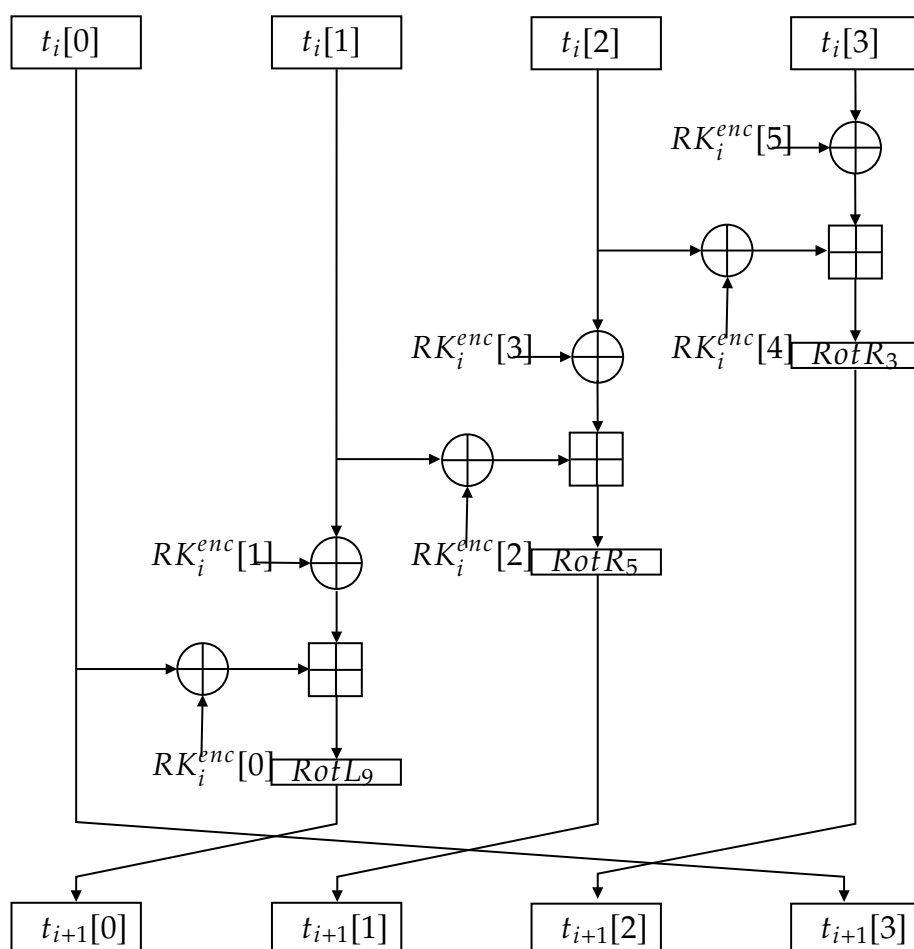


Lightweight Encryption Algorithm

- LEA -

Ji Yong-Hyeon



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

January 22, 2024

List of Symbols

Contents

1	LEA: Implementation in Little-Endian	1
1.1	Specification	1
1.2	State Representation	1
1.3	Key Schedule	2
1.3.1	Round Constant	3
1.3.2	Rotation Function	3
1.3.3	Encryption Key Schedule of LEA-128	3
1.3.4	Decryption Key Schedule of LEA-128	4
1.4	Encryption of LEA-128	5
1.5	Decryption of LEA-128	6
2	Modes of Operation	7
2.1	Padding	7
2.1.1	PKCS#7	8
2.2	ECB (Electronic CodeBook)	9
2.3	CBC (Cipher Block Chaining)	10
3	Mode of Operations Validation System (MOVS)	11
3.1	Structure	11
3.1.1	freeCryptoData()	12
3.1.2	parseHexLine()	12
3.1.3	parseHexLineVariable()	12
3.1.4	determineLength()	13
3.1.5	readCryptoData()	14
3.1.6	compareCryptoData()	15
3.2	Known Answer Test (KAT)	16
3.2.1	Overview	16
3.2.2	create_LEA128CBC_KAT_ReqFile()	17
3.2.3	create_LEA128CBC_KAT_FaxFile()	18
3.2.4	create_LEA128CBC_KAT_RspFile()	18
3.3	Multi-block Message Test (MMT)	19
3.4	Monte Carlo Test (MCT)	20
A	Additional Data A	21
A.1	Substitution-BOX	21

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 (ARX - Add-Rotation-Xor)
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 $\text{state}[0], \text{state}[1], \dots$ be representation of arrays of bytes. Note that

$$\text{state}[i] := \{input_{8i}, input_{8i+1}, \dots, input_{8i+7}\} \in \mathbb{F}_{2^8}$$

for $input_i \in \mathbb{F}_2$. For example, $\text{state}[0] = \{input_0, input_1, \dots, input_7\}$.

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

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

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	...	31	16	...	23	8	...	15	0	...	7
Word Number	0											
Byte Number	3			2			1			0		
Bit Numbers in Word	31	...										1

Example 1.1.

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

```

1 void stringToWordArray(const char* hexString, u32* wordArray) {
2     size_t length = strlen(hexString);
3     for (size_t i = 0; i < length; i += 8) {
4         sscanf(&hexString[i], "%8x", &wordArray[i / 8]);
5     }
6 }
7
8 const char* inputString = "0f1e2d3c4b5a69788796a5b4c3d2e1f0";
9 u32 key[4];
10 stringToWordArray(inputString, key);

```

```
(gdb) x/16xb key
```

```

0x7fffffffdd9c0: 0x3c  0x2d  0x1e  0x0f  0x78  0x69  0x5a  0x4b
0x7fffffffdd9c8: 0xb4  0xa5  0x96  0x87  0xf0  0xe1  0xd2  0xc3

```

1.3 Key Schedule

$$\text{KeySchedule}_{128}^{\text{enc}} : \{0, 1\}^{128=32 \cdot 4} \rightarrow \{0, 1\}^{192 \cdot 24=4608=32 \cdot 144}$$

$$\text{KeySchedule}_{192}^{\text{enc}} : \{0, 1\}^{192=32 \cdot 6} \rightarrow \{0, 1\}^{192 \cdot 28=5376=32 \cdot 168}$$

$$\text{KeySchedule}_{256}^{\text{enc}} : \{0, 1\}^{256=32 \cdot 8} \rightarrow \{0, 1\}^{192 \cdot 32=6144=32 \cdot 192}$$

1.3.1 Round Constant

The constant $\delta[i] \in \mathbb{F}_{2^{32}}$ ($i \in \{1, \dots, 7\}$) 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

```

/* RotL :  $\{0, 1\}^{32} \times \{0, 1\}^{32} \rightarrow \{0, 1\}^{32}$  */
1 Function RotL(value, shift):
2   | return (value  $\ll$  shift) | (value  $\gg$  (32 – shift));
3 end

/* RotR :  $\{0, 1\}^{32} \times \{0, 1\}^{32} \rightarrow \{0, 1\}^{32}$  */
4 Function RotR(value, shift):
5   | return (value  $\gg$  shift) | (value  $\ll$  (32 – shift));
6 end

```

1.3.3 Encryption Key Schedule of LEA-128

Algorithm 2: Encryption Key Schedule (LEA-128)

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

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

/* $UK \in \{0, 1\}^{128}$ is 16-byte and $\{RK_i^{\text{enc}}\}_{i=0}^{23} \in \{0, 1\}^{4608}$ is 576-byte */

```

1 for  $i = 0$  to 3 do
2   |  $T[i] = UK[i]$  //  $T = T[0] \parallel \dots \parallel T[3] \in \{0, 1\}^{128=32*4}$ 
3 end
4 for  $i = 0$  to 23 do
5   |  $T[0] \leftarrow \text{RotL}(T[0] \boxplus \text{RotL}(\delta[i \bmod 4], i + 0), 1)$  //  $T[i] \in \{0, 1\}^{32}$ 
6   |  $T[1] \leftarrow \text{RotL}(T[1] \boxplus \text{RotL}(\delta[i \bmod 4], i + 1), 3)$ 
7   |  $T[2] \leftarrow \text{RotL}(T[2] \boxplus \text{RotL}(\delta[i \bmod 4], i + 2), 6)$ 
8   |  $T[3] \leftarrow \text{RotL}(T[3] \boxplus \text{RotL}(\delta[i \bmod 4], i + 3), 11)$ 
9   |  $RK_i^{\text{enc}} \leftarrow T[0] \parallel T[1] \parallel T[2] \parallel T[1] \parallel T[3] \parallel T[1]$  //  $RK_i^{\text{enc}} \in \{0, 1\}^{196=32*6}$ 
10 end
11 return  $\{RK_i^{\text{enc}}\}_{i=0}^{23}$ 

```

1.3.4 Decryption Key Schedule of LEA-128

Algorithm 3: Decryption Key Schedule (LEA-128)

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

Output: Decryption Round-keys $\{RK_i^{\text{dec}}\}_{i=0}^{23}$ ($RK_i^{\text{dec}} \in \{0, 1\}^{192}$)

/ $UK \in \{0, 1\}^{128}$ is 16-byte and $\{RK_i^{\text{dec}}\}_{i=0}^{23} \in \{0, 1\}^{4608}$ is 576-byte */*

```

1 for  $i = 0$  to 3 do
2    $T[i] = UK[i]$                                 //  $T = T[0] \parallel \dots \parallel T[3] \in \{0, 1\}^{128=32*4}$ 
3 end
4 for  $i = 0$  to 23 do
5    $T[0] \leftarrow \text{RotL}(T[0] \boxplus \text{RotL}(\delta[i \bmod 4], i + 0), 1)$                                 //  $T[i] \in \{0, 1\}^{32}$ 
6    $T[1] \leftarrow \text{RotL}(T[1] \boxplus \text{RotL}(\delta[i \bmod 4], i + 1), 3)$ 
7    $T[2] \leftarrow \text{RotL}(T[2] \boxplus \text{RotL}(\delta[i \bmod 4], i + 2), 6)$ 
8    $T[3] \leftarrow \text{RotL}(T[3] \boxplus \text{RotL}(\delta[i \bmod 4], i + 3), 11)$ 
9    $RK_{23-i}^{\text{dec}} \leftarrow T[0] \parallel T[1] \parallel T[2] \parallel T[1] \parallel T[3] \parallel T[1]$                                 //  $RK_i^{\text{dec}} \in \{0, 1\}^{196=32*6}$ 
10 end
11 return  $\{RK_i^{\text{dec}}\}_{i=0}^{23}$ 

```

1.4 Encryption of LEA-128

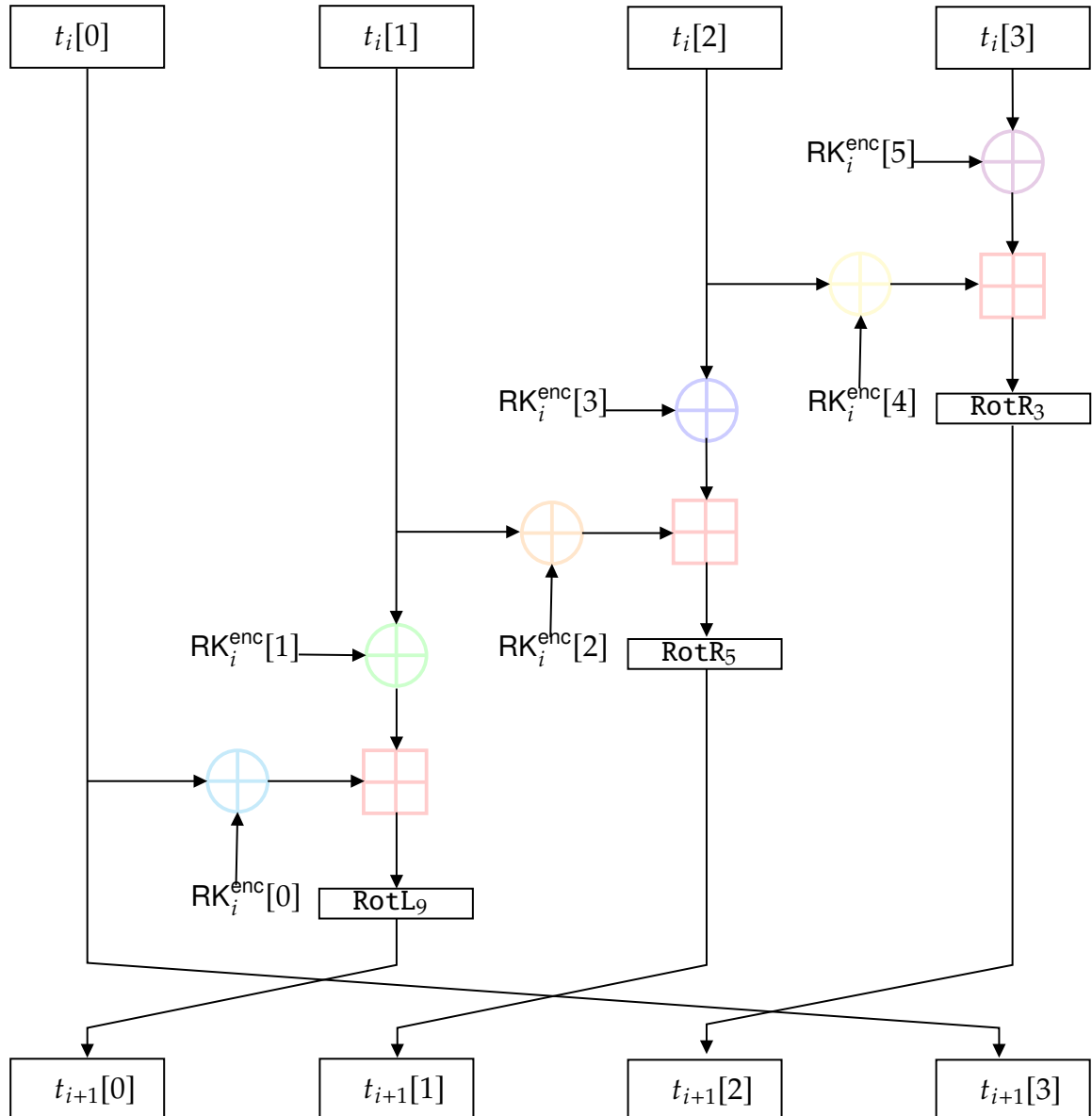
Algorithm 4: Encryption of LEA-128

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

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

```

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



1.5 Decryption of LEA-128

Algorithm 5: Encryption of LEA-128

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

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

```

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

```

Algorithm 6: Decryption of LEA-128

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

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

```

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

```

Chapter 2

Modes of Operation

Table 2.1: Comparison of Modes

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

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 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 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 dd dd 2e 49 1b c1 aa 06

2.1.1 PKCS#7

```
1 void PKCS7_PAD(u32* block, size_t block_len, size_t input_len) {  
2     if (block_len < input_len) {  
3         fprintf(stderr,  
4             "Block length must be greater than input length.\n");  
5         return;  
6     }  
7  
8     u8 padding_value = block_len - input_len;  
9     for (size_t i = input_len; i < block_len; ++i) {  
10         block[i] = padding_value;  
11     }  
12 }
```

2.2 ECB (Electronic CodeBook)

Algorithm 7: Electronic CodeBook

<p>Input: K and $P = P_1 \parallel \dots \parallel P_N$ ($P_i \in \{\mathbf{0}, \mathbf{1}\}^n$)</p> <p>Output: $C = C_1 \parallel \dots \parallel C_N$ ($C_i \in \{\mathbf{0}, \mathbf{1}\}^n$)</p> <pre> 1 for $i \leftarrow 1$ to N do 2 $C_i \leftarrow \text{EncryptBlk}(K, P_i);$ 3 end 4 return $C = C_1 \parallel \dots \parallel C_N;$ </pre>	<p>Input: K and $C = C_1 \parallel \dots \parallel C_N$ ($C_i \in \{\mathbf{0}, \mathbf{1}\}^n$)</p> <p>Output: $P = P_1 \parallel \dots \parallel P_N$ ($P_i \in \{\mathbf{0}, \mathbf{1}\}^n$)</p> <pre> 1 for $i \leftarrow 1$ to N do 2 $P_i \leftarrow \text{DecryptBlk}(K, C_i);$ 3 end 4 return $C = C_1 \parallel \dots \parallel C_N;$ </pre>
--	--

2.3 CBC (Cipher Block Chaining)

Algorithm 8: Cipher Block Chaining

Input: K and $P = P_1 \parallel \dots \parallel P_N$ ($P_i \in \{0, 1\}^{128}$)

Output: $C = C_1 \parallel \dots \parallel C_N$ ($C_i \in \{0, 1\}^{128}$)

```

1  $C_0 \leftarrow IV$ ;
2 for  $i \leftarrow 1$  to  $N$  do
3    $C_i \leftarrow \text{EncryptBlk}(K, P_i \oplus C_{i-1})$ ;
4 end
5 return  $C = C_1 \parallel \dots \parallel C_N$ ;

```

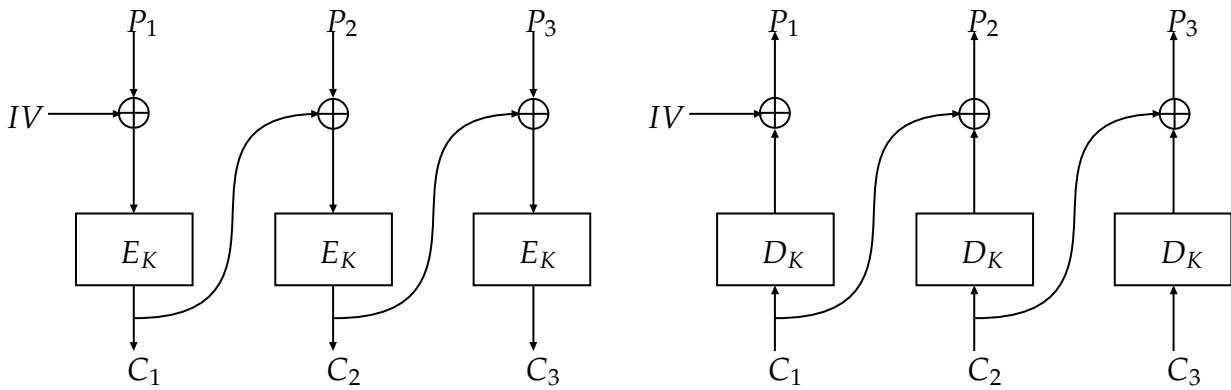
Input: K and $C = C_1 \parallel \dots \parallel C_N$

Output: $P = P_1 \parallel \dots \parallel P_N$

```

1  $C_0 \leftarrow IV$ ;
2 for  $i \leftarrow 1$  to  $N$  do
3    $P_i \leftarrow C_{i-1} \oplus \text{DecryptBlk}(K, C_i)$ ;
4 end
5 return  $P = P_1 \parallel \dots \parallel P_N$ ;

```



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
```

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

3.1.1 freeCryptoData()

```
1 void freeCryptoData(CryptoData* cryptoData) {
2     if (cryptoData != NULL) {
3         // Free the dynamically allocated memory for pt and ct
4         free(cryptoData->pt);
5         free(cryptoData->ct);
6
7         // Set the pointers to NULL to avoid dangling pointers
8         cryptoData->pt = NULL;
9         cryptoData->ct = NULL;
10
11        // Reset the lengths to zero
12        cryptoData->ptLength = 0;
13        cryptoData->ctLength = 0;
14    }
15 }
```

3.1.2 parseHexLine()

```
1 void parseHexLine(u32* arr, const char* line) {
2     for (int i = 0; i < 4; i++) {
3         u32 value;
4         sscanf(line + i * 8, "%8x", &value);
5         *(arr + i) = value;
6     }
7 }
```

3.1.3 parseHexLineVariable()

```
1 void parseHexLineVariable(u32* arr, const char* line, size_t
length) {
2     for (size_t i = 0; i < length; i++) {
3         u32 value;
4         // Ensure not to read beyond the line's end
5         if (sscanf(line + i * 8, "%8x", &value) != 1) {
6             // Handle parsing error, such as setting a default
              value or logging an error
7             arr[i] = 0; // Example: set to zero if parsing fails
8         } else {
9             arr[i] = value;
10        }
11    }
12 }
```


3.1.4 determineLength()

```
1 size_t determineLength(const char* hexString) {  
2     // Calculate the length of the hexadecimal string  
3     size_t hexLength = strlen(hexString);  
4  
5     // Convert hex length to the length of the u32 array  
6     size_t u32Length = hexLength / 8;  
7  
8     // If the hex string length is not a multiple of 8, add an  
9     // extra element  
10    if (hexLength % 8 != 0) {  
11        u32Length++;  
12    }  
13    return u32Length;  
14 }
```

3.1.5 readCryptoData()

```

1  int readCryptoData(FILE* fp, CryptoData* cryptoData) {
2      // Assuming each line will not exceed this length
3      char line[INITIAL_BUF_SIZE];
4
5      while (fgets(line, sizeof(line), fp) != NULL) {
6          if (strncmp(line, "KEY =", 5) == 0) {
7              // Skip "KEY = "
8              parseHexLine(cryptoData->key, line + 6);
9          } else if (strncmp(line, "IV =", 4) == 0) {
10             // Skip "IV = "
11             parseHexLine(cryptoData->iv, line + 5);
12          } else if (strncmp(line, "PT =", 4) == 0) {
13             // Calculate length
14             cryptoData->ptLength = determineLength(line + 5);
15             cryptoData->pt = (u32*)malloc(cryptoData->ptLength *
16                 sizeof(u32));
17             if (cryptoData->pt == NULL) return -1;
18             parseHexLineVariable(cryptoData->pt, line + 5,
19                 cryptoData->ptLength);
20          } else if (strncmp(line, "CT =", 4) == 0) {
21             // Calculate length
22             cryptoData->ctLength = determineLength(line + 5);
23             cryptoData->ct = (u32*)malloc(cryptoData->ctLength *
24                 sizeof(u32));
25             if (cryptoData->ct == NULL) {
26                 // Free pt if ct allocation fails
27                 free(cryptoData->pt);
28                 return -1;
29             }
30             parseHexLineVariable(cryptoData->ct, line + 5,
31                 cryptoData->ctLength);
32         }
33         // Add more conditions here if there are more data types
34     }
35
36     return 0; // Return 0 on successful read
37 }

```

3.1.6 compareCryptoData()

```
1  bool compareCryptoData(const CryptoData* data1, const CryptoData*
    data2) {
2      // Compare fixed-size arrays: key and iv
3      for (int i = 0; i < 4; i++) {
4          if (data1->key[i] != data2->key[i] || data1->iv[i] !=
            data2->iv[i]) {
5              return 0; // Not equal
6          }
7      }
8
9      // Compare lengths and contents of pt
10     if (data1->ptLength != data2->ptLength) {
11         return 0; // Not equal
12     }
13     for (size_t i = 0; i < data1->ptLength; i++) {
14         if (data1->pt[i] != data2->pt[i]) {
15             return 0; // Not equal
16         }
17     }
18
19     // Compare lengths and contents of ct
20     if (data1->ctLength != data2->ctLength) {
21         return 0; // Not equal
22     }
23     for (size_t i = 0; i < data1->ctLength; i++) {
24         if (data1->ct[i] != data2->ct[i]) {
25             return 0; // Not equal
26         }
27     }
28
29     return 1; // All comparisons passed, data structures are equal
30 }
```

3.2 Known Answer Test (KAT)

3.2.1 Overview

```

1 void MOVS_LEA128CBC_KAT_TEST() {
2     const char* folderPath = "../LEA128(CBC)MOVS/";
3     char txtFileName[50]; char reqFileName[50];
4     char faxFileName[50]; char rspFileName[50];
5
6     // Construct full paths for input and output files
7     snprintf(txtFileName, sizeof(txtFileName), "%s%s", folderPath,
8         "LEA128(CBC)KAT.txt");
9     snprintf(reqFileName, sizeof(reqFileName), "%s%s", folderPath,
10        "LEA128(CBC)KAT.req");
11    snprintf(faxFileName, sizeof(faxFileName), "%s%s", folderPath,
12        "LEA128(CBC)KAT.fax");
13    snprintf(rspFileName, sizeof(rspFileName), "%s%s", folderPath,
14        "LEA128(CBC)KAT.rsp");
15
16    create_LEA128CBC_KAT_ReqFile(txtFileName, reqFileName);
17    create_LEA128CBC_KAT_FaxFile(txtFileName, faxFileName);
18    create_LEA128CBC_KAT_RspFile(reqFileName, rspFileName);
19
20    printf("\nLEA128-CBC-KAT-TEST:\n");
21    FILE* file1 = fopen(faxFileName, "r");
22    FILE* file2 = fopen(rspFileName, "r");
23    if (!file1 || !file2) {
24        perror("Error opening files"); return;
25    }
26
27    CryptoData data1, data2;
28    memset(&data1, 0, sizeof(CryptoData));
29    memset(&data2, 0, sizeof(CryptoData));
30    int result = 1; // Default to pass
31    int idx = 1;
32    int totalTests = 275; // Assuming a total of 275 tests
33    int passedTests = 0;
34    while (idx <= totalTests) {
35        if (readCryptoData(file1, &data1) == -1 || readCryptoData(
36            file2, &data2) == -1) {
37            result = 0; // Indicate failure if read fails
38            break;
39        }
40
41        if (!compareCryptoData(&data1, &data2)) {
42            result = 0; // Fail
43            printf("\nFAIL\n");
44            break;
45        }
46    }

```

```

41
42     // Free the dynamically allocated memory
43     freeCryptoData(&data1);
44     freeCryptoData(&data2);
45
46     // Reset the structures for the next iteration
47     memset(&data1, 0, sizeof(CryptoData));
48     memset(&data2, 0, sizeof(CryptoData));
49
50     passedTests++;
51     printProgressBar(idx++, totalTests);
52 }
53
54 printf("\n\nTesting Summary:\n");
55 printf("Passed: %d/%d\n", passedTests, totalTests);
56 if (result) {
57     printf("Perfect PASS !!!\n");
58 } else {
59     printf("Some tests FAILED.\n");
60 }
61
62 fclose(file1);
63 fclose(file2);
64 }

```

3.2.2 create_LEA128CBC_KAT_ReqFile()

```

1 void create_LEA128CBC_KAT_ReqFile(const char* inputFileName, const
  char* outputFileName) {
2     FILE *infile, *reqFile;
3     char* line;
4     size_t bufsize = INITIAL_BUF_SIZE;
5     // Flag to check if it's the first KEY line
6     int isFirstKey = 1;
7
8     // Open the source text file for reading
9     infile = fopen(inputFileName, "r");
10    if (infile == NULL) {
11        perror("Error opening input file");
12        return;
13    }
14
15    // Open the .req file for writing
16    reqFile = fopen(outputFileName, "w");
17    if (reqFile == NULL) {
18        perror("Error opening .req file");
19        fclose(infile);
20        return;
21    }

```

```

22
23 // Allocate initial buffer
24 line = (char*)malloc(bufsize * sizeof(char));
25 if (line == NULL) {
26     perror("Unable to allocate memory");
27     fclose(infile);
28     fclose(reqFile);
29     return;
30 }
31
32 // Read the source file line by line
33 while (fgets(line, bufsize, infile) != NULL) {
34     if (strncmp(line, "KEY", 3) == 0) {
35         if (!isFirstKey) {
36             // If not the first KEY, add a newline before
37             // writing the line
38             fputc('\n', reqFile);
39         }
40         isFirstKey = 0;
41         fputs(line, reqFile);
42     } else if (strncmp(line, "IV", 2) == 0 || strncmp(line, "
43     PT", 2) == 0) {
44         fputs(line, reqFile);
45     }
46 }
47
48 // Free the allocated line buffer and close files
49 free(line);
50 fclose(infile);
51 fclose(reqFile);
52
53 printf("LEA128(CBC)KAT.req file has been successfully created
54     in 'LEA128(CBC)MOVS' folder.\n");
55 }

```

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