1992

An encryption package for UNIX

Dzung Le
University of Wollongong

AN ENCRYPTION PACKAGE FOR UNIX

A minor thesis submitted in partial fulfilment of the requirements for the award of the degree of

HONOURS MASTER OF SCIENCE

(Computer Science)

from

THE UNIVERSITY OF WOLLONGONG

by

DZUNG LE, BE(Saigon),BAppSc,MAppSc(UTS)

DEPARTMENT OF COMPUTER SCIENCE

1992

Abstract

Cryptography has a much wider application than secrecy, such as authentication and digital signature. There are two common types of cryptographic algorithms - symmetric and asymmetric. The Data Encryption Standard (DES) is the first and only, publicly available cryptographic algorithm that has been widely used in commercial communication. The DES is a block cipher symmetric algorithm and its design is based on the Shannon's two general principles - diffusion and confusion. With the decreased cost of hardware and a better understanding of block ciphers and cryptanalysis techniques, a number of DES-like ciphers have been proposed as the replacement for DES. One-way hashing functions are useful in implementing any digital signature schemes. A hashing function accepts a variable size message M as input and outputs a fixed size representation of the message H(M). A number of hashing functions of fixed size or variable size message digest have been proposed. The cryptographic primitives (des, feal, loki, khufu, and khafre), block cipher based hashing algorithms (sbh and dbh), and key-less hashing algorithms (md4, md4x, md5 and haval) have been implemented as standard commands and C library calls for the UNIX Operating System.

Acknowlegement

I wish to thank A/Prof J.Pieprzyk, my supervisor, and Dr.Y.Zheng for introducing me to computer security and for guiding me throughout the period of study.

Table of contents

1. Introduction

The first indications of cryptography go back almost as far as the origin of writing [Kah67]. Secure transmission of private information is a basic requirement between a sender and a receiver. The problem in cryptography is devising procedures to transform message (plaintext) into cryptograms (ciphertext) that can withstand intense cryptanalysis. Today, cryptography can be used to hide the meaning of information in any form, such as data stored on a disk or messages in transit through a communication network. Cryptography has a much wider application than secrecy, such as authentication and digital signature.

There are two types of cryptographic algorithms - symmetric and asymmetric algorithms. In the paradigm of the symmetric cryptography or single key cryptography (Figure 1.1), there is only a single piece of private and necessarily secret information - the key - known to and used by the sender and also known and used by intended receiver to decrypt the cipher. With the asymmetric cryptography or public key cryptography (Figure 1.2) - [Seb89], there are two keys, one publicly known to and used by the sender, and the other secret key known and used by the receiver to decrypt the cipher.

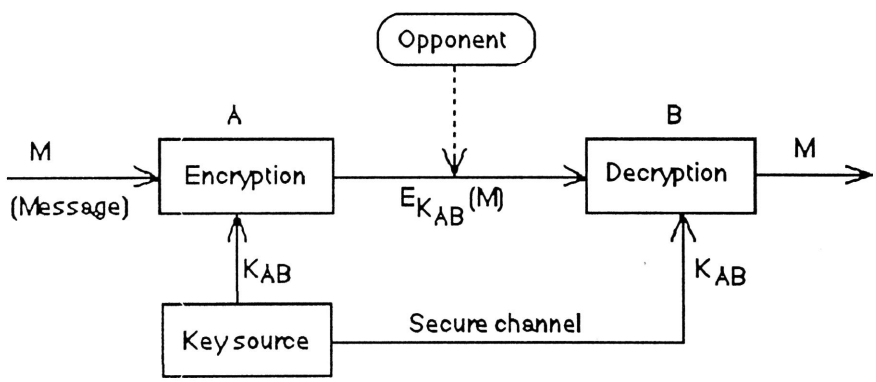


Figure 1.1 A schematic for a symmetric cryptosystem

It is possible to design unbreakable ciphers. This concept is based on the Shannon theory of perfect security [Sha49]. A necessary and sufficient condition for perfect secrecy is that for every ciphertext C, $P_M(C) = P(C)$, that is, the probability of receiving ciphertext C is independent of encrypting it with plaintext M. To do so, the key must be randomly selected - the key must have the same chance of being chosen - and used only once. Furthermore, the length of the key must be equal to or greater than the length of the plaintext to be

enciphered. This type of cipher is known as one-time pad cipher, is impractical for most applications, because of the key length is as long as the length of the message.

There are two ways to design a strong cryptographic algorithm. First, the algorithm is designed based on the possible methods of solution available to the cryptanalyst. Second, the algorithm is constructed in such a way that breaking it requires the solution of some known problem, one that is difficult to solve, or computationally infeasible. The Data Encryption Standard (DES) [NBS77] was designed using the first approach. While the public key RSA scheme was designed using the second approach.

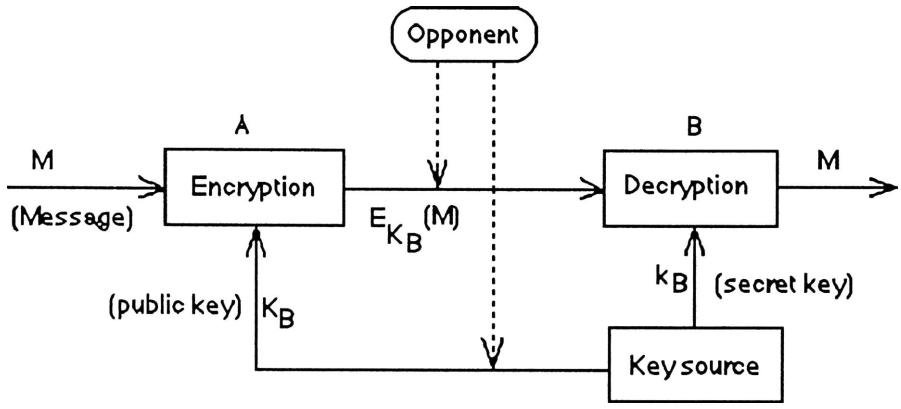


Figure 1.2 A schematic for an asymmetric cryptosystem

The rest of this report is divided into two main parts. The first part is used to introduce symmetric block ciphers, the modes of operation and their cryptographic primitives - DES, FEAL-NX, Loki, Khufu, and Khafre. The second part is used to introduce one-way hash functions, their applications and schemes of implementing key hashed functions based on symmetric block ciphers and keyless hash functions. The UNIX implementations of cryptographic primitives and one-way hash functions are described in the appendix.

2. Symmetric Block Ciphers

Cryptographic systems are also classified into block and stream ciphers. The classification is based on the size of the objects to which the cryptographic transformation is applied [Sim92]. If the objects are single symbols (normally an alphabetic or numeric character) the system is called a stream cipher, while if the object is made up of several symbols, the system is said to be a block cipher. In the second case, blocks could be considered to be symbols from a larger symbol alphabet, however the distinction between stream and block ciphers is a useful device when considering such problems as error propagation, synchronisation, and especially of the achievable communication data rates and delays.

By block cipher, a symmetric key cryptosystem transforms message (plaintext) blocks of fixed length (k bits) into ciphertext blocks of the same length, under the control of a key k (m bits). Thus, if M is an n-bit message block then $C = E_k(M)$ is the corresponding n-bit ciphertext block when key k is used. In addition, we have $M = D_k(C) = D_k(E_k(M))$.

The search for a secure cryptographic systems has often resulted with a mix of success and failure, and the subsequent development of more secure cryptosystems. One of such events is the best known and widely used symmetric algorithm - the Data Encryption Standard. The DES [NBS77] is the first, and only, publicly available cryptographic algorithm that has been endorsed by the US Government. Although there has been some controversy about its key size, DES has successfully resisted all cryptanalysis attacks and been widely accepted in commercial communication. The DES is a symmetric block cipher algorithm which transforms a 64-bit input data to a 64-bit output cipher.

Although the use of DES was reaffirmed only until 1992, the algorithm serves as the best example of using the Shannon's two general principles [Sha49] - diffusion and confusion. The diffusion is to spread out of the influence of a single plaintext digit over many ciphertext digits so as to hide the statistical structure of the plaintext. An extension of this idea is to spread the influence of a single key digit over many digits of ciphertext so as to frustrate a piecemeal attack on the key. The confusion is to use enciphering transformations that complicate the determination of how the statistics of the ciphertext depending on the statistics of the plaintext.

The common approach to apply the Shannon's principles is the product cipher, that is, a cipher that can be implemented as succession of simple ciphers, each of which adds its modest share to the overall large amount of diffusion and confusion. The approach is to use a sequence of transformations - substitutions and transpositions. That is, iterated

cryptosystems are a family of cryptographically strong functions based on iterating a weaker function n times.

The DES is the product ciphers in terms of employing 16 rounds of successive encipherment, each round consisting of rather simple transpositions and substitutions of 4-bit groups. Only 48 key bits are used to control each round, but these are selected in a random-appearing way for successive rounds from the full 56-bit key. That is, the round function is a function of the output of the previous round and of a subkey which a key-dependent value calculated via a key scheduling algorithm.

The round function is usually based on S-boxes, bit permutations, arithmetic operations and the addition modulo 2 operations. The S-boxes are nonlinear translation tables mapping a small number of input bits to a small number of output bits. The security of the cryptosystem crucially depends on their choice [Bih91a]. The bit permutation is used to rearrange the output bits of the S-boxes in order to make the input bits of each S box in the following round depend on the output of as many S-boxes as possible.

The basic design principles in DES seem sound [Mer90], due to the fact that DES has not been publicly broken. But from the day of designing DES until now, there are several changing factors, such as the decreased cost of hardware, a better understanding of the cryptanalysis techniques. Several DES-like block ciphers have been designed as a replacement for DES. The Fast Encryption Algorithm (FEAL-NX), by A.Shimizu and S.Miyaguchi [Shi87], [Miy90], suitable for software and hardware implementation. The LOKI encryption algorithm, by L.Brown, J.Pieprzyk, and J.Seberry [Bro90], [Bro91], the Loki detailed structure has been designed to remove operations which impede analysis or hinder efficient implementation which do not add to the cryptographic security of the algorithm. The KHUFU and KHAFRE encryption algorithm, by R.Merkle [Mer90], Khufu is designed for fast bulk encryption of large amounts of data by using precomputed tables, and Khafre is designed to encrypt small amounts of data. The alternative algorithms are claimed to have the equal security level with DES, but they remain to be tested.

3. Applications of DES-like Ciphers

The symmetric block cipher algorithms are the basic building block for data protection. That is, the algorithms provide the user with a set of primitive functions each of which transforms a 64-bit input to a 64-bit output under the control of 64-bit key. In the DES for example, the user selects one of 2^{56} transformation functions which is equivalent to 10^{17} to be used by a particular 56-bit key (64-bit DES key with 8 parity bits). All DES-like algorithms can be used for data encryption, data authentication, or both data encryption and data authentication. The security of the application is based on the security of the selected cryptographic primitive.

3.1. Data encryption

To make DES-like ciphers more useful than just encrypting or decrypting 64-bit block of data, we need to expand the capability of algorithm by using a suitable mode when encrypt data having greater than 64 bits.

The DES algorithm can be used in four modes [NBS80]: electronic code book (**ECB**), cipher block chaining mode (**CBC**), cipher feedback (**CFB**), and output feedback (**OFB**) [Gar91]. These four modes are used to implement UNIX commands of the block cipher algorithms - des, feal, loki, khufu, and khafre - to provide four operational modes to encrypt and decrypt 64-bit blocks. The cipher block chaining (cbc) mode is the default and the recommended mode when encrypting messages or files.

The following is a general description of these four modes:

3.1.1. ECB mode. In electronic code book (Figure 3.1), each block of the input is encrypted using the key, and the output is written as a block. This method is simple encryption of a message, a block at a time. If the same block of 64 bits occurs twice, the cipher block will also be repeated. With random data this would not matter because of its rarity, but redundancy in real messages makes it vulnerable to statistical attack. This method may not indicate when portions of a message have been inserted or removed. It works well with noisy channels - alteration of a few bits will corrupt only a single 64-bit block.

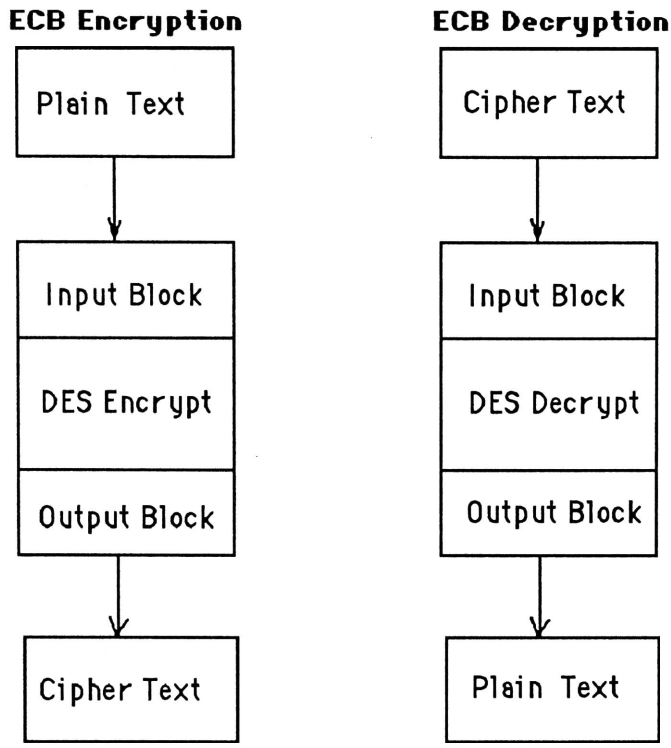
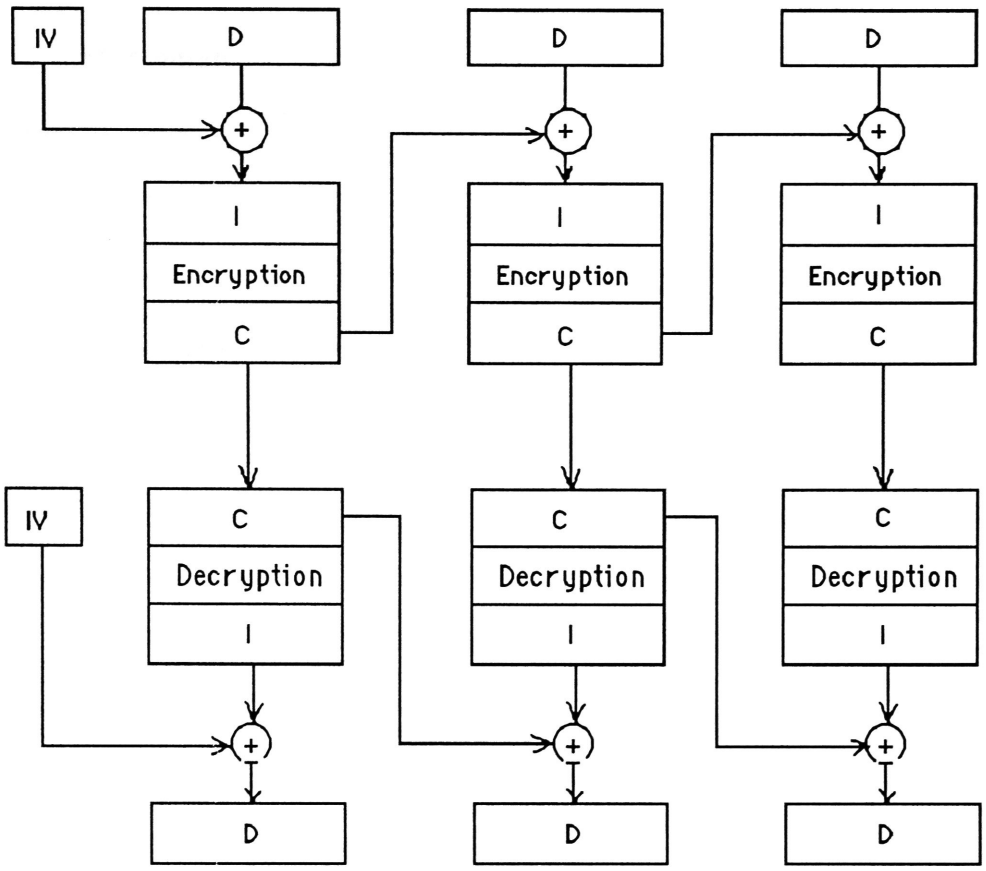


Figure 3.1 Electronic Code Book (ECB) mode

ECB should be used only where the data being encrypted are truly random or have a large random component. Usually these are keys which are being encrypted under another key for storage or transmission through the network.

3.1.2. CBC mode. A message can be divided into blocks of 64 bits and the sequence encrypted with safety if a special feedback arrangement is used. In cipher block chaining mode (Figure 3.2), the plaintext is first XOR'ed with the encrypted value of the previous block. Some known value - initial vector - is used for the first block. The result is then encrypted using the key. Because bits in the message propagate through the entire cipher, the last block can be used as a checksum/signature to check that the ciphertext has not been altered (also see section 6.1). Furthermore, long runs of repeated characters in the plaintext will be masked in the output. CBC mode is the default mode in the des, feal, loki, khufu, and khafre programs. If there is a single error occurred in the ciphertext it will completely change the output of the 64-bit block when it is decrypted.



Legend: D Data block, I Encryption block, C Cipher block
IV Intialization vector, ⊕ Exclusive-OR

Figure 3.2 Cipher Block Chaining (CBC) mode

3.1.3. CFB mode. In cipher feedback mode (Figure 3.3), the key is initially set to some value, and after each block encryption, a portion of the output is shifted into the key. The new key is then used to encrypt the next block of text, and so on. This method is self-synchronising, and enables the user to decipher just a portion of a large database by starting a fixed distance before the start of the desired data. If there is a single error occurred in the ciphertext it will affect both the corresponding plaintext and all of the subsequent block.

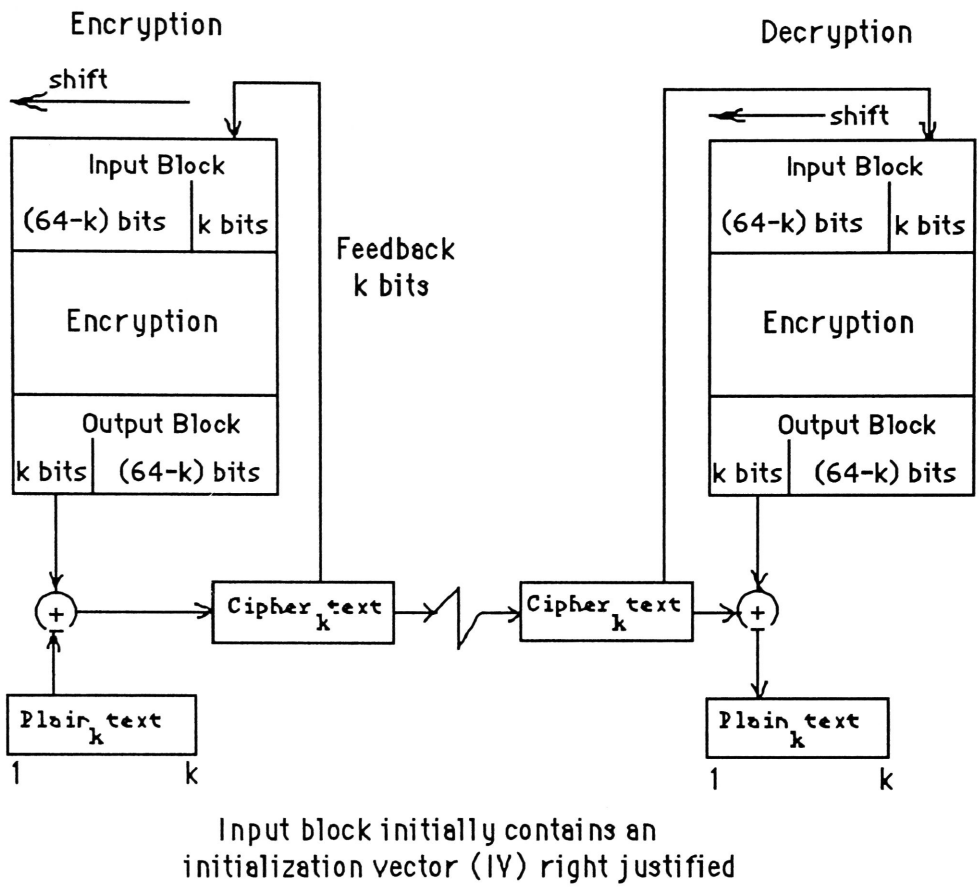


Figure 3.3 K-bit Cipher Feedback (CFB) mode

3.1.4. OFB mode. In output feedback mode (Figure 3.4), the basic algorithm is used to generate a pseudorandom sequence. That is, the key field is encrypted using the ECB mode of des, feal, loki, khufu, or khafre. The result is then used as the key to encrypt the data block, and that output is also used as the new key. Meanwhile, this same output is used as a form of one-time pad to XOR with the plaintext. If there is a single error occurred in the ciphertext it will only affect the corresponding plaintext bit. Also, this mode of encryption is not self-synchronising, and the loss of synchronisation will result in corruption of all subsequent plaintext.

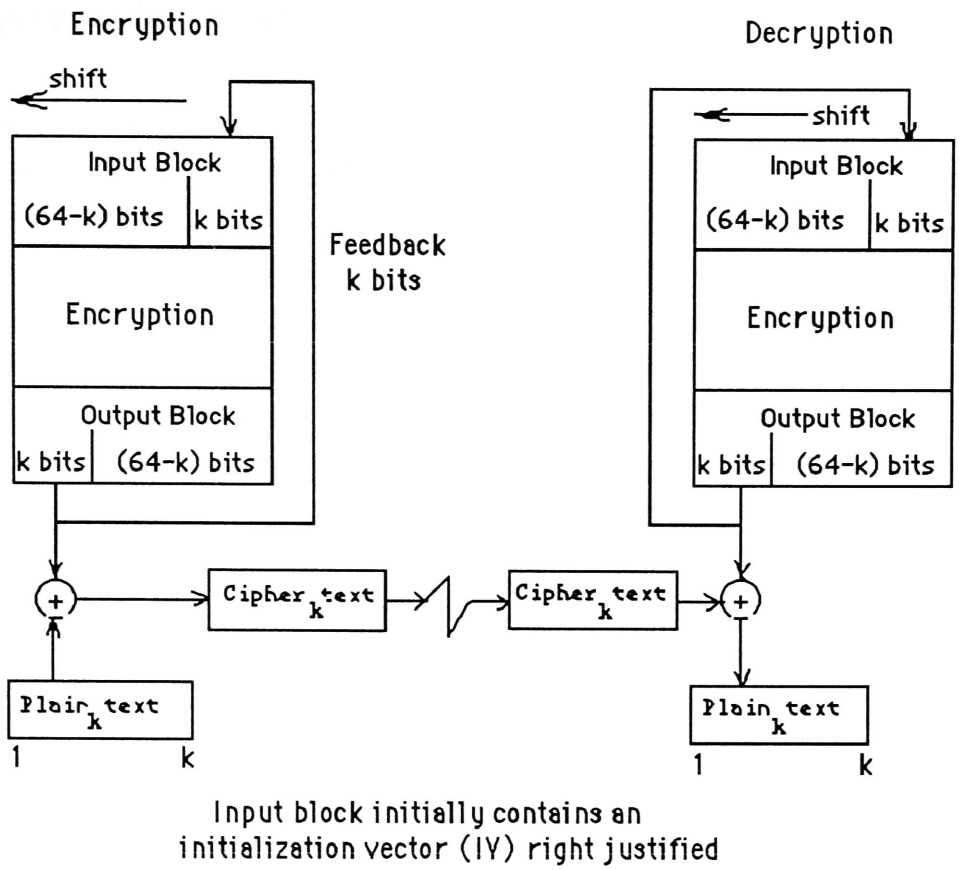


Figure 3.4 K-bit Output Feedback (OFB) mode

3.2. Data Authentication

DES-like ciphers can be used to produce a cryptographic checksum that can be used to protect against both accidental and intentional, but unauthorised data modification. Essentially the data is encrypted by using CBC, CFB, or OFB mode to obtain the final 64-bit cipher block which represent a complex function for the whole data file, or data string. The property of this method is that it is the authentication method in which the user can input the key. And any single bit modification will be propagated to the final cipher block. Section 6 gives a more details of algorithms and actual implementation of this authentication method.

3.3. Data Encryption and Authentication

The same data can be protected by both encryption and authentication. The data is protected from disclosure by encryption and modification is detected by authentication. The authentication process can be applied to either plaintext or ciphertext.

4. Cryptographic Primitives for Block Ciphers

The following section is the general description of symmetric block cipher primitives - the Data Encryption Standard (DES), the Fast Encryption Algorithm family (FEAL), the LOKI encryption algorithm, the Khufu encryption algorithm (KHUFU) and the Khafre encryption algorithm (KHAFRE). The primitives are the basic building blocks for encryption and decryption. The algorithms share the same property that 64-bit plaintext is encrypted to produce 64-bit ciphertext under the control of 64-bit key. There are a few differences among them, such as the extended version of FEAL (FEAL-NX) uses 128-bit key, a different number of rounds, and Khufu uses precomputed S-boxes.

4.1. Data Encryption Standard (DES)

The Data Encryption Standard design was based on Lucifer. Its proper name is DEA (Data Encryption Algorithm) in the US and DEA1 (Data Encryption Algorithm 1) in other countries. It is used to encrypt and decrypt blocks of data consisting of 64 bits under control of a 64-bit key. Decryption is done by using the same key as for encryption, but with the schedule of addressing the key bits altered so that the decrypting process is the reverse of the encrypting process.

A block to be encrypted is subject to an initial permutation IP, then to a 16-round complex key-dependent computation and finally to a permutation which is the inverse of the initial permutation IP^{-1}. The key-dependent computation can be simply defined in terms of a function f, called the cipher function, and a function KS, called the key schedule.

Each round of the algorithm is four separate operations. First, a right half is expanded from 32 bits to 48 bits. Then it is combined with a form of the key. The result of this operation is then substituted for another result and condensed to 32 bits. The sketch of each round calculation is in Figure 4.1.

Let T_i denote the result of the ith iteration and let L_i and R_i denote the left and right halves of T_i respectively; that is, $T_i = L_i R_i$, where

$L_i = t_1 \ldots\ldots t_{32}$,

$R_i = t_{33} \ldots\ldots t_{64}$, then

$L_i = R_{i-1}$

$R_i = L_{i-1} \text{ xor } f(R_{i-1}, K_i)$

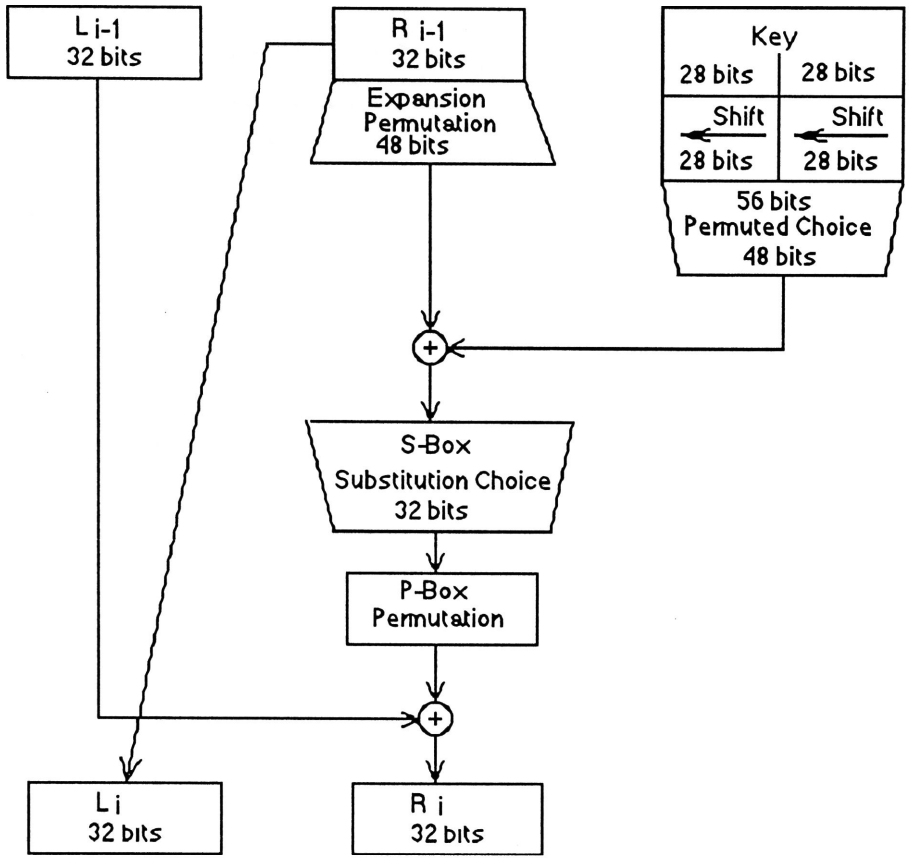


Figure 4.1 Detail of one round calculation

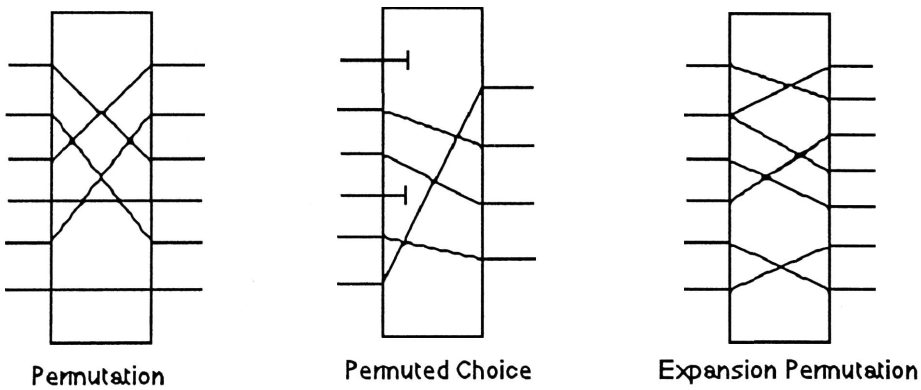


Figure 4.2 Types of permutations

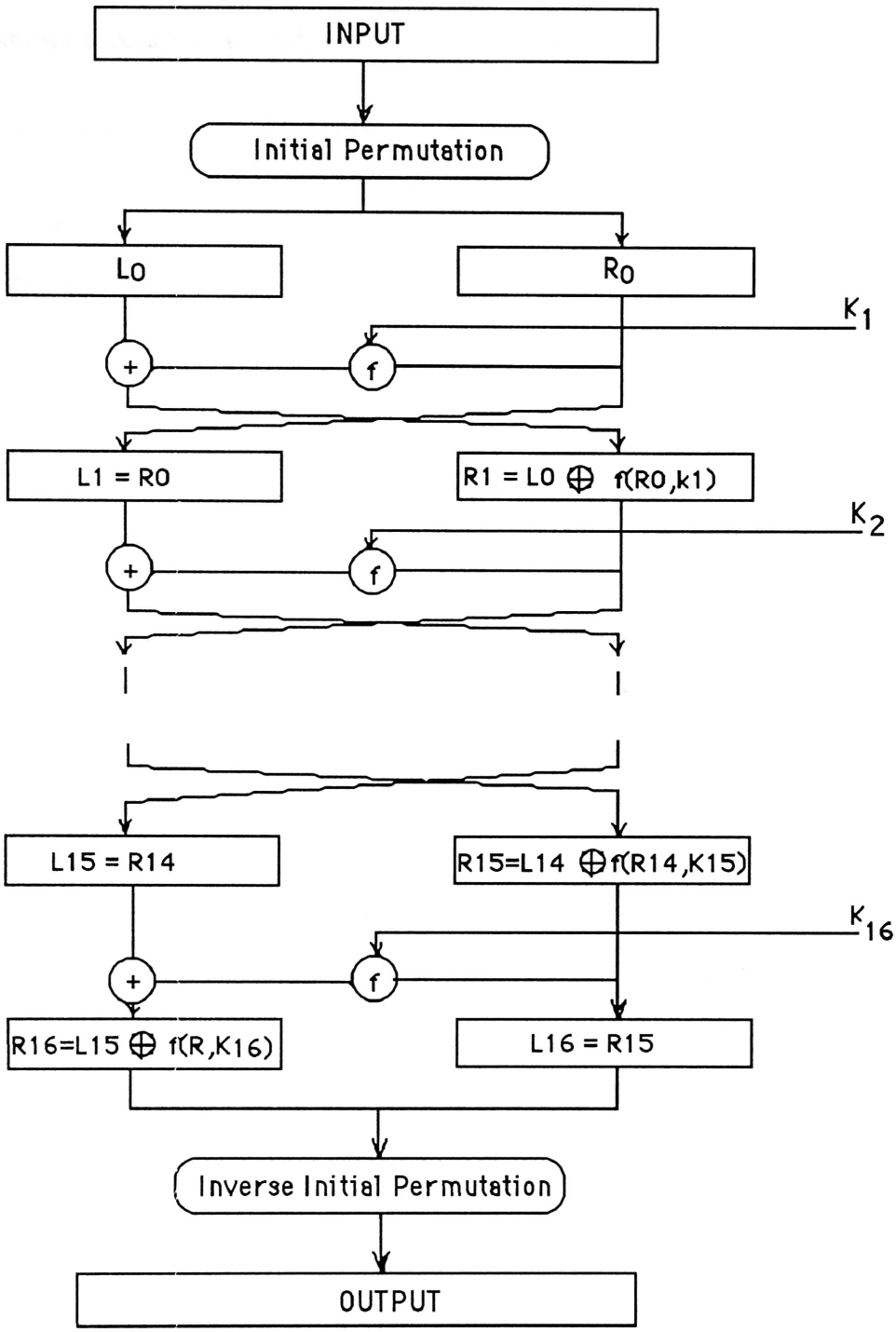


Figure 4. 3 DES Encipherment Computation

Where K_i is a 48-bit key. After the last iteration, the left and the right halves are not interchanged; instead the concatenated block $R_{16}L_{16}$ is input to the final permutation IP^{-1}.

Decipherment is performed by using the same algorithm, except that K_{16} is used in the first iteration, K_{15} in the second, and so on. The order of the key is reversed, the algorithm is itself not.

Figure 4.3 shows a sketch of the DES encipherment calculation

Figure 4.4 shows a sketch of the function $f(R_{i-1},K_i)$. First R_{i-1} is expanded to a 48-bit block $E(R_{i-1})$ using the bit-selection table E. Next, the exclusive-or of $E(R_{i-1})$ and K_i is calculated and the result broken into eight 6-bit blocks $B_1,.....,B_8$. Each 6-bit block is then used as an input to a selection function S box which returns a 4-bit block. The resulting 32-bit block is transposed by the permutation P.

A sketch of the calculation of cipher function f(R,K) is as follows:

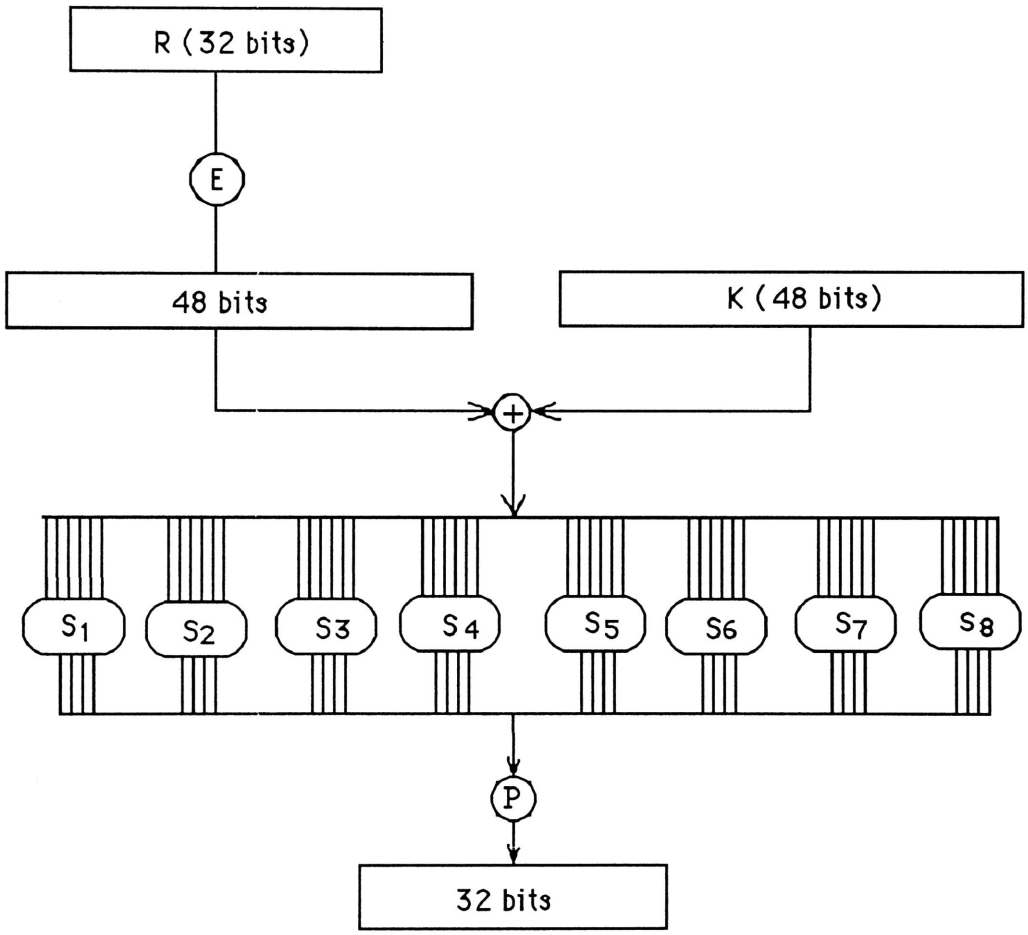


Figure 4.4 Calculation of f(R,K)

The S-boxes introduce nonlinearity into the DES. There are eight S-boxes in the DES, each of which is a set of four permutations on sixteen elements. S-boxes were not chosen at random, but there is no known cryptographic weakness resulting from these properties [Pfl89]. The full list of S-box design properties have never been released (they are classified by the US), although a number of research papers have discussed possible

properties interpolated from the published design, for example [Web85], [Bro89]. Some S-box properties can be partially described as follows:

- No S-box is a linear function of its input; that is, the four output bits cannot be expressed as a system of linear equations of the six input bits;

- Changing one bit in the input of an S-box results in changing at least two output bits; that is, the S-boxes diffuse their information well throughout their outputs;

- The S-boxes were chosen to minimize the difference between the number of 1s and 0s when any single input bit is held constant; that is, holding a single bit constant as 0 or 1 and changing the bits around it should not lead to disproportionately many 0s or 1s in the output.

Figure 4.5 shows a sketch of key schedule calculation. Each iteration i uses a different 48-bit key K_1 derived from the initial key K.

There are a set of DES keys - weak keys, semi-weak keys, and others reducing the cryptographic strength of the algorithm. They are should be avoided, although they represent a very small percentage of the total key numbers (2^{56}).

For a detailed description, all specific table values and weak keys of DES algorithm, the reader is referred to [NBS77], [Seb89], [Pfl89]. The past and future of DES are discussed in [Sim92].

A C program **des** is implemented by using the DES specification of the US National Bureau of Standards [NBS77]. The full source listing of the program is presented in the appendix section.

The processing time (in seconds) of the program des which is run to encrypt 100Kb block under Fujitsu Mainframe M-780, and Hewlett Packard Workstation for 8, 16, 24, 32 rounds and four modes in table 4.1.

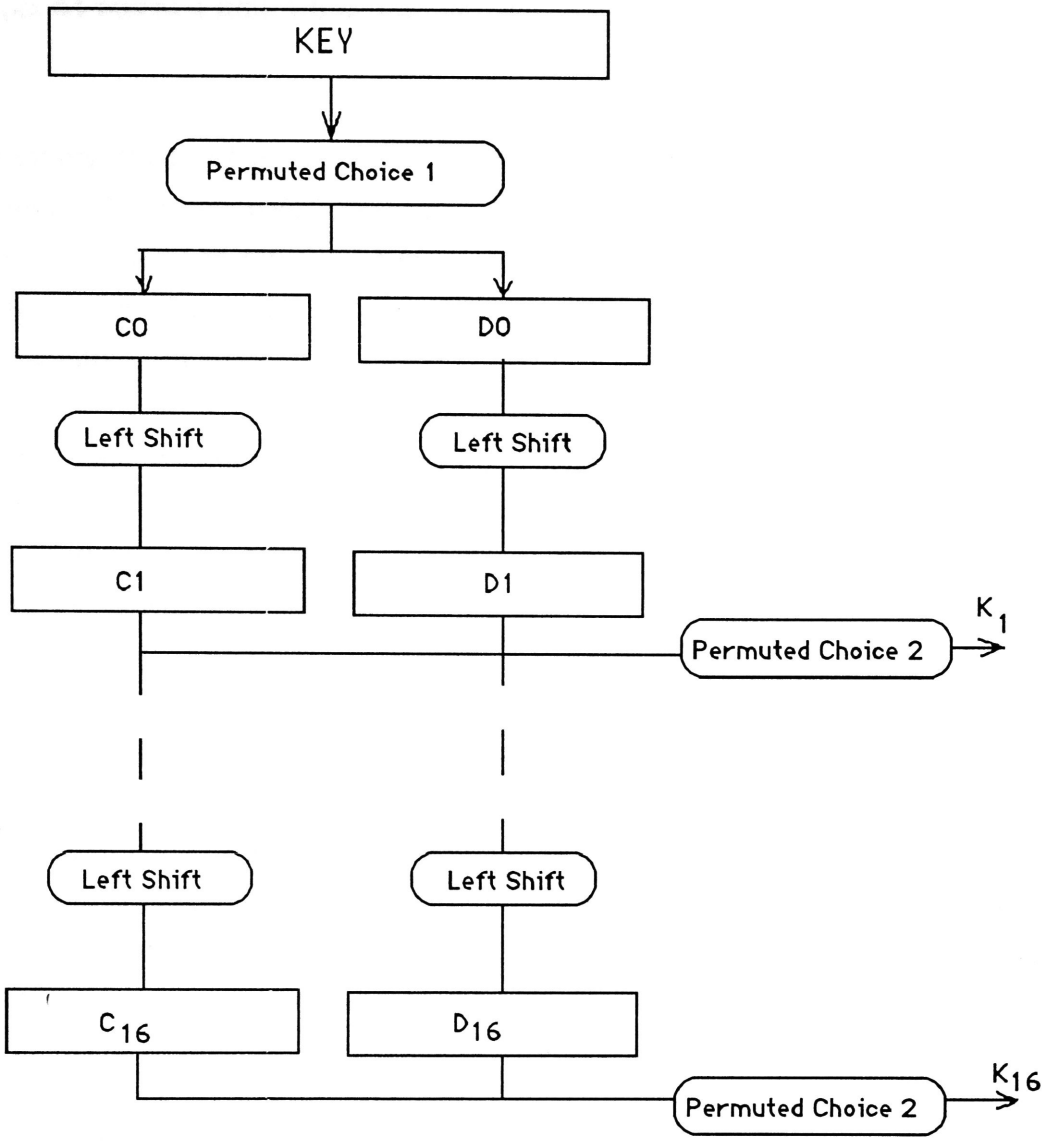


Figure 4.5 Key Schedule Calculation

Table 4.1 DES program performance

	Fujitsu				HP			
mode	ECB	CBC	CFB	OFB	ECB	CBC	CFB	OFB
round								
8	8.7	8.7	69.6	69.6	14.3	14.6	117.1	116.2
16	16.0	16.1	128.5	128.2	27.2	27.3	216.7	218.1
24	23.3	23.3	186.7	186.7	38.5	38.7	316.0	318.6
32	31.2	31.1	233.5	233.6	52.6	52.9	421.4	422.1

The results show that for the same number of rounds, ECB and CBC modes take less time to process than the CFB and OFM modes. Due to the fact that with ecb and cbc modes, des

encrypts 64 bits at a time while with cfb and ofb, des encrypts only k bits (less than 64 bits).

The running speed of program des is relatively slower than other symmetric cipher algorithms. This is due to that the program DES uses extensive table lookup. The extensive use of permutation in DES to spread information (diffusion) is expensive in software implementation.This is not to fault DES, it is one of the design objectives in DES for hardware implementation.

Each individual S-box in DES provides only 64 entries of 4 bits each, or 32 bytes per S-box. DES uses 8 S-boxes and looks up 8 different values in them simultaneously. While this is appropriate for hardware, it seems an unreasonable restriction to software with sequential program execution.

L.Brown, Australian Defense Force Academy, suggests that there are ways to improve the executing speed of DES. The techniques can be used include the use of pre-computed tables for many of the functions, incorporation of permutation P in DES into the S-box lookup tables, use of macro's and lookup tables in large permutations such as IP in DES, better functional decomposition, careful selection of internal data structures, and unravelling all inner loops. Some of these techniques are applied for the revised version of DES program, the results indicate that the speed can be improved by a factor of at least two.

4.2. The FEAL Cryptographic Primitive Family

FEAL is 64-bit symmetric block cipher algorithm acting on 64 bits of plaintext to produce a 64-bit ciphertext controlled by 64-bit key. But in contrast to DES, a software implementation does not require a table look-up. The processing speed is expected to be faster than DES.

Feal consists of two processing parts. One is the key schedule which generates the 256-bit extended key from the 64-bit secret key for 8-round FEAL (FEAL-8). It is designed to generate different extended keys for different secret keys. The other is the data randomizer which generates 64-bit ciphertext from 64-bit plain text under control of the extended key.

One program can perform two functions, encipherment and decipherment, except for the extended key entry.

FEAL working with no parity in a key block is safe from all-key attack because it is controlled by a 64-bit key, which is longer than the 56-bit DES key.

FEAL-4 and FEAL-8 have been expanded to FEAL-N (N round FEAL with 64-bit key), where FEAL-N with N is even and $N = 2^x$ is recommended). FEAL-N has also been expanded to FEAL-NX (X: key expansion, N round FEAL with 128-bit key) that accepts 128-bit keys. When the right half of the 128-bit key is all zeros, FEAL-NX works as FEAL-N.

A sketch of FEAL-NX data randomization is in figure 4.6 and calculation of extended key is in figure 4.7.

One of the building blocks of the FEAL cipher is a transformation S.

The S function in the f function is defined as:

S(x,y,delta) = ROT2(T);
T = x+y+delta mod 256
where x,y: one-byte data; delta: constant (0 or 1);
ROT2(T): 2-bit left rotation operation on T.

For a detailed description of the algorithm, the reader is referred to [Shi87] and [Miy90].

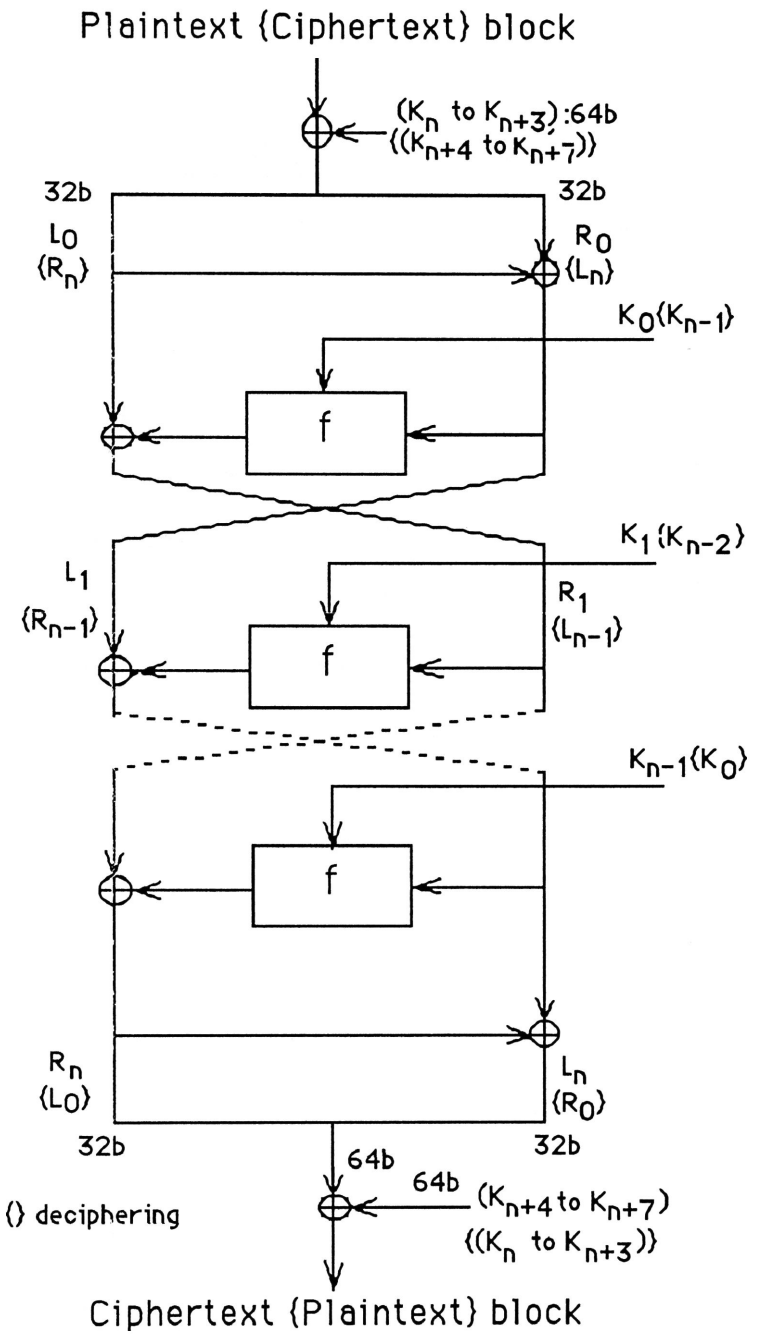


Figure 4.6 Feal Data Randomization

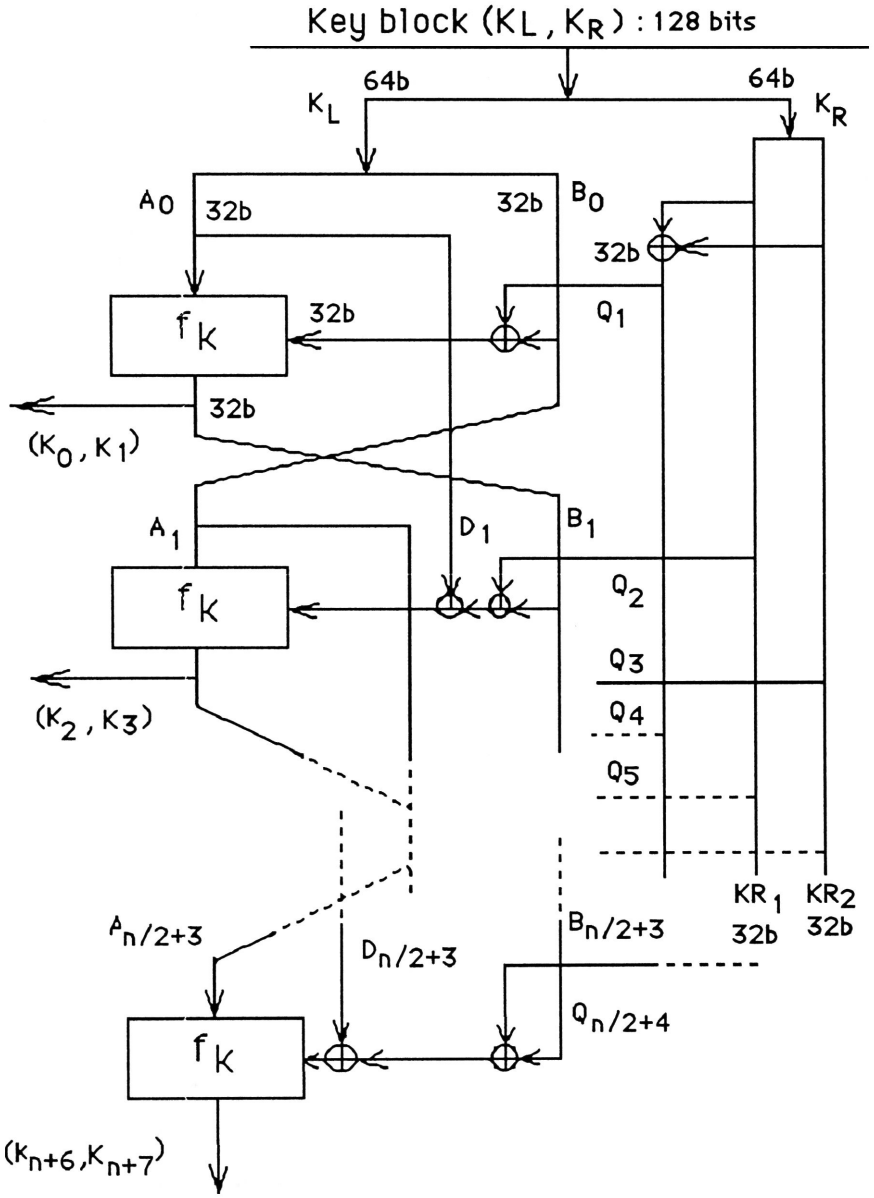


Figure 4.7 Key Schedule of FEAL (FEAL-NX)

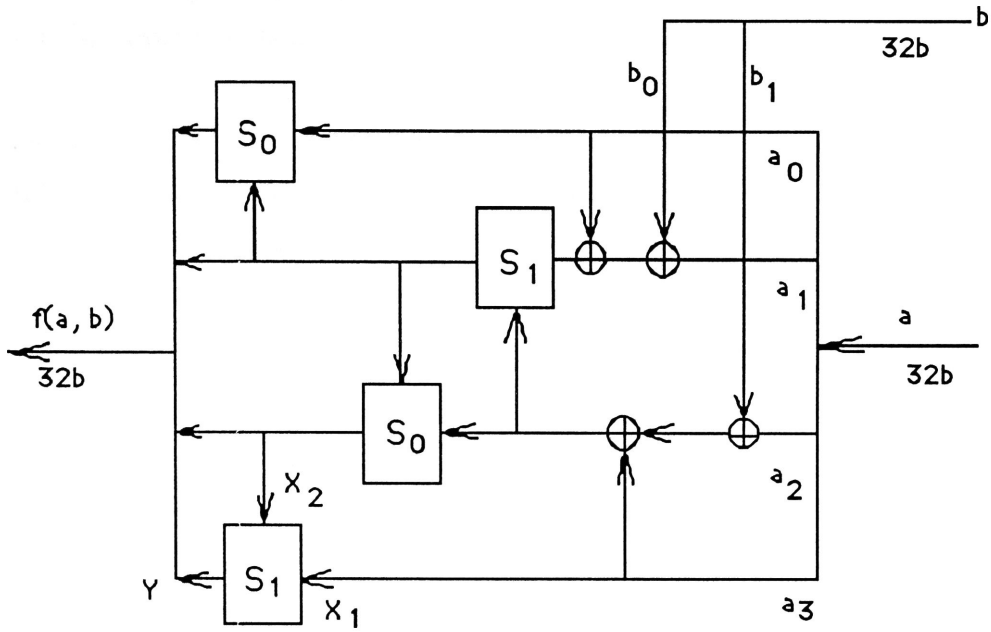


Figure 4.8 FEAL Function f

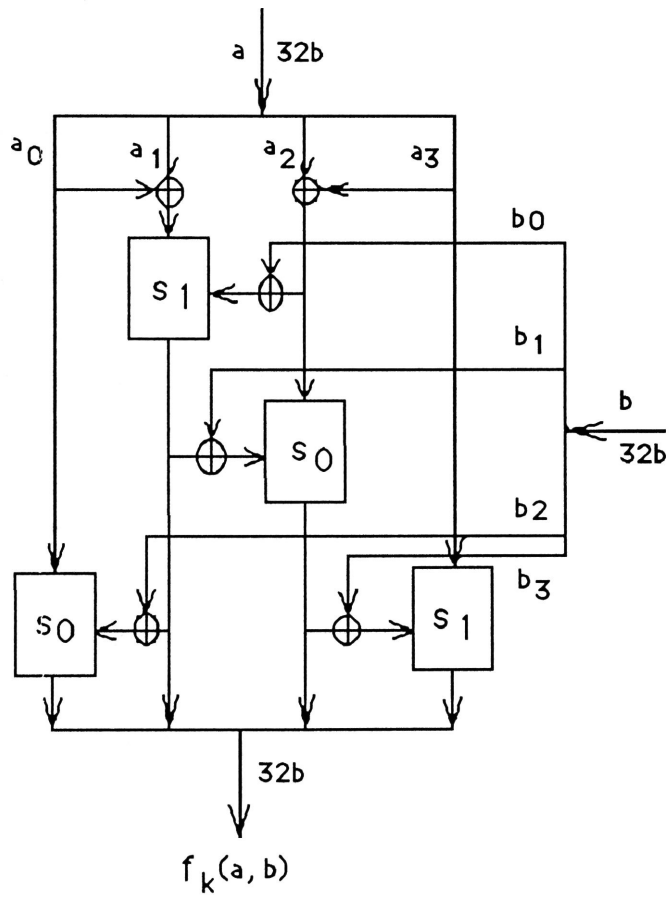


Figure 4.9 FEAL Function F_k

A C program **feal** is implemented by using algorithm proposed in [Shi87] and the extensions to the original design in [Miy90]. The full source listing of the program is presented in the appendix section.

The processing time (in seconds) of the program feal, which is run to encrypt 100Kb block under Fujitsu Mainframe M-780, and Hewlett Packard Workstation for 8, 16, 24, 32 rounds and four modes in table 4.2.

Table 4.2 Program feal performance

	Fujitsu				HP			
mode	ECB	CBC	CFB	OFB	ECB	CBC	CFB	OFB
round								
8	0.3	0.3	2.3	2.3	0.6	0.7	5.4	5.5
16	0.5	0.5	4.3	4.3	1.2	1.2	10.3	9.9
24	0.8	0.8	6.4	6.4	1.7	1.6	14.4	15.1
32	1.0	1.0	8.4	8.4	2.5	2.5	20.0	19.1

The results show that for the same number of rounds, ECB and CBC modes take less time to process than the CFB and OFM modes. And it runs faster than des by a factor of 30.

4.3. The LOKI Cryptographic Primitive

LOKI is another 64-bit encryption and decryption algorithm proposed by L.Brown, M.Kwan, J.Pieprzyk, and J.Seberry [Bro90], [Bro91]. The algorithm is designed to encrypt or decrypt blocks of data under the control of 64-bit key.

A block to be encrypted is added modulo 2 to the key, is then processed in 16 rounds of a complex key-dependent computation, and finally is added modulo 2 to the key again. The key-dependent computation can be defined in terms of a confusion-diffusion function f, and a key schedule KS. The same key is used for both encryption and decryption, but with the schedule of addressing the key bits altered so that the decryption process is the reverse of the encryption process.

The Loki uses one twelve-bit to eight-bit S box based on irreducible polynomials four times each rounds. The sketch of S-box calculation is in Figure 4.10.

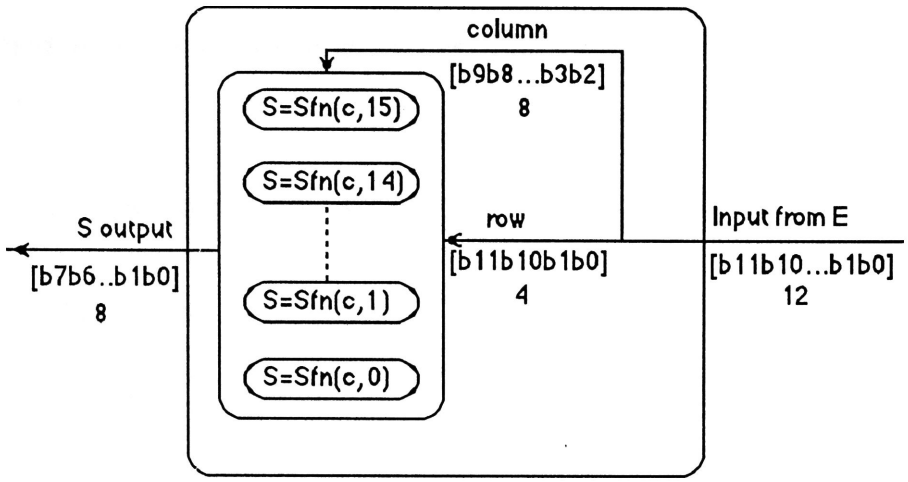


Figure 4.10 Loki S-box detail

The same structure with alternate substitution functions may be used to build private variants of the original algorithm.

The original version of LOKI is now designated LOKI89, the revised version has the following modifications:

- Change key schedule to interchange halves after every second round,
- Change key rotations to alternate between ROT13 and ROT12,
- Remove initial and final XORs of key with plaintext and ciphertext,
- Alter the S-box function.

The general description of the revised algorithm can be described as follows.

Encryption: The 64-bit input block X is partitioned into two 32-bit blocks L and R. Similarly, the 64 key is partitioned into two 32-bit blocks KL and LR.

The key-dependent computation consists of 16 rounds of a set of operations (except for a final interchanged of blocks). Each round is calculated as follows:

$$L_i = R_{i-1}$$
$$R_i = L_{i-1} \ (+) \ f(R_{i-1}, KL_i)$$
$$f(R_{i-1}, K_i) = P(S(E(R_{i-1} \ (+) \ K_i)))$$

The encryption function f is a concatenation of a modulo 2 addition and three functions E, S, and P, which takes as input the 32-bit right data half R_{i-1} and the 32-bit left key half KL_i, and produces a 32-bit result which is added modulo 2 to the left data half L_{i-1}.

The key schedule KS is responsible for deriving the sub-keys K_i, and is defined in the following way.

the 64-bit key K is partitioned into two 32-bit halves KL and KR.

For odd numbered rounds

$$K_i = KL_{i-1}$$
$$KL_i = ROL(KL_{i-1},13)$$
$$KR_i = KR_{i-1}$$

For even numbered rounds

$$K_i = KL_{i-1}$$
$$KL_i = ROL(KL_{i-1},12)$$
$$KR_i = KR_{i-1}$$

After the 16 rounds, the two output block halves L_{16} and R_{16} are then concatenated together to form the output block.

The decryption computation is identical to that used for encryption, save that the partial keys used as input to the function f in each round are calculated in reverse order.

For a detailed description of LOKI algorithms, the reader is referred to [Bro90] and [Bro91].

A C program **loki** is implemented by using the algorithm proposed in [Bro90] and [Bro91]. The full source listing of the program is presented in the appendix section.

The processing time (in seconds) of the program loki, which is run to encrypt 100Kb block under Fujitsu Mainframe M-780, and Hewlett Packard Workstation for 8, 16, 24, 32 rounds and four modes in table 4.3.

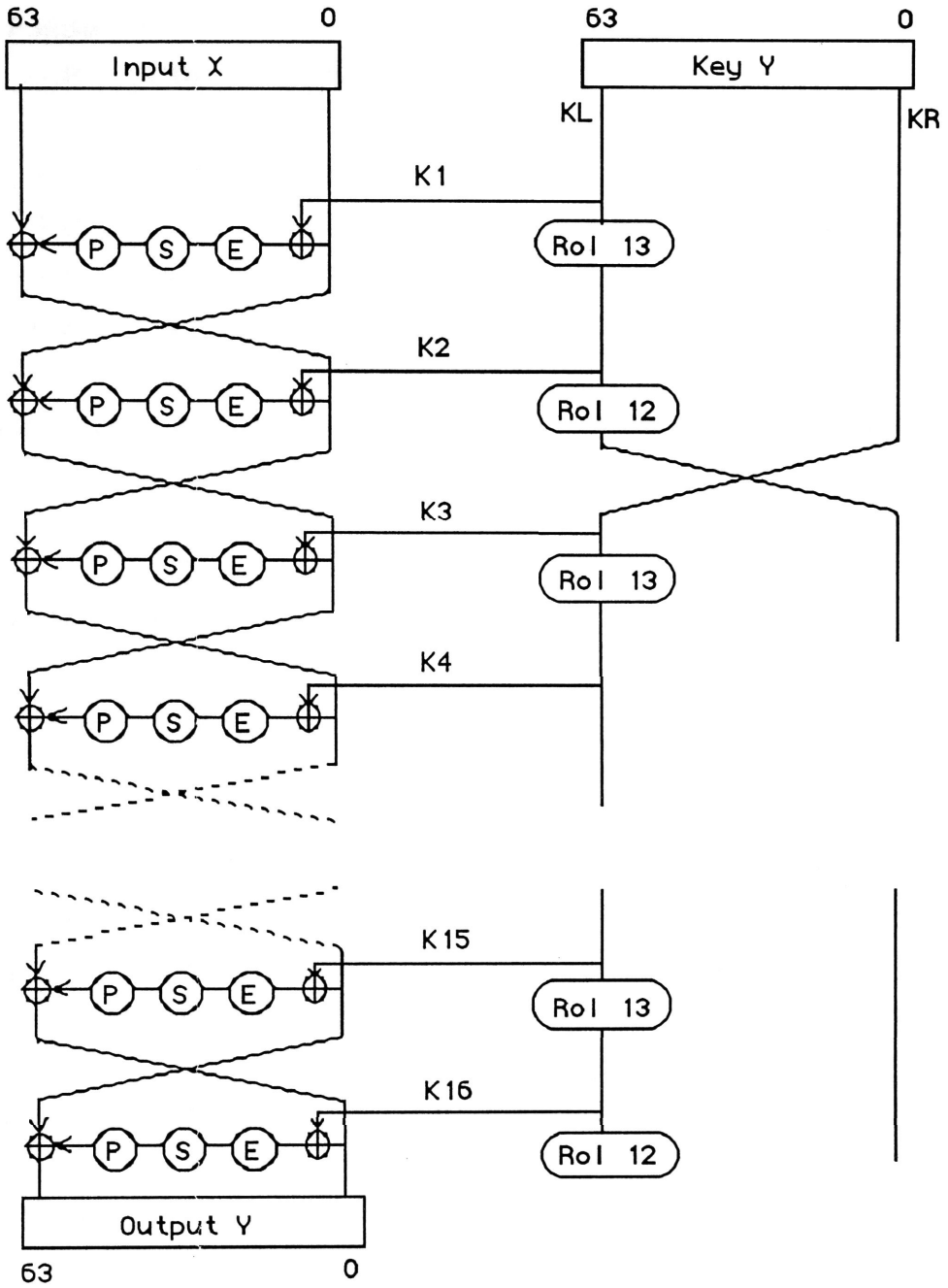


Figure 4.11 LOKI Overall Structure

Table 4.3 Program loki performance

	Fujitsu				HP			
mode	ECB	CBC	CFB	OFB	ECB	CBC	CFB	OFB
round								
8	1.5	1.5	12.4	12.4	2.8	2.8	22.1	23.2
16	3.1	3.1	24.6	24.6	5.7	5.8	44.1	43.6
24	4.6	4.6	36.9	37.0	8.4	8.5	68.7	68.8
32	6.2	6.2	49.2	49.1	10.8	11.2	90.8	89.4

The results show that for the same number of rounds, ECB and CBC modes take less time to process than the CFB and OFM modes.

5.4. The KHUFU Cryptographic Primitive

The KHUFU is another block cipher operating on 64-bit blocks proposed by R.C.Merkle [Mer90]. It is a multi-round encryption algorithm in which two 32-bit halves (called L and R) are used alternately in the computation. Each half is used as input to a function F, whose output is XORed with the other half. The number of rounds is a multiple of 8.

The 64-bit plaintext is first divided into two 32-bit words designated L and R. L is bytes 0 through 3, and R is bytes 4 through 7 of the 64-bit plaintext. L and R are then XORed with two 32-bit words of auxiliary key material. Then the main loop is started, in which byte 3 (the least significant byte) of L is used as the input to a 256-entry S-box. Each S box entry is 32 bits wide. The selected 32-bit entry is XORed with R. L is then rotated to bring a new byte into position, after which L and R are swapped. The S box itself is changed to a new S box after every 8 rounds. This means that the number of S-boxes required depends on the number of rounds of encryption used: one new S-box for every octet (8 rounds). Finally, after the main loop has been completed, we again XOR L and R with two new 32-bit auxiliary key values to produce the ciphertext.

The decryption algorithm is the reverse of the encryption algorithm.

The encryption of a single 64-bit plaintext by Khufu can be viewed algorithmically as follows:

```
L,R: int32
enough: integer;
SBoxes: ARRAY [1..enough/8] OF ARRAY [0..255] OF int32
AuxiliaryKeys: ARRAY[1..4] OF int32;
rotateSchedule: ARRAY[1..9] = [16,16,8,8,16,16,24,24]
octet: integer

L = L XOR AuxiliaryKeys[1]
R = R XOR AuxiliaryKeys[2]

octet = 1

FOR round=1 TO enough DO
  Begin
    R = R XOR SBoxes[octet][L AND #FF]
    L = RotateRight[L,rotateSchedule[(round-1) mod 8 + 1]
    SWAP[L,R]
    if (round mod 8 = 0) then octet = octet + 1
  End

L = L XOR AuxiliaryKeys[3]
R = R XOR AuxiliaryKeys[4]
```

The purpose of the rotation schedule is to bring new bytes into position so that all 8 bytes of input are used in the first 8 rounds. This means that a change in any single input bit is guaranteed to force the use of a different S-box entry with 8 rounds. And the secondary purpose of the rotation schedule is to restore the data to its original rotational position after 8 rounds.The same S-box entry will never be used twice.

The S-boxes are most of the key. So the computation of it is the most crucial component in ensuring the security of the algorithm. The essential idea is to generate the S-boxes in a pseudo-random fashion from a user supplied key so that they satisfy one property: all four of the one-byte columns in each S box must be permutations.

In practice, this procedure is done as follows: first, generate a stream of good pseudo-random bytes; second, use the stream of pseudo-random bytes to generate four pseudo-random permutations that map 8 bits to 8 bits. These four pseudo-random permutations are the generated S-box. We repeat this process until additional S-boxes are enough for encryption process.

The process of calculating S-box can be viewed algorithmically as follows:

```
FOR octet=1 TO enough/8 DO
  SBox = initialSBox
  FOR column = 0 TO 3 DO
    BEGIN
      FOR i=0 TO 255 DO
        BEGIN
        randomRow = RandomInRange[i,255]
        SwapBytes[ SBox[i,column], SBox[randomRow,column]];
        END
      SBoxes[octet] = SBox
    END
```

For a detailed description of the Khufu algorithm, the reader is referred to [Mer90].

A C program **khufu** is implemented by using the algorithm proposed in [Mer90]. The full source listing of the program is presented at the appendix section.

In the implementation of the program khufu, the initial S box is initialised with fixed values. The S box is computed by using the cipher chaining mode of the khufu algorithm itself to encrypt the initial S-box with a user-supplied key and a fixed initial vector V. This process ensures that the computed S-box is unique and dependent on the user key.

The processing time (in seconds) of the program khufu, which is run to encrypt 100Kb block under Fujitsu Mainframe M-780, and Hewlett Packard Workstation for 8, 16, 24, 32 rounds and four modes in table 4.4.

Table 4.4 Program khufu performance

	Fujitsu				HP			
mode	ECB	CBC	CFB	OFB	ECB	CBC	CFB	OFB
round								
8	0.1	0.1	0.8	0.8	0.2	0.2	1.9	2.0
16	0.2	0.2	1.5	1.5	0.4	0.4	3.3	3.3
24	0.3	0.3	2.1	2.1	0.5	0.6	4.8	4.5
32	0.4	0.4	2.7	2.7	0.6	0.7	5.7	6.1

The results show that for the same number of rounds, ECB and CBC modes take less time to process than the CFB and OFM modes.

4.5. The KHAFRE Cryptographic Primitive

The Khafre is another 64-bit encryption and decryption algorithm proposed by R.Merkle [Mer90], which is similar with the Khufu algorithm except that Khafre does not precompute its S-box. Instead, Khafre uses a set of standard S-boxes. The standard S boxes are different with the initial S boxes in Khufu.

The use of standard S-boxes means that Khafre can quickly encrypt a single 64-bit block without the lengthy computation used in Khufu.

The decryption algorithm is the reverse of the encryption algorithm.

The encryption of a single 64-bit plaintext by Khafre can be described algorithmically as follows:

```
L, R: int32
standardSBoxes: ARRAY[1..enough/8] OF ARRAY [0..255] OF int32
key: ARRAY[0..keySize-1] OF ARRAY [0..1] of int32
keyIndex: [0..keySize-1]
rotateSchedule: ARRAY[1..8] = [16,16,8,8,16,16,24,24]

L = L XOR key[0][0]
R = R XOR key[0][1]
keyIndex = 1 MOD keySize
```

```
octet = 1

FOR round = 1 TO enough DO
  BEGIN
  L = L XOR standardSBoxes[octet][R AND #FF]
  R = RotateRight[R,rotateSchedule[round mod 8 + 1]]
  SWAP[L,R]
  IF round MOD 8 = 0 THEN
    BEGIN
      L = L XOR rotateRight[key[keyIndex][0],octet]
      R = R XOR rotateRight[key[keyIndex][1],octet]
      keyIndex = keyIndex + 1
      IF keyIndex = keySize THEN keyIndex = 0
      octet = octet + 1
    END
  END
```

For a detailed description of the Khafre algorithm, the reader is referred to [Mer90].

In the program khrafre, the standard S-box is computed by initialising it with fixed values. According to the khafre encryption algorithm, this is sufficient because the standard S-box can not serve as the key, the key mixing is computed by exclusive-oring the key with the 64-bit data block before the first round and thereafter following every 8 rounds.

A C program **khafre** is implemented by using the algorithm proposed in [Mer90]. The full source listing of C program is presented in the appendix section.

The processing time (in seconds) of the program khafre, which is run to encrypt 100Kb block under Fujitsu Mainframe M-780, and Hewlett Packard Workstation for 8, 16, 24, 32 rounds and four modes table 4.5.

Table 4.5 Program khafre performance

	Fujitsu				HP			
mode	ECB	CBC	CFB	OFB	ECB	CBC	CFB	OFB
round								
8	0.1	0.1	0.8	0.8	0.2	0.2	2.0	2.0
16	0.2	0.2	1.5	1.5	0.4	0.4	3.4	3.5
24	0.3	0.3	2.2	2.2	0.6	0.6	5.1	5.2
32	0.4	0.4	2.8	2.9	0.8	0.8	6.7	6.7

The results show that for the same number of rounds, ECB and CBC modes take less time to process than the CFB and OFM modes.

5. Block Cipher Cryptanalysis

Any discussion on symmetric block cryptosystems is not complete without mentioning cryptanalysis attacks against the cryptosystems. There are several types of attacks which can be used to determine the security level of a cryptosystem: ciphertext-only attack, known-plaintext attack, chosen plaintext attack.

A ciphertext-only attack is where a system is attempted to be compromised by examining encrypted messages, or cipher, and referring to related secondary information. Any system whose security cannot weather a ciphertext-only attack is considered inadequate and totally insecure [Seb89].

A known-plaintext attack is when a system is attempted to be compromised with the cryptanalyst being in possession of plaintext and its corresponding ciphertext. If a system can withstand a known-plaintext attack, this is taken as a reasonable indication that the system is secure. In 1977 the NBS accepted the DES system on the basis of it resisting a known plaintext attack.

A chosen-plaintext attack is when a system is attempted to be compromised with unlimited amount of plaintext and its corresponding ciphertext chosen by the cryptanalyst.

The common attacks against DES require $2^{56}/2$ encryptions [Sim92]. Although there has been no success against the full DES algorithm, there has been cryptanalysis success in breaking one of the proposed modes of operation of DES. In output feedback mode (ofb), DES is used to generate a pseudorandom number, which is then used as one-time pad to encrypt the message. If the number of shift-left bits is less than 64, the expected cycle size for a random function on N elements is only about $N^{1/2}$. Therefore only the number of shift-left bits is equal to 64 should be considered secure for ofb mode [Dav82].

The most powerful tool for analysing block ciphers currently known is differential cryptanalysis [Bih91a], [Bro91]. It has been used to find design deficiencies in many of the new ciphers. Differential cryptanalysis is a dynamic attack against a cipher and it belongs to the chosen plaintext attack type. In general, differential cryptanalysis is much faster than exhaustive search for a certain number of rounds in the cipher, however there is a break-even point where it becomes slower than exhaustive search. The differential cryptanalysis technique uses a very large number of chosen plaintext pairs, which through a statistical analysis of the resulting ciphertext pairs, can be used to determine the key in use. By using this technique Biham and Shamir [Bih91a] showed that DES with six rounds was broken in less than 0.3 seconds on a personal computer using 240 ciphertext.

DES with 15 rounds can be broken in about 2^{52} steps but DES with 16 rounds still requires 2^{58} steps (which is slightly higher than the exhaustive search - 2^{56}). At Crypto'92, Biham and Shamir presented a new variant of differential cryptanalysis that is capable of breaking a 16-round DES significantly faster than by exhaustive search.

Using a chosen plaintext attack with 100-10000 encryptions, FEAL-4 was broken by Den-Boer [Boe88]. And by using differential cryptanalysis, FEAL-8 can be broken with less than 2000 chosen plaintexts, and to break FEAL-N for N less than 32 with fewer chosen plaintexts than the number of encryptions needed in an exhaustive search [Sim92].

The original Loki encryption algorithm, LOKI89, with up to eleven rounds is breakable faster than exhaustive search by differential cryptanalysis attacks, either chosen plaintext or known plaintext attacks. Biham and Shamir also showed that every key of Loki has 15 equivalent keys due to a key complementation property and thus the complexity of a known plaintext attack on the full 16-round version can be reduced to 2^{60}. An alternate attack using a 3 round characteristic that can break up to 14 rounds of LOKI89 was found independently by Knudsen and Kwan. Up to 10 rounds of LOKI89 can be broken using a 2 round characteristic, and up to 12 rounds using a 3 round characteristic [Knu91].

Khafre with 16 rounds is breakable by a differential cryptanalysis chosen plaintext attack using about 1500 encryptions within about an hour on a personal computer. By a differential cryptanalysis known plaintext attack it is breakable using about 2^{38} encryptions. Khafre with 24 rounds is breakable by a chosen plaintext attack using about 2^{53} encryptions and using a differential known plaintext attack it is breakable using about 2^{59} encryptions.

Differential cryptanalysis techniques are also applicable to hash functions, in addition to cryptosystems.

6. Block Cipher-Based Hash Functions

The simplest way to obtain the digital signature of a message is to apply asymmetric cryptography to the whole message. But the public key systems generally encrypt more slowly than conventional ciphers such as DES, FEAL, etc. And it is usually undesirable to apply digital signature techniques direct to a long message because it doubles the amount of intormation to be transmitted or stored. But on the other hand, the requirements that the entire message must be signed. This is seemingly contradictory, but a heuristic solution can be obtained by using a hash function as an intermediary. A hash function accepts a variable size message M as input and outputs a fixed size representation H(M) of M, sometimes called a message digest. In general, H(M) is much smaller than M. A digital signature may be applied to H(M) directly and processing time is relatively quick. A hash function can also serve to detect modification of a message, independently from any signature schemes. That is, it can serve as a cryptographic checksum or message authentication code (MAC). The general scheme can be describe in Figure 6.1.

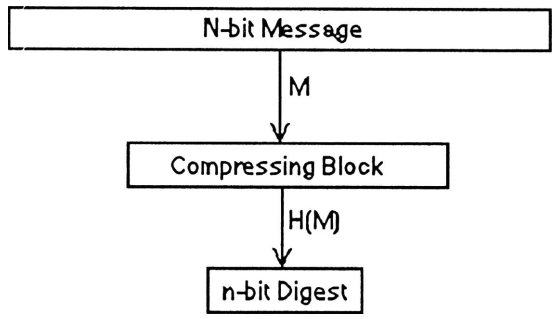


Figure 6.1 Using compress block to obtain n-bit message digest

A one-way function is defined as a function F such that for every x in the domain of F, F(x) is easy to compute; but for virtually all y in the range of F, it is computationally infeasible to find an x such that y = F(x). This is not a precise mathematical definition.

Merkle [Mer82] defines a hash function f to be a transformation with the following properties:

1. F can be applied to an argument of any size,
2. F produces a fixed size output,
3. F(x) is relatively easy to compute for any given x,
4. For any given y it is computationally infeasible to find x with F(x) = y,
5. For any fixed x it is computationally infeasible to find x' (not equal to x) with F(x') = F(x).

The security of the hash function may be classified further as weak and strong hash functions. A function satisfying 1-5 is called a weak hash function. On the other hand, a hash function is called a strong hash function when modifying the condition 5 to that it is computationally infeasible to find any {x1, x2} with H(x1) = H(x2).

The use of one-way hash functions to produce message digests is very different from use of trapdoor one-way functions to generate encryption functions in an asymmetric cipher. In asymmetric cryptosystems, the condition 4 becomes that it is computationally infeasible for anyone to find plaintext from ciphertext except intended receiver who has the private key to decrypt the ciphertext.

To be useful in conjunction with public key systems, a hash function should ideally be keyless. This is in contradistinction to message authentication codes used in connection with secret key systems.

Many proposals have been made for one-way hash functions based on block ciphers. Hash functions based on symmetric block cryptosystems are examples of key-hash function. A family of message digest md4 [Riv90], md5 [Riv91], and haval [Zhe92] are examples of keyless hash functions.

6.1. Cipher block chaining hashing

The most obvious way of using a block cipher is to construct a message digest is based on the standard mode of use for a block cipher called cipher block chaining (cbc). This is an example of key hashed function. In this mode of use the message is divided into a sequence of n-bit blocks, M_1, M_2,...,M_k. The sequence C_i is then derived, $C_i = E_k(C_{i-1}$ XOR $M_i)$. Then, the derived message digest is C_k. In order to protect messages against their digests being forged using the birthday attack, a scheme (Figure 6.2) is used [Seb89]. In this case, every key input is the same, namely K. It can be considered an initial condition of the system. The sequence of successive cryptograms generated in the system is described as follows:

$C_1 = E_k(M_1)$

$C_2 = E_k(M_2 \text{ XOR } C_1)$

.

$C_k = E_k(M_k \text{ XOR } C_{k-1})$

$H_k(M) = E_k(M_1 \text{ XOR } M_2 \text{ XOR } ....\text{XOR } M_k \text{ XOR } C_k)$

The sketch of the algorithm is as follows:

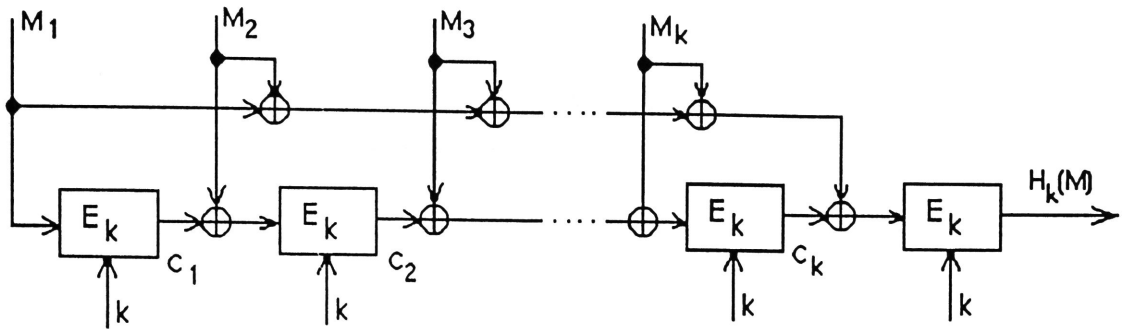


Figure 6.2 Compressing system with a single round

6.2. Single block hash (SBH) mode

The SBH mode is proposed by L.Brown, J.Pieprzyk, and J.Seberry in [Bro90]. Data for which a hash is to be computed is divided into 64-bit blocks, the final block being padded with nulls if required. A 64-bit key is supplied, and is used as the initial hash value IV. For each message block M_i: that block is added modulo 2 to the previous hash value to form a key. That key is used to encrypt the previous hash value. The encrypted value is added modulo 2 to the previous hash value to form the new hash value. The SBH code is the final hash value.

The SBH process may be described as:

$$H_0 = IV$$
$$H_i = \text{Encrypt}_{M_i \text{ XOR } H_{i-1}}(H_{i-1}) \text{ XOR } H_{i-1}$$

$$SBH = H_n$$

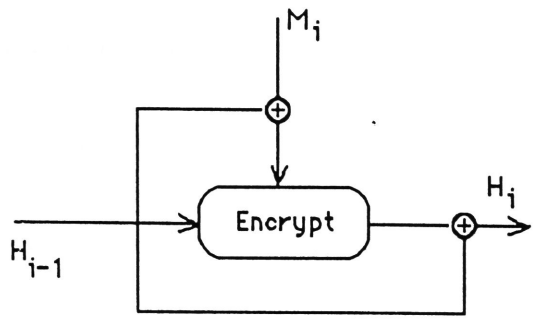


Figure 6.3 Single Block Hashing (SBH)

A C program **sbh** is implemented by using the above schemes (cbc and sbh modes) with five 64-bit block cipher algorithms (des, feal, loki, khufu, and khafre). The full source listing of the program is presented in the appendix section.

6.3. Two-n bit hash function using n-bit block cipher

This hash function has been proposed by J.Quisquater and M.Girault [Qui89] as a means to provides 2n-bit results (double block hash), using any n-bit symmetric block cipher algorithms. The function can be considered as an extension of an already known one, which only provided n-bit hash results.

The following is the general description of algorithm:

Let e be a symmetric block-cipher algorithm, whose block length is n and key length is k (n=64 and k=56 for DES). We denote the encipherment of input X under key K by $E_k(X)$ (Figure 6.4). Let I and J be two n-bit initializing values. Then, the message digest H of a binary message M is calculated in four steps.

Step 1 (splitting):

M is split into k-bit blocks M_1, M_2, ...

Step 2

If the number of blocks is odd, a supplementary block filled with 0's is added. Let n = 2m be the number of blocks at the end of this step.

Step 3

Two supplementary blocks are added to the message. The first one, M_{n+1}, is equal to the XOR of all the preceding blocks:

$$M_{n+1} = M_1 + M_2 + ... + M_n$$

The second one, M_{n+2}, is equal to the addition modulo $2^k - 1$ of the same blocks:

$$M_{n+2} = M_1 + M_2 + ... + M_n \quad \text{modulo } 2^k - 1$$

Step 4

The output values $H_1, H_2,.....,H_{n+1}, H_{n+2}$ are calculated in the following iterative way

for i from 0 to m
do
$H_{-1} = I$
$H_0 = J$
$H^1_{(2i+1)} = E_{2i+1}(H_{2i-1})$ XOR H_{2i}
$H^1_{(2i+2)} = E_{2i+2}(H^1_{2i+1})$ XOR H_{2i}
$H_{2i+1} = H^1_{2i+2}$ XOR H_{2i}
$H_{2i+2} = H^1_{2i+1}$ XOR H_{2i-1}
done

The hashing value is formed by concatenating the final two hash values: $DBH = H_{n-1} \| H_n$

The sketch of the algorithm is as follows:

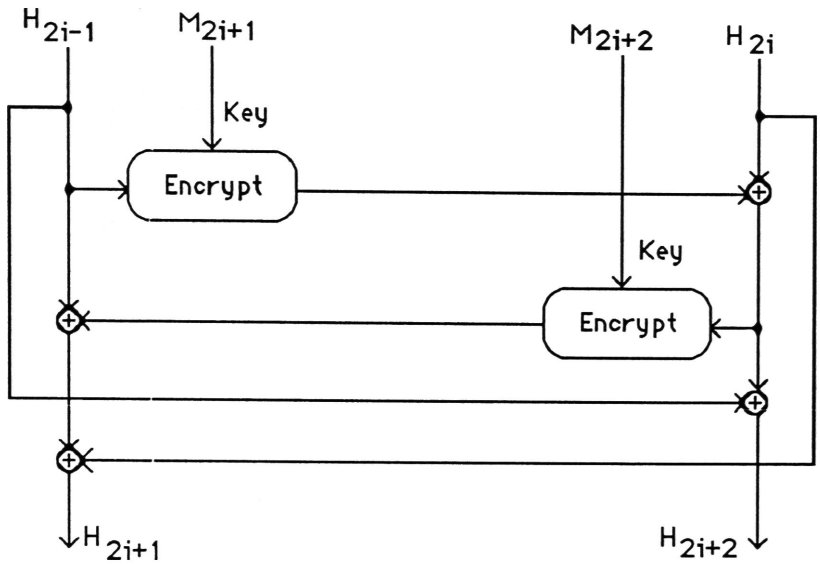


Figure 6.4 Two-n bit hash function using n-bit symmetric block cipher algorithm

For a detailed description of the algorithm, the reader is referred to [Qui89].

6.4. Double block hashing (DBH)

The double block hash, proposed by L.Brown, J.Pieprzyk, and J.Seberry [Bro90] is a variant of the two-n bit hash in the above section. Data for which a hash is to be computed is divided into pairs of 64-bit blocks M_{2i+1}, M_{2i+2}, the final block being padded with nulls if required. A 128-bit key is supplied, composed of two 64-bit blocks, which are used as the initial hash value IV_{-1}, IV_0.

$$H_{-1} = IV_{-1}$$
$$H_0 = IV_0$$

For each pair of message blocks M_{2i+1} M_{2i+2}, the following calculation is performed:

$$T = \text{Encrypt}_{M_{2i+1} \text{ XOR } H_{2i-1}}(H_{2i-1} \text{ XOR } M_{2i+2}) \text{ XOR } M_{2i+2} \text{ XOR } H_{2i}$$

$$H_{2i+1} = \text{Encrypt}_{M_{2i+2} \text{ XOR } H_{2i}}(T \text{ XOR } M_{2i+1}) \text{ XOR } M_{2i+1} \text{ XOR } H_{2i-1} \text{ XOR } H_{2i}$$

$$H_{2i+2} = T \text{ XOR } H_{2i-1}$$

The DBH is formed by concatenating the final two hash values: $DBH = H_{n-1} \| H_n$

The sketch of the algorithm is as follows:

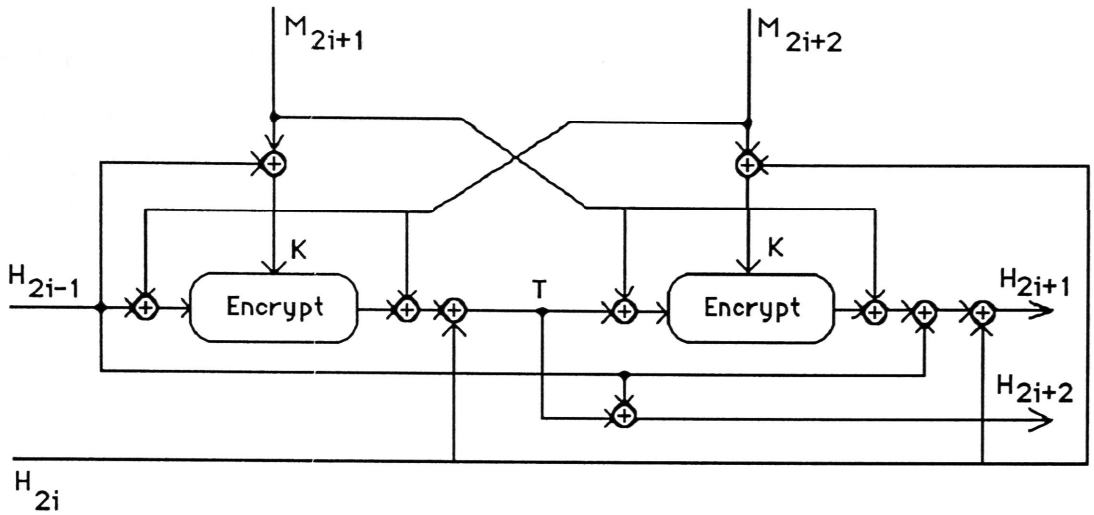


Figure 6.5 Double Block Hashing (DBH)

This scheme is an extended version of Quisquater and Girault [Qui89] in the section 6.4, by using addition modulo 2 of the previous hash value to the current message block before using it as key input to the encryption primitive.

A C program **dbh** is implemented by applying the above algorithm. The program produces 128-bit message digest from an arbitrarily long message, and it uses a known 64-bit block cipher encryption algorithm, such as des, feal, loki, khufu, or khafre (by selecting the appropriate option). The full source listing of the program is presented in the appendix section.

7. Keyless Hashing Algorithms

7.1. Message digest algorithms - MD4, MD4X and MD5

The MD4 and MD5 Message Digest Algorithms, by R.Rivest and RSA Data Security, Inc., are intended for file hashing and they do not require a key. Although this can be modified to accept a key and initialize the four long words (processing buffers) with the key values: The algorithms accept arbitrarily large inputs and produce an output of 128 bits.

Both algorithms are based on complex nonlinear functions which are difficult to invert in practice, similar to the philosophy used to design DES-like ciphers. But the formal proof of the security of the two hash functions have not yet been established.

Below is a general description of the MD4 message digest algorithm is given. For a detailed description, the reader is referred to [Riv90].

The processing of MD4 consists of four steps:

Step 1

Append padding bits and the message length: a single "1" bit, (l-1) zero bits (l is between 1 and 512) and 64-bit representation of b mod 2^{64} are appended to the message such that the length b + l + 64 of the extended message is a multiple of 512. The 64 bits containing the message length are appended as two 32-bit words, low-order word first. The new message is now represented as a sequence M[0], M[1],....,M[n-1] of n words, where n is a multiple of 16.

Step 2

Initialize a 4-word buffer (A,B,C,D) in hexadecimal with the following values.
A=67452301; B=EFCDAB89; C=98BADCFE; D=10325476

Step 3

Process the message in 16-word blocks:

```
for i=0 to (n/16)-1 do
  begin
    for j=0 to 15 do
      X[j] = M[16i+j];
    (AA,BB,CC,DD) = (A,B,C,D)
    [Round 1]
```

Let [A B C D i s] denote the operation A = (A + f(B,C,D) + X[i]) << s, where f(X,Y,Z) = ((X and Y) or ((not X) and Z)).
Do the following 16 operations:

[A B C D 0 3]
[D A B C 1 7]
[C D A B 2 11]
[B C D A 3 19]
[A B C D 4 3]
[D A B C 5 7]
[C D A B 6 11]
[B C D A 7 19]
[A B C D 8 3]
[D A B C 9 7]
[C D A B 10 11]
[B C D A 11 19]
[A B C D 12 3]
[D A B C 13 7]
[C D A B 14 11]
[B C D A 15 19]

[Round 2]
Let [A B C D i s] denote the operation A = (A + g(B,C,D) + X[i] + 5a827999) << s, where g(X,Y,Z) = ((X and Y) or (X and Z) or (Y and Z)).
Do the following 16 operations:

[A B C D 0 3]
[D A B C 4 5]
[C D A B 8 9]
[B C D A 12 13]
[A B C D 1 3]
[D A B C 5 5]
[C D A B 9 9]
[B C D A 13 13]
[A B C D 2 3]
[D A B C 6 5]
[C D A B 10 9]
[B C D A 14 13]
[A B C D 3 3]
[D A B C 7 5]
[C D A B 11 9]

[B C D A 15 13]

[Round 3]

Let [A B C D i s] denote the operation A = (A + h(B,C,D) + X[i] + 6ed9eba1) << s, where h(X,Y,Z) = (X xor Y xor Z).

Do the following 16 operations:

[A B C D 0 3]
[D A B C 8 9]
[C D A B 4 11]
[B C D A 12 15]
[A B C D 2 3]
[D A B C 10 9]
[C D A B 6 11]
[B C D A 14 15]
[A B C D 1 3]
[D A B C 9 9]
[C D A B 5 11]
[B C D A 13 15]
[A B C D 3 3]
[D A B C 11 9]
[C D A B 7 11]
[B C D A 15 15]

Then perform the following additions:

A = A + AA;
B = B + BB;
C = C + CC;
D = D + DD;
end;

Step 4

The output is the 4-word buffer (A,B,C,D)

The extension of md4: the basic MD4 algorithm is extended to produce 256-bit digest message. Rivest suggested [Riv90] the use of two copies of MD4 are run in parallel over the input. The first copy is standard as above. The second copy is modified as follows:

The initial state of the second copy is 33221100, 77665544, bbaa9988, and ffeeddcc. After every 16-word block is processed (including the last block), the values of the A registers in the two copies are exchanged. The final message digest is obtained by appending the result of the second copy of MD4 to the end of the result of the first copy of MD4.

The MD5 message-digest algorithm is a strengthened version of MD4 [Riv91]. It has four rounds instead of three, and incorporates other features: each step having a unique additive constant, the function g in round 2 was changed to make it less symmetric, each step adding the result of the previous step to promote a faster avalanche effect, the order of input words are accessed in round 2 and round 3 is changed.

For a detailed description of MD5, the reader is referred to [Riv91].

Three C programs **md4**, **md4x**, and **md5** are implemented based on the algorithms proposed in [Riv90] and [Riv91]. The program md4x is the extended version of md4 to produce 256-bit message digest. The full source listings of the programs are presented in the appendix section.

The processing time (in seconds) of the program md4, which is run to produce message digest for 10Mb block under Fujitsu Mainframe M-780, and Hewlett Packard Workstation are 14.3 secs and 80.3 secs respectively.

The processing time (in seconds) of the program md4x, which is run to produce message digest for 10Mb block under Fujitsu Mainframe M-780, and Hewlett Packard Workstation are 22.4 secs and 150.5 secs respectively.

The processing time (in seconds) of the program md5, which is run to produce message digest for 10Mb block under Fujitsu Mainframe M-780, and Hewlett Packard Workstation are 2.0 secs and 10.5 secs respectively.

7.2. HAVAL - A one-way hashing algorithm with variable length of output

The one-way hashing algorithm HAVAL, by Y.Zheng, Centre for Computer Security Research, University of Wollongong, is a deterministic algorithm that compresses an arbitrary long message into a message digest of specified length. The output value represents the digest or fingerprint of the message. HAVAL compresses a message of arbitrary length into a digest of 128 bits, 160 bits, 192 bits, 224 bits or 256 bits. In addition, HAVAL has a parameter that controls the number of passes a message block (of 1024 bits) is to be processed. A message block can be processed in two, three or four passes, the minimum number of passes to be processed is 2.

The algorithm makes use of different Boolean function with each pass operation. The Boolean functions have very nice properties [Zhe92], which are

- 0-1 balanced,
- High non-linearity,
- Satisfying the Strict Avalanche Criterion,
- Linearly inequivalent in structure, and
- Output-uncorrelated.

The sketch of the algorithm can be summarised as follows. The full details of the algorithm can be found in [Zeh92].

HAVAL processes a message M in the following three steps:

Step 1:

Pad the message M so that its length becomes a multiple of 1024. The last (or the most significant) block of the padded message indicates the length of the original (unpadded) message M, the required length of the digest of M, the number of passes each block is processed and the version number of HAVAL.

Step 2

Calculate repeatedly D_{i+1} - $H(D_i,B_i)$ for i from 0 to n-1, where D_0 is a 8-word (256-bit) constant string and n is the total number of blocks in the padded message.

Step 3

Adjust the 256-bit value Dn obtained in the above calculation according to the digest length specified in the last block B_{n-1}, and output the adjusted value as the digest of the message M.

For justification of the design and the boolean functions, the reader is referred to [Zhe92].

A C program **haval** is implemented by using the algorithm proposed in [Zhe92]. The full source listing of the program is presented in the appendix section.

The processing time (in seconds) of the program md4, which is run to produce message digest for 10Mb block under Fujitsu Mainframe M-780, and Hewlett Packard Workstation is as follows

Fujitsu Digest Pass	128 bits	160 bits	192 bits	224 bits	256 bits
2	4.8	4.8	4.8	4.8	4.8
3	7.0	7.0	7.0	7.0	7.0
4	9.6	9.6	9.6	9.6	9.6

HP Digest Pass	128 bits	160 bits	192 bits	224 bits	256 bits
2	19.5	19.1	18.9	19.0	19.0
3	28.1	28.0	28.2	28.2	28.2
4	38.6	38.7	38.8	39.2	38.9

The results show that for a given number of passes (2, 3, or 4 passes), the processing time for different message digests (128, 160, 192, 224, or 256 bits) varies insignificantly.

8. UNIX Style Manual Pages

The followings are UNIX manual pages for all command implemented with the algorithms described in the previous sections.

Commands are presented with UNIX-style convention for documentation. A manual page starting with a number of (1) is the command can be entered at the shell prompt, and a manual page starting with a number of (3c) is the C library subroutine which provides a low level access to a cryptographic primitive.

The manual pages are also presented with illustrated examples and other related information.

des(1)

NAME

des - data encryption standard

SYNOPSIS

des [[-e|-d] -k key -mode -r round -f file | -t]

DESCRIPTION

DES is the official encryption standard published by the US National Bureau of Standards, 1977. DES is symmetric block cipherment algorithm acting on 64 bits of plaintext to produce a 64-bit ciphertext controlled by 64 bit key. Deciphering is done by using the same key as for enciphering. DES operates with four modes: electronic code book (ecb), cipher block chaining (cbc), cipher feedback (cfb), and output feedback (ofb). The cbc mode is the default mode and it is recommended for encryption for increased security. The cfb and ofb modes have the same level of security with cbc mode, but the running time is a bit slower.

The standard version of DES run with 8 rounds and 64-bit key (8 characters). The extensions to DES have been implemented with this version. The number of rounds has been increased to 32.

Options:

-e encryption; the default is encryption;

-d decryption;

-k user-supplied key; if the key is not supplied with the command line, user will be asked for the key interactively; the minimum number of characters for the key is 5.

-m mode, valid modes are ecb, cbc, cfb, ofb; cipher block chaining is the default mode of operation;

-r round, the number of rounds used in enciphering or deciphering a block of data, valid values from 4 to 32 rounds. The more number of rounds of the enciphering, the higher security of the cipher data; the default value is 8;

-f input file, if this option is used, the resulting encryption file will be written to the same name with an extension .e;

-t time trial testing for algorithm running speed.

Since the key is an argument to the des command, it is potentially visible to users executing ps(1) or a derivative. It is advisable to let the des command prompt for the key and enter it interactively.

EXAMPLES

1. To encrypt a file foo, using cipher block chaining with 8 rounds, and fred12345 as the key:

des -k fred12345 -f foo

2. To decrypt the file foo.e

des -d -kfred12345 -f foo.e

3. To encrypt a file foo, using cipher feedback mode (cfb) with 32 rounds:

des -m cfb -r 32 -f foo

4. To decrypt the file foo.e:

des -d -m cfb -r 32 foo.e

5. To encrypt a file foo, and the encrypted output is sent to file foo1:

des < foo > foo1

6. To time trial test FEAL with mode cbc and 32 rounds

des -t -m cbc -r 32

WARNING

If two or more files encrypted with the same key are concatenated and an attempt is made to decrypt the result, only the first of original files is decrypted correctly.

SEE ALSO

feal, loki, khufu, khafre, sbh, dbh, md4, md4x, md5, and haval.

AUTHORS

National Bureau of Standards, US.

des(3c)

NAME

EnDes, DeDes, Des_GetKeySchedule - encrypt and decrypt a DES data block

SYNOPSIS

EnDes(unsigned long block[2], unsigned long des_keyschedule[64], int round)

DeDes(unsigned long block[2], unsigned long des_keyschedule[64], int round)

Des_GetKeySchedule(char key[8], unsigned long des_keyschdule[64], int round)

DESCRIPTION

The C routines - EnDes, DeDes, Des_GetKeySchedule - provide low level access to the Data Encryption Standard algorithm.

DES is the official encryption standard published by the US National Bureau of Standards, 1977. DES is symmetric block cipherment algorithm acting on 64 bits of plaintext to produce a 64-bit ciphertext controlled by 64 bit key. Deciphering is done by using the same key as for enciphering.

The standard version of DES run with 8 rounds and 64-bit key. The extensions to DES have been implemented with this version. The number of rounds has been increased to 32.

The routine Des_GetKeySchedule sets up the key schedule from the 64-bit standard key (8-character key). This routine must be called before calling EnDes, or DeDes.

The routine EnDes encrypts a 64-bit block with the supplied key schedule.

The routine DeDes decrypts a 64-bit block with the supplied key schedule.

Round is the number of rounds to encrypt or decrypt the data block (minimum number is 16, maximum number is 32).

SEE ALSO

feal(3c), loki,(3c) khufu(3c), khafre(3c).

feal(1)

NAME

feal - fast encryption algorithm

SYNOPSIS

feal [[-el-d] -k key -mode -r round -f file | -t]

DESCRIPTION

FEAL is symmetric block encryption algorithm acting on 64 bits of plaintext to produce a 64-bit ciphertext controlled by 64 bit key. It is designed to run fast with software implementation. Deciphering is done by using the same key as for enciphering. FEAL operates with four modes: electronic code book (ecb), cipher block chaining (cbc), cipher feedback (cfb), and output feedback (obf). The cbc mode is the default mode and it is recommended for encryption for increased security. The cfb and ofb modes have the same level of security with cbc mode, but the running time is a bit slower.

The standard version of FEAL run with 8 rounds and 64-bit key (8 characters). The extensions to FEAL have been implemented with this version. The number of rounds has been increased to 32, and the number of key bits increased to 128 (the maximum number of characters of the key is 16).

Options:
-e encryption; the default is encryption;
-d decryption;
-k user-supplied key; if the key is not supplied with the command line, user will be asked for the key interactively; the minimum number of characters for the key is 5;
-m mode, valid modes are ecb, cbc, cfb, ofb; cipher block chaining is the default mode of operation;
-r round, the number of rounds used in enciphering or deciphering a block of data, valid values from 4 to 32 rounds. The more number of rounds of the enciphering, the higher security of the cipher data; the default value is 8;
-f input file, if this option is used, the resulting encryption file will be written to the same name with an extension .e;
-t time trial testing for algorithm running speed.

Since the key is an argument to the feal command, it is potentially visible to users executing ps(1) or a derivative. It is advisable to let the feal command prompt for the key and enter it interactively.

EXAMPLES

1. To encrypt a file foo, using cipher block chaining with 8 rounds, and fred12345 as the key:

feal -k fred12345 -f foo

2. To decrypt the file foo.e

feal -d -kfred12345 -f foo.e

3. To encrypt a file foo, using cipher feedback mode (cfb) with 32 rounds:

feal -m cfb -r 32 -f foo

4. To decrypt the file foo.e:

feal -d -m cfb -r 32 foo.e

5. To encrypt a file foo, and the encrypted output is sent to file foo1:

feal < foo > foo1

6. To time trial test FEAL with mode cbc and 32 rounds

feal -t -m cbc -r 32

WARNING

If two or more files encrypted with the same key are concatenated and an attempt is made to decrypt the result, only the first of original files is decrypted correctly.

SEE ALSO

des, loki, khufu, khafre, sbh, dbh, md4, md4x, md5, and haval.

AUTHORS

The algorithm was originally designed by A.Shimizu and S.Miyaguchi, Electrical Communication Laboratories, NTT, Japan.

feal(3c)

NAME

EnFeal, DeFeal, Feal_GetKeySchedule - encrypt and decrypt a FEAL data block

SYNOPSIS

EnFeal(unsigned long block[2], unsigned long feal_keyschedule[64], int round)

DeFeal(unsigned long block[2], unsigned long feal_keyschedule[64], int round)

Feal_GetKeySchedule(char key[8], unsigned long feal_keyschdule[64], int round)

DESCRIPTION

The C routines - EnFeal, DeFeal, Feal_GetKeySchedule - provide low level access to the Fast Encryption algorithm.

FEAL is symmetric block encryption algorithm acting on 64 bits of plaintext to produce a 64-bit ciphertext controlled by 64 bit key. It is designed to run fast with software implementation. Deciphering is done by using the same key as for enciphering.

The routine Feal_GetKeySchedule sets up the key schedule from the 64-bit standard key (8-character key). This routine must be called before calling EnFeal, or DeFeal.

The routine EnFeal encrypts a 64-bit block with the supplied key schedule.

The routine DeFeal decrypts a 64-bit block with the supplied key schedule.

Round is the number of rounds to encrypt or decrypt the data block (minimum number is 16, maximum number is 32).

SEE ALSO

des(3c), loki,(3c) khufu(3c), khafre(3c).

loki(1)

NAME

loki - LOKI encryption algorithm

SYNOPSIS

loki [[-e|-d] -k key -mode -r round -f file | -t]

DESCRIPTION

LOKI is symmetric block encryption algorithm acting on 64 bits of plaintext to produce a 64-bit ciphertext controlled by 64 bit key. Decryption is done by using the same key as for encryption. LOKI operates with four modes: electronic code book (ecb), cipher block chaining (cbc), cipher feedback (cfb), and output feedback (obf). The cbc mode is the default mode and it is recommended for encryption for increased security. The cfb and ofb modes have the same level of security with cbc mode, but the running time is a bit slower.

The standard version of LOKI run with 8 rounds and 64-bit key (8 characters). The extensions to LOKI have been implemented with this version. The number of rounds has been increased to 32.

Options:

-e encryption; the default is encryption;

-d decryption;

-k user-supplied key; if the key is not supplied with the command line, user will be asked for the key interactively; the minimum number of characters for the key is 5;

-m mode, valid modes are ecb, cbc, cfb, ofb; cipher block chaining is the default mode of operation;

-r round, the number of rounds used in enciphering or deciphering a block of data, valid values from 4 to 32 rounds. The more number of rounds of the enciphering, the higher security of the cipher data; the default value is 8;

-f input file, if this option is used, the resulting encryption file will be written to the same name with an extension .e;

-t time trial testing for algorithm running speed.

Since the key is an argument to the loki command, it is potentially visible to users executing ps(1) or a derivative. It is advisable to let the loki command prompt for the key and enter it interactively.

EXAMPLES

1. To encrypt a file foo, using cipher block chaining with 8 rounds, and fred12345 as the key:

loki -k fred12345 -f foo

2. To decrypt the file foo.e

loki -d -kfred12345 -f foo.e

3. To encrypt a file foo, using cipher feedback mode (cfb) with 32 rounds:

loki -m cfb -r 32 -f foo

4. To decrypt the file foo.e:

loki -d -m cfb -r 32 foo.e

5. To encrypt a file foo, and the encrypted output is sent to file foo1:

loki < foo > foo1

6. To time trial test LOKI with mode cbc and 32 rounds

loki -t -m cbc -r 32

WARNING

If two or more files encrypted with the same key are concatenated and an attempt is made to decrypt the result, only the first of original files is decrypted correctly.

SEE ALSO

des, feal, khufu, khafre, sbh, dbh, md4, md4x, md5, and haval.

AUTHORS

The algorithm was designed by L.Brown, M. Kwan, J.Pieprzyk, and J.Seberry; Department of Computer Science, University College, UNSW, Australian Defense Force Academy, Canberra.

loki(3c)

NAME

EnLoki, DeLoki, Loki_GetKeySchedule - encrypt and decrypt a Loki data block

SYNOPSIS

EnLoki(unsigned long block[2], unsigned long feal_keyschedule[64], int round)

DeLoki(unsigned long block[2], unsigned long feal_keyschedule[64], int round)

Loki_GetKeySchedule(char key[8], unsigned long feal_keyschdule[64], int round)

DESCRIPTION

The C routines - EnLoki, DeLoki, Loki_GetKeySchedule - provide low level access to the Loki Encryption algorithm.

LOKI is symmetric block encryption algorithm acting on 64 bits of plaintext to produce a 64-bit ciphertext controlled by 64 bit key. Decryption is done by using the same key as for encryption.

The routine Loki_GetKeySchedule sets up the key schedule from the 64-bit standard key (8-character key). This routine must be called before calling EnLoki, or DeLoki.

The routine EnLoki encrypts a 64-bit block with the supplied key schedule.

The routine DeLoki decrypts a 64-bit block with the supplied key schedule.

Round is the number of rounds to encrypt or decrypt the data block (minimum number is 16, maximum number is 32).

SEE ALSO

des(3c), feal(3c), khufu(3c), khafre(3c).

khufu(1)

NAME

khufu - KHUFU encryption algorithm

SYNOPSIS

khufu [[-e|-d] -k key -mode -r round -f file | -t]

DESCRIPTION

KHUFU is symmetric block encryption algorithm acting on 64 bits of plaintext to produce a 64-bit ciphertext controlled by 64 bit key. Decryption is done by using the same key as for encryption. KHUFU operates with four modes: electronic code book (ecb), cipher block chaining (cbc), cipher feedback (cfb), and output feedback (obf). The cbc mode is the default mode and it is recommended for encryption for increased security. The cfb and ofb modes have the same level of security with cbc mode, but the running time is a bit slower.

The standard version of KHUFU run with 8 rounds and 64-bit key (8 characters). The number of rounds must be a multiple of 8. The extensions to KHUFU have been implemented with this version. The number of rounds has been increased to 32.

Options:

-e encryption; the default is encryption;

-d decryption;

-k user-supplied key; if the key is not supplied with the command line, user will be asked for the key interactively; the minimum number of characters for the key is 5;

-m mode, valid modes are ecb, cbc, cfb, ofb; cipher block chaining is the default mode of operation;

-r round, the number of rounds used in enciphering or deciphering a block of data, valid values from 8 to 32 rounds. The more number of rounds of the enciphering, the higher security of the cipher data; the default value is 8;

-f input file, if this option is used, the resulting encryption file will be written to the same name with an extension .e;

-t time trial testing for algorithm running speed.

Since the key is an argument to the khufu command, it is potentially visible to users executing ps(1) or a derivative. It is advisable to let the khufu command prompt for the key and enter it interactively.

EXAMPLES

1. To encrypt a file foo, using cipher block chaining with 8 rounds, and fred12345 as the key:

khufu -k fred12345 -f foo

2. To decrypt the file foo.e

khufu -d -kfred12345 -f foo.e

3. To encrypt a file foo, using cipher feedback mode (cfb) with 32 rounds:

khufu -m cfb -r 32 -f foo

4. To decrypt the file foo.e:

khufu -d -m cfb -r 32 foo.e

5. To encrypt a file foo, and the encrypted output is sent to file foo1:

khufu < foo > foo1

6. To time trial test KHUFU with mode cbc and 32 rounds

khufu -t -m cbc -r 32

WARNING

If two or more files encrypted with the same key are concatenated and an attempt is made to decrypt the result, only the first of original files is decrypted correctly.

SEE ALSO

des, feal, loki, khafre, sbh, dbh, md4, md4x, md5, and haval.

AUTHORS

The algorithm was designed by R.C.Merkle, Xerox PARC.

khufu(3c)

NAME

EnKhufu, DeKhufu - encrypt and decrypt a Khufu data block

InitialSboxSchedule, SboxSchedule - get khufu S-box schedule

SYNOPSIS

EnKhufu(unsigned long block[2], unsigned long Sbox[4][256], int round)

DeKhufu(unsigned long block[2], unsigned long Sbox[4][256], int round)

InitialSboxSchedule(unsigned long Sbox[4][256])

SboxSchedule(unsigned long initialSbox[4][256],unsigned long Sbox[4][256], char key[8])

DESCRIPTION

The C routines - EnKhufu, DeKhufu, InitialSboxSchedule, SboxSchedule - provide low level access to the Khufu Encryption algorithm.

Khufu is symmetric block encryption algorithm acting on 64 bits of plaintext to produce a 64-bit ciphertext controlled by 64 bit key. Decryption is done by using the same key as for encryption.

The routines InitialSboxSchedule and SboxSchedule set up the S-box schedule from the 64-bit standard key (8-character key). The routine InitialSboxSchedule must be called first before calling SboxSchedule. EnKhufu or DeKhufu is called after routine SboxSchedule.

The routine EnKhufu encrypts a 64-bit block with the supplied S-box schedule.

The routine DeKhufu decrypts a 64-bit block with the supplied S-box schedule.

Round is the number of rounds to encrypt or decrypt the data block and it must be a multiple of 8. The maximum number is 32.

SEE ALSO

des(3c), feal(3c), loki(3c), khafre(3c).

khafre(1)

NAME

khafre - KHAFRE encryption algorithm

SYNOPSIS

khafre [[-el-d] -k key -mode -r round -f file | -t]

DESCRIPTION

KHAFRE is symmetric block encryption algorithm acting on 64 bits of plaintext to produce a 64-bit ciphertext controlled by 64 bit key. Decryption is done by using the same key as for encryption. KHAFRE operates with four modes: electronic code book (ecb), cipher block chaining (cbc), cipher feedback (cfb), and output feedback (obf). The cbc mode is the default mode and it is recommended for encryption for increased security. The cfb and ofb modes have the same level of security with cbc mode, but the running time is a bit slower.

The standard version of KHAFRE run with 8 rounds and 64-bit key. The number of rounds must be a multiple of 8. The extensions to KHAFRE have been implemented with this version. The number of rounds has been increased to 32.

Options:

-e encryption; the default is encryption;

-d decryption;

-k user-supplied key; if the key is not supplied with the command line, user will be asked for the key interactively; the minimum number of characters for the key is 5;

-m mode, valid modes are ecb, cbc, cfb, ofb; cipher block chaining is the default mode of operation;

-r round, the number of rounds used in enciphering or deciphering a block of data, valid values from 8 to 32 rounds. The more number of rounds of the enciphering, the higher security of the cipher data; the default value is 8;

-f input file, if this option is used, the resulting encryption file will be written to the same name with an extension .e;

-t time trial testing for algorithm running speed.

Since the key is an argument to the khafre command, it is potentially visible to users executing ps(1) or a derivative. It is advisable to let the khafre command prompt for the key and enter it interactively.

EXAMPLES

1. To encrypt a file foo, using cipher block chaining with 8 rounds, and fred12345 as the key:

khafre -k fred12345 -f foo

2. To decrypt the file foo.e

khafre -d -kfred12345 -f foo.e

3. To encrypt a file foo, using cipher feedback mode (cfb) with 32 rounds:

khafre -m cfb -r 32 -f foo

4. To decrypt the file foo.e:

khafre -d -m cfb -r 32 foo.e

5. To encrypt a file foo, and the encrypted output is sent to file foo1:

khafre < foo > foo1

6. To time trial test KHAFRE with mode cbc and 32 rounds

khafre -t -m cbc -r 32

WARNING

If two or more files encrypted with the same key are concatenated and an attempt is made to decrypt the result, only the first of original files is decrypted correctly.

SEE ALSO

des, feal, loki, khufu, sbh, dbh, md4, md4x, md5, and haval.

AUTHORS

The algorithm was designed by R.C.Merkle, Xerox PARC.

khafre(3c)

NAME

EnKhafre, DeKhafre - encrypt and decrypt a Khufu data block

KhafreSboxSchedule - get Khafre standard s-box schedule

SYNOPSIS

EnKhafre(unsigned long block[2], unsigned long Sbox[4][256], unsigned long key[2][2], int round)

DeKhafre(unsigned long block[2], unsigned long Sbox[4][256], unsigned long key[2][2], int round)

KhafreSboxScheduleSchedule(unsigned long Sbox[4][256])

DESCRIPTION

The C routines - EnKhafre, DeKhafre, KhafreSboxSchedule - provide low level access to the Khafre Encryption algorithm.

Khafre is symmetric block encryption algorithm acting on 64 bits of plaintext to produce a 64-bit ciphertext controlled by 64 bit key. Decryption is done by using the same key as for encryption.

The routine KhafreSboxSchedule must be called first, before any attempt to call EnKhafre and DeKhafre.

The routine EnKhafre encrypts a 64-bit block with the supplied key.

The routine DeKhafre decrypts a 64-bit block with the supplied key.

Round is the number of rounds to encrypt or decrypt the data block and it must be a multiple of 8. The maximum number is 32.

SEE ALSO

des(3c), feal(3c), loki(3c), khufu(3c).

sbh(1)

NAME

sbh - Single Block Hashing Algorithm

SYNOPSIS

sbh -f file | -s string -a algorithm -k key -m mode -r round -F Format

DESCRIPTION

sbh is a 64-bit hashing algorithm which uses 64-bit symmetric block encryption algorithms - des, feal, loki, khufu, or khafre - to produce a 64-bit hashing message serving a fingerprint of a string or a file.

The algorithm makes use of three modes - cipher chaining mode (cbc), single block hash (sbh) - to produce the hashing message.

The basic algorithm (des, feal, loki, khufu, and khafre) can be selected through the use of option -a.

Options:
-f input file for hashing;
-s input string for hashing;
-a algorithm for hashing, the valid algorithms are des, feal, loki, khufu, and khafre; the default is des;
-k key for the basic algorithm; the minimum number of characters for the key is 5;
-m mode, the valid modes are cbc, sbh, the default is sbh;
-r round, number of rounds for the basic algorithm, default is 8, the maximum value is 32;
-F user-supplied format.

The output hashing message can be printed according to the user-supplied format. Any characters in the format not followed the character % will be printed without change; otherwise options will be substituted according to the rule as follows:

h message hashing
a algorithm used for hashing
r number of rounds used for hashing
m mode used for hashing
n print a new line character
t print a tab character
% print the % character

The hashing message is printed in hexadecimal format.

Since the key is an argument to the smh command, it is potentially visible to users executing ps(1) or a derivative. It is advisable to let the smh command prompt for the key and enter it interactively.

EXAMPLES

1. To produce the hashing message for file foo, using algorithm des, 8 rounds.

sbh -f foo

or

sbh < foo

2. To produce the hashing message for string "encryption algorithm", using algorithm feal, 32 rounds, cipher block chaining.

sbh -s"encryption algorithm" -a feal -r 32 -m cbc

3. To produce the hashing message for file foo, using algorithm khufu and print with the user-supplied format.

sbh -f foo -a khufu -F"File: foo, Hashing message: %h"

SEE ALSO

des, feal, loki, khufu, khafre, dbh, md4, md4x, md5, and haval.

AUTHORS

The algorithm was proposed by L.Brown., J.Pieprzyk, and J.Seberry, University of New South Wales. See des, feal, loki, khufu, and khafre.

dbh(1)

NAME

dbh - Double Block Hashing Algorithm

SYNOPSIS

dbh -f file | -s string -a algorithm -k key -m mode -r round -F Format

DESCRIPTION

dbh is a 128-bit hashing algorithm which uses 64-bit symmetric block encryption algorithms - des, feal, loki, khufu, or khafre - to produce a 128-bit hashing message serving a fingerprint of a string or a file.

The basic algorithm (des, feal, loki, khufu, and khafre) can be selected through the use of option -a.

Options:
-f input file for hashing;
-s input string for hashing;
-a algorithm for hashing, the valid algorithms are des, feal, loki, khufu, and khafre; the default is des;
-k key for the basic algorithm; the minimum number of characters for the key is 5;
-r round, number of rounds for the basic algorithm, default is 8, the maximum value is 32;
-F user-supplied format.

The output message hashing can be printed according to the user-supplied format. Any characters in the format not followed the character % will be printed without change; otherwise options will be substituted according to the rule as follows:

h message hashing
a algorithm used for hashing
r number of rounds used for hashing
n print a new line character
t print a tab character
% print the % character

The hashing message is printed in hexadecimal format.

Since the key is an argument to the dbh command, it is potentially visible to users executing ps(1) or a derivative. It is advisable to let the dbh command prompt for the key and enter it interactively.

EXAMPLES

1. To produce the hashing message for file foo, using algorithm des, 8 rounds.

dbh -f foo

or

dbh < foo

2. To produce the hashing message for string "encryption algorithm", using algorithm feal, 32 rounds.

dbh -s"encryption algorithm" -a feal -r 32

3. To produce the hashing message for file foo, using algorithm khufu and print with the user-supplied format.

dbh -f foo -a khufu -F"File: foo, Hashing message: %h"

SEE ALSO

des, feal, loki, khufu, khafre, sbh, md4, md4x, md5, and haval.

AUTHORS

The algorithm was proposed by J.J.Quisquater and M.Girault; Phillips Research Laboratory. The variant of the original algorithm, by L.Brown, J.Pieprzyk, and J.Seberry; University of New South Wales, is implemented for this command. See also des, feal, loki, khufu, and khafre.

md4(1)

NAME

md4 - MD4 message digest algorithm

SYNOPSIS

md4 -f file | -s string | -t -F Format

DESCRIPTION

MD4 is a message digest algorithm which takes input message (or file) of arbitrary length and produces an output 128-bit "fingerprint" or "message digest", in such a way that it is computationally infeasible to produce two message digests, or to produce any message having a given prespecified target message digest.

Options:

-f input file for hashing;

-s input string for hashing;

-t time trial testing;

-F user-supplied format.

The output message hashing can be printed according to the user-supplied format. Any characters in the format not followed the character % will be printed without change; otherwise options will be substituted according to the rule as follows:

h message hashing

a algorithm name (md4)

n print a new line character

t print a tab character

% print the % character

The hashing message is printed as two long words in hexadecimal format.

EXAMPLES

1. To produce the message digest for file foo.

md4 -f foo

or

md4 < foo

2. To produce a message digest for string "encryption algorithm".

md4 -s"encryption algorithm"

3. To produce the hashing message for file foo and print with the user-supplied format.

md4 -f foo -F"File: foo, Hashing message: %h"

4. To time trial test for the algorithm

md4 -t

SEE ALSO

des, feal, loki, khufu, khafre, sbh, dbh, md4x, md5, and haval

AUTHORS

The algorithm was designed by R.L.Rivest, Laboratory for Computer Science, MIT.

md4x(1)

NAME

md4x - MD4 message digest algorithm (extended version)

SYNOPSIS

md4x -f file | -s string | -t -F Format

DESCRIPTION

MD4X is the extended version of the message digest algorithm MD4, which takes input message (or file) of arbitrary length and produces an output 256-bit "fingerprint" or "message digest", in such a way that it is computationally infeasible to produce two message digests, or to produce any message having a given prespecified target message digest.

Options:
-f input file for hashing;
-s input string for hashing;
-t time trial testing;
-F user-supplied format.

The output message hashing can be printed according to the user-supplied format. Any characters in the format not followed the character % will be printed without change; otherwise options will be substituted according to the rule as follows:

h message hashing
a algorithm name (md4x)
n print a new line character
t print a tab character
% print the % character

The hashing message is printed as two long words in hexadecimal format.

EXAMPLES

1. To produce the message digest for file foo.

md4x -f foo

or

md4x < foo

2. To produce a message digest for string "encryption algorithm".

md4x -s"encryption algorithm"

3. To produce the hashing message for file foo and print with the user-supplied format.

md4x -f foo -F"File: foo, Hashing message: %h"

4. To time trial test for the algorithm

md4x -t

SEE ALSO

des, feal, loki, khufu, khafre, sbh, dbh, md4, md5, and haval

AUTHORS

The algorithm was designed by R.L.Rivest, Laboratory for Computer Science, MIT.

md5(1)

NAME

md5 - MD5 message digest algorithm

SYNOPSIS

md5 -f file | -s string | -t -F Format

DESCRIPTION

MD5 is the revised version of the message digest algorithm MD4, which takes input message (or file) of arbitrary length and produces an output 128-bit "fingerprint" or "message digest", in such a way that it is computationally infeasible to produce two message digests, or to produce any message having a given prespecified target message digest.

Options:

-f input file for hashing;

-s input string for hashing;

-t time trial testing;

-F user-supplied format.

The output message hashing can be printed according to the user-supplied format. Any characters in the format not followed the character % will be printed without change; otherwise options will be substituted according to the rule as follows:

h message hashing

a algorithm name (md5)

n print a new line character

t print a tab character

% print the % character

The hashing message is printed as two long words in hexadecimal format.

EXAMPLES

1. To produce the message digest for file foo.

md5 -f foo

or

md5 < foo

2. To produce a message digest for string "encryption algorithm".

md5 -s"encryption algorithm"

3. To produce the hashing message for file foo and print with the user-supplied format.

md5 -f foo -F"File: foo, Hashing message: %h"

4. To time trial test for the algorithm

md5 -t

SEE ALSO

des, feal, loki, khufu, khafre, sbh, dbh, md4, md4x, and haval.

AUTHORS

The algorithm was designed by R.L.Rivest, Laboratory for Computer Science, MIT.

haval(1)

NAME

haval - HAVAL hashing algorithm

SYNOPSIS

haval -f file | -s string | -t -p pass -d digest_length -F Format

DESCRIPTION

HAVAL is a one-way hashing algorithm with variable length of output. It is a deterministic algorithm which compresses an arbitrary long message (or a file) into a value of specified length. It is conjectured that it computationally infeasible to find two distinct messages that have the same digest. The output length of the HAVAL is a variable, which is 128, 160, 192, 224, or 256 bits, and is specified with option -d. HAVAL has a parameter (option -p) to control the number of passes (2, 3, or 4) to process the message digest. The higher is the number of passes the more security is the output hashing message.

Combining output length with pass, HAVAL obtains fifteen (15) choices for practical applications where different levels of security required.

Options:
-f input file for hashing;
-s input string for hashing;
-t time trial testing;
-p the number of pass to process; the valid values are 2, 3, and 4; the default value is 2
-d required output length of digest message in bits; the valid values are 128, 160, 192, 224, and 256; the default value is 128;
-F user-supplied format.

The output message hashing can be printed according to the user-supplied format. Any characters in the format not followed the character % will be printed without change; otherwise options will be substituted according to the rule as follows:

h message hashing
a algorithm name (haval)
p number of passes
d digest length
n print a new line character
t print a tab character

% print the % character

The hashing message is printed as two long words in hexadecimal format.

EXAMPLES

1. To produce the 128-bit message digest for file foo.

haval -f foo

or

haval < foo

2. To produce a 128-bit message digest for string "encryption algorithm".

haval -s"encryption algorithm"

3. To produce the 128-bit hashing message for file foo and print with the user-supplied format.

haval -f foo -F"File: foo, Hashing message: %h"

4. To produce the 192-bit hashing message for string "haval", using 4 passes to process.

haval -s"haval" -p4 -d192

5. To produce the 256-bit hashing message for string "haval", using 3 passes to process and print with the user-supplied format.

haval -s"haval" -p3 -d256 -F"String: haval, %a%h%p%d"

6. To time trial test for the algorithm

haval -t

SEE ALSO

des, feal, loki, khufu, khafre, sbh, dbh, md4, md4x, and md5.

AUTHORS

The algorithm was designed by Y.Zheng, Centre for Computer Security Research, Department of Computer Science, University of Wollongong.

9. References

[Bih91a] Differential Cryptanalysis of DES-like Cryptosystems; E.Biham, A.Shamir; Journal of Cryptology, vol 4(1), pp3-72, 1991; or Advances in Cryptology, Proceedings of Crypto'90, pp2-21, Springer-Verlag, LNCS 537.

[Bib91b] Differential Cryptanalysis of Snefru, Khafre, REDOC-II, LOKI and Lucifer; E.Biham and A.Shamir; Advances in Cryptology - Proceedings of Crypto'91, pp156-171, Springer-Verlag, LNCS 576.

[Bib91c] Differential Cryptanalysis of FEAL and N-Hash; E.Biham and A.Shamir; Advances in Cryptology - Proceedings of Eurocrypt'91, pp1-16, Springer-Verlag, LNCS 547.

[Boe88] Cryptanalysis of FEAL; B.Den Boer; presented at Advances in Cryptology - Abstracts of Crypto'87, pp37-49, Springer-Verlag, LNCS 293.

[Boe91] An attack on the last two rounds of MD4; B.den Boer and A.Bosselaers; Advances in Cryptology - Proceeding of Crypto'91, pp194-203, Springer-Verlag, LNCS 576.

[Bro89] On the design of permutation P in DES type cryptosystems; L.Brown and J.Seberry; Advances in Cryptology - Proceedings in Eurocrypt'89, pp696-705, Springer-Verlag, LNCS 434.

[Bro90] LOKI - A Cryptographic primitive for authentication and secrecy applications; L.Brown, J.Pieprzyk, and J.Seberry; Advances in Cryptology - Proceedings of Austcrypt '90, pp229-236, Springer-Verlag, LNCS 453.

[Bro91] Improving Resistance to Differential Cryptanalysis and Redesign of LOKI; L.Brown, M.Kwan, J.Pieprzyk and J.Seberry; Technical report CS38/91; Department of Computer Science, UNSW, Australian Defense Force Academy, Canberra.

[Dav82] The expected cycle size of the key stream in output feedback encipherment; D.W.Davies and G.I.P.Parkin; Advances in cryptology - Proceedings of Crypto'82, pp12-34.

[Fel89] UNIX password Security - ten years later; D.C.Feldmeier and P.R.Karn; Advances in Cryptology - Proceedings of Crypto'89, pp44-63, Springer-Verlag, LNCS 435.

[Gar91] Practical UNIX Security; Simson Garfinkel and Gene Spafford; O'Reilly and Associates, Inc, 1991.

[Kah67] The code breakers, the story of secret writing; Kahn; MacMillan, New York, 1967.

[Knu91] Cryptanalysis of LOKI; LR.Knudsen; Asiacrypt'91 Abstracts, pp19-24.

[Lai92] Hash functions based on block ciphers; X.Lai and J.Massey; Advances in Cryptology - Proceedings of Eurocrypt'92, pp55-70, Springer-Verlag, LNCS 658.

[Mat92] A new method for known plaintext attack of FEAL cipher; M.Matsui and A.Yamgishi; Advances in Cryptology - Proceedings of Eurocrypt'92, pp81-91, Springer-Verlag, LNCS 658.

[Mer82] Secrecy, authentication, and public key systems; R.C.Merkle; Ann Arbor: UMI Research Press, 1982.

[Mer90] Fast software encryption functions; R.C.Merkle; Advances in Cryptology - Proceedings of Crypto'90, pp476-501, Springer-Verlag, LNCS 537.

[Mey82] Cryptography: a new dimension in computer security; C.H.Meyer, S.M.Matyas; John Wiley & Sons, New York, 1982.

[Miy90] The FEAL Cipher Family; S.Miyaguchi; Advances in Cryptology - Proceedings of Crypto'90, pp627-638, Springer-Verlag, LNCS 537.

[NBS77] Data Encryption Standard, NBS, FIPS Pub 46, US National Bureau of Standards, Washington, DC, 1977.

[NBS80] DES modes of operation; National Bureau of Standards; Federal Information Processing Standard, U.S. Department of Commerce, FIPS Pub 81, 1980.

[Pfl89] Security in computing; Charles P. Pfleeger; Prentice Hall, 1989.

[Qui89] 2n-bit hash function using n-bit symmetric block cipher algorithms; J.J.Quisquater and M.Girault; Advances in Cryptology - Proceedings of Eurocrypt'89, pp102-109, Springer-Verlag, LNCS 434.

[Riv90] The MD4 Message Digest Algorithm - Technical Report MIT/LCS/TM-434; R.L.Rivest; Laboratory for Computer Science; Massachusetts Institute of Technology; Oct 1990; or Advances in Cryptology - Proceedings of Crypto'90, pp.303-311, Springer-Verlag, LNCS 537.

[Riv91] The MD5 Message Digest Algorithm - RFC (Internet Draft); R.L.Rivest and S.Dusse; Network Working Group Internet Draft, RSA Data Security Inc., 10 July 1991.

[Seb89] Cryptography : an introduction to computer security; Jennifer Seberry and Josef Pieprzyk; Prentice Hall, 1989.

[Sha49] Communication theory of secrecy systems; C.E. Shannon; Bell System Technical Journal, 1949, Vol.28, pp656-715.

[Shi87] Fast Data Encipherment Algorithm FEAL; A.Shimizu, S.Miyaguchi; Advances in Cryptology - Proceedings of Eurocrypt'87, pp267-278, Springer-Verlag, LNCS 304.

[Sim92] Contemporary Cryptology - The science of information integrity; Edited by G.J.Simmons; IEEE Press, 1992.

[Tar91] A Known-plaintext attack of FEAL-4 and FEAL-6; Anne Tardy-Corfdir and Henri Gilbert; Advances in Cryptology - Proceedings of Crypto'91, pp172-182, Springer-Verlag, LNCS 576.

[Web85] On the design of S-boxes; A.F.Webster and S.E.Tavares; Advances in Cryptology - Proceedings of Crypto'85, pp523-534, Springer-Verlag, LNCS 434.

[Zhe92] HAVAL - A One-Way Hashing Algorithm with Variable Length of Output; Y.Zheng; Centre for Computer Security Research; Department of Computer Science, University of Wollongong, 1992.

Appendix

C Program listings

1. main.c - main routine for 64-bit cryptographic primitives
2. des.c - data encryption standard
3. feal.c - fast encryption algorithm
4. loki.c - loki encryption algorithm
5. khufu.c - khufu encryption algorithm
6. khafre.c - khafre encryption algorithm
7. sbh.c - single block hashing algorithm
8. dbh.c - double block hashing algorithm
9. md4.c - message digest algorithm MD4
10. md4x.c - message digest algorithm MD4 (extended version)
11. md5.c - message digest algorithm MD5
12. haval.c - one-way hashing algorithm with variable length output

```
/*
** Routine : main.c
** Main routine for 64-bit block cipher symmetric algorithms -
** des, feal, loki, khufu, and khafre.
** Dzung Le - Oct, 92
**
*/
#include <stdio.h>
#include <time.h>

#define ECB   0   /* Electronic code book */
#define CBC   1   /* Cipher block chaining */
#define CFB   2   /* Cipher feedback */
#define OFB   3   /* Output feedback */

typedef unsigned long Long;

/*
** Function prototypes:
** Feal encryption algorithm
*/
#if FEAL
void Feal_GetKeySchedule(), EnFeal(), DeFeal();
#endif

/*
** Des encryption standard algorithm
*/
#if DES
void EnDes(), DeDes(), Des_GetKeySchedule();
#endif

/*
** Loki encryption standard algorithm
*/
#if LOKI
void EnLoki(), DeLoki(), Loki_GetKeySchedule();
#endif

/*
** Khufu encryption algorithm
*/
#if KHUFU
void InitialSboxSchedule(), SboxSchedule(), EnKhufu(), DeKhufu();
#endif

/*
** Khafre encryption algorithm
*/
#if KHAFRE
void KhafreSboxSchedule(), EnKhafre(), DeKhafre();
#endif

/*
** General encryption, decryption, and time trial test routines
*/
void Do_Encrypt(), Do_Decrypt(), TimeTrialTest();

/*
** Global variables:
** Initial vector, command name,
** and default input, output file pointers
*/
Long iv[2] = {(Long)0, (Long)0};
char *cmdname;
FILE *fpi, *fpo;

/*
** Main rountine
**
*/
main(argc,argv)
int argc;
char **argv;
{
```

```
int c,eflag,tflag,dflag,rflag,fflag,mflag,kflag,errflag;
extern char *optarg;
char filename[160];
int  mode = CBC;
int  round = 8;
char tkey[20],*key=tkey;
long time1, time2;

/*
** Option processing
*/
tflag=dflag=eflag=rflag=fflag=mflag=kflag=errflag= 0;

cmdname = argv[0];

while ((c=getopt(argc,argv,"edf:k:r:m:t")) != EOF) {
  switch (c) {
    case 'e': eflag++;
              break;
    case 'd': dflag++;
              break;
    case 'f': fflag++;
              strcpy(filename,optarg);
              break;
    case 'k': kflag++;
              key = optarg;
              break;
    case 't': tflag++;
              break;
    case 'm': if ((strcmp(optarg,"ecb")) && (strcmp(optarg,"cbc")) &&
                (strcmp(optarg,"cfb")) && (strcmp(optarg,"ofb"))) {
                errflag++;
              }
              else {
                if (!strcmp(optarg,"ecb")) mode = ECB;
                if (!strcmp(optarg,"cbc")) mode = CBC;
                if (!strcmp(optarg,"cfb")) mode = CFB;
                if (!strcmp(optarg,"ofb")) mode = OFB;
              }
              break;
    case 'r': if (((round = atoi(optarg)) == 0) ||
                (round > 32) || (round < 4))
                errflag++;
              break;
    default:  errflag++;
  }
}

/*
** Error checking
*/
if ((!eflag) && (!dflag))
  eflag++;

if (!kflag) {
  while ((key = (char *)getpass("Key: ")) == (char *)0) {
    fprintf(stderr,"%s: invalid password!\n");
  }
}
else if (strlen(key) < 5) {
  fprintf(stderr,"\n%s: key length is too short (min: 5 chars).\n\n",
    argv[0]);
  exit(1);
}

if (errflag) {
  fprintf(stderr,"\nUsage: %s [ [ -e | -d ] -k key ", cmdname);
  fprintf(stderr,"-m mode -r round -f infile ]\n\n");
  exit(1);
}

if (tflag) {
  time(&time1);
  TimeTrialTest(mode, key, round);
  time(&time2);
  fprintf(stdout,"Real time to encrypt 100-Kb block: %d", time2-time1);
```

```
      switch (mode) {
        case ECB: fprintf(stdout," - mode(ECB)"); break;
        case CBC: fprintf(stdout," - mode(CBC)"); break;
        case CFB: fprintf(stdout," - mode(CFB)"); break;
        case OFB: fprintf(stdout," - mode(OFB)"); break;
      }
      fprintf(stdout," - round(%d)\n", round);
    }
    else {
      if (fflag) {
        if ((fpi = fopen(filename, "r")) == NULL) {
          fprintf(stderr,"Couldn't open input file %s!\n", filename);
          exit(1);
        }
        if (!strcmp(filename + strlen(filename) - 2, ".e"))
          *(filename + strlen(filename) - 2) = '\0';
        else strcat(filename,".e");
        if ((fpo = fopen(filename, "w")) == NULL) {
          fprintf(stderr,"Couldn't open input file %s!\n", filename);
          exit(1);
        }
      }
      else {
        fpi = stdin; fpo = stdout;
      }

      if (eflag)
        Do_Encrypt(mode,key,round);

      if (dflag)
        Do_Decrypt(mode,key,round);
    }
  }

  /*
  ** Do_Encrypt() : encrypt a file with a mode, key schedule,
  ** and a number of rounds.
  */
  void Do_Encrypt(mode,key,round)
  int mode;
  char *key;
  int round;
  {
    int i, cnt=0;
    Long work[2];
    unsigned char char1, char2, *cptr = (char *)iv;
  #if FEAL
    char ekey[17];
    Long feal_keyschedule[64];

    /*
    ** feal: default key initialisation (128 bits)
    */
    for (i=0;i < 16;i++) ekey[i] = 0;

    if (strlen(key) > 16)
      *(key + 16) = '\0';

    for (i=0;i < strlen(key);i++)
      ekey[i] = key[i];
    /*
    ** create key schedule for feal
    */
    Feal_GetKeySchedule(ekey,feal_keyschedule,round);
  #endif

  #if DES
    char dkey[9];
    Long des_keyschedule[32][2];

    /*
    ** des: default key initialisation (64 bits)
    */
    for (i=0;i < 8;i++) dkey[i] = 0;

    if (strlen(key) > 8)
```

```
    *(key + 8) = '\0';

  for (i=0;i < strlen(key);i++)
    dkey[i] = key[i];
  /*
  ** create key schedule for des
  */
  Des_GetKeySchedule(dkey, des_keyschedule, round);
#endif

#if LOKI
  char lkey[9];
  Long loki_keyschedule[64];
  /*
  ** loki: default key initialisation (64 bits)
  */
  for (i=0;i < 8;i++) lkey[i] = 0;

  if (strlen(key) > 8)
    *(key + 8) = '\0';

  for (i=0;i < strlen(key);i++)
    lkey[i] = key[i];
  /*
  ** create key schedule for loki
  */
  Loki_GetKeySchedule(lkey, loki_keyschedule, round);
#endif

#if KHUFU
  Long InitialSbox[4][256];
  Long Sbox[4][256];

  InitialSboxSchedule(Sbox);
  InitialSboxSchedule(Sbox,InitialSbox,key);
#endif

#if KHAFRE
  char Tkey[17];
  Long **Lkey = (Long **)Tkey;
  Long StandardSBox[4][256];

  /*
  ** khafre: default key initialisation (128 bits)
  */
  for (i=0;i < 16;i++)
    Tkey[i] = i + 11;

  if (strlen(key) > 16)
    *(key + 16) = '\0';

  for (i=0;i < strlen(key);i++)
    Tkey[i] = key[i];

  KhafreSboxSchedule(StandardSBox);
#endif

  work[0] = work[1] = 0L;

  switch (mode) {
    case ECB:    /* Electronic Code Book mode */
      do {
        if ((cnt = fread((char *)work, 1, 8, fpi)) != 8)
          ((char *)work)[7] = cnt;

#if FEAL
        EnFeal(work,feal_keyschedule,round);
#endif

#if DES
        EnDes(work,des_keyschedule,round);
#endif

#if LOKI
        EnLoki(work,loki_keyschedule,round);
#endif
```

```
#if KHUFU
        EnKhufu(work, Sbox, round);
#endif

#if KHAFRE
        EnKhafre(work, StandardSBox, Lkey,round);
#endif

        fwrite((char *)work, 1, 8, fpo);

      } while (cnt == 8);

      break;

    case CBC:    /* Cipher Block Chaining mode */
      do {
        if ((cnt = fread((char *)work, 1, 8, fpi)) != 8)
          ((char *)work)[7] = cnt;

        work[0] ^= iv[0];
        work[1] ^= iv[1];

#if FEAL
        EnFeal(work,feal_keyschedule,round);
#endif

#if DES
        EnDes(work,des_keyschedule,round);
#endif

#if LOKI
        EnLoki(work,loki_keyschedule,round);
#endif

#if KHUFU
        EnKhufu(work, Sbox, round);
#endif

#if KHAFRE
        EnKhafre(work, StandardSBox, Lkey,round);
#endif

        iv[0] = work[0];
        iv[1] = work[1];

        fwrite((char *)work, 1, 8, fpo);

      } while (cnt == 8);

      break;

    case CFB:      /* 8-bit Cipher FeedBack mode */
      do {
        if ((cnt = fread((char *)&char1, 1, 1, fpi)) == 1) {

#if FEAL
          EnFeal(iv,feal_keyschedule,round);
#endif

#if DES
          EnDes(iv,des_keyschedule,round);
#endif

#if LOKI
          EnLoki(iv,loki_keyschedule,round);
#endif

#if KHUFU
          EnKhufu(iv, Sbox, round);
#endif

#if KHAFRE
          EnKhafre(iv, StandardSBox, Lkey,round);
#endif
```

```
        char1 ^= *cptr;

        fwrite((char *)&char1, 1, 1, fpo);

        for (i=0;i < 7;i++)
          *(cptr + i) = *(cptr + i + 1);

        *(cptr + 7) = char1;
      }
    } while (cnt == 1);

    break;

  case OFB:    /* 8-bit Output FeedBack mode */
    do {
      if ((cnt = fread((char *)&char1, 1, 1, fpi)) == 1) {

#if FEAL
        EnFeal(iv,feal_keyschedule,round);
#endif

#if DES
        EnDes(iv,des_keyschedule,round);
#endif

#if LOKI
        EnLoki(iv,loki_keyschedule,round);
#endif

#if KHUFU
        EnKhufu(iv, Sbox, round);
#endif

#if KHAFRE
        EnKhafre(iv, StandardSBox, Lkey,round);
#endif

        char1 ^= *cptr; char2 = *cptr;

        fwrite((char *)&char1, 1, 1, fpo);

        for (i=0;i < 7;i++)
          *(cptr + i) = *(cptr + i + 1);

        *(cptr + 7) = char2;
      }
    } while (cnt == 1);

    break;
  }
}

/*
** Do_Decrypt() : decrypt a file with a mode, key schedule,
** and a number of rounds.
*/
void Do_Decrypt(mode,key,round)
int mode;
char *key;
int round;
{
  int i, cnt = 0;
  Long work[2];
  Long prev[2];
  Long nextiv[2];
  unsigned char char1, char2, *cptr = (char *)iv;

#if FEAL
  char ekey[17];
  int feal_keyschedule[64];

  /*
  ** feal: default key initialisation
  */
  for (i=0;i < 16;i++) ekey[i] = 0;
```

```
  if (strlen(key) > 16)
    *(key + 16) = '\0';

  for (i=0;i < strlen(key);i++)
    ekey[i] = key[i];
  /*
  ** create key schedule for feal
  */
  Feal_GetKeySchedule(ekey,feal_keyschedule,round);
#endif

#if DES
  char dkey[9];
  Long des_keyschedule[32][2];

  /*
  ** des: default key initialisation (64 bits)
  */
  for (i=0;i < 8;i++) dkey[i] = 0;

  if (strlen(key) > 8)
    *(key + 8) = '\0';

  for (i=0;i < strlen(key);i++)
    dkey[i] = key[i];
  /*
  ** create key schedule for des
  */
  Des_GetKeySchedule(dkey, des_keyschedule, round);
#endif

#if LOKI
  char lkey[9];
  Long loki_keyschedule[64];

  /*
  ** loki: default key initialisation (64 bits)
  */
  for (i=0;i < 8;i++) lkey[i] = 0;

  if (strlen(key) > 8)
    *(key + 8) = '\0';

  for (i=0;i < strlen(key);i++)
    lkey[i] = key[i];
  /*
  ** create key schedule for loki
  */
  Loki_GetKeySchedule(lkey, loki_keyschedule, round);
#endif

#if KHUFU
  Long InitialSbox[4][256];
  Long Sbox[4][256];

  InitialSboxSchedule(Sbox);
  InitialSboxSchedule(Sbox,InitialSbox,key);
#endif

#if KHAFRE
  char Tkey[17];
  Long **Lkey = (Long **)Tkey;
  Long StandardSBox[4][256];

  /*
  ** khafre: default key initialisation (128 bits)
  */
  for (i=0;i < 16;i++)
    Tkey[i] = i + 11;

  if (strlen(key) > 16)
    *(key + 16) = '\0';

  for (i=0;i < strlen(key);i++)
    Tkey[i] = key[i];
```

```
  KhafreSboxSchedule(StandardSBox);
#endif

  work[0] = work[1] = 0L;

  switch (mode) {
    case ECB:    /* Electronic Code Book mode */
      cnt = fread((char *)work,1,8,fpi);
      do {
#if FEAL
        DeFeal(work,feal_keyschedule,round);
#endif

#if DES
        DeDes(work,des_keyschedule,round);
#endif

#if LOKI
        DeLoki(work,loki_keyschedule,round);
#endif

#if KHUFU
        DeKhufu(work,Sbox,round);
#endif

#if KHAFRE
        DeKhafre(work,StandardSBox,Lkey,round);
#endif

        prev[0] = work[0]; prev[1] = work[1];

        if ((cnt = fread((char *)work,1,8,fpi)) != 8) {
          cnt = ((char *)prev)[7];
          if (cnt < 0 || cnt > 7) {
            fprintf(stderr,"%s: Corrupted ciphertext\n", cmdname);
            exit(-1);
          }
          else if (cnt > 0)
            fwrite((char *)prev,1,cnt,fpo);
        }
        else fwrite((char *)prev, 1, 8, fpo);

      } while (cnt == 8);

      break;

    case CBC:    /* Cipher Block Chaining mode */
      cnt = fread((char *)work,1,8,fpi);
      do {
        nextiv[0] = work[0]; nextiv[1] = work[1];

#if FEAL
        DeFeal(work,feal_keyschedule,round);
#endif

#if DES
        DeDes(work,des_keyschedule,round);
#endif

#if LOKI
        DeLoki(work,loki_keyschedule,round);
#endif

#if KHUFU
        DeKhufu(work,Sbox,round);
#endif

#if KHAFRE
        DeKhafre(work,StandardSBox,Lkey,round);
#endif

        work[0] ^= iv[0]; work[1] ^= iv[1];
        iv[0] = nextiv[0]; iv[1] = nextiv[1];

        prev[0] = work[0]; prev[1] = work[1];
```

```
        if ((cnt = fread((char *)work,1,8,fpi)) != 8) {
          cnt = ((char *)prev)[7];
          if (cnt < 0 || cnt > 7) {
            fprintf(stderr,"%s: Corrupted ciphertext\n", cmdname);
            exit(-1);
          }
          else if (cnt > 0)
            fwrite((char *)prev,1,cnt,fpo);
        }
        else fwrite((char *)prev, 1, 8, fpo);

      } while (cnt == 8);

      break;

    case CFB:       /* 8-bit Cipher FeedBack mode */
      do {
        if ((cnt = fread((char *)&char1, 1, 1, fpi)) == 1) {

#if FEAL
          EnFeal(iv,feal_keyschedule,round);
#endif

#if DES
          EnDes(iv,des_keyschedule,round);
#endif

#if LOKI
          EnLoki(iv,loki_keyschedule,round);
#endif

#if KHUFU
          EnKhufu(iv, Sbox, round);
#endif

#if KHAFRE
          EnKhafre(iv, StandardSBox, Lkey,round);
#endif

          char2 = char1; char1 ^= *cptr;

          fwrite((char *)&char1, 1, 1, fpo);

          for (i=0;i < 7;i++)
            *(cptr + i) = *(cptr + i + 1);

          *(cptr + 7) = char2;
        }
      } while (cnt == 1);

      break;

    case OFB:       /* 8-bit Output FeedBack mode */
      do {
        if ((cnt = fread((char *)&char1, 1, 1, fpi)) == 1) {

#if FEAL
          EnFeal(iv,feal_keyschedule,round);
#endif

#if DES
          EnDes(iv,des_keyschedule,round);
#endif

#if LOKI
          EnLoki(iv,loki_keyschedule,round);
#endif

#if KHUFU
          EnKhufu(iv, Sbox, round);
#endif

#if KHAFRE
          EnKhafre(iv, StandardSBox, Lkey,round);
#endif
          char1 ^= *cptr;
```

```
        fwrite((char *)&char1, 1, 1, fpo);

        char2 = *cptr;

        for (i=0;i < 7;i++)
          *(cptr + i) = *(cptr + i + 1);

        *(cptr + 7) = char2;
      }
    } while (cnt == 1);

    break;
  }
}

/*
** TimeTrialTest() : encryption test of a file with a mode, key schedule,
** and a number of rounds.
*/
void TimeTrialTest(mode,key,round)
int mode;
char *key;
int round;
{
  int i, j, k, cnt=0;
  Long work[2];
  unsigned char char1=1, char2=2, *cptr = (char *)iv;
#if FEAL
  char ekey[17];
  Long feal_keyschedule[64];

  /*
  ** feal: default key initialisation (128 bits)
  */
  for (i=0;i < 16;i++) ekey[i] = 0;

  if (strlen(key) > 16)
    *(key + 16) = '\0';

  for (i=0;i < strlen(key);i++)
    ekey[i] = key[i];
  /*
  ** create key schedule for feal
  */
  Feal_GetKeySchedule(ekey,feal_keyschedule,round);
#endif

#if DES
  char dkey[9];
  Long des_keyschedule[32][2];

  /*
  ** des: default key initialisation (64 bits)
  */
  for (i=0;i < 8;i++) dkey[i] = 0;

  if (strlen(key) > 8)
    *(key + 8) = '\0';

  for (i=0;i < strlen(key);i++)
    dkey[i] = key[i];
  /*
  ** create key schedule for des
  */
  Des_GetKeySchedule(dkey, des_keyschedule, round);
#endif

#if LOKI
  char lkey[9];
  Long loki_keyschedule[64];
  /*
  ** loki: default key initialisation (64 bits)
  */
  for (i=0;i < 8;i++) lkey[i] = 0;
```

```
  if (strlen(key) > 8)
    *(key + 8) = '\0';

  for (i=0;i < strlen(key);i++)
    lkey[i] = key[i];
  /*
  ** create key schedule for loki
  */
  Loki_GetKeySchedule(lkey, loki_keyschedule, round);
#endif

#if KHUFU
  Long InitialSbox[4][256];
  Long Sbox[4][256];

  InitialSboxSchedule(Sbox);
  InitialSboxSchedule(Sbox,InitialSbox,key);
#endif

#if KHAFRE
  char Tkey[17];
  Long **Lkey = (Long **)Tkey;
  Long StandardSBox[4][256];

  /*
  ** khafre: default key initialisation (128 bits)
  */
  for (i=0;i < 16;i++)
    Tkey[i] = i + 11;

  if (strlen(key) > 16)
    *(key + 16) = '\0';

  for (i=0;i < strlen(key);i++)
    Tkey[i] = key[i];

  KhafreSboxSchedule(StandardSBox);
#endif

  work[0] = work[1] = 0L;

  switch (mode) {
    case ECB:    /* Electronic Code Book mode */

      for (i=0;i < 12500;i++) {
#if FEAL
          EnFeal(work,feal_keyschedule,round);
#endif

#if DES
          EnDes(work,des_keyschedule,round);
#endif

#if LOKI
          EnLoki(work,loki_keyschedule,round);
#endif

#if KHUFU
          EnKhufu(work, Sbox, round);
#endif

#if KHAFRE
          EnKhafre(work, StandardSBox, Lkey,round);
#endif
      }
      break;

    case CBC:    /* Cipher Block Chaining mode */
      for (i=0;i < 12500;i++) {

          work[0] ^= iv[0];
          work[1] ^= iv[1];

#if FEAL
          EnFeal(work,feal_keyschedule,round);
#endif
```

```
#if DES
        EnDes(work,des_keyschedule,round);
#endif

#if LOKI
        EnLoki(work,loki_keyschedule,round);
#endif

#if KHUFU
        EnKhufu(work, Sbox, round);
#endif

#if KHAFRE
        EnKhafre(work, StandardSBox, Lkey,round);
#endif

        iv[0] = work[0];
        iv[1] = work[1];
      }
      break;

    case CFB:     /* 8-bit Cipher FeedBack mode */
      for (k=0;k < 12500;k++) {
        for (j=0;j < 8;j++) {

#if FEAL
        EnFeal(iv,feal_keyschedule,round);
#endif

#if DES
        EnDes(iv,des_keyschedule,round);
#endif

#if LOKI
        EnLoki(iv,loki_keyschedule,round);
#endif

#if KHUFU
        EnKhufu(iv, Sbox, round);
#endif

#if KHAFRE
        EnKhafre(iv, StandardSBox, Lkey,round);
#endif

        char1 ^= *cptr;

        for (i=0;i < 7;i++)
          *(cptr + i) = *(cptr + i + 1);

        *(cptr + 7) = char1;
      }
    }
    break;

    case OFB:     /* 8-bit Output FeedBack mode */
      for (k=0;k < 12500;k++) {
        for (j=0;j < 8;j++) {

#if FEAL
        EnFeal(iv,feal_keyschedule,round);
#endif

#if DES
        EnDes(iv,des_keyschedule,round);
#endif

#if LOKI
        EnLoki(iv,loki_keyschedule,round);
#endif

#if KHUFU
        EnKhufu(iv, Sbox, round);
#endif
```

```
#if KHAFRE
        EnKhafre(iv, StandardSBox, Lkey,round);
#endif

        char1 ^= *cptr; char2 = *cptr;

        for (i=0;i < 7;i++)
          *(cptr + i) = *(cptr + i + 1);

        *(cptr + 7) = char2;
      }
    }
    break;
  }
}

/*
** End of main.c
**
*/
```

```
/*
** Include file: des.h
** Data Encryption Standard
** Dzung Le - Oct, 92
**
*/

#define MASK28  0xFFFFFFF0L
#define KEYBIT1 0x20
#define DESBIT1 0x80000000L
#define MASK6   0x3F

/*
** 28-bit left rotation
*/
#define RotLeft28(b,pos) (b = ((b << pos) | (b >> (28 - pos))) & MASK28)

/*
** Initial Permutation
*/
char des_IP[64] = {
  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};

/*
** Final Permutation
*/
char des_FP[64] = {
  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};

/*
** Expansion Table
*/
char des_E[64] = {
  32,  1,  2,  3,  4,  5, 33, 34,
   4,  5,  6,  7,  8,  9, 33, 34,
   8,  9, 10, 11, 12, 13, 33, 34,
  12, 13, 14, 15, 16, 17, 33, 34,
  16, 17, 18, 19, 20, 21, 33, 34,
  20, 21, 22, 23, 24, 25, 33, 34,
  24, 25, 26, 27, 28, 29, 33, 34,
  28, 29, 30, 31, 32,  1, 33, 34};

/*
** Permuted Choice 1
*/
char des_PC1[64] = {
  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,
   8, 16, 24, 32,
  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,
  40, 48, 56, 64};

/*
** Permuted Choice 2
*/
char des_PC2[64] = {
```

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

/*
** Key Rotation Schedule
*/
char des_keyrot[16] = { 1,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1 };

/*
** S-boxes
*/
char des_S[8][64] = {
  /*
  ** S1
  */
  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,
  /*
  ** S2
  */
  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,
  /*
  ** S3
  */
  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,
  /*
  ** S4
  */
   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,
  /*
  ** S5
  */
   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,
  /*
  ** S6
  */
  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,
  /*
  ** S7
  */
   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,
  /*
  ** S8
  */
  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};

/*
```

```
** 32-bit Permutation
*/
char des_P[32] = {
  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};

typedef unsigned long Long;

/*
** Des function prototypes
*/
Long Des_F();
int  Des_Bit32(),
     Des_Bit64();
void EnDes(),
     DeDes(),
     Des_Perm32(),
     Des_Perm64(),
     Des_Expand(),
     Des_GetKeySchedule();

/*
** End of des.h
**
*/
```

```
/*
/*
** Routine: des.c (Data Encryption Standard)
** (Some code has been inspired by Lawrence Brown, Computer Science Dept.
** UC, UNSW, Canberra, ACT 2600).
** Compile : cc -o des -DDES main.c des.c
** Dzung Le - Jul/92
*/
#include <stdio.h>

#include "des.h"

/*
** EnDes() : encrypt a 64-bit DES block with a key schedule,
** and a number of rounds.
*/
void EnDes(b,des_keyschedule,round)
Long b[2];
Long des_keyschedule[32][2];
int round;
{
  Long Temp, work[2];
  register int i;

  Des_Perm64(work, b, des_IP);

  for (i=0;i < round;i++) {
    Temp = work[0] ^ Des_F(work[1], des_keyschedule[i]);
    work[0] = work[1];
    work[1] = Temp;
  }

  Temp = work[0];
  work[0] = work[1];
  work[1] = Temp;

  Des_Perm64(b, work, des_FP);
}

/*
** DeDes() : decrypt a 64-bit DES block with a key schedule,
** and a number of rounds.
*/
void DeDes(b,des_keyschedule,round)
Long b[2];
Long des_keyschedule[32][2];
int round;
{
  Long Temp, work[2];
  register int i;

  Des_Perm64(work, b, des_IP);

  for (i=round-1;i >= 0; i--) {
    Temp = work[0] ^ Des_F(work[1], des_keyschedule[i]);
    work[0] = work[1];
    work[1] = Temp;
  }

  Temp = work[0];
  work[0] = work[1];
  work[1] = Temp;

  Des_Perm64(b, work, des_FP);
}

/*
** Des_F() : DES function F.
**
*/
Long Des_F(r,k)
Long r;
char *k;
{
```

```
  Long b = 0L, out = 0L;
  char a[8];
  register Long s;
  register Long rc;
  register int j;

  Des_Expand(a, r);

  for (j=0;j < 8;j++) {
    rc = a[j] ^ k[j];
    rc >>= 2;
    rc = (rc & 0x20) | ((rc << 4) & 0x10) | ((rc >> 1) & 0x0F);
    s = des_S[j][rc];
    b = (b << 4) | s;
  }

  Des_Perm32(&out, b, des_P);
  return out;
}

/*
** Des_Expand() : DES expand function.
**
*/
void Des_Expand(out,in)
char *out;
Long in;
{
  Long work[2], Lsp = (Long)out;

  work[0] = in;
  work[1] = 0L;
  Des_Perm64(Lsp,work,des_E);
}

/*
** Des_GetKeySchedule() : DES key initialization.
**
*/
void Des_GetKeySchedule(key,des_keyschedule,round)
char *key;
Long des_keyschedule[32][2];
int round;
{
  Long work1[2],work2[2];
  Long *LKey = (Long *)key;
  int i;

  Des_Perm64(work2,LKey,des_PC1);

  work2[0] &= MASK28;
  work2[1] &= MASK28;

  for (i=0;i < round;i++) {

    RotLeft28(work2[0], des_keyrot[i]);
    RotLeft28(work2[1], des_keyrot[i]);

    Des_Perm64(work1,work2,des_PC2);

    des_keyschedule[i][0] = work1[0];
    des_keyschedule[i][1] = work1[1];
  }
}

/*
** Des_Perm32() : DES 32-bit permutation function.
**
*/
void Des_Perm32(out,in,perm)
Long *out,in;
char *perm;
{
  Long mask = DESBIT1;
  register int i;
```

```
  *out = 0L;
  for (i=1;i <= 32;i++) {
    if (Des_Bit32(in,*perm++)) *out |= mask;
    mask >>= 1;
  }
}

/*
** Des_Perm64() : DES 64-bit permutation function.
**
*/
void Des_Perm64(out,in,perm)
Long *out,*in;
char *perm;
{
  register i;
  Long mask = DESBIT1;

  out[0] = out[1] = 0L;
  for (i=1;i <= 32; i++) {
    if (Des_Bit64(in,*perm++)) out[0] |= mask;
    mask >>= 1;
  }

  mask = DESBIT1;
  for (i=1;i <= 32; i++) {
    if (Des_Bit64(in,*perm++)) out[1] |= mask;
    mask >>= 1;
  }
}

/*
** Des_Bit32() : 32-bit test function.
**
*/
int Des_Bit32(n,pos)
Long n;
int pos;
{
  Long tmp = n;

  tmp <<= (pos-1);
  if ((tmp & DESBIT1) == DESBIT1) return 1;
  else return 0;
}

/*
** Des_Bit64() : 64-bit test function.
**
*/
int Des_Bit64(n,pos)
Long *n;
int pos;
{
  if (pos > 32) return Des_Bit32(n[1],pos-32);
  else return Des_Bit32(n[0],pos);
}

/*
** End of des.c
**
*/
```

```
/*
** Routine: feal.c
** Fast encryption algorithm
** Compile: cc -o feal -DFEAL main.c feal.c
** Dzung Le - Oct, 92
*/

#include <stdio.h>

typedef unsigned long Long;

#define  Feal_Rot2(n)      ((n << 2) | (n >> 6))
#define  Feal_S(x,y,delta)  (Feal_Rot2((char) (((int) x + y + delta) % 256)))

/*
** Function prototypes
*/
Long Feal_F(),
    Feal_Fk(),
    Feal_F(),
    Feal_Fk();
void EnFeal(),
    DeFeal(),
    Feal_GetKeySchedule();

/*
** EnFeal() : encrypt a 64-bit FEAL block with a key schedule,
** and a number of rounds.
*/
void EnFeal(work,feal_keyschedule,round)
Long *work, *feal_keyschedule;
int round;
{
  Long  k89, kab, kcd, kef;
  Long L,R,Temp;
  int i;

  /*
  ** working keys at begining and end processing
  */
  k89 = feal_keyschedule[(round / 2)];
  kab = feal_keyschedule[(round / 2) + 1];
  kcd = feal_keyschedule[(round / 2) + 2];
  kef = feal_keyschedule[(round / 2) + 3];

  work[0] ^= k89;
  work[1] ^= kab;

  L = work[0];
  R = work[1] ^ L;

  for (i=0;i < round;i++) {

    Temp = L;
    L = R;

    if ((i % 2) == 0)
      R = Temp ^ Feal_F(R, (feal_keyschedule[i / 2] >> 16));
    else
      R = Temp ^ Feal_F(R, (feal_keyschedule[i / 2] & 0xFFFF));
  }

  L ^= R;

  R ^= kcd;
  L ^= kef;

  work[0] = R;
  work[1] = L;
}

/*
** DeFeal() : decrypt a 64-bit FEAL block with a key schedule, and
** a number of rounds.
*/
```

```
void DeFeal(work,feal_keyschedule,round)
Long *work, *feal_keyschedule;
int round;
{
  Long k89, kab, kcd, kef;
  Long L,R,Temp;
  int i;

  /*
  ** working keys at begining and end processing
  */
  k89 = feal_keyschedule[(round / 2)];
  kab = feal_keyschedule[(round / 2) + 1];
  kcd = feal_keyschedule[(round / 2) + 2];
  kef = feal_keyschedule[(round / 2) + 3];

  work[0] ^= kcd;
  work[1] ^= kef;

  R = work[0];
  L = work[1] ^ R;

  for (i=round-1;i >= 0;i--) {

    Temp = R;
    R = L;

    if ((i % 2) == 0)
      L = Temp ^ Feal_F(R, (feal_keyschedule[i / 2] >> 16));
    else
      L = Temp ^ Feal_F(R, (feal_keyschedule[i / 2] & 0xFFFF));
  }

  R ^= L;

  L ^= k89;
  R ^= kab;

  work[0] = L;
  work[1] = R;
}

/*
** Feal_GetKeySchedule() : FEAL key initialization.
**
*/
void Feal_GetKeySchedule(key,feal_keyschedule,round)
char *key;
Long *feal_keyschedule,round;
{
  int i;
  Long *LKey = (Long *)key;
  /*
  ** 64-bit left part of the key
  */
  Long A = LKey[0], B = LKey[1];
  /*
  ** 64-bit right part of the key
  */
  Long KR1 = LKey[2], KR2 = LKey[2];

  Long D = (Long)0, QR = KR1 ^ KR2, Temp;

  for (i=0;i < 64;i++)
    feal_keyschedule[i] = (Long)0;

  for (i=0;i < (round * 2);i++) {

    if (i > 0) {
      D = A; A = B; B = Temp;
    }

    if ((i % 3) == 0)
      Temp = Feal_Fk( A, (B ^ QR) ^ D );

    if ((i % 3) == 1)
```

```
      Temp = Feal_Fk( A, (B ^ KR1) ^ D );

    if ((i % 3) == 2)
      Temp = Feal_Fk( A, (B ^ KR2) ^ D );

    feal_keyschedule[i] = Temp;
  }
}

/*
** Feal_F() : FEAL function f.
**
*/
Long Feal_F(a,b)
Long a,b;
{
  char f0, f1, f2, f3,
      a0 = (char) ((a >> 24) & 0xFF),
      a1 = (char) ((a >> 16) & 0xFF),
      a2 = (char) ((a >> 8) & 0xFF),
      a3 = (char) a & 0xFF,
      b0 = (char) ((b >> 8) & 0xFF),
      b1 = (char) b & 0xFF;

  Long retn=0L;

  f1 = a1 ^ b0 ^ a0;
  f2 = a2 ^ b1 ^ a3;

  f1 = Feal_S( f1, f2, 1);
  f2 = Feal_S( f2, f1, 0);
  f0 = Feal_S( a0, f1, 0);
  f3 = Feal_S( a3, f2, 1);

  retn |= f0;
  retn <<= 8;
  retn |= f1;
  retn <<= 8;
  retn |= f2;
  retn <<= 8;
  retn |= f3;

  return retn;
}


/*
** Feal_Fk() : FEAL function fk.
**
*/
Long Feal_Fk(a,b)
Long a,b;
{
  char fk0, fk1, fk2, fk3,
      a0 = (char) ((a >> 24) & 0xFF),
      a1 = (char) ((a >> 16) & 0xFF),
      a2 = (char) ((a >> 8) & 0xFF),
      a3 = (char) a & 0xFF,
      b0 = (char) ((b >> 24) & 0xFF),
      b1 = (char) ((b >> 16) & 0xFF),
      b2 = (char) ((b >> 8) & 0xFF),
      b3 = (char) b & 0xFF;

  Long retn=0L;

  fk1 = a1 ^ a0;
  fk2 = a2 ^ a3;

  fk1 = Feal_S( fk1, fk2 ^ b0, 1);
  fk2 = Feal_S( fk2, fk1 ^ b1, 0);
  fk0 = Feal_S( a0,  fk1 ^ b2, 0);
  fk3 = Feal_S( a3,  fk2 ^ b3, 1);

  retn |= fk0;
  retn <<= 8;
  retn |= fk1;
```

```
  retn <<= 8;
  retn |= fk2;
  retn <<= 8;
  retn |= fk3;

  return retn;
}

/*
**
** End of feal.c
*/
```

```
/*
** Include file: loki.h
** Loki Encryption Algorithm
** Dzung Le - Oct, 92
**
*/
#define LOKIBIT1 0x80000000L
#define MASK28  0xFFFFFFF0L
#define Erow    31

#define ECB     0
#define CBC     1
#define CFB     2
#define OFB     3

#define Loki_RotLeft(n,pos)  ((n << pos) | (n >> (32 - pos)))
#define Loki_RotRight(n,pos) ((n << (32 - pos)) | (n >> pos))

typedef unsigned long Long;

/*
**
** Global variables
*/

char loki_P[32] = {
  31, 23, 15,  7, 30, 22, 14,  6,
  29, 21, 13,  5, 28, 20, 12,  4,
  27, 19, 11,  3, 26, 18, 10,  2,
  25, 17,  9,  1, 24, 16,  8, 0};

int loki_Sbox[16] = { 375, 379, 391, 395, 397, 415, 419, 425,
            433, 445, 451, 463, 471, 477, 487, 499};

/*
** Function prototypes
**
*/
void Loki_Expand(),
    Loki_GetKeySchedule(),
    loki_swap(),
    EnLoki(),
    DeLoki(),
    loki_do_round(),
    loki_do_decrypt(),
    loki_do_encrypt();

int  Loki_Bit32(),
    Loki_Bit16(),
    Loki_Bit64();

Long loki_S(),
    loki_perm32(),
    loki_F();
/*
** End of loki.h
**
*/
```

```
/*
** Routine: loki.c
** Loki encryption algorithm
** Compile: cc -o loki -DLOKI main.c loki.c
** Dzung Le - Oct, 92
*/

#include <stdio.h>

#include "loki.h"

#define Loki_RotLeft(n,pos)  ((n << pos) | (n >> (32 - pos)))
#define Loki_RotRight(n,pos) ((n << (32 - pos)) | (n >> pos))

/*
** EnLoki() : encrypt a 64-bit block with extended keys.
**
*/
void EnLoki(work,loki_keyschedule,round)
Long *work,*loki_keyschedule;
int round;
{
  Long L = work[0], R = work[1], Temp;
  int i;

  for (i=0;i < round;i++) {

    Temp = L;
    L = R;
    R = Temp ^ loki_F( R, loki_keyschedule[i]);

    work[0] = L;
    work[1] = R;
  }

  Temp = work[0];
  work[0] = work[1];
  work[1] = Temp;
}

/*
** DeLoki() : decrypt a 64-bit block with extended keys.
**
*/
void DeLoki(work,loki_keyschedule,round)
Long *work,*loki_keyschedule;
int round;
{
  int i;
  Long L = work[0], R = work[1], Temp;

  for (i=round-1;i >= 0;i--) {

    Temp = L;
    L = R;
    R = Temp ^ loki_F( R, loki_keyschedule[i]);

    work[0] = L;
    work[1] = R;
  }

  Temp = work[0];
  work[0] = work[1];
  work[1] = Temp;
}

/*
** loki_F() : loki function f.
**
*/
Long loki_F(r,k)
Long r,k;
{
  Long Temp;
  int E[4];
```

```
  Temp = r ^ k;

  Loki_Expand(E,Temp);

  return loki_perm32(loki_S(E));
}

/*
** loki_S() : loki S box.
**
*/
Long loki_S(e)
int *e;
{
  int i, row, col, temp1, temp2;
  Long retn=0L;

  for (i=0;i < 4;i++) {

    row = ((e[i] & 0x3) | ((e[i] >> 8) & 0xC));
    col = ((e[i] >> 2) & 0xFF);

    temp1 = (row * 17) ^ 0xFF;

    temp2 = (col + temp1) ^ 0xFF;

    retn |= (Long) (temp2 & 0xFF);

    if (i < 3) retn <<= 8;
  }

  return retn;
}

/*
** Loki_GetKeySchedule() : key initialization.
**
*/
void Loki_GetKeySchedule(key,loki_keyschedule,round)
Long *key,*loki_keyschedule;
int round;
{
  Long KL = key[0], KR = key[1];
  Long Temp;
  int i;

  for (i=0;i < round;i++) {

    if ((i % 2) == 0) { /* even */
      loki_keyschedule[i] = KL;
      Temp   = loki_keyschedule[i];
      KL = Temp;
      Loki_RotLeft(KL, 13);
    }
    else { /* even */
      loki_keyschedule[i] = KL;
      Temp = loki_keyschedule[i];
      KL   = KR;
      KR = Temp;
      Loki_RotLeft(KR, 12);
    }
  }
}

/*
** loki_perm32() : 32-bit permutation.
**
*/
Long loki_perm32(in)
Long in;
{
  Long out, mask = LOKIBIT1;
  register int i;
  char *perm = loki_P;
```

```
  out = 0L;
  for (i=31;i >= 0;i--) {
    if (Loki_Bit32(in,*perm++)) out |= mask;
    mask >>= 1;
  }

  return out;
}

/*
** Loki_Expand() :
**
*/
void Loki_Expand(out,in)  /* 12 least significant bits of 4 integers */
Long *out;
Long in;
{
  *out = *(out +1) = *(out + 2) = *(out + 3) = 0;
  *out      = (int)(((in & 0xFF000000L) >> 24) | ((in & 0xFL) << 8));
  *(out + 1) = (int)((in & 0xFFF0000L) >> 16);
  *(out + 2) = (int)((in & 0xFFF00L) >> 8);
  *(out + 3) = (int)(in & 0xFFFL);
}

/*
**
** Loki_Bit32() : return bit position pos
*/
int Loki_Bit32(n,pos)
Long n;
int pos; /* 31 30  .... 1 0 */
{
  return (int)((Long)((n >> pos) & 1L));
}

/*
**
** End of loki.c
*/
```

```
/*
** Routine: khufu.c
** Khufu encryption algorithm
** Compile: cc -o khufu -DKHUFU main.c khufu.c
** Dzung Le - Oct, 92
*/
#include <stdio.h>

#define RotRight(W, s)   ((W >> s) | (W << (32 - s)))
#define RotLeft(W, s)    ((W << s) | (W >> (32 - s)))

typedef unsigned long Long;

/*
** Rotate schedule
*/
int RotSchedule[8] = {16, 16, 8, 8, 16, 16, 24, 24};

/*
** Initial auxialiary keys taken from the fraction of Pi
*/
Long auxkey[4] = {0xEC4E6C89, 0x082EFA98, 0x299F31D0, 0xA4093822};

void EnKhufu(),
   DeKhufu(),
   InitialSboxSchedule(),
   SboxSchedule(),
   dump32hex();

/*
**
** EnKhufu() : encrypt a 64-bit Khufu block with Sbox schedule
*/
void EnKhufu(work, Sbox, round)
Long *work, **Sbox;
int round;
{
  Long L, R, Temp;
  int i, octet;

  L = work[0];
  R = work[1];

  L ^= auxkey[0];
  R ^= auxkey[1];

  octet = 0;

  for (i=0;i < round;i++) {

    R ^= Sbox[octet][(int)L & 0xFF];
    L = (Long) RotRight(L,RotSchedule[i % 8]);

    Temp = L;
    L = R;
    R = Temp;

    if (((i + 1) % 8) == 0)
      octet++;
  }

  L ^= auxkey[2];
  R ^= auxkey[3];

  work[0] = L;
  work[1] = R;
}

/*
**
** DeKhufu() : decrypt a 64-bit Khufu block with Sbox schedule
*/
void DeKhufu(work, Sbox, round)
Long *work, **Sbox;
int round;
```

```
{
  Long L, R, Temp;
  int i, octet;

  L = work[0];
  R = work[1];

  L ^= auxkey[2];
  R ^= auxkey[3];

  octet = round / 8;

  for (i=round-1;i >= 0;i--) {

    if (((i + 1) % 8) == 0)
      octet--;

    R = (Long) RotLeft(R,RotSchedule[i % 8]);
    L ^= Sbox[octet][(int)R & 0xFF];

    Temp = L;
    L = R;
    R = Temp;
  }

  L ^= auxkey[0];
  R ^= auxkey[1];

  work[0] = L;
  work[1] = R;
}

/*
**
** InitialSboxSchedule() : Initialize the initial S box
*/
void InitialSboxSchedule(Sbox)
Long Sbox[4][256];
{
  int i, j;

  /*
  ** Initial S box
  */
  for (i=0;i < 4;i++) {
    for (j=0;j < 256;j++) {
      Sbox[i][j] = (Long)((i+7) * (j+17)) % 256;
    }
  }
}

/*
**
** SboxSchedule() : create the S box from the initial S box,
** and the user-supplied key.
*/
void SboxSchedule(InitialSbox,Sbox,key)
Long InitialSbox[4][256];
Long Sbox[4][256];
char *key;
{
  int i, j;
  /*
  ** Store 4 * 256 random numbers between 0 and 255 inclusive
  */
  int SRand[1024];
  Long Lkey[2];
  /*
  ** initial vector for cipher block chaining
  */
  Long iv[2];

  /*
  ** Initial S box
  */
  for (i=0;i < 4;i++) {
```

```
  for (j=0;j < 256;j++) {
    Sbox[i][j] = (Long)((i+7) * (j+17)) % 256;
  }
}

/*
** key initialization
*/
for (i=1;i < 8;i++)
   ((char *)Lkey)[i] = i;

for (i=0;i < strlen(key);i++)
   ((char *)Lkey)[i] = key[i];

/*
** Generate Sbox which is enough for 32 rounds,
** using EnKhufu routine, initial S box and the user-supplied key.
** Cipher block chaining mode, round = 32.
*/
iv[0] = iv[1] = (Long)0;
for (i=0;i < 256;i++) {

   Lkey[0] ^= iv[0];
   Lkey[1] ^= iv[1];

   EnKhufu(Lkey, InitialSbox, 32);

   iv[0] = Lkey[0];
   iv[1] = Lkey[1];

   Sbox[0][2*i     ] = ((unsigned char *)Lkey)[0];
   Sbox[1][2*i     ] = ((unsigned char *)Lkey)[1];
   Sbox[2][2*i     ] = ((unsigned char *)Lkey)[2];
   Sbox[3][2*i     ] = ((unsigned char *)Lkey)[3];
   Sbox[0][2*i + 128] = ((unsigned char *)Lkey)[4];
   Sbox[1][2*i + 128] = ((unsigned char *)Lkey)[5];
   Sbox[2][2*i + 128] = ((unsigned char *)Lkey)[6];
   Sbox[3][2*i + 128] = ((unsigned char *)Lkey)[7];
 }

 /*
 ** Reset the auxiliary keys
 */
 Lkey[0] ^= iv[0];
 Lkey[1] ^= iv[1];

 EnKhufu(Lkey, Sbox, 32);

 auxkey[0] = ((unsigned char *)Lkey)[7];
 auxkey[1] = ((unsigned char *)Lkey)[5];
 auxkey[2] = ((unsigned char *)Lkey)[3];
 auxkey[3] = ((unsigned char *)Lkey)[1];

}

/*
** End of khufu.c
**
*/
```

```
/*
** Routine: khafre.c
** Khafre encryption algorithm
** Compile: cc -o khafre -DKHAFRE main.c khafre.c
** Dzung Le - Oct, 92
*/

#include <stdio.h>

#define RotRight(W, s)   ((W >> s) | (W << (32 - s)))
#define RotLeft(W, s)    ((W << s) | (W >> (32 - s)))

typedef unsigned long Long;

/*
** Rotate schedule
*/
int KhafreRotSchedule[8] = {16, 16, 8, 8, 16, 16, 24, 24};

void EnKhafre(),
   DeKhafre(),
   KhafreSboxSchedule();

/*
**
** EnKhafre() : encrypt a 64-bit Khafre block with t
** standard S box.
*/
void EnKhafre(work, Sbox, key, round)
Long work[2], Sbox[4][256], key[2][2];
int round;
{
  Long L, R, Temp;
  int i, octet;
  int keysize=2, keyind;

  L = work[0];
  R = work[1];

  L ^= key[0][0];
  R ^= key[0][1];

  keyind = 0;

  octet = (round / 8);

  octet = 0;

  for (i=0;i < round;i++) {

    R ^= Sbox[octet][(int)L & 0xFF];
    L = (Long) RotRight(L,KhafreRotSchedule[i % 8]);

    Temp = L;
    L = R;
    R = Temp;

    if (((i + 1) % 8) == 0) {
      L ^= RotRight(key[keyind][0], octet);
      R ^= RotRight(key[keyind][1], octet);
      keyind++;
      if (keyind == keysize)
        keyind = 0;

      octet++;
    }
  }

  work[0] = L;
  work[1] = R;
}

/*
**
** DeKhafre() : decrypt a 64-bit Khafre block with t
```

```
** standard S box.
*/
void DeKhafre(work, Sbox, key, round)
Long work[2], Sbox[4][256], key[2][2];
int round;
{
  Long L, R, Temp;
  int i, octet;
  Long KTemp[2];
  int keysize=2, keyind;

  L = work[0];
  R = work[1];

  switch (round) {
    case  8: keyind = 0;
           break;
    case 16: keyind = 1;
           break;
    case 24: keyind = 0;
           break;
    case 32: keyind = 1;
           break;
  }

  octet = (round / 8);

  for (i=round-1;i >= 0;i--) {

    if (((i + 1) % 8) == 0) {

      octet--;

      L ^= RotRight(key[keyind][0], octet);
      R ^= RotRight(key[keyind][1], octet);
      keyind++;
      if (keyind == keysize)
        keyind = 0;
    }

    R = (Long) RotLeft(R,KhafreRotSchedule[i % 8]);
    L ^= Sbox[octet][(int)R & 0xFF];

    Temp = L;
    L = R;
    R = Temp;
  }

  L ^= key[0][0];
  R ^= key[0][1];

  work[0] = L;
  work[1] = R;
}

/*
**
** KhafreSboxSchedule() : intialialize the S box
*/
void KhafreSboxSchedule(Sbox)
Long Sbox[4][256];
{
  int i, j;

  for (i=0;i < 4;i++) {
    for (j=0;j < 256;j++) {
      Sbox[i][j] = (Long)((i+7) * (j+17)) % 256;
    }
  }
}

/*
** End of khafre.c
**
*/
```

```
/*
** Program : sbh.c
** Single block hash using 64-bit block cipher symmetric algorithms -
** des, feal, loki, khufu, khafre.
** Compile : cc -o sbh sbh.c des.o feal.o loki.o khufu.o khafre.o
** Dzung Le - Oct, 92
**
*/
#include <stdio.h>

/*
** Encryption mode
*/
#define CBC    1  /* Cipher block chaining */
#define SBH    2  /* Single block hash */

typedef unsigned long Long;

/*
** Function prototypes:
** Feal encryption algorithm
*/
void Feal_GetKeySchedule(), EnFeal(), DeFeal(),

/*
** Des encryption standard algorithm
*/
EnDes(), DeDes(), Des_GetKeySchedule(),

/*
** Loki encryption standard algorithm
*/
EnLoki(), DeLoki(), Loki_GetKeySchedule(),

/*
** Khufu encryption algorithm
*/
InitialSboxSchedule(), SboxSchedule(), EnKhufu(), DeKhufu(),

/*
** Khafre encryption algorithm
*/
KhafreSboxSchedule(), EnKhafre(), DeKhafre(),

/*
** General hashing routines
*/
PrintWord(), PrintFormat(), Do_CBC_hash(), Do_SBH_hash();

/*
** Global variables:
** Initial vector, command name,
** and default values
*/
Long Digest[2] = {(Long)0, (Long)0};
Long Mesg[2]   = {(Long)0, (Long)0};
char *hashtype = "des";
int  mode = SBH;
int  round = 8;
char *Instring;
int  sflag=0;
FILE *fpi;

/*
** Main program
**
*/
main(argc,argv)
int argc;
char **argv;
{
  int c,Fflag,aflag,rflag,fflag,mflag,kflag,errflag;
  extern char *optarg;
  char filename[160];
  char tkey[20],*key=tkey;
```

```
char *Format;

/*
** Option processing
*/
Fflag=aflag=rflag=fflag=mflag=kflag=errflag= 0;

while ((c=getopt(argc,argv,"F:f:k:r:m:a:s:")) != EOF) {
  switch (c) {
    case 'f': fflag++;
              strcpy(filename,optarg);
              break;
    case 's': sflag++;
              Instring = optarg;
              break;
    case 'k': kflag++;
              key = optarg;
              break;
    case 'F': Fflag++;
              Format = optarg;
              break;
    case 'm': mflag++;
              if ((strcmp(optarg,"cbc")) && (strcmp(optarg,"sbh"))) {
                fprintf(stderr,"%s: invalid hashing mode - ", argv[0]);
                fprintf(stderr,"cbc, or sbh only.\n\n");
                exit(1);
              }
              else {
                if (!strcmp(optarg,"cbc")) mode = CBC;
                if (!strcmp(optarg,"sbh")) mode = SBH;
              }
              break;
    case 'a': aflag++;
              if ((!strcmp(optarg,"des")) || (!strcmp(optarg,"feal")) ||
                 (!strcmp(optarg,"loki")) || (!strcmp(optarg,"khufu")) ||
                 (!strcmp(optarg,"khafre"))
                ) {
                hashtype = optarg;
              }
              else {
                fprintf(stderr,"\n%s: invalid hash algorithm\n",argv[0]);
                fprintf(stderr,"       (des, feal, loki, khufu, ");
                fprintf(stderr,"and khafre only)\n\n");
                exit(1);
              }
              break;
    case 'r': if (((round = atoi(optarg)) == 0) ||
                 (round > 32) || (round < 4))
                errflag++;
              break;
    default:  errflag++;
  }
}

/*
** Error checking
*/
if (errflag) {
  fprintf(stderr,"\nUsage: %s -f file | -s string ", argv[0]);
  fprintf(stderr,"[ -k key -m mode -r round -a algorithm -F format ]\n\n");
  exit(1);
}

if (!kflag) {
  key = (char *)getpass("Key: ");
}
else if (strlen(key) < 5) {
  fprintf(stderr,"\n%s: key length is too short (min: 5 chars).\n\n",
    argv[0]);
  exit(1);
}

if (fflag) {
  if ((fpi = fopen(filename, "r")) == NULL) {
    fprintf(stderr,"Couldn't open input file %s!\n", filename);
    exit(1);
```

```
    }
  }
  else if (!sflag)
    fpi = stdin;

  /*
  ** Process message digest
  */
  if (mode == CBC)
    Do_CBC_hash(Digest,key,round);

  if (mode == SBH)
    Do_SBH_hash(Digest,key,round);

  /*
  ** Print out the digest
  */
  if (!Fflag) {
    PrintWord(Digest[0]); PrintWord(Digest[1]);
    fprintf(stdout,"\n");
  }
  else {
    PrintFormat(Format, Digest, hashtype, round, mode);
  }
}

/*
** Do_CBC_hash() : Cipher block chaining hash for a file or a string with
** key schedule, and a number of rounds.
*/
void Do_CBC_hash(Digest,key,round)
Long *Digest;
char *key;
int round;
{
  int i, cnt=0, done=0, count=0;
  Long work[2];
  unsigned char *cptr = (char *)Digest;
  /*
  ** feal
  */
  char ekey[17];
  Long feal_keyschedule[64];

  /*
  ** des
  */
  char dkey[9];
  Long des_keyschedule[32][2];

  /*
  ** loki
  */
  char lkey[9];
  Long loki_keyschedule[64];

  /*
  ** khufu
  */
  Long InitialSbox[4][256];
  Long Sbox[4][256];

  /*
  ** Khafre
  */
  char Tkey[17];
  Long **Lkey = (Long **)Tkey;
  Long StandardSBox[4][256];

  if (!strcmp(hashtype,"feal")) {
    /*
    ** feal: default key initialisation (128 bits)
    */
    for (i=0;i < 16;i++) ekey[i] = 0;

    if (strlen(key) > 16)
```

```
    *(key + 16) = '\0';

  for (i=0;i < strlen(key);i++)
    ekey[i] = key[i];
  /*
  ** create key schedule for feal
  */
  Feal_GetKeySchedule(ekey,feal_keyschedule,round);
}

if (!strcmp(hashtype,"des")) {
  /*
  ** des: default key initialisation (64 bits)
  */
  for (i=0;i < 8;i++) dkey[i] = 0;

  if (strlen(key) > 8)
    *(key + 8) = '\0';

  for (i=0;i < strlen(key);i++)
    dkey[i] = key[i];
  /*
  ** create key schedule for des
  */
  Des_GetKeySchedule(dkey, des_keyschedule, round);
}

if (!strcmp(hashtype,"loki")) {
  /*
  ** loki: default key initialisation (64 bits)
  */
  for (i=0;i < 8;i++) lkey[i] = 0;

  if (strlen(key) > 8)
    *(key + 8) = '\0';

  for (i=0;i < strlen(key);i++)
    lkey[i] = key[i];
  /*
  ** create key schedule for loki
  */
  Loki_GetKeySchedule(lkey, loki_keyschedule, round);
}

if (!strcmp(hashtype,"khufu")) {
  InitialSboxSchedule(Sbox);
  InitialSboxSchedule(Sbox,InitialSbox,key);
}

if (!strcmp(hashtype,"khafre")) {
  /*
  ** khafre: default key initialisation (128 bits)
  */
  for (i=0;i < 16;i++)
    Tkey[i] = i + 11;

  if (strlen(key) > 16)
    *(key + 16) = '\0';

  for (i=0;i < strlen(key);i++)
    Tkey[i] = key[i];

  KhafreSboxSchedule(StandardSBox);
}

work[0] = work[1] = 0L;

do {
  if (!sflag) {
    if ((cnt = fread((char *)work, 1, 8, fpi)) != 8) {
      ((char *)work)[7] = cnt;
      done++;
    }
  }
  else if (*Instring) {
    if (strlen(Instring) >= 8) {
```

```
        strncpy((char *)work, Instring, 8);
        Instring += 8;
      }
      else {
        strcpy((char *)work, Instring);
        ((char *)work)[7] = strlen(Instring);
        done++;
      }
    }
    else done++;

    /*
    ** Addition modulo 2 for all 64-bit blocks of plaintext
    */
    Mesg[0] ^= work[0];
    Mesg[1] ^= work[1];

    /*
    ** Chain with the previous ciphertext
    */
    work[0] ^= Digest[0];
    work[1] ^= Digest[1];

    if (!strcmp(hashtype,"feal")) {
      EnFeal(work,feal_keyschedule,round);
    }

    if (!strcmp(hashtype,"des")) {
      EnDes(work,des_keyschedule,round);
    }

    if (!strcmp(hashtype,"loki")) {
      EnLoki(work,loki_keyschedule,round);
    }

    if (!strcmp(hashtype,"khufu")) {
      EnKhufu(work, Sbox, round);
    }

    if (!strcmp(hashtype,"khafre")) {
      EnKhafre(work, StandardSBox, Lkey,round);
    }

    Digest[0] = work[0];
    Digest[1] = work[1];

  } while (!done);

  /*
  ** Final digests
  */
  Digest[0] ^= Mesg[0];
  Digest[1] ^= Mesg[1];

}

/*
** Do_SBH_hash() : Single-block hashing a file or a string
*/
void Do_SBH_hash(Digest,key,round)
Long *Digest;
char *key;
int round;
{
  int i, cnt=0, done=0, count=0;
  Long work[2];
  unsigned char *cptr = (char *)Digest;
  /*
  ** feal
  */
  char ekey[17];
  Long feal_keyschedule[64];

  /*
  ** des
  */
```

```
char dkey[9];
Long des_keyschedule[32][2];

/*
** loki
*/
char lkey[9];
Long loki_keyschedule[64];

/*
** khufu
*/
Long InitialSbox[4][256];
Long Sbox[4][256];

/*
** Khafre
*/
char Tkey[17];
Long **Lkey = (Long **)Tkey;
Long StandardSBox[4][256];

/*
** Initialize the digest with key value
*/
for (i=0;i < strlen(key);i++)
  if (i < 8)
    ((char *)Digest)[i] = key[i];

do {
  work[0] = work[1] = 0L;

  /*
  ** Read plaintext into work[2]
  */
  if (!sflag) {
    if ((cnt = fread((char *)work, 1, 8, fpi)) != 8) {
      ((char *)work)[7] = cnt;
      done++;
    }
  }
  else if (*Instring) {
    if (strlen(Instring) >= 8) {
      strncpy((char *)work, Instring, 8);
      Instring += 8;
    }
    else {
      strcpy((char *)work, Instring);
      ((char *)work)[7] = strlen(Instring);
      done++;
    }
  }
  else done++;

  /*
  ** Keep the plaintext for later addition modulo 2
  */
  Mesg[0] = work[0];
  Mesg[1] = work[1];

  /*
  ** Encrypt the digest with key value in work[2]
  */
  if (!strcmp(hashtype,"feal")) {

    for (i=0;i < 8;i++)
      ekey[i] = ((char *)work)[i];
    /*
    ** create key schedule for feal
    */
    Feal_GetKeySchedule(ekey,feal_keyschedule,round);

    EnFeal(Digest,feal_keyschedule,round);
  }

  if (!strcmp(hashtype,"des")) {
```

```
      for (i=0;i < 8;i++)
        dkey[i] = ((char *)work)[i];
      /*
      ** create key schedule for des
      */
      Des_GetKeySchedule(dkey, des_keyschedule, round);

      EnDes(Digest,des_keyschedule,round);
    }

    if (!strcmp(hashtype,"loki")) {

      for (i=0;i < 8;i++)
        lkey[i] = ((char *)work)[i];
      /*
      ** create key schedule for loki
      */
      Loki_GetKeySchedule(lkey, loki_keyschedule, round);

      EnLoki(Digest,loki_keyschedule,round);
    }

    if (!strcmp(hashtype,"khufu")) {
      for (i=0;i < 8;i++)
        Tkey[i] = ((char *)work)[i];

      InitialSboxSchedule(Sbox);
      InitialSboxSchedule(Sbox,InitialSbox,Tkey);

      EnKhufu(Digest, Sbox, round);
    }

    if (!strcmp(hashtype,"khafre")) {

      for (i=0;i < strlen(key);i++)
        Tkey[i] = ((char *)work)[i];

      KhafreSboxSchedule(StandardSBox);

      EnKhafre(Digest, StandardSBox, Lkey,round);
    }

    /*
    ** Addition modulo 2 the new ciphertext with the old plaintext
    */
    Digest[0] ^= Mesg[0];
    Digest[1] ^= Mesg[1];

  } while (!done);

}

/*
** PrintWord() : print a 32-bit word in hexadecimal format.
**
*/
void PrintWord(Word)
Long Word;
{
  fprintf(stdout,"%02x%02x%02x%02x",
    (unsigned char)((Word >> 24) & 0xFF),
    (unsigned char)((Word >> 16) & 0xFF),
    (unsigned char)((Word >>  8) & 0xFF),
    (unsigned char)( Word & 0xFF));
}

/*
**
** Print out the digest with the user-supplied format
*/
void PrintFormat(Format,Digest,alg,round,mode)
char *Format,*alg;
int round, mode;
Long *Digest;
{
```

```
  int i=0;

  while (i < strlen(Format)) {
    if (Format[i] != '%') {
      fprintf(stdout,"%c", Format[i]);
      i++;
    }
    else {
      switch (Format[i+1]) {
        case 'h': PrintWord(Digest[0]);
                  PrintWord(Digest[1]);
                  break;
        case 'a': fprintf(stdout,"%s", alg);
                  break;
        case 'r': fprintf(stdout,"%02d", round);
                  break;
        case 'm': if (mode == CBC)
                    fprintf(stdout,"cbc");
                  if (mode == SBH)
                    fprintf(stdout,"sbh");
                  break;
        case 'n': fprintf(stdout,"\n");
                  break;
        case 't': fprintf(stdout,"\t");
                  break;
        case '%': fprintf(stdout,"%%");
                  break;
      }
      i += 2;
    }
  }
  fprintf(stdout,"\n");
}

/*
** End of sbh.c
**
*/
```

```
/*
** Program : dbh.c
** Double block hash using 64-bit encryption algorithms -
** des, feal, loki, khufu, khafre.
** Compile : cc -o dbh dbh.c des.o feal.o loki.o khufu.o khafre.o
** Dzung Le - Oct, 92
**
*/
#include <stdio.h>

typedef unsigned long Long;

/*
** Function prototypes:
** Feal encryption algorithm
*/
void Feal_GetKeySchedule(), EnFeal(), DeFeal(),

/*
** Des encryption standard algorithm
*/
EnDes(), DeDes(), Des_GetKeySchedule(),

/*
** Loki encryption standard algorithm
*/
EnLoki(), DeLoki(), Loki_GetKeySchedule(),

/*
** Khufu encryption algorithm
*/
InitialSboxSchedule(), SboxSchedule(), EnKhufu(), DeKhufu(),

/*
** Khafre encryption algorithm
*/
KhafreSboxSchedule(), EnKhafre(), DeKhafre(),

/*
** General hashing routines
*/
Update(),PrintWord(), PrintFormat(), Do_DBH_Hash();

/*
** Global variables:
** Initial vector, command name,
** and default input, output file pointers
*/
Long Digest1[2] = {(Long)0, (Long)0};
Long Digest2[2] = {(Long)0, (Long)0};
char *hashtype = "des";
char *Instring;
int sflag=0;
FILE *fpi;

/*
** Main program
**
*/
main(argc,argv)
int argc;
char **argv;
{
  int c,Fflag,aflag,rflag,fflag,kflag,errflag;
  extern char *optarg;
  char filename[160];
  int round = 8;
  char tkey[20],*key=tkey;
  char *Format;

  /*
  ** Option processing
  */
  Fflag=aflag=rflag=fflag=kflag=errflag= 0;
```

```
while ((c=getopt(argc,argv,"F:f:k:r:a:s:")) != EOF) {
  switch (c) {
    case 'f': fflag++;
              strcpy(filename,optarg);
              break;
    case 's': sflag++;
              Instring = optarg;
              break;
    case 'k': kflag++;
              key = optarg;
              break;
    case 'F': Fflag++;
              Format = optarg;
              break;
    case 'a': aflag++;
              if ((!strcmp(optarg,"des")) || (!strcmp(optarg,"feal")) ||
                (!strcmp(optarg,"loki")) || (!strcmp(optarg,"khufu")) ||
                (!strcmp(optarg,"khafre"))
                ) {
                hashtype = optarg;
              }
              else {
                fprintf(stderr,"\n%s: invalid hash algorithm\n",argv[0]);
                fprintf(stderr,"      (des, feal, loki, khufu, ");
                fprintf(stderr,"and khafre only)\n\n");
                exit(1);
              }
              break;
    case 'r': if (((round = atoi(optarg)) == 0) ||
                (round > 32) || (round < 4))
                errflag++;
              break;
    default:  errflag++;
  }
}

/*
** Error checking
*/
if (errflag) {
  fprintf(stderr,"\nUsage: %s -f file | -s string ", argv[0]);
  fprintf(stderr,"[ -k key -m mode -r round -a algorithm -F Format ]\n\n");
  exit(1);
}

if (!kflag) {
  key = (char *)getpass("Key: ");
}
else if (strlen(key) < 5) {
  fprintf(stderr,"\n%s: key length is too short (min: 5 chars).\n\n",
    argv[0]);
  exit(1);
}

if (fflag) {
  if ((fpi = fopen(filename, "r")) == NULL) {
    fprintf(stderr,"Couldn't open input file %s!\n", filename);
    exit(1);
  }
}
else fpi = stdin;

/*
** Processing message digest
*/
Do_DBH_Hash(Digest1,Digest2,key,round);

/*
** Print out the final digest (128 bits) without format.
*/
if (!Fflag) {
  PrintWord(Digest1[0]); PrintWord(Digest1[1]);
  PrintWord(Digest2[0]); PrintWord(Digest2[1]);
  fprintf(stdout,"\n");
}
else {
```

```
    /*
    ** Print out the digests with the user-supplied format.
    */
    PrintFormat(Format, Digest1, Digest2, hashtype, round);
  }
}

/*
** Do_DBH_Hash() : Double block hashing for a file or a string with a mode,
** key schedule, and a number of rounds.
**
*/
void Do_DBH_Hash(Digest1,Digest2,key,round)
Long *Digest1,*Digest2;
char *key;
int round;
{
  Long key1[2], key2[2];
  int i, done=0, cnt1=0, cnt2=0;
  Long M1[2], M2[2];

  /*
  ** Initialise the digests with key value
  */
  if (strlen(key) > 16) key[16] = '\0';

  for (i=0;i < 8;i++)
    if (key) ((char *)Digest1)[i] = *key++;

  for (i=0;i < 8;i++)
    if (key) ((char *)Digest2)[i] = *key++;

  M1[0] = M1[1] = M2[0] = M2[1] = (Long)0;

  do {
    key1[0] = key1[1] = 0L;
    key2[0] = key2[1] = 0L;

    if (!sflag) {
      if ((cnt1 = fread((char *)key1, 1, 8, fpi)) != 8) done++;
      if (!done)
        if ((cnt2 = fread((char *)key2, 1, 8, fpi)) != 8) done++;
    } else if (*Instring) {
        if (strlen(Instring) >= 8) {
          strncpy((char *)key1, Instring, 8);
          Instring += 8;
        }
        else {
          strcpy((char *)key1, Instring);
          *Instring = '\0';
          done++;
        }

        if (!done) {
          if (strlen(Instring) >= 8) {
            strncpy((char *)key2, Instring, 8);
            Instring += 8;
          }
          else {
            strcpy((char *)key2, Instring);
            *Instring = '\0';
            done++;
          }
        }
    }
    else done++;

    /*
    ** Addition modulo 2 of all plaintext block
    */
    M1[0] ^= key1[0]; M1[1] ^= key1[1];
    M1[0] ^= key2[0]; M1[1] ^= key2[1];

    M2[0] += key1[0]; M1[1] += key1[1];
    M2[0] += key2[0]; M2[1] += key2[1];
```

```
    /*
    ** Encrypt digests with key being plaintext
    */
    Update(key1, key2, Digest1, Digest2, round);

    /*
    ** Addition modulo 2 of the cipher text with the previous plaintext
    */
    Digest1[0] ^= key1[0]; Digest1[1] ^= key1[1];
    Digest2[0] ^= key2[0]; Digest2[1] ^= key2[1];

  } while (!done);

  /*
  ** Process two additional blocks
  */
  Update(M1, M2, Digest1, Digest2, round);
}


/*
** Update() : Update message digests Digest1, Digest2 with key1 and key2.
**
*/
void Update(key1, key2, Digest1, Digest2, round)
Long *key1, *key2, *Digest1, *Digest2;
int round;
{
  Long T1[2], T2[2], Temp1[2], Temp2[2], Temp3[2];
  int i;

  /*
  ** feal
  */
  char ekey1[17],ekey2[17];
  Long feal_keyschedule1[64],feal_keyschedule2[64];

  /*
  ** des
  */
  char dkey1[9],dkey2[9];
  Long des_keyschedule1[32][2],des_keyschedule2[32][2];

  /*
  ** loki
  */
  char lkey1[9],lkey2[9];
  Long loki_keyschedule1[64],loki_keyschedule2[64];

  /*
  ** khufu
  */
  Long InitialSbox1[4][256],InitialSbox2[4][256];
  Long Sbox1[4][256],Sbox2[4][256];

  /*
  ** Khafre
  */
  char Tkey1[17],Tkey2[17];
  Long **Lkey1 = (Long **)Tkey1;
  Long **Lkey2 = (Long **)Tkey2;
  Long StandardSBox1[4][256],StandardSBox2[4][256];

  if (!strcmp(hashtype,"feal")) {
    /*
    ** feal: default key initialisation (128 bits)
    */
    for (i=0;i < 16;i++) ekey1[i] = 0;
    for (i=0;i < 16;i++) ekey2[i] = 0;

    for (i=0;i < strlen(key1);i++) ekey1[i] = key1[i];
    for (i=0;i < strlen(key2);i++) ekey2[i] = key2[i];
    /*
    ** create key schedule for feal
    */
    Feal_GetKeySchedule(ekey1,feal_keyschedule1,round);
```

```
  Feal_GetKeySchedule(ekey2,feal_keyschedule2,round);
}

if (!strcmp(hashtype,"des")) {
  /*
  ** des: default key initialisation (64 bits)
  */
  for (i=0;i < 8;i++) dkey1[i] = 0;
  for (i=0;i < 8;i++) dkey2[i] = 0;

  for (i=0;i < strlen(key1);i++) dkey1[i] = key1[i];
  for (i=0;i < strlen(key2);i++) dkey2[i] = key2[i];
  /*
  ** create key schedule for des
  */
  Des_GetKeySchedule(dkey1, des_keyschedule1, round);
  Des_GetKeySchedule(dkey2, des_keyschedule2, round);
}

if (!strcmp(hashtype,"loki")) {
  /*
  ** loki: default key initialisation (64 bits)
  */
  for (i=0;i < 8;i++) lkey1[i] = 0;
  for (i=0;i < 8;i++) lkey2[i] = 0;

  for (i=0;i < strlen(key1);i++) lkey1[i] = key1[i];
  for (i=0;i < strlen(key2);i++) lkey2[i] = key2[i];
  /*
  ** create key schedule for loki
  */
  Loki_GetKeySchedule(lkey1, loki_keyschedule1, round);
  Loki_GetKeySchedule(lkey2, loki_keyschedule2, round);
}

if (!strcmp(hashtype,"khufu")) {
  InitialSboxSchedule(Sbox1);
  InitialSboxSchedule(Sbox2);
  InitialSboxSchedule(Sbox1,InitialSbox1,key1);
  InitialSboxSchedule(Sbox2,InitialSbox2,key2);
}

if (!strcmp(hashtype,"khafre")) {
  /*
  ** khafre: default key initialisation (128 bits)
  */
  for (i=0;i < 16;i++) Tkey1[i] = i + 11;
  for (i=0;i < 16;i++) Tkey2[i] = i + 13;

  for (i=0;i < strlen(key1);i++) Tkey1[i] = key1[i];
  for (i=0;i < strlen(key2);i++) Tkey2[i] = key2[i];

  KhafreSboxSchedule(StandardSBox1);
  KhafreSboxSchedule(StandardSBox2);
}


Temp1[0] = Digest1[0]; Temp1[1] = Digest1[1];
Temp2[0] = Digest2[0]; Temp2[1] = Digest1[1];

if (!strcmp(hashtype,"feal"))
  EnFeal(Digest1, feal_keyschedule1, round);

if (!strcmp(hashtype,"des"))
  EnDes(Digest1,des_keyschedule1,round);

if (!strcmp(hashtype,"loki"))
  EnLoki(Digest1,loki_keyschedule1,round);

if (!strcmp(hashtype,"khufu"))
  EnKhufu(Digest1, Sbox1, round);

if (!strcmp(hashtype,"khafre"));
  EnKhafre(Digest1, StandardSBox1, Lkey1,round);

T1[0] = Digest1[0] ^ Digest2[0]; T1[1] = Digest1[1] ^ Digest2[1];
```

```
  Temp3[0] = Digest1[0] ^ Digest2[0]; Temp3[1] = Digest1[1] ^ Digest2[1];

  if (!strcmp(hashtype,"feal"))
    EnFeal(Temp3, feal_keyschedule2, round);

  if (!strcmp(hashtype,"des"))
    EnDes(Temp3,des_keyschedule2,round);

  if (!strcmp(hashtype,"loki"))
    EnLoki(Temp3,loki_keyschedule2,round);

  if (!strcmp(hashtype,"khufu"))
    EnKhufu(Temp3, Sbox2, round);

  if (!strcmp(hashtype,"khafre"));
    EnKhafre(Temp3, StandardSBox2, Lkey2,round);

  T1[0] = Digest1[0] ^ Digest2[0]; T1[1] = Digest1[1] ^ Digest2[1];
  T2[0] = Temp3[0] ^ Temp1[0]; T2[1] = Temp3[1] ^ Temp1[1];

  Digest1[0] = T2[0] ^ Temp2[0]; Digest1[1] = T2[1] ^ Temp2[1];
  Digest2[0] = T1[0] ^ Temp1[0]; Digest2[1] = T1[1] ^ Temp1[1];
}


/*
** PrintWord() : print a 32-bit word in hexadecimal format.
**
*/
void PrintWord(Word)
Long Word;
{
  fprintf(stdout,"%02x%02x%02x%02x",
    (unsigned char)((Word >> 24) & 0xFF),
    (unsigned char)((Word >> 16) & 0xFF),
    (unsigned char)((Word >>  8) & 0xFF),
    (unsigned char)( Word & 0xFF));
}

/*
**
** Print the formatted digest
*/
void PrintFormat(Format,Digest1,Digest2,alg,round)
char *Format,*alg;
int round;
Long *Digest1,*Digest2;
{
  int i=0;

  while (i < strlen(Format)) {
    if (Format[i] != '%') {
      fprintf(stdout,"%c", Format[i]);
      i++;
    }
    else {
      switch (Format[i+1]) {
        case 'h': PrintWord(Digest1[0]);
                  PrintWord(Digest1[1]);
                  PrintWord(Digest2[0]);
                  PrintWord(Digest2[1]);
                  break;
        case 'a': fprintf(stdout,"%s", alg);
                  break;
        case 'r': fprintf(stdout,"%02d", round);
                  break;
        case 'n': fprintf(stdout,"\n");
                  break;
        case 't': fprintf(stdout,"\t");
                  break;
        case '%': fprintf(stdout,"%%");
                  break;
      }
      i += 2;
    }
  }
```

```
  fprintf(stdout,"\n");
}

/*
** End of dbh.c
**
*/
```

```
/*
** Program: Md4.c
** Message Digest Algorithm MD4
** Compile: cc -o md4 md4.c
** Dzung Le - Oct 92
**
*/
#include <stdio.h>
#include <time.h>

typedef unsigned long Long;

#define C2   013240474631   /* square root of 2 */
#define C3   015666365641   /* square root of 3 */

/*
** Md4 Functions
**
*/
#define f(X,Y,Z)       ((X&Y) | ((~X)&Z))
#define g(X,Y,Z)       ((X&Y) | (X&Z) | (Y&Z))
#define h(X,Y,Z)       (X^Y^Z)
#define rot(X,S)       (tmp=X,(tmp<<S) | (tmp>>(32-S)))
#define ff(A,B,C,D,i,s) A = rot((A + f(B,C,D) + X[i]), s)
#define gg(A,B,C,D,i,s) A = rot((A + g(B,C,D) + X[i] + C2),s)
#define hh(A,B,C,D,i,s) A = rot((A + h(B,C,D) + X[i] + C3),s)

/*
** Initialised digest values
**
*/
#define I0   0x67452301
#define I1   0xefcdab89
#define I2   0x98badcfe
#define I3   0x10325476

Long Digest[4] = {I0, I1, I2, I3};

/*
** Input file
*/
FILE *fpi;

/*
** Function prototypes
*/
void Md4TimeTrial(),
    GetDigestFromFile(),
    GetDigestFromString(),
    UpdateMd4A(),
    UpdateMd4B(),
    Transform(),
    PrintDigestFormat(),
    PrintWord();

/*
** main() : main routine.
**
*/
main(argc,argv)
int argc;
char **argv;
{
  int  c,tflag,fflag,sflag,Fflag,errflag;
  char *infile,*instring;
  extern char *optarg;
  extern int   optind;
  char *Format;

  /*
  ** option initializaton
  */
  tflag = fflag = sflag = Fflag = errflag = 0;

  /*
```

```
** Process options
*/
while (( c = getopt(argc,argv,"ts:f:F:")) != EOF) {
  switch (c) {
  case 't': tflag++;
          break;
  case 's': sflag++;
          instring = optarg;
          break;
  case 'f': infile = optarg;
          fflag++;
          break;
  case 'F': Format = optarg;
          Fflag++;
          break;
  case '?': errflag++;
          break;
  }
}

/*
** Error checking
*/
if (errflag) {
  fprintf(stderr,"Usage: %s -s string | ", argv[0]);
  fprintf(stderr,"-f input_file | -t [ -F Format ]\n\n");
  exit(1);
}

/*
** Time trial testing
*/
if (tflag)
  Md4TimeTrial();

/*
** Get message digest from input file.
*/
else if (fflag) {
  if ((fpi = fopen(infile,"r")) == NULL) {
    fprintf(stderr,"%s: Couldn't open file %s for input.\n",
      argv[0], infile);
    exit(1);
  }
  else GetDigestFromFile(fpi);
}

/*
** Get message digest from input string.
*/
else if (sflag) {
  GetDigestFromString(instring);
}

/*
** Get message digest from standard output
*/
else
  GetDigestFromFile(stdout);

if (!Fflag) {
  PrintWord(Digest[0]); PrintWord(Digest[1]);
  PrintWord(Digest[2]); PrintWord(Digest[3]);
  fprintf(stdout,"\n");
}
else {
  PrintDigestFormat(Format, Digest);
}
}

/*
** GetDigestFromString() : Get message digest from an input string
**
*/
void GetDigestFromString(inString)
char *inString;
```

```
{
  int  Len, inLen;
  char *inStr = inString;
  Long bits[2];    /* max file length (in bits) 2 powered 64 */

  /*
  ** Process 64-char blocks
  */
  bits[0] = bits[1] = 0L;

  while ((Len = strlen(inStr)) > 0) {

    if (Len >= 64)
      inLen = 64;
    else
      inLen = Len;

    /* increment the number of Len bits */
    if ((bits[0] + (inLen << 3)) < bits[0])
      bits[1]++;
    bits[0] += (inLen << 3);

    if (inLen == 64) {
      UpdateMd4A(inStr);
      inStr += 64;  /* increment pointer to next block */
    }
    else {
      /*
      ** process the last block (less than 64 chars)
      */
      UpdateMd4B(inStr, inLen, bits);
      inStr += Len;
    }

  }

  /*
  ** The length is multiples of 64 chars - process the last block here.
  */
  if (inLen == 64)
    UpdateMd4B(inStr, 0, bits);
}

/*
** GetDigestFromFile() : Get message digest from an input file.
**
*/
void GetDigestFromFile(fp)
FILE *fp;
{
  char inStr[65];
  int  Len;
  Long bits[2];    /* max file length (in bits) 2 powered 64 */

  bits[0] = bits[1] = (Long)0;

  /*
  ** Process 64-char blocks from input file.
  */
  while ((Len = fread(inStr, 1, 64, fp)) > 0) {

    /* increment the number of Len bits */
    if ((bits[0] + (Len << 3)) < bits[0])
      bits[1]++;
    bits[0] += (Len << 3);

    if (Len == 64)
      UpdateMd4A(inStr);
    else
      /* process the last block (less than 64 chars) */
      UpdateMd4B(inStr, Len, bits);
  }

  /*
  ** The length is multiples of 64 chars - process the last block here.
  */
```

```
  if (Len == 64)
    UpdateMd4B(inStr, 0, bits);
}

/*
** Md4TimeTrial() : Timing test for 10-Mb block (real time only).
** For system time, refer to UNIX command timex.
*/
void Md4TimeTrial()
{
  long i, t1, t2;
  char inString[65];
  Long bits[2];

  for (i=0;i < 64;i++)
    inString[i] = i;

  bits[0] = (Long)(10 * 1024 * 1024);
  bits[1] = 0L;

  time(&t1);

  /*
  ** Process 10-Mb block (multiple of 64-byte blocks).
  */
  for (i=0; i < 163840;i++)
    UpdateMd4A(inString);

  /*
  ** Final block
  */
  UpdateMd4B(inString,0,bits);

  time(&t2);

  fprintf(stdout,"Real Time - seconds to process 10MB block: %d\n", t2-t1);
}

/*
** UpdateMd4A() : process 512-bit blocks.
**
*/
void UpdateMd4A(inString)
char *inString;
{
  Long Word[16];
  int  i;

  /*
  ** Process 512-bit blocks
  */
  for (i=0;i < 16;i++) {
    Word[i] = ((inString[(i*4) + 3] << 24) |
            (inString[(i*4) + 2] << 16) |
            (inString[(i*4) + 1] <<  8) |
            (inString[(i*4) + 0]));
  }

  Transform(Digest,Word);
}

/*
** UpdateMd4B() : process a block having no of bits less than 512.
**
*/
void UpdateMd4B(inString,inLen,bits)
char *inString;
int inLen;
Long *bits;
{
  unsigned char buf[64];
  Long Word[16];
  int cnt,i;

  for (i = 0;i < inLen;i++) buf[i] = inString[i];
  cnt = inLen;
```

```
  /*
  ** Process last block having less than 512 bits
  */
  if (((cnt << 3) & 0x1FF) <= 0x1C0) {

    buf[cnt] = 0x80;    /* append 1 to buf */

    for (i=cnt+1;i < 64;i++) buf[i] = 0x0;

    for (i=0;i < cnt;i++) {
      Word[i] = ((buf[(i*4) + 3] << 24) |
              (buf[(i*4) + 2] << 16) |
              (buf[(i*4) + 1] << 8) |
              (buf[(i*4) + 0]));
    }

    /*
    ** store bit length in the last two words
    */
    Word[14] = bits[0];
    Word[15] = bits[1];

    Transform(Digest,Word);
  }
  else {

    buf[cnt] = 0x80;  /* append 1 to buf */

    for (i=cnt+1;i < 64;i++) buf[i] = 0x0;

    for (i=0;i < 32;i++) {
      Word[i] = ((buf[(i*4) + 3] << 24) |
              (buf[(i*4) + 2] << 16) |
              (buf[(i*4) + 1] << 8) |
              (buf[(i*4) + 0]));
    }

    Transform(Digest,Word);

    for (i=15;i >= 0;i--) Word[i] = 0L;

    /*
    ** store bit length in the last two words
    */
    Word[14] = bits[0];
    Word[15] = bits[1];

    Transform(Digest,Word);
  }
}

/*
** Transform() : Transform to new Digest with input Word.
**
*/
void Transform(Digest, Word)
Long *Digest;
Long *Word;
{
  Long tmp, A = Digest[0], B = Digest[1], C = Digest[2], D = Digest[3];
  Long X[16];
  int i;

  for (i=0;i < 16;i++) {

    X[i] = Word[i];

#define fs1  3
#define fs2  7
#define fs3  11
#define fs4  19

    ff(A, B, C, D, 0, fs1);  /* round 1 */
    ff(D, A, B, C, 1, fs2);
    ff(C, D, A, B, 2, fs3);
```

```
    ff(B, C, D, A, 3, fs4);
    ff(A, B, C, D, 4, fs1);
    ff(D, A, B, C, 5, fs2);
    ff(C, D, A, B, 6, fs3);
    ff(B, C, D, A, 7, fs4);
    ff(A, B, C, D, 8, fs1);
    ff(D, A, B, C, 9, fs2);
    ff(C, D, A, B,10, fs3);
    ff(B, C, D, A,11, fs4);
    ff(A, B, C, D,12, fs1);
    ff(D, A, B, C,13, fs2);
    ff(C, D, A, B,14, fs3);
    ff(B, C, D, A,15, fs4);

#define gs1  3
#define gs2  5
#define gs3  9
#define gs4  13

    gg(A, B, C, D, 0, gs1);  /* round 2 */
    gg(D, A, B, C, 4, gs2);
    gg(C, D, A, B, 8, gs3);
    gg(B, C, D, A,12, gs4);
    gg(A, B, C, D, 1, gs1);
    gg(D, A, B, C, 5, gs2);
    gg(C, D, A, B, 9, gs3);
    gg(B, C, D, A,13, gs4);
    gg(A, B, C, D, 2, gs1);
    gg(D, A, B, C, 6, gs2);
    gg(C, D, A, B,10, gs3);
    gg(B, C, D, A,14, gs4);
    gg(A, B, C, D, 3, gs1);
    gg(D, A, B, C, 7, gs2);
    gg(C, D, A, B,11, gs3);
    gg(B, C, D, A,15, gs4);

#define hs1  3
#define hs2  9
#define hs3  11
#define hs4  15

    hh(A, B, C, D, 0, hs1);  /* round 3 */
    hh(D, A, B, C, 8, hs2);
    hh(C, D, A, B, 4, hs3);
    hh(B, C, D, A,12, hs4);
    hh(A, B, C, D, 2, hs1);
    hh(D, A, B, C,10, hs2);
    hh(C, D, A, B, 6, hs3);
    hh(B, C, D, A,14, hs4);
    hh(A, B, C, D, 1, hs1);
    hh(D, A, B, C, 9, hs2);
    hh(C, D, A, B, 5, hs3);
    hh(B, C, D, A,13, hs4);
    hh(A, B, C, D, 3, hs1);
    hh(D, A, B, C,11, hs2);
    hh(C, D, A, B, 7, hs3);
    hh(B, C, D, A,15, hs4);
  }

  Digest[0] += A; Digest[1] += B; Digest[2] += C; Digest[3] += D;
}

/*
**
** Print out the digest with the user-supplied format
*/
void PrintDigestFormat(Format,Digest)
char *Format;
Long *Digest;
{
  int i=0;

  while (i < strlen(Format)) {
    if (*(Format + i) != '%') {
      fprintf(stdout,"%c", *(Format + i));
      i++;
```

```
    }
    else {
      switch (*(Format + i + 1)) {
        case 'h': PrintWord(Digest[0]);
                  PrintWord(Digest[1]);
                  PrintWord(Digest[2]);
                  PrintWord(Digest[3]);
                  break;
        case 'a': fprintf(stdout,"md4");
                  break;
        case 'n': fprintf(stdout,"\n");
                  break;
        case 't': fprintf(stdout,"\t");
                  break;
        case '%': fprintf(stdout,"%%");
                  break;
      }
      i += 2;
    }
  }
  fprintf(stdout,"\n");
}

/*
** PrintWord() : print a 32-bit word in hexadecimal format.
**
*/
void PrintWord(Word)
Long Word;
{
  fprintf(stdout,"%02x%02x%02x%02x",
    (unsigned char)((Word >> 24) & 0xFF),
    (unsigned char)((Word >> 16) & 0xFF),
    (unsigned char)((Word >>  8) & 0xFF),
    (unsigned char)( Word & 0xFF));
}
/*
** End of md4.c
**
*/
```

```
/*
** Program: Md4x.c
** Message Digest Algorithm MD4 (extended version - 128-bit digest)
** Compile: cc -o md4x md4x.c
** Dzung Le - Oct 92
**
*/
#include <stdio.h>
#include <time.h>

typedef unsigned long Long;

#define C2   012050505746
#define C3   013423350444

/*
** Md4 Functions
**
*/
#define f(X,Y,Z)       ((X&Y) | ((~X)&Z))
#define g(X,Y,Z)       ((X&Y) | (X&Z) | (Y&Z))
#define h(X,Y,Z)       (X^Y^Z)
#define rot(X,S)       (tmp=X,(tmp<<S) | (tmp>>(32-S)))
#define ff(A,B,C,D,i,s) A = rot((A + f(B,C,D) + X[i]), s)
#define gg(A,B,C,D,i,s) A = rot((A + g(B,C,D) + X[i] + C2),s)
#define hh(A,B,C,D,i,s) A = rot((A + h(B,C,D) + X[i] + C3),s)

/*
** Initialised digest values
**
*/
#define I0   0x67452301
#define I1   0xefcdab89
#define I2   0x98badcfe
#define I3   0x10325476

#define I4   0x33221100
#define I5   0x77665544
#define I6   0xbbaa9988
#define I7   0xffeeddcc

Long Digest1[4] = {I0, I1, I2, I3};
Long Digest2[4] = {I4, I5, I6, I7};

/*
** Input file
*/
FILE *fpi;

/*
** Function prototypes
*/
void Md4TimeTrial(),
   GetDigestFromFile(),
   GetDigestFromString(),
   UpdateMd4A(),
   UpdateMd4B(),
   Transform(),
   Transform1(),
   Transform2(),
   Transform3(),
   PrintDigestFormat(),
   PrintWord();

/*
** main() : main routine.
**
*/
main(argc,argv)
int argc;
char **argv;
{
  int  c,tflag,fflag,sflag,Fflag,errflag;
  char *infile,*instring;
  extern char *optarg;
```

```
extern int  optind;
char *Format;

/*
** option initializaton
*/
tflag = fflag = sflag = Fflag = errflag = 0;

/*
** Process options
*/
while (( c = getopt(argc,argv,"ts:f:F:")) != EOF) {
  switch (c) {
  case 't': tflag++;
        break;
  case 's': sflag++;
        instring = optarg;
        break;
  case 'f': infile = optarg;
        fflag++;
        break;
  case 'F': Format = optarg;
        Fflag++;
        break;
  case '?': errflag++;
        break;
  }
}

/*
** Error checking
*/
if (errflag) {
  fprintf(stderr,"Usage: %s -s string | ", argv[0]);
  fprintf(stderr,"-f input_file | -t [ -F Format ]\n\n");
  exit(1);
}

/*
** Time trial testing
*/
if (tflag)
  Md4TimeTrial();

/*
** Get message digest from input file.
*/
else if (fflag) {
  if ((fpi = fopen(infile,"r")) == NULL) {
    fprintf(stderr,"%s: Couldn't open file %s for input.\n",
      argv[0], infile);
    exit(1);
  }
  else GetDigestFromFile(fpi);
}

/*
** Get message digest from input string.
*/
else if (sflag) {
  GetDigestFromString(instring);
}

/*
** Get message digest from standard output
*/
else
  GetDigestFromFile(stdout);

if (!Fflag) {
  PrintWord(Digest1[0]); PrintWord(Digest1[1]);
  PrintWord(Digest1[2]); PrintWord(Digest1[3]);
  PrintWord(Digest2[0]); PrintWord(Digest2[1]);
  PrintWord(Digest2[2]); PrintWord(Digest2[3]);
  fprintf(stdout,"\n");
}
```

```
  else {
    PrintDigestFormat(Format, Digest1, Digest2);
  }
}

/*
** GetDigestFromString() : Get message digest from an input string
**
*/
void GetDigestFromString(inString)
char *inString;
{
  int  Len, inLen;
  char *inStr = inString;
  Long bits[2];    /* max file length (in bits) 2 powered 64 */

  /*
  ** Process 64-char blocks
  */
  bits[0] = bits[1] = 0L;

  while ((Len = strlen(inStr)) > 0) {

    if (Len >= 64)
      inLen = 64;
    else
      inLen = Len;

    /* increment the number of Len bits */
    if ((bits[0] + (inLen << 3)) < bits[0])
      bits[1]++;
    bits[0] += (inLen << 3);

    if (inLen == 64) {
      UpdateMd4A(inStr);
      inStr += 64;  /* increment pointer to next block */
    }
    else {
      /*
      ** process the last block (less than 64 chars)
      */
      UpdateMd4B(inStr, inLen, bits);
      inStr += Len;
    }

  }

  /*
  ** The length is multiples of 64 chars - process the last block here.
  */
  if (inLen == 64)
    UpdateMd4B(inStr, 0, bits);
}

/*
** GetDigestFromFile() : Get message digest from an input file.
**
*/
void GetDigestFromFile(fp)
FILE *fp;
{
  char inStr[65];
  int  Len;
  Long bits[2];    /* max file length (in bits) 2 powered 64 */

  bits[0] = bits[1] = (Long)0;

  /*
  ** Process 64-char blocks from input file.
  */
  while ((Len = fread(inStr, 1, 64, fp)) > 0) {

    /* increment the number of Len bits */
    if ((bits[0] + (Len << 3)) < bits[0])
      bits[1]++;
    bits[0] += (Len << 3);
```

```
    if (Len == 64)
      UpdateMd4A(inStr);
    else
      /* process the last block (less than 64 chars) */
      UpdateMd4B(inStr, Len, bits);
  }

  /*
  ** The length is multiples of 64 chars - process the last block here.
  */
  if (Len == 64)
    UpdateMd4B(inStr, 0, bits);
}

/*
** Md4TimeTrial() : Timing test for 10-Mb block (real time only).
** For system time, refer to UNIX command timex.
*/
void Md4TimeTrial()
{
  long i, t1, t2;
  char inString[65];
  Long bits[2];

  for (i=0;i < 64;i++)
    inString[i] = i;

  bits[0] = (Long)(10 * 1024 * 1024);
  bits[1] = 0L;

  time(&t1);

  /*
  ** Process 10-Mb block (multiple of 64-byte blocks).
  */
  for (i=0; i < 163840;i++)
    UpdateMd4A(inString);

  /*
  ** Final block
  */
  UpdateMd4B(inString,0,bits);

  time(&t2);

  fprintf(stdout,"Real Time - seconds to process 10MB block: %d\n", t2-t1);
}

/*
** UpdateMd4A() : process 512-bit blocks.
**
*/
void UpdateMd4A(inString)
char *inString;
{
  Long Word[16];
  int i;

  /*
  ** Process 512-bit blocks
  */
  for (i=0;i < 16;i++) {
    Word[i] = ((inString[(i*4) + 3] << 24) |
            (inString[(i*4) + 2] << 16) |
            (inString[(i*4) + 1] << 8) |
            (inString[(i*4) + 0]));
  }

  Transform(Digest1,Digest2,Word);
}

/*
** UpdateMd4B() : process a block having no of bits less than 512.
**
*/
```

```
void UpdateMd4B(inString,inLen,bits)
char *inString;
int inLen;
Long *bits;
{
  unsigned char buf[64];
  Long Word[16];
  int cnt,i;

  for (i = 0;i < inLen;i++) buf[i] = inString[i];
  cnt = inLen;

  /*
  ** Process last block having less than 512 bits
  */
  if (((cnt << 3) & 0x1FF) <= 0x1C0) {

    buf[cnt] = 0x80;    /* append 1 to buf */

    for (i=cnt+1;i < 64;i++) buf[i] = 0x0;

    for (i=0;i < cnt;i++) {
      Word[i] = ((buf[(i*4) + 3] << 24) |
              (buf[(i*4) + 2] << 16) |
              (buf[(i*4) + 1] << 8) |
              (buf[(i*4) + 0]));
    }

    /*
    ** store bit length in the last two words
    */
    Word[14] = bits[0];
    Word[15] = bits[1];

    Transform(Digest1,Digest2,Word);
  }
  else {

    buf[cnt] = 0x80;  /* append 1 to buf */

    for (i=cnt+1;i < 64;i++) buf[i] = 0x0;

    for (i=0;i < 32;i++) {
      Word[i] = ((buf[(i*4) + 3] << 24) |
              (buf[(i*4) + 2] << 16) |
              (buf[(i*4) + 1] << 8) |
              (buf[(i*4) + 0]));
    }

    Transform(Digest1,Digest2,Word);

    for (i=15;i >= 0;i--) Word[i] = 0L;

    /*
    ** store bit length in the last two words
    */
    Word[14] = bits[0];
    Word[15] = bits[1];

    Transform(Digest1,Digest2,Word);
  }
}

/*
** Transform() : Transform to new Digest with input Word (3 rounds)
**
*/
void Transform(Digest1, Digest2, Word)
Long *Digest1, *Digest2;
Long *Word;
{
  Long Temp;

  /*
  ** Round 1
  */
```

```
  Transform1(Digest1, Word);
  Transform1(Digest2, Word);
  /*
  ** The contents of two A registers are interchanged
  */
  Temp = Digest1[0]; Digest1[0] = Digest2[0]; Digest2[0] = Temp;

  /*
  ** Round 2
  */
  Transform2(Digest1, Word);
  Transform2(Digest2, Word);
  /*
  ** The contents of two A registers are interchanged
  */
  Temp = Digest1[0]; Digest1[0] = Digest2[0]; Digest2[0] = Temp;

  /*
  ** Round 3
  */
  Transform3(Digest1, Word);
  Transform3(Digest2, Word);
  /*
  ** The contents of two A registers are interchanged
  */
  Temp = Digest1[0]; Digest1[0] = Digest2[0]; Digest2[0] = Temp;
}

/*
** Transform1() : Transform to new Digest with input Word (round 1).
**
*/
void Transform1(Digest, Word)
Long *Digest;
Long *Word;
{
  Long tmp, A = Digest[0], B = Digest[1], C = Digest[2], D = Digest[3];
  Long X[16];
  int i;

  for (i=0;i < 16;i++) {

    X[i] = Word[i];

#define fs1  3
#define fs2  7
#define fs3  11
#define fs4  19

    ff(A, B, C, D, 0, fs1);  /* round 1 */
    ff(D, A, B, C, 1, fs2);
    ff(C, D, A, B, 2, fs3);
    ff(B, C, D, A, 3, fs4);
    ff(A, B, C, D, 4, fs1);
    ff(D, A, B, C, 5, fs2);
    ff(C, D, A, B, 6, fs3);
    ff(B, C, D, A, 7, fs4);
    ff(A, B, C, D, 8, fs1);
    ff(D, A, B, C, 9, fs2);
    ff(C, D, A, B,10, fs3);
    ff(B, C, D, A,11, fs4);
    ff(A, B, C, D,12, fs1);
    ff(D, A, B, C,13, fs2);
    ff(C, D, A, B,14, fs3);
    ff(B, C, D, A,15, fs4);
  }

  Digest[0] += A; Digest[1] += B; Digest[2] += C; Digest[3] += D;
}


/*
** Transform2() : Transform to new Digest with input Word (round 2).
**
*/
void Transform2(Digest, Word)
```

```
Long *Digest;
Long *Word;
{
  Long tmp, A = Digest[0], B = Digest[1], C = Digest[2], D = Digest[3];
  Long X[16];
  int i;

  for (i=0;i < 16;i++) {

    X[i] = Word[i];

#define gs1  3
#define gs2  5
#define gs3  9
#define gs4  13

    gg(A, B, C, D, 0, gs1);  /* round 2 */
    gg(D, A, B, C, 4, gs2);
    gg(C, D, A, B, 8, gs3);
    gg(B, C, D, A,12, gs4);
    gg(A, B, C, D, 1, gs1);
    gg(D, A, B, C, 5, gs2);
    gg(C, D, A, B, 9, gs3);
    gg(B, C, D, A,13, gs4);
    gg(A, B, C, D, 2, gs1);
    gg(D, A, B, C, 6, gs2);
    gg(C, D, A, B,10, gs3);
    gg(B, C, D, A,14, gs4);
    gg(A, B, C, D, 3, gs1);
    gg(D, A, B, C, 7, gs2);
    gg(C, D, A, B,11, gs3);
    gg(B, C, D, A,15, gs4);
  }

  Digest[0] += A; Digest[1] += B; Digest[2] += C; Digest[3] += D;
}


/*
** Transform3() : Transform to new Digest with input Word (round 3).
**
*/
void Transform3(Digest, Word)
Long *Digest;
Long *Word;
{
  Long tmp, A = Digest[0], B = Digest[1], C = Digest[2], D = Digest[3];
  Long X[16];
  int i;

  for (i=0;i < 16;i++) {

    X[i] = Word[i];

#define hs1  3
#define hs2  9
#define hs3  11
#define hs4  15

    hh(A, B, C, D, 0, hs1);  /* round 3 */
    hh(D, A, B, C, 8, hs2);
    hh(C, D, A, B, 4, hs3);
    hh(B, C, D, A,12, hs4);
    hh(A, B, C, D, 2, hs1);
    hh(D, A, B, C,10, hs2);
    hh(C, D, A, B, 6, hs3);
    hh(B, C, D, A,14, hs4);
    hh(A, B, C, D, 1, hs1);
    hh(D, A, B, C, 9, hs2);
    hh(C, D, A, B, 5, hs3);
    hh(B, C, D, A,13, hs4);
    hh(A, B, C, D, 3, hs1);
    hh(D, A, B, C,11, hs2);
    hh(C, D, A, B, 7, hs3);
    hh(B, C, D, A,15, hs4);
  }
```

```
  Digest[0] += A; Digest[1] += B; Digest[2] += C; Digest[3] += D;
}

/*
**
** Print out the digest with the user-supplied format
*/
void PrintDigestFormat(Format,Digest1,Digest2)
char *Format;
Long *Digest1, *Digest2;
{
  int i=0;

  while (i < strlen(Format)) {
    if (*(Format + i) != '%') {
      fprintf(stdout,"%c", *(Format + i));
      i++;
    }
    else {
      switch (*(Format + i + 1)) {
        case 'h': PrintWord(Digest1[0]); PrintWord(Digest1[1]);
                  PrintWord(Digest1[2]); PrintWord(Digest1[3]);
                  PrintWord(Digest2[0]); PrintWord(Digest2[1]);
                  PrintWord(Digest2[2]); PrintWord(Digest2[3]);
                  break;
        case 'a': fprintf(stdout,"md4");
                  break;
        case 'n': fprintf(stdout,"\n");
                  break;
        case 't': fprintf(stdout,"\t");
                  break;
        case '%': fprintf(stdout,"%%");
                  break;
      }
      i += 2;
    }
  }
  fprintf(stdout,"\n");
}

/*
** PrintWord() : print a 32-bit word in hexadecimal format.
**
*/
void PrintWord(Word)
Long Word;
{
  fprintf(stdout,"%02x%02x%02x%02x",
    (unsigned char)((Word >> 24) & 0xFF),
    (unsigned char)((Word >> 16) & 0xFF),
    (unsigned char)((Word >>  8) & 0xFF),
    (unsigned char)( Word & 0xFF));
}
/*
** End of md4x.c
**
*/
```

```
/*
** Program: Md5.c
** Message Digest Algorithm MD5
** Compile: cc -o md5 md5.c
** Dzung Le - Oct, 92
**
*/
#include <stdio.h>
#include <time.h>

typedef unsigned long Long;

/*
** Md5 functions
**
*/
#define F(x, y, z) (((x) & (y)) | ((~x) & (z)))
#define G(x, y, z) (((x) & (z)) | ((y) & (~z)))
#define H(x, y, z) ((x) ^ (y) ^ (z))
#define I(x, y, z) ((y) ^ ((x) | (~z)))

#define ROTATE_LEFT(x, n) (((x) << (n)) | ((x) >> (32-(n))));

#define ff(a, b, c, d, x, s, ac) \
 {(a) += F ((b), (c), (d)) + (x) + (Long)(ac); \
  (a) = ROTATE_LEFT ((a), (s)); (a) += (b); \
 }

#define gg(a, b, c, d, x, s, ac) \
 { (a) += G ((b), (c), (d)) + (x) + (Long)(ac); \
  (a) = ROTATE_LEFT ((a), (s)); \
  (a) += (b); \
 }

#define hh(a, b, c, d, x, s, ac) \
 {(a) += H((b), (c), (d)) + (x) + (Long)(ac); \
  (a) = ROTATE_LEFT ((a), (s)); \
  (a) += (b); \
 }

#define ii(a, b, c, d, x, s, ac) \
 {(a) += I ((b), (c), (d)) + (x) + (Long)(ac); \
  (a) = ROTATE_LEFT ((a), (s)); \
  (a) += (b); \
 }

/*
** Initial digest values
*/
#define I0    0x67452301
#define I1    0xefcdab89
#define I2    0x98badcfe
#define I3    0x10325476

Long Digest[4] = {I0, I1, I2, I3};

/*
** Input file
*/
FILE *fpi;

/*
** Function prototypes
*/
void Md5TimeTrial(),
    GetDigestFromFile(),
    GetDigestFromString(),
    UpdateMd5A(),
    UpdateMd5B(),
    Transform(),
    PrintDigestFormat(),
    PrintWord();

/*
** Main routine.
```

```
**
*/
main(argc,argv)
int argc;
char **argv;
{
  int  c,sflag,tflag,fflag,Fflag,errflag;
  char *infile,*Format,*instring;
  extern char *optarg;
  extern int   optind;

  /*
  ** initialisaton
  */
  sflag = tflag = fflag = Fflag = errflag = 0;

  /*
  ** Process options
  */
  while (( c = getopt(argc,argv,"ts:f:F:")) != EOF) {
    switch (c) {
    case 't': tflag++;
            break;
    case 'f': infile = optarg;
            fflag++;
            break;
    case 'F': Format = optarg;
            Fflag++;
            break;
    case 's': instring = optarg;
            sflag++;
            break;
    case '?': errflag++;
            break;
    }
  }

  /*
  ** Error checking
  */
  if (errflag) {
    fprintf(stderr,"Usage: %s -s string | ", argv[0]);
    fprintf(stderr,"-f input_file | -t [ -F Format ]\n\n");
    exit(1);
  }

  /*
  ** Time trial testing
  */
  if (tflag)
    Md5TimeTrial();

  /*
  ** Get message disgest from input file.
  */
  else if (fflag) {
    if ((fpi = fopen(infile,"r")) == NULL) {
      fprintf(stderr,"%s: Couldn't open file %s for input.\n",
        argv[0], infile);
      exit(1);
    }
    else GetDigestFromFile(fpi);
  }

  /*
  ** Get message digest from input string
  */
  else if (sflag)
    GetDigestFromString(instring);

  /*
  ** Get message digest from standard output
  */
  else
    GetDigestFromFile(stdout);
```

```
  if (!Fflag) {
    PrintWord(Digest[0]); PrintWord(Digest[1]);
    PrintWord(Digest[2]); PrintWord(Digest[3]);
    fprintf(stdout,"\n");
  }
  else {
    PrintDigestFormat(Format, Digest);
  }
}

/*
** GetDigestFromString() : Get message digest from an input string.
**
*/
void GetDigestFromString(inString)
char *inString;
{
  int  Len, inLen;
  char *inStr = inString;
  Long bits[2];    /* max file length (in bits) 2 powered 64 */

  bits[0] = bits[1] = 0L;

  /*
  ** Process 64-char blocks
  */
  while ((Len = strlen(inStr)) > 0) {

    if (Len >= 64)
      inLen = 64;
    else
      inLen = Len;

    /* increment the number of Len bits */
    if ((bits[0] + (inLen << 3)) < bits[0])
      bits[1]++;
    bits[0] += (inLen << 3);

    if (inLen == 64) {
      UpdateMd5A(inStr);
      inStr += 64;  /* increment pointer to next block */
    }
    else {
      /*
      ** process the last block (less than 64 chars)
      */
      UpdateMd5B(inStr, inLen, bits);
      inStr += Len;
    }
  }

  /*
  ** The length is multiples of 64 chars - process the last block here.
  */
  if (inLen == 64)
    UpdateMd5B(inStr, 0, bits);

}

/*
** GetDigestFromFile() : Get message digest from an input file.
**
*/
void GetDigestFromFile(fp)
FILE *fp;
{
  char inStr[65];
  int  Len;
  Long bits[2];

  bits[0] = bits[1] = 0L;

  /*
  ** Process 64-char blocks from input file.
  */
  while ((Len = fread(inStr, 1, 64, fp)) > 0) {
```

```
    /* increment the number of Len bits */
    if ((bits[0] + (Len << 3)) < bits[0])
      bits[1]++;
    bits[0] += (Len << 3);

    if (Len == 64)
      UpdateMd5A(inStr);
    else
      /*
      ** process the last block (less than 64 chars)
      */
      UpdateMd5B(inStr, Len);
  }

  /*
  ** The length is multiples of 64 chars - process the last block here.
  */
  if (Len == 64)
    UpdateMd5B(inStr, 0, bits);
}

/*
** Md5TimeTrial() : Timing test for 10-Mb block (real time only).
** For system time, refer to UNIX command timex.
*/
void Md5TimeTrial()
{
  long i, t1, t2;
  char inString[65];
  Long bits[2];

  for (i=0;i < 64;i++)
    inString[i] = i;

  bits[0] = (Long)(10 * 1024 * 1024);
  bits[1] = 0L;

  time(&t1);

  /*
  ** Process 10-Mb block (multiple of 64-byte blocks).
  */
  for (i=0; i < 163840;i++)
    UpdateMd5A(inString);

  /*
  ** Final block
  */
  UpdateMd5B(inString,0,bits);

  time(&t2);

  fprintf(stdout,"Real Time - seconds to process 10MB block: %d\n", t2-t1);
}

/*
** UpdateMd5A() : process 512-bit blocks.
**
*/
void UpdateMd5A(inString)
char *inString;
{
  Long Word[16];
  int i;

  /*
  **
  ** Process 512-bit blocks
  */
  for (i=0;i < 16;i++) {
    Word[i] = ((inString[(i*4) + 3] << 24) |
            (inString[(i*4) + 2] << 16) |
            (inString[(i*4) + 1] << 8) |
            (inString[(i*4) + 0]));
  }
```

```
  Transform(Digest,Word);
}

/*
** UpdateMd5B() : process a block having no of bits less than 512.
**
*/
void UpdateMd5B(inString,inLen,bits)
char *inString;
int inLen;
Long *bits;
{
  unsigned char buf[64];
  Long Word[16];
  int cnt,i;

  for (i = 0;i < inLen;i++) buf[i] = inString[i];
  cnt = inLen;

  /*
  ** Process last block having less than 512 bits
  */
  if (((cnt << 3) & 0x1FF) <= 0x1C0) {

    buf[cnt] = 0x80;    /* append 1 to buf */

    for (i=cnt+1;i < 64;i++) buf[i] = 0x0;

    for (i=0;i < cnt;i++) {
      Word[i] = ((buf[(i*4) + 3] << 24) |
              (buf[(i*4) + 2] << 16) |
              (buf[(i*4) + 1] << 8) |
              (buf[(i*4) + 0]));
    }

    /*
    ** store bit length in the last two words
    */
    Word[14] = bits[0];
    Word[15] = bits[1];

    Transform(Digest,Word);
  }
  else {
    buf[cnt] = 0x80;    /* append 1 to buf */

    for (i=cnt+1;i < 64;i++) buf[i] = 0x0;

    for (i=0;i < 32;i++) {
      Word[i] = ((buf[(i*4) + 3] << 24) |
              (buf[(i*4) + 2] << 16) |
              (buf[(i*4) + 1] << 8) |
              (buf[(i*4) + 0]));
    }

    Transform(Digest,Word);

    for (i=15;i >= 0;i--) Word[i] = 0L;

    /*
    ** store bit length in the last two words
    */
    Word[14] = bits[0];
    Word[15] = bits[1];

    Transform(Digest,Word);
  }
}

/*
** Transform() : Transform to new Digest with input Word.
**
*/
void Transform(Digest, Word)
Long *Digest;
```

```
Long *Word;
{
  Long a = Digest[0], b = Digest[1], c = Digest[2], d = Digest[3];

#define s11 7
#define s12 12
#define s13 17
#define s14 22

  ff (a, b, c, d, Word[ 0], s11, 3614090360);
  ff (d, a, b, c, Word[ 1], s12, 3905402710);
  ff (c, d, a, b, Word[ 2], s13,  606105819);
  ff (b, c, d, a, Word[ 3], s14, 3250441966);
  ff (a, b, c, d, Word[ 4], s11, 4118548399);
  ff (d, a, b, c, Word[ 5], s12, 1200080426);
  ff (c, d, a, b, Word[ 6], s13, 2821735955);
  ff (b, c, d, a, Word[ 7], s14, 4249261313);
  ff (a, b, c, d, Word[ 8], s11, 1770035416);
  ff (d, a, b, c, Word[ 9], s12, 2336552879);
  ff (c, d, a, b, Word[10], s13, 4294925233);
  ff (b, c, d, a, Word[11], s14, 2304563134);
  ff (a, b, c, d, Word[12], s11, 1804603682);
  ff (d, a, b, c, Word[13], s12, 4254626195);
  ff (c, d, a, b, Word[14], s13, 2792965006);
  ff (b, c, d, a, Word[15], s14, 1236535329);

#define s21 5
#define s22 9
#define s23 14
#define s24 20

  gg (a, b, c, d, Word[ 1], s21, 4129170786);  /* 17 */
  gg (d, a, b, c, Word[ 6], s22, 3225465664);  /* 18 */
  gg (c, d, a, b, Word[11], s23,  643717713);  /* 19 */
  gg (b, c, d, a, Word[ 0], s24, 3921069994);  /* 20 */
  gg (a, b, c, d, Word[ 5], s21, 3593408605);  /* 21 */
  gg (d, a, b, c, Word[10], s22,   38016083);  /* 22 */
  gg (c, d, a, b, Word[15], s23, 3634488961);  /* 23 */
  gg (b, c, d, a, Word[ 4], s24, 3889429448);  /* 24 */
  gg (a, b, c, d, Word[ 9], s21,  568446438);  /* 25 */
  gg (d, a, b, c, Word[14], s22, 3275163606);  /* 26 */
  gg (c, d, a, b, Word[ 3], s23, 4107603335);  /* 27 */
  gg (b, c, d, a, Word[ 8], s24, 1163531501);  /* 28 */
  gg (a, b, c, d, Word[13], s21, 2850285829);  /* 29 */
  gg (d, a, b, c, Word[ 2], s22, 4243563512);  /* 30 */
  gg (c, d, a, b, Word[ 7], s23, 1735328473);  /* 31 */
  gg (b, c, d, a, Word[12], s24, 2368359562);  /* 32 */


#define S31 4
#define S32 11
#define S33 16
#define S34 23

  hh (a, b, c, d, Word[ 5], S31, 4294588738);  /* 33 */
  hh (d, a, b, c, Word[ 8], S32, 2272392833);  /* 34 */
  hh (c, d, a, b, Word[11], S33, 1839030562);  /* 35 */
  hh (b, c, d, a, Word[14], S34, 4259657740);  /* 36 */
  hh (a, b, c, d, Word[ 1], S31, 2763975236);  /* 37 */
  hh (d, a, b, c, Word[ 4], S32, 1272893353);  /* 38 */
  hh (c, d, a, b, Word[ 7], S33, 4139469664);  /* 39 */
  hh (b, c, d, a, Word[10], S34, 3200236656);  /* 40 */
  hh (a, b, c, d, Word[13], S31,  681279174);  /* 41 */
  hh (d, a, b, c, Word[ 0], S32, 3936430074);  /* 42 */
  hh (c, d, a, b, Word[ 3], S33, 3572445317);  /* 43 */
  hh (b, c, d, a, Word[ 6], S34,   76029189);  /* 44 */
  hh (a, b, c, d, Word[ 9], S31, 3654602809);  /* 45 */
  hh (d, a, b, c, Word[12], S32, 3873151461);  /* 46 */
  hh (c, d, a, b, Word[15], S33,  530742520);  /* 47 */
  hh (b, c, d, a, Word[ 2], S34, 3299628645);  /* 48 */


#define s41 6
#define s42 10
#define s43 15
#define s44 21
```

```
  ii (a, b, c, d, Word[ 0], s41, 4096336452); /* 49 */
  ii (d, a, b, c, Word[ 7], s42, 1126891415); /* 50 */
  ii (c, d, a, b, Word[14], s43, 2878612391); /* 51 */
  ii (b, c, d, a, Word[ 5], s44, 4237533241); /* 52 */
  ii (a, b, c, d, Word[12], s41, 1700485571); /* 53 */
  ii (d, a, b, c, Word[ 3], s42, 2399980690); /* 54 */
  ii (c, d, a, b, Word[10], s43, 4293915773); /* 55 */
  ii (b, c, d, a, Word[ 1], s44, 2240044497); /* 56 */
  ii (a, b, c, d, Word[ 8], s41, 1873313359); /* 57 */
  ii (d, a, b, c, Word[15], s42, 4264355552); /* 58 */
  ii (c, d, a, b, Word[ 6], s43, 2734768916); /* 59 */
  ii (b, c, d, a, Word[13], s44, 1309151649); /* 60 */
  ii (a, b, c, d, Word[ 4], s41, 4149444226); /* 61 */
  ii (d, a, b, c, Word[11], s42, 3173756917); /* 62 */
  ii (c, d, a, b, Word[ 2], s43,  718787259); /* 63 */
  ii (b, c, d, a, Word[ 9], s44, 3951481745); /* 64 */

  Digest[0] += a; Digest[1] += b; Digest[2] += c; Digest[3] += d;
}

/*
**
** Print out the digest with the user-supplied format
*/
void PrintDigestFormat(Format,Digest)
char *Format;
Long *Digest;
{
  int i=0;

  while (i < strlen(Format)) {
    if (*(Format + i) != '%') {
      fprintf(stdout,"%c", *(Format + i));
      i++;
    }
    else {
      switch (*(Format + i + 1)) {
        case 'h': PrintWord(Digest[0]);
                  PrintWord(Digest[1]);
                  PrintWord(Digest[2]);
                  PrintWord(Digest[3]);
                  break;
        case 'a': fprintf(stdout,"md4");
                  break;
        case 'n': fprintf(stdout,"\n");
                  break;
        case 't': fprintf(stdout,"\t");
                  break;
        case '%': fprintf(stdout,"%%");
                  break;
      }
      i += 2;
    }
  }
  fprintf(stdout,"\n");
}

/*
** PrintWord() : print a 32-bit word in hexadecimal format.
**
*/
void PrintWord(Word)
Long Word;
{
  fprintf(stdout,"%02x%02x%02x%02x",
    (unsigned char)((Word >> 24) & 0xFF),
    (unsigned char)((Word >> 16) & 0xFF),
    (unsigned char)((Word >>  8) & 0xFF),
    (unsigned char)( Word & 0xFF));
}
/*
** End of md5.c
**
*/
```

```
/*
** Include File: haval.h
** One-way hashing algorithm with variable length output - HAVAL
** Dzung Le - Oct, 92
*/
typedef unsigned long Long;

/*
** Right rotation of a word X by s bits
*/
#define ROT_R(X,s) (((X) >> (s)) | ((X) << (32-(s))))


/*
** Haval Boolean Functions
*/
#define F1(x6,x5,x4,x3,x2,x1,x0) \
  (((x5) & (x0)) ^ ((x2) & (x1)) ^ ((x3) & (x1)) ^ ((x6) & (x4)) ^ (x2))

#define F2(x6,x5,x4,x3,x2,x1,x0) \
  (((x6) & (x2) & (x0)) ^ ((x6) & (x5) & (x3)) ^ \
   ((x3) & (x0)) ^ ((x2) & (x1)) ^ ((x5) & (x2)) ^ \
   ((x6) & (x2)) ^ ((x5) & (x3)) ^ ((x6) & (x4)) ^ (x1))

#define F3(x6,x5,x4,x3,x2,x1,x0) \
  (((x6) & (x3) & (x2)) ^ ((x6) & (x0)) ^ ((x2) & (x1)) ^ \
   ((x4) & (x3)) ^ ((x6) & (x5)) ^ (x5))

#define F4(x6,x5,x4,x3,x2,x1,x0) \
  (((x6) & (x5) & (x0)) ^ ((x6) & (x3) & (x1)) ^ ((x5) & (x4) & (x3)) ^ \
   ((x3) & (x0)) ^ ((x5) & (x0)) ^ ((x6) & (x0)) ^ ((x5) & (x1)) ^ \
   ((x6) & (x1)) ^ ((x4) & (x2)) ^ ((x6) & (x4)) ^ ((x6) & (x5)) ^ (x2))


/*
** First 256 bits of the fraction of Pi
*/
Long Digest[8] = {
  0xEC4E6C89,0x082EFA98,0x299F31D0,0xA4093822,
  0x03707344,0x13198A2E,0x85A308D3,0x243F6A88
};

/*
** Next 1024 bits of the fraction of Pi
*/
Long K2[32] = {
  0xC25A59B5,0x7B54A41D,0x82154AEE,0x718BCD58,
  0x728EB658,0x0D95748F,0xF4933D7E,0xA458FEA3,
  0x71574E69,0x636920D8,0x858EFC16,0x0801F2E2,
  0xB3916CF7,0x24A19947,0xF12C7F99,0xBA7C9045,
  0x6A267E96,0xB8E1AFED,0xD01ADFB7,0x2FFD72DB,
  0x98DFB5AC,0xD1310BA6,0x8979FB1B,0x9216D5D9,
  0xB5470917,0x3F84D5B5,0xC97C50DD,0xC0AC29B7,
  0x34E90C6C,0xBE5466CF,0x38D01377,0x452821E6
};

/*
** Next 1024 bits of the fraction of Pi
*/
Long K3[32] = {
  0x6C24CF5C,0xAFD6BA33,0x9B87931E,0xCE5C3E16,
  0x741831FC,0x2BA9C55D,0x636FBC2A,0xB3EE1411,
  0x7C72E993,0xA15486AF,0x1141E8CE,0xB4CC5C34,
  0x2AAB10B6,0x55CA396A,0x63E81440,0x57489862,
  0xAA55AB94,0xE65525F3,0x55605C60,0x78AF2FDA,
  0xBD314B27,0xD71577C1,0xB01E8A3E,0x6C9E0E8B,
  0x603A180E,0x8E79DCB0,0xB8DB38EF,0xCA417918,
  0x286085F0,0xC5D1B023,0x2AF26013,0x9C30D539
};

/*
** Next 1024 bits of the fraction of Pi
*/
Long K4[32] = {
  0x137A3BE4,0x6EEF0B6C,0xAB5133A3,0x960FA728,
```

```
  0xD8542F68,0x6A51A0D2,0xABD388F0,0x670C9C61,
  0xF6E96C9A,0x21C66842,0x9E1F9B5E,0x69C8F04A,
  0xA4842004,0x2E0B4482,0x83F44239,0x0F6D6FF3,
  0xD396ACC5,0x23893E81,0xEB651B88,0xDC262302,
  0xE98575B1,0xEF845D5D,0x5DEC8032,0x487CAC60,
  0xFB21A991,0x61D809CC,0x66282193,0xC4BFE81B,
  0x6B4BB9AF,0x3B8F4898,0x28958677,0x7A325381
};

/*
** Word Processing Orders for Boolean functions H2, H3, H4
*/
int WPH2[32] = {
   5, 14, 26, 18, 11, 28,  7, 16,  0, 23, 20, 22,  1, 10,  4,  8,
  30,  3, 21,  9, 17, 24, 29,  6, 19, 12, 15, 13,  2, 25, 31, 27
};

int WPH3[32] = {
  19,  9,  4, 20, 28, 17,  8, 22, 29, 14, 25, 12, 24, 30, 16, 26,
  31, 15,  7,  3,  1,  0, 18, 27, 13,  6, 21, 10, 23, 11,  5,  2
};

int WPH4[32] = {
  24,  4,  0, 14,  2,  7, 28, 23, 26,  6, 30, 20, 18, 25, 19,  3,
  22, 11, 31, 21,  8, 27, 12,  9,  1, 29,  5, 15, 17, 10, 16, 13
};

/*
**
** Global variables
*/
int  ver = 92;          /* Version number */
int  noPass,            /* No of passes to be processed */
     digLen;            /* Output digest length */

FILE *fpi;

/*
**
** Function prototypes
*/
void UpdatePass1(),
     UpdatePass2(),
     UpdatePass3(),
     UpdatePass4(),
     HavalA(),
     HavalB(),
     HavalTimeTrial(),
     FinalDigest(),
     PrintDigest(),
     PrintDigestFormat(),
     GetDigestFromString(),
     GetDigestFromFile();

int bit32();
void dumpbuf(),gch(),dump32(),dump64(),dump16hex(),dump32hex();

/*
** End of haval.h
**
*/
```

```
/*
** Program: haval.c
** One-way Hashing Algorithm with Variable Length of output.
** Compile: cc -o haval haval.c
** Dzung Le - Aug,92;
**
*/
#include <stdio.h>
#include <time.h>

#include "haval.h"

/*
** Main routine.
**
*/
main(argc,argv)
int argc;
char **argv;
{
  int  c,Fflag,pflag,dflag,tflag,fflag,sflag,errflag;
  char *infile,*instring;
  extern char *optarg;
  extern int   optind;
  char *Format;

  /*
  ** option initializaton
  */
  pflag = dflag = tflag = fflag = sflag = Fflag = errflag = 0;

  /*
  ** Process options
  */
  while (( c = getopt(argc,argv,"tf:p:d:s:F:")) != EOF) {
    switch (c) {
    case 'p': pflag++;
            if (sscanf(optarg,"%d",&noPass) == 0)
              errflag++;
            break;
    case 'd': dflag++;
            if (sscanf(optarg,"%d",&digLen) == 0)
              errflag++;
            break;
    case 's': sflag++;
            instring = optarg;
            break;
    case 'F': Fflag++;
            Format = optarg;
            break;
    case 't': tflag++;
            break;
    case 'f': infile = optarg;
            fflag++;
            break;
    case '?': errflag++;
            break;
    }
  }

  /*
  ** Error checking
  */
  if (!pflag) {
    noPass = 2;
  }
  else if ((noPass < 2) || (noPass > 4)) {
    fprintf(stderr,"\n%s: invalid number of passes\n", argv[0]);
    fprintf(stderr,"      (2, 3, and 4 passes only).\n\n");
    exit(1);
  }

  if (!dflag) {
    digLen  = 256;
  }
```

```
  else if ((digLen != 128) && (digLen != 160) && (digLen != 192) &&
        (digLen != 224) && (digLen != 256)) {
    fprintf(stderr,"\n%s: invalid message digest length\n", argv[0]);
    fprintf(stderr,"      (128, 160, 192, 224 and 256 only).\n\n");
    exit(1);
  }

  if (errflag) {
    fprintf(stderr,"\nUsage: %s -f input_file | -s string | -t ",argv[0]);
    fprintf(stderr,"[ -p pass -d digest_length ]\n\n");
    exit(1);
  }

  /*
  ** Time trial testing
  */
  if (tflag)
    HavalTimeTrial();

  /*
  ** Get message disgest from input file.
  */
  else if (fflag) {
    if ((fpi = fopen(infile,"r")) == NULL) {
      fprintf(stderr,"%s: Couldn't open file %s for input.\n",
        argv[0], infile);
      exit(1);
    }
    else GetDigestFromFile(fpi);
  }

  /*
  ** Get message digest from input string
  */
  else if (sflag)
    GetDigestFromString(instring);

  /*
  ** Get message digest from standard output
  */
  else
    GetDigestFromFile(stdout);

  if (!Fflag)
    PrintDigest(Digest, digLen);
  else
    PrintDigestFormat(Format, Digest, digLen, noPass);
}

/*
** GetDigestFromString() : Get message digest from an input string.
**
*/
void GetDigestFromString(inString)
char *inString;
{
  int  Len, inLen;
  char *inStr = inString;
  Long bits[2];    /* max file length (in bits) 2 powered 64 */

  bits[0] = bits[1] = 0L;

  /*
  ** Process 128-char blocks
  */
  while ((Len = strlen(inStr)) > 0) {

    if (Len >= 128)
      inLen = 128;
    else
      inLen = Len;

    /* increment the number of Len bits */
    if ((bits[0] + (inLen << 3)) < bits[0])
      bits[1]++;
    bits[0] += (inLen << 3);
```

```
    if (Len == 128) {
      HavalA(inStr);
      inStr += 128; /* increment pointer to next block */
    }
    else {
      /*
      ** process the last block (less than 128 chars)
      */
      HavalB(inStr, inLen, bits);
      inStr += Len;
    }

  }

  /*
  ** The length is multiples of 128 chars - process the last block here.
  */
  if (inLen == 128)
    HavalB(inStr, 0, bits);

  /*
  ** Adjust the digest to the required length
  */
  FinalDigest(Digest, digLen);
}

/*
** GetDigestFromFile() : Get message digest from an input file.
**
*/
void GetDigestFromFile(fp)
FILE *fp;
{
  char inStr[130];
  int  Len;
  Long bits[2];

  bits[0] = bits[1] = 0L;

  /*
  ** Process 128-char blocks from input file.
  */
  while ((Len = fread(inStr, 1, 128, fp)) > 0) {

    /* increment the number of Len bits */
    if ((bits[0] + (Len << 3)) < bits[0])
      bits[1]++;
    bits[0] += (Len << 3);

    if (Len == 128)
      HavalA(inStr);
    else
      /* process the last block (less than 128 chars) */
      HavalB(inStr, Len, bits);
  }

  /*
  ** The length is multiples of 128 chars - process the last block here.
  */
  if (Len == 128)
    HavalB(inStr, 0, bits);

  /*
  ** Adjust the digest to the required length
  */
  FinalDigest(Digest, digLen);
}

/*
** HavalTimeTrial() : Timing test for 10-Mb block (real time only).
** For system time, refer to UNIX command timex.
*/
void HavalTimeTrial()
{
  long i, t1, t2;
```

```
  unsigned char inString[128];
  Long bits[2];

  for (i=0;i < 128;i++)
    inString[i] = i;

  bits[0] = (Long) (10 * 1024 * 1024);
  bits[1] = 0L;

  time(&t1);

  /*
  ** Process 10-Mb block (multiple of 128-byte blocks).
  */
  for (i=0; i < 81920;i++)
    HavalA(inString);

  /*
  ** Final block
  */
  HavalB(inString, 0, bits);

  /*
  ** Adjust the digest to the required length
  */
  FinalDigest(Digest,digLen);

  time(&t2);

  fprintf(stdout,"Real Time - seconds to process 10MB block: %d\n", t2-t1);
}

/*
** HavalA() : process 1024-bit blocks.
**
*/
void HavalA(inString)
char *inString;
{
  Long B[32];
  int i;

  /*
  ** Process 1024-bit blocks
  */
  for (i=0;i < 32;i++) {
     B[i] = ((inString[(i*4) + 3] << 24) |
           (inString[(i*4) + 2] << 16) |
           (inString[(i*4) + 1] <<  8) |
           (inString[(i*4) + 0]));
  }

  /*
  ** Update the digest message
  */
  UpdatePass1(Digest,B,noPass);
  UpdatePass2(Digest,B,noPass);
  if (noPass > 2) UpdatePass3(Digest,B,noPass);
  if (noPass > 3) UpdatePass4(Digest,B,noPass);
}

/*
** HavalB() : process a block having no of bits less than 1024.
**
*/
void HavalB(inString,inLen,bits)
char *inString;
int inLen;
Long *bits;
{
  unsigned char buf[128];
  Long B[32];
  int cnt,i;

  for (i = 0;i < inLen;i++) buf[i] = inString[i];
  cnt = inLen;
```

```
/*
** Process last block having less than 1024 bits
*/
if (((cnt << 3) & 0x3FF) <= 0x3B0) {
  /*
  ** append 1, store version, pass, digest length and
  ** process the block.
  */
  buf[cnt] = 0x80;

  for (i=cnt+1;i < 128;i++) buf[i] = 0x0;

  for (i=0;i < 32;i++) {
    B[i] = ((buf[(i*4) + 3] << 24) |
          (buf[(i*4) + 2] << 16) |
          (buf[(i*4) + 1] << 8) |
          (buf[(i*4) + 0]));
  }

  /*
  ** Store version no, pass, digest length, and message length
  */
  B[29] = (Long) (((ver - 91) << 13) | (noPass << 10) | (digLen));
  B[30] = bits[0];
  B[31] = bits[1];

  /*
  ** Update the digest message
  */
  UpdatePass1(Digest,B,noPass);
  UpdatePass2(Digest,B,noPass);
  if (noPass > 2) UpdatePass3(Digest,B,noPass);
  if (noPass > 3) UpdatePass4(Digest,B,noPass);

}
else {
  /*
  ** only append 1 and process half-filled block
  */
  buf[cnt] = 0x80;

  for (i=cnt+1;i < 128;i++) buf[i] = 0x0;

  for (i=0;i < 32;i++) {
    B[i] = ((buf[(i*4) + 3] << 24) |
          (buf[(i*4) + 2] << 16) |
          (buf[(i*4) + 1] << 8) |
          (buf[(i*4) + 0]));
  }

  /*
  ** Update the digest message with half-filled block.
  */
  UpdatePass1(Digest,B,noPass);
  UpdatePass2(Digest,B,noPass);
  if (noPass > 2) UpdatePass3(Digest,B,noPass);
  if (noPass > 3) UpdatePass4(Digest,B,noPass);

  /*
  ** Store version no, pass, digest length, message length in a new block
  ** and process the block.
  */
  for (i=31;i >= 0;i--) B[i] = 0L;

  B[29] = (Long) (((ver - 91) << 13) | (noPass << 10) | (digLen));
  B[30] = bits[0];
  B[31] = bits[1];

  /*
  ** Update the digest message with half-filled block.
  */
  UpdatePass1(Digest,B,noPass);
  UpdatePass2(Digest,B,noPass);
  if (noPass > 2) UpdatePass3(Digest,B,noPass);
  if (noPass > 3) UpdatePass4(Digest,B,noPass);
```

```
  }
}

/*
** UpdatePass1() : Haval update for pass 1.
**
*/
void UpdatePass1(D,W,pass)
Long *D,*W;
int pass;
{
  int i;
  Long P,R,E[8],T[8];

  for (i=0;i < 8;i++) E[i] = D[i];

  /*
  ** Step 1: Modify the initial E
  */
  switch (pass) {
    case 2: for (i=0;i <= 3;i++) E[i] = ROT_R(E[i],13);
            break;
    case 3: for (i=0;i <= 2;i++) E[i] = ROT_R(E[i],13);
            break;
    case 4: for (i=0;i <= 1;i++) E[i] = ROT_R(E[i],13);
            break;
  }

  for (i=0;i < 8;i++) T[i] = E[i];

  /*
  ** Step 2
  */
  for (i=0;i < 32;i++) {

    /* Bit-wise operations with Boolean function F1 */

    P = F1(T[6],T[5],T[4],T[3],T[2],T[1],T[0]);

    R = ROT_R(P,7) + ROT_R(T[7],11) + W[i];

    T[7] = T[6]; T[6] = T[5]; T[5] = T[4]; T[4] = T[3];
    T[3] = T[2]; T[2] = T[1]; T[1] = T[0]; T[0] = R;
  }

  /*
  ** Step 3: word-wise integer addition modulo 2^32
  */
  for (i=0;i < 8;i++)
    D[i] = T[i] + E[i];
}

/*
** UpdatePass2() :  Haval update for pass 2.
**
*/
void UpdatePass2(D,B,pass)
Long *D,*B;
int pass;
{
  int i;
  Long P,R,E[8],T[8],X[32];

  for (i=0;i < 8;i++) E[i] = D[i];

  /*
  ** Re-arrange the word processing order
  */
  for (i=0;i < 32;i++)
    X[0] = B[ WPH2[i] ];

  /*
  ** Step 1: Modify the initial E
  */
  switch (pass) {
    case 2: for (i=4;i <= 7;i++) E[i] = ROT_R(E[i],13);
```

```
        break;
    case 3: for (i=3;i <= 4;i++) E[i] = ROT_R(E[i],13);
        break;
    case 4: for (i=2;i <= 3;i++) E[i] = ROT_R(E[i],13);
        break;
  }

  for (i=0;i < 8;i++) T[i] = E[i];

  /*
  ** Step 2
  */
  for (i=0;i < 32;i++) {

    /*
    ** Bit-wise operations with Boolean function F2
    */
    P = F2(T[6],T[5],T[4],T[3],T[2],T[1],T[0]);

    R = ROT_R(P,7) + ROT_R(T[7],11) + ROT_R(X[i],9) + K2[i];

    T[7] = T[6]; T[6] = T[5]; T[5] = T[4]; T[4] = T[3];
    T[3] = T[2]; T[2] = T[1]; T[1] = T[0]; T[0] = R;
  }

  /*
  ** Step 3: word-wise integer addition modulo 2^32
  */
  for (i=0;i < 8;i++)
    D[i] = T[i] + E[i];
}

/*
** UpdatePass3() : Haval update for pass 3.
**
*/
void UpdatePass3(D,B,pass)
Long *D,*B;
int pass;
{
  int i;
  Long P,R,E[8],T[8],Y[32];

  for (i=0;i < 8;i++) E[i] = D[i];

  /*
  ** Re-arrange the word processing order
  */
  for (i=0;i < 32;i++)
    Y[0] = B[ WPH3[i] ];

  /*
  ** Step 1: Modify the initial E
  */
  switch (pass) {
    case 3: for (i=5;i <= 7;i++) E[i] = ROT_R(E[i],13);
        break;
    case 4: for (i=4;i <= 5;i++) E[i] = ROT_R(E[i],13);
        break;
  }

  for (i=0;i < 8;i++) T[i] = E[i];

  /*
  ** Step 2
  */
  for (i=0;i < 32;i++) {
    /*
    ** Bit-wise operations with Boolean function F3
    */
    P = F3(T[6],T[5],T[4],T[3],T[2],T[1],T[0]);

    R = ROT_R(P,7) + ROT_R(T[7],11) + ROT_R(Y[i],15) + K3[i];

    T[7] = T[6]; T[6] = T[5]; T[5] = T[4]; T[4] = T[3];
    T[3] = T[2]; T[2] = T[1]; T[1] = T[0]; T[0] = R;
```

```
  }

  /*
  ** Step 3: word-wise integer addition modulo 2^32
  */
  for (i=0;i < 8;i++)
    D[i] = T[i] + E[i];
}

/*
** UpdatePass4() : Haval update for pass 4.
**
*/
void UpdatePass4(D,B,pass)
Long *D,*B;
int pass;
{
  int i;
  Long P,R,E[8],T[8],Z[32];

  for (i=0;i < 8;i++) E[i] = D[i];

  /*
  ** Re-arrange the word processing order
  */
  for (i=0;i < 32;i++)
    Z[0] = B[ WPH4[i] ];

  /*
  ** Step 1: Modify the initial E
  */
  if (pass == 4)
    for (i=6;i <= 7;i++) E[i] = ROT_R(E[i],13);

  for (i=0;i < 8;i++) T[i] = E[i];

  /*
  ** Step 2
  */
  for (i=0;i < 32;i++) {
    /*
    ** Bit-wise operations with Boolean function F4
    */
    P = F4(T[6],T[5],T[4],T[3],T[2],T[1],T[0]);

    R = ROT_R(P,7) + ROT_R(T[7],11) + ROT_R(Z[i],23) + K4[i];

    T[7] = T[6]; T[6] = T[5]; T[5] = T[4]; T[4] = T[3];
    T[3] = T[2]; T[2] = T[1]; T[1] = T[0]; T[0] = R;
  }

  /*
  ** Step 3: word-wise integer addition modulo 2^32
  */
  for (i=0;i < 8;i++)
    D[i] = T[i] + E[i];
}

/*
** FinalDigest() : final processing to get the required digest length.
**
*/
void FinalDigest(D,Len)
Long *D;
int Len;
{
  switch (Len) {

    case 128:
      D[3] += (Long) (((D[7])      & 0xFF000000) |
                      ((D[6])      & 0xFF0000) |
                      ((D[5])      & 0xFF00) |
                      ((D[4])      & 0xFF));
      D[2] += (Long) (((D[7] << 8) & 0xFF000000) |
                      ((D[6] << 8) & 0xFF0000) |
                      ((D[5] << 8) & 0xFF00) |
```

```
                ((D[4] >> 24) & 0xFF));
        D[1] += (Long) (((D[7] << 16) & 0xFF000000) |
                ((D[6] << 16) & 0xFF0000) |
                ((D[5] >> 16) & 0xFF00) |
                ((D[4] >> 16) & 0xFF));
        D[0] += (Long) (((D[7] << 24) & 0xFF000000) |
                ((D[6] >>  8) & 0xFF0000) |
                ((D[5] >>  8) & 0xFF00) |
                ((D[4] >>  8) & 0xFF));
        D[4] = D[5] = D[6] = D[7] = 0L;

        break;

      case 160:

        D[4] += (Long) ((((D[7] >> 25) & 0x7F) << 13) |
                (((D[6] >> 19) & 0x3F) <<  7) |
                 ((D[5] >> 12) & 0x7F));
        D[3] += (Long) ((((D[7] >> 19) & 0x3F) << 13) |
                (((D[6] >> 12) & 0x7F) <<  6) |
                 ((D[5] >>  6) & 0x3F));
        D[2] += (Long) ((((D[7] >> 12) & 0x7F) << 12) |
                (((D[6] >>  6) & 0x3F) <<  6) |
                 ((D[5] >>  0) & 0x3F));
        D[1] += (Long) ((((D[7] >>  6) & 0x3F) << 13) |
                (((D[6] >>  0) & 0x3F) <<  6) |
                 ((D[5] >> 25) & 0x7F));
        D[0] += (Long) ((((D[7] >>  0) & 0x3F) << 13) |
                (((D[6] >> 25) & 0x7F) <<  6)|
                 ((D[5] >> 19) & 0x3F));
        D[5] = D[6] = D[7] = 0L;

        break;

      case 192:

        D[5] += (Long) ((((D[7] >> 26) & 0x3F) << 5) |
                 ((D[6] >> 21) & 0x1F));
        D[4] += (Long) ((((D[7] >> 21) & 0x1F) << 5) |
                 ((D[6] >> 16) & 0x1F));
        D[3] += (Long) ((((D[7] >> 16) & 0x1F) << 6) |
                 ((D[6] >> 10) & 0x3F));
        D[2] += (Long) ((((D[7] >> 10) & 0x3F) << 5) |
                 ((D[6] >>  5) & 0x1F));
        D[1] += (Long) ((((D[7] >>  5) & 0x1F) << 5) |
                 ((D[6] >>  0) & 0x1F));
        D[0] += (Long) ((((D[7] >>  0) & 0x1F) << 6)|
                 ((D[6] >> 26) & 0x3F));
        D[6] = D[7] = 0L;

        break;

      case 224:

        D[6] += (Long) ((D[7] >>  0) & 0xF);
        D[5] += (Long) ((D[7] >>  4) & 0x1F);
        D[4] += (Long) ((D[7] >>  9) & 0xF);
        D[3] += (Long) ((D[7] >> 13) & 0x1F);
        D[2] += (Long) ((D[7] >> 18) & 0xF);
        D[1] += (Long) ((D[7] >> 22) & 0x1F);
        D[0] += (Long) ((D[7] >> 27) & 0x1F);
        D[7] = 0L;

        break;
    }
  }

/*
** PrintWord() : print a 32-bit word in hexadecimal format.
**
*/
void PrintWord(Word)
Long Word;
{
  fprintf(stdout,"%02x%02x%02x%02x",
    (unsigned char)((Word >> 24) & 0xFF),
```

```
    (unsigned char)((Word >> 16) & 0xFF),
    (unsigned char)((Word >>  8) & 0xFF),
    (unsigned char)( Word & 0xFF));
}

/*
**
** Print out the digest with the user-supplied format
*/
void PrintDigestFormat(Format,Digest,digLen,noPass)
char *Format;
Long *Digest;
int digLen;
{
  int i=0;

  while (i < strlen(Format)) {
    if (*(Format + i) != '%') {
      fprintf(stdout,"%c", *(Format + i));
      i++;
    }
    else {
      switch (*(Format + i + 1)) {
        case 'h': PrintDigest(Digest, digLen);
                  break;
        case 'a': fprintf(stdout,"haval");
                  break;
        case 'd': fprintf(stdout,"%d",digLen);
                  break;
        case 'p': fprintf(stdout,"%d",noPass);
                  break;
        case 'n': fprintf(stdout,"\n");
                  break;
        case 't': fprintf(stdout,"\t");
                  break;
        case '%': fprintf(stdout,"%%");
                  break;
      }
      i += 2;
    }
  }
  fprintf(stdout,"\n");
}

/*
** PrintDigest() : print out the digest with the specified length.
**
*/
void PrintDigest(D,digLen)
Long *D;
int digLen;
{
  int i;

  switch (digLen) {

    case 256: PrintWord(D[7]);

    case 224: PrintWord(D[6]);

    case 192: PrintWord(D[5]);

    case 160: PrintWord(D[4]);

    case 128: for (i=3;i >= 0;i--)
                PrintWord(D[i]);
              fprintf(stdout,"\n");

              break;

    default : break;
  }
}
/*
** End of haval.c
**
```

```
*/
```