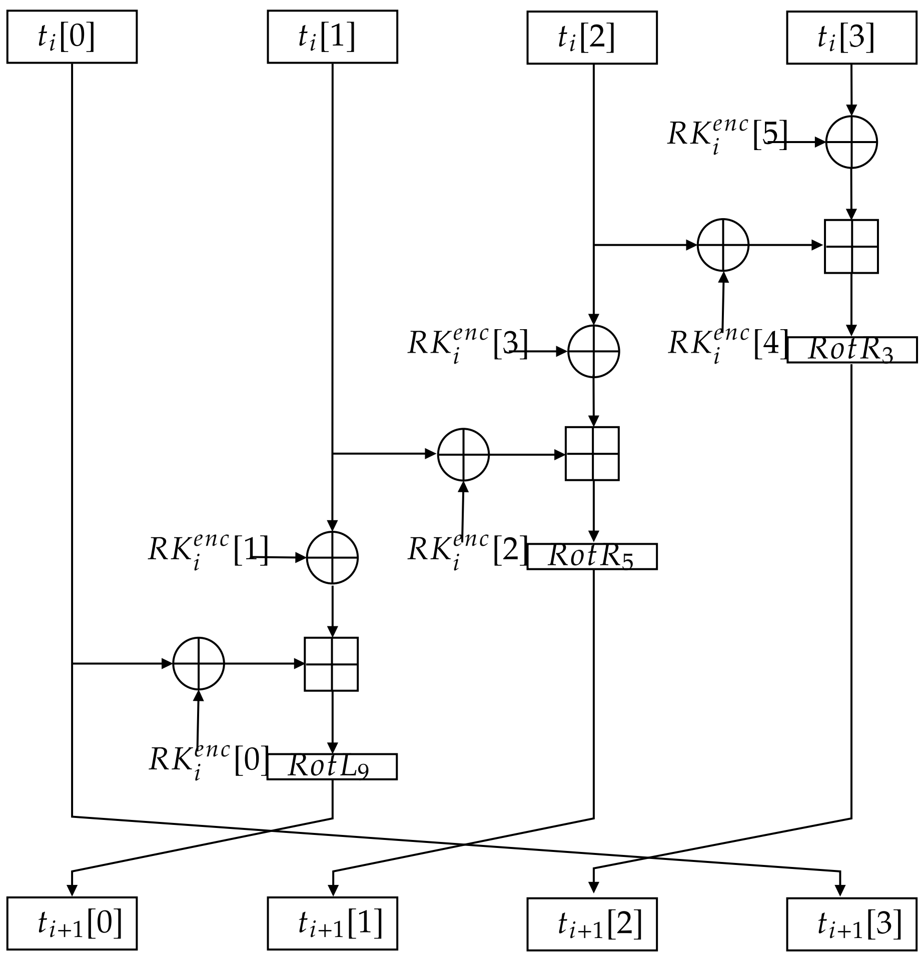
Lightweight Encryption Algorithm
- LEA -

Ji Yong-Hyeon



Department of Information Security, Cryptology, and Mathematics
College of Science and Technology
Kookmin University

January 15, 2024

List of Symbols

Contents

Chapter 1

LEA: Implementation in Little-Endian

1.1 Specification

Table 1.1: Specification Comparison between AES and LEA Block Ciphers

Specification	**AES**	**LEA**
Block Size (bits)	128	128
Key Size (bits)	128/192/256	128/192/256
Structure	Substitution-Permutation Network	Generalized Feistel Network
Rounds	10/12/14 (depends on key size)	24/28/32 (depends on key size)
Design Year	1998	2013

Table 1.2: Parameters of the Block Cipher LEA (1-word = 32-bit)

Algorithms	Block Size (N_b-byte)	Key Length (N_k-byte)	Number of Rounds (N_r)	Round-Key Length (byte)	Total Size of Round-Keys (($N_r * 192$)-bit)
LEA-128	16(4-word)	16(4-word)	24	24	4608 (144-word)
LEA-192	16(4-word)	24(6-word)	28	24	5376 (168-word)
LEA-256	16(4-word)	32(8-word)	32	24	6144 (192-word)

1.2 State Representation

Let $\mathsf{state}[0], \mathsf{state}[1], \ldots$ be representation of arrays of bytes. Note that

$$\mathsf{state}[i] := \left\{input_{8i}, input_{8i+1}, \ldots, input_{8i+7}\right\} \in \mathbb{F}_{2^8}$$

for $input_i \in \mathbb{F}_2$. For example, $\mathsf{state}[0] = \left\{input_0, input_1, \ldots, input_7\right\}$.

The 128-bit plaintext P of LEA is represented as an array of four 32-bit words $P[0], P[1], P[2]$ and $P[3]$. Then

$$P[i] = \mathsf{state}[4i+3] \parallel \mathsf{state}[4i+2] \parallel \mathsf{state}[4i+1] \parallel \mathsf{state}[4i] \quad \text{for} \quad 0 \le i \le 3.$$

Here, $P[i] \in \mathbb{F}_{2^{32=8\cdot 4}}$The key K of LEA is also represented as the same way.

Table 1.3: Representations for words, bytes, and bits

Input Bit Sequence	24	$\cdots$	31	16	$\cdots$	23	8	$\cdots$	15	0	$\cdots$	7
Word Number	0											
Byte Number	3			2			1			0		
Bit Numbers in Word	31	$\cdots$										1

Example 1.1.

128-bit Input String	`0x0f1e2d3c4b5a69788796a5b4c3d2e1f0`			
Split into Words	0x0f1e2d3c $P[0]$	0x4b5a6978 $P[1]$	0x8796a5b4 $P[2]$	0xc3d2e1f0 $P[3]$
$P[0]$ **(Word)**	0x0f1e2d3c			
$P[0]$ **(Bit)**	0b 0000:1111:0001:1110:0010:1101:0011:1010			
Split into Bytes	0x0f state[3]	0x1e state[2]	0x2d state[1]	0x3c state[0]
state[0] **(Byte)**	0x3c			
Split into Bits	1111:0000 24 $\cdots$ 31	- 16 $\cdots$ 23	- 8 $\cdots$ 15	- 0 $\cdots$ 7

```c
void stringToWordArray(const char* hexString, u32* wordArray) {
    size_t length = strlen(hexString);
    for (size_t i = 0; i < length; i += 8) {
        sscanf(&hexString[i], "%8x", &wordArray[i / 8]);
    }
}

const char* inputString = "0f1e2d3c4b5a69788796a5b4c3d2e1f0";
u32 key[4];
stringToWordArray(inputString, key);
```

```
(gdb) x/16xb key
0x7fffffffd9c0: 0x3c  0x2d  0x1e  0x0f  0x78  0x69  0x5a  0x4b
0x7fffffffd9c8: 0xb4  0xa5  0x96  0x87  0xf0  0xe1  0xd2  0xc3
```

1.3 Key Schedule

$$\mathrm{KeySchedule}^{\mathsf{enc}}_{128} : \{0,1\}^{128=8\cdot 16} \longrightarrow \{0,1\}^{4608=192\cdot 24}$$

$$\mathrm{KeySchedule}^{\mathsf{enc}}_{192} : \{0,1\}^{192=8\cdot 24} \longrightarrow \{0,1\}^{5376=192\cdot 28}$$

$$\mathrm{KeySchedule}^{\mathsf{enc}}_{256} : \{0,1\}^{256=8\cdot 32} \longrightarrow \{0,1\}^{6144=192\cdot 24}$$

1.3.1 Round Constant

The constant $\delta[i] \in \mathbb{F}_{2^{32}}$ $(i \in \{1,\ldots,7\})$ is as follows:

i	$\delta[i]$	value
0	$\delta[0]$	`0xc3efe9db`
1	$\delta[1]$	`0x44626b02`
2	$\delta[2]$	`0x79e27c8a`
3	$\delta[3]$	`0x78df30ec`
4	$\delta[4]$	`0x715ea49e`
5	$\delta[5]$	`0xc785da0a`
6	$\delta[6]$	`0xe04ef22a`
7	$\delta[7]$	`0xe5c40957`

1.3.2 Rotation Function

Algorithm 1: Rotation to Left and Right

/* $\mathtt{RotL} : \{0,1\}^{32} \times \{0,1\}^{32} \rightarrow \{0,1\}^{32}$ */

Function RotL(value, shift):

  return (value $\lll$ shift) | (value $\ggg$ (32 − shift));

end

/* $\mathtt{RotR} : \{0,1\}^{32} \times \{0,1\}^{32} \rightarrow \{0,1\}^{32}$ */

Function RotR(value, shift):

  return (value $\ggg$ shift) | (value $\lll$ (32 − shift));

end

1.3.3 Encryption Key Schedule of LEA-128

Algorithm 2: Encryption Key Schedule (LEA-128)

Input: User-key $\mathsf{UK} = \mathsf{UK}[0] \parallel \mathsf{UK}[1] \parallel \mathsf{UK}[2] \parallel \mathsf{UK}[3]$ $(\mathsf{UK}[i] \in \{0,1\}^{32})$

Output: Encryption Round-keys $\{\mathsf{RK}_i^{\mathsf{enc}}\}_{i=0}^{23}$ $(\mathsf{RK}_i^{\mathsf{enc}} \in \{0,1\}^{192})$

/* $\mathsf{UK} \in \{0,1\}^{128}$ is 16-byte and $\left\{\mathsf{RK}_i^{\mathsf{enc}}\right\}_{i=0}^{23} \in \{0,1\}^{4608}$ is 576-byte */

for $i = 0$ **to** 3 **do**

  $T[i] = \mathsf{UK}[i]$ // $T = T[0] \parallel \cdots \parallel T[3] \in \{0,1\}^{128=32*4}$

end

for $i = 0$ **to** 23 **do**

  $T[0] \leftarrow \mathtt{RotL}(T[0] \boxplus \mathtt{RotL}(\delta[i \bmod 4], i+0), 1)$ // $T[i] \in \{0,1\}^{32}$

  $T[1] \leftarrow \mathtt{RotL}(T[1] \boxplus \mathtt{RotL}(\delta[i \bmod 4], i+1), 3)$

  $T[2] \leftarrow \mathtt{RotL}(T[2] \boxplus \mathtt{RotL}(\delta[i \bmod 4], i+2), 6)$

  $T[3] \leftarrow \mathtt{RotL}(T[3] \boxplus \mathtt{RotL}(\delta[i \bmod 4], i+3), 11)$

  $\mathsf{RK}_i^{\mathsf{enc}} \leftarrow T[0] \parallel T[1] \parallel T[2] \parallel T[1] \parallel T[3] \parallel T[1]$ // $\mathsf{RK}_i^{\mathsf{enc}} \in \{0,1\}^{196=32*6}$

end

return $\left\{\mathsf{RK}_i^{\mathsf{enc}}\right\}_{i=0}^{23}$

1.3.4 Decryption Key Schedule of LEA-128

Algorithm 3: Decryption Key Schedule (LEA-128)

Input: User-key $\mathsf{UK} = \mathsf{UK}[0] \,\|\, \mathsf{UK}[1] \,\|\, \mathsf{UK}[2] \,\|\, \mathsf{UK}[3]$ ($\mathsf{UK}[i] \in \{0,1\}^{32}$)
Output: Decryption Round-keys $\{\mathsf{RK}_i^{\mathsf{dec}}\}_{i=0}^{23}$ ($\mathsf{RK}_i^{\mathsf{dec}} \in \{0,1\}^{192}$)

/* $\mathsf{UK} \in \{0,1\}^{128}$ is 16-byte and $\left\{\mathsf{RK}_i^{\mathsf{dec}}\right\}_{i=0}^{23} \in \{0,1\}^{4608}$ is 576-byte */

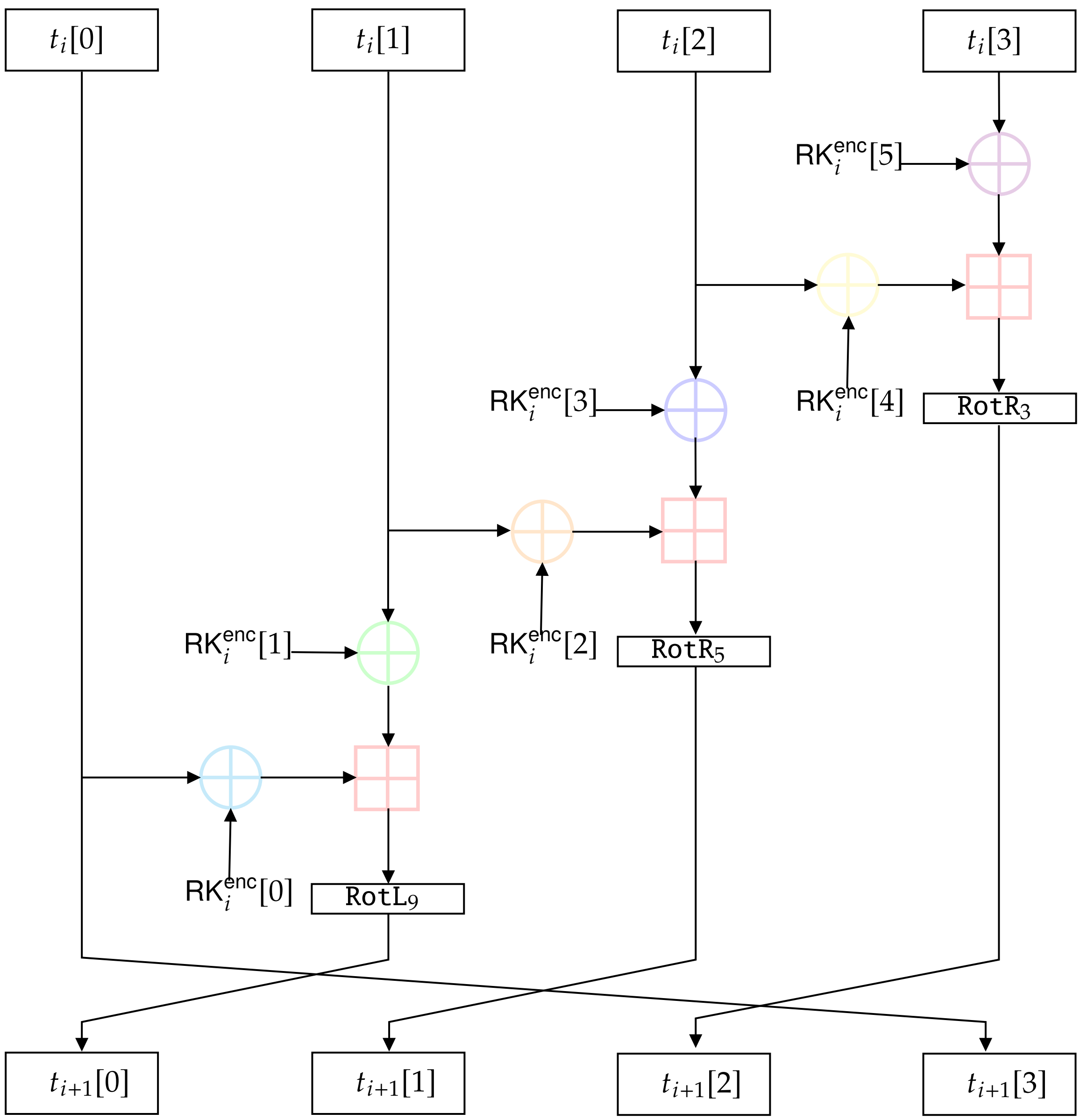
for $i = 0$ **to** 3 **do**
  $T[i] = \mathsf{UK}[i]$ // $T = T[0] \,\|\, \cdots \,\|\, T[3] \in \{0,1\}^{128=32*4}$
end
for $i = 0$ **to** 23 **do**
  $T[0] \leftarrow \mathtt{RotL}(T[0] \boxplus \mathtt{RotL}(\delta[i \bmod 4], i+0), 1)$ // $T[i] \in \{0,1\}^{32}$
  $T[1] \leftarrow \mathtt{RotL}(T[1] \boxplus \mathtt{RotL}(\delta[i \bmod 4], i+1), 3)$
  $T[2] \leftarrow \mathtt{RotL}(T[2] \boxplus \mathtt{RotL}(\delta[i \bmod 4], i+2), 6)$
  $T[3] \leftarrow \mathtt{RotL}(T[3] \boxplus \mathtt{RotL}(\delta[i \bmod 4], i+3), 11)$
  $\mathsf{RK}_{\mathbf{23-i}}^{\mathsf{dec}} \leftarrow T[0] \,\|\, T[1] \,\|\, T[2] \,\|\, T[1] \,\|\, T[3] \,\|\, T[1]$ // $\mathsf{RK}_i^{\mathsf{dec}} \in \{0,1\}^{196=32*6}$
end
return $\left\{\mathsf{RK}_i^{\mathsf{dec}}\right\}_{i=0}^{23}$

1.4 Encryption of LEA-128

Algorithm 4: Encryption of LEA-128

Input: block src = src[0] ‖ src[1] ‖ src[2] ‖ src[3] $\in \{0, 1\}^{128=32*4}$ and $\{\mathsf{RK}_i^{\mathsf{enc}}\}_{i=0}^{N_r-1=23}$
Output: block dst = dsc[0] ‖ dsc[1] ‖ dsc[2] ‖ dsc[3] $\in \{0, 1\}^{128=32*4}$

$t_0 = t[0] \parallel t[1] \parallel t[2] \parallel t[3] \leftarrow \mathsf{src}$
for $i = 0$ **to** 23 **do**
  $tmp \leftarrow t[0]$
  $t_{i+1}[0] \leftarrow \mathtt{RotL}(\, t_i[0] \oplus \mathsf{RK}_i^{\mathsf{enc}}[0] \boxplus (t_i[1] \oplus \mathsf{RK}_i^{\mathsf{enc}}[1])\,, 9)$
  $t_{i+1}[1] \leftarrow \mathtt{RotR}(\, t_i[1] \oplus \mathsf{RK}_i^{\mathsf{enc}}[2] \boxplus (t_i[2] \oplus \mathsf{RK}_i^{\mathsf{enc}}[3])\,, 5)$
  $t_{i+1}[2] \leftarrow \mathtt{RotR}(\, t_i[2] \oplus \mathsf{RK}_i^{\mathsf{enc}}[4] \boxplus (t_i[3] \oplus \mathsf{RK}_i^{\mathsf{enc}}[5])\,, 3)$
  $t_{i+1}[3] \leftarrow tmp$
end
return dst $\leftarrow t_{N_r}$



1.5 Decryption of LEA-128

Algorithm 5: Encryption of LEA-128

Input: block $\mathsf{src} = \mathsf{src}[0] \parallel \mathsf{src}[1] \parallel \mathsf{src}[2] \parallel \mathsf{src}[3] \in \{0, 1\}^{128=32*4}$ and $\{\mathsf{RK}_i^{\mathsf{enc}}\}_{i=0}^{N_r-1=23}$

Output: block $\mathsf{dst} = \mathsf{dsc}[0] \parallel \mathsf{dsc}[1] \parallel \mathsf{dsc}[2] \parallel \mathsf{dsc}[3] \in \{0, 1\}^{128=32*4}$

$t_0 = t[0] \parallel t[1] \parallel t[2] \parallel t[3] \leftarrow \mathsf{src}$
for $i = 0$ **to** 23 **do**
  $tmp \leftarrow t[0]$
  $t_{i+1}[0] \leftarrow \mathtt{RotL}(\, t_i[0] \oplus \mathsf{RK}_i^{\mathsf{enc}}[0] \boxplus (t_i[1] \oplus \mathsf{RK}_i^{\mathsf{enc}}[1])\, , 9)$
  $t_{i+1}[1] \leftarrow \mathtt{RotR}(\, t_i[1] \oplus \mathsf{RK}_i^{\mathsf{enc}}[2] \boxplus (t_i[2] \oplus \mathsf{RK}_i^{\mathsf{enc}}[3])\, , 5)$
  $t_{i+1}[2] \leftarrow \mathtt{RotR}(\, t_i[2] \oplus \mathsf{RK}_i^{\mathsf{enc}}[4] \boxplus (t_i[3] \oplus \mathsf{RK}_i^{\mathsf{enc}}[5])\, , 3)$
  $t_{i+1}[3] \leftarrow tmp$
end
return $\mathsf{dst} \leftarrow t_{N_r}$

Algorithm 6: Decryption of LEA-128

Input: block $\mathsf{src} \in \{0, 1\}^{128=8*16}$, decryption round-keys $\{\mathsf{RK}_i^{\mathsf{dec}}\}_{i=0}^{N_r-1=23}$

Output: block $\mathsf{dst} \in \{0, 1\}^{128=8*16}$

$t_0 \leftarrow \mathsf{src}$
for $i = 0$ **to** $N_r - 1$ **do**
  $t_{i+1}[0] \leftarrow t_i[3]$
  $t_{i+1}[1] \leftarrow (\mathtt{RotR}(t_i[0], 9) \boxminus (t_{i+1}[0] \oplus \mathsf{RK}_i^{\mathsf{dec}}[0])) \oplus \mathsf{RK}_i^{\mathsf{dec}}[1]$
  $t_{i+1}[2] \leftarrow (\mathtt{RotL}(t_i[1], 5) \boxminus (t_{i+1}[1] \oplus \mathsf{RK}_i^{\mathsf{dec}}[2])) \oplus \mathsf{RK}_i^{\mathsf{dec}}[3]$
  $t_{i+1}[3] \leftarrow (\mathtt{RotL}(t_i[2], 3) \boxminus (t_{i+1}[2] \oplus \mathsf{RK}_i^{\mathsf{dec}}[4])) \oplus \mathsf{RK}_i^{\mathsf{dec}}[5]$
end
return $\mathsf{dst} \leftarrow t_{N_r}$

Chapter 2

Modes of Operation

Table 2.1: Comparison of Modes

Mode	Integrity	Authentication	EncryptBlk	DecryptBlk	Padding	IV	$\lvert P\rvert \stackrel{?}{=} \lvert C\rvert$
ECB	O	X	O	O	O	X	$\lvert P\rvert < \lvert C\rvert$
CBC	O	X	O	O	O	O	$\lvert P\rvert < \lvert C\rvert$
OFB	O	X	O	X	X	O	$\lvert P\rvert = \lvert C\rvert$
CFB	O	X	O	X	X	O	$\lvert P\rvert = \lvert C\rvert$
CTR	O	X	O	X	X	O	$\lvert P\rvert = \lvert C\rvert$
CBC-CS	O	X	O	O	X	O	$\lvert P\rvert = \lvert C\rvert$

2.1 Padding

Block ciphers require input lengths to be a multiple of the block size. Padding is used to extend the last block of plaintext to the required length. Without proper padding, the encryption process may be insecure or infeasible.

There are several padding schemes used in practice, such as:

Table 2.2: Padding Standards in Block Ciphers

Standard Name	**Padding Method**
PKCS#7	Pad with bytes all the same value as the number of padding bytes `...dd | dd dd dd dd dd dd dd dd dd dd dd dd 04 04 04 04 |`
ANSI X9.23	Pad with zeros, last byte is the number of padding bytes `...dd | dd dd dd dd dd dd dd dd dd dd dd 00 00 00 00 05 |`
ISO/IEC 7816-4	First byte is '80' (hex), followed by zeros `...dd | dd dd dd dd dd dd dd dd dd dd 80 00 00 00 00 00 |`
ISO 10126	Pad with random bytes, last byte is the number of padding bytes `...dd | dd dd dd dd dd dd dd dd dd dd 2e 49 1b c1 aa 06 |`

2.1.1 PKCS#7

```
void PKCS7_PAD(u32* block, size_t block_len, size_t input_len) {
    if (block_len < input_len) {
        fprintf(stderr,
            "Block length must be greater than input length.\n");
        return;
    }

    u8 padding_value = block_len - input_len;
    for (size_t i = input_len; i < block_len; ++i) {
        block[i] = padding_value;
    }
}
```

2.2 ECB (Electronic CodeBook)

Algorithm 7: Electronic CodeBook

Input: K and $P = P_1 \parallel \cdots \parallel P_N$ ($P_i \in \{0, 1\}^n$)
Output: $C = C_1 \parallel \cdots \parallel C_N$ ($C_i \in \{0, 1\}^n$)

for $i \leftarrow 1$ **to** N **do**
  $C_i \leftarrow \mathsf{EncryptBlk}(K, P_i)$;
end
return $C = C_1 \parallel \cdots \parallel C_N$;

Input: K and $C = C_1 \parallel \cdots \parallel C_N$ ($C_i \in \{0, 1\}^n$)
Output: $P = P_1 \parallel \cdots \parallel P_N$ ($P_i \in \{0, 1\}^n$)

for $i \leftarrow 1$ **to** N **do**
  $P_i \leftarrow \mathsf{DecryptBlk}(K, C_i)$;
end
return $C = C_1 \parallel \cdots \parallel C_N$;

Chapter 3

Mode of Operations Validation System (MOVS)

3.1 Structure

LEA128(CBC)KAT.txt:

KEY = 00000000000000000000000000000000
IV = 00000000000000000000000000000000
PT = 80000000000000000000000000000000
CT = CE8DCF04DD60982B1D8F5035FD534DE2

LEA128(CBC)MMT.txt:

KEY = 724F8279123B9307658A109101466A15
IV = FEF995090770B941B0BF2B120A859BB8
PT = 76718FF5B60510FB4A9AA28CF9A57A60
CT = D65A9EC458B768C4E9BE62C6FE04A51D

KEY = AE38ECC785CC238F263D14285216B406
IV = BB0F694719D4BF967A085D4FD98A37E3
PT = F3F7057F5670F3E8BB9D9AAA95F12F71EA30FAB7622F0A9F9EDC2821CA7D0968
CT = D504FFC845EF447C516799AD8877F967628F49F15A8B64FA48AD215232884A52

LEA128(CBC)MCT.txt:

COUNT = 0
KEY = 724F8279123B9307658A109101466A15
IV = FEF995090770B941B0BF2B120A859BB8
PT = 76718FF5B60510FB4A9AA28CF9A57A60
CT = 92E58A0B613CA94D922F7E6A445A4EC7

```
typedef struct {
    u32 key[4];      // Fixed 128 bits for key
    u32 iv[4];       // Fixed 128 bits for iv
    u32* pt;         // Pointer for arbitrary length plaintext
    size_t ptLength; // Length of pt
    u32* ct;         // Pointer for arbitrary length ciphertext
    size_t ctLength; // Length of ct
} CryptoData; // 64-byte (16 + 16 + 8 + 8 + 8 + 8)
```

3.1.1 freeCryptoData()

```
void freeCryptoData(CryptoData* cryptoData) {
    if (cryptoData != NULL) {
        // Free the dynamically allocated memory for pt and ct
        free(cryptoData->pt);
        free(cryptoData->ct);

        // Set the pointers to NULL to avoid dangling pointers
        cryptoData->pt = NULL;
        cryptoData->ct = NULL;

        // Reset the lengths to zero
        cryptoData->ptLength = 0;
        cryptoData->ctLength = 0;
    }
}
```

3.1.2 parseHexLine()

```
void parseHexLine(u32* arr, const char* line) {
    for (int i = 0; i < 4; i++) {
        u32 value;
        sscanf(line + i * 8, "%8x", &value);
        *(arr + i) = value;
    }
}
```

3.1.3 parseHexLineVariable()

```
void parseHexLineVariable(u32* arr, const char* line, size_t
   length) {
    for (size_t i = 0; i < length; i++) {
        u32 value;
        // Ensure not to read beyond the line's end
        if (sscanf(line + i * 8, "%8x", &value) != 1) {
            // Handle parsing error, such as setting a default
               value or logging an error
            arr[i] = 0; // Example: set to zero if parsing fails
        } else {
            arr[i] = value;
        }
    }
}
```

3.1.4 determineLength()

```c
size_t determineLength(const char* hexString) {
    // Calculate the length of the hexadecimal string
    size_t hexLength = strlen(hexString);

    // Convert hex length to the length of the u32 array
    size_t u32Length = hexLength / 8;

    // If the hex string length is not a multiple of 8, add an
       extra element
    if (hexLength % 8 != 0) {
        u32Length++;
    }

    return u32Length;
}
```

3.1.5 readCryptoData()

```c
int readCryptoData(FILE* fp, CryptoData* cryptoData) {
     // Assuming each line will not exceed this length
    char line[INITIAL_BUF_SIZE];

    while (fgets(line, sizeof(line), fp) != NULL) {
        if (strncmp(line, "KEY =", 5) == 0) {
            // Skip "KEY = "
            parseHexLine(cryptoData->key, line + 6);
        } else if (strncmp(line, "IV =", 4) == 0) {
            // Skip "IV = "
            parseHexLine(cryptoData->iv, line + 5);
        } else if (strncmp(line, "PT =", 4) == 0) {
            // Calculate length
            cryptoData->ptLength = determineLength(line + 5);
            cryptoData->pt = (u32*)malloc(cryptoData->ptLength * sizeof(u32));
            if (cryptoData->pt == NULL) return -1;
            parseHexLineVariable(cryptoData->pt, line + 5, cryptoData->ptLength);
        } else if (strncmp(line, "CT =", 4) == 0) {
            // Calculate length
            cryptoData->ctLength = determineLength(line + 5);
            cryptoData->ct = (u32*)malloc(cryptoData->ctLength * sizeof(u32));
            if (cryptoData->ct == NULL) {
                // Free pt if ct allocation fails
                free(cryptoData->pt);
                return -1;
            }
            parseHexLineVariable(cryptoData->ct, line + 5, cryptoData->ctLength);
        }
        // Add more conditions here if there are more data types
    }

    return 0; // Return 0 on successful read
}
```

3.1.6 compareCryptoData()

```
bool compareCryptoData(const CryptoData* data1, const CryptoData*
    data2) {
    // Compare fixed-size arrays: key and iv
    for (int i = 0; i < 4; i++) {
        if (data1->key[i] != data2->key[i] || data1->iv[i] !=
            data2->iv[i]) {
            return 0; // Not equal
        }
    }

    // Compare lengths and contents of pt
    if (data1->ptLength != data2->ptLength) {
        return 0; // Not equal
    }
    for (size_t i = 0; i < data1->ptLength; i++) {
        if (data1->pt[i] != data2->pt[i]) {
            return 0; // Not equal
        }
    }

    // Compare lengths and contents of ct
    if (data1->ctLength != data2->ctLength) {
        return 0; // Not equal
    }
    for (size_t i = 0; i < data1->ctLength; i++) {
        if (data1->ct[i] != data2->ct[i]) {
            return 0; // Not equal
        }
    }

    return 1; // All comparisons passed, data structures are equal
}
```

3.2 Known Answer Test (KAT)

3.2.1 Overview

```
void MOVS_LEA128CBC_KAT_TEST() {
    const char* folderPath = "../LEA128(CBC)MOVS/";
    char txtFileName[50]; char reqFileName[50];
    char faxFileName[50]; char rspFileName[50];

    // Construct full paths for input and output files
    snprintf(txtFileName, sizeof(txtFileName), "%s%s", folderPath,
        "LEA128(CBC)KAT.txt");
    snprintf(reqFileName, sizeof(reqFileName), "%s%s", folderPath,
        "LEA128(CBC)KAT.req");
    snprintf(faxFileName, sizeof(faxFileName), "%s%s", folderPath,
        "LEA128(CBC)KAT.fax");
    snprintf(rspFileName, sizeof(rspFileName), "%s%s", folderPath,
        "LEA128(CBC)KAT.rsp");

    create_LEA128CBC_KAT_ReqFile(txtFileName, reqFileName);
    create_LEA128CBC_KAT_FaxFile(txtFileName, faxFileName);
    create_LEA128CBC_KAT_RspFile(reqFileName, rspFileName);

    printf("\nLEA128-CBC-KAT-TEST:\n");
    FILE* file1 = fopen(faxFileName, "r");
    FILE* file2 = fopen(rspFileName, "r");
    if (!file1 || !file2) {
        perror("Error opening files"); return;
    }

    CryptoData data1, data2;
    memset(&data1, 0, sizeof(CryptoData));
    memset(&data2, 0, sizeof(CryptoData));
    int result = 1; // Default to pass
    int idx = 1;
    int totalTests = 275; // Assuming a total of 275 tests
    int passedTests = 0;
    while (idx <= totalTests) {
        if (readCryptoData(file1, &data1) == -1 || readCryptoData(
           file2, &data2) == -1) {
            result = 0; // Indicate failure if read fails
            break;
        }

        if (!compareCryptoData(&data1, &data2)) {
            result = 0; // Fail
            printf("\nFAIL\n");
            break;
        }
```

```c

        // Free the dynamically allocated memory
        freeCryptoData(&data1);
        freeCryptoData(&data2);

        // Reset the structures for the next iteration
        memset(&data1, 0, sizeof(CryptoData));
        memset(&data2, 0, sizeof(CryptoData));

        passedTests++;
        printProgressBar(idx++, totalTests);
    }

    printf("\n\nTesting Summary:\n");
    printf("Passed: %d/%d\n", passedTests, totalTests);
    if (result) {
        printf("Perfect PASS !!!\n");
    } else {
        printf("Some tests FAILED.\n");
    }

    fclose(file1);
    fclose(file2);
}
```

3.2.2 create_LEA128CBC_KAT_ReqFile()

```c
void create_LEA128CBC_KAT_ReqFile(const char* inputFileName, const
    char* outputFileName) {
    FILE *infile, *reqFile;
    char* line;
    size_t bufsize = INITIAL_BUF_SIZE;
    // Flag to check if it's the first KEY line
    int isFirstKey = 1;

    // Open the source text file for reading
    infile = fopen(inputFileName, "r");
    if (infile == NULL) {
        perror("Error opening input file");
        return;
    }

    // Open the .req file for writing
    reqFile = fopen(outputFileName, "w");
    if (reqFile == NULL) {
        perror("Error opening .req file");
        fclose(infile);
        return;
    }
```

```c

    // Allocate initial buffer
    line = (char*)malloc(bufsize * sizeof(char));
    if (line == NULL) {
        perror("Unable to allocate memory");
        fclose(infile);
        fclose(reqFile);
        return;
    }

    // Read the source file line by line
    while (fgets(line, bufsize, infile) != NULL) {
        if (strncmp(line, "KEY", 3) == 0) {
            if (!isFirstKey) {
                // If not the first KEY, add a newline before
                   writing the line
                fputc('\n', reqFile);
            }
            isFirstKey = 0;
            fputs(line, reqFile);
        } else if (strncmp(line, "IV", 2) == 0 || strncmp(line, "
           PT", 2) == 0) {
            fputs(line, reqFile);
        }
    }

    // Free the allocated line buffer and close files
    free(line);
    fclose(infile);
    fclose(reqFile);

    printf("LEA128(CBC)KAT.req file has been successfully created
       in 'LEA128(CBC)MOVS' folder.\n");
}
```

3.2.3 create_LEA128CBC_KAT_FaxFile()

3.2.4 create_LEA128CBC_KAT_RspFile()

3.3 Multi-block Message Test (MMT)

3.4 Monte Carlo Test (MCT)

Appendix A

Additional Data A

A.1 Substitution-BOX