The Block Cipher SC2000

Cryptographic Techniques Specifications

Takeshi Shimoyama†, Hitoshi Yanami†, Kazuhiro Yokoyama†,
Masahiko Takenaka†, Kouichi Itoh†, Jun Yajima†,
Naoya Torii†, Hidema Tanaka‡

† Fujitsu Laboratories LTD.
‡ Science University of Tokyo

2001.9.26

Contents

List of Figures

1 Introduction

This document describes the specifications for the block cipher SC2000. The algorithm has 128-bit data inputs and outputs and 128-bit, 192-bit, and 256-bit keys can be used.

2 Symbols and Notations

2.1 Notation

Basically, the following rules are used for notation:

- For input/output variables, a, b, c, d, e, f, g, and h and these characters subscripted with a number ($a1$ etc.) are used.
- For primary variables, $q, r, s, t, u, v, w, x, y$, and z and these characters subscripted with a number ($s0$ etc.) are used.
- The number of bits in a variable is expressed by each variable superscripted with a decimal number enclosed by parentheses as $a^{(4)}$. For 32-bit variables, the number of bits is omitted as a.
- A bit within a variable is expressed by a variable a subscripted with a decimal number as a_0. For example, second highest bit of variable $a^{(4)}$ is expressed as $a_1^{(4)}$.
- A list or an array of variables is expressed by parenthesized variables each of which is separated by a comma such as (a, b, c, d).
- For a list or array of variables such as (a, b), a is called the first entry, and b is called the second entry.
- A table is expressed by its name followed by []. For example, the extended key table is expressed as $ek[]$.
- Lowercase characters are used to express the name of a table of variables; uppercase characters are used to express the name of a table of constants. For example, the extended key table is expressed as $ek[]$, and a 4-bit S-Box table is expressed as $S_4[]$.
- A function is expressed as 'output variables = function name (input variables)'. Tables can be used as input/output variables.
- A number without a prefix expresses a decimal number, and a number prefixed with `0x` expresses a hexadecimal number like as `0x01234567`.

2.2 Operators and Operator Symbols

- XOR is the exclusive-or of two variables and expressed as $a \oplus b$, and in figures, expressed as $\oplus$.
- AND is the logical product of two variables and expressed as $a \wedge b$, and in figures, expressed as $\wedge$.
- OR is the logical sum of two variables and expressed as $a \vee b$, and in figures, expressed as $\wedge$.
- NOT is an operation that inverts all bits of a 32-bit variable and expressed as $\bar{b}$, and in figures, expressed as $\bar{b}$.
- ADD is an operation that applies modulo 2^{32} operation ($a + b \pmod{2^{32}}$) to the result of addition of two 32-bit variables and expressed as $a \boxplus b$, and in figures, expressed as $\boxplus$.

- SUB is an operation that applies modulo 2^{32} operation ($a - b \pmod{2^{32}}$) to the result of subtraction of two 32-bit variables and expressed as $a \boxminus b$, and in figures, expressed as $\boxminus$.

- MUL is an operation that applies modulo 2^{32} operation ($a \times b \pmod{2^{32}}$) to the result of multiplication of two 32-bit variables and expressed as $a \boxtimes b$, and in figures, expressed as $\boxtimes$.

- Rotate left one bit is an operation that rotates a 32-bit variable left by 1 bit ($(a_0, a_1, \cdots, a_{31}) \rightarrow (a_1, a_2, \cdots, a_0)$) and expressed as $a \lll_1$, and in figures expressed as $\lll_1$.

- A repeat operation is expressed by 'for' statement like as the C language, where i and n are used as loop variables.

- A conditional branch is expressed by 'if' statement like as the C language.

2.3 Endian

Big-Endian is used as Endian, where the highest bit is the 0-th bit. For example, $a^{(4)} = 10$ is expressed as $(a_0^{(4)}, a_1^{(4)}, a_2^{(4)}, a_3^{(4)}) = (1, 0, 1, 0)$.

3 Data Randomizing Specifications

3.1 Encryption Function

Explanation

The encryption function executes encryption. The encryption function consists of the following three functions: I function, B function, and R function, each of which has a (32 bits $\times$ 4) input/output. Among these three functions, the I function XORs the key and the B and R functions stir the data. The encryption function for a 128-bit key consists of 14 rounds of the I function, which XORs the key, and as data randomizing, 7 rounds of the B function and 12 rounds of the R function, totaling 19 rounds. The encryption function for a 192-bit or 256-bit key consists of 16 rounds of the I function, which XORs the key, and as data randomizing, 8 rounds of the B function and 14 rounds of the R function, totaling 22 rounds. Each function is executed in the order of I-B-I-$R \times R \cdots$ repeatedly. For two consecutive functions, the output (a, b, c, d) of the function in the preceding round is passed to the next round as the input (a, b, c, d). However, for two consecutive R functions, the output (a, b, c, d) of the function in the preceding round is passed to the next round as the input (c, d, a, b). For a 128-bit key, fifty-six 32-bit extended keys are used; for a 192-bit or 256-bit key, sixty-four 32-bit extended keys are used.

Syntax

$(e, f, g, h) = encrypt(a, b, c, d, ek[\ ], KeyLength)$

Input

a, b, c, d	:	32-bit data
$ek[\ ]$	:	32-bit extended key table
$KeyLength$	:	Key length(128/192/256)

output

e, f, g, h	:	32-bit data

Processing

$(s0,t0,u0,v0) = (a,b,c,d)$
$(s1,t1,u1,v1) = I\_func(s0,t0,u0,v0,ek[0],ek[1],ek[2],ek[3])$
$(s2,t2,u2,v2) = B\_func(s1,t1,u1,v1)$
$(s3,t3,u3,v3) = I\_func(s2,t2,u2,v2,ek[4],ek[5],ek[6],ek[7])$
$(s4,t4,u4,v4) = R\_func(s3,t3,u3,v3,\ \mathrm{0x55555555}\ )$
$(s5,t5,u5,v5) = R\_func(u4,v4,s4,t4,\ \mathrm{0x55555555}\ )$
$(s6,t6,u6,v6) = I\_func(s5,t5,u5,v5,ek[8],ek[9],ek[10],ek[11])$
$(s7,t7,u7,v7) = B\_func(s6,t6,u6,v6)$
$(s8,t8,u8,v8) = I\_func(s7,t7,u7,v7,ek[12],ek[13],ek[14],ek[15])$
$(s9,t9,u9,v9) = R\_func(s8,t8,u8,v8,\ \mathrm{0x33333333}\ )$
$(s10,t10,u10,v10) = R\_func(u9,v9,s9,t9,\ \mathrm{0x33333333}\ )$
$\vdots$
$(s29,t29,u29,v29) = R\_func(s28,t28,u28,v28,\ \mathrm{0x33333333}\ )$
$(s30,t30,u30,v30) = R\_func(u29,v29,s29,t29,\ \mathrm{0x33333333}\ )$
$(s31,t31,u31,v31) = I\_func(s30,t30,u30,v30,ek[48],ek[49],ek[50],ek[51])$
$(s32,t32,u32,v32) = B\_func(s31,t31,u31,v31)$
$(s33,t33,u33,v33) = I\_func(s32,t32,u32,v32,ek[52],ek[53],ek[54],ek[55])$
if $(KeyLength! = 128)$ {
  $(s34,t34,u34,v34) = R\_func(s33,t33,u33,v33,\mathrm{0x55555555}\ )$
  $(s35,t35,u35,v35) = R\_func(u34,v34,s34,t34,\ \mathrm{0x55555555}\ )$
  $(s36,t36,u36,v36) = I\_func(s35,t35,u35,v35,ek[56],ek[57],ek[58],ek[59])$
  $(s37,t37,u37,v37) = B\_func(s36,t36,u36,v36)$
  $(s38,t38,u38,v38) = I\_func(s37,t37,u37,v37,ek[60],ek[61],ek[62],ek[63])$
  $(e,f,g,h) = (s38,t38,u38,v38)$
} else {
  $(e,f,g,h) = (s33,t33,u33,v33)$
}

Configuration

The following lists the entire configuration of the encryption function. Symbols used in the configuration are as follows:

Symbol	Meaning
(in)	Input
(out)	Output
I	I function
B	B function
$R5$	R function with $mask$ = 0x55555555
$R3$	R function with $mask$ = 0x33333333
$-$	Straight connection $(a,b,c,d) \rightarrow (a,b,c,d)$
$\times$	Cross connection $(a,b,c,d) \rightarrow (c,d,a,b)$

Configuration for 128 bit key:
(in)-I-B-I-$R5 \times R5$-I-B-I-$R3 \times R3$-I-B-I-$R5 \times R5$-I-B-I-$R3 \times R3$-I-B-I-$R5 \times R5$-I-B-I-$R3 \times R3$-I-B-I-(out)
Configuration for 192/256 bit key:
(in)-I-B-I-$R5 \times R5$-I-B-I-$R3 \times R3$-I-B-I-$R5 \times R5$-I-B-I-$R3 \times R3$-I-B-I-$R5 \times R5$-I-B-I-$R3 \times R3$-I-B-I-$R5 \times R5$-I-B-I-(out)

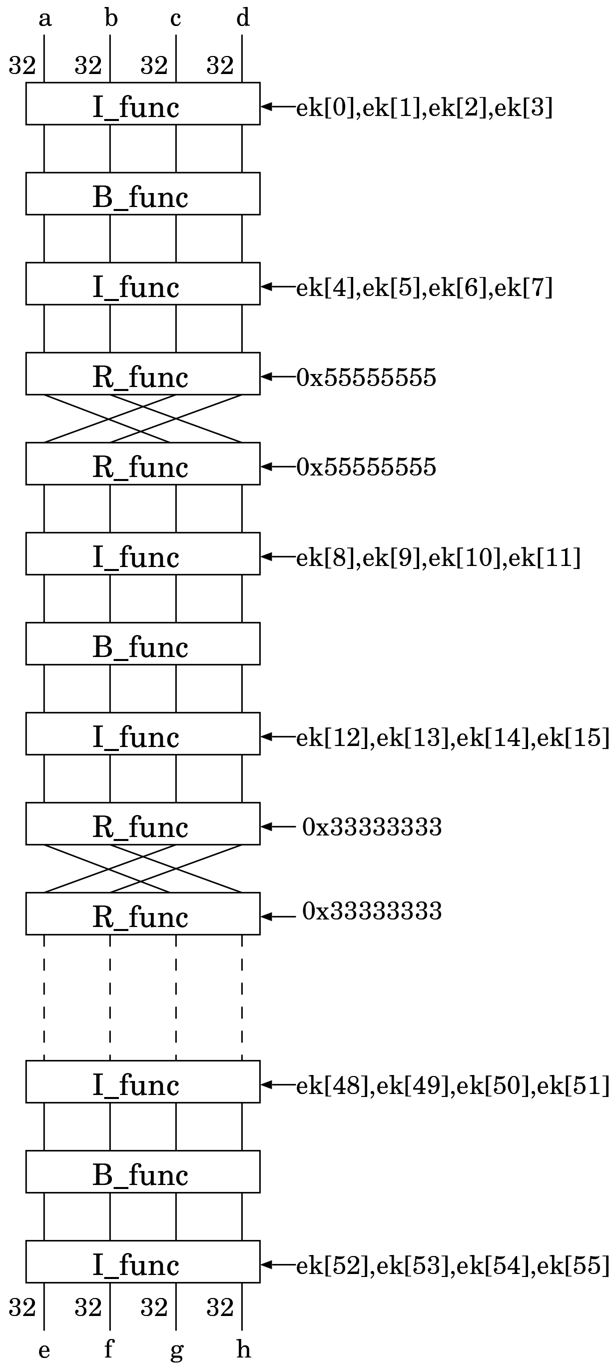


Figure 1: Encryption function (128-bit key)

3.2 Decryption Function

Explanation
The decryption function executes decryption. The decryption function consists of the following three functions: I function, B^{-1} function, and R function, each of which has a (32 bits $\times$ 4) input/output. Among these three functions, the I function XORs the key and the B^{-1} and R functions stir the data. The decryption function for a 128-bit key consists of 14 rounds of the I function, which XORs the key, and as data randomizing, 7 rounds of the B^{-1} function and 12 rounds of the R function, totaling 19 rounds. The decryption function for a 192-bit or 256-bit key consists of 16 rounds of the I function, which XORs the key, and as data randomizing, 8 rounds of the B^{-1} function and 14 rounds of the R function, totaling 22 rounds. Each function is executed in the order of I-B^{-1}-I-$R \times R \cdots$ repeatedly. For two consecutive functions, the output (a, b, c, d) of the function in the preceding round is passed to the next round as the input (a, b, c, d). However, for two consecutive R functions, the output (a, b, c, d) of the function in the preceding round is passed to the next round as the input (c, d, a, b). For a 128-bit key, fifty-six 32-bit extended keys are used; for a 192-bit or 256-bit key, sixty-four 32-bit extended keys are used.

Syntax

$$(e, f, g, h) = decrypt(a, b, c, d, ek[\,], KeyLength)$$

Input

a, b, c, d	:	32-bit data
$ek[\,]$	:	32-bit extended key table
$KeyLength$	:	Key length (128/192/256)

Output

e, f, g, h : 32-bit Data

Processing

if $(KeyLength != 128)$ {
$(s38, t38, u38, v38) = (a, b, c, d)$
$(s37, t37, u37, v37) = I\_func(s38, t38, u38, v38, ek[60], ek[61], ek[62], ek[63])$
$(s36, t36, u36, v36) = B^{-1}\_func(s37, t37, u37, v37)$
$(s35, t35, u35, v35) = I\_func(s36, t36, u36, v36, ek[56], ek[57], ek[58], ek[59])$
$(s34, t34, u34, v34) = R\_func(s35, t35, u35, v35, \text{0x55555555})$
$(s33, t33, u33, v33) = R\_func(u34, v34, s34, t34, \text{0x55555555})$
} else {
$(s33, t33, u33, v33) = (a, b, c, d)$
}
$(s32, t32, u32, v32) = I\_func(s33, t33, u33, v33, ek[52], ek[53], ek[54], ek[55])$
$(s31, t31, u31, v31) = B^{-1}\_func(s32, t32, u32, v32)$
$(s30, t30, u30, v30) = I\_func(s31, t31, u31, v31, ek[48], ek[49], ek[50], ek[51])$
$(s29, t29, u29, v29) = R\_func(s30, t30, u30, v30, \text{0x33333333})$
$(s28, t28, u28, v28) = R\_func(u29, v29, s29, t29, \text{0x33333333})$
$\vdots$
$(s9, t9, u9, v9) = R\_func(s10, t10, u10, v10, \text{0x33333333})$
$(s8, t8, u8, v8) = R\_func(u9, v9, s9, t9, \text{0x33333333})$
$(s7, t7, u7, v7) = I\_func(s8, t8, u8, v8, ek[12], ek[13], ek[14], ek[15])$
$(s6, t6, u6, v6) = B^{-1}\_func(s7, t7, u7, v7)$
$(s5, t5, u5, v5) = I\_func(s6, t6, u6, v6, ek[8], ek[9], ek[10], ek[11])$
$(s4, t4, u4, v4) = R\_func(s5, t5, u5, v5, \text{0x55555555})$
$(s3, t3, u3, v3) = R\_func(u4, v4, s4, t4, \text{0x55555555})$
$(s2, t2, u2, v2) = I\_func(s3, t3, u3, v3, ek[4], ek[5], ek[6], ek[7])$
$(s1, t1, u1, v1) = B^{-1}\_func(s2, t2, u2, v2)$
$(s0, t0, u0, v0) = I\_func(s1, t1, u1, v1, ek[0], ek[1], ek[2], ek[3])$
$(e, f, g, h) = (s0, t0, u0, v0)$

Configuration

The following lists the entire configuration of the decryption function. Symbols used in the configuration are as follows:

Symbol	Meaning
(in)	Input
(out)	Output
I	I function
B^{-1}	B^{-1} function
$R5$	R function with $mask$ = `0x55555555`
$R3$	R function with $mask$ = `0x33333333`
$-$	Straight connection $(a,b,c,d) \to (a,b,c,d)$
$\times$	Cross connection $(a,b,c,d) \to (c,d,a,b)$

Configuration for 128-bit key:
(in)-I-B^{-1}-I-$R3 \times R3$-I-B^{-1}-I-$R5 \times R5$-I-B^{-1}-I-$R3 \times R3$-I-B^{-1}-I-$R5 \times R5$-I-B^{-1}-I-$R3 \times R3$-I-B^{-1}-I-$R5 \times R5$-I-B^{-1}-I-(out)

Configuration for 192-bit or 256-bit key:
(in)-I-B^{-1}-I-$R5 \times R5$-I-B^{-1}-I-$R3 \times R3$-I-B^{-1}-I-$R5 \times R5$-I-B^{-1}-I-$R3 \times R3$-I-B^{-1}-I-$R5 \times R5$-I-B^{-1}-I-$R3 \times R3$-I-B^{-1}-I-$R5 \times R5$-I-B^{-1}-I-(out)

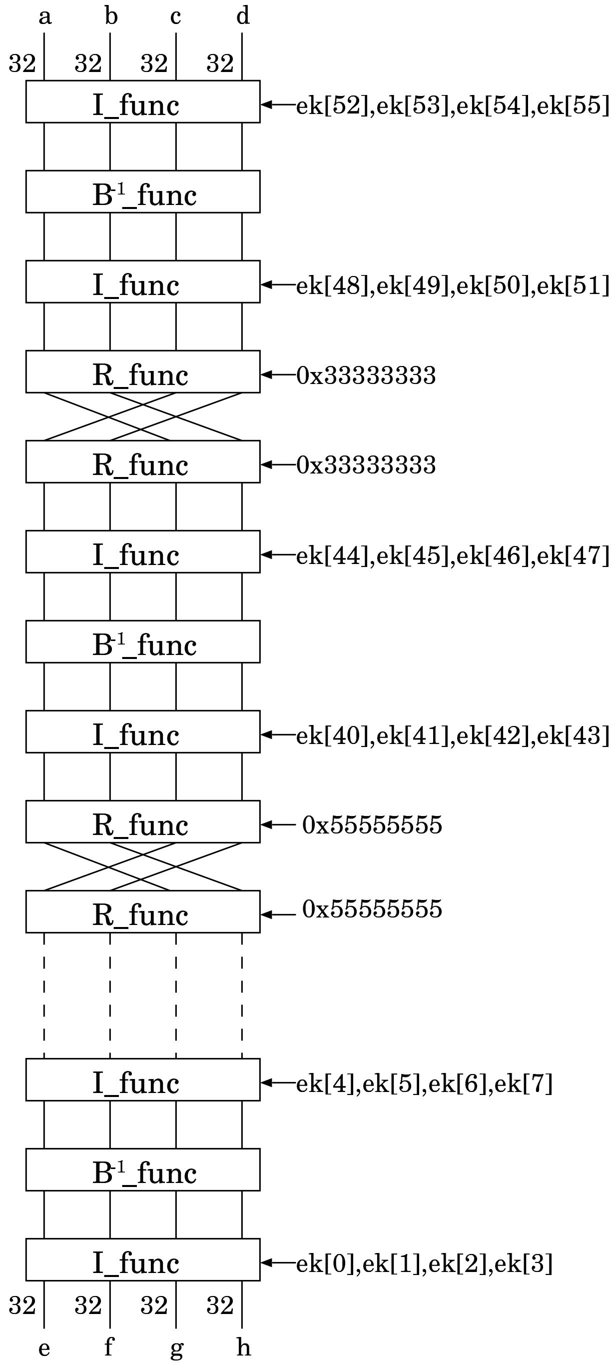


Figure 2: Decryption function (128-bit key)

3.3 *I* function

Explanation
The I function is given four 32-bit variables and four 32-bit extended keys as inputs, and it outputs four 32-bit variables. Each input data is XORed with the extended key. The extended key is input only to an I function.

Syntax
$(e, f, g, h) = I\_func(a, b, c, d, ka, kb, kc, kd)$

Input
a, b, c, d : 32b-it Data
ka, kb, kc, kd : 32-bit Extended key data

Output
e, f, g, h : 32-bit data

Processing
$e = a \oplus ka$
$f = b \oplus kb$
$g = c \oplus kc$
$h = d \oplus kd$

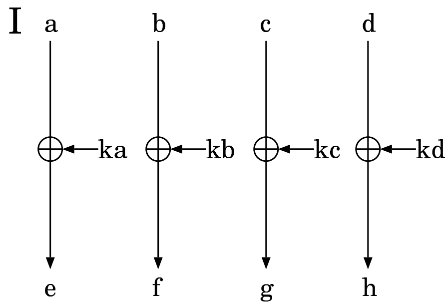


Figure 3: I function

3.4 R function

Explanation

The R function is a Feistel-type data randomizing function having four 32-bit variables as the input and the output. The R function inputs the third entry and the fourth entry (c, d) of the input data and the constant data mask to the F function, and it XORs the two outputs from the F function with the first entry and the second entry (a, b) of the input data.

Syntax

$(e, f, g, h) = R\_func(a, b, c, d, mask)$

Input

a, b, c, d : 32-bit data
$mask$: 32-bit constant

Output

e, f, g, h : 32-bit data

Processing

$(s, t) = F\_func(c, d, mask)$
$e = a \oplus s$
$f = b \oplus t$
$g = c$
$h = d$

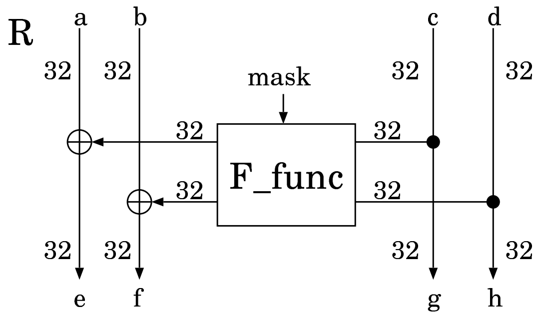


Figure 4: R function

3.5 F function

Explanation
The F function is given two 32-bit variables and constant as inputs, and outputs two 32-bit variables. The F function processes two variables with the S function and the M function respectively and then processes the two outputs with the L function.

Syntax

$(c, d) = F\_func(a, b, mask)$

Input

a, b	:	32-bit data
$mask$	:	32-bit constant

Output

c, d	:	32-bit data

Processing

$s = S\_func(a)$
$s' = M\_func(s)$
$t = S\_func(b)$
$t' = M\_func(t)$
$(c, d) = L\_func(s', t', mask)$

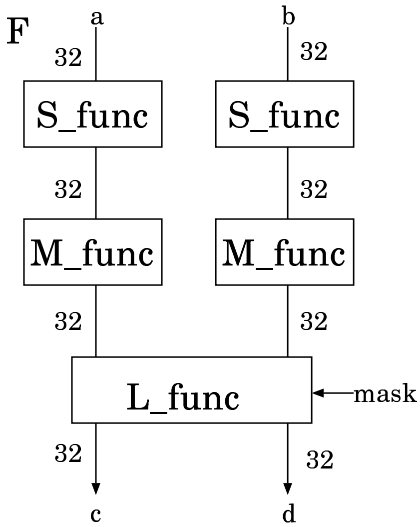


Figure 5: F function

3.6 S function

Explanation
The S function is a 32-bit input/output nonlinear function. The S function splits the 32-bit input into 6 bits, 5 bits, 5 bits, 5 bits, 5 bits and 6 bits. The S function then looks up the 6-bit S-Box table S_6 with the original 6 bits if there are 6 bits; it looks up the 5-bit S-Box table S_5 with the 5 bits if there are 5 bits. The function then aligns individual outputs in the same order to generate 32 bits.

Syntax

$b = S\_func(a)$

Input

a : 32-bit data

Output

b : 32-bit data

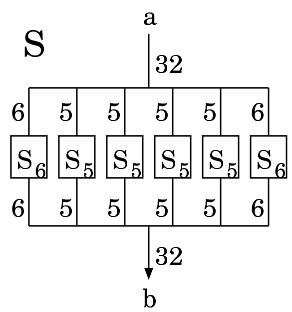


Figure 6: S function

Processing

$q^{(6)} = (a_0, \ldots, a_5)$	/*Extract 6 bits 0 to 5 of a*/
$r^{(5)} = (a_6, \ldots, a_{10})$	/*Extract 5 bits 6 to 10 of a*/
$s^{(5)} = (a_{11}, \ldots, a_{15})$	/*Extract 5 bits 11 to 15 of a*/
$t^{(5)} = (a_{16}, \ldots, a_{20})$	/*Extract 5 bits 16 to 20 of a*/
$u^{(5)} = (a_{21}, \ldots, a_{25})$	/*Extract 5 bits 21 to 25 of a*/
$v^{(6)} = (a_{26}, \ldots, a_{31})$	/*Extract 6 bits 26 to 31 of a*/
$q'^{(6)} = S_6[q^{(6)}]$	/*Look up the 6-bit S-Box S_6*/
$r'^{(5)} = S_5[r^{(5)}]$	/*Look up the 5-bit S-Box S_5*/
$s'^{(5)} = S_5[s^{(5)}]$	/*Look up the 5-bit S-Box S_5*/
$t'^{(5)} = S_5[t^{(5)}]$	/*Look up the 5-bit S-Box S_5*/
$u'^{(5)} = S_5[u^{(5)}]$	/*Look up the 5-bit S-Box S_5*/
$v'^{(6)} = S_6[v^{(6)}]$	/*Look up the 6-bit S-Box S_6*/
$(b_0, \ldots, b_5) = q'^{(6)}$	/*Store 6 bits in bits 0 to 5 of b*/
$(b_6, \ldots, b_{10}) = r'^{(5)}$	/*Store 5 bits in bits 6 to 10 of b*/
$(b_{11}, \ldots, b_{15}) = s'^{(5)}$	/*Store 5 bits in bits 11 to 15 of b*/
$(b_{16}, \ldots, b_{20}) = t'^{(5)}$	/*Store 5 bits in bits 16 to 20 of b*/
$(b_{21}, \ldots, b_{25}) = u'^{(5)}$	/*Store 5 bits in bits 21 to 25 of b*/
$(b_{26}, \ldots, b_{31}) = v'^{(6)}$	/*Store 6 bits in bits 26 to 31 of b*/

3.7 M function

Explanation
The M function is a 32-bit input/output linear function. The M function considers the input as a vector of 32 entries of 1 bit and multiplies it with the `M-table` as (32,32)-matrix of 1 bit.

Syntax

$b = M\_func(a)$

Input

a : 32-bit data

Output

b : 32-bit data

Processing

$b = 0;$
for $(i = 0; i < 32; i++)$ {
    /* If the i-th bit of a is 1, XOR b with the i-th row of the `M-table` */
    if $(a_i == 1)$ $b = b \oplus$ `M[i]`
}

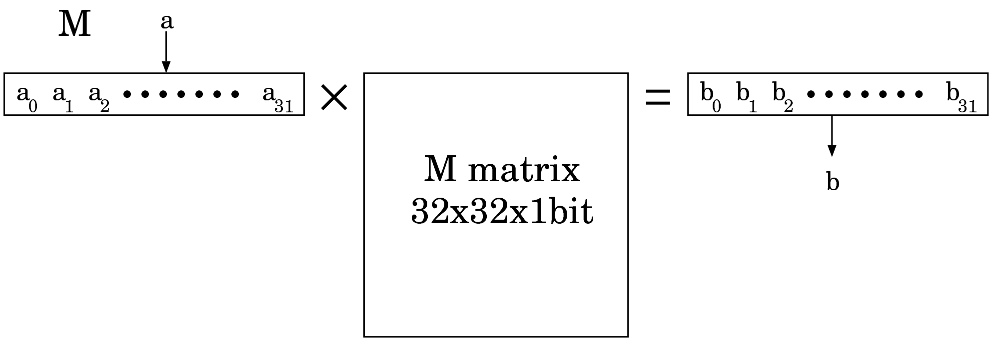


Figure 7: M function

3.8 L function

Explanation

The L function is given two 32-bit variables and a constant as inputs and outputs two 32-bit variables. The L function outputs two data by processing 2 input values (a, b): (1) (a AND constant) XOR b (2) (b AND (NOT of constant)) XOR a

Syntax

$(c, d) = L\_func(a, b, mask)$

Input

a, b	:	32-bit data
$mask$	:	32-bit constant

Output

c, d	:	32-bit data

Processing

$imask = \overline{mask}$

$s = a \wedge mask$

$t = b \wedge imask$

$c = s \oplus b$

$d = t \oplus a$

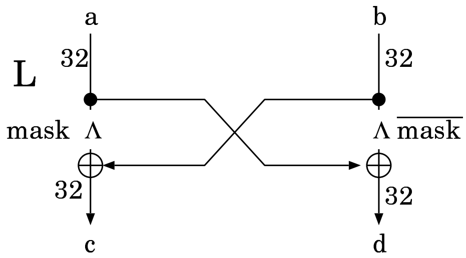


Figure 8: L function

3.9 B/B^{-1} function

Explanation

This function converts four 32-bit inputs into thirty-two 4-bit values with the T function, each of which is processed with the 4-bit S-Box table S_4. This function then recovers the output values into four 32-bit values and outputs those values with the T^{-1} funciton. The B^{-1} function uses S_4i, which is the inverse of the 4-bit S-Box table S_4 .

Syntax

$(e, f, g, h) = B\_func(a, b, c, d)$

$(e, f, g, h) = B^{-1}\_func(a, b, c, d)$

Input

a, b, c, d : 32-bit data

Output

e, f, g, h : 32-bit data

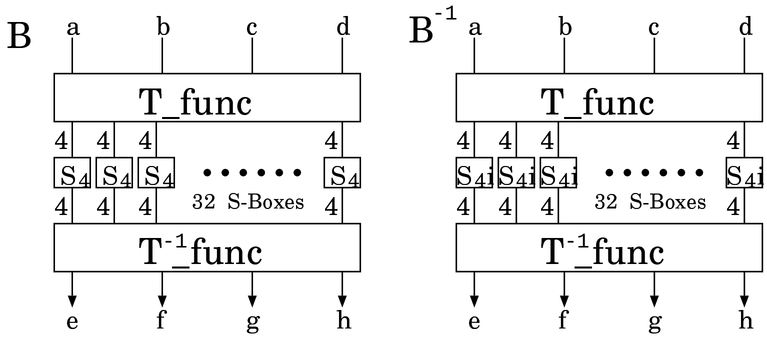


Figure 9: B/B^{-1} function

Processing

B_func:

$(s0^{(4)}, s1^{(4)}, ..., s31^{(4)}) = T\_func(a, b, c, d)$

$t0^{(4)} = S_4[s0^{(4)}]$

$t1^{(4)} = S_4[s1^{(4)}]$

$\vdots$

$t31^{(4)} = S_4[s31^{(4)}]$

$(e, f, g, h) = T^{-1}\_func(t0^{(4)}, t1^{(4)}, ..., t31^{(4)})$

B^{-1}_func:

$(s0^{(4)}, s1^{(4)}, ..., s31^{(4)}) = T\_func(a, b, c, d)$

$t0^{(4)} = S_4i[s0^{(4)}]$

$t1^{(4)} = S_4i[s1^{(4)}]$

$\vdots$

$t31^{(4)} = S_4i[s31^{(4)}]$

$(e, f, g, h) = T^{-1}\_func(t0^{(4)}, t1^{(4)}, ..., t31^{(4)})$

In the program implementation, the T and T^{-1} functions can be written as a loop by merging them as follows:

B_func:
for $(i = 0; i < 32; i++)$ {
  /*begin T-func*/
  $s_0^{(4)} = a_i$
  $s_1^{(4)} = b_i$
  $s_2^{(4)} = c_i$
  $s_3^{(4)} = d_i$
  /*end T-func*/
  $t^{(4)} = S_4[s^{(4)}]$
  /*begin T^{-1}-func*/
  $e_i = t_0^{(4)}$
  $f_i = t_1^{(4)}$
  $g_i = t_2^{(4)}$
  $h_i = t_3^{(4)}$
  /*end T^{-1}-func*/
}

B^{-1}_func:
for $(i = 0; i < 32; i++)$ {
  /*T-func*/
  $s_0^{(4)} = a_i$
  $s_1^{(4)} = b_i$
  $s_2^{(4)} = c_i$
  $s_3^{(4)} = d_i$
  $t^{(4)} = S_4i[s^{(4)}]$
  /*T^{-1}-func*/
  $e_i = t_0^{(4)}$
  $f_i = t_1^{(4)}$
  $g_i = t_2^{(4)}$
  $h_i = t_3^{(4)}$
}

3.10 T/T^{-1} function

Explanation
The T function considers four 32-bit values as a (32,4)-matrix of 1 bit. The T function converts the matrix into the transpose, a (4,32)-matrix and then converts it into thirty-two 4-bit data values. The T^{-1} function is the inverse function of T. The T^{-1} function considers thirty-two 4-bit data values as a (4,32)-matrix and then converts the matrix into the transpose, a (32,4)-matrix and then converts it into four 32-bit data values.

Syntax

$(s0^{(4)}, s1^{(4)}, \ldots s31^{(4)}) = T\_func(a, b, c, d)$
$(e, f, g, h) = T^{-1}\_func(t0^{(4)}, t1^{(4)}, ...t31^{(4)})$

Input

a, b, c, d	:	32-bit data
$t0^{(4)}, t1^{(4)}, \ldots t31^{(4)}$	:	4bit data

Output

$s0^{(4)}, s1^{(4)}, \ldots s31^{(4)}$	:	4bit data
e, f, g, h	:	32-bit data

Processing

T_func:

$$s0^{(4)} = (s0_0^{(4)}, s0_1^{(4)}, s0_2^{(4)}, s0_3^{(4)}) = (a_0, b_0, c_0, d_0)$$
$$s1^{(4)} = (s1_0^{(4)}, s1_1^{(4)}, s1_2^{(4)}, s1_3^{(4)}) = (a_1, b_1, c_1, d_1)$$
$$\vdots$$
$$s31^{(4)} = (s31_0^{(4)}, s31_1^{(4)}, s31_2^{(4)}, s31_3^{(4)}) = (a_{31}, b_{31}, c_{31}, d_{31})$$

T^{-1}_func:

$$e = (e_0, e_1, ..., e_{31}) = (t0_0^{(4)}, t1_0^{(4)}, ..., t31_0^{(4)})$$
$$f = (f_0, f_1, ..., f_{31}) = (t0_1^{(4)}, t1_1^{(4)}, ..., t31_1^{(4)})$$
$$g = (g_0, g_1, ..., g_{31}) = (t0_2^{(4)}, t1_2^{(4)}, ..., t31_2^{(4)})$$
$$h = (h_0, h_1, ..., h_{31}) = (t0_3^{(4)}, t1_3^{(4)}, ..., t31_3^{(4)})$$

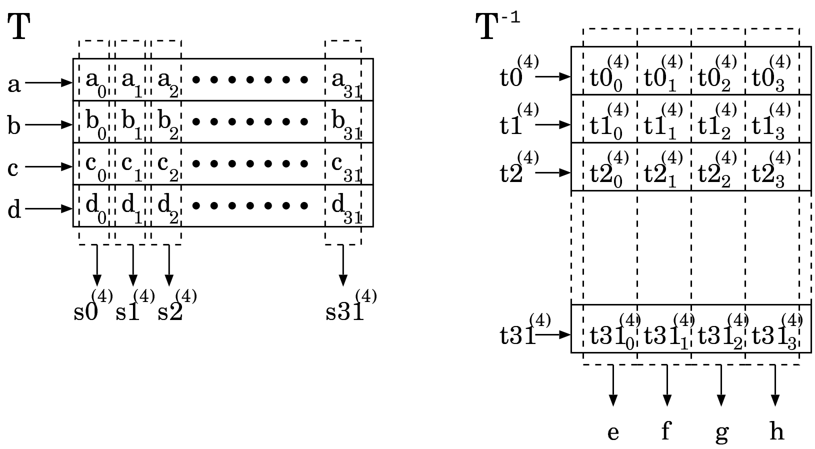


Figure 10: T/T^{-1} function

4 Key Schedule Specifications

4.1 Entire Configuration of Key Schedule

Explanation

The key schedule generates fifty-six 32-bit extended keys (for 128-bit key) or sixty-four 32-bit extended keys (for 192-bit or 256-bit key) from the secret key. The key schedule is a function consisting of the intermediate key generation function and the extended key generation function. For a 256-bit key, the key schedule splits the 256-bit secret key into eight 32-bit user keys $uk[0], \ldots, uk[7]$. The highest 32-bit of secret key is set to $uk[0]$ and lowest its 32-bit is set to $uk[7]$. For a 192-bit key, the key schedule splits the 192-bit secret key into six 32-bit user keys $uk[0], \ldots, uk[5]$ like as 256-bit key, then the function expands the 6 user keys to 8 user keys by $uk[6] = uk[0]$ and $uk[7] = uk[1]$. In the same way, for a 128-bit key, the key schedule splits the 128-bit secret key into four 32-bit user keys $uk[0], \ldots, uk[3]$, then expands the 4 user keys to 8 user keys by $uk[4] = uk[0]$, $uk[5] = uk[1]$, $uk[6] = uk[2]$ and $uk[7] = uk[3]$. From the user keys $uk[]$, it generates the intermediate keys $imkey[]$ with the intermediate key generation function and then executes the extended key generation function to generate 32-bit extended keys $ek[]$. As a result, fifty-six extended keys are generated for a 128-bit key, and sixty-four extended keys are generated for a 192-bit or 256-bit key length.

Syntax

$(ek[\,]) = make\_keys(uk[\,], KeyLength)$

Input

$uk[\ ]$	:	32-bit user key table (4 for128-bit key, 6 for 192-bit key, 8 for 256-bit key
$KeyLength$	:	The length of the secret key (128/192/256)

Output

$ek[\ ]$: 32-bit extended key (56 for 128-bit key, 64 for 192-bit or 256-bit key)

Processing

```
if (KeyLength == 128) {
    uk[4] = uk[0]
    uk[5] = uk[1]
    uk[6] = uk[2]
    uk[7] = uk[3]
} else if (KeyLength == 192) {
    uk[6] = uk[0]
    uk[7] = uk[1]
}
(a[ ],b[ ],c[ ],d[ ]) = make_imkeys(uk[ ])
(ek[ ]) = make_ekeys(a[ ],b[ ],c[ ],d[ ],KeyLength)
```

4.2 Intermediate Key Generation Function

Explanation

This function generates twelve 32-bit intermediate keys from eight 32-bit user keys.

Syntax

$(a[\ ], b[\ ], c[\ ], d[\ ]) = make\_imkeys(uk[\ ])$

Input

$uk[\ ]$: eight 32-bit user key

Output

$a[\ ], b[\ ], c[\ ], d[\ ]$: Intermediate key tables (three for each table)

Processing

for $(i = 0; i < 3; i++)$ {

$$a[i] = M\_func(S\_func((M\_func(S\_func(4i)) \boxplus M\_func(S\_func(uk[0]))) \oplus (M\_func(S\_func(uk[1])) \boxtimes (i+1))))$$

$$b[i] = M\_func(S\_func((M\_func(S\_func(4i+1)) \boxplus M\_func(S\_func(uk[2]))) \oplus (M\_func(S\_func(uk[3])) \boxtimes (i+1))))$$

$$c[i] = M\_func(S\_func((M\_func(S\_func(4i+2)) \boxplus M\_func(S\_func(uk[4]))) \oplus (M\_func(S\_func(uk[5])) \boxtimes (i+1))))$$

$$d[i] = M\_func(S\_func((M\_func(S\_func(4i+3)) \boxplus M\_func(S\_func(uk[6]))) \oplus (M\_func(S\_func(uk[7])) \boxtimes (i+1))))$$

}

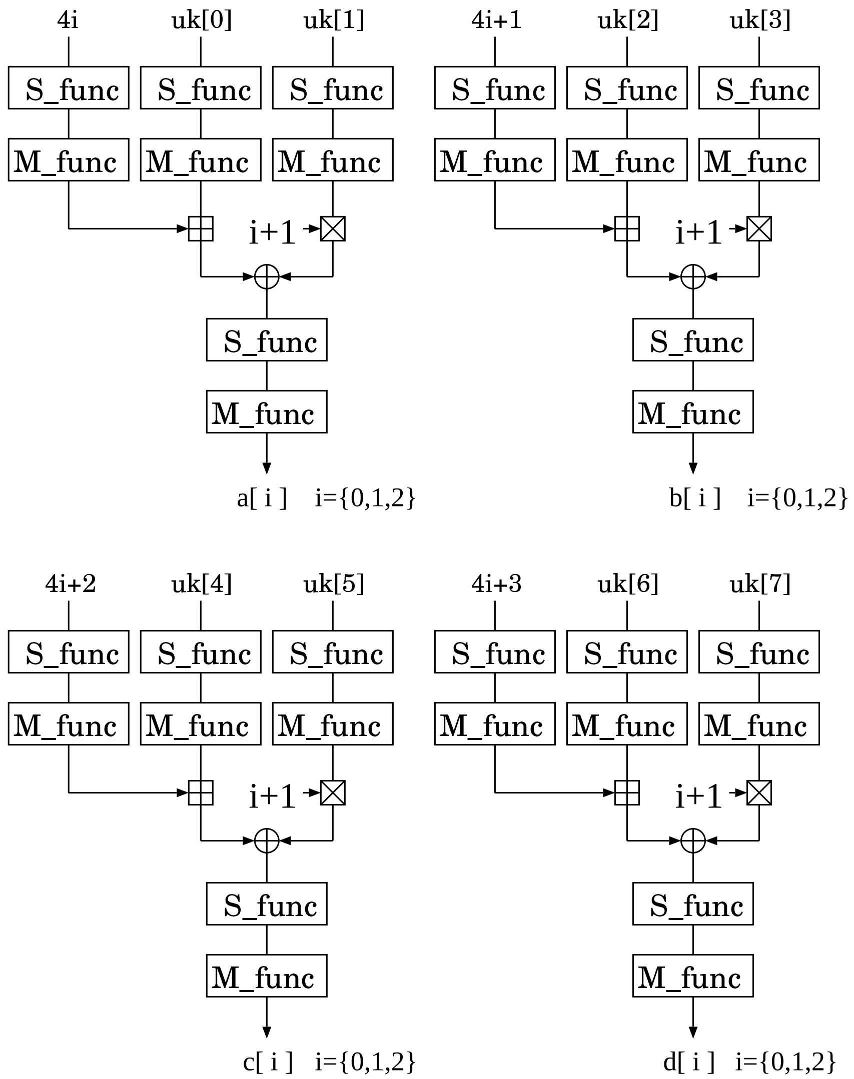


Figure 11: Intermediate key generation function

4.3 Extended Key Generation Function

Explanation

This function generates fifty-six 32-bit extended keys from twelve 32-bit intermediate keys for a 128-bit key; and also generates sixty-four 32-bit extended keys for a 192-bit or 256-bit key. The symbol $\lfloor a \rfloor$ means the largest integer not grater than a.

Syntax

$(ek[\ ]) = make\_ekeys(a[\ ], b[\ ], c[\ ], d[\ ], KeyLength)$

Input

$a[\ ], b[\ ], c[\ ], d[\ ]$: Intermediate key tables (3 for each table)

$KeyLength$: The length of the secret key(128/192/256)

Output

$ek[\ ]$: 32-bit extended keys (56 for 128-bit key, 64 for 192-bit or 256-bit key)

Processing

if $(KeyLength == 128)$ $num\_ekey = 56$

else $num\_ekey = 64$

for $(n = 0; n < num\_ekey; n++)$ {

$s = n \pmod 9$

$t = (n + \lfloor n/36 \rfloor) \pmod{12}$

$X = Order[t][0]$

$x = Index[s][0]$

$Y = Order[t][1]$

$y = Index[s][1]$

$Z = Order[t][2]$

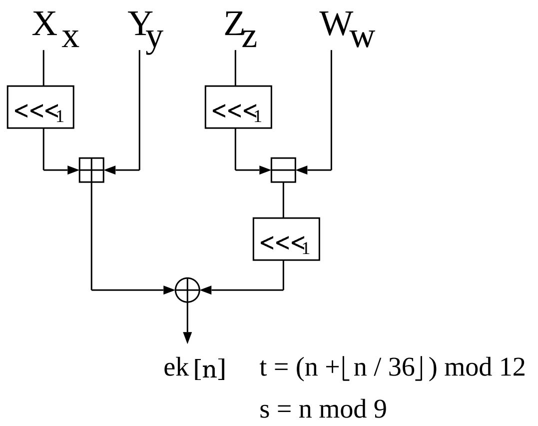
$z = Index[s][2]$

$W = Order[t][3]$

$w = Index[s][3]$

$ek[n] = ((X[x] \lll_1) \boxplus Y[y]) \oplus (((Z[z] \lll_1) \boxminus W[w]) \lll_1)$

}



Order (abcd)

t	X	Y	Z	W
0	a	b	c	d
1	b	a	d	c
2	c	d	a	b
3	d	c	b	a
4	a	c	d	b
5	b	d	c	a
6	c	a	b	d
7	d	b	a	c
8	a	d	b	c
9	b	c	a	d
10	c	b	d	a
11	d	a	c	b

Index (i)

s	x	y	z	w
0	0	0	0	0
1	1	1	1	1
2	2	2	2	2
3	0	1	0	1
4	1	2	1	2
5	2	0	2	0
6	0	2	0	2
7	1	0	1	0
8	2	1	2	1

Figure 12: Extended key generation function

5 Table

The following shows the tables used by functions for the cipher:

5.1 Data Randomizing

```
S6[64]   =  {47,59,25,42,15,23,28,39,26,38,36,19,60,24,29,56,
             37,63,20,61,55, 2,30,44, 9,10, 6,22,53,48,51,11,
             62,52,35,18,14,46, 0,54,17,40,27, 4,31, 8, 5,12,
              3,16,41,34,33, 7,45,49,50,58, 1,21,43,57,32,13};
S5[32]   =  {20,26, 7,31,19,12,10,15,22,30,13,14, 4,24, 9,18,
             27,11, 1,21, 6,16, 2,28,23, 5, 8, 3, 0,17,29,25};
S4[16]   =  { 2, 5,10,12, 7,15, 1,11,13, 6, 0, 9, 4, 8, 3,14};
S4i[16]  =  {10, 6, 0,14,12, 1, 9, 4,13,11, 2, 7, 3, 8,15, 5};

/* M-Table */
 M[32]  =  {0xd0c19225,0xa5a2240a,0x1b84d250,0xb728a4a1,
            0x6a704902,0x85dddbe6,0x766ff4a4,0xecdfe128,
            0xafd13e94,0xdf837d09,0xbb27fa52,0x695059ac,
            0x52a1bb58,0xcc322f1d,0x1844565b,0xb4a8acf6,
            0x34235438,0x6847a851,0xe48c0cbb,0xcd181136,
            0x9a112a0c,0x43ec6d0e,0x87d8d27d,0x487dc995,
            0x90fb9b4b,0xa1f63697,0xfc513ed9,0x78a37d93,
            0x8d16c5df,0x9e0c8bbe,0x3c381f7c,0xe9fb0779};
```

5.2 Key Schedule

Order

t	X	Y	Z	W
0	a	b	c	d
1	b	a	d	c
2	c	d	a	b
3	d	c	b	a
4	a	c	d	b
5	b	d	c	a
6	c	a	b	d
7	d	b	a	c
8	a	d	b	c
9	b	c	a	d
10	c	b	d	a
11	d	a	c	b

Index

s	x	y	z	w
0	0	0	0	0
1	1	1	1	1
2	2	2	2	2
3	0	1	0	1
4	1	2	1	2
5	2	0	2	0
6	0	2	0	2
7	1	0	1	0
8	2	1	2	1

6 Speedup Techniques

This subsection describes the techniques to speed up the cipher. The optimum processing can be realized by selecting a speedup technique corresponding to the platform on which the cipher is implemented.

6.1 Data randomizing

6.1.1 Aligning tables

In the explanation of the S function in Section 3.6, 5-bit and 6-bit S-box tables are looked up by (6,5,5,5,5,6). However, executing the table lookup by (6,10,10,6) or (11,10,11), which isobtained by jointing two consecutive S-boxes into a new one, speeds up the operations because the number of table lookup is reduced.

6.1.2 Composing functions

In the F function, the S function, the M function and the L function are executed individually in this order. However, the S and M functions can be composed because both functions execute table lookups. In this case, creating the table S_5_M(5-bit input, 32-bit output) or S_6_M(6-bit input, 32-bit output) by composing two tables can reduce the number of table lookups, leading to a speedup of operations. Also, creating the table $S_5\_M\_L$(5-bit input, 64-bit output) or $S_6\_M\_L$(6-bit input, 64-bit output) by composing the L function can realize a further speedup. The technique can be used with the table composing described in Section 6.1.1. In this case, a table such as S_{10}_M or $S_{11}\_M\_L$ is used.

6.1.3 64-bit processing

Speedup of the I function and the R function can be realized by collectively executing 64-bit processing of two pairs of 32-bit input/output values instead of processing of four 32-bit input/output values. For the I function, the number of key table readings can be reduced. For the R function, 64-bit processing can be realized by composing the S function, the M function and the L function as described in Section 6.1.2.

6.1.4 Bitslicing

The B/B^{-1} function can execute Bitslicing by expressing the S_4 table in logical form. Bitslicing enables an aggregate processing of thirty-two S-box lookups, leading to calculation speedup. For example, the B and B^{-1} function can be expressed in logical form as shown in the following:

Syntax

$(e, f, g, h) = B\_func(a, b, c, d)$

$(e, f, g, h) = B^{-1}\_func(a, b, c, d)$

Processing

$B\_func()$	$B^{-1}\_func()$
$t1 = \overline{b}$	$t1 = c \oplus d$
$t2 = a \vee d$	$t2 = t1 \vee b$
$t3 = c \oplus t2$	$t3 = t2 \oplus c$
$t4 = t1 \wedge t3$	$t4 = t3 \wedge a$
$e = d \oplus t4$	$t5 = \overline{c}$
$t6 = t2 \oplus e$	$t6 = t5 \vee b$
$t7 = a \oplus b$	$t7 = t6 \oplus d$
$t8 = t6 \vee t7$	$e = t7 \oplus t4$
$f = c \oplus t8$	$t8 = a \oplus t6$
$t10 = c \wedge t3$	$t9 = t8 \vee d$
$t11 = t1 \oplus t10$	$g = t9 \oplus b$
$t12 = c \oplus t4$	$t10 = c \oplus g$
$t13 = t6 \wedge t12$	$t11 = t10 \vee t4$
$h = t7 \oplus t13$	$t12 = t11 \wedge t9$
$t15 = c \vee h$	$f = t12 \oplus e$
$g = t11 \oplus t15$	$t13 = t6 \oplus g$
	$t14 = t13 \wedge t1$
	$h = t14 \oplus a$

6.2 Key Schedule

6.2.1 Preliminary calculation of constants

The intermediate key generation function includes a part that processes the constants $(4i, 4i+1, 4i+2, 4i+3)$ $\{i = 0, 1, 2\}$ by using the S function and the M function. Since this part is independent of the key value, preliminary calculation of the twelve values such as $M\_func(S\_func(4i))$ can reduce the amount of calculation.

6.2.2 Reuse of values

The intermediate key generation function includes a part that processes the values $uk[0]$ to $uk[7]$ by using the S function and the M function. Since this part is independent of the key value, saving the calculated value of $M\_func(S\_func(uk[0]))$ to $M\_func(S\_func(uk[7]))$ can reduce the amount of calculation because recalculation is not required even if the i value is updated. Using the technique and the preliminary calculation of the constant described in Section 6.2.1 can reduce forty-eight calculations of S function and M function to twenty.

6.2.3 Alternate processing for multiplication

The intermediate generation function has multiplication, but only multipliers 1,2 and 3 are provided. Therefore, using additions in place of multiplication may sometimes speed up the calculations.

7 Test Vector

This section lists the intermediate keys, extended keys, and intermediate value test vectors for the 128-bit key in the cipher. The values after processing each function are indicated here.

```
KEYSIZE=128

KEY=0x00000000000000000000000000000000

InterMidiateKey:
A[0]=0x1E13B607, A[1]=0x8037CF49, A[2]=0x7A1DC8C8
B[0]=0xFFECA80A, B[1]=0xF2062F0C, B[2]=0x224A06EC
C[0]=0x968A7468, C[1]=0x064C0837, C[2]=0xDF864B05
D[0]=0x79FA438A, D[1]=0x937B7E58, D[2]=0xE4ABD855

ExtendedKey:
EXKEY[0]=0x5A215E97, EXKEY[1]=0x2511C596
EXKEY[2]=0x005B7B29, EXKEY[3]=0x05038ED3
EXKEY[4]=0xD6AC0212, EXKEY[5]=0xFF7F916B
EXKEY[6]=0x91685E19, EXKEY[7]=0xF529F0ED
EXKEY[8]=0xFB2704AA, EXKEY[9]=0x12399574
EXKEY[10]=0xB3E065AB, EXKEY[11]=0x7AF0674C
EXKEY[12]=0x1D1F4FE9, EXKEY[13]=0xD0CB45B9
EXKEY[14]=0xD19B0A98, EXKEY[15]=0xD80DDC82
EXKEY[16]=0xD8EEBBB5, EXKEY[17]=0xA5A601B4
EXKEY[18]=0x409687CF, EXKEY[19]=0xECBA0704
EXKEY[20]=0x12FCEC43, EXKEY[21]=0x57728321
EXKEY[22]=0x77507089, EXKEY[23]=0x9954BAB1
EXKEY[24]=0xCEA35202, EXKEY[25]=0x22F904B3
EXKEY[26]=0x56E2D16B, EXKEY[27]=0x49F5CF61
EXKEY[28]=0x6F5A3D80, EXKEY[29]=0xA0E27CAB
EXKEY[30]=0x75F71B60, EXKEY[31]=0x0893A481
EXKEY[32]=0x3226E7FB, EXKEY[33]=0x71A8BC68
EXKEY[34]=0x1D42352C, EXKEY[35]=0xD383B20B
EXKEY[36]=0xA7392344, EXKEY[37]=0xBCC151C8
EXKEY[38]=0x3C317191, EXKEY[39]=0x41AFC455
EXKEY[40]=0xEC4CB923, EXKEY[41]=0x4813D88F
EXKEY[42]=0xAF7CCC12, EXKEY[43]=0xE16A317F
EXKEY[44]=0x8B60307F, EXKEY[45]=0x86C032C0
EXKEY[46]=0x920D093E, EXKEY[47]=0xA244E311
EXKEY[48]=0x5B41E2E5, EXKEY[49]=0x4D08C78C
EXKEY[50]=0x12E28AB1, EXKEY[51]=0xB8F8B742
EXKEY[52]=0x830E156C, EXKEY[53]=0x5D757A55
EXKEY[54]=0xB8D8C053, EXKEY[55]=0x286BB72E

PT=0x00000000000000000000000000000000

Encrypt
I :0x5A215E97 0x2511C596 0x005B7B29 0x05038ED3
B :0x5F6BAEBB 0x7F238044 0xA5CDE028 0x7F30CB41
I :0x89C7ACA9 0x805C112F 0x34A5BE31 0x8A193BAC
R5:0x62F7FC9C 0xB31028AA 0x34A5BE31 0x8A193BAC
R5:0x6D990BE6 0x4A4C5B16 0x62F7FC9C 0xB31028AA
```

```
I :0x96BE0F4C 0x5875CE62 0xD1179937 0xC9E04FE6
B :0xCF685F7F 0x0FC84888 0x78D47833 0xDE5DC1AE
I :0xD2771096 0xDF030D31 0xA94F72AB 0x06501D2C
R3:0xE42994F4 0x01F0E34A 0xA94F72AB 0x06501D2C
R3:0xA1248C59 0x85E78E06 0xE42994F4 0x01F0E34A
I :0x79CA37EC 0x20418FB2 0xA4BF133B 0xED4AE44E
B :0xB47E840B 0xFD00E8C4 0x6B34AB67 0x100BCB3A
I :0xA6826848 0xAA726BE5 0x1C64DBEE 0x895F718B
R5:0xE3DFED88 0x45AE01FF 0x1C64DBEE 0x895F718B
R5:0xC3192C59 0xEA3FF103 0xE3DFED88 0x45AE01FF
I :0x0DBA7E5B 0xC8C6F5B0 0xB53D3CE3 0x0C5BCE9E
B :0x3C5BCC92 0x40C1870C 0x7240B788 0xC45CB9A6
I :0x5301F112 0xE023FBA7 0x07B7ACE8 0xCCCF1D27
R3:0x8A48F64F 0x9CB676B1 0x07B7ACE8 0xCCCF1D27
R3:0xCAD86200 0xFB78A8DF 0x8A48F64F 0x9CB676B1
I :0xF8FE85FB 0x8AD014B7 0x970AC363 0x4F35C4BA
B :0x2F10C6B2 0x65E5502E 0xC2243A2F 0xA2019005
I :0x8829E5F6 0xD92401E6 0xFE154BBE 0xE3AE5450
R5:0x86C215CB 0xC234FC75 0xFE154BBE 0xE3AE5450
R5:0x33CAE854 0x849E683C 0x86C215CB 0xC234FC75
I :0xDF865177 0xCC8DB0B3 0x29BED9D9 0x235ECD0A
B :0x313EC90E 0xF6552C2C 0x0AADB211 0x19CBF595
I :0xBA5EF971 0x70951EEC 0x98A0BB2F 0xBB8F1684
R3:0x0A18287A 0xF53200CF 0x98A0BB2F 0xBB8F1684
R3:0x78CDCB6D 0x865DE218 0x0A18287A 0xF53200CF
I :0x238C2988 0xCB552594 0x18FAA2CB 0x4DCAB78D
B :0x79EAAFCF 0xE607BE95 0xD86164F6 0xECD91C1C
I :0xFAE4BAA3 0xBB72C4C0 0x60B9A4A5 0xC4B2AB32

CT=0xFAE4BAA3BB72C4C060B9A4A5C4B2AB32

Decrypt
I :0x79EAAFCF 0xE607BE95 0xD86164F6 0xECD91C1C
B :0x238C2988 0xCB552594 0x18FAA2CB 0x4DCAB78D
I :0x78CDCB6D 0x865DE218 0x0A18287A 0xF53200CF
R3:0x98A0BB2F 0xBB8F1684 0x0A18287A 0xF53200CF
R3:0xBA5EF971 0x70951EEC 0x98A0BB2F 0xBB8F1684
I :0x313EC90E 0xF6552C2C 0x0AADB211 0x19CBF595
B :0xDF865177 0xCC8DB0B3 0x29BED9D9 0x235ECD0A
I :0x33CAE854 0x849E683C 0x86C215CB 0xC234FC75
R5:0xFE154BBE 0xE3AE5450 0x86C215CB 0xC234FC75
R5:0x8829E5F6 0xD92401E6 0xFE154BBE 0xE3AE5450
I :0x2F10C6B2 0x65E5502E 0xC2243A2F 0xA2019005
B :0xF8FE85FB 0x8AD014B7 0x970AC363 0x4F35C4BA
I :0xCAD86200 0xFB78A8DF 0x8A48F64F 0x9CB676B1
R3:0x07B7ACE8 0xCCCF1D27 0x8A48F64F 0x9CB676B1
R3:0x5301F112 0xE023FBA7 0x07B7ACE8 0xCCCF1D27
I :0x3C5BCC92 0x40C1870C 0x7240B788 0xC45CB9A6
B :0x0DBA7E5B 0xC8C6F5B0 0xB53D3CE3 0x0C5BCE9E
I :0xC3192C59 0xEA3FF103 0xE3DFED88 0x45AE01FF
R5:0x1C64DBEE 0x895F718B 0xE3DFED88 0x45AE01FF
R5:0xA6826848 0xAA726BE5 0x1C64DBEE 0x895F718B
```

```
I :0xB47E840B 0xFD00E8C4 0x6B34AB67 0x100BCB3A
B :0x79CA37EC 0x20418FB2 0xA4BF133B 0xED4AE44E
I :0xA1248C59 0x85E78E06 0xE42994F4 0x01F0E34A
R3:0xA94F72AB 0x06501D2C 0xE42994F4 0x01F0E34A
R3:0xD2771096 0xDF030D31 0xA94F72AB 0x06501D2C
I :0xCF685F7F 0x0FC84888 0x78D47833 0xDE5DC1AE
B :0x96BE0F4C 0x5875CE62 0xD1179937 0xC9E04FE6
I :0x6D990BE6 0x4A4C5B16 0x62F7FC9C 0xB31028AA
R5:0x34A5BE31 0x8A193BAC 0x62F7FC9C 0xB31028AA
R5:0x89C7ACA9 0x805C112F 0x34A5BE31 0x8A193BAC
I :0x5F6BAEBB 0x7F238044 0xA5CDE028 0x7F30CB41
B :0x5A215E97 0x2511C596 0x005B7B29 0x05038ED3
I :0x00000000 0x00000000 0x00000000 0x00000000

PT=0x00000000000000000000000000000000
```