AN APPLICATION OF A FAST DATA ENCRYPTION STANDARD IMPLEMENTATION

Matt Bishop

Technical Report PCS-TR88-138

An Application of a Fast Data Encryption Standard Implementation

Matt Bishop

Department of Mathematics and Computer Science
Dartmouth College
Hanover, NH 03755
and
Research Institute for Advanced Computer Science
NASA Ames Research Center
Moffett Field, CA 94035

ABSTRACT

The Data Encryption Standard is used as the basis for the UNIX password encryption scheme. Some of the security of that scheme depends on the speed of the implementation. This paper presents a mathematical formulation of a fast implementation of the DES in software, discusses how the mathematics can be translated into code, and then analyzes the UNIX password scheme to show how these results can be used to implement it. Experimental results are provided for several computers to show that the given method speeds up the computation of a password by roughly 20 times (depending on the specific computer.)

1. Introduction

Passwords and encryption schemes are used to prevent unauthorized access to files and data as well as to deter break-ins to computer systems. The Numerical Aerodynamic Simulator project at NASA'a Ames Research Center administers approximately 40 UNIX-based† computers on a local area network which is connected to both long-haul networks and telephone lines. Password security is of paramount importance at this site since, with knowledge of a secret password, unknown parties can gain access to individual accounts and to the system as a whole.

This work was supported by NASA grant NCC 2-398 from the National Aeronautics and Space Administration (NASA) to the Research Institute for Advanced Computer Science (RIACS). Part of the work was done while the author was at RIACS, This report has also been issued as TR 87.18 (revised) by the Research Institute for Advanced Computer Science.

†UNIX is a registered trademark of AT&T Bell Laboratories.

For these reasons, it is desirable to have a password checking program to examine encrypted passwords and determine whether they can be guessed‡. Such a program obtains a list of potential passwords, such as a list of words, the user's names, initials, login name, and so forth, and encrypts each one using additional information (called "the salt") associated with the user whose password is being tested. If the result matches the encrypted password (called a "hit"), the password has been compromised and the user is requested to change it.

On the surface, the UNIX password scheme appears to be quite secure [10] It uses a variant of the Data Encryption Standard [11] to provide a one-way encryption function. This function depends both on the password and on a two-character salt, so in effect there are 4096 different possible encryption functions. This is intended to discourage attacks in which the dictionary is encrypted and compared to a list of passwords; if a list of 100 passwords each have a different salt, it would take about 220 days to test a dictionary of 25,000 words against the list. Of course, only encrypted passwords and their salts are stored on line; when the user types his or her password, it is encrypted using the salt associated with that user, and compared to the on line encrypted password. Validation occurs if there is a match.

Properties of the DES have been discussed in detail [8,9], for example. Much analysis has been done to determine how best to implement the DES, both in hardware [6,7] and software [6]. In these papers, and in some studies of the DES algorithm itself [4,5], a number of useful transformations, all of which show various strengths and weaknesses of the algorithm, are discussed from the point of view of speeding up implementations.

In this paper, we describe these earlier results using mathematical notation. After that, we shall discuss some practical hints for implementors, and then analyze the UNIX password algorithm to see how using the salt and the repeated application of the DES algorithm affects the analysis. More implementation hints will be given, and finally some timings of routines that embody the hits will be discussed.

A word about notation. In both the theoretical and application sections of this paper, certain operations occur very often. In all sections, $\neg$ means "bitwise negation", $\&$ means "bitwise and", $|$ means "bitwise (inclusive) or", and $\oplus$ means "bitwise exclusive or".

---

‡It would be even more desirable to do rigorous password checking when the user sets his or her password. Unfortunately, to implement this would require changing vendor-supplied software, and add to the maintenance burdens. Also, when the new password programs were installed, everyone would have to change their passwords to ensure they meet the new (presumably more rigorous) standards; forcing users to do this would cause serious problems.

2. Analysis of the DES Algorithm

The DES algorithm consists of a series of expansions, permutations, and substitutions. In this section, η will represent an expansion (or contraction), π a permutation, and σ a substitution. The DES function uses a series of tables to indicate how the operation is to take place; a subscript will indicate which table is used by the operation. Some tables, such as the PC-1 table, indicate both an expansion (in this case, a contraction) and a permutation. Strictly speaking, such a table should be a subscript on the product of an expansion and a permutation, that is $(\pi\eta)_{PC\text{-}1}$; but this is confusing, so we shall just represent this as $\Pi_{PC\text{-}1}$.

2.1. Derivation of the Key Schedule

The DES first computes a "key schedule" of 16 elements from a 56 bit key. Let the key be K. (Actually, K is 64 bits, but the first permutation discards 8 parity bits.) Then $\Pi_{PC\text{-}1}$ is applied:

$$K^{(0)} = \Pi_{PC\text{-}1} K \tag{1}$$

The next two steps are applied 16 times, the output of the first being the input of the next set; i is the iteration number, beginning with 0. $K^{(i)}$ is divided into two halves, each half is shifted left as indicated by the table LSH. (This is really a permutation, so we shall write it as π_{LSH}.) Thus,

$$K^{(i)} = \pi_{LSH} K^{(i-1)} \qquad \text{for } i = 1, \ldots, 16 \tag{2}$$

These intermediate keys are transformed by $\Pi_{PC\text{-}2}$ to generate a new element of the key schedule k_i:

$$k_i = \Pi_{PC\text{-}2} K^{(i)} \qquad \text{for } i = 1, \ldots, 16 \tag{3}$$

So, there are 16 elements $k_1, \ldots, k_{16}$ of the key schedule.

Now consider equation (2). By iterating the recurrence and substituting (1), we have

$$K^{(i)} = \pi_{LSH}^{i} K^{0} = \pi_{LSH}^{i} \Pi_{PC\text{-}1} K \tag{4}$$

and so by (3) and (4),

$$k_i = \Pi_{PC\text{-}2} \pi_{LSH}^{i} K^{0} = \Pi_{PC\text{-}2} \pi_{LSH}^{i} \Pi_{PC\text{-}1} K \tag{5}$$

Defining G_i by $G_i = \Pi_{PC\text{-}2} \pi_{LSH}^{i} \Pi_{PC\text{-}1}$, (5) may be rewritten

$$k_i = G_i K \tag{6}$$

We shall refer to the set of G_i as G; that is, $G = \{G_i \mid i = 1, \ldots, 16\}$.

2.2. Encryption of the Message

Now let $m = m_1 \cdots m_{64}$ be the message to be encrypted. First an initial permutation is applied:

$$T_0 = t_1 \cdots t_{64} = \pi_{IP} m \tag{7}$$

Divide the 64 bits of T_0 into two halves, $l_0 = t_1 \cdots t_{32}$ and $r_1 = t_{33} \cdots t_{64}$. Then, the next 16 steps are the same, the output of each being used as the input to its successor. For rounds $i = 0, \ldots, 15$,

$$l_{i+1} = r_i \tag{8a}$$

$$r_{i+1} = l_i \oplus \pi_P \sigma_S (\eta_E r_i \oplus k_i) \tag{8b}$$

Finally, the halves of the result $T_{16} = l_{16} r_{16}$ are exchanged and the inverse of the initial permutation is applied:

$$x = \pi_{IP}^{-1} (r_{16} l_{16}) \tag{9}$$

Now, consider equations (8a) and (8b). Since both l_i and r_i are 32 bit vectors, apply η_E to both sides:

$$\eta_E l_{i+1} = \eta_E r_i \tag{10a}$$

$$\eta_E r_{i+1} = \eta_E (l_i \oplus \pi_P \sigma_S (\eta_E r_i \oplus k_i)) \tag{10b}$$

As $\oplus$ is a homomorphism from the group of bit vectors in the range of η_E to itself, the $\oplus$ may be taken outside the expansion, yielding

$$\eta_E r_{i+1} = \eta_E l_i \oplus \eta_E \pi_P \sigma_S (\eta_E r_i \oplus k_i) \tag{11}$$

Now define $L_i = \eta_E l_i$ and $R_i = \eta_E r_i$. By (10a) and (11),

$$L_{i+1} = R_i$$

$$R_{i+1} = L_i \oplus \eta_E \pi_P \sigma_S (R_i \oplus k_i)$$

These imply

$$L_{i+2} = L_i \oplus \eta_E \pi_P \sigma_S (R_i \oplus k_i) \tag{12a}$$

$$\begin{aligned} R_{i+2} &= R_i \oplus \eta_E \pi_P \sigma_S (R_{i+1} \oplus k_i) \\ &= R_i \oplus \eta_E \pi_P \sigma_S (L_i \oplus \eta_E \pi_P \sigma_S (R_i \oplus k_i) \oplus k_{i+1}) \end{aligned} \tag{12b}$$

$$R_{i+2} = R_i \oplus \eta_E \pi_P \sigma_S (L_{i+2} \oplus k_{i+1}) \tag{13}$$

Define $F_2 = \eta_E \pi_P \sigma_S$. Then (12a) and (13) become

$$L_{i+2} = L_i \oplus F_2 (R_i \oplus k_i) \tag{14a}$$

$$R_{i+2} = R_i \oplus F_2 (L_{i+2} \oplus k_{i+1}) \tag{14b}$$

Now consider (7). (14a) and (14b) require that $\eta_E l_0$ and $\eta_E r_0$ be found, so define η_{EE} to be the expansion of a 64 bit vector to a 96 bit vector by splitting the vector in half and applying η_E to each half. Then to obtain L_0 and R_0, define $T' = L_0 R_0$; and

$$T' = \eta_{EE} \pi_{IP} m$$

Defining $F_1 = \eta_{EE} \pi_{IP}$, this becomes

$$T' = F_1 m \tag{15}$$

Finally, look at (9). Exchanging the halves l_{16} and r_{16} may be treated as a permutation π_X. Also, note that by definition L_i and R_i are in the range of η_E, so both $\eta_E^{-1}L_i$ and $\eta_E^{-1}R_i$ exist and are unique. Let $\eta_{EE}^{-1}(L_{16}R_{16})$ represent the result of concatenating $\eta_E^{-1}L_{16}$ and $\eta_E^{-1}R_{16}$. (This is actually the inverse of the η_{EE} discussed above.) Combining all this with (9), we have

$$x = \pi_{IP}^{-1}\pi_X\eta_{EE}^{-1}(L_{16}R_{16})$$

Defining $F_3 = \pi_{IP}^{-1}\pi_X\eta_{EE}^{-1}$, this becomes

$$x = F_3(L_{16}R_{16}) \tag{16}$$

3. Application of the Analysis

With this analysis, we are ready to implement a fast version of the DES algorithm. We will consider several aspects of the mathematics. The first is representation. How are the functions and bit strings to be stored?

The simplest representation of the data used in the DES algorithm is to store one bit in each storage location (byte or word); for example, since the message to be encrypted or decrypted is 64 bits long, it would occupy 64 storage units, each being 0 or 1. This representation requires no bit operations to permute the data. Unfortunately it is quite inefficient in terms of space and time; in space, because (usually) at least 8 bits can be stored in a single storage unit, and in time, because it forces the algorithm to operate on bits one by one. As an example of the effects of this restriction, the substitution function σ_S, and hence the function F_2, uses 8 sets of 6 adjacent bits from the representation of R_i. There are 16 rounds in which this is done, and obtaining those sets of bits from the representation requires 48 memory accesses, 5 left shifts and five logical ors, for a total of $(48+5+5)\times8\times16 = 7424$ operations. Were each set of 6 bits stored in a single storage unit, obtaining those sets would require one memory access per set, for a total of $1\times8\times16 = 512$ operations. Were 24 bits stored per storage unit, obtaining those sets would require one memory access and one bit field extraction per set, for a total of $(1+1)\times8\times16 = 1024$ operations.

For reasons that will shortly become clear, we chose to store bits as multiples of 24. So, in a 32 bit per word machine, we store the 48 bit quantities L_i and R_i in two words with 24 bits each; in a 64 bit per word machine, we store them in one word with 48 bits each.

Using the mathematical analysis in the previous section, we rearranged the key schedule computation and message transformation so that there are four sets of combinations of permutations, expansions, and substitutions. We used these four functions rather than η_E, π_P, π_{IP}, and σ_S given in the tables in Appendix I.

Appendix I represents each of the permutations as a vector of numbers; the number in the ith position is the number of the bit of the input that will occupy that position in the output. Hence, it seems natural to represent the functions F_i, $i = 1,2,3$ and G in the same manner. However, since F_2 begins

with a substitution, we shall for the moment only combine π_P and η_E, call this F_{2INT}, and include the σ_S later. The functions in Appendix II may be used to compute F_1, F_{2INT}, F_3, and G; the corresponding bit tables are in Appendix III. Remember, the precomputation of G will produce 16 permutations, each of which is applied to the key to get the key schedule.

Once this is done, another optimization becomes obvious. Since a permutation is simply a function with inputs from a known domain, those functions can be precomputed and stored in an array. Then computing the function during the running of the algorithm requires as many array indexing operations as there are arguments to the function. For example, F_1 takes as input the 64 bit message and returns as output 96 bits, so ideally F_1 should be precomputed and stored as an array of 2^{64} elements of 96 bits.

This would probably use too much space and time on most computers. So, we broke up the 64 bit message into 8 sets of 8 bits, and considered F_1 to be a function of two arguments, the first being 8 bits from the message, and the second being the position of the first argument within the 64 bit message. We thus stored F_1 as an array of 8 arrays, one per position, of 2^8 words of 96 bits; given a machine with B bits per word, this uses

$$8\times 2^8 \times \frac{96}{\lfloor B/24 \rfloor \times 24} \text{ words} = \begin{cases} 2048 \text{ words} & \text{if } B = 32 \\ 1024 \text{ words} & \text{if } B = 64 \end{cases}$$

This requires that the eight 96 bit vectors, each corresponding to a byte in the first through eighth position, must be combined:

```
output := 0;
for i := 1 to 8 do
        output := output or fsub1[i][input[i]];
```

To precompute F_3, note that the input is a 96 bit vector and the output is 64 bits. In most implementations, this will be represented as eight 8 bit characters. So, splitting F_3's input into 16 sets of 6 bits, the total storage required is

$$16\times 2^6 \times \frac{64}{8} \text{ characters} = 8192 \text{ characters}$$

Precomputing F_2 is a bit more complicated because the input of 48 bits is run through a substitution. This function, σ_S, is really eight substitution functions $\sigma_S^{(1)}, \ldots, \sigma_S^{(8)}$, $\sigma_S^{(1)}$ operating on the first 6 bits of the input, $\sigma_S^{(2)}$ operating on the second 6 bits of the input, and so on. These substitution functions are defined independently of each other, so to precompute F_2, vary the input to one of the $\sigma_S^{(i)}$ and keep the input for all the others as 0. Apply that $\sigma_S^{(i)}$ to all 2^6 inputs, and then apply F_{2INT} to the result. Hence, F_2 will have to be stored as an array of 8 arrays of 2^6 words of 48 bits; this requires

$$8\times 2^6 \times \frac{48}{\lfloor B/24 \rfloor \times 24} \text{ words} = \begin{cases} 1024 \text{ words} & \text{if } B = 32 \\ 512 \text{ words} & \text{if } B = 64 \end{cases}$$

Note this is really a rather small storage requirement.

Alas, the same is not true for the precomputation of G. The key is divided into eight 8 bit groups, and one permutation per element of the key schedule is needed; in short, G is really a set of 16 permutations. The output of each of these 16 permutations is a word with 48 bits. So the storage required is

$$16\times 8\times 2^8\times \frac{48}{\lfloor B/24\rfloor\times 24}\text{ words} = \begin{cases}65{,}536\text{ words} & \text{if } B=32\\ 32{,}768\text{ words} & \text{if } B=64\end{cases}$$

For some machines, this is so large that the increase in system time due to accessing the elements offsets the gain in user processing time. In such a case it is better to precompute $\pi_{PC\text{-}1}$, which requires

$$8\times 2^8\times \frac{56}{\lfloor B/24\rfloor\times 24}\text{ words} = \begin{cases}4096\text{ words} & \text{if } B=32\\ 2048\text{ words} & \text{if } B=64\end{cases}$$

and $\pi_{PC\text{-}2}$, which requires

$$8\times 2^7\times \frac{48}{\lfloor B/24\rfloor\times 24}\text{ words} = \begin{cases}2048\text{ words} & \text{if } B=32\\ 1024\text{ words} & \text{if } B=64\end{cases}$$

and do the left shifts as one operation if possible. To do this, recall that if the 56 bit $K^{(i)}$ is stored in one word, the following operations split $K^{(i)}$ into two 28 bit halves, shifts each half left one bit, and recombines them to give $K^{(i+1)}$:

$$K^{(i+1)} = (((K^{(i)} \& \neg 0\text{x}80000008000000) \ll 1) \mid$$
$$((K^{(i)} \gg 27) \& 0\text{x}10000001)) \& 0\text{xfffffffffffffff}$$

(where $\ll$ and $\gg$ are left and right shifts, respectively.) Shifting two bits to the left, and rearranging this to work on a 32 bit word machine, is left as an exercise to the reader.

Incidentally, the way the functions were precomputed determined the choice of the data representation. Because we had to access bits in groups of 6 while iterating, and in groups of 8 when applying F_3, the data representation had to be a multiple of lcm(6,8) = 24 bits per word.

Now let us look in detail at the formulae we have derived. Using the conventional formulation (8), 16 iterations are needed to compute L_{16} and R_{16}. Equation (14) requires only 8 iterations. This cuts the time required approximately in half; so, we should use the formulation in (14) to compute L_{16} and R_{16}.

As a second point, we described how to precompute the function F_2 as an array. Recall that the $L_{i+2}\oplus k_{i+1}$ and $R_i\oplus k_i$ are split into 8 groups of 6 bits, each group being the input to one $\sigma_S^{(i)}$. By combining the groups of bits and $\sigma_S^{(i)}$, substantial speed can be gained. The penalty is that more space is needed; if, for example, 12 bits is used rather than 6 bits, there are 4 groups of 12 bits, and the number of words required is

$$4\times 2^{12}\times \frac{48}{\lfloor B/24\rfloor\times 24}\text{ words} = \begin{cases}32768\text{ words} & \text{if } B=32\\ 16384\text{ words} & \text{if } B=64\end{cases}$$

So, when we precomputed F_2, we used as wide a data path as was feasible.

We can eliminate the cost of using some temporary storage if we exclusive-or F_2 into L_{n+2} directly. Suppose a and b were two bit numbers. Then the first bit of $a \oplus b$ depends only on the first bit of a and the first bit of b, and the second bit of $a \oplus b$ depends only on the second bit of a and the second bit of b. Hence, $\oplus$ operates on bits independently of one another. Combined with the associativity of $\oplus$, this means that F_2 can be directly $\oplus$ed into L_i or R_i without a temporary variable; F_2 need not be saved in an intermediate variable, and then $\oplus$ed into L_i or R_i.

Now that we have dealt with the message-handling part of the formulation, Rather than computing a key schedule and testing at each iteration whether to use the nth element or the $(16-n)$th element, we shall compute the key schedule for one direction, and reverse the elements at the end of the computation if going in the other.

The key schedule $\{k_i \mid i = 1, \ldots, 16\}$ is used to encrypt and decrypt. To encrypt, the k_is are used in the order $k_1, \ldots, k_{16}$ and to decrypt, they are used in the order $k_{16}, \ldots, k_1$. The key schedule should be in the proper order before the message is encrypted or decrypted to avoid 16 comparisons and (possibly) 16 subtractions when transforming the message. Let us compare the cost in time T_{GEN} of storing the key schedule in the proper order as it is generated and the cost in time T_{END} of storing the key schedule in encrypting order as it is generated and then reversing it at the end.

Let the probability that an encryption is to be done be P_{ENC}. Let the cost of a comparison operation be "c", the cost of an indexing operation be "i", and the cost of an assignment be "a" Then assigning the keys in the proper order costs

$$\begin{aligned} T_{GEN} &= (16\text{c} + 16\text{i} + 16\text{a})P_E + (16\text{c} + 16\text{i} + 16\text{a})(1 - P_E) \\ &= (16\text{c}+16\text{i}+16\text{a}) \end{aligned}$$

Assigning the keys in encryption order and exchanging them after all have been computed requires

$$\begin{aligned} T_{END} &= (1\text{c} + 16\text{a})P_E + (1\text{c} + 16\text{i} + 32\text{a})(1 - P_E) \\ &= 1\text{c} + 16\text{i} + 32\text{a} - (16\text{i} + 16\text{a})(1 - p_E) \end{aligned}$$

(No indexing for p_E is required since the indices are constants, and the addresses are therefore computed at compile time.) If $p_E = 0.5$, then

$$T_{END} = 0.5\text{c}+8\text{i}+24\text{a}$$

which – assuming c = a = i – means that $T_{END} < T_{GEN}$, so if the probability of encryption and decryption are the same, it is better to reorder the key schedule at the end.

We will later have occasion to run the DES algorithm when the probability of encryption is 1. In this case,

$$T_{GEN} = (16\text{c}+16\text{i}+16\text{a}) > (1\text{c}+16\text{a}) = T_{END}$$

So it is very definitely desirable to reorder after the key schedule has been computed.

Finally, we can take advantage of several features of the compiler and underlying machine architecture. For example, we can unroll iterative loops whenever possible, replacing a conditional test, an increment or decrement (or addition or subtraction), and several memory accesses, since replacing the loop variable with constants in the body of the loop allows the use of "quick" or "immediate" mode, where the constant is stored in the instruction. Such an access is considerably faster than accessing a memory location. When it is too messy to eliminate an iterative loop, rather than incrementing the counter, we will count down to 0. Most computers have an instruction which decrements a counter and compares the result to 0 in one operation. Taking advantage of this instruction decreases the number of memory accesses substantially, as well as the number of instructions to be executed.. Using pointers or constants instead of indexing into an array also speeds up the computation; even though an array access is semantically equivalent to the use of a pointer, using pointers to access n elements of an array sequentially results in one index computation to initialize the pointer and $n-1$ additions, rather than n index computations. Finally, the specific code generated by the compiler for the target machine offers several possibilities. Some machines have slow bit field extraction operations; on these machines, using shifting and masking to extract bit fields may be faster. Other compilers use autoincrement address mode if available. If so, it should be used to step through a sequence of arrays by incrementing the pointer containing the base address of each element. If not, use constants for the bases of the arrays; this will save an addition. (In fact, the arrays storing the F_{2INT}'s are "fsub2_1[64], . . . , fsub2_8[64]" rather than in one array "fsub2[8][64], . . . , fsub2_8[64]". This way, if autoincrement mode is used, the bases "fsub2_1, . . . , fsub2_8" are saved in another array and a pointer used to access the bases. If autoincrement mode is not used, the bases themselves replace the pointers.)

4. Analysis of the UNIX *crypt*(3) Algorithm

The UNIX password encryption algorithm is based on the DES algorithm described in 2. A password, chosen from strings of one to eight characters with characters from an alphabet consisting of the upper and lower case letters, the digits, "/", and ".", is used as a key to encrypt the message 0 (that is, the message of eight ASCII NULs.) However, the DES algorithm is repeated 25 times and the expansion η_E is altered.

Associated with each password is a two-character "salt", with characters chosen from the same alphabet as the password characters. This allows a possible $64\times64 = 4096 = 2^{12}$ salts. The two character salt is used to compute a number $0 \leq s < 4096$; then, if the ith bit of that number is set, elements i and $i+24$ of the table for η_E are exchanged. Let this new permutation be $\eta_{E'}$.

Let $w = w_1 \cdots w_{32}$ be a 32 bit vector; applying η_E to it gives a 48 bit vector $v = v_1 \cdots v_{48}$, and applying $\eta_{E'}$ to it gives another 48 bit vector $v' = v'_1 \cdots v'_{48}$. Let $s_1 \cdots s_{12}$ be the binary representation of s. From the

way $\eta_{E'}$ is derived from η_E, it is clear that for $k = 1, \ldots, 12$, if $s_k = 0$, $v_k = v'_k$ and $v_{k+24} = v'_{k+24}$, and if $s_k = 1$, $v_k = v'_{k+24}$ and $v_{k+24} = v'_k$. Our goal is to find a way of transforming v into v' given s and nothing else.

Split v into two parts, $v_{(1 \cdots 24)}$ and $v_{(25 \cdots 48)}$, where $v_{(1 \cdots 24)} = v_1 \cdots v_{24}$ and $v_{(25 \cdots 48)} = v_{25} \cdots v_{48}$. Similarly, split v' into two parts, $v'_{(1 \cdots 24)}$ and $v'_{(25 \cdots 48)}$, where $v'_{(1 \cdots 24)} = v'_1 \cdots v'_{24}$ and $v'_{(25 \cdots 48)} = v'_{25} \cdots v'_{48}$. Define a 24 bit vector $m = m_1 \cdots m_{24}$ where

$$m_k = \begin{cases} s_k = 0 & \text{if } i \leq 12 \\ s_k = 1 & \text{if } i > 12 \end{cases}$$

Now, notice that $v_{(1 \cdots 24)} \& m = (v'_1 \cdots v'_{24}) \& (s_1 \cdots s_{12} 0 \cdots 0)$ has those bits of $v_{(1 \cdots 24)}$ which would be exchanged with the corresponding elements of $v_{(25 \cdots 48)}$, and all other bits are 0. Similarly, $v_{(1 \cdots 24)} \& \neg m$ has those bits of $v_{(1 \cdots 24)}$ which are not to be exchanged, and all other bits are 0. So, to compute v' given v and s, first construct m as indicated and then:

$$v'_{(1 \cdots 24)} = (v_{(1 \cdots 24)} \& \neg m) | (v_{(25 \cdots 48)} \& m) \tag{17a}$$

$$v'_{(25 \cdots 48)} = (v_{(25 \cdots 48)} \& \neg m) | (v_{(1 \cdots 12)} \& m) \tag{17b}$$

Robert Baldwin [3] has derived an equivalent expression for v' in terms of v by noting that if bits v_i and v_{i+24} differ, exchanging them is the same as $\oplus$ing them with 1. So, he suggests computing a mask SS to be $\oplus$ed with $v_{(1 \cdots 24)}$ and $v_{(25 \cdots 48)}$ to achieve the effect of the salting:

$$SS = (v_{(1 \cdots 24)} \oplus v_{(25 \cdots 48)}) \& m \tag{18a}$$

Then,

$$v'_{(1 \cdots 24)} = v_{(1 \cdots 24)} \oplus SS \tag{18b}$$

$$v'_{(25 \cdots 48)} = v_{(25 \cdots 48)} \oplus SS \tag{18c}$$

A more formal proof showing this is equivalent to (17) uses the following two lemmas:

LEMMA: $a \oplus (a \& c) \equiv (a \& \neg c)$

PROOF: By definition of $\oplus$,

$$a \oplus (a \& c) \equiv (a \& \neg(a \& c)) | (\neg a \& (a \& c))$$

Now, as $\&$ is associative, $\neg a \& (a \& c) \equiv (\neg a \& a) \& c$, which is always false. Moreover, as $\neg(a \& c) \equiv (\neg a) | (\neg c)$, by distributivity of $\&$ over $|$ we have $a \& \neg(a \& c) \equiv (a \& \neg a) | (a \& \neg c) \equiv a \& \neg c$ as claimed. QED.

LEMMA: $a \oplus ((a \oplus b) \& c) \equiv ((a \& \neg c) | (b \& c))$

PROOF: As $\&$ distributes over $\oplus$,

$$a \oplus ((a \oplus b) \& c) \equiv a \oplus ((a \& c) \oplus (b \& c))$$

Since $\oplus$ is associative, this becomes

$$\equiv (a \oplus (a \& c)) \oplus (b \& c)$$

But by the previous lemma,

$$\equiv (a\&\neg c)\oplus(b\&c)$$

as claimed. QED.

We can now show

THEOREM: (17) and (18) are equivalent.

PROOF: Substitute $a = v_{(1\ldots 24)}$, $b = v_{(25\ldots 48)}$, and $c = m$ in the preceding lemma. QED.

5. Application of the Analysis

Given this analysis, there are a few simplifications that can be used to speed up the computation. First, recall that the UNIX password encryption scheme calls the DES algorithm 25 times sequentially. Notice that $F_1 = \pi_X F_3^{-1}$, so the left and right halves of the result of the iteration must be swapped, but then can be immediately put back into the iteration. This saves 48 permutations. Following this train of thought, we do not need to precompute F_1, since the value returned by F_1 does not depend on the key schedule, and hence not at all on the key (password); just on the message being encrypted. This message is 0. Therefore, $F_1 0 = 0$, so start the first iteration with the 96 bit vector with all bits clear. Finally, the exchange of halves (that is, the permutation π_X) may be completely eliminated by unrolling a loop. This requires that the 25 iterations be done in 12 sets of 2, and one more, and that in the second set the L_i and R_i be interchanged.

Two more changes are possible, but they only help under certain conditions. First, suppose you will be encrypting p passwords all with the same salt. Using (18), each of the 8 loops will take 4 extra instructions to process the salt; these loops are repeated 25 times, so for each password to be encrypted, handling the salt requires 800 extra instructions. Precomputing F_1, F_2, and F_3 using $\eta_{E'}$ would eliminate this overhead.

Now suppose these routines are being used to compare p passwords to w suspected cleartext passwords. If F_3 is applied in the encryption routine, it must be applied w times; if F_3^{-1} is applied to the list of encrypted passwords, it need only be applied p times (actually, there is some extra overhead that is negligible.) Presumably, both F_3 and F_3^{-1} are precomputed and so take equally long to apply. Thus, if $w < p$, apply F_3^{-1} to the list of encrypted passwords, and omit the application of F_3 at the end of the encryption.

6. An Experiment

In order to substantiate our claim that the suggested optimizations provided substantial increase in speed, we implemented four versions of the faster DES algorithm, and four versions of the faster UNIX password algorithm:

- one without G and with a 6 bit data path to F_2
- one with G and with a 6 bit data path to F_2

- one without G and with a 12 bit data path to F_2
- one with G and with a 12 bit data path to F_2

The routines were timed on several different architectures, and the results collected below. As a control, the standard UNIX implementation, which stores one bit per storage unit (byte) and uses a straightforward translation of the algorithm into C, was also timed.

Table I. The Computers

computer	timer intervals (per sec)	bits (per word)	manufacturer
Amdahl 5880	60	32	Amdahl Corp.
Convex 1 (32)	100	32	Convex Computer Corp.
Convex 1 (64)	100	64	Convex Computer Corp.
Cray 2	243902439	64	Cray Research Inc.
IBM PC/RT	100	32	IBM Corp.
IRIS 2500T	60	32	Silicon Graphics Inc.
Sequent 21000	100	32	Sequent Computer Systems
Sun 3/50	60	32	Sun Microsystems, Inc.
VAX 11/780	100	32	Digital Equipment Corp.
VAX 11/785	100	32	Digital Equipment Corp.

Table I summarizes the machines and their relevant characteristics. (More detailed information is in appendix IV.) In particular, note the clock rates; with all except the Cray 2, the routines were expected to execute much faster than the clock tick. The usual way to make timings is to start the clock, run the routine, stop the clock, and determine the elapsed time. However, since this would be on the order of one tick, the results gleaned this way would be highly suspect. So, what was done instead was to start the clock and execute a loop for approximately 10 seconds (virtual time.) A counter in this loop was incremented every time the routine completed execution. When the time was up, the next return ended the loop, and the clock was stopped and the elapsed time computed. Another loop, just like the first but without the call to the routine, was executed for the same number of iterations; the time to complete this loop was then subtracted from the elapsed time. This way, the total time was subject to an error of at most two clock ticks. This procedure was repeated ten times, and the times and ratios computed by averaging over the ten results. (The tables in Appendix V give the actual timings.)

We should also note that the timings are not meaningful when comparing among many machines, because the loads were very different. On the assumption that none of the loads changed dramatically during the testing (an assumption that in fact held), the ratios express the amount of speedup quite accurately; this is why we use them in this section, rather than the times.

Table II. Ratio of Mean Execution Times: DES Encryption Routines UNIX Interface					
computer	6 bit path		12 bit path		standard
	without G	with G	without G	with G	function
Amdahl 5880	7.18	7.19	7.97	8.00	1.00
Convex 1 (32)	4.29	4.26	5.09	5.04	1.00
Convex 1 (64)	4.85	4.84	5.62	5.60	1.00
Cray 2	9.50	9.51	10.28	10.30	1.00
IBM PC/RT	7.88	7.89	8.67	8.74	1.00
IRIS 2500T	8.37	8.32	10.98	10.95	1.00
Sequent 21000	8.89	8.67	9.74	9.77	1.00
Sun 3/50	7.33	7.21	9.45	9.29	1.00
VAX 11/780	6.12	5.88	6.15	6.00	1.00
VAX 11/785	6.78	6.82	7.56	7.77	1.00

Table II gives the ratio of the mean execution time of the DES encryption routines to the UNIX standard DES encryption routines. The interface is the same; the standard UNIX routines *setkey* and *encrypt* (see *crypt* (3) in [2] or *crypt* (3C) in [1]) can be replaced by these simply by naming a library to the linking loader. In all cases, there is a substantial speedup, from a factor of 4.3 for the Convex running with 32 bits per word and using the 6 bit path version not using G, to 11.0 on the IRIS 2500T running the 12 bit path version.

Table III. Ratio of Mean Execution Times: DES Encryption Routines					
computer	6 bit path		12 bit path		standard
	without G	with G	without G	with G	function
Amdahl 5880	11.99	12.04	14.46	14.80	1.00
Convex 1 (32)	5.85	5.87	7.52	7.31	1.00
Convex 1 (64)	6.99	7.00	8.64	8.59	1.00
Cray 2	18.51	18.55	21.60	21.66	1.00
IBM PC/RT	13.14	13.24	15.39	15.69	1.00
IRIS 2500T	12.99	12.96	20.69	20.66	1.00
Sequent 21000	17.14	17.22	20.97	21.27	1.00
Sun 3/50	11.29	11.07	17.36	16.71	1.00
VAX 11/780	12.24	11.44	11.41	11.69	1.00
VAX 11/785	15.66	14.55	17.65	20.07	1.00

Based on these figures, we suspect that a good portion of the time involved is bit packing; the interface packs 64 bits, each initially in its own byte, into 8 bytes. So, if we eliminate this packing, we should see the new routines speed up. In fact this is what happened. As evidence of this claim, consider table III. Except for the last column, all counts are from calling the routines directly, without the UNIX interface; so, there is no bit packing done. In all cases, the new routines are substantially faster than the standard UNIX functions, with the improvement ranging from a factor of 5.9 for the Convex using 32 bits per word and running the 6 bit path version without G to 21.7 on

the Cray 2 running the 12 bit path version without G.

Table IV. Ratio of Mean Execution Times: Password Encryption Routines
UNIX Interface

computer	6 bit path		12 bit path		standard
	without G	with G	without G	with G	function
Amdahl 5880	17.25	16.82	19.52	19.50	1.00
Convex 1 (32)	10.56	10.55	16.29	16.11	1.00
Convex 1 (64)	13.15	13.01	19.60	19.67	1.00
Cray 2	42.09	41.99	59.35	59.92	1.00
IBM PC/RT	19.25	18.93	24.12	23.96	1.00
IRIS 2500T	16.66	16.87	30.55	31.47	1.00
Sequent 21000	23.62	23.39	29.96	29.41	1.00
Sun 3/50	14.70	14.88	26.00	26.49	1.00
VAX 11/780	17.01	15.67	16.57	15.90	1.00
VAX 11/785	19.71	17.89	24.10	22.36	1.00

The increase in speed of the UNIX password encryption routine is far more dramatic; Table IV documents them. Although the password encryption algorithm is essentially 25 iterations of the DES encryption routine, all F_1 and all but one of the F_3 are omitted; hence, we expect the ratios to be more dramatic than those in Table III. Table IV shows our expectations are fulfilled. The factors of improvement range from a low of 10.6 for the Convex using 32 bits per word and running the 6 bit path version (with or without G) to a high of 60.0 for the IRIS 2500T running the 12 bit path version using G.

Table V. Ratio of Mean Execution Times: Password Encryption Routines

computer	6 bit path		12 bit path		standard
	without G	with G	without G	with G	function
Amdahl 5880	18.24	17.89	21.56	21.58	1.00
Convex 1 (32)	11.13	10.99	17.55	17.38	1.00
Convex 1 (64)	14.11	14.03	22.08	21.91	1.00
Cray 2	47.40	47.06	71.55	71.26	1.00
IBM PC/RT	22.77	22.77	28.00	27.66	1.00
IRIS 2500T	17.33	17.58	32.82	33.78	1.00
Sequent 21000	25.04	24.12	32.63	31.97	1.00
SUN 3/50	15.26	15.38	27.71	28.12	1.00
VAX 11/780	18.87	15.41	17.74	17.06	1.00
VAX 11/785	20.42	19.97	27.61	25.11	1.00

Finally consider the speedup mentioned in the previous section, namely omitting the transformation F_3. How much this affects the speed depends quite a bit on the architecture of the machine; if it can handle bytes well, there should be little effect, but if it is optimized to work with words the omission may improve things substantially. Table V bears this out; the incerase for byte-oriented machines is typically 1 or 2 more iterations than when F_3 is

included, but for machines such as the Cray 2, the speedup is far more substantial (on the order of 11 iterations when a 12 bit data path is used).

These timings demonstrate that there is a substantial performance gain by using the suggested techniques to speed up the DES routines and the UNIX algorithm. In fact, the speedup is so substantial that trying each word in a dictionary to see if it matches a user's encrypted password becomes feasible. (One of the goals of salting was to avoid this attack [10].) Say an on-line dictionary contains 25,000 words. Using the standard password encryption function on a VAX 11/780, it would take 17,730.5 seconds (about 5 hours) to check a particular encrypted password; to check a collection of 100 passwords, it would take 1,773,049.6 seconds (about 20.5 days) if each used a different salt. But using this method, it would take 1002 seconds (about 17 minutes) to check a particular encrypted password, and 100,200.0 seconds (about 28 hours) to check 100 encrypted passwords each with a different salt. This shows the danger of relying on a routine's computing something slowly to provide protection; it also demonstrates the old adage that making a cipher more complex does not make it more secure.

7. Conclusion

The results of the previous section demonstrate that the UNIX password routines can be made significantly faster. In fact, even if the salt were 24 bits rather than 12, the algorithm would still be amenable to the same type of attack. The problem is that the method used to perturb η_E can be implemented as a bit operation rather than forcing it to be a permutation of tables.

There is one area not touched upon in this discussion, and that is the role of assembly language coding. It has been the author's experience that coding in assembly language can cut speed by up to 40%. We did not do this because there are too many different machine languages and architectures involved. To implement that part properly would require a special postprocessing pass different for each computer. That would conflict with the goal of a very portable package. (Currently the user must set two compile-time flags properly, and may set two others that improve performance on some machines should he or she so desire.)

We intend to use this version to test passwords, and see if the characteristics noted in [10] hold for our site. If so, we will try to change user habits. This version of the encryption routine is a great step forward in realizing our goal of a system where no passwords can be found by using a dictionary, or a list of words, to compromise passwords.

Acknowledgements: Thanks to Bob Van Cleef, who first suggested the project that led to this work, and to David Balenson, who pointed me to many of the papers that led me to think of this mathematical representation of the DES. Thanks also to Robert Baldwin, who made the observation (referred to in section 4) that the salt could be computed in four instructions per iteration; and I am grateful for his very clear exposition. Finally, I am grateful to Peter Salus for the valuable comments and suggestions on earlier drafts of this paper; his

comments materially improved its clarity.

References

1. -, "UNIX System V – Release 2.0 Programmer Reference Manual", AT&T Technologies 307-113, Issue 2, 1984.
2. -, *UNIX Programmer's Reference Manual, 4.3 Berkeley Software Distribution, Virtual VAX-11 Version*, Computer Systems Research Group, Computer Science Division, EECS, University of California, Berkeley, Berkeley, CA 94720, November 1986. as reprinted by the USENIX Association.
3. Baldwin, R., Fast DES and Password Transforms in Software, private communication, 1987.
4. Davies, D., "A New Look at the DES Complementation Property and Weak Keys", *unpublished*, (April 1987).
5. Davio, M., Desmedt, Y., Fosseprez, M., Govaerts, R., Hulsbosch, J., Neutjens, P., Piret, P., Quisquater, J.-J., Vandewalle, J. and Wouters, P., "Analytical characteristics of the DES", in *Advances in Cryptology: Proceedings of Crypto 83*, D. Chaum (ed.), Plenum Press, New York, NY, August 1983.
6. Davio, M., Desmedt, Y., Goubert, J., Hoornaert, F. and Quisquater, J.-J., "Efficient hardware and software implementations for the DES", in *Advances in Cryptology: Proceedings of Crypto 84*, vol. 196, G. Blakley and D. Chaum (ed.), Springer-Verlag, New York, NY, August 1984.
7. Hoornaert, F., Goubert, J. and Desmedt, Y., "Efficient hardware implementation of the DES", in *Advances in Cryptology: Proceedings of Crypto 84*, vol. 196, G. Blakley and D. Chaum (ed.), Springer-Verlag, New York, NY, August 1984.
8. Konheim, A., *Cryptography: A Primer*, John Wiley & Sons, Inc., New York, NY, 1981.
9. Meyer, C. and Matyas, S., *Cryptography: A New Dimension in Data Security*, John Wiley & Sons, Inc., New York, NY, 1982.
10. Morris, R. and Thompson, K., "Password Security: A Case History", *Communications of the ACM, 22*, 11 (November 1979) 594-597.
11. "Data Encryption Standard", Federal Information Processing Standards Publication 46, National Bureau of Standards, January 1977.

Appendix I. The DES Tables

The permutations, expansions, and substitutions are controlled by the following tables. For permutations and expansions, the number in the ith position is the number of the input bit to be output at that position; for the substitution, the first and last bits indicate the row number, the middle four bits the column number, and the position of the 6 bits within the 48 bit data the number of the table used. Read left to right, starting at the top row; so with the table for *IP*, the first twenty bits of the output will be bits 58, 50, 42,

34, 26, 18, 10, 2, 60, 52, 44, 36, 28, 20, 12, 4, 62, 54, 46, and 38. of the input.

The following are the tables used for encrypting and decrypting the message.

Table for π_{IP}

58	50	42	34	26	18	10	2	60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6	64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1	59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5	63	55	47	39	31	23	15	7

Table for π_{IP}^{-1}

40	8	48	16	56	24	64	32	39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30	37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28	35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26	33	1	41	9	49	17	57	25

Table for η_E

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

Table for π_P

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

Table for $\sigma_S^{(1)}$

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

Table for $\sigma_S^{(2)}$

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

Table for $\sigma_S^{(3)}$

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

Table for $\sigma_S^{(4)}$

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

Table for $\sigma_S^{(5)}$

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

Table for $\sigma_S^{(6)}$

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

Table for $\sigma_S^{(7)}$

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

Table for $\sigma_S^{(8)}$

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

The following are the tables for the functions to generate the key schedule.

Table for $\pi_{PC\text{-}1}$

57	49	41	33	25	17	9	1	58	50	42	34	26	18
10	2	59	51	43	35	27	19	11	3	60	52	44	36
63	55	47	39	31	23	15	7	62	54	46	38	30	22
14	6	61	53	45	37	29	21	13	5	28	20	12	4

Table for $\pi_{PC\text{-}2}$

14	17	11	24	1	5	3	28	15	6	21	10
23	19	12	4	26	8	16	7	27	20	13	2
41	52	31	37	47	55	30	40	51	45	33	48
44	49	39	56	34	53	46	42	50	36	29	32

Schedule of left shifts for key schedule computation
1 1 2 2 2 2 2 2 1 2 2 2 2 2 2 1

Appendix II. Precomputation of F_1, F_{2INT}, F_3, and G

The following program precomputes F_1, F_{2INT}, and F_3 in the arrays *fsub1*, *fsub2INT*, and *fsub3*, respectively.

```
var   pisubIP: array[1..64] of integer; (* π_IP as an array *)
      etasubE: array[1..48] of integer; (* η_E as an array *)
      etasubEinv: array[1..32] of integer; (* η_E^-1 as an array *)
      sigmasubs: array[1..8][1..64] of integer; (* σ_S as an array *)
      pisubP: array[1..32] of integer; (* π_P as an array *)
      pisubIPinv: array[1..64] of integer; (* π_IP^-1 as an array *)

      fsub1: array[1..96] of integer; (* for F_1 *)
      fsub2INT: array[1..32] of integer; (* for F_2INT as an array *)
      fsub3: array[1..64] of integer; (* for F_3 *)
      tmp: array[1..64] of integer; (* used to store η_EE^-1 *)

(* compute the left half of F_1 as an array *)
for i := 1 to 48 do
      fsub1[i] = pisubIP[etasubE[i]];
(* compute the right half of F_1 as an array *)
for i := 49 to 96 do
      fsub1[i] = pisubIP[etasubE[i−47]+32];
(* combine the permutations in F_2INT *)
for i := 1 to 48 do
      fsub2INT[i] = pisubP[etasubE[i]];
(* compute π_X η_EE^-1 and put it in tmp *)
for i := 1 to 32 do
      tmp[i] = etasubEinv[i] + 48;
for i := 33 to 64 do
      tmp[i] = etasubEinv[i−32];
(* compute f_3 as an array *)
for i := 1 to 64 do
      fsub3[i] = tmp[pisubIPinv[i]];
```

The following shows how to compute G; its values are stored in the array g.

```
var   pisubPC1: array[1..56] of integer; (* π_PC-1 as an array *)
      pisubLSH: array[1..16] of integer; (* π_LSH as an array *)
      pisubPC2: array[1..48] of integer; (* π_PC-2 as an array *)
      g: array[1..16][1..48] of integer; (* for G *)
      tmp: integer;     (* used to exchange elements *)

(* compute g as an array *)
```

```
for i := 1 to 16 do begin
      (* compute the elements of π_LSH π_PC-1 as an array *)
      for j := 1 to pisubLSH[i] do begin
            tmp := pc1[1];
            for k := 2 to 28 do
                  pc1[j−1] := pc1[j];
            pc1[28] := tmp;
            tmp := pc1[29];
            for k := 30 to 56 do
                  pc1[j−1] := pc1[j];
            pc1[56] := tmp;
      end;
      (* now apply π_PC-2 *)
      for j := 1 to 48 do
            g[i][j] = pisubPC1[pisubPC2[j]];
end;
```

Appendix III. The Functions F_1, F_{2INT}, F_3, and G

The following are tables of bit permutations in the same format as those tables in appendix I. Note that these are *not* used directly in the fast implementation, but are used to compute bit vectors which are.

The meaning and nature of each function is discusses in sections 2 and 3.

F_1

8	58	50	42	34	26	34	26	18	10	2	60
2	60	52	44	36	28	36	28	20	12	4	62
4	62	54	46	38	30	38	30	22	14	6	64
6	64	56	48	40	32	40	32	24	16	8	58
7	57	49	41	33	25	33	25	17	9	1	59
1	59	51	43	35	27	35	27	19	11	3	61
3	61	53	45	37	29	37	29	21	13	5	63
5	63	55	47	39	31	39	31	23	15	7	57

F_{2INT}

25	16	7	20	21	29	21	29	12	28	17	1
17	1	15	23	26	5	26	5	18	31	10	2
10	2	8	24	14	32	14	32	27	3	9	19
9	19	13	30	6	22	6	22	11	4	25	16

F_3

11	59	23	71	35	83	47	95
10	58	22	70	34	82	46	94
9	57	21	69	33	81	45	93
8	56	20	68	32	80	44	92
5	53	17	65	29	77	41	89
4	52	16	64	28	76	40	88
3	51	15	63	27	75	39	87
2	50	14	62	26	74	38	86

G_0

10	51	34	60	49	17	33	57	2	9	19	42	3	35	26	25
44	58	59	1	36	27	18	41	22	28	39	54	37	4	47	30
5	53	23	29	61	21	38	63	15	20	45	14	13	62	55	31

G_1

2	43	26	52	41	9	25	49	59	1	11	34	60	27	18	17
36	50	51	58	57	19	10	33	14	20	31	46	29	63	39	22
28	45	15	21	53	13	30	55	7	12	37	6	5	54	47	23

G_2

51	27	10	36	25	58	9	33	43	50	60	18	44	11	2	1
49	34	35	42	41	3	59	17	61	4	15	30	13	47	23	6
12	29	62	5	37	28	14	39	54	63	21	53	20	38	31	7

G_3

35	11	59	49	9	42	58	17	27	34	44	2	57	60	51	50
33	18	19	26	25	52	43	1	45	55	62	14	28	31	7	53
63	13	46	20	21	12	61	23	38	47	5	37	4	22	15	54

G_4

19	60	43	33	58	26	42	1	11	18	57	51	41	44	35	34
17	2	3	10	9	36	27	50	29	39	46	61	12	15	54	37
47	28	30	4	5	63	45	7	22	31	20	21	55	6	62	38

G_5

3	44	27	17	42	10	26	50	60	2	41	35	25	57	19	18
1	51	52	59	58	49	11	34	13	23	30	45	63	62	38	21
31	12	14	55	20	47	29	54	6	15	4	5	39	53	46	22

G_6

52	57	11	1	26	59	10	34	44	51	25	19	9	41	3	2
50	35	36	43	42	33	60	18	28	7	14	29	47	46	22	5
15	63	61	39	4	31	13	38	53	62	55	20	23	37	30	6

G_7

36 41 60 50 10 43 59 18 57 35 9 3 58 25 52 51
34 19 49 27 26 17 44 2 12 54 61 13 31 30 6 20
62 47 45 23 55 15 28 22 37 46 39 4 7 21 14 53

G_8

57 33 52 42 2 35 51 10 49 27 1 60 50 17 44 43
26 11 41 19 18 9 36 59 4 46 53 5 23 22 61 12
54 39 37 15 47 7 20 14 29 38 31 63 62 13 6 45

G_9

41 17 36 26 51 19 35 59 33 11 50 44 34 1 57 27
10 60 25 3 2 58 49 43 55 30 37 20 7 6 45 63
38 23 21 62 31 54 4 61 13 22 15 47 46 28 53 29

G_{10}

25 1 49 10 35 3 19 43 17 60 34 57 18 50 41 11
59 44 9 52 51 42 33 27 39 14 21 4 54 53 29 47
22 7 5 46 15 38 55 45 28 6 62 31 30 12 37 13

G_{11}

9 50 33 59 19 52 3 27 1 44 18 41 2 34 25 60
43 57 58 36 35 26 17 11 23 61 5 55 38 37 13 31
6 54 20 30 62 22 39 29 12 53 46 15 14 63 21 28

G_{12}

58 34 17 43 3 36 52 11 50 57 2 25 51 18 9 44
27 41 42 49 19 10 1 60 7 45 20 39 22 21 28 15
53 38 4 14 46 6 23 13 63 37 30 62 61 47 5 12

G_{13}

42 18 1 27 52 49 36 60 34 41 51 9 35 2 58 57
11 25 26 33 3 59 50 44 54 29 4 23 6 5 12 62
37 22 55 61 30 53 7 28 47 21 14 46 45 31 20 63

G_{14}

26 2 50 11 36 33 49 44 18 25 35 58 19 51 42 41
60 9 10 17 52 43 34 57 38 13 55 7 53 20 63 46
21 6 39 45 14 37 54 12 31 5 61 30 29 15 4 47

G_{15}

18 59 42 3 57 25 41 36 10 17 27 50 11 43 34 33
52 1 2 9 44 35 26 49 30 5 47 62 45 12 55 38
13 61 31 37 6 29 46 4 23 28 53 22 21 7 63 39

Appendix IV. Detailed Information about the Computers

Amdahl 5880

MANUFACTURER: Amdahl Corp.
OPERATING SYSTEM: UTS 580 version 1.1.3
VERSION OF C COMPILER: (Version.c) 1.27.1.1
TIMER ACCURACY (INTERVALS/SECOND): 60
BITS PER WORD: 32

Convex C-1

MANUFACTURER: Convex Computer Corp.
OPERATING SYSTEM: Convex UNIX v6.1.33.22
VERSION OF C COMPILER: v2.0.0.1
TIMER ACCURACY (INTERVALS/SECOND): 100
BITS PER WORD: 32 or 64
The vectorizing abilities of this machine were not used.

Cray 2

MANUFACTURER: Cray Research Inc.
OPERATING SYSTEM: UNICOS 3.0
TIMER ACCURACY (INTERVALS/SECOND): 243902439
BITS PER WORD: 64
The vectorizing abilities of this machine were not used.

IBM PC/RT

MANUFACTURER: IBM Corp.
OPERATING SYSTEM: 4.3 BSD UNIX (NORTHSTAR)
TIMER ACCURACY (INTERVALS/SECOND): 100
BITS PER WORD: 32

IRIS 2500T

MANUFACTURER: Silicon Graphics Inc.
OPERATING SYSTEM: GL2-W3.6
TIMER ACCURACY (INTERVALS/SECOND): 60
BITS PER WORD: 32

Sequent Balance 21000

MANUFACTURER: Sequent Computer Systems
OPERATING SYSTEM: DYNIX(TM) v3.0.4 NFS
PROCESSOR: National Semiconductor 32032
TIMER ACCURACY (INTERVALS/SECOND): 100
BITS PER WORD: 32

Sun 3

MANUFACTURER: Sun Microsystems, Inc.
OPERATING SYSTEM: Sun UNIX 4.2 Release 3.4
PROCESSOR: Motorola Corporation 68020
TIMER ACCURACY (INTERVALS/SECOND): 60
BITS PER WORD: 32

VAX 11/780

MANUFACTURER: Digital Equipment Corp.
OPERATING SYSTEM: 4.3 BSD UNIX
TIMER ACCURACY (INTERVALS/SECOND): 100
BITS PER WORD: 32

VAX 11/785

MANUFACTURER: Digital Equipment Corp.
OPERATING SYSTEM: 4.3 BSD UNIX
TIMER ACCURACY (INTERVALS/SECOND): 100
BITS PER WORD: 32

Appendix V. Timings for the DES and Password Encryption Routines

The tables below indicate the mean time per call to the routines. These tables were used to derive the tables in section 6.

Execution Time Per Call (seconds): DES Encryption Routines
UNIX Interface

computer	6 bit path		12 bit path		standard
	without *G*	with *G*	without *G*	with *G*	function
Amdahl 5880	2.829e-04	2.823e-04	2.547e-04	2.539e-04	2.031e-03
Convex 1 (32)	1.789e-03	1.800e-03	1.506e-03	1.523e-03	7.670e-03
Convex 1 (64)	1.572e-03	1.578e-03	1.358e-03	1.363e-03	7.632e-03
Cray 2	5.398e-04	5.391e-04	4.989e-04	4.977e-04	5.126e-03
IBM PC/RT	1.035e-03	1.034e-03	9.417e-04	9.343e-04	8.163e-03
IRIS 2500T	2.023e-03	2.035e-03	1.541e-03	1.545e-03	1.692e-02
Sequent 21000	5.687e-03	5.837e-03	5.192e-03	5.179e-03	5.058e-02
Sun 3/50	1.983e-03	2.017e-03	1.538e-03	1.565e-03	1.454e-02
VAX 11/780	4.556e-03	4.741e-03	4.536e-03	4.649e-03	2.788e-02
VAX 11/785	3.194e-03	3.176e-03	2.867e-03	2.788e-03	2.166e-02

Execution Time Per Call (seconds): DES Encryption Routines				
computer	6 bit path		12 bit path	
	without *G*	with *G*	without *G*	with *G*
Amdahl 5880	1.694e-04	1.688e-04	1.404e-04	1.373e-04
Convex 1 (32)	1.312e-03	1.308e-03	1.020e-03	1.049e-03
Convex 1 (64)	1.092e-03	1.090e-03	8.829e-04	8.889e-04
Cray 2	2.769e-04	2.764e-04	2.373e-04	2.367e-04
IBM PC/RT	6.214e-04	6.166e-04	5.303e-04	5.204e-04
IRIS 2500T	1.303e-03	1.305e-03	8.180e-04	8.192e-04
Sequent 21000	2.951e-03	2.938e-03	2.412e-03	2.378e-03
Sun 3/50	1.288e-03	1.314e-03	8.373e-04	8.701e-04
VAX 11/780	2.277e-03	2.436e-03	2.443e-03	2.386e-03
VAX 11/785	1.384e-03	1.489e-03	1.227e-03	1.079e-03

Execution Time Per Call (seconds): Password Encryption Routines UNIX Interface					
computer	6 bit path		12 bit path		standard
	without *G*	with *G*	without *G*	with *G*	function
Amdahl 5880	3.055e-03	3.133e-03	2.699e-03	2.701e-03	5.268e-02
Convex 1 (32)	1.854e-02	1.856e-02	1.202e-02	1.215e-02	1.957e-01
Convex 1 (64)	1.488e-02	1.504e-02	9.984e-03	9.949e-03	1.957e-01
Cray 2	3.151e-03	3.158e-03	2.234e-03	2.213e-03	1.326e-01
IBM PC/RT	1.091e-02	1.110e-02	8.712e-03	8.770e-03	2.101e-01
IRIS 2500T	2.616e-02	2.584e-02	1.427e-02	1.385e-02	4.360e-01
Sequent 21000	5.499e-02	5.553e-02	4.335e-02	4.416e-02	1.299e+00
Sun 3/50	2.518e-02	2.489e-02	1.424e-02	1.398e-02	3.702e-01
VAX 11/780	4.181e-02	4.538e-02	4.291e-02	4.474e-02	7.111e-01
VAX 11/785	2.815e-02	3.102e-02	2.304e-02	2.483e-02	5.551e-01

Execution Time Per Call (seconds): Password Encryption Routines					
computer	6 bit path		12 bit path		standard
	without *G*	with *G*	without *G*	with *G*	function
Amdahl 5880	2.888e-03	2.945e-03	2.443e-03	2.442e-03	5.268e-02
Convex 1 (32)	1.758e-02	1.781e-02	1.115e-02	1.127e-02	1.957e-01
Convex 1 (64)	1.387e-02	1.395e-02	8.860e-03	8.929e-03	1.957e-01
Cray 2	2.798e-03	2.818e-03	1.854e-03	1.861e-03	1.326e-01
IBM PC/RT	9.226e-03	9.229e-03	7.504e-03	7.596e-03	2.101e-01
IRIS 2500T	2.516e-02	2.481e-02	1.328e-02	1.291e-02	4.360e-01
Sequent 21000	5.187e-02	5.385e-02	3.980e-02	4.062e-02	1.299e+00
Sun 3/50	2.426e-02	2.407e-02	1.336e-02	1.316e-02	3.702e-01
VAX 11/780	3.768e-02	4.614e-02	4.008e-02	4.169e-02	7.111e-01
VAX 11/785	2.719e-02	2.780e-02	2.010e-02	2.211e-02	5.551e-01