROC(CURR) RECURSIVE;
        DCL (CURR,ACTUAL,P) PTR;
        DCL BITPOS FIXED BIN(15);
        DCL HYPSET CHAR(1);

        ACTUAL=CURR->SUPER.NODE;
        /* CHECK FOR NO MORE DISJUNCTIVE POSSIBILITIES */
        IF ACTUAL->XNODE.COUNT=2 THEN CALL BACKTRACK(CURR);

        ELSE DO;
           /* IF ONE EXISTS, DELETE OLD SUBTREE */
           IF ACTUAL->XNODE.COUNT=1 THEN CALL DELETE((ACTUAL->XNODE.RPT
R));
           IF ACTUAL->XNODE.VAL = ACTUAL->XNODE.COUNT
            THEN P=CREATE_RNODE(E_PERM_INV(CURR->SUPER.POS) ,
                                CURR->SUPER.LVL-1,'0'B);
            ELSE P=CREATE_RNODE(E_PERM_INV(CURR->SUPER.POS) ,
                                CURR->SUPER.LVL-1,'1'B);

           P->SUPER.FATHER=CURR;   /* CHAIN NEW NODE INTO TREE */
           ACTUAL->XNODE.RPTR=P;

           /* MAKE THE KEY HYPOTHESIS */

           /* DETERMINE POSITION OF AFFECTED KEY BIT */
           BITPOS=KEYPERM(CURR->SUPER.LVL,CURR->SUPER.POS);
           /* DETERMINE WHAT KEY BIT SHOULD BE, BASED ON COUNT */
```

```
PAUSE:
            IF ACTUAL->XNODE.COUNT=0 THEN HYPSET='0';
                                     ELSE HYPSET='1';
            ACTUAL->XNODE.COUNT=ACTUAL->XNODE.COUNT + 1;

            /* BACKTRACK IF THIS REPRESENTS A KEY BIT CONTRADICTION */
            IF KEY(BITPOS)~=' ' & KEY(BITPOS)~=HYPSET
               THEN CALL X_EXPAND(CURR); /* RETRY EXPANSION, COUNT INCR
*/
               ELSE DO;
                  KEY(BITPOS)=HYPSET;  /* SET KEY BIT */
                  CALL ADD_TO_OPEN(P); /* ADD NODE TO OPEN Q */
               END;
          END;

         END X_EXPAND;


        END SEARCH;
       //GO.SPFILE DD DSN=GULLICH.DES.SPTERMS,DISP=SHR
       //GO.SPCFILE DD DSN=GULLICH.DES.SPTERMSC,DISP=SHR
       //GO.KSFILE DD DSN=GULLICH.DES.KEYSCHED,DISP=SHR
       //GO.PCFILE DD DSN=GULLICH.DES.PCPAIRS,DISP=SHR
```

APPENDIX G

APL ROUTINES FOR DES ENCRYPTION

```
      ∇ CT←KEY DESΔENCRYPT PT;LVL;ROUNDS;L;R;RNEW;LNEW
[ 1]  ⍝ NOTE: NO IP OR IPINV PERMUTATIONS APPLIED.
[ 2]  ⍝ KEY, PT AND CT ARE ALL 64 BIT BINARY VECTORS
[ 3]  ⍝
[ 4]  ROUNDS←2
[ 5]  LVL←1
[ 6]  L←32↑PT
[ 7]  R←¯32↑PT
[ 8]  LOOP:LNEW←R
[ 9]  RNEW←L≠R F KEY[KEYSCHED[LVL;]]
[10]  L←LNEW
[11]  R←RNEW
[12]  →((LVL←LVL+1)≤ROUNDS)/LOOP
[13]  CT←R,L
      ∇


      ∇ Z←R F K;X;CT
[1]   Z←⍳0
[2]   X←K≠R[E]
[3]   CT←1
[4]   SLOOP:Z←Z,(4⍴2)⊤SBOX[⌈CT÷6;1+2⊥X[CT+ 0 5];1+2⊥X[CT+⍳4]]
[5]   →((CT←CT+6)<49)/SLOOP
[6]   Z←Z[P]
      ∇


      ∇ Z←GENΔKEYSCHED;KIDX;ROUND
[1]   ⍝ TO GEN 16×48 MTX OF INDICES FOR KEY BIT SELECTION
[2]   Z← 0 48 ⍴0
[3]   KIDX←⍳64
[4]   KIDX←KIDX[PC1]
[5]   ROUND←1
[6]   LOOP: ⍝ LEFT SHIFTS
[7]   KIDX[⍳28]←NUMLS[ROUND]⌽KIDX[⍳28]
[8]   KIDX[28+⍳28]←NUMLS[ROUND]⌽KIDX[28+⍳28]
[9]   Z←Z,[1] KIDX[PC2]
[0]   →((ROUND←ROUND+1)≤16)/LOOP
      ∇
```

```
      ∇ GEN△PCPAIRS;K;KEYS;KEY;PT;CT;T
[ 1]  PCPAIRS←KEYS← 0 64 ρ' '
[ 2]  K←1
[ 3]  LOOP:KEY←¯1+?64ρ2
[ 4]  PT←¯1+?64ρ2
[ 5]  CT←KEY DES△ENCRYPT PT
[ 6]  ⍝
[ 7]  ⍝ SAVE INTO GLOBAL SCHEDULES
[ 8]  PCPAIRS←PCPAIRS,[1](' '≠T)/T←⍕PT
[ 9]  'PLAINTEXT'
[10]  PPRINT PT
[11]  PCPAIRS←PCPAIRS,[1](' '≠T)/T←⍕CT
[12]  'INTO CIPHERTEXT'
[13]  PPRINT CT
[14]  KEYS←KEYS,[1](' '≠T)/T←⍕KEY
[15]  'BY KEY:'
[16]  PPRINT KEY
[17]  ''
[18]  →((K←K+1)≤5)/LOOP
[19]  ''
[20]  'KEYS:'
[21]  ''
[22]  (5 3 ρ 3 0 ⍕⍳5),' ',KEYS
      ∇

      ∇ PPRINT BIN;T
[1]   ⍝ TO PRETTY-PRINT A BINARY VECTOR IN BIN AND HEX
[2]   ((' '≠T)/T←⍕BIN),' ','0123456789ABCDEF'[1+16⊤2⊥⍉ 16 4 ρBIN]
      ∇
```

END OF APPENDIX

APPENDIX H

PROLOG SYSTEM FOR THE SYMBOLIC SIMPLIFICATION OF BOOLEAN EXPRESSIONS

```
$
/* PROLOG axioms and implications */
/* to simplify arbitrary boolean expression with ANDs (&),
  ORs (\), and NOTs (~) into 2 level sum-of-products */

/* top level simplification driver.
   pattern match with an s simplification pattern,repeatedly */

/* operator declarations */

?-op(8,fx,~).     /* logical negation */
?-op(9,xfx,&).     /* conjunction */
?-op(10,xfx,\).     /* disjunction */

simplify(Exp) :- simp(Exp,SExp) ,
printstring("SExp: "),nl,write(SExp),nl,display(SExp),nl,nl,
                  ((Exp=SExp , write(Exp))  ;
                   ( ! , simplify(SExp))).


simp(Exp,SExp)  :- s(Exp,SExp) .

simp(Larg\Rarg,Res)  :- simp(Larg,SLarg) , simp(Rarg,SRarg) , ! ,
      ( (Larg\Rarg = SLarg\SRarg , ! , bind(Res,Larg\Rarg) )  ;
      ( ! , simp(SLarg\SRarg,Res) )).

simp(Larg&Rarg,Res)  :- simp(Larg,SLarg) , simp(Rarg,SRarg) , ! ,
      ( (Larg&Rarg = SLarg&SRarg , ! , bind(Res,Larg&Rarg) );
      ( ! , simp(SLarg&SRarg,Res) )).

/* bottom out on trivial cases */
/* literal atoms can be simplified no further */

s(X,X) :- atomic(X) , ! .
s(~X,~X) :- atomic(X) , ! .

/* neither can top-level & or \ composed only of literals */
/* order the literals by quicksort to detect and transform:
   a&a->a   a&~a->0   a\a->a    a\~a->1              */

s(X,Y) :- mklist(X,Op,List) ,
          alliteral(List) , ! , /* X must be a one-level exprn */
          sortll(List,Slist,[]) ,
          scan(Op,Slist,Redlist) , /* form reduced list, Redlist */
          mklist1(Y,Op,Redlist).  /* back to expression form */


/* is a list all literals? */
alliteral([X|Y]) :- ! , literal(X) , ! , alliteral(Y).
alliteral(X) :- literal(X).
literal(X) :- atomic(X) , ! .
literal(~X) :- atomic(X) .

/* to sort a list of literals */
```

```
sortll([H|T],S,X) :- split(H,T,A,B) , ! ,
                     sortll(A,S,[H|Y]),
                     sortll(B,Y,X).
sortll([],X,X).

split(H,[A|X],[A|Y],Z) :- order(A,H) , split(H,X,Y,Z).
split(H,[A|X],Y,[A|Z]) :- order(H,A) , split(H,X,Y,Z).
split(_,[],[],[]).

/* order predicate determines if 2 literals are in correct order */
/* as name fails for integers 0 and 1 */
order(X,1) :- !.
order(X,0) :- !.
order(0,X) :- ! , fail.
order(1,X) :- ! , fail.
order(~X,~Y) :- ! , aless(X,Y).
order(~X,Y) :- ! , aless(X,Y).
order(X,~Y) :- ! , aless(X,Y).
order(X,Y) :- aless(X,Y).

/* alphabetic less-than predicate for atoms */
aless(X,Y) :- name(X,L) , name(Y,M) , alessx(L,M).
alessx([],[_|_]).
alessx([],[]).
alessx([H|X],[H|Y]) :- ! , alessx(X,Y).
alessx([X|_],[Y|_]) :- X=<Y.

/* scan: to look at sorted top level atoms 2-at-a-time,
   and perform appropriate reductions */

scan(&,(~A).A._,[0]) :- !.
scan(\,(~A).A._,[1]) :- !.
scan(&,A.0._,[0]) :- !.
scan(\,A.1._,[1]) :- !.
scan(Op,A.A.X,Y) :- ! , scan(Op,A.X,Y).
scan(Op,A.X,A.Y) :- ! , scan(Op,X,Y).
scan(_,A,A) :- atomic(A).


/* involution */

s(~(~X),Y)  :-  ! , simp(X,Y).

/* demorgan */

s( ~(X\Y) , Z)  :-  ! , simp(~X&(~Y) , Z).
s( ~(X&Y) ,Z)  :-  ! , simp(~X\(~Y) , Z).


/* driver routine to match combinations of top level \ or &
   2 terms at a time against patterns in s2 implications */

s(X,Y) :- mklist(X,Op,List), /* break formula X into a list */
```

```
        comb2(List,T1,T2,Remlist), /* select 2 terms from list */
        Poss =.. [Op,T1,T2] , s2(Poss,Simp),
        ! , append([Simp],Remlist,Simplist),
        mklist1(Y,Op,Simplist).



/* to break an expression with \ or & into a list */
mklist(~A,X,X) :- ! , fail.
mklist(Exp,Op,List) :-
                        /* make top lvl into a list */
                        mklist1(Exp,Op,Toplist),
                        mklistsub(Toplist,List,Op).

mklistsub([X],[X],Op) :- atomic(X) , ! .
mklistsub([X],[X],Op) :- ! , mklist1(X,Op2,_) , Op\=Op2.
mklistsub([X|Y],Res,Op) :- mklist(X,Op,L1) , ! ,
                           mklistsub(Y,L2,Op) , ! ,
                           append(L1,L2,Res).

/* terminator if X cannot be further done: */
mklistsub([X|Y],[X|L2],Op) :- ! , mklistsub(Y,L2,Op).

/* top level list breaker */
mklist1(A&B,&,A.R) :- mklist1(B,&,R).
mklist1(A\B,\,A.R) :- mklist1(B,\,R).
mklist1(X,&,[X]).
mklist1(X,\,[X]).

/* force pattern match */
bind(X,X).

/* to select all transpositions of combinations
   of objects from a list */

comb2(List,E1,E2,Rest) :- member(E1,List,Pos1),
                          member(E2,List,Pos2),
                          Pos1 \= Pos2,
                          remelt(E1,List,Temp),
                          remelt(E2,Temp,Rest).

/* to remove element from a list */

remelt(X,[],[]) :- !.
remelt(X,[X|Y],R)  :- ! ,  remelt(X,Y,R) .
remelt(X,[A|Y],[A|R])  :- ! ,  remelt(X,Y,R) .


/* list membership predicate, including position */


member(X,[X|_],1).
member(X,[_|Y],P1) :- member(X,Y,P) , P1 is P+1.
```

```
/* list append function */

append([],L,L).
append([X|L1],L2,[X|L3])  :- append(L1,L2,L3).


/* 2-at-a-time pattern match implications */


s2( X\0 , Y)  :-   ! , simp(X,Y).
s2( X\1 , 1)  :-  !.
s2( X&1  ,Y)  :-   ! , simp(X,Y).
s2( X&0 , 0)  :-  !.

/* idempotent */

s2( X\X , Y)  :-   ! , simp(X,Y).
s2( X&X , Y)  :-   ! , simp(X,Y).


/* complementarity */

s2( X&(~X) , 0)  :-  !.
s2( X\(~X) , 1)  :-  !.


/* distributivity 1 */

s2( (A\B)&(A\C) , Y)  :-   ! , simp( A\(B&C) , Y).
s2( (B\A)&(A\C) , Y)  :-   ! , simp( A\(B&C) , Y).
s2( (A\B)&(C\A) , Y)  :-   ! , simp( A\(B&C) , Y).
s2( (B\A)&(C\A) , Y)  :-   ! , simp( A\(B&C) , Y).


/* absorption */

s2( A&B\A&(~B) , Y)  :-   ! , simp(A,Y).
s2( B&A\A&(~B) , Y)  :-   ! , simp(A,Y).
s2( A&B\(~B)&A , Y)  :-   ! , simp(A,Y).
s2( B&A\(~B)&A , Y)  :-   ! , simp(A,Y).

s2( A\A&B , Y)  :-   ! , simp(A,Y).
s2( A\B&A , Y)  :-   ! , simp(A,Y).

s2( A&(A\B) , Y)  :-   ! , simp(A,Y).
s2( A&(B\A) , Y)  :-   ! , simp(A,Y).

s2( A&(~B)\B , Y)  :-   ! , simp(A\B,Y).
s2( (~B)&A\B , Y)  :-   ! , simp(A\B,Y).

/* consensus (only 1- that with 2 terms on lhs */
```

```
s2( (X\Y)&(~X\Z) , A)  :-   ! , simp(X&Z\(~X)&Y , A).
s2( (Y\X)&(~X\Z) , A)  :-   ! , simp(X&Z\(~X)&Y , A).
s2( (X\Y)&(Z\(~X)) , A)  :-   ! , simp(X&Z\(~X)&Y , A).
s2( (Y\X)&(Z\(~X)) , A)  :-   ! , simp(X&Z\(~X)&Y , A).

/* distributivity 2- last as it makes more terms */

s2( X&(Y\Z) , A)  :-  ! , simp(X&Y\X&Z ,A).

/*zoom a&b&c \ a&b&x&c -> a&b&c kludge */

s2(A\B,S) :-
              mklist(A,&,Al) , alliteral(Al) , ! ,
              mklist(B,&,Bl) , alliteral(Bl) , ! ,
              ((subset(Al,Bl) , ! , simp(A,S));
              (subset(Bl,Al) , ! , simp(B,S))).

subset([],Y) :- !.
subset([A|X],Y) :- mem(A,Y) , subset(X,Y).

mem(X,[X|_]) :- !.
mem(X,[_|Y]) :- mem(X,Y).

/* utilities */
printstring([]).
printstring([H|T]) :- put(H) , printstring(T).

test1 :- simplify(~((~(a\b))\(~(c\d))\(~(e\f)))).

test2 :- simplify(~( ~a&(~b) \ (~c&(~d)) )).

test3 :- simplify( (a&(~b)&c&(~d)&e \
                    (~a)&(~b)&(~e)&f \
                    (~a)&(~b)&c&(~d) )   &
                   (a&(~d) \
                    (~a)&(~b)&c&(~d) \
                    (~a)&b&(~c)&d&(~f) )).

/* editor utility */
ed :- shell("ed min"),[$min].
```

```
                TEST OF SYMBOLIC BOOLEAN SIMPLIFIER
                ***********************************

$ prolog
PROLOG Version NU7.1

?- [min].

min consulted.

yes
?- listing(test1).

test1
     :- simplify(~(~(a\b)\~(c\d)\~(e\f))).

yes
?- test1.

SExp:
a&c&e\b&c&e\a&d&e\b&d&e\a&c&f\b&c&f\a&d&f\b&d&f
\(\(\(&(a,&(c,e)),&(b,&(c,e))),\(&(a,&(d,e)),&(b,&(d,e)))),
\(\(&(a,&(c,f)),&(b,&(c,f))),\(&(a,&(d,f)),&(b,&(d,f)))))

SExp:
a&c&e\b&c&e\a&d&e\b&d&e\a&c&f\b&c&f\a&d&f\b&d&f
\(\(\(&(a,&(c,e)),&(b,&(c,e))),\(&(a,&(d,e)),&(b,&(d,e)))),
\(\(&(a,&(c,f)),&(b,&(c,f))),\(&(a,&(d,f)),&(b,&(d,f)))))

a&c&e\b&c&e\a&d&e\b&d&e\a&c&f\b&c&f\a&d&f\b&d&f
yes.

?- listing(test2).

test2
     :- simplify(~(~a&~b\~c&~d)).

yes
?- test2.

SExp:
a&c\b&c\a&d\b&d
\(\(&(a,c),&(b,c)),\(&(a,d),&(b,d)))

SExp:
a&c\b&c\a&d\b&d
\(\(&(a,c),&(b,c)),\(&(a,d),&(b,d)))

a&c\b&c\a&d\b&d
yes
?-
```

APPENDIX I

APL CODE FOR AND/OR TREE FORMATION AND TRAVERSAL

```
      ∇  Z←X AND Y;XMAX;YMAX;XC;YC;NEW;ONEXWITHY
[ 1]  →(0≠(ρX)[2])/NOTNULLX
[ 2]  Z←Y
[ 3]  →0
[ 4]  NOTNULLX:XMAX←(ρX)[2]
[ 5]  YMAX←(ρY)[2]
[ 6]  Z←(2,0,¯1↑ρX)ρ0
[ 7]  XC←1
[ 8]  XLOOP:ONEXWITHY←(2,0,¯1↑ρX)ρ0
[ 9]  YC←1
[10]  YLOOP:→(∨/∧/NEW←X[;,XC;]∨Y[;,YC;])/NEXTY
[11]  ONEXWITHY←ONEXWITHY OR NEW
[12]  NEXTY:→((YC←YC+1)≤YMAX)/YLOOP
[13]  →(0=(ρONEXWITHY)[2])/NEXTX
[14]  Z←Z OR ONEXWITHY
[15]  NEXTX:→((XC←XC+1)≤XMAX)/XLOOP
      ∇
```

```
      ∇  Z←X AND1 Y
[]    Z←TO∆1X0(TO∆BIT X) AND TO∆BIT Y
      ∇
```

```
        ∇ TYPE BUILDSUB AT∆LVL;AT;LVL;VAL;SP;NUMSP;TERMNUM;T;K;LOC
TERM;POSN
[ 1]    AT←AT∆LVL[1]
[ 2]    LVL←AT∆LVL[2]
[ 3]    ⍝ CHOOSE TYPE OF NODE TO EXPAND
[ 4]    →((1↑TYPE)='T','P','X','R')/TOP,PTERM,XTERM,RTERM
[ 5]    ⍝
[ 6]    ⍝ INITIALIZE GLOBAL TREE IN PARALLEL VECS
[ 7]    TOP:TREE∆TYPE← 10000 2 ⍴0
[ 8]    TREE∆PTR←10000⍴0
[ 9]    TREE∆NKIDS←10000⍴0
[10]    FREE←2
[11]    VAL←CTEXT[POS+32×NROUNDS=2]≠PTEXT[POS+32×NROUNDS=2]
[12]    RPOS←POS
[13]    →REXPAND
[14]    ⍝
[15]    ⍝ ----------------------------------------
[16]    PTERM:TREE∆TYPE[AT;]←∆AND
[17]    TREE∆PTR[AT]←FREE
[18]    TREE∆NKIDS[AT]←+/TERM≠'X'
[19]    POSN←FREE
[20]    FREE←FREE++/TERM≠'X'
[21]    ⍝ LOOP FOR ALL LITERALS IN TERM
[22]    K←0
[23]    LOCTERM←TERM
[24]    LITLOOP:→(LOCTERM[K+1]='X')/NEXTLIT
[25]    ⍝ SET PARMS FOR XTERM TO BE EXPANDED
[26]    RPOS←E[SBOXINP+K]
[27]    KNUM←KEYSCHED[LVL;SBOXINP+K]
[28]    XVAL←LOCTERM[K+1]='1'
[29]    'XTERM' BUILDSUB(POSN+K),LVL
[30]    NEXTLIT:→((K←K+1)<6)/LITLOOP
[31]    →0
[32]    ⍝
[33]    ⍝ ----------------------------------------
[34]    XTERM:→(LVL=1)/BOTTOMOUT
[35]    TREE∆TYPE[AT;]←∆OR
[36]    TREE∆PTR[AT]←FREE
[37]    TREE∆NKIDS[AT]←2
[38]    ⍝ LEFT SUBTREE OF XOR
[39]    TREE∆TYPE[FREE;]←∆AND
[40]    TREE∆PTR[FREE]←FREE+2
[41]    TREE∆NKIDS[FREE]←2
[42]    ⍝ RIGHT SUBTREEE OF XOR
[43]    TREE∆TYPE[FREE+1;]←∆AND
[44]    TREE∆PTR[FREE+1]←FREE+4
[45]    TREE∆NKIDS[FREE+1]←2
[46]    FREE←FREE+6
[47]    POSN←FREE
```

```
[48]    ⍝ EXPAND RTERMS
[49]    RVAL←1
[50]    'RTERM' BUILDSUB(POSN-4),LVL-1
[51]    RVAL←0
[52]    'RTERM' BUILDSUB(POSN-2),LVL-1
[53]    ⍝ SETUP KEY HYP FROM LEFT SUBTREE
[54]    TREE∆TYPE[POSN-3;]←∆KEY
[55]    TREE∆PTR[POSN-3]←1+(⍴KEY)[2]
[56]    KEY←KEY,[2] 2 64 ⍴0
[57]    KEY[;(⍴KEY)[2];KNUM]←~XVAL,~XVAL
[58]    ⍝ SETUP KEY HYP FROM RIGHT SUBTREE
[59]    TREE∆TYPE[POSN-1;]←∆KEY
[60]    TREE∆PTR[POSN-1]←1+(⍴KEY)[2]
[61]    KEY←KEY,[2] 2 64 ⍴0
[62]    KEY[;(⍴KEY)[2];KNUM]←XVAL,~XVAL
[63]    →0
[64]    BOTTOMOUT:TREE∆TYPE[AT;]←∆KEY
[65]    TREE∆PTR[AT]←1+(⍴KEY)[2]
[66]    KEY←KEY,[2] 2 64 ⍴0
[67]    KEY[;(⍴KEY)[2];KNUM]←T,~T←PTEXT[RPOS+32]≠XVAL
[68]    →0
[69]    ⍝
[70]    ⍝ ----------------------------------------
[71]    ⍝ RPOS ESTAB IN PTERM OR TOP
[72]    RTERM:VAL←RVAL≠PTEXT[RPOS]
[73]    REXPAND:→(~VAL)/COMPL
[74]    SP←SPTERMS[⌈P[RPOS]÷4;1+4|P[RPOS]-1;;]
[75]    →JOIN
[76]    COMPL:SP←SPCTERMS[⌈P[RPOS]÷4;1+4|P[RPOS]-1;;]
[77]    JOIN:NUMSP←+/' '≠SP[;1]
[78]    TREE∆TYPE[AT;]←∆OR
[79]    TREE∆PTR[AT]←FREE
[80]    TREE∆NKIDS[AT]←NUMSP
[81]    POSN←FREE
[82]    FREE←FREE+NUMSP
[83]    SBOXINP←1+6×⌊(P[RPOS]-1)÷4
[84]    TERMNUM←0
[85]    TERMLOOP:TERM←SP[TERMNUM+1;]
[86]    'PTERM' BUILDSUB(POSN+TERMNUM),LVL
[87]    →((TERMNUM←TERMNUM+1)<NUMSP)/TERMLOOP
[88]    →0
        ∇
```

```
      ∇  KEY←DECRYPT PCPAIR;POS;TOP;MASK;NROUNDS;PTEXT;CTEXT
[ 1]  PTEXT←PCPAIR[1;]
[ 2]  CTEXT←PCPAIR[2;]
[ 3]  ⍝ LEFTMOST 32 BITS HAVE BEEN ENCRYPTED IN 2 ROUNDS
[ 4]  NROUNDS←2
[ 5]  KEY← 2 0 64 ⍴0
[ 6]  ⍝ BUILD AND TRAVERSE AND/OR TREE FOR EACH BIT OF CTEXT
[ 7]  POS←1
[ 8]  LOOP:'TOP' BUILDSUB 1,NROUNDS
[ 9]  MASK←TRAVERSE 1
[10]  KEY←KEY AND MASK
[11]  POS←POS+1
[12]  NROUNDS←NROUNDS-POS=33
[13]  POS←POS-32×POS=33
[14]  →(POS≤64)/LOOP
      ∇


      ∇  DUMP
[1]   ⍝ DUMP ALL TREE PARALLEL VECTORS
[2]   K←1
[3]   LOOP:TREE K
[4]   →((K←K+1)≤⍴TREE∆PTR)/LOOP
      ∇


     ∇  X←GEN
[]   X←'01X'[?((1+?3),4)⍴3]
     ∇


      ∇  Z←MIN1 INTERSECT MIN2
[1]   ⍝ SET INTERSECTION OF 2 VECTORS OF MINTERM NUMBERS
[2]   Z←(∨/MIN1∘.=MIN2)/MIN1
      ∇
```

```
        ∇  Z←X OR Y;XMAX;YMAX;XTAKE;YTAKE;XC;YC;XCUBE;YCUBE;C;CY;CX
;CXEQX;CYEQY
[ 1]    →(0≠(ρX)[2])/NOTNULLX
[ 2]    Z←Y
[ 3]    →0
[ 4]    NOTNULLX:→(3=ρρX)/OKX
[ 5]    X←((1↑ρX),1,¯1↑ρX)ρX
[ 6]    OKX:→(3=ρρY)/OKY
[ 7]    Y←((1↑ρY),1,¯1↑ρY)ρY
[ 8]    OKY:XMAX←(ρX)[2]
[ 9]    YMAX←(ρY)[2]
[10]    XTAKE←XMAXρ1
[11]    YTAKE←YMAXρ1
[12]    ⍝
[13]    XC←1
[14]    XLOOP:XCUBE←X[;XC;]
[15]    YC←1
[16]    YLOOP:→(~YTAKE[YC])/NEXTY
[17]    YCUBE←Y[;YC;]
[18]    C←XCUBE∨YCUBE
[19]    ⍝ CHECK IF C SAME AS EITHER ORIGINAL CUBE
[20]    →(~∧/∧/C=XCUBE)/CHECKY
[21]    XTAKE[XC]←0
[22]    →NEXTX
[23]    CHECKY:→(~∧/∧/C=YCUBE)/CONSENS
[24]    YTAKE[YC]←0
[25]    →NEXTY
[26]    ⍝ CHECK IF C IS A CONSENSUS TERM:
[27]    CONSENS:→(1≠+/T←∧≠C)/NEXTY
[28]    C[1;]←C[1;]-T
[29]    C[2;]←C[2;]-T
[30]    CX←C∨XCUBE
[31]    CY←C∨YCUBE
[32]    CXEQX←∧/∧/CX=XCUBE
[33]    CYEQY←∧/∧/CY=YCUBE
[34]    →(~CXEQX∧CYEQY)/A1
[35]    XTAKE[XC]←0
[36]    Y[;YC;]←C
[37]    →NEXTX
[38]    A1:→(~(~CXEQX)∧CYEQY)/A2
[39]    Y[;YC;]←C
[40]    →NEXTY
[41]    A2:→(~CXEQX∧~CYEQY)/NEXTY
[42]    X[;XC;]←C
[43]    ⍝ XCUBE←X[;XC;]←C  →XLOOP   ?
[44]    NEXTY:→((YC←YC+1)≤YMAX)/YLOOP
[45]    NEXTX:→((XC←XC+1)≤XMAX)/XLOOP
[46]    Z←(XTAKE/[2] X),[2] YTAKE/[2] Y
        ∇
```

```
      ∇  Z←X OR1 Y
[]    Z←TO∆1X0(TO∆BIT X) ORVERB TO∆BIT Y
      ∇


      ∇  PP
[]    0 PP1 1
      ∇


        ∇  INDENT PP1 AT;K
[ 1]    →(TREE∆TYPE[AT;]∧.= 2 4 ρ 1 0 1 0 0 1 1 0)/AND,OR,KEY,NULL
[ 2]    AND:(INDENTρ' '),'AND'
[ 3]    →JOIN
[ 4]    OR:(INDENTρ' '),'OR'
[ 5]    JOIN: ⍝ D1ST RECURSE
[ 6]    K←0
[ 7]    LOOP:(INDENT+3) PP1 TREE∆PTR[AT]+K
[ 8]    →((K←K+1)<TREE∆NKIDS[AT])/LOOP
[ 9]    →0
[10]    KEY:(INDENTρ' '),T,' KEY BIT ',▼('X'≠T←,TO∆1X0 KEY[;TREE∆PT
R[AT];])⍳1
[11]    →0
[12]    NULL:(INDENTρ' '),'  ⌽'
        ∇


      ∇  TESTAND;X;Y;R
[1]   □←X←GEN
[2]   ''
[3]   □←Y←GEN
[4]   ''
[5]   R←X AND1 Y
[6]   ''
[7]   'RESULT OF AND:'
[8]   R
[9]   ⍝ TEST THAT RESULT IS INTERSECTION OF THE 2 COVERS
[0]   (∧/(ONFOR R)=(ONFOR X) INTERSECT ONFOR Y)/'*** SUCCESS ***'
      ∇
```

```
      ∇ TESTOR;X;Y;R
[1]   □←X←GEN
[2]   ''
[3]   □←Y←GEN
[4]   ''
[5]   R←X OR1 Y
[6]   ''
[7]   'RESULT OF OR:'
[8]   R
[9]   ⍝ TEST THAT RESULT IS UNION OF THE 2 COVERS
[0]   (∧/(ONFOR R)=(ONFOR X) UNION ONFOR Y)/'*** SUCCESS ***'
      ∇


      ∇ Z←TO∆BIT X
[]    Z←(X='1'),[0.5] X='0'
      ∇


      ∇ Z←TO∆1X0 X
[1]   →(3=ρρX)/OK
[2]   ⍝ HAVE BEEN GIVEN JUST 1 CUBE
[3]   X←((1↑ρX),1,¯1↑ρX)ρX
[4]   OK:X[2;;]←X[2;;]∨X[1;;]
[5]   Z←'X01'[1++⌿X]
      ∇
```

```
      ∇  RES←TRAVERSE ROOT;NUMKIDS;K
[ 1]  ⍝ TO TRAVERSE AND/OR TREE AND RETURN KEY CONSTRAINT
[ 2]  TYPE←TREE∆TYPE[ROOT;]
[ 3]  →(∧/TYPE=∆AND)/AND
[ 4]  →(∧/TYPE=∆OR)/OR
[ 5]  →(∧/TYPE=∆KEY)/KEY
[ 6]  'ERRONEOUS TREE TYPE'
[ 7]  0÷0
[ 8]  AND:NUMKIDS←TREE∆NKIDS[ROOT]
[ 9]  K←0
[10]  RES← 2 0 64 ⍴0
[11]  ANDLOOP:RES←RES AND TRAVERSE TREE∆PTR[ROOT]+K
[12]  →((K←K+1)<NUMKIDS)/ANDLOOP
[13]  →0
[14]  OR:NUMKIDS←TREE∆NKIDS[ROOT]
[15]  K←0
[16]  RES← 2 0 64 ⍴0
[17]  ORLOOP:RES←RES OR TRAVERSE TREE∆PTR[ROOT]+K
[18]  →((K←K+1)<NUMKIDS)/ORLOOP
[19]  →0
[20]  KEY:RES← 2 1 64 ⍴KEY[;TREE∆PTR[ROOT];]
      ∇


      ∇  Z←TREE L
[1]   ⍝ PRINTOUT ONE SLICE OF THE PARALLEL
[2]   ⍝ VECTORS WHICH REP THE TREE
[3]   Z←(,(TREE∆TYPE[L;]∧.= 2 4 ⍴ 1 0 1 0 0 1 1 0)⌿ 4 3 ⍴'ANDOR KE
Y⍀  '),' ',(⍕TREE∆PTR[L]),' ',⍕TREE∆NKIDS[L]
      ∇


      ∇  Z←MIN1 UNION MIN2
[1]   ⍝ SET UNION OF 2 VECTORS OF MINTERM NUMBERS
[2]   Z←Z[⍋Z←MIN1,MIN2]
[3]   Z←((Z⍳Z)=⍳⍴Z)/Z
      ∇
```

END OF APPENDIX

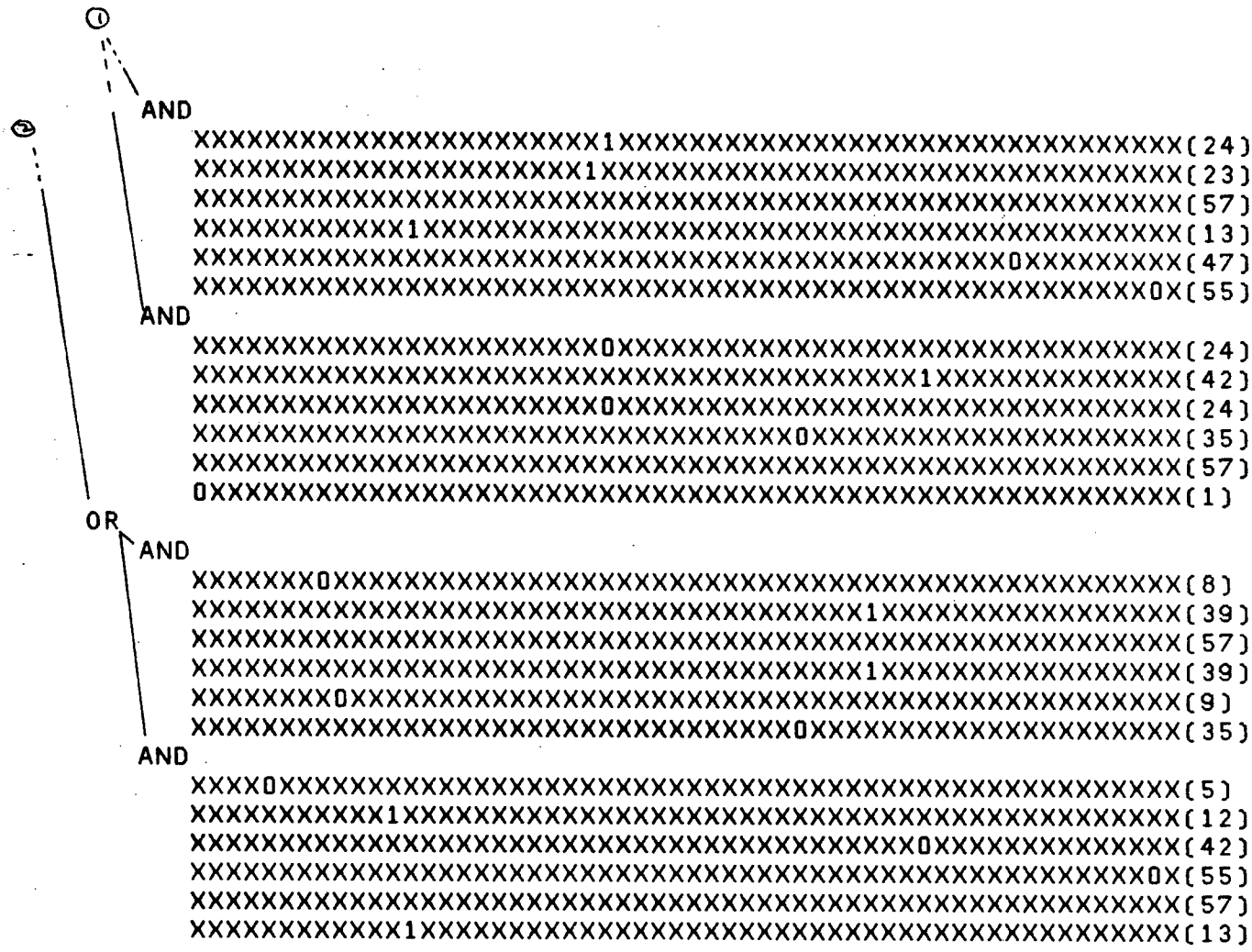
EXAMPLE OF TREE TRAVERSAL
****

V S A P L

```
CLEAR WS
      )LOAD SEARCH
SAVED 23:18:53 02/05/83
WSSIZE IS 1890896

/* DISPLAY THE TREE STRUCTURE */

      PP
OR
  AND
    OR
      AND
        XXXXXXXXXXXXXXXXXXXXXXXXXXX1XXXXXXXXXXXXXXXXXXXXXXXXXXX(28)
        XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX1X(55)
        XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX0XXXXXXXX(48)
        XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX0(56)
        XXXXXXXX1XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX(9)
        XXXXXXXXXXXXXXXXXXXXXXX0XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX(23)
      AND
        XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX(44)
        XXXXXXXXXXXXXXXXXXXXXXXXXX1XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX(27)
        XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX(36)
        XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX(57)
        XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX(16)
        XXXXXXXXXX1XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX(11)
    OR
      AND
        XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX(45)
        XXXX1XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX(5)
        XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX(48)
        XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX(54)
        XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX(54)
        XXXXXXXXXXXXXXXXXXXX1XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX(22)
      AND
        XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX(48)
        XXXXXXXXXXXXX1XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX(14)
        XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX(57)
        XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX(55)
        XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX(57)
        XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX(43)
  AND
    OR
      ①
      ②
```




```
/* TRAVERSE THE TREE */


      TO_1X0 TRAVERSE 1
XXXXXXXX1XXXXXXXXXXX10XXXX1XXXXXXXXXXXXXXXXXXXXXXXXXXX10
XXXXXXXX1XXXXXXXXXXXX0XXXX1XXXXXXXXXXXXXXXXXXXXXXXXXXX10
XXXXXXXXXX1XXXX0XXXXX1XXXX1XXXXXXXX0XXXXXXX0XXXXXXXXXXX
XXXXXXXXXX1XXXX0XXXXXXXXXX1XXXXXXXX0XXXXXX00XXXXXXXXXXX
XXXXXXX00XXX1XXXXXXXXX11XXXXXXXXXX0XXX1XXXXXXX0XXXXXXX0X
XXXX0XXXXXX11XXXXXXXXX11XXXXXXXXXXXXXXXXX0XXXX0XXXXXXX0X
0XXXXXX00XXXXXXXXXXXXX0XXXXXXXXXX0XXX1XX1XXXXXXXXXXXXXX


      )OFF
```