Contemporary Engineering Sciences, Vol. 2, 2009, no. 5, 209 - 238

On the Security and Performance of the FDE Block Cipher

Alireza Shafieinejad
Faramarz Hendessi

Dept. of Electrical and Computer Engineering
Isfahan University of Technology, Isfahan, Iran

T. Aaron Gulliver

Dept. of Electrical and Computer Engineering
University of Victoria, Victoria, BC Canada
agullive@ece.uvic.ca

Abstract

FDE is a block cipher which has a Substitution-Permutation (SP) network structure. In this paper, the security and performance of this cipher are discussed. We show that FDE has a good security margin against differential and linear attacks. The best differential 15-round characteristic of FDE has a probability of success of less than 2^{-140}. Moreover, for linear cryptanalysis we show that the best linear probability of success is less than 2^{-55}, which requires at least 2^{110} plaintext-ciphertext pairs, which is greater than all possible pairs. We also present two fast software implementations of FDE which eliminate the permutations in the round functions. In the first method, this is achieved by merging the permutations with the s-box tables, which necessitates that all permutations in the round functions are located after the substitutions. This is achieved by moving the P^{-1} permutations from the XOR functions of each round to the previous round. The second implementation uses bitsliced processing in which the data is mapped to a new space where the permutations are simply done by register renaming. Versions of both implementations are shown to be 2 to 4 times faster than the original FDE implementation.

Mathematics Subject Classification: 94A60

Keywords: DES, Differential attack, Linear attack, Bitslicing

1 Introduction

Fast Data Encryption (FDE) was invented by Hendessi and Aref in 1988 and introduced in [1,2,3]. The FDE structure was originally designed based on the principles of the Data Encryption Standard (DES), and has been the subject of continued research for over 10 years [1-9]. In the first version of FDE, the substitution boxes were not identified, but their criteria was introduced [1,2,3]. In subsequent papers [4,5,6], strong s-boxes were created. In [5] an algorithm was presented to construct strong s-boxes and to complete the FDE structure.

In this paper, we consider the security and performance of FDE. For security, we concentrate on differential and linear attacks. As in other papers [10], we try to find the best differential and linear probabilities, and determine whether they are less than the probability of exhaustive search, or the available number of plaintext-ciphertext pairs. Since the s-boxes are key dependent, we employ a worst case assumption for the differential characteristics.

To improve the performance of FDE, we consider two approaches which also apply to DES. The first is to merge the permutations with the s-box tables, and the second is bitslicing, which transfers the data to a new space [21]. In the first method, the challenge is that the P^{-1} permutations are located before the substitution boxes, and so cannot be merged with them. We rearrange the structure of FDE so that all permutations are placed after the substitutions. In the second method, we consider bitslicing and analyze several approaches to s-box representation. Conventional methods represent the s-boxes as a function of the inputs [21]. We propose a new method which combines lookup tables and a boolean function. It is shown that this method is superior to other approaches.

As a first step, the FDE structure and key scheduling algorithm are introduced in Sections 2.1 and 2.2. In Section 2.2.3, we describe the ideas behind FDE, and compare FDE with some AES candidates, many of which were designed after FDE. We analyze the differential and linear attacks in Sections 3.1 and 3.2, respectively. In Section 4.1 we discuss the round structure problem with respect to eliminating the permutations, and then create a new structure for the FDE rounds which merges the permutations with the s-box tables. In Section 4.2, we employ bitslicing for FDE implementation. Various bitslicing approaches are proposed and the results compared.

The new approaches employed to implement FDE efficiently in software such as rearrangement of the round structure and combining lookup tables with bitslicing, can be extended and used for other algorithms (both block and stream ciphers).

2 The Fast Data Encryption (FDE) Structure

In this section we present an overview of the FDE structure. A more detailed description can be found in [1-9].

2.1 Functional Structure of FDE

The even and odd FDE round functions are shown in Figs. 1 and 2, respectively. In these figures, L_i and R_i are the first and last 32 bits of the 64 bit data block in the i-th round, respectively. The figures show that the round process is done in two stages. First, in either G or G', the data is XORed with the 32 bit partial keys X_{3i}, X_{3i+1} and X_{3i+2}. Then in the second stage, substitution and permutation are done in F using the 16 bit keys Z_{2i} and Z_{2i+1}. Note that in each round, five partial keys derived from the 128 bit main key are employed.

The substitution and permutation operations are applied concurrently on each of the two halves of the data block. On the right, S and P are employed while on the left, their inverses S^{-1} and P^{-1} are used. As in DES, S is a substitution network that consists of eight 6×4 balanced s-boxes that map a 32 bit input to a 32 bit output according to a 16 bit key Z as shown in Fig. 3. In the F function, P is the same as in DES.

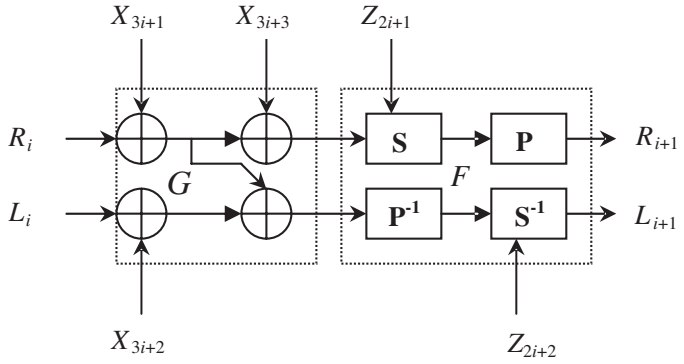

Figure 1: An even round in FDE.

The complete FDE structure is illustrated in Fig. 4, and consists of an initial permutation, 15 rounds with G' at the end of the 15th round, and a final permutation. The G, G', F, T and hl functions are defined as

$$
\begin{aligned}
&G_{X_{i+2},X_{i+1},X_i}(R,L) = (R \oplus X_i \oplus X_{i+2}, L \oplus R \oplus X_i \oplus X_{i+1}) \\
&G'_{X_{i+2},X_{i+1},X_i}(R,L) = (R \oplus L^{\oplus} X_i \oplus X_{i+1}, L \oplus X_i \oplus X_{i+2}) \\
&F_{Z_{i+1},Z_i}(R,L) = (P(S_{Z_i}(R)), S^{-1}_{Z_{i+1}}(P^{-1}(L))) \\
&T(R,L) = (L,R) \quad T \circ T(R,L) = (R,L) \quad \Rightarrow \quad T^{-1} = T \\
&hl(r_{63}r_{62}r_{61}\cdots r_2 r_1 r_0) = (r_{63}r_{62}\cdots r_{33}r_{32}, r_{31}r_{30}\cdots r_1 r_0)
\end{aligned}
\tag{1}
$$

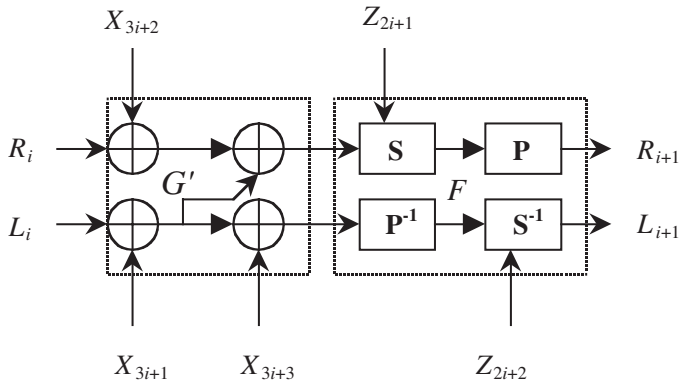

Figure 2: An odd round in FDE.

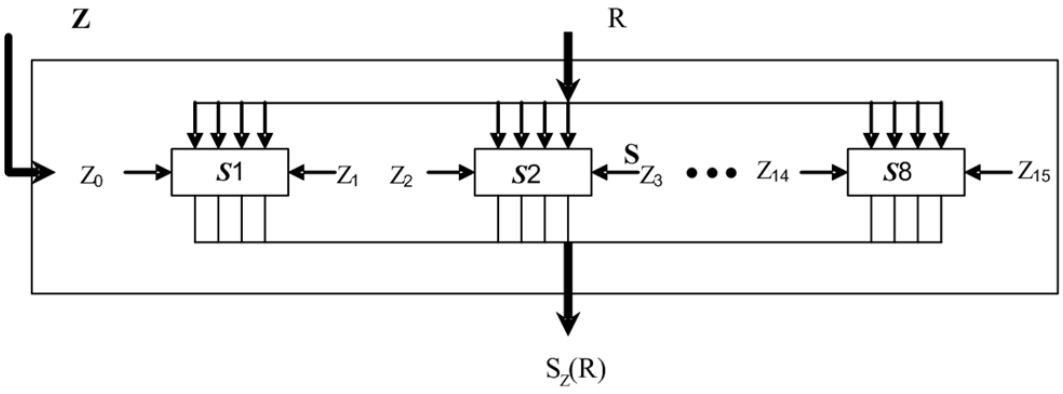

Figure 3: The FDE S structure.

The $\circ$ operator denotes the combination of two functions

$$f \circ g(x) = f(g(x))$$

T is a function that exchanges the two halves of a 64 bit block and is used after the initial permutation. Obviously the inverse of T is equal to itself. hl is a function that converts one 64 bit block into two 32 bit blocks. The inverse of hl merges a pair of 32 bit blocks into one 64 bit block. Consequently, FDE encryption with these definitions is

$$\begin{aligned}&\text{Encrypt}(P) = IP^{-1} \circ hl^{-1} \circ G'_{X_{48},X_{47},X_{46}} \circ F_{Z_{30},Z_{29}} \circ G_{X_{45},X_{44},X_{43}} \circ \cdots \circ \\ &F_{Z_4,Z_3} \circ G'_{X_6,X_5,X_4} \circ F_{Z_2,Z_1} \circ G_{X_3,X_2,X_1} \circ T \circ hl \circ IP(P)\end{aligned} \tag{2}$$

Similar to Feistel networks it can be shown that decryption is the same as encryption except that the order of the partial keys is reversed

$$\begin{aligned}&\text{Decrypt}(C) = IP^{-1} \circ G'_{X_1,X_2,X_3} \circ F_{Z_1,Z_2} \circ G_{X_4,X_5,X_6} \circ F_{Z_3,Z_4} \circ \cdots \circ \\ &G_{X_{43},X_{44},X_{45}} \circ F_{Z_{29},Z_{30}} \circ G'_{X_{46},X_{47},X_{48}} \circ IP(C)\end{aligned} \tag{3}$$

2.2 The FDE Key Scheduling Structure

Categorizing the keys in each round, the following vectors are defined

$$\begin{aligned}&U_{2i+1} = (X_{3i+1}, X_{3i+2}) && i = 0,\ 1,\ \ldots\ 7 \\ &U_{2i+2} = (X_{3i+3}, Z_{2i+1}, Z_{2i+2}) && i = 0,\ 1,\ \ldots\ 6 \\ &U_{30-2i} = (X_{48-3i}, X_{47-3i}) && i = 0,\ 1,\ \ldots\ 7 \\ &U_{29-2i} = (X_{46-3i}, Z_{30-2i}, Z_{29-2i}) && i = 0,\ 1,\ \ldots\ 6 \\ &V = (X_{24}, Z_{15}, Z_{16}, X_{25})\end{aligned} \tag{4}$$

Each U_i is a 64 bit vector and V is a 96 bit vector. With these definitions, FDE is simplified to

$$C = \text{FDE}_{U_1,U_2,U_3,\ldots,U_{15},V,U_{16},\ldots,U_{30}}(P) \tag{5}$$

The key generation algorithm is shown below. U is the 128-bit main key while X and Y are the first and the last 32 bits of U, respectively.

$$\begin{aligned}&V = \text{Choose\_96bit}(U) \\ &U_1 = \text{FDE}_{Y,Y,\ldots,Y,V,Y,\ldots,Y}(X) \\ &U_2 = \text{FDE}_{U_1,Y,\ldots,Y,V,Y,\ldots,Y}(X) \\ &U_3 = \text{FDE}_{U_1,U_2,Y,\ldots,Y,V,Y,\ldots,Y}(X) \\ &U_{30} = \text{FDE}_{U_1,U_2,\ldots,U_{15},V,U_{16},\ldots,U_{29},Y}(X)\end{aligned} \tag{6}$$

Choose_96bit is a selection function which chooses 96 bits from the 128 bits (U) for the partial key V.

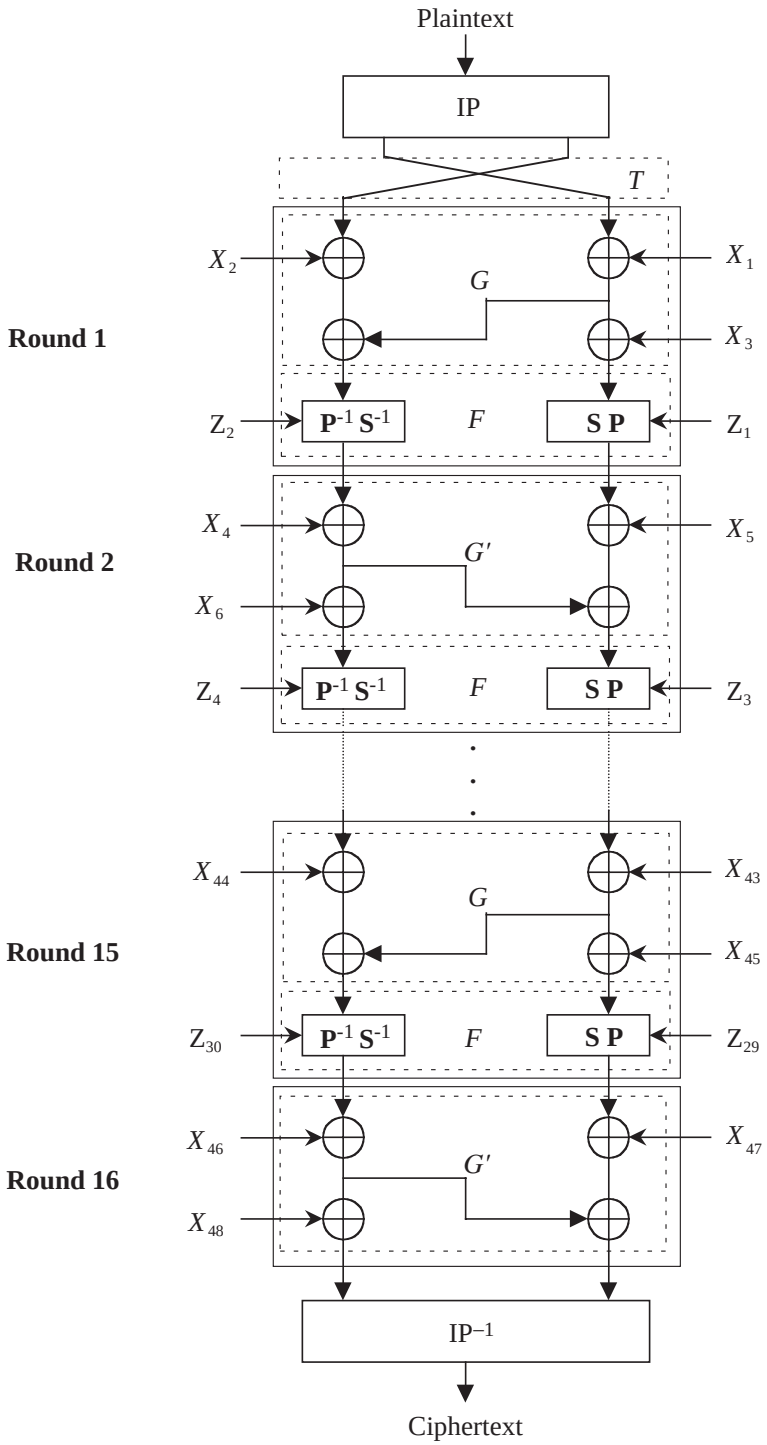

Figure 4: The FDE sturcture.

2.3 A Comparison Between FDE and Some AES Candidates

2.3.1 Round function

FDE is a substitution-permutation network. Unlike a Feistel network where the round function is applied to a part of a block, in FDE the round function is applied to all input bits. In addition, in each round FDE uses an s-box and its inverse ($S_1 - S_8$ and $S_1^{-1} - S_8^{-1}$) to create more confusion per round than a Feistel network. Serpent [10] uses the same idea and applies substitution to all input bits by means of a set of eight 4x4 s-boxes in each round. Rijndal [11] and Crypton [12] have one and two 8x8 bijective s-boxes, respectively, and apply substitution to all input bytes in each round. In SAFER [13], confusion is done by exponentiation and logarithmic operations. In each round, both functions are applied to all input bytes.

2.3.2 Key Scheduling Algorithm

Key scheduling is an extremely important aspect of cipher design since suboptimal key schedules can lead to exploitable weaknesses in the cipher such as weak keys, equivalent keys, complementation properties, and susceptibility to related-key attacks. Moreover, overly complicated key schedules can lead to prohibitively long setup times which limit cipher use in some environments such as IPSec. Considering this tradeoff, FDE was designed for key strength, so complexity was not the major concern. This goal was achieved by using FDE encryption to generate the round keys, which was a new approach when FDE was developed. As FDE has 15 rounds and each round uses a subkey of 128 bits, the entire set of subkeys can be computed by 30 block encryptions.

The use of encryption elements to make complex key scheduling has been done in various ways. CAST256, E2 and LOKI97 [14] use the complete structure of the round function to produce subkeys [15,16]. The CAST256 key scheduling algorithm uses 192 round functions which equals 4 block encryptions (128 bits). E2 uses 72 round functions, which is equivalent to 6 block encryptions (128 bits). LOKI97 uses 3 round functions to generate each round key, thus the key computations equal 3 block encryptions (128 bits). Serpent only uses the substitution part of the round function to generate subkeys [10]. To generate all round keys requires 33 s-boxes, which equals about one block encryption.

The reasons for increasing the complexity of key scheduling algorithms are:

- Make deriving subkeys or the master key from other subkeys computationally infeasible.
- Ensure all bits of the master key equally influence all bits of the subkeys.

- Increase confidence that the set of key values will appear to be pair-wise independent under statistical analysis.
- Ensure attacks that derive a set of round keys require a computational effort similar to that of deriving the initial key.

2.3.3 Keyed s-boxes

FDE uses the idea of "keyed s-boxes" or "key dependent s-boxes". It has 8 s-boxes of size 6×4, where 2 input bits of each s-box come from the round subkey and the others from the data itself. Each 4×4 subbox of the 6×4 s-box is a bijective mapping.

The benefits of keyed s-boxes are:

- Fixed s-boxes (e.g., as in DES) allow an attacker to study the s-boxes and find weak points.
- With key-dependent s-boxes, an attacker does not know what the s-boxes are.
- Defense against "unknown attacks."
- Complexity of keyed s-boxes depends on the length of the key.

Of the AES candidates, only Twofish [17] employs this idea. It builds four bijective key-dependent 8×8-bit S-boxes using a key/permutation "sandwich", as shown below.

$$\begin{aligned} S_0(x) &= q_1[q_0[q_0[x]\hat{}k_0]\hat{}k_1] \\ S_1(x) &= q_0[q_0[q_1[x]\hat{}k_2]\hat{}k_3] \\ S_2(x) &= q_1[q_1[q_0[x]\hat{}k_4]\hat{}k_5] \\ S_3(x) &= q_0[q_1[q_1[x]\hat{}k_6]\hat{}k_7] \end{aligned}$$

In the above expressions q_0 and q_1 are two fixed 8-bit permutations. The downside of this structure is the setup time to build the s-boxes. In Q [18], which is a proposal for the European NESSIE project, the concept of a key dependent s-box is employed in that the key mixing operations surrounding an s-box equates to a keyed s-box.

2.3.4 Balanced Encryption and Decryption

FDE inherits a property of traditional Feistel networks, namely encryption and decryption are identical except for the order in which the round keys are employed. This property eliminates the need for a separate structure for decryption and makes it possible to use the same structure for encryption and decryption in both hardware and software implementations. This a valuable

feature for constrained platforms, especially for hardware or firmware implementations that are very resource limited, and software implementations such as on smartcards or microchips where memory (especially RAM) is an expensive and limited resource. In addition, encryption and decryption run at the same speed. CAST256 and CRYPTON are not balanced, which is a serious concern.

3 Cryptanalysis

3.1 Differential Cryptanalysis

In FDE, the nonlinear part of the round function is F. By removing the linear portion of the round functions, the differential characteristics of the FDE rounds can be obtained. FDE has keyed s-boxes in the S and S^{-1} functions. For simplicity we ignore the fact that they are keyed s-boxes, and assume that FDE has only 4×4 s-boxes which have the maximum differential properties, i.e.

$$P_{S[i]}(a,b) = \max_{j=0}^{3} P_{S[i,j]}(a,b) \tag{7}$$

In (7) $S[i]$ indicates the i-th 6×4 s-box, $S[i,j]$ indicates the j-th subbox of $S[i]$, and $P(a,b)$ denotes the differential properties of the pair (a,b).

3.1.1 The best round characteristics

In this section we try to find a tight bound for the best differential round characteristics. First we focus on the 2-round characteristic and count the minimum number s-boxes that are activated by changing a single bit in the input of a 2-round FDE (branch factor). We assume that our s-boxes and their inverses have the property that a change in one input bit will never lead to a one-bit output difference. We found that this property is critical for the FDE structure to increase the number of active s-boxes. Actually this property was not considered in the design of the original FDE s-boxes. However, s-boxes with this property can be constructed without compromising security. All DES 4×4 s-boxes and Serpent s-boxes have this differential property [10].

We use a solid dot for a bit change and a solid rectangle for an active s-box. Figs. 5 and 6 show the data flow of the bits when one input bit (either top or bottom) is changed. Fig. 5 shows that with a change at the R-side, both sides of the data block will be changed after two rounds. There will be four active s-boxes, one in the first layer and the others in the second layer. When a change occurs at the L-side, the minimum number of active s-boxes cannot easily be determined because the XOR branches sometimes cause the state change to vanish. We found by exhaustive search that the minimum number

of active s-boxes is 7. Thus the minimum number of active s-boxes occurs with R-side changes. We extended this approach up to five rounds by exhaustive

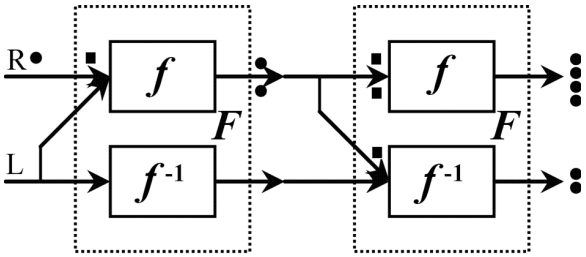

Figure 5: Data flow by changing R-side bits ($f = P \circ S$).

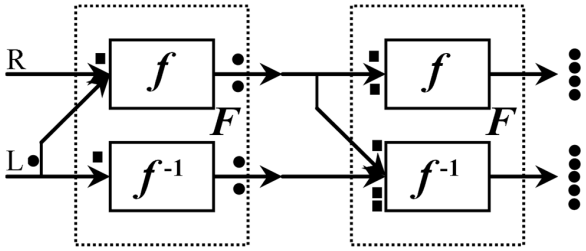

Figure 6: Data flow by changing L-side bits ($f = P \circ S$).

search and obtained the best differential round probability assuming that DES s-boxes are used for FDE. The results including the number of active s-boxes in each round and the differential probabilities are shown in Table 1. For the case of four and five round we used a pruned method to search all cases.

Based on the bound achieved for five rounds, the bound for the full 15-round differential characteristic of FDE is obtained from

$$\begin{aligned} P_{FDE5} &\leq 2^{-46.8} \\ P_{FDE15} &\leq (P_{FDE5})^3 = 2^{-140.4} \end{aligned} \tag{8}$$

This probability is less than that for an exhaustive search. In the actual case where the s-boxes are selected based on the subkeys Z_i, the probabilities will be lower. To show this, we repeat the previous computations with the assumption that one of four possible subboxes is selected. The results are shown in Table 2, and show that in all cases we have a probability of less than $2^{-45.4}$. This is sufficient to state that the round probability of reduced 12-round FDE is less than that for exhaustive search. Therefore the condition stated in (7) is valid.

Table 1: Differential Round Characteristics of Reduced 1- to 5-Round FDE

Round	Number of Active s-boxes	Differential Probabilities in the Rounds	Total Active s-boxes	Overall Probability
1	1	2^{-1}	1	2^{-1}
2	1, 3	$2^{-1}, 2^{-9}$	4	2^{-10}
3	1, 3, 3	$2^{-1.5}, 2^{-5.8}, 2^{-9.4}$	7	$2^{-16.7}$
4	1, 3, 4, 2	$2^{-1}, 2^{-9.4}, 2^{-11.4}, 2^{-3.4}$	10	$2^{-25.2}$
5	1, 3, 3, 5, 6	$2^{-2}, 2^{-10}, 2^{-7}, 2^{-14.4}, 2^{-13.4}$	18	$2^{-46.8}$

Note that in our analysis, the use of key dependent s-boxes does not improve the security margins against differential and linear (which is proved in the same way as above) attacks. However, replacing a 6×4 s-box with 4×4 s-boxes, in addition to providing provable security, allows for better software and hardware implementation.

3.2 Linear Cryptanalysis

Linear cryptanalysis exploits the probability of occurrence of linear expressions involving plaintext bits, ciphertext bits (actually the 2nd last round output), and subkeys bits. It is a known plaintext attack that is premised on the attacker having information on a set of plaintexts and the corresponding ciphertexts [19, 20]. The basic idea is to approximate the operation of a portion of the encryption with an expression that is linear, where linearity refers to mod-2 bit-wise operations. Such an expression has the form

$$X_{i_1} \oplus X_{i_2} \cdots \oplus X_{i_u} \oplus Y_{j_1} \oplus Y_{j_2} \cdots \oplus Y_{j_v} = 0 \tag{9}$$

where X_i represents the i-th bit of the input $X = [X_1, X_2, \ldots]$ and Y_j represents the j-th bit of the output $Y = [Y_1, Y_2, \ldots]$. Equation (9) represents the exclusive-OR of u input bits and v output bits.

3.2.1 Piling-up principle

Before considering the construction of linear expressions, we present some basic tools. Consider two random binary variables, X_1 and X_2. We begin by noting that the relationship $X_1 \oplus X_2 = 0$ is a linear expression which is equivalent to $X_1 = X_2$ and $X_1 \oplus X_2 = 1$ is equivalent to $X_1 \neq X_2$. Now, assume that the

Table 2: Differential Round Characteristics of Reduced 4-round FDE

Assumption	Number of Active s-boxes	Differential Probabilities in the Rounds	Total Active s-boxes	Overall Probability
$P_{S[i]}(a,b) = P_{S[i,0]}(a,b)$	1, 3, 4, 8	$2^{-1.4}, 2^{-9.4}, 2^{-13}, 2^{-27}$	16	$2^{-50.8}$
$P_{S[i]}(a,b) = P_{S[i,1]}(a,b)$	1, 3, 5, 8	$2^{-3}, 2^{-11}, 2^{-14}, 2^{-25}$	17	2^{-53}
$P_{S[i]}(a,b) = P_{S[i,2]}(a,b)$	1, 3, 5, 6	$2^{-2}, 2^{-10}, 2^{-15.4}, 2^{-18}$	15	$2^{-45.4}$
$P_{S[i]}(a,b) = P_{S[i,3]}(a,b)$	1, 3, 5, 8	$2^{-1.4}, 2^{-9.4}, 2^{-16}, 2^{-26}$	17	$2^{-52.8}$
$P_{S[i]}(a,b) = \left\lceil \frac{1}{4} \sum_{j=0}^{3} P_{S[i,j]}(a,b) \right\rceil$	1, 3, 5, 8	$2^{-1.4}, 2^{-9.4}, 2^{-16}, 2^{-26}$	17	$2^{-52.8}$

probability distributions are given by

$$\begin{aligned} \Pr(X_1 = i) &= \begin{cases} p_1 & i = 0 \\ 1 - p_1 & i = 1 \end{cases} \\ \Pr(X_2 = i) &= \begin{cases} p_2 & i = 0 \\ 1 - p_2 & i = 1 \end{cases} \end{aligned} \tag{10}$$

If the two random variables are independent, we have

$$\begin{aligned} &\Pr(X_1 \oplus X_2 = 0) \\ &= \Pr(X_1 = X_2) \\ &= \Pr(X_1 = 1, X_2 = 1) + \Pr(X_1 = 0, X_2 = 0) \\ &= p_1 p_2 + (1 - p_1)(1 - p_2) \end{aligned} \tag{11}$$

Another approach is to let $p_1 = 1/2 + \varepsilon_1$ and $p_2 = 1/2 + \varepsilon_2$, where ε_1 and ε_2 are the probability biases with $-1/2 \leq \varepsilon_1, \varepsilon_2 \leq 1/2$. Hence, it follows that

$$\Pr(X_1 \oplus X_2 = 0) = 1/2 + \varepsilon_1 \varepsilon_2$$

and the bias of $X_1 \oplus X_2 = 0$ is

$$\varepsilon_{12} = 2\varepsilon_1 \varepsilon_2$$

***Piling-up Lemma*(Matsui [19])**

The above result can be extended to n independent random binary variables, $X_1, X_2, \ldots, X_n$, giving

$$\Pr(X_1 \oplus X_2 \cdots \oplus X_n = 0) = 1/2 + 2^{n-1} \prod_{i=1}^{n} \varepsilon_i \tag{12}$$

or equivalently

$$\varepsilon_{12\cdots n} = 2^{n-1} \varepsilon_1 \varepsilon_2 \cdots \varepsilon_n \tag{13}$$

3.2.2 The best linear probability

We assume that the FDE s-boxes are such that their linear probability is bounded by $\frac{1}{2} \pm \frac{1}{4}$, and for a single input bit and a single output bit is $\frac{1}{2} \pm \frac{1}{8}$. We denote the latter as p_{1-1} and its bias as ε_{1-1}. All Serpent s-boxes have the linear properties [10]

$$p_{1-1} = \frac{1}{2} \pm \frac{1}{8} \quad \varepsilon_{1-1} = \frac{1}{8} \tag{14}$$

We search for the case that there is a minimum number of active s-boxes in the linear relation. As far as swapping blocks in each round is concerned, the minimum number of active s-boxes is two, except for the first round in which there may only be one active s-box. In other words, in each half of the blocks, only one bit to one bit relations lead to a minimum number of active s-boxes, which is the best case for an attacker.

If we assume that in any round of FDE two linear approximations between input bit i_1 and output bit j_1, and between input bit i_2 and output bit j_2, have occurred, we have

$$\begin{array}{rcl} p_{FDE-one-round} & = & P(X_{i_1} \oplus X_{i_2} \oplus Y_{j_1} \oplus Y_{j_2} = 1) \\ & = & P(X_{i_1} \oplus Y_{j_1} = X_{i_2} \oplus Y_{j_2}) \\ & = & \frac{1}{2} \pm 2\varepsilon_{i_1,j_1}\varepsilon_{i_2,j_2} = \frac{1}{2} \pm 2(\varepsilon_{1-1})^2 = \frac{1}{2} \pm \frac{1}{32} \\ \varepsilon_{FDE-one-round} & = & \pm\frac{1}{32} \end{array} \tag{15}$$

If in the first round only one s-box is active, for the entire FDE algorithm we have

$$\begin{array}{rcl} P_{FDE-15-rounds} & = & \frac{1}{2} \pm 2^{14} \varepsilon_{FDE-first-round} \times (\varepsilon_{FDE-one-round})^{13} \\ & = & \frac{1}{2} \pm 2^{14-3-13*5} = \frac{1}{2} \pm 2^{14-3-13*5} = \frac{1}{2} \pm 2^{-55} \\ N_{Plain-cipher} & = & (2^{-55})^{-2} = 2^{110} \end{array} \tag{16}$$

In fact, the actual probability will be less than this number because the number of active s-boxes is more than 29. If we consider situations that relate to more than one bit relation, there will be even more active s-boxes, so the round characteristics will be lower. Since this amount of ciphertext-plaintext is greater than the total number of pairs for a fixed key (2^{64}), FDE is secure against a linear attack.

4 FDE Implementation

FDE rounds consist of two main parts, the linear part G and G', and the nonlinear part F which contains the components S, P, S^{-1} and P^{-1}. G and G' can be simply implemented using three XOR instructions, but in the remainder of the round function, the operations are bit oriented and these are unavailable in general purpose processors. Thus in F, and therefore in S, P, S^{-1} and P^{-1}, these bit logical and shift operations lead to a large number of instructions. The number of operations for S and P are

Cost[S] = 16 AND + 24 SHIFT + 8 OR + 8 MemRef

Cost[P] = 32 OR + 32 AND + 32 SHIFT

The costs for S^{-1} and P^{-1} are the same as for S and P, respectively. IP and IP^{-1} have a regular structure and can be implemented with 25 instructions (15 XORs and 10 shifts).

Cost[IP] = 15 XOR + 10 SHIFT

Thus the cost of block encryption for a 64 bit word is

Cost[FDE] = 8(Cost[G] + Cost[G']) + 15(Cost[P] + Cost[S] + Cost[P^{-1}] + Cost[S^{-1}]) + Cost[IP] + Cost[IP^{-1}]

= 4370 LOGIC + 240 MemRef

This shows that P, S and their inverses are FDE bottlenecks. Therefore, any reduction in their number of instructions leads to a higher encryption rate.

4.1 Fast implementation (non-bitslicing)

In this section, some new techniques are presented to eliminate P and reducing the instructions for S.

4.1.1 Changing 6×4 s-boxes into equivalent 6×32 s-boxes

In the execution of S, after the lookup of each S_i (ith s-box), the output must be shifted $4i-4$ bits to the left to be in the correct position in the output block. By increasing the size of the s-boxes to 6×32 and shifting each S_i element $4i-4$ bit to the left, operations can be saved.

4.1.2 Permutation elimination

The permutation that follows the substitution can be removed by merging it into the s-box tables. This can be simply done for the right branch of FDE provided that the values in the s-box tables are changed to the permuted values. In other words, $S_i[x][y]$ must be changed to $P(S_i[x][y])$. In the fast DES implementation, this optimization is the most important [22]. This elimination is not feasible for the left branch of FDE because of the restriction on the permutation which is located before substitution. However, P^{-1} can be

eliminated, but we must change the position of P^{-1} so that it is located after the substitution.

It is possible to put P^{-1} after S^{-1} in the previous round since the only operator in the P^{-1} to S^{-1} path is XOR, which is a linear function, so that

$$P^{-1}(X \oplus Y) = P^{-1}(X) \oplus P^{-1}(Y)$$

Considering the structure of the original even rounds of FDE (Fig. 1), this process is shown first in Fig. 7 by combining the XOR operations in G, and then in Fig. 8 by removing P^{-1} from the output and applying it to the input branches. The same process can be done for the odd rounds in FDE, as

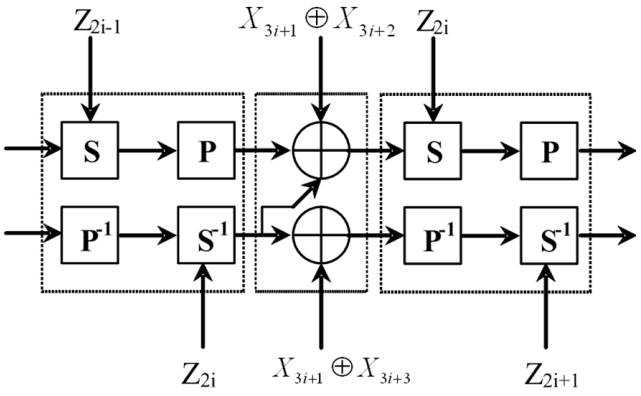

Figure 7: Even rounds of FDE combining the XOR operations in G.

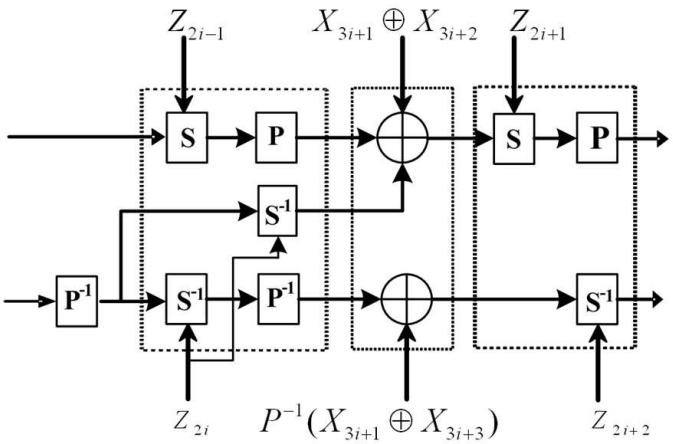

Figure 8: P^{-1} moved to the left for the even rounds in FDE.

shown in Figs. 9 and 10, respectively, to obtain to an $S^{-1} \circ P^{-1}$-free structure. The structures in Figs. 8 and 10, which are the even and odd rounds in FDE (except the last ones), will be denoted by F_o and F_e, and are shown in Figs. 11 and 12, respectively.

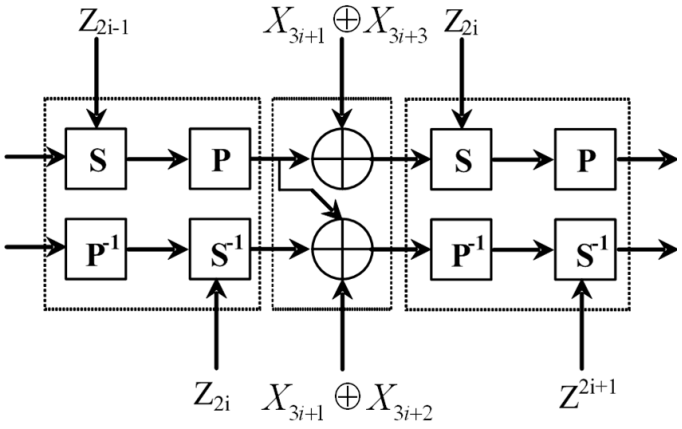

Figure 9: Odd rounds in FDE combining the XOR operations.

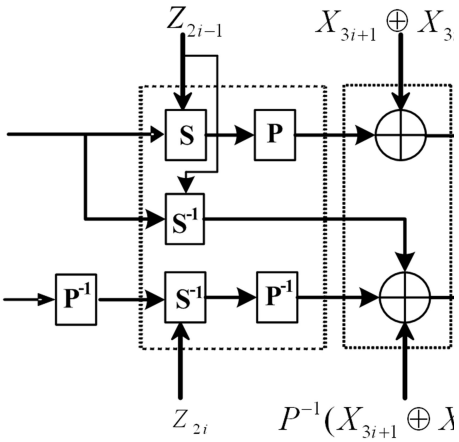

Figure 10: P^{-1} moved to the left for the odd rounds in FDE.

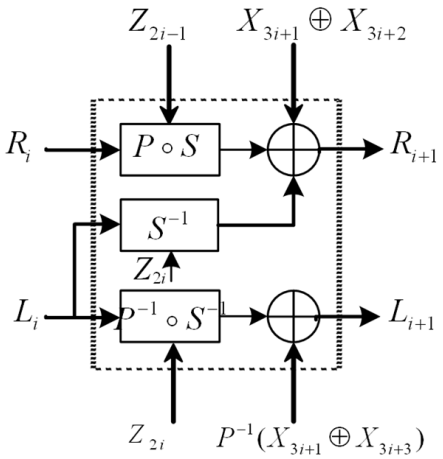

Figure 11. Odd rounds in FDE (F_o).

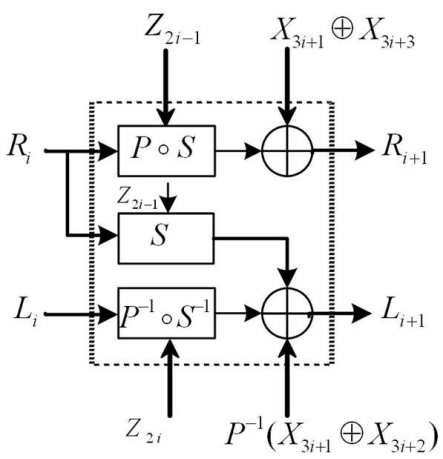

Figure 12. Even rounds in FDE (F_e).

From Figs. 11 and 12, we define the following functions

$$\begin{array}{l} F_{o_{Z_{2i},Z_{2i-1},Y_{2i+2},Y_{2i+1}}}(R,L) = \\ (P \circ S_{Z_{2i-1}}(R) \oplus S^{-1}_{Z_{2i}}(L) \oplus Y_{2i+1}, P^{-1} \circ S^{-1}_{Z_{2i}}(L) \oplus Y_{2i+2}) \\ F_{e_{Z_{2i},Z_{2i-1},Y_{2i+2},Y_{2i+1}}}(R,L) = \\ (P \circ S_{Z_{2i-1}}(R) \oplus Y_{2i+1}, P^{-1} \circ S^{-1}_{Z_{2i}}(L) \oplus S_{Z_{2i-1}}(R) \oplus Y_{2i+2}) \end{array} \tag{17}$$

with Y_i defined as

$$\begin{array}{l} Y_{2i+1} = \left\{ \begin{array}{ll} X_{3i+1} \oplus X_{3i+3} & i \text{ even} \\ X_{3i+1} \oplus X_{3i+2} & i \text{ odd} \end{array} \right. \\ Y_{2i+2} = \left\{ \begin{array}{ll} P^{-1}(X_{3i+1} \oplus X_{3i+2}) & i \text{ even} \\ P^{-1}(X_{3i+1} \oplus X_{3i+3}) & i \text{ odd} \end{array} \right. \end{array} \tag{18}$$

Eliminating P^{-1} in the first and last rounds results in two new structures, denoted G_0 and F_1, and these are shown in Figs. 13 and 14, respectively.

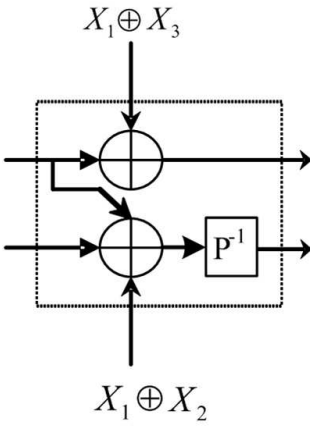

Figure 13: FDE before the first round (G_0)

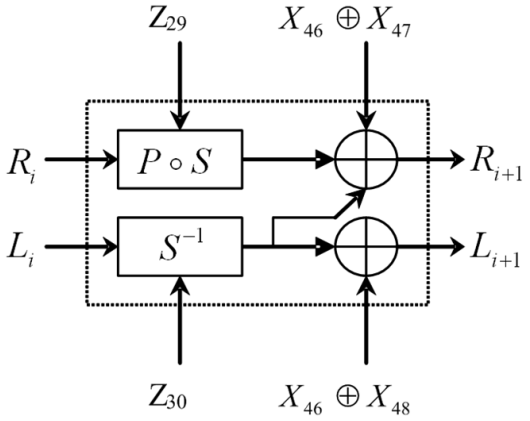

Figure 14: FDE in the 15th round (F_1)

These can be expressed by the following functions

$$\begin{aligned} G_0(R,L) &= (R \oplus Y_1, P^{-1}(R \oplus L \oplus Y_2)) \\ F_1(R,L) &= (SP_{Z_{29}}(R) \oplus S^{-1}_{Z_{2i}}(L) \oplus Y_{31}, S^{-1}_{Z_{30}}(L) \oplus Y_{32}) \end{aligned} \tag{19}$$

in which Y_1, Y_2, Y_{31} and Y_{32} are defined as

$$\begin{aligned} Y_1 &= X_1 \oplus X_3 \\ Y_2 &= X_1 \oplus X_2 \\ Y_{31} &= X_{46} \oplus X_{47} \\ Y_{32} &= X_{46} \oplus X_{48} \end{aligned} \tag{20}$$

Then the FDE encryption can be rewritten as

$$\begin{aligned} &\text{Encrypt}(P) = IP^{-1} \circ F_1 \circ F_{e_{Z_{28},Z_{27},Y_{30},Y_{29}}} \\ &\circ F_{o_{Z_{26},Z_{25},Y_{28},Y_{27}}} \circ \cdots \circ F_{o_{Z_2,Z_1,Y_4,Y_3}} \circ G_0 \circ IP(P) \end{aligned} \tag{21}$$

In (19), it is possible to merge all P^{-1} (except in G_0) into the S^{-1} tables in the same way as P is merged into the S tables. This merging creates two groups of tables in addition to S and S^{-1}. Thus there are four types of tables

1. PS_1 to PS_8 for the combination of P and S
2. $P^i\ S_1^i$ to $P^i\ S_8^i$ for the combination of P^{-1} and S^{-1}
3. S_1 to S_8 for S
4. S_1^{-i} to S_8^{-i} for S^{-1}

Each of S, S^{-1}, $P \circ S$ and $P^{-1} \circ S^{-1}$ can be implemented with 14 shifts, 31 logic and 8 memory instructions. For example, $P \circ S$ can be written in the C language as

```
long PoS(short z /*16bit key*/, /*left half block*/ long x) {
register int Res, u;
Res = PS1[((z<<4) & 0x30) | (u & 0xF)]; u>=4;
Res |= PS2[((z<<2 ) & 0x30) | (u & 0xF)]; u>=4;
Res |= PS3[((z ) & 0x30) | (u & 0xF)]; u>=4;
Res |= PS4[((z>>2 ) & 0x30) | (u & 0xF)]; u>=4;
Res |= PS5[((z>>4 ) & 0x30) | (u & 0xF)]; u>=4;
Res |= PS6[((z>>6 ) & 0x30) | (u & 0xF)]; u>=4;
Res |= PS7[((z>>8 ) & 0x30) | (u & 0xF)]; u>=4;
Res |= PS8[((z>>12) & 0x30) | (u & 0xF)];
Return Res
}
```

Therefore we have

Cost[PS] = 15 OR+14 SHIFT + 16 AND + 8 MEMREF = 45 LOGIC + 8 MEMREF

Thus the cost of F_o, F_e and F_l are

Cost[F_o] = Cost[F_e] = 3XOR + Cost[S] + Cost[$P \circ S$] + Cost[$P^{-1} \circ S^{-1}$] = 138 LOGIC + 24 MemRef

Cost[F_l] = 3XOR + Cost[PS] + Cost[S^{-1}] = 93 LOGIC + 16 MemRef

As a result, the cost of a block encryption is

Cost[FDE] = 2Cost[IP] + Cost[G_0] + 7(Cost[F_o] + Cost[F_e]) + Cost[F_l] = 2174 LOGIC + 360 MemRef

4.1.3 Using 12×32 s-boxes

Another means of reducing the S instructions is grouping eight s-boxes into four pairs of s-boxes [9,11]. Each pair of 6-input s-boxes is then merged into a single 12-input s-box. This merging affects the implementation of S, S^{-1}, $P \circ S$ and $P^{-1} \circ S^{-1}$, and reduces their instructions. Assume that S_{ij} is the result of merging S_i and S_j. S can then be implemented as

```
long S(short z, long x) {
register int Res, u;
Res = S12[ ((z << 8) & 0x00F) | (u & 0xFF) ]; u >= 8;
Res | = S34[ ((z << 4) & 0x00F) | (u & 0xFF) ]; u >= 8;
Res | = S56[ ((z ) & 0x00F) | (u & 0xFF) ]; u >= 8;
Res | = S78[ ((z >>4) & 0x00F) | (u & 0xFF) ];
return Res
```

}
Therefore we have
Cost[S] = 7 OR + 6 SHIFT + 8 AND + 4 MemRef = 21 LOGIC + 4 MemRef
Counting as done previously shows that
Cost[FDE] = 1454 LOGIC + 180 MemRef
This reduction was gained at a cost of increasing the s-box memory by a factor of 32.

4.1.4 Performance Results

Table 2 presents the performance using the techniques presented in this section under different computing environments. The results show an improvement in speed by a factor of 2 to 3. In a Windows environment, in spite of the reduction in instructions for the 12-input s-boxes, the rate of encryption is slower than with 6-input s-boxes. This is due to the lookup tables being too large to fit in L1 cache memory, thus a high percentage of the memory references in S, S^{-1}, $P \circ S$ and $P^{-1} \circ S^{-1}$ are cache misses. Assembly language implementations of DES with 12-input s-boxes provides similar results [22].

Table 3: Implementation Results without Bitslicing

	C-language FDE Implementation	Pentium I 133 Windows VC	Pentium III 550 MMX Linux Gcc	Pentium III 550 MMX Windows VC	Instruction Count
1	Original FDE with 6×4 s-boxes	1.1 Mbps	4.1 Mbps	5.05 Mbps	4370 LOGIC 240 MemRef
2	Fast FDE with 6×32 s-boxes	2.63 Mbps	8.5 Mbps	16.8 Mbps	2174 LOGIC 360 MemRef
3	Fast FDE with 12×32 s-boxes	1.96 Mbps	10 Mbps	12.5 Mbps	1454 LOGIC 180 MemRef

4.2 Fast bitslicing implementation

4.2.1 Bitsliced FDE

Bitsliced DES was introduced by Biham [21]. He showed that this implementation on a 64 bit Alpha processor is five times faster than the fastest known implementations. Bitslicing has also been effectively employed in the design

of the Serpent block cipher [10]. The main idea of this approach is to view an n-bit processor as n 1-bit processors. This implementation uses a nonstandard representation of the data and in each time step, n n-bit words can be encrypted. In this new representation, the i-th word includes all bits of the block word that are in position i. For example, if A below shows the representation of n blocks in standard form, then B represents the corresponding bitsliced form.

$$A = \begin{bmatrix} A_1 \\ A_2 \\ \vdots \\ A_n \end{bmatrix} = \begin{bmatrix} a_{1,n} & a_{1,n-1} & \cdots & a_{1,1} \\ a_{2,n} & a_{2,n-1} & \cdots & a_{2,1} \\ \vdots & \vdots & & \vdots \\ a_{n,n} & a_{n,n-1} & \cdots & a_{n,1} \end{bmatrix}$$

$$B = \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{bmatrix} = \begin{bmatrix} a_{n,1} & a_{n-1,1} & \cdots & a_{1,1} \\ a_{n,2} & a_{n-1,2} & \cdots & a_{1,2} \\ \vdots & \vdots & & \vdots \\ a_{n,n} & a_{n-1,n} & \cdots & a_{1,n} \end{bmatrix}$$

This conversion requires that each AND, OR and SHIFT instruction be implemented n^2 times. In our implementations with $n = 32$, this mapping is performed by 3008 logic instructions. The transformation for each block encryption must be done twice, before the bitsliced encryption and after encryption to return the block to standard format.

Mapping the data to this new space provides bit access. For example, exchanging B_1 and B_2 in the bitsliced blocks is the same as exchanging the first and second bits of A_1, A_2, ..., A_n in the A block. Thus the costly bit permutations in standard format become simple variable renaming with bitslicing. The major shortcoming of this approach is that it cannot be applied to operational modes which have a sequential ordering such as CBC or OFB encryption.

For FDE in bitsliced format both G and G' can be simply implemented with 96 XOR instructions. The other functions can easily be implemented by variable renaming in P, P^{-1}, IP and IP^{-1}. The renaming of P and P^{-1} can be incorporated into the functions located either before (S for P and G for P^{-1}) or after (G' for P and S^{-1} for P^{-1}). For IP and IP^{-1} we are restricted to renaming in G in the first round and G' in the last round, respectively. Implementing S and S^{-1} is not as straightforward as G or P. In the next section, we consider the implementation of S and S^{-1}.

4.2.2 S-box representation in bitsliced FDE

Bitsliced substitution implementation via table lookup is inefficient since six bits must be obtained from six different words. Moreover after lookup, the four-bit result must be distributed to four words. A better approach for s-box implementation is to employ boolean function representations. Since each

operation in the boolean representation of an s-box is performed by a logical instruction, the ideal representation is the one that has the lowest number of operations (or gates in hardware). The problem of finding the minimum number of gates to realize a boolean function is an open problem [21].

The conventional approach for boolean function implementation is divide and conquer. An n input function is designed by first selecting one of the functions and dividing it into two $n-1$ input functions and a multiplex operation. Conventional multiplexing (using AND, OR and NOT) requires four gates

$$f(a_1, a_2, \ldots, a_{n-1}, a_n) = f(a_1, a_2, \ldots, a_{n-1}, 1) \bullet a_n + f(a_1, a_2, \ldots, a_{n-1}, 0) \bullet \bar{a}_n$$

while using an XOR operation requires only two gates [10,13]

$$f(a_1, a_2, \ldots, a_{n-1}, a_n) = f_0 \oplus \bullet f_{XOR})$$

where

$$f_{XOR} = f_0 \oplus f_1$$

and

$$\begin{aligned} f_0 &= f(a_1, a_2, \ldots, a_{n-1}, 0) \\ f_1 &= f(a_1, a_2, \ldots, a_{n-1}, 1) \end{aligned}$$

Thus the number of gates to implement an n-input function by the latter method can be obtained from the following recursive equation

$$g_n = 2g_{n-1} + 2$$

If we end the recursion at $n = 1$ (not the best choice), g_6 yields 62, so that for a 6×4 s-box 248 gates are required. If the recursion ends at $n = 2$ (which requires precomputation of a_1 and a_2), g_6 equals 30. In this case the number of gates necessary for each 6×4 s-box is 132, since in addition to the 120 gates for the four columns of the s-box, 12 gates are needed to construct all boolean functions of a_1 and a_2 (trivial functions 0, 1, a_1 and a_2 do not require any gates). For the FDE s-boxes the numbers given above (248 and 142) are reduced to 160 and 120, respectively, by simplifying the multiplexing when f_0, f_1 or f_{XOR} are constants (0 or 1).

Another reduction technique is eliminating common subexpressions, which means avoiding the computation of repeated functions. This is done by saving a function computed during intermediate stages and reusing it in subsequent stages. In FDE, the construction of all functions of a_1 and a_2 falls into this category. With this technique, there is a time-memory tradeoff and in a software implementation, decreases the number of instructions while increasing the number of temporary variables. The efficiency of this approach depends on the destination machine (number of registers), function size, and number of times an expression is reused. This approach is explored in the next section.

4.2.3 Implementation results

Table 4 compares five different bitsliced implementations with regards to speed and number of instructions. The first, second and third implementations (I1, I2 and I3) use 6-input boolean functions for the s-box representations, while the others (I4 and I5) use 4-input boolean functions with the assumption of a constant key for all blocks. The difference between each group (6-input and 4-input) is the level of common subexpression elimination for the boolean function representations.

In I1 and I4, there are no common subexpression eliminations, while in I2 and l5 boolean functions in the first and second data variables ($x[4i]$ and $x[4i+1]$ for S_i) are reused in subsequent stages. The implementation of I3 employed the boolean functions in Kwan [23] for the DES s-boxes, which have an average gate count of 56. We next explain each implementation in detail. The instruction counts for each case consist of two terms, one for the conversions from standard format to bitsliced format and back, and the other for the actual encryption computations.

Conversion of a block of 32 32-bit words to bitsliced form has a cost of 3008 logical instructions. Since we always encrypt a block which contains 32 words of length 64-bits, and the bitsliced transformation must be done twice (to and from standard format), the cost of this conversion is 4×3008 instructions for a block and 376 instructions for each 64-bit word. Thus the cost of encryption in bitsliced form is

Cost[Bitsliced_Encrypt] = 8 Cost[G] + 8 Cost[G'] + 15 × 2 (Cost[P] + Cost[S]) + 2Cost[IP]

= $(16 \times 32 \times 3) + 30 \times 8 \times$S_avg_gates) / Block($32 \times 64$ bits)

= 48 + 7.5×S_avg_gates / 64 bits

where S_avg_gates is the average number of gates in the boolean representation of the s-box. Thus the total cost is

Cost[Encrypt] = Cost[Bitsliced_Encrypt] + Cost[bitslice_transform]

= 424 + 7.5×S_avg_gates / 64 bits

I1, I2: Fast implementation using 6-input boolean functions

In this case, the Z_i and X_i subkeys of a 32-word block are extended to vectors of 16 and 32 words, respectively. The S function is then implemented as

```
Block S(z :  Key_Block, x :  Input_Block) {
Block temp = 0;
// S1 = [F3, F2, F1, F0]
temp[0] = S1_F0 (z[0], x[0], x[1], x[2], x[3], z[1]);
temp[1] = S1_F1 (z[0], x[0], x[1], x[2], x[3], z[1]);
temp[2] = S1_F2 (z[0], x[0], x[1], x[2], x[3], z[1]);
```

Table 4: Comparison of Five Fast FDE Implementations using Bitslicing

		Pentium I 133	Pentium III 550 MMX	Instruction count	Average s-box gate count
I0	No bitslicing	1.1 Mbps	5.05 Mbps	4776 LOGIC + 240 MemRef	–
11	Bitsliced with 6-input boolean functions and no common subexpression elimination	1.57 Mbps	11.9 Mbps	1624 LOGIC	160
I2	Bitsliced with 6-input boolean functions and common subexpression elimination	1.76 Mbps	16.04 Mbps	1324 LOGIC	108 + 12 = 120
I3	Bitsliced with Kwan's s-box implementation	2.11 Mbps	11.1 Mbps	844 LOGIC	56
I4	Bitsliced with 4-input boolean functions and no common subexpression elimination	2.76 Mbps	21.36 Mbps	709 LOGIC + 30 IF	38
I5	Bitsliced with 4-input boolean functions and common subexpression elimination	2.63 Mbps	20.48 Mbps	679 LOGIC + 30 IF	22 + 12 = 34

```
temp[3] = S1_F3 (z[0], x[0], x[1], x[2], x[3] , z[1]);
// S2 = [F3, F2, F1, F0]
temp[4] = S2_F0 (z[2], x[4], x[5], x[6], x[7], z[3]);
temp[5] = S2_F1 (z[2], x[4], x[5], x[6], x[7], z[3]);
temp[6] = S2_F2 (z[2], x[4], x[5], x[6], x[7], z[3]);
temp[7] = S2_F3 (z[2], x[4], x[5], x[6], x[7] , z[3]);
⋮
// S8 = [F3, F2, F1, F0]
temp[28] = S8_F0 (z[14], x[28], x[29], x[30], x[31], z[15]);
temp[29] = S8_F1 (z[14], x[28], x[29], x[30], x[31], z[15]);
temp[30] = S8_F2 (z[14], x[28], x[29], x[30], x[31], z[15]);
temp[31] = S8_F3 (z[14], x[28], x[29], x[30], x[31] , z[15]);
return temp;
}
```

`Si_Fj` is the boolean representation of column j of S_i [3]. Results show that for both platforms, the encryption rate with I2 is higher than with I1, which means that the subexpression elimination in I2 has a positive effect on the encryption speed.

I3: Fast implementation using 6-input boolean functions with increased common subexpression elimination

I3 employs Kwan's method for DES s-box implementation [23]. In this representation, the number of gates varies between 42 and 63, as shown in Table 5. The same number of temporary variables are used in the computation of each s-box, thus there are no repeated expressions (even of size 2), which means no subexpression eliminations.

Table 5: Number of Gates using Kwan's Representation for the DES S-boxes

S-box	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8
Gates	63	56	57	42	62	57	57	54

The FDE s-boxes can be designed such that the average gates count is the same as with Kwan's method, but this is not considered here as we are only interested in the relative performance on the target Pentium processor. The I3 performance relative to I1 and I2 shows that on a Pentium-I processor the speed is higher than I1 and I2, while on a Pentium-III processor the speed is lower. The I3 implementation significantly reduces the number of instructions, while the number of temporary variables is increased. Since Pentium processors have only seven general-purpose registers, most of these variables are maintained in

memory and thus most of the register-register instructions in I2 and I3 are replaced with register-memory ones in I5.

As mentioned previously, there is a trade-off between instruction reduction and memory reference increases when using common subexpression elimination. Thus based on the ratio of machine cycles required for register-register instructions and register-memory instructions, this approach can produce different results depending on the processor. Because this ratio for Pentium III is more than for Pentium I (processor speed is increased relative to memory), we obtain performance improvements on the Pentium I, but not on the Pentium III.

Fast implementation of I4 and I5 using 4-input boolean functions

The implementation of I4 and I5 assumes similar subkeys for the 32 bit words, which leads to a value of 0 or 0xFFFFFFFF for the converted subkeys Z_i and X_i. We can represent each s-box output as a table that includes four 4-bit boolean functions, and based on the four different cases of Z subkeys, we can choose the proper representation. This method is a combination of table lookup and boolean functions. For example, the code for the implementation of S is shown below.

```
Block S(Block z /*16bit key*/ , Block x /*block of 32 word*/) {
Block temp;
// S1 = {{F3-11, F2-11, F1-11, F0-11}, {F3-10, F2-10, F1-10, F0-10},
{F3-01, F2-01, F1-01, F0-01}, {F3-00, F2-00, F1-00, F0-00}}
if (z[1] == 0 && z[0]==0) {
   temp[0] = S1_F0-00(x[0], x[1], x[2], x[3]);
   temp[1] = S1_F1-00(x[0], x[1], x[2], x[3]);
   temp[2] = S1_F2-00(x[0], x[1], x[2], x[3]);
   temp[3] = S1_F3-00(x[0], x[1], x[2], x[3]);
} else
if (z[1] == 0 && z[0]==0xFFFFFFFF) {
   temp[0] = S1_F0-01(x[0], x[1], x[2], x[3]);
   temp[1] = S1_F1-01(x[0], x[1], x[2], x[3]);
   temp[2] = S1_F2-01(x[0], x[1], x[2], x[3]);
   temp[3] = S1_F3-01(x[0], x[1], x[2], x[3]);
}else
if (z[1] == 0FFFFFFFF && z[0]==0) {
   temp[0] = S1_F0-10(x[0], x[1], x[2], x[3]);
   temp[1] = S1_F1-10(x[0], x[1], x[2], x[3]);
   temp[2] = S1_F2-10(x[0], x[1], x[2], x[3]);
   temp[3] = S1_F3-10(x[0], x[1], x[2], x[3]);
}else
```

```
if (z[1] == 0xFFFFFFFF && z[0]== 0xFFFFFFFF) {
   temp[0] = S1_F0_11(x[0], x[1], x[2], x[3]);
   temp[1] = S1_F1-11(x[0], x[1], x[2], x[3]);
   temp[2] = S1_F2-11(x[0], x[1], x[2], x[3]);
   temp[3] = S1_F3-11(x[0], x[1], x[2], x[3]);
}
⋮
// S8
if (z[15] == 0 && z[14]== 0) {
   temp[28] = S8_F0-00(x[28], x[29], x[30], x[31]);
   temp[29] = S8_F1-00(x[28], x[29], x[30], x[31]);
   temp[30] = S8_F2-00(x[28], x[29], x[30], x[31]);
   temp[31] = S8_F3-00(x[28], x[29], x[30], x[31]);
} else
if (z[15] == 0 && z[14]== 0xFFFFFFFF) {
   temp[28] = S8_F0-01(x[28], x[29], x[30], x[31]);
   temp[29] = S8_F1-01(x[28], x[29], x[30], x[31]);
   temp[30] = S8_F2-01(x[28], x[29], x[30], x[31]);
   temp[31] = S8_F3-01(x[28], x[29], x[30], x[31]);
}else
if (z[15] == 0xFFFFFFFF && z[14]== 0) {
   temp[28] = S8_F0-10(x[28], x[29], x[30], x[31]);
   temp[29] = S8_F1-10(x[28], x[29], x[30], x[31]);
   temp[30] = S8_F2-10(x[28], x[29], x[30], x[31]);
   temp[31] = S8_F3-10(x[28], x[29], x[30], x[31]);
}else
if (z[15] == 0xFFFFFFFF && z[14]== 0xFFFFFFFF) {
   temp[28] = S8_F0-11(x[28], x[29], x[30], x[31]);
   temp[29] = S8_F1-11(x[28], x[29], x[30], x[31]);
   temp[30] = S8_F2-11(x[28], x[29], x[30], x[31]);
   temp[31] = S8_F3-11(x[28], x[29], x[30], x[31]);
{
return temp;
}
```

In this example, `Si_Fj_00`, `Si_Fj_01`, `Si_Fj_10` and `Si_Fj_11` are boolean representations of column j in S_i when the corresponding bits of subkey Z_i are 00, 01, 10 and 11, respectively [6].

Our results show that these implementations are better than I1, I2 and I3. Also we found that the common subexpression elimination in I5 does not provide any improvement in speed for both platforms even though there are fewer instructions than the I4 implementation.

4.3 Comparison with other algorithms

In this section, we compare FDE with four block cipher algorithms in terms of encryption and decryption. The results are shown in Table 6. The FDE performance corresponds to the non-bitsliced implementation described in Section 4.2.1, and is about one-fifth of AES and is similar to Triple DES.

Table 6: Performance of FDE and Several Known Algorithms on an Intel Pentium IV Processor at 3.4 GHz

	AES	DES	Triple DES	Serpent	FDE
Encryption	85 MB/s	44 MB/s	17.5 MB/s	47 MB/s	16.5 MB/s
Decryption	80 MB/s	44 MB/s	17.5 MB/s	44 MB/s	16.5 MB/s

As shown in Figs. 11 and 12, the FDE computations in each round are about 3 times that of DES, so the results in Table 6 for FDE, DES and Triple DES are as expected.

5 Conclusions

In this paper, the concepts surrounding FDE were considered and compared with several AES candidates. We have shown that most of the concepts used in designing the AES candidates were originally employed in developing the FDE algorithm. Differential and linear attacks against FDE were considered, and it was shown that the FDE structure is provably immune. We found that a lower bound for a differential attack is 2^{-140}. In addition, for linear cryptanalysis we have shown that the probability is less than 2^{-55}, which requires at least 2^{110} plaintext-ciphertext pairs. This is greater than the total number of pairs.

To improve the performance of FDE in software, we considered two different strategies to eliminate the permutations in the round functions. First, the permutations were merged into the s-box tables. This required a particular pattern and a solution was obtained by transferring the P^{-1} permutations to the previous round. Second, we employed bitslice processing, in which case the data was transformed so that the permutations can be done simply by register renaming. Versions of both implementations were shown to have an encryption rate of 2 to 4 times higher than the original implementation.

References

[1] A. Shafieinejad, F. Hendessi, and T.A. Gulliver, "A structure for fast data encryption," *Int. J. of Contemporary Mathematical Science,* vol. 2, no. 29,

pp. 1401–1424, 2008.

[2] M. Aref, F. Hendessi, and M. Omoumi, "A new algorithm for fast data encryption," (in Persian), *Esteghlal - J. of Eng.*, vol. 11, no. 1, Fall 1992.

[3] F. Hendessi, M.R. Aref, T.A. Gulliver, and A.U.H. Sheikh, "Fast data encryption standard (FDES), A good replacement for DES," *Proc. Queen's Biennial Symp. on Commun.*, pp. 191–193, June 1996.

[4] F. Hendessi, *Analyzing of DES Cryptosystem*, M.Sc. Thesis, Isfahan University of Technology, Isfahan, Iran, Dec. 1988.

[5] A. Shafieinejad, F. Hendessi, and M. Esmaeili, "Designing strong S-boxes to complete the FDE block cipher structure," (in Persian) *Proc. Iranian Conf. Elec. Eng.*, pp. 413–420, May 2003.

[6] A. Shafieinejad, *Analyzing of FDE Cryptosystem and Fast Implementation in Software*, M.Sc. Thesis, Isfahan University of Technology, Isfahan, Iran, July 2002.

[7] A. Shafieinejad, F. Hendessi, and M. Esmaeili, "Fast FDE implementation in software using bitslicing," (in Persian), *Proc. Iranian Conference on Computer Engineering*, pp. 510–518, Mar. 2002.

[8] A. Shafieinejad and F. Hendessi, "Fast C-implementation of FDE in software," *Proc. Iranian Society of Cryptology Conference*, pp. 11–25, Oct. 2003.

[9] A. Shafeinejad and F. Hendessi, "FDE randomness evaluation by stochastic tests," *Proc. Iranian Society of Cryptology Conference*, pp. 190–204, Oct. 2003.

[10] R. Anderson, E. Biham, and L. Knudsen, "Serpent: A proposal for the Advanced Encryption Standard," *Proc. Advanced Encryption Standard Candidate Conf.*, Aug. 1998.

[11] J. Daemen and V. Rijmen, "AES proposal: Rijndael," *Proc. Advanced Encryption Standard Candidate Conf.*, Aug. 1998.

[12] C.H. Lim, "A revised version of Crypton: Crypton V1.0," *Proc. Int. Workshop on Fast Software Encryption*, Lecture Notes in Computer Science, vol. 1636, pp. 31–45, 1999.

[13] L. Chen, J.L. Massey, G.H. Khachatrian, and M.K. Kuregian, "SAFER+: Cylink Corporation's submission for the Advanced Encryption Standard," *Proc. Advanced Encryption Standard Candidate Conf.*, Aug. 1998.

[14] L. Brown, J. Pieprzyk, and J. Seberry, "LOKI97 - A submission for the Advanced Encryption Standard by the Australian Defence Force Academy," *Proc. Advanced Encryption Standard Candidate Conf.*, Aug. 1998.

[15] C. Adams, H. Heys, S. Tavares, and M. Wiener, "CAST-256: A submission for the Advanced Encryption Standard," *Proc. Advanced Encryption Standard Candidate Conf.*, Aug. 1998.

[16] M. Kanda, S. Moriai, K Aoki, H. Ueda, Y. Takashima, K. Ohta, and T. Matsumoto, "E2 - A new 128-bit block cipher," *IEICE Trans. on Fundamentals of Electronics, Commun. and Computer Sciences*, vol. E83-A, no. 1, pp. 48–59, Jan. 2000.

[17] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, and C. Hall, "Twofish: A 128-bit block cipher," *Proc. Advanced Encryption Standard Candidate Conf.*, Aug. 1998.

[18] L. McBride, "Q A proposal for NESSIE v2.00," *Proc. Open NESSIE Workshop*, Nov. 2000.

[19] M. Matsui, "Linear cryptanalysis method for DES cipher," *Advances in Cryptology - Proc. EUROCRYPT '93*, Lecture Notes in Computer Science, vol. 765, pp. 386–397, 1994.

[20] H.M. Heys, A Tutorial on Linear and Differential Cryptanalysis, Technical Report CORR 2001-17, Centre for Applied Cryptographic Research, Dept. of Combinatorics and Optimization, University of Waterloo, Mar. 2001, also Cryptologia, vol. XXVI, no. 3, pp. 189–221, 2002.

[21] E. Biham, "A fast new DES implementation in software," *Proc. Int. Workshop on Fast Software Encryption*, Lecture Notes in Computer Science, vol. 1267, pp. 26–272, 1997, also Technical Report CS0891, Computer Science Department, Technion, Israel.

[22] F. Correla, "A fast implementation of DES and Triple-DES on PA-RISC 2.0," *Proc. Conf. on Industrial Experiences with Systems Software*, p. 12, Oct. 2000, also www.usenix.org/events/osdi2000/wiess2000/full_papers /corella/corella.PDF.

[23] M. Kwan, Bitslice DES, http://www.darkside.com.au/bitslice, 1998.