A High-Speed Software DES Implementation

David C. Feldmeier
Computer Communication Research Group
Bellcore,Morristown, NJ 07962

June 1989

Abstract

This paper describes a high-speed software implementation of the Data Encryption Standard (DES) cipher algorithm and the design tradeoffs necessary for high-speed operation. Many of the techniques presented in this paper may be applicable to other product ciphers.

1 Introduction

Data security is becoming more important as more information is sent across networks. Once information is injected into the network, it may be intercepted in any number of locations. One way to protect information, whether it is transmitted or remains on a host computer, is to encrypt it. A popular form of encryption is the Data Encryption Standard or DES.

This paper describes a high-speed software implementation of the DES encryption algorithm. The encryption rate on a DEC 3100 or a Sun 4 is about 1.1 megabits per second. The implementation and the design tradeoffs necessary for high-speed encryption are described. Many of the techniques used in this paper may be applicable to other product ciphers.

The first section of this paper discusses the DES algorithm, as well as some additions that increase the security provided by encryption. A variety of techniques are described that can be used to increase the performance of a DES implementation. In addition to these techniques, unique aspects of this particular implementation are described. The DES implementation is also compared with other high-speed DES implementations. Finally, performance measurements for the DES implementation are presented.

2 Data Encryption Standard

The Date Encryption Standard (DES) was adopted as a Federal standard in 1976. The DES (technically the Data Encryption Algorithm or DEA) is based on the "Lucifer" encryption algorithm that was developed by IBM.

2.1 The DES Algorithm

Figure 1 shows the block diagram of the DES algorithm. DES is a product cipher that takes a 64-bit plaintext block and a 56-bit key and returns a 64-bit ciphertext block. The algorithm begins by permuting the plaintext with the *initial permutation* (IP). The initial permutation is followed by sixteen product-transformation/block-transformation rounds. The product-transformation is denoted f and the block-transformation is the swap of L and R after each round.

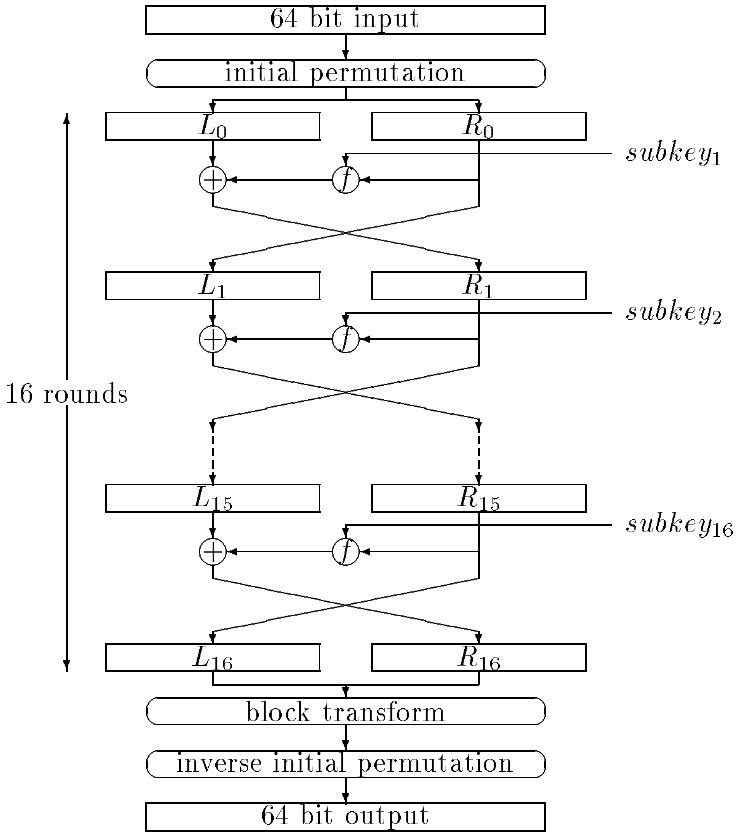

Figure 1: A Block Diagram of the DES Algorithm.

The 16 product-transformation/block-transformation rounds are followed by another permutation, called the *inverse initial permutation* (IP^{-1}). The IP and the IP^{-1} permutations are inverses of one another, which means that a 64 bit block that is passed through one and then the other returns the original 64 bit block.

Each of the 16 product-transformation/block-transformation rounds uses a different 48-bit *subkey* that is derived from the original 56-bit key. Each subkey is produced by selecting 48 of the 56 bits in the key and then permuting these 48 bits. The selection/permutation operation is constant and does not depend on the specific key, although all of the subkeys are dependent on the key.

The product transformation operates as follows. The 64 bit block from the previous step is broken into two pieces of 32 bits each, called *left* (L) and *right* (R). R is passed through the f transformation along with a 48-bit subkey; the result then is exclusive-ORed with L. Figure 2 shows a block diagram of the f function. R is first passed through an expansion function, E. The expansion function takes the 32-bit input and produces a 48-bit output by replicating some of the input bits. The output of the expansion function E is then exclusive-ORed with the 48-bit subkey for this particular round.

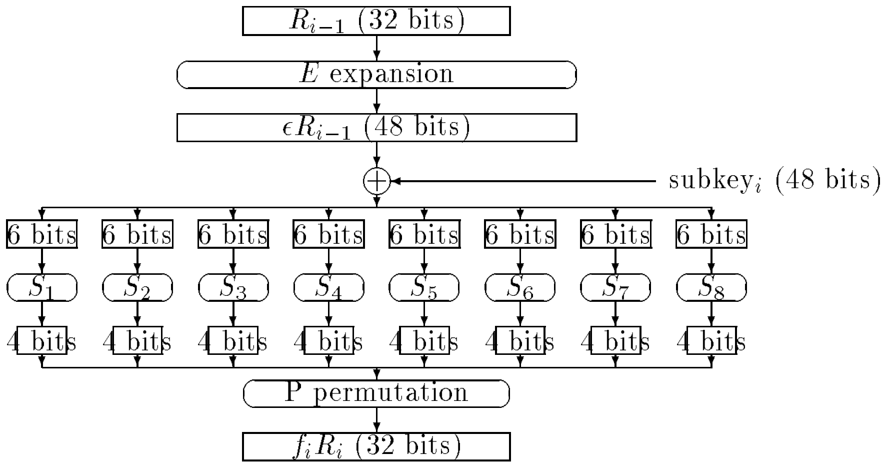

Figure 2: A Block Diagram of the f function in DES.

The resulting 48-bit quantity is now passed through eight S boxes; each S box computes a different function. Each of the S boxes takes a 6-bit input and produces a 4-bit output. The S boxes are designed so that a one-bit change in the input will change multiple bits in the output. Bits 1-6 are passed through the S1 function, bits 7-12 are passed through the S2 function, etc. This produces a 32-bit output from the 48-bit input, which then passes through permutation P. Since P is a pure permutation, the output of the P function is also 32 bits. The 32-bit quantity leaving the P permutation is exclusive-ORed with L from above (recall that R was passed through the f function and that L,R is the 64-bit input from the previous step). This completes the product transformation.

Let ϵ represent the E expansion, σ represent the S function and π represent the P permutation. Let K_i be the i^{th} subkey, R_i be the i^{th} value of Right, and L_i be the i^{th} value of Left. By definition:

$$L_i \equiv R_{i-1}$$

$$R_i \equiv L_{i-1} \oplus f_i R_{i-1}$$

$$f_i R_{i-1} \equiv \pi\sigma(K_i \oplus \epsilon R_{i-1})$$

where f_i is the f transformation that uses subkey K_i.

The block transformation is simply swapping L with R. Notice that after the complete round, what was R is unchanged but is now L, and what was L has been altered by the product-transformation and is now R. Therefore two of these rounds are required before all 64 bits have been altered. In the DES algorithm, the product-transformation/block-transformation round is done 16 times.

Identical algorithms are used for encryption and decryption. The only difference is that the order of the subkeys are reversed. The subkeys are generated from the key in encryption order. For decryption, a new set of subkeys is needed such that:

$$subkey'_i \leftarrow subkey_{16-i}$$

2.2 Encryption Modes

An obvious way to encrypt something longer than 64 bits is to break the original message into 64-bit blocks, each of which is encrypted separately. The is referred to as the Electronic Code Book (ECB) mode of DES. A problem with this approach is that identical blocks of plaintext will produce identical blocks of encrypted text, thus perhaps revealing something about the nature of the plaintext. A way to prevent this is to operate DES in the Cipher Block Chain (CBC) mode. The CBC mode requires that an Initialization Vector (IV) be chosen. Let P_i be the i^{th} 64-bit block of the plaintext, let C_i be the i^{th} 64-bit block of the ciphertext, and let $DES(x)$ be the DES encryption of x. For encryption:

$$C_i = \begin{cases} DES(P_0 \oplus IV) & \text{if } i = 0 \\ DES(P_i \oplus C_{i-1}) & \text{if } 0 < i < n \end{cases}$$

where n is the number of blocks to be encrypted. For decryption:

$$P_i = \begin{cases} IV \oplus DES^{-1}(C_0) & \text{if } i = 0 \\ C_{i-1} \oplus DES^{-1}(C_i) & \text{if } 0 < i < n \end{cases}$$

Each block depends on all of the blocks before it, thus patterns in the plaintext are obscured.

CBC mode also has the property of being self-healing; if one of the ciphertext blocks is damaged, at most two blocks of the recovered plaintext will be affected. If C_i is damaged, the only plaintext blocks that depend on C_i are P_i and P_{i+1}, so only these two plaintext blocks are affected.

The implementation described in this paper can do either ECB or CBC mode encryption and decryption; CBC mode is the default. Also, the IV is set to all zeros to be compatible with other DES implementations.

This has been a basic introduction to the DES algorithm. For a more complete description, see [6, 7].

3 DES Implementation

This section describes some of the ideas behind the implementation of the DES algorithm and why it is fast. The implementation is designed for 32-bit machines; it is possible to run it on other size machines with minor modifications.

3.1 Algorithm Modifications

The basic DES algorithm can be modified in a number of ways that do not change the function computed by the program. Within each DES encryption there are 16 product-transformation/block-transformation pairs. The block transformation is simply the swapping of the R 32 bits with the L 32 bits. To avoid the block transformation at the end of each product-transformation, there are two different product transformations: one works the usual way and the other operates on L as if it were R and vice-versa. Using these two product transformations alternately eliminates the need for a block transformation between product transformations. The only problem with this scheme is that after the last product-transformation, the L and R blocks are swapped. This reversal is incorporated into the final IP^{-1} because a swap and a permutation is just a permutation. By definition:

$$L_i \equiv R_{i-1}$$

$$R_i \equiv L_{i-1} \oplus f_i R_{i-1}$$

$$f_i R_{i-1} \equiv \pi\sigma(K_i \oplus \epsilon R_{i-1})$$

where f_i is the f transformation that uses subkey K_i. After two rounds:

$$L_{i+1} = L_{i-1} \oplus f_i R_{i-1}$$

$$R_{i+1} = R_{i-1} \oplus f_{i+1}(L_{i-1} \oplus f_i R_{i-1})$$

Since there are 16 rounds in DES, 8 double-rounds of the following form can be used instead:

$$L_{i+1} \equiv L_{i-1} \oplus f_i R_{i-1}$$

$$R_{i+1} \equiv R_{i-1} \oplus f_{i+1} L_{i+1}$$

Notice that no intermediate values of R_i and L_i need be retained. After one double-round:

$$L_{i+1} = L_{i-1} \oplus f_i R_{i-1}$$

$$R_{i+1} = R_{i-1} \oplus f_{i+1}(L_{i-1} \oplus f_i R_{i-1})$$

which is the same as before. In effect, the swap has been built into the iteration.

3.2 Table Lookup

The program gets a lot of its speed from a space-for-time tradeoff - almost everything in the program is done by table lookup. Ideally, a table lookup would take the entire input (up to 64 bits) and return the entire output (up to 64 bits). Of course, the maximum size of the input and output of a table are limited by the virtual memory size and the bus width. Thus, on a 32-bit bus, a table with a 64-bit output requires that two lookups be done, one for the low 32 bits and one for the high 32 bits. As for the input to the table, even a 32-bit input would be completely impractical because it is desirable to keep all of the tables in RAM for fast access time and to prevent swapping.

To replicate the effect of a single large lookup table in several small lookup tables presents a problem. The problem is that groups of input bits may exist such that all the bits must be read simultaneously to produce a result. An example of this is the S boxes in DES. Each S box takes a 6-bit input and produces a 4-bit output. Because all 6 bits are simultaneously necessary (because the S box is non-affine), all S box lookups must be done in multiples of 6 bits.

However, if groups are independent, then the table lookup can be broken into manageable pieces (ideally of equal size to minimize the total table size), each of which can be looked up independently. The results can then be logical-ORed together to produce the appropriate output. Because the physical memory available to an application in most workstations is only a few megabytes, this limits the size of the tables.

Changing one lookup to two produces a substantial change in total table size. The ratio is $2^{(n/2)+1}/2^n$ or $2^{1-(n/2)}$, where n is the number of bits used for the table index. The fewer the table lookups, the higher the speed, but also the more memory that is used. The DES implementations should use as few table lookups as possible given the memory constraints.

3.3 System Issues

The basic round within DES is run 16 times and anything that can be done to speed up this basic round will be multiplied by 16 so it is important that the basic round run efficiently.

It is best to avoid instructions whenever possible. For example, with the P table lookup, rather than logical-ORing the table entries together and then exclusive-ORing the result into the L register, it is faster to exclusive-OR the intermediate results into the L register directly, saving one instruction per round (16 instructions total).

In general it is best to keep the number of variables used by the DES program small so that most of them can be kept in registers. On a fast processor, particularly those with caches, memory fetches slow down the system.

For processors with a cache, the main DES rounds should be executed within loops. The code within loops of the programs are kept small enough to fit into a processor cache. The execution speedup achieved by the cache more than compensates for the running time of the extra instructions that support the loop. Without a cache, the loops should be unwound so that the overhead of instructions for the loop is avoided. The program has a compile-time option to structure the program appropriately depending on whether there is a cache.

Another important point is to take advantage of the processor instruction set, specifically whether the machine is a *Reduced Instruction Set Computer* (RISC) or a *Complex Instruction Set Computer* (CISC). Examples of RISC machines are the Sun 4 and the DEC 3100. Examples of CISC machines are the Sun 3 (68020 based) and the VAX.

A useful feature of CISC processors is auto-increment mode. If possible, it is best to step through tables one element at a time so that auto-increment mode can be used. This is particularly useful for accessing the subkeys. The high and low words of the subkeys alternate and they are extracted one at a time with auto-increment mode. On a RISC machine, a separate addition must be done for each increment, so it does not matter what the step size is. Sometimes a different step size can lead to more efficient operation. Another feature of CISC processors are instructions specifically designed for efficient loops. In particular, it is faster to count down to zero with a single instruction that does the compare-decrement-branch function.

RISC machines generally contain a large number of registers and performance is enhanced if often-used constants are kept in registers. Large constants take two instructions to load into the processor rather than one (this is probably because the fixed-length RISC instructions can piggyback only small constants). This means that performance can be increased by keeping commonly used constants in registers, so that only a single instruction is needed to perform an operation with a constant.

The DES implementation is written in C and should be portable with little trouble. Although the programs are in C, for some machines there are special cases inserted into the code. In particular, sometimes a compiler cannot be convinced to generate efficient assembly code for some portion of C code. An example of poorly generated code is the loop instructions. Compilers often generate non-optimal loops, particularly for CISC processors, which often have good instructions for loops. One case where the compiler is not at fault is the lack of a bit rotation operator in C. Many processors have a bit rotation instruction, but the equivalent operation in C takes 3 assembly language instructions. In both of these cases there are compile time options to replace some portions of the C code directly with assemble language instructions.

Another possibility is to edit the assembly language that the compiler produces. Because the VAX compiler is reluctant to use all of the processor registers, speed-des on the VAX uses a SED script that goes through the assembly code and replaces certain memory references with register references.

Fast DES implementations by Bishop[1] and Karn[5] each utilize most of the suggestions above.

4 The Speed-Des Implementation

The DES algorithm has 48-bit, 56-bit and 64-bit wide paths. This program was written with 32-bit wide CPU paths in mind, so the DES data path is broken into high and low pieces of 4 bytes each. Since even 48 bits is more than 32 bits, each operation requires two word manipulations in any case. The DES uses two basic operations: subkey generation and repeated DES encryptions. This section describes subkey generation and table construction as it is done for the DES implementation.

4.1 Subkey Generation

Subkey generation means taking the 56-bit key and generating the 16 48-bit subkeys for DES. The generation of subkeys involves taking the 56-bit key, putting it through a permutation (Permuted Choice 1), breaking the result into low and high 28-bit wide paths, doing some number of left circular shifts on each path (the number depending on the particular subkey), combining the paths and passing the results through another permutation (Permuted Choice 2).

Because the key is computed once for each encryption, the speed of subkey generation has relatively little effect on the program speed. The subkey generation data representation is a one bit per byte format. The original key is expanded from its 8-byte format (7 bits per byte) to a 56-byte format (1 bit per byte). The permutations Permuted Choice 1 and Permuted Choice 2 are table lookups. The rotations are simply done by modular addition to the representation table indices. The resulting subkeys are compressed from a 48-bit format to a 8-byte format (6 bits per byte). The subkeys are always generated in the order necessary for encryption. If a decryption is desired, the subkey order is reversed after generation.

4.2 Data Buffers

Because both the input and output of each encryption is eight bytes, a single data buffer is used for both. In ECB mode, as each 64-bit block is encrypted and written back to the same location.

For encryption in chaining mode, not only is the output written back, but it is also exclusive-ORed with the next input in the buffer to implement chaining. In CBC mode, some care must be taken if the plaintext and ciphertext are longer than the data buffer to handle the boundary conditions properly. When an entire buffer is finished, data buffers 0 through $n-1$ are written to the output file, the next n blocks are read into data buffers 0 through $n-1$ and data buffer n is exclusive-ORed into data buffer 0 and buffer n is cleared. The process then continues with data buffer 0.

For decryption in chaining mode, each of the ciphertext blocks is shifted by one, i.e. ciphertext block i goes into data buffer $i+1$, and data buffer 0 is set to all zeros. Then data buffer i (for $1 \leq i \leq n$) is decrypted and exclusive-ORed with data buffer $i-1$. Once again, some care must be taken if the plaintext and ciphertext are longer than the data buffer. When an entire buffer is finished, data buffers 0 through $n-1$ are written to the output file, data buffer n is moved into data buffer 0, and n blocks are read from the input file into data buffers 1 through n. The process continues with data buffer 0.

4.3 Function Computation

One unique feature of this implementation is that the expansion function E is done as a simple register copy. Why this is possible is explained below.

Another unique feature of this implementation is table lookups for the IP and the IP^{-1} functions. Both functions take 64-bit inputs and produce 64-bit outputs. Since the implementation is targeted for a 32-bit wide machine, the obvious implementation is to do eight 1-byte lookups, each producing two 4-byte words, for a total of 16 memory fetches. Since both IP and IP^{-1} are regular permutations (perfect shuffles), it is possible to cut the number of memory fetches in half by grouping the 32 input bits that produce outputs in the high 32 bits and the 32 input bits that produce outputs in the low 32 bits. These bits then are compacted into a single word, because otherwise the number of lookups would not be reduced. Because the permutations are so regular, this selection and compaction can be done with a few operations. In fact, the same table is used for both the high and low 32 bits of IP, thus cutting the required table size in half. The trade-off is a few logical operations in exchange for 8 memory fetches. Particularly for RISC processors, this is a good exchange.

Because the S boxes and the P permutation are adjacent functions in the DES algorithm, they are combined into a single table lookup to increase speed and reduce memory use.

4.4 Data Representation

Because the expansion function E is so regular, it is possible to do it without table lookup. A representation is devised that does not require a table lookup for the expansion function E and allows the SP function to be implemented as 4 table lookups without the manipulation of individual bits. Such a representation is possible, but requires a strange bit order.

Consider the mapping from R (32 bits) to ϵR (48 bits) as shown in table 1. The first thing to notice is that the table is presented in groups of 6. The reason for this is that the eight S boxes each use six bits for their lookup. Because the S boxes are non-affine, the groups of six cannot be broken up. Therefore, they define the granularity of the table lookups (multiples of six bits along the boundaries shown).

An important aspect of the expansion E is that no input bit from R ever becomes more than two output bits in ϵR. This suggests that simply copying R into a second register will give all 48 bits necessary for the expansion E in two 32-bit words. Designate the two copies of R as A and B.

Since as little manipulation as possible of the 32-bit quantities before the SP lookup is desired, the bit order of the 32-bit R (and symmetrically L) is critical. The aim is to do four lookups, two S boxes at a time. Notice that there is a circular structure to the E expansion and that bits from R that occur on line n also occur on lines $n+1$ and $n-1$ mod 8. This means that the lookup of all odd lines must occur in one copy of R (say A) and the lookup of even lines must occur in the other (B) so that overlapping bits of R may be exclusive-ORed with different key bits.

To minimize table size, lines 1 & 5, 2 & 6, 3 & 7, and 4 & 8 must be organized as blocks of 12 bits with no intervening bits. In addition, the circular structure of the E expansion

ϵR (48 bit)	1	2	3	4	5	6
R (32 bit)	32	1	2	3	4	5
ϵR (48 bit)	7	8	9	10	11	12
R (32 bit)	4	5	6	7	8	9
ϵR (48 bit)	13	14	15	16	17	18
R (32 bit)	8	9	10	11	12	13
ϵR (48 bit)	19	20	21	22	23	24
R (32 bit)	12	13	14	15	16	17
ϵR (48 bit)	25	26	27	28	29	30
R (32 bit)	16	17	18	19	20	21
ϵR (48 bit)	31	32	33	34	35	36
R (32 bit)	20	21	22	23	24	25
ϵR (48 bit)	37	38	39	40	41	42
R (32 bit)	24	25	26	27	28	29
ϵR (48 bit)	43	44	45	46	47	48
R (32 bit)	28	29	30	31	32	1

Table 1: Expansion Function E (mapping from R to ϵR)

requires that the bits for lines 8 and 2 be adjacent to those for line 1, lines 1 and 3 be adjacent to those for line 2, etc. One way to achieve this is to interleave the bits of the pairs of lines. Thus lines 1 & 5 are interleaved, lines 2 & 6 are interleaved, etc. Of course, interleaved lines 1 & 5 are adjacent to interleaved lines 4 & 8 and 2 & 6.

Assume that the bits of a word a numbered such that 0 is the least significant bit and 31 is the most significant bit. The data representation for R starts with the first bit of line 1 in bit 2, the first bit of line 5 in bit 3 and so on, thus interleaving lines 1 and 5. This continues with lines 2 and 6. Notice that bit 10 represents not only the fifth bit of line 1 but also the first bit of line 2, and bit 11 represents the fifth bit of line 5 as well as the first bit of line 6. This pattern continues for lines 3 & 7 and 4 & 8. Notice that lines 4 & 8 wrap around the end of the word, and thus the fourth bit of line 4 is represented by bit 0. This is the 32-bit representation of R (and L):

30	14	29	13	28	12	27	11	26	10	25	09	24	08	23	07	22	06	21	05	20	04	19	03	18	02	17	01	16	32	31	15

To get the 32-bit representation of L (this which is not expanded), add 32 to each of the above numbers. Notice that lines 1 & 5 do not start immediately at the least significant bit; there are two extra bits in the least significant byte. This is because when doing pointer arithmetic, normally the value to be looked up in a table on a machine with a 32-bit (4 byte) word size would have to be multiplied by 4 before addition. In effect, the data representation above "premultiplies" by 4, thus saving an operation in a critical section of the code.

Now apply the mapping from R to ϵR for 1 & 5 and 3 & 7 (which have no overlapping bits in R) to obtain the representation of the 24 of the 48 bits in A:

xx	xx	42	18	41	17	40	16	39	15	38	14	37	13	xx	xx	xx	xx	30	06	29	05	28	04	27	03	26	02	25	01	xx	xx

Now apply the mapping from R to ϵR for 2 & 6 and 4 & 8 (which have no overlapping bits in R) to obtain the representation of the 24 of the 48 bits in B:

45	21	44	20	43	19	xx	xx	xx	xx	36	12	35	11	34	10	33	09	32	08	31	07	xx	xx	xx	xx	24	48	23	47	46	22

This data representation is one of several possible representations that has the desired properties. This particular representation was chosen because this program was an outgrowth of a high-speed Unix crypt implementation[4]. Table 2 shows the table sizes for speed-des.

Table	Bytes
SP	65536
IP	4096
IP^{-1}	8192
total	77,824

Table 2: Table Size in Speed-Des

5 Implementation Alternatives

The speed-des implementation is faster than any other known DES implementation (except possibly an implementation by Davio - see below). Others have also implemented high-speed DES functions and this section discusses their implementations.

Matt Bishop wrote his *deszip* code at Dartmouth[1]. The deszip code has a variety of options to allow various speed/size tradeoffs. Keys can be computed either the standard way or with a permutation per subkey. The SPE lookups can be done 6 or 12 bits at a time. It is also sophisticated about taking advantage of the machine architecture to improve its speed.

His implementation uses a transformation of DES described by Davio[2]. The recurrence equation is rewritten so the the E expansion is combined with the S function and the P permutation; thus, only a single lookup table is needed for the DES rounds. An E expansion (ϵ) must be combined with the IP table. An E expansion inverse (ϵ^{-1}) is needed at the end of the DES encryption, but this can be combined with the IP^{-1} table. The problem with this transformation is that the entire datapath of the DES function is 48-bits wide. This is not a problem on a machine that is 48-bits wide or wider, but it is on a machine with a smaller bus. This doubles the number of memory accesses for the SPE table because each 48-bit word requires two 32-bit fetches. The basic SPE table lookup requires 4 basic lookups, each of which has an 8 byte output, which means that each lookup requires two memory fetches on a 32-bit machine. Because memory fetches slow down the DES program, speed-des executes faster because fewer (4) memory references are needed. The E expansion technique used for speed-des has little advantage over a combined SPE table for machines that are 48-bits wide or wider.

Davio, et al present a very sketchy paper on a software DES implementation[3]. It is very difficult to determine anything about the implementation from the paper except for the speed. They claim a speed of 150 kbit/second on a VAX 11/780, which is twice the speed of the speed-des implementation presented in this paper. However, it is difficult to evaluate the validity of this claim from the paper. In particular, the speed seems inconsistent with an implementation that uses the DES transformations suggested by Davio in a previous paper[2].

Phil Karn also did a fast implementation of DES[5]. Karn's subkey generation is identical to that is speed-des, for the good reason that speed-des "borrowed" the code from his implementation. His implementation combines the S and P lookups into a single table that requires eight 6-bit lookups. How he deals with the E expansion is similar to speed-des: he takes the appropriate 6 bits of the 32-bit R, exclusive-ORs it with 6 bits of subkey and then does the appropriate SP lookup. This is not quite as good as the E expansion used in speed-des because exclusive-ORing the subkey takes eight operations rather than two.

6 Speed Measurements

The following timings were done for encrypting a 5 megabyte file of random data. These result are for DES in block-chaining mode. The results for speed-des can be seen in table 3.

Machine	User Time (Seconds)	System Time (Seconds)	Total Time (Seconds)	Encryptions per Second	Microseconds per Encryption	Kilobits per Second
DEC 3100	29.7	6.7	36.4	17170	58.2	1098.9
Sun 3/50	135.5	8.9	144.4	4328	231.0	277.0
Sun 3/75	112.4	5.4	117.8	5306	188.5	339.6
Sun 3/200	75.9	6.4	82.3	7594	131.7	486.0
Sun 4/280	28.7	3.9	32.6	19172	52.2	1227.0
VAX 11/780	535.6	9.5	545.1	1147	872.2	73.4
VAX 8650	76.0	5.3	81.3	7688	130.1	491.9

Table 3: Speed-Des Speeds on Various Machines

On a DEC 3100, profiling of speed-des gave the results in table 4. As expected, the 16 product-transformation rounds account for the majority of the instructions. The IP^{-1} function uses more instructions than the IP function because of the internal data representation used by speed-des complicates the bit-compression needed to reduce the number of table lookups.

Table 5 shows the speed of speed-des compared with other DES implementations. The implementation by Davio seems to be twice as fast as speed-des, although it is hard to evaluate the validity of this claim from the information presented in the paper[3]. In particular, their DES may not include block chaining and if performance was tested in a loop rather than by encrypting a string of random numbers, then the cache would allow the program to run faster than it would when actually performing an encryption.

The reason that implementations by others do not speed up as well when they are moved to a RISC machine is that they do many table lookups. The speed-des program is larger, but the number of table lookups is smaller, thus allowing it to run faster on a RISC machine.

Purpose	Percentage
IP	8.54%
Rounds	71.82%
IP^{-1}	11.55%
Other	8.09%
Total	100.00%

Table 4: Percentage of Instructions for DES Functions

Person	Version	DEC 3100	Sun 3/50	Sun 4/280	VAX 11/780
Sun	SunOS		608.1	2906	
Bishop	deszip		650.2		220.5
Feldmeier	speed	17170	4328	19172	1147
Karn	?	3217	760.5	3362	286.2
Davio	?				2344?

Table 5: DES Encryption Speeds of Various Implementations (Encryptions/Second)

7 Conclusion

This paper describes a high-speed implementation of the DES written in C. It discusses how the DES algorithm works and how it can be modified for higher speed. Implementation decisions and programming tricks for high speed are also discussed. It is worth pointing out that no real breakthroughs were required for the results obtained. What was required is a good understanding of the algorithm and of the computer systems that it was implemented on. The most unique part of the implementation is the unusual bit order used in the 48-bit expansion of the f function that allows fast E expansion and fast SP table lookup. The implementations of the IP and the IP^{-1} functions is also unique because they require only half as many memory fetches as a more conventional implementation. Both of these contribute to high performance on 32-bit wide machines. The performance of the described DES implementation is about 1.1 megabits per second on a DEC 3100 or a Sun 4.

References

[1] Matt Bishop, "An application of a fast data encryption standard implementation", *Computing Systems*, 1(3):221–254, Summer 1988.

[2] Marc Davio, Yvo Desmedt, Marc Fosseprez, Rene Govaerts, Jan Hulsbosch, Patrik Neutjens, Joos Vandewalle, and Pascal Wouters, "Analytical characteristics of the DES", In *Proceedings of Crypto '83*, pages 171–202, August 1983.

[3] Marc Davio, Yvo Desmedt, Jo Goubert, Frank Hoornaert, and Jean-Jacques Quisquater, "Efficient hardware and software implementations for the DES", In *Proceedings of Crypto '84*, pages 144–146, August 1984.

[4] David C. Feldmeier, "A high speed crypt program", Technical Memo TM-ARH-013711, Bell Communications Research, April 1989.

[5] Philip R. Karn, Karn's DES implementation source code.

[6] Alan G. Konheim, *Cryptography: A Primer*, John Wiley & Sons, 1981.

[7] Charles P. Pfleeger, *Security in Computing*, Prentice Hall, 1989.

Glossary

ϵ Denotes the E expansion function of DES.

π Denotes the P permutation of DES.

σ Denotes the S function of DES.

CISC Complex Instruction Set Computer - refers to processors such as the MC68020 that have many machine instructions available.

DES Data Encryption Standard.

E expansion An expansion from 32 bits to 48 bits in the product transformation of DES.

f Denotes the product transformation of DES.

f_i Denotes the product transformation of DES that uses the i^{th} subkey.

IP Denotes the Initial Permutation of the DES.

IP^{-1} Denotes the Inverse Initial Permutation of DES.

K_i Denotes the subkey of DES for iteration i.

P permutation A 32-bit permutation in the product transformation of DES.

RISC Reduced Instruction Set Computer - processors with relatively few instructions that execute in a single clock cycle.

S function A function in the product transformation of DES that consists of eight non-affine functions, each of which takes a 6-bit input and produces a 4-bit result.

SED The Unix Stream Editor.

SP table The combination of the S function and the P permutation realized as a single lookup table.

SPE table The combination of the S function, the P permutation and the E expansion realized as a single lookup table.