UNIT 4 SYMMETRIC KEY BLOCK CIPHERS

Structure **Page No.**

4.1 INTRODUCTION

In the first block, we had discussed some simple cryptosystems. In those cryptosystems, we encrypt and decrypt using pencil and paper. The cryptosystems were used mainly for military and diplomatic purposes. However, in the 1960s it was possible to connect computers in a network and communicate information from one computer to another through these networks. It also became possible to access computers from a remote network. Computers were also used for storing a lot of confidential and sensitive information. It became necessary to store and communicate information securely so that unauthorised persons had no access to them. For this, most of the encryption methods used earlier became useless. Methods for handling the challenges arising from the developments in the communication technology had to be evolved.

In response to a call for a cryptographic algorithm in 1973 from National Bureau of Standards(NBS), a US government agency, IBM submitted an algorithm called LUCIFER in 1974. The National Security Agency made some modifications and the modified algorithm, which is essentially the Data Encryption Standard(DES), was released by NBS with a free license in 1975. NBS made the algorithm the official standard for data encryption in 1977. The DES is an example of a **block cipher**. In Sec. 4.2, we will discuss block ciphers in general and the principles behind their design. We will discuss the DES in Sec. 4.3 of this unit.

NBS was renamed as National Institute of Standards and Technology(NIST) later.

DES remained the standard cryptographic system in 80s and 90s, but due to improvements in technology it became vulnerable. We discuss the weaknesses that were discovered in Sec. 4.3. In Sec. 4.4, we will discuss the Advanced Encryption Standard algorithm(AES) that was developed to replace the DES algorithm.

Objectives

After studying this unit, you should be able to

- define a block cipher;
- define a stream cipher;
- explain the general design principles behind block ciphers as enunciated by Claude Shannon;
- encrypt and decrypt using a toy block cipher;
- explain how DES encrypts and decrypts messages;
- explain the various modes of operation;

- explain why DES had to be replaced; and
- explain how the AES encrypts and decrypts messages.

4.2 BLOCK CIPHERS

The ciphers are broadly classified under two different types, the **block ciphers** and **stream ciphers**. Let us now formally define what they are.

Definition 1: A **block cipher** breaks a message P into blocks $P_1P_2\ldots$ of some fixed length n and encrypts each block P_i with the same key $k \in \mathcal{K}$, that is

$$E_k(P) = E_k(P_1)E_k(P_2)\ldots$$

A **stream cipher** breaks a message P into characters or bits $p_1p_2\ldots$ and encrypts with a **key stream** $k_1k_2\ldots$, that is

$$E_k(P) = E_{k_1}(p_1)E_{k_2}(p_2)\ldots$$

A **synchronous stream cipher** is one in which we generate the key stream independent of the plain text.

A stream cipher is **periodic** of period d if the stream repeats after d characters.

The Vigenere cipher is an example of a synchronous, periodic stream cipher. This is because, if $c_1c_2\cdots c_n$ is the key, we use the key stream $k_1k_2\cdots$ where

$$k_i = \begin{cases} c_i & \text{if } 1 \le i \le n \\ c_m & 1 \le m \le n, i \equiv m \bmod n \end{cases}$$

In the previous block we discussed some simple substitution and transposition ciphers. They are not strong due to two attributes of the plain text. One is the **redundancy** in the plain text. For example, in English, the letter 'Q' is almost always followed by the letter 'U'. The redundancy is preserved by substitution ciphers. So, while decrypting a text that was encrypted with a substitution cipher, if a character decrypts to Q, then it is almost sure that the next character decrypts to U. However, this redundancy is destroyed by transposition ciphers.

Another attribute is the statistical characteristics of the plain text. For example, we know that the character that appears most frequently is the letter 'E'. The statistical characteristics of the text are preserved by a transposition cipher, but destroyed by a substitution cipher.

In his immensely influential paper, [20], Claude Shannon presented the principles of **confusion** and **diffusion**. Here, confusion is to make the relation between the key and the cipher text as complex as possible so that the redundancy is reduced; diffusion is to spread the influence of individual plain text characters over as much of the cipher text as possible so that the statistical properties of the plain text are hidden.

So, while designing a block cipher, we have to use the two principles so that the cipher is hard to break. In the same paper, Shannon also suggests a method for achieving confusion and diffusion, namely using a mixture of some simple transformations like transpositions and substitutions. We can get a **product cipher** by mixing different, suitable, combinations of some simpler ciphers.

Feistel was a part of the team at IBM which designed the LUCIFER cipher.

In the **iterated block ciphers**, we usually use a complex **round function** repeatedly, using the input from the previous round as the output for the next round. One such function is derived from the **Feistel cipher**. We will now present a simplified round function which is quite similar to the Feistel cipher. The design of many of the block

ciphers, including DES, are similar to the Feistel cipher. The discussion is based on [21].

Since using the DES manually is laborious and cumbersome—remember, it was meant to be used on computers as opposed to the other classical ciphers like the Vigenere cipher in which encryption was done by hand—we can't use it in our examples. To help you understand the basic ideas behind a block cipher, we will discuss a simpler cipher discussed in Sec. 4.2 of [21]. This cipher is similar to the DES, but much smaller. It has an S-box like many real world ciphers. We will call it 'Toy block cipher' since it can't be used in practice and yet we can understand how product block ciphers work using this. In this cipher, the message we have to encrypt consists of a single block of 12 bits and our key is nine bits long.

We will now discuss how to carry out one round of encryption using this block cipher with the help of an example. Suppose the message we want to encrypt is $m = 101011001101$ and the key is $k = 010100110$.

Step 1 We write the message in the form L_0R_0 where L_0 consists of the first six bits and R_0 consists of the remaining six bits. In our case $L_0 = 101011$ and $R_0 = 001101$.

Step 2 We obtain an eight-bit key k_1 from the nine-bit key for this round as follows: We use the first eight digits in the key from the left. Here, $k_1 = 01010011$. In general, in the i^{th} round of encryption, we use the eight digits of the key starting from i^{th} digit. If we reach the end of the nine digit key, we 'wrap around' and continue from the first digit. For example, in the third round, we use the seven digits of starting from the third digit as the first seven digits of k_3 and use the first digit of k as the last digit of k_3. For our key k, the key for the third round encryption is $k_3 = 01001100$.

We use a function $f\colon \{0,1\}^6 \times \{0,1\}^8 \to \{0,1\}^6$ to get a six bit output from (R_0, k_1).

Let us see how this function works. First, it expands the given six bit number to eight bits as follows:

- The first and the second bits of the output are the same as the first and second bits of the input, respectively.
- The third bit and the fifth bit of the output are the same as the fourth bit of the input.
- The fourth and sixth bit of the output are, respectively, the same as the third bit of the input.
- The seventh and the eighth bit of the output are, respectively, the fifth and sixth bit of the output.

To summarise, if the six-bit input is $a_1a_2a_3a_4a_5a_6$, the eight-bit output is $a_1a_2a_4a_3a_4a_3a_5a_6$. In this example, $R_0 = 101011$, so $a_1 = 1$, $a_2 = 0$, $a_3 = 1$, $a_4 = 0$, $a_5 = 1$ and $a_6 = 1$. So, from $R_0 = 101011$, we get the output 10010111

We now XOR the eight bit output with the key k_1 for the first round. We have

Recall the exclusive OR operator ^ in the C language from the programming course.

$$\begin{array}{r} 10010111 \\ 01010011 \\ \hline 11000100 \\ \hline \end{array}$$

Note that XORing is just bit by bit addition modulo two of the digits of the two numbers.

Next, we split the eight-bit output into two halves and use the **S-boxes** to get two numbers of three bits each, one from each S-box. Let us see how to do this. Splitting 11000100 into two halves, we get 1100 and 0100. We use the

following two S-boxes:

$$S_1 \begin{bmatrix} 101 & 010 & 001 & 110 & 011 & 100 & 111 & 000 \\ 001 & 100 & 110 & 010 & 000 & 111 & 101 & 011 \end{bmatrix}$$

$$S_2 \begin{bmatrix} 100 & 000 & 110 & 101 & 111 & 001 & 011 & 010 \\ 101 & 011 & 000 & 111 & 110 & 010 & 001 & 100 \end{bmatrix}$$

Let us find the three-bit output corresponding to the left half 1100. For this we use the first S-box, namely S_1. Here, the first digit is 1, so we use the second row of S_1. The next three digits are 100 whose decimal value is four. We choose the entry in the fifth column in the second row of S_1 which is 000. In general, if the decimal value of the last three digits is k, we choose the entry in the $k+1$st column. Note that $k \leq 7$ and the number of columns in the S-boxes is eight.

Similarly, for the right half 0100 we use S_2 and choose the entry in the first row, fifth column of S_2 since the first digit is 0 and the next three digits have decimal value 4. The three bit number we get in this case is 111.

Concatenating the two outputs, we get 000111. This is our $f(R_0, k_1)$.

Step 3 Next, we XOR the $f(R_0, k_1)$ with L_0. We get

$$\begin{array}{r} 000111 \\ 101011 \\ \hline 101100 \end{array}$$

Step 4 The answer 101100 is our R_1 and we take R_0 as L_1, i.e. the output is 001101101100. This is the input for the second round and we use the steps 1 to 4 to get L_2R_2 the output for the second round. We can carry out the procedure many times, i.e. carry out many rounds of encryption.

Let us now discuss the method of encryption in general terms. We assume that our key k is nine bits long. In the ith round of the algorithm, we transform an input $L_{i-1}R_{i-1}$ to the output L_iR_i using a key k_i of size eight bits that is derived from the key k.

We use a function $f(R_{i-1}, k_i)$ that takes a six-bit input R_{i-1} and an eight-bit input k_i and produces an output of size six. As we saw, the function is actually a composition of many functions. The first function is the **expander** function. It takes an input of size six bits and 'expands' it to eight bits. Given a six-bit input $a_1a_2a_3a_4a_5a_6$ it outputs the eight-bit number $a_1a_2a_4a_3a_4a_3a_5a_6$.

Now, the key used for encryption enters the picture. The key k consists of nine bits. The k_i for the i^{th} round of encryption is obtained by using eight bits of k, starting with i^{th} bit. If $k = 001100111$, $k_5 = 00111001$. Here, after four bits, we reach the end of the key, so we 'wrap around', start from the beginning and use the first four bits of k.

We then XOR the eight-bit output from the expander part with the key to get another eight-bit number. We split the eight-bit output we get into two halves. We use the left half as the input for the first S-box. Of the four bits, the first bit specifies the row. If the first bit is zero, we choose the first row, if the first digit is one we choose the second row. If the decimal value of the last three bits is k, we choose $k+1$th column. For example, if the last three bits are 011, we choose the fourth column since the decimal value of 011 is three. Similarly, we get the output for the second half using the second S-box. We concatenate the two outputs to get the 6-bit output. We then XOR the six-bit output with L_{i-1} to get R_i.

We define the output for the i^{th} round by $L_i = R_{i-1}$, $R_i = L_{i-1} \oplus f(R_{i-1}, k_i)$. Here $\oplus$ denotes XOR operation. We have shown the encryption operation in Fig. 1 on the facing page. We simply apply this operation a specified number of times, say n to get the output L_nR_n.

Let us now look at an example to understand this.

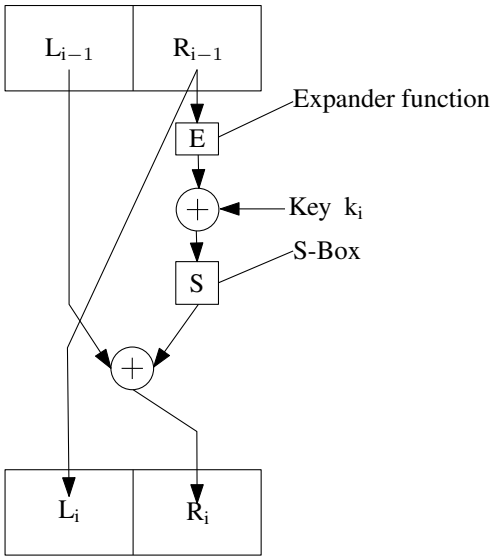


Fig. 1: One round of the encryption procedure.

Example 1: Suppose the key is 110101110 and the input is 111000101011. Carry out two rounds of encryption by using the function f.

Solution:

Step 1 We split the 12 bit input into two halves, R_0 and L_0. Here $L_0 = 111000$ and $R_0 = 101011$.

Step 2 If $a_1a_2a_3a_4a_5a_6 = 101011$, by expanding, we get the number $a_1a_2a_4a_3a_4a_3a_5a_6 = 10010111$.

Step 3 The key for the first round is simply the first eight digits of the key, which is 11010111. So, $k_1 = 11010111$. We XOR the eight-bit number we got by expanding R_0, viz. 10010111, with k_1 to get 01000000. We divide the output into two parts and use the S-boxes. From the left half 0100, we get 011 from the first S-box. From the right half 0000, we get 100 from the second S-box. Concatenating, we get the six-bit number $f(R_0, k_1) = 011100$.

Step 4 To get R_1, we XOR L_0 with $f(R_0, k_1)$:

$$R_1 = L_0 \oplus f(R_0, k_1) = 111000 \oplus 011100 = 100100$$

Since we take $L_1 = R_0$, the output L_1R_1 is 101011100100.

For the second round, we start from the second digit and read off eight-digits to get $k_2 = 10101110$. We leave the remaining details as an exercise to you.

* * *

Let us now summarise the procedure for encryption:

Encryption using Toy Cipher in the ith round

1) Split the 12 bit input into left and right halves, L_i, R_i of size six each. Expand the right half to eight bits according to the rule $a_1a_2a_3a_4a_5a_6 \rightarrow a_1a_2a_4a_3a_4a_3a_5a_6$ where a_i are the bits in R_i.
2) After expanding, XOR the result with the key for the ith round.
3) After XORing split the result into two halves of four bits each. Use the S-boxes on each of the two halves to get two strings of length three each and concatenate them to get a six bit string.
4) Get R_{i+1} by XORing the six bit string obtained in the last step with L_i. $L_{i+1} = R_i$.

Here are some exercises for you.

E1) Complete Example 1 on the previous page by carrying out one more round of encryption using the Toy block cipher.

E2) If the 9 bit key for the Toy block cipher is 011001110 what are the 8-bit keys for the second and fourth rounds of encryption?

We now describe the decryption procedure. Given L_nR_n, we first switch L_n and R_n and obtain R_nL_n. We now use the procedure that we used for encryption, but we use the keys in reverse order, i.e. k_n, k_{n-1}, …, k_1. In the first step, from the input R_nL_n, we get the output $[L_n]\,[R_n \oplus f(L_n, k_n)]$.

From the way the encryption is carried out, we know that

$$L_n = R_{n-1} \text{ and } R_n = L_{n-1} \oplus f(R_{n-1}, k_n).$$

Since $L_n = R_{n-1}$, we have

$$f(R_{n-1}, k_n) \oplus f(L_n, k_n) = f(R_{n-1}, k_n) \oplus f(R_{n-1}, k_n) = 0.$$

Therefore, $R_n \oplus f(L_n, k_n) = L_{n-1} \oplus f(R_{n-1}, k_n) \oplus f(L_n, k_n) = L_{n-1}$. So, under the procedure $[L_n]\,[R_n \oplus f(L_n, k_n)]$ is decrypted as $[R_{n-1}]\,[L_{n-1}]$. Similarly, in the second step $R_{n-1}L_{n-1}$ decrypts to $R_{n-2}L_{n-2}$. Proceeding this way, we get R_0L_0. Switching the left and right halves, we get back the plain text L_0R_0. This is summarised in the box below:

Decryption using the Toy cipher

1) Swap the right and the left halves.
2) Carry out the procedure used for encryption n times using the keys in the reverse order: k_n, k_{n-1}, …, k_1.
3) Swap the right and the left halves.

Let us look at an example to understand the decryption procedure.

Example 2: The text 100100000000 was got by encrypting twice using the key 110101110. Decrypt it using the same key.

Solution:

1) We swap the right and the left halves to get 000000100100.
2) We carry out two rounds of the same process we used for encryption, using the keys in the reverse order.
   First round: The key is $k_2 = 10101110$. Expanding the right half 100100, we get 10101000. XORing with the k_2, we get 00000110. Using the S-boxes, we get 101011. XORing with the left half 000000, we get 101011. Output for the first round is 100100101011.
   Second Round: The key is $k_1 = 11010111$. Expanding 101011, we get 10010111. XORing with the key k_1, we get 01000000. Using the S-boxes, we get 011100. We XOR this with 100100, to get 111000. So, the output of the second round is 101011111000.
3) We swap the left and right halves to get 111000101011. It is worth noting that this is the plain text we started with in Example 1.

* * *

So, for decryption, we use a procedure similar to the encryption procedure. We interchange the left and right halves and use the keys k_i in the reverse order. Since the procedures for encryption and decryption are the same, the sender and receiver use the

same machines. Of course, the receiver has to reverse the left and right inputs. Here is an exercise for you to check your understanding of the decryption process.

E3) Decrypt the text 001011011001 that was encrypted once with the Toy block cipher using the key 010100110.

We conclude this section here. In the next section, we will discuss the DES algorithm.

4.3 THE DATA ENCRYPTION STANDARD

The DES was one of the most popular ciphers for 70s, 80s and early 90s. In a survey of block ciphers [15], written in 1994 , Robshaw remarked "DES is still secure, but it is now at the end of its useful life. It will, however, continue to be used in some other mode, perhaps in some form of triple encryption until an alternative block cipher can be found." It is also one of the most extensively researched ciphers. It has survived many attempts to break it. When it finally become obsolete, it was not because of the cryptanalytic techniques, but because of advances in technology that made it possible to crack the DES by brute force. Because of this, the design principles behind the DES are used in many other modern block ciphers. In this section we will describe the cipher. The discussion in this section is based on Sec. 4.4 and Sec. 4.5 of [21].

In DES, a block of cipher text is 64-bit long and the actual key is 56-bit long. It is padded with 8 more bits to get a 64 bit key by adding parity bits at the 8th, 16th, 24th, …. We choose the parity bits in such a way that each block of 8 bits has an odd number of ones. For example, if there are three 1s and four 0s in the first seven bits, we choose 0 as the eighth bit; if there are, for example, four 1s and three 0s in the first seven bits, we choose 1 as the eighth bit. The parity bits are used for detecting errors.

Suppose $\mathcal{M}$ is a plain text of 64 bits. We encrypt using the DES algorithm as follows:

Initial permutation We apply an initial fixed permutation IP(i.e. IP is independent of $\mathcal{M}$ and the key k) to $\mathcal{M}$ and obtain $\mathcal{M}_0 = \text{IP}(\mathcal{M})$. As we did in the Toy cipher, we divide this into two parts, L_0R_0, each 32 bits long.

Apply round function For $1 \leq i \leq 16$, we do the following:

$$L_i = R_{i-1} \text{ and } R_i = L_{i-1} \oplus f(R_{i-1}, k_i),$$

where k_i is a string of length 64 that we obtain from the key k. We will describe f and the selection of k_i in detail later.

Switch and permute We switch the left and right halves to get $R_{16}L_{16}$, then apply the inverse of the initial permutation to get the cipher text $c = \text{IP}^{-1}(R_{16}L_{16})$.

Note the similarity between this encryption procedure and the encryption procedure in the Toy block cipher.

As we did in the Toy Block Cipher, we decrypt by using the same procedure we use for encryption, but the keys are used in reverse order, k_{16}, k_{15}, …, k_2, k_1. Because of the left-right switch performed in the third step, we need not switch the left-right switch during decryption.

The initial permutation is given in Table 1. According to this table, 58th bit of $\mathcal{M}$ is the first bit of $\mathcal{M}_0$, 50th bit of $\mathcal{M}$ is the second bit of $\mathcal{M}_0$, etc.

Table 1: Initial Permutation

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

We now describe the function $f(R, k_i)$.

Expansion We expand R to E(R) according to Table 2.

Table 2: Expansion permutation

32	1	2	3	4	5	4	5	6	7	8	9
8	9	10	11	12	13	12	13	14	15	16	17
16	17	18	19	20	21	20	21	22	23	24	25
24	25	26	27	28	29	28	29	30	31	32	1

This means that the 32nd bit of R is the first bit of E(R), the first bit of R is the second bit of E(R), etc.

XOR with the key We compute $E(R) \oplus k_i$, which has 48 bits, and write it as $B_1B_2 \ldots B_8$ where each B_j is 6 bits long and is the input for S-box S_j.

S-boxes We use the eight S-boxes in Table 3 on the facing page, the first S-box for B_1, the second S-box for B_2, etc. Writing $B_j = b_1b_2 \cdots b_6$, the row of the S-box is specified by b_1b_6 while $b_2b_3b_4b_5$ determines the column. For example, if $B_2 = 101110$, $b_1b_6 = 10$ which is two in base 10. So, we add one to two and choose the third row. Also, $b_2b_3b_4b_5 = 0111$ is seven in base 10. So, we add one to seven and choose the eigth column. The entry in this location in the second S-box is one which is 0001 in binary. In this way, we get eight four-bit outputs $C_1C_2 \cdots C_8$.

Final permutation We permute the bits in the 32-bit number $C_1C_2 \cdots C_8$ according to Table 4. The resulting 32-bit string is $f(R, k_j)$.

Table 4: Final permutation.

16	7	20	21	29	12	28	17	1	15	23	26	5	18	31	10
2	8	24	14	32	27	3	9	19	13	30	6	22	11	4	25

It remains to explain how to get the keys k_1,k_2, …, k_{16} from the 64 bit key k.

1) We discard the parity bits the permute the remaining bits according to Table 5. Let us write the output as $k_0^1k_0^2$ where k_0^1 and k_0^2 have 28 bits each.

Table 5: Key permutation.

57	49	41	33	25	17	9	1	58	50	42	34	26	18
10	2	59	51	43	35	27	19	11	3	60	52	44	36
63	55	47	39	31	23	15	7	62	54	46	38	30	22
14	6	61	53	45	37	29	21	13	5	28	20	12	4

2) For $1 \leq i \leq 16$, in the ith round, we get k_i^1 and k_i^2 by shifting the bits in k_{i-1}^1 and k_{i-1}^2 to the left circularly by the number of bits in Table 6. See the appendix at the end of this unit for a discussion of left and right circular shifts.

Table 6: Key shifts

Round	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Shift	1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

For example, in the third round, according to the table, we have to shift the bits by two positions. So, the first bit becomes the 27th bit, the second bit becomes the 28th bit and all the bits in positions three to 28th are shifted to the left by two positions. The third bit becomes the first bit, the fourth bit becomes the second bit etc.

Table 3: S-boxes

S-box 1															
14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13
S-box 2															
15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9
S-box 3															
10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12
S-box 4															
7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14
S-box 5															
2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3
S-box 6															
12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13
S-box 7															
4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12
S-box 8															
13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

3) We choose the 48 bits of the key k_i from $k_i^1 k_i^2$ according to Table 7.

Let us now look at an example of encryption by DES. This example is taken from `http://orlingrabbe.com/des.htm`.

Example 3: Let us take the message

$$\mathcal{M} = 0000\ 0001\ 0010\ 0011\ 0100\ 0101\ 0110\ 0111\ 1000\ 1001\ 1010\ 1011\ 1100\ 1101\ 1110\ 1111$$

which is 64 bits long. Let the 64 bit key be

$$00010011\ 00110100\ 01010111\ 01111001\ 10011011\ 10111100\ 11011111\ 11110001$$

Table 7: Key selection

14	17	11	24	1	5	3	28	15	6	21	10
23	19	12	4	26	8	16	7	27	20	13	2
41	52	31	37	47	55	30	40	51	45	33	48
44	49	39	56	34	53	46	42	50	36	29	32

After applying the initial permutation to $\mathcal{M}$, we get

$$\mathcal{M}_0 = 1100\ 1100\ 0000\ 0000\ 1100\ 1100\ 1111\ 1111$$
$$1111\ 0000\ 1010\ 1010\ 1111\ 0000\ 1010\ 1010.$$

From this we get

$$L_0 = 11001100000000001100110011111111 \quad (1)$$
$$R_0 = 11110000101010101111000010101010 \quad (2)$$

Using the key permutation in Table 5, we get the following 56-bit key.

$$11110000110011001010101011111010101010110011001111100011111$$

Dividing this into two halves, we get

$$k_0^1 = 1111000\ 0110011\ 0010101\ 0101111$$
$$k_0^2 = 0101010\ 1011001\ 1001111\ 0001111$$

Here are the value of k_0^1, k_0^2, k_1^1, k_1^2, …, k_{16}^1, k_{16}^2. Concatenating the values k_i^1 and k_i^2 in

Table 8: The values of k_i^1 and k_i^2.

i	k_i^1	k_i^2
0	11110000110011001010101111	01010101011001100111100011111
1	11100001100110010101011111	10101010110011001111000111110
2	11000011001100101010111111	01010101100110011110001111101
3	00001100110010101010111111111	01010110011001111000111101101
4	00110011001010101011111111100	01011001100111100011110101101
5	11001100101010101111111110000	01100110011110001111010101101
6	00110010101010111111111000011	10011001111000111101010101101
7	11001010101011111111100001100	01100111100011110101010101110
8	00101010101111111110000110011	10011110001111010101010110011
9	01010101011111111100001100110	00111100011110101010101100111
10	01010101111111110000110011001	11110001111010101010110011001100
11	01010111111111000011001100101	11000111101010101011001100111
12	01011111111000011001100101101	00011110101010101100110011111
13	01111111100001100110010101101	01111010101010110011001111100
14	11111110000110011001010101101	11101010101011001100111110001
15	11111000011001100101010101111	10101010101100110011111000111
16	11110000110011001010101011111	01010101011001100111100011111

Table 8 and using Table 7, we get the 48-bit keys k_1, k_2, …, k_{16} in Table 9. Let us now calculate L_1, R_1 from Eqn. (1) and Eqn. (2). We have
$L_1 = R_0 = 11110000101010101111000010101010$. We have $R_1 = L_0 \oplus f(R_0, k_1)$. Let us first calculate $f(R_0, k_1)$. Expanding R_0 according to Table 2, we get
$E(R_0) = 011110\ 100001\ 010101\ 010101\ 011110\ 100001\ 010101\ 010101$. We then XOR the output with the key k_1:

$E(R_0)$	011110 100001 010101 010101 011110 100001 010101 010101
k_1	000110 110000 001011 101111 111111 000111 000001 110010
	011000 010001 011110 111010 100001 100110 010100 100111

Table 9: Keys for 16 rounds.

k_1	000110 110000 001011 101111 111111 000111 000001 110010
k_2	011110 011010 111011 011001 110110 111100 100111 100101
k_3	010101 011111 110010 001010 010000 101100 111110 011001
k_4	011100 101010 110111 010110 110110 110011 010100 011101
k_5	011111 001110 110000 000111 111010 110101 001110 101000
k_6	011000 111010 010100 111110 010100 000111 101100 101111
k_7	111011 001000 010010 110111 111101 100001 100010 111100
k_8	111101 111000 101000 111010 110000 010011 101111 111011
k_9	111000 001101 101111 101011 111011 011110 011110 000001
k_{10}	101100 011111 001101 000111 101110 100100 011001 001111
k_{11}	001000 010101 111111 010011 110111 101101 001110 000110
k_{12}	011101 010111 000111 110101 100101 000110 011111 101001
k_{13}	100101 111100 010111 010001 111110 101011 101001 000001
k_{14}	010111 110100 001110 110111 111100 101110 011100 111010
k_{15}	101111 111001 000110 001101 001111 010011 111100 001010
k_{16}	110010 110011 110110 001011 000011 100001 011111 110101

We have

$$B_1 = 011000 \qquad B_2 = 010001 \qquad B_3 = 011110 \qquad B_4 = 111010$$
$$B_5 = 100001 \qquad B_6 = 100110 \qquad B_7 = 010100 \qquad B_8 = 100111$$

Using the S-box S_i for B_i, $1 \leq i \leq 8$, we get

$$S_1(B_1)\,S_2(B_2)\,S_3(B_3)\,S_4(B_4)\,S_5(B_5)\,S_6(B_6)\,S_7(B_7)\,S_8(B_8)$$
$$= 0101\ 1100\ 1000\ 0010\ 1011\ 0101\ 1001\ 0111$$

Next, we apply the permutation in Table 4 to
$S_1(B_1)\,S_2(B_2)\,S_3(B_3)\,S_4(B_4)\,S_5(B_5)\,S_6(B_6)\,S_7(B_7)\,S_8(B_8)$. We get

$$f(R_0, k_1) = 0010\ 0011\ 0100\ 1010\ 1010\ 1001\ 1011\ 1011$$

We have $R_1 = L_0 \oplus f(R_0, k_1)$. So,

L_0	1100 1100 0000 0000 1100 1100 1111 1111
$f(R_0, k_1)$	0010 0011 0100 1010 1010 1001 1011 1011
	1110 1111 0100 1010 0110 0101 0100 0100

Proceeding like this, we get

$$L_{16} = 01000011010000100011001000110100$$
$$R_{16} = 00001010010011001101100110010101$$

We then switch the halves apply the permutation IP^{-1} which is the inverse of the permutation IP, to $R_{16}L_{16}$ to get

$$10000101\ 11101000\ 00010011\ 01010100$$
$$00001111\ 00001010\ 10110100\ 00000101$$

* * *

This concludes our discussion of DES.

The design criteria behind DES was not clear because it was not made public. In 1992, IBM published some of the details. See [4]. We give a brief summary below:

1) Each S-box has 6 input and 4 output bits. This was the largest that could be put on one chip in 1974.
2) The relationship between in inputs and outputs of the S-boxes should not be linear. If the relationship is linear, cryptanalysis will be easier.
3) Each row of an S-box contains all numbers from 0 to 15.
4) If two inputs to an S-box differ by 1 bit, the output must differ by at least 2 bits.
5) If two inputs to an S-box differ in their first 2 bits, but have the same last 2 bits, their outputs should not be the same.
6) There are 32 pairs of input having the same XOR. Out of these, no more than 8 pairs should have the same XOR for their outputs.

4.3.1 Modes of Operation

As we saw earlier, DES is a block cipher that encrypts plain texts which are 64-bits long in one go. But, sometimes, it may become necessary to encrypt plain texts of larger or smaller sizes. We can run block ciphers in different modes according to the demands of the application of the ciphers. The following are five common modes:

1) Electronic code cook(ECB).
2) Cipher block chaining(CBC).
3) Cipher feedback(CFB).
4) Output feedback(OFB).
5) Counter(CTR)

All these modes can work with chunks of different sizes. However, in our discussion we will confine ourselves to the case where all the chunks are of 8 bits. Let us discuss these modes one by one.

Electronic code book(ECB) By the definition of a block cipher, we break up the text into blocks of fixed size. This mode of operation is called Electronic code book operation. If $P = P_1P_2\cdots P_k$ is the plain text, then $C = C_1C_2\cdots C_k$ is cipher text where $C_i = E_k(P_i)$.

There is a weakness in this mode. Once we fix a key, the relationship between the plain text and cipher text is fixed. If an adversary is able to decrypt a block of cipher text from the context or by any other means, he/she can decrypt all the occurrences of that cipher text.

Cipher Block Chaining(CBC) In this mode, the encryption of a block depends upon the encryption of the previous blocks and so removes some of the problems in the ECB mode. We use an **initialisation vector(IV)** C_0 in this mode. We encrypt as follows:

$$C_1 = E_k(P_1 \oplus C_0)$$
$$C_i = E_k(P_i \oplus C_{i-1}) \text{ for } i > 1$$

We decrypt as follows:

$$P_i = D_k(C_i) \oplus C_{i-1}$$

Again, we have to change the IVs often to keep the system secure.

Cipher Feedback(CFB): In ECB and CBC, encryption and decryption cannot take place till we receive all the 64 bits of text. In CFB we can encrypt 8 bits of plain text without waiting for all the 64 bits. We break the plain text into 8-bit pieces: $P = P_1P_2\cdots$ where each P_j has eight bits. We encrypt as follows: We choose an initial 64 bit text X_1. Then, for $j = 1,2,3,\ldots$ we encrypt the the P_js:

$$O_j = L_8\left(E_k\left(X_j\right)\right) \tag{3a}$$
$$C_j = P_j \oplus O_j \tag{3b}$$
$$X_{j+1} = R_{56}\left(X_j\right)||C_j \tag{3c}$$

Here $L_8(X)$ denotes the 8 leftmost bits of $X, R_{56}(X)$ denotes the rightmost 56 bits of X, and $X \parallel Y$ denotes the string obtained by writing X followed by Y.

We decrypt as follows:

$$P_j = C_j \oplus L_8\left(E_k\left(X_j\right)\right)$$
$$X_{j+1} = R_{56}\left(X_j\right) \parallel C_j.$$

You would have probably noticed that decryption does not involve calling the decryption function, D_k. This would be an advantage of running a block cipher in a stream mode in a case where the decryption function for the block cipher is slower than the encryption function.

Let us look at one round of encryption using the CFB mode. First, we initialise a 64-bit register with the value of X_1. We then encrypt these 64 bits using E_k. We extract the leftmost 8 bits of $E_k(X_1)$ and XOR it with the 8-bit P_1 to form C_1. We send C_1 to the recipient. We then update the 64-bit register X_1 as follows: We concatenate the rightmost 56 bits of X_1 with 8 bits of C_1 and load the resulting 64 bits into the register. We then encrypt P_2 by the same process, but use X_2 in place of X_1. After encrypting P_2 to C_2, we update the 64-bit register as follows:

A **register** is a memory location in a hardware device. In the context of cryptography, it refers to a device that is used for encryption and decryption.

$$X_3 = R_{56}(X_2) \parallel C_2 = R_{48}(X_1) \parallel C_1 \parallel C_2.$$

After 8th rounds, the initial X_1 has disappeared from the 64-bit register and $X_9 = C_1 \parallel C_2 \parallel \cdots \parallel C_8$. The C_j continue to pass through the register, so for example $X_{20} = C_{12} \parallel C_{13} \parallel \cdots \parallel C_{19}$.

Since CFB mode of transmission can recover form errors in transmission of the cipher text, this mode is useful in practice. Suppose that the transmitter sends the cipher text blocks $C_1, C_2, \ldots, C_k \ldots$, and C_1 is corrupted during transmission, so that the receiver observes $C_1', C_2, \ldots$. On decryption of C_1', the receiver gets a garbled version of P_1 with bit errors in the location that C_1 had bit errors. Since the receiver will get X_2 by concatenating the rightmost 56 bits of X_1 with C_1' and so the value of X_2 obtained also will be wrong. Note that, each time the receiver calculates X_i, the leftmost 8 bits of the register will be pushed out. So,

$$X_2 = \left(v_1, v_2, v_3, v_4, v_5, v_6, v_7, C_1'\right)$$

where v_1, v_2, ... were the values in the shift register before C_1' was received.

$$X_3 = \left(v_2, v_3, v_4, v_5, v_6, v_7, C_1', C_2\right)$$
$$X_4 = \left(v_3, v_4, v_5, v_6, v_7, C_1', C_2, C_3\right)$$
$$\vdots$$
$$X_7 = \left(v_8, C_1', C_2, C_3, \ldots, C_7\right)$$
$$X_8 = \left(C_1', C_2, C_3, \ldots, C_8\right)$$
$$X_9 = \left(C_2, C_3, \ldots, C_8, C_9\right)$$

So, the corrupted cipher is ultimately flushed out of the register and decryption proceeds correctly afterwards.

Output Feedback (OFB): In this mode errors do not propagate like CFB mode where an error in one of the encrypted chunks propagates to the next 8 chunks.

Like CFB, OFB also works on chunks of different sizes. For our discussion, we will focus on the 8-bit version of OFB, where OFB is used to encrypt 8-bit chunks of plain text in a streaming mode. The procedure in OFB is very similar to the procedure in CFB. In, for $j = 1, 2, 3, \ldots$, we encrypt as follows:

$$O_j = L_8\left(E_k\left(X_j\right)\right) \tag{4a}$$
$$X_{j+1} = R_{56}\left(X_j\right) \parallel O_j \tag{4b}$$
$$C_j = P_j \oplus O_j. \tag{4c}$$

Let us compare the methods of encryptions for CFB and OFB modes given by Eqn. (3) and Eqn. (4), respectively. The difference is in computation of X_js. From Eqn. (3b), we see that, in the CFB mode we get X_{j+1} by concatenating the rightmost 56 bits of X_j with $C_j = P_j \oplus E_k\left(X_j\right)$. In OFB mode, from Eqn. (4b), we see that, we get X_j by concatenating the rightmost 56 bits with $O_j = L_8\left(E_k\left(X_j\right)\right)$.

Let us see what is the advantage in OFB when compared with CFB. First, the generation of the O_j output key stream may be performed completely without any plain text. So, the output key stream can be computed in advance thus speeding up the decryption process in those situations where this gain in speed is important. The second advantage is that, when there are errors in some of the bits in the cipher text, this will affect only the corresponding bits in the plain text when we decrypt.

There is a problem with OFB, however, that is common to all stream ciphers that are obtained by XORing pseudo-random numbers with plain text. If an adversary knows a particular plain text P_j and cipher text C_j, he/she can modify the messages. He/She first calculates

$$O_j = C_j \oplus P_j$$

to get out the key stream. He/She may then create any false plain text P'_j he/she wants. Now, to produce a cipher text, he merely has to XOR with output stream he/she calculated:

$$C'_j = P'_j \oplus O_j.$$

Counter (CTR): The counter (CTR) mode builds upon the ideas that were used in the OFB mode. Just like OFB, CTR creates an output key stream that is XORed with chunks of plain text to produce cipher text. However, the CTR mode avoids the problems associated with the OFB mode by delinking the output stream O_j from the previous output streams.

In CTR we start by breaking up the plain text into 8-bit pieces, $P = [P_1, P_2, \ldots]$. We begin with an initial value X_1, which has a length equal to the block length of the cipher, for example, 64 bits of output, and we extract the leftmost 8-bits of the cipher text and XOR it with P_1 to produce 8 bits of cipher text, C_1.

However, for updating the register X_2, we do not use the output of the block cipher, we simply take $X_2 = X_1 + 1$. So, X_2 does not depend on previous output. We create new output stream in CTR by encrypting X_2. Similarly, we proceed by using $X_3 = X_2 + 1$, and so on. We produce the jth cipher text by XORing the left 8 bits from the encryption of the jth register with the corresponding plain text P_j.

In general, the procedure for CTR is

$$\begin{aligned} X_j &= X_{j-1} + 1 \\ O_j &= L_8\left(E_k\left(X_j\right)\right) \\ C_j &= P_j \oplus O_j \end{aligned}$$

for $j = 2, 3, \cdots,$. If we continually add 1 to X_j, it could eventually become too large. This is unlikely to happen, but if it does, we simply wrap around and start back at 0.

We can calculate the registers X_j ahead of time just like OFB and the actual encryption of plain text is simple in that it involves just the XOR operation. As a result, it fares as well as the OFB mode when errors are introduced in the cipher text. The advantage that CTR enjoys over OFB is that we can calculate many output chunks O_J in parallel. We do not have to calculate O_J before calculating O_{J+1}. This makes CTR mode ideal for parallelising.

In the next subsection, we will briefly explain the cryptanalysis techniques used to break the DES. We will also see why the algorithm became obsolete.

4.3.2 Cryptanalysis of DES

DES was the standard cryptographic system from the time it was approved as the standard in the late 90s. It was resistant to differential cryptanalysis, a powerful technique discovered by Shamir and Biham(See [2]). The technique was used successfully against Khafre, REDOC-11, FEAL and LOKI. See [1]. Differential Cryptanalysis can break DES with less than 16 rounds of encryption. The DES was also resistant to linear cryptanalysis, discovered by Matsui(See [9]) in 1994. It turned out that the designers of DES knew about differential and designed DES to be resistant to this attack. However, it seems that they didn't know about linear cryptanalysis.

If this was the situation regarding cryptanalysis techniques, the situation regarding brute force attack was somewhat different. From the time DES was released, the academic community felt that the key length was too small. In fact, a few months after the NBS release of DES, Whitfield Diffie and Martin Hellman published a paper titled "Exhaustive cryptanalysis of the NBS Data Encryption Standard", [5] in which they estimated that a machine, specially built for purpose of attacking the DES, could be built for $20 million. The proposed machine could crack DES in roughly a day. .

By brute force attack, we mean trying all the different 2^{56} possible keys.

Another such device to attack DES was proposed by Michael Wiener in 1993, a researcher at Bell-Northern researcher. This machine used the switching technology used by the telephone industry. In the year 1996, three different approaches were proposed for attacking symmetric ciphers like DES.

1. Distributed computing using a large collection of machines. This was relatively cheap and the cost could distributed amongst many people.
2. Design custom architecture(like Wiener's machine). While this is effective, it is expensive.
3. Using programmable arrays which is somewhere in between the two earlier approaches.

In 1997, RSA Data Security challenged the computing community to find the key and decrypt a message encrypted using the DES. It offered a prize money of $ 10,000 to anybody who succeeds in doing so.

Rock Verser submitted the winning DES key after five months. The noteworthy point in this success is the fact that Rock Verser used distributive computing approach. He used the free time available on the internet. People allowed the use of their computers with the understanding that Verser will share 40% of the prize money with the owner of the computer that found the key. The correct key was found after checking 25% of the possible keys. In 1997, RSA issued another similar challenge. This was cracked much faster by Distributed Computing Technologies, in only 39 days, that too after searching more keys, roughly 85% of the keys.

In 1998, the DES cracker built by the Electronic Frontier Foundation could crack DES in roughly 4.5 days on average. The aim of building the machine was to expose the vulnerability of the DES. The machine cost around $ 200,000 in 1998. See [7].

All these developments signalled the need to find a suitable replacement for DES. One approach was to encrypt using DES more than once. Merkle and Hellman, in [12], showed that encrypting twice with DES is not secure enough. However, encrypting thrice is secure enough and this algorithm is called **Triple DES**. Many other variants of the DES have been proposed. See [19] for a discussion.

Another approach was to develop a new algorithm. This approach lead to the development of the Advanced Encryption standards. We will take this up in the next section.

4.4 THE ADVANCED ENCRYPTION STANDARD

In 1997, the National Institute of Standards and Technology(NIST) called for candidates to replace DES. The following are some of the stipulations made by the NIST:

1. The algorithm should support various key lengths like 128, 192 and 256 bits.
2. The algorithm should work in various kinds of hardware and software.
3. The algorithm should be usable in various modes, ECB, CFB, etc.
4. The algorithm should be widely available on a non-exclusive, royalty-free basis.

The five algorithms that were ultimately shortlisted were

1. MARS from IBM.
2. RC6 from RSA laboratories.
3. Rijndael, proposed by Joan Daeman and Vincent Rijmen.
4. Serpent, proposed by Ross Anderson, Eli Biham, and Lars Knudsen.
5. Twofish, proposed by Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall and Niels Ferguson.

Acceptable pronunciations of Rijndael, according to its designers, are 'Rain doll', 'Reign Dahl' or 'Rhine Dahl'.

Finally, Rijndael was chosen as the Advanced Encryption Standard, the successor of DES. In this section, we will describe this algorithm.

This algorithm, as required by NIST, can be used with 128, 192 and 256 bit keys. Also, it could work in various modes and in various hardware like 8-bit processors used in smart cards and in 32 bit processors in PCs. It can be used in various modes like ECB, CBC, CFB, OFB, and CTR. It is also available worldwide on a non-exclusive, royalty-free basis.

4.4.1 The Basic Algorithm

Rijndael is designed for use with keys of lengths 128, 192, and 256 bits. For simplicity, we'll restrict to 128 bits. First, we give a brief outline of the algorithm, then describe the various components in more detail.

The algorithm consists of 10 rounds when the key has 128 bit, 12 rounds when there are 192 bits, and 14 rounds when the key has 256 bits. Each round has a round key, derived from the original key. There is also a 0th round key, which is the original key. A round starts with an input of 128 bits and produces an output of 128 bits.

There are four basic steps, called **layers**, that are used to form the rounds:

1) The SubBytes Transformation (BS)
2) The ShiftRows Transformation (SR)
3) The MixColumns Transformation (MC)
4) AddRoundKey (ARK)

Putting everything together, we obtain the following:

Rijndael Encryption

1. ARK, using the 0th round key.
2. Nine rounds of BS, SR, MC, ARK, using round keys 1 to 9.
3. A final round: BS, SR, ARK, using the 10th round key.

4.4.2 The Layers

We now describe the steps in more detail. The 128 input bits are grouped into 16 bytes of 8 bits each, call them

$$a_{0,0}, a_{0,1}, a_{0,2}, a_{0,3}, a_{1,0}, a_{1,1}, a_{1,2}, a_{1,3} \ldots, a_{3,3}$$

These are arranged into a 4×4 matrix

$$\begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix} \tag{5}$$

We can regard the entries of the matrix in Eqn. (5) as elements of $\mathbf{F}_{2^8}$. Let us see how. Recall that the finite field $L = \mathbf{F}_{2^8}$ is a vector space of dimension 8 over $\mathbf{F}_2$. So, once we fix a basis for L over $\mathbf{F}_2$, every element of $\mathbf{F}_{2^8}$ can be represented by a 8-tuple of 0s and 1s. Alternately, we can represent an element of $\mathbf{F}_{2^8}$ by a string of length 8 in 0 and 1. For example, we can write the vector $(0,1,1,0,1,0,1,1) \in \mathbf{F}_{2^8}$ as the string 01101011. So, we can represent every element of $\mathbf{F}_{2^8}$ by a byte and conversely, every byte can be regarded as an element of $\mathbf{F}_{2^8}$. If we identify $\mathbf{F}_{2^8}$ with $\mathbf{F}_2[X]/\langle g(X)\rangle$ for some irreducible polynomial $g(X) \in \mathbf{F}_2[X]$ of degree 8, one natural choice for the basis is the set $\left\{\overline{1}, \overline{X}, \overline{X}^2, \ldots, \overline{X}^7\right\}$ where $\overline{X} = X + \langle g(X)\rangle$ and $\overline{1} = 1 + \langle g(X)\rangle$. When there is no possiblity of confusion we will simply write 1 instead of $\overline{1}$.

Let us see how to carry out arithmetic operations with bytes. We identify a byte $a_7a_6\cdots a_1a_0$ with the element

$$a_7\overline{X}^7 + a_6\overline{X}^6 + \cdots + a_1\overline{X} + a_0 \in \mathbf{F}_{2^8} = \mathbf{F}_2[X]/\langle g(X)\rangle.$$

So, we can regard the entries of the 4×4 matrix in Eqn. (5) as elements of L. We can add two such entries by XORing them.

Multiplication is a lot more complicated. We have to find a proper representation of $\mathbf{F}_{2^8}$ for this. In AES, we represent $\mathbf{F}_{2^8}$ as $\mathbf{F}_2[X]/\langle g(X)\rangle$ where

$$g(X) = X^8 + X^4 + X^3 + X + 1.$$

Suppose we want to multiply two bytes, $b = 10110111$ and $b' = 10001100$. Under our representation for $\mathbf{F}_{2^8}$ the byte b corresponds to the element

$$X^7 + X^5 + X^4 + X^2 + X + 1 + \langle g(X)\rangle \in \mathbf{F}_2[X]/\langle g(X)\rangle$$

and b' corresponds to

$$X^7 + X^3 + X^2 + \langle g(X)\rangle \in \mathbf{F}_2[X]/\langle g(X)\rangle.$$

To multiply b and b' we multiply the polynomials

$$X^7 + X^5 + X^4 + X^2 + X^2 + X + 1,\ X^7 + X^3 + X^2 \in \mathbf{F}_2[X]$$

and divide the product by $g(X)$. We leave it to you to verify that

$$\begin{aligned} &\left(X^7 + X^5 + X^4 + X^2 + X^2 + X + 1\right)\left(X^7 + X^3 + X^2\right) \\ &= X^{14} + X^{12} + X^{11} + X^{10} + X^7 + X^6 + X^5 + X^2. \end{aligned}$$

If we divide the polynomial on the RHS in the above equation by $g(X)$, the remainder is

$$X^6 + X^5 + 1 \in \mathbf{F}_2[X]$$

So, $b \cdot b' = 01100001$.

Since $\mathbf{F}_{2^8}$ is a field, every non-zero element is invertible. So, each byte b except the zero byte has a multiplicative inverse; that is, there is a byte b' such that $b \cdot b' = 00000001$. To find the inverse of b, we proceed as follows: Suppose b corresponds to $p(X) + \langle g(X) \rangle$, $p(X) \in \mathbf{F}_2[X]$ of degree less than 8. Since $g(X)$ is irreducible, $(g(X), p(X)) = 1$. So, using extended Euclid's algorithm for finding the gcd of polynomials, we can find $h(X)$, $q(X) \in \mathbf{F}_2[X]$ such that

$$g(X)h(X) + q(X)p(X) = 1.$$

Then, the byte corresponding to $q(X)$ is the inverse of b.

For example, if we take $b = 10001100$ we have

$$\left(X^7 + X^3 + X^2\right)\left(X^7 + X^6 + X^5 + X^4 + X^2 + X + 1\right) + g(X)\left(X^6 + X^5 + X^4 + X^3 + X + 1\right) = 1$$

So, the inverse of $b = 10001100$ is 11110111. Thus, since we can do arithmetic operations on bytes, we can work with matrices whose entries are bytes.

Before you proceed further, try the following exercises to check your understanding.

E4) a) Find the product of the bytes 10011011 and 10001001.

b) Find the inverse of 10001001.

The BytesSub Transformation

This is a non-linear transformation using S-boxes. The purpose is to make the algorithm resistant to differential and linear cryptanalysis attacks.

In this step, each of the bytes in the matrix is changed to another byte by Table 10, called the S-box.

As before, write a byte as 8 bits: $a_7a_6a_5a_4a_3a_2a_1a_0$. We split it into two parts $a_7a_6a_5a_4$ and $a_3a_2a_1a_0$. Note that a four bit number can be represented as a single digit number in hexadecimal system. For example, 1111 is 15 in the decimal system and hence f in the hexadecimal system. Look for the entry in the $a_7a_6a_5a_4$ row and $a_3a_2a_1a_0$ column (the rows and columns are numbered using the hexadecimal digits 0, 1, 2, 3, …, 9, a, b, c, …, f.). This entry, when converted to binary, is the output. For example, suppose the input byte is 10001010. Then, 1000 is 8 in the decimal system and it is represented by 8 in the hexadecimal system also. We have that 1010 is 10 in decimal and it is represented by a in the hexadecimal system. We look in the row labelled 8 and the column labelled a. The entry is 7e which is $7 \cdot 16 + 14 = 126$ in decimal and 1111110 in binary. This is the output of the S-box.

The output of SubBytes is again a 4×4 matrix of bytes, let's call it

$$\begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix}$$

The ShiftRows Transformation is a linear mixing step for diffusion of the bits over multiple rounds. It uses arithmetic over the finite field with 2^8 elements. The four rows of the matrix are shifted cyclically to the left by offsets of 0, 1, 2, and 3, to obtain

$$\begin{pmatrix} c_{0,0} & c_{0,1} & c_{0,2} & c_{0,3} \\ c_{1,0} & c_{1,1} & c_{1,2} & c_{1,3} \\ c_{2,0} & c_{2,1} & c_{2,2} & c_{2,3} \\ c_{3,0} & c_{3,1} & c_{3,2} & c_{3,3} \end{pmatrix} = \begin{pmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,1} & b_{1,2} & b_{1,3} & b_{1,0} \\ b_{2,2} & b_{2,3} & b_{2,0} & b_{2,1} \\ b_{3,3} & b_{3,0} & b_{3,1} & b_{3,2} \end{pmatrix}$$

Table 10: S-Box for the AES

		X														
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
Y	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

The MixColumns Transformation layer has a purpose similar to ShiftRows. We regard a byte as an element of $\mathbf{F}_{2^8}$ as we explained earlier. Then the output of the ShiftRows step is a 4×4 matrix $\left(c_{i,j}\right)$ with entries in $\mathbf{F}_{2^8}$. We then multiply this by a matrix with entries in $\mathbf{F}_{2^8}$, to produce the output $\left(d_{i,j}\right)$, as follows:

$$\begin{pmatrix} 00000010 & 00000011 & 00000001 & 00000001 \\ 00000001 & 00000010 & 00000011 & 00000001 \\ 00000001 & 00000001 & 00000010 & 00000011 \\ 00000011 & 00000001 & 00000001 & 00000010 \end{pmatrix} \begin{pmatrix} c_{0,0} & c_{0,1} & c_{0,2} & c_{0,3} \\ c_{1,0} & c_{1,1} & c_{1,2} & c_{1,3} \\ c_{2,0} & c_{2,1} & c_{2,2} & c_{2,3} \\ c_{3,0} & c_{3,1} & c_{3,2} & c_{3,3} \end{pmatrix}$$
$$= \begin{pmatrix} d_{0,0} & d_{0,1} & d_{0,2} & d_{0,3} \\ d_{1,0} & d_{1,1} & d_{1,2} & d_{1,3} \\ d_{2,0} & d_{2,1} & d_{2,2} & d_{2,3} \\ d_{3,0} & d_{3,1} & d_{3,2} & d_{3,3} \end{pmatrix}$$

The RoundKey Addition: In AddRoundKey layer we XOR the round key with the result of the above layer. The round key, derived from the key in a way we'll describe later, consists of 128 bits, which are arranged in a 4×4 matrix $\left(k_{i,j}\right)$ consisting of bytes. This is XORed with the output of the MixColumns step:

$$\begin{pmatrix} d_{0,0} & d_{0,1} & d_{0,2} & d_{0,3} \\ d_{1,0} & d_{1,1} & d_{1,2} & d_{1,3} \\ d_{2,0} & d_{2,1} & d_{2,2} & d_{2,3} \\ d_{3,0} & d_{3,1} & d_{3,2} & d_{3,3} \end{pmatrix} \oplus \begin{pmatrix} k_{0,0} & k_{0,1} & k_{0,2} & k_{0,3} \\ k_{1,0} & k_{1,1} & k_{1,2} & k_{1,3} \\ k_{2,0} & k_{2,1} & k_{2,2} & k_{2,3} \\ k_{3,0} & k_{3,1} & k_{3,2} & k_{3,3} \end{pmatrix} = \begin{pmatrix} e_{0,0} & e_{0,1} & e_{0,2} & e_{0,3} \\ e_{1,0} & e_{1,1} & e_{1,2} & e_{1,3} \\ e_{2,0} & e_{2,1} & e_{2,2} & e_{2,3} \\ e_{3,0} & e_{3,1} & e_{3,2} & e_{3,3} \end{pmatrix}$$

This is the final output of the round.

The Key Schedule

We arrange the original key, which consists of 128 bits, into a 4×4 matrix of bytes. We then add 40 more columns to this matrix as follows. Let us label the first four columns $W(0), W(1), W(2), W(3)$. We then generate the new columns recursively. Suppose columns up through $W(i-1)$ have been defined. If i is a not a multiple of 4, then

$$W(i) = W(i-4) \oplus W(i-1).$$

If i is a multiple of four then

$$W(i) = W(i-4) \oplus T(W(i-1)).$$

where $T(W(i-1))$ is the transformation of $W(i-1)$ obtained as follows. Let the elements of the column $W(i-1)$ be a, b, c, d. We shift these cyclically to obtain b, c, d, a. Next, we replace each of these bytes with the corresponding element in the S-box from the SubBytes step, to get 4 bytes e, f, g, h. Finally, we compute the round constant.

$$r(i) = 00000010^{(i-4)/4}$$

in GF (2^8) (recall that we are in the case where i is a multiple of 4). Then $T(W(i-1))$ is the column vector

$$(e \oplus r(i), f, g, h).$$

In this way, we generate columns $W(4), \ldots, W(43)$ from the initial four columns.

The **round key** for the ith round consists of the columns

$$W(4i), W(4i+1), W(4i+2), W(4i+3).$$

The Construction of the S-Box

Although the S-box is implemented as a lookup table, it has a simple mathematical description. We start with a byte $a_7a_6a_5a_4a_3a_2a_1a_0$, where each a_i is a binary bit. We compute its inverse as we described earlier. If the byte is 00000000, it has no inverse, so we use 00000000 in place of its inverse. The resulting byte $b_7b_6b_5b_4b_3b_2b_1b_0$ represents an eight-dimensional column vector, with the rightmost bit b_0 in the top position. We multiply this column vector by a matrix and add the column vector $(1,1,0,0,0,1,1,0)$ to obtain a vector $(c_0, c_1, c_2, c_3, c_4, c_5, c_6, c_7)$ as follows:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ c_6 \\ c_7 \end{pmatrix}$$

The byte $c_7c_6c_5c_4c_4c_3c_2c_1c_0$ is the entry in the S-box. For example, suppose we start with the byte 11001011. Its inverse in $\mathbf{F}_{2^8}$ is 00000100. We have

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

This yields the byte 00011111. The first four bits 1100 represent 12 in binary and the last 4 bits 1011 represent 11 in binary. We add 1 to each of these numbers (since the first row and column are numbers 0) and look in the 13th row and 12th column of the S-box. The entry is 31, which in binary is 00011111.

Some of the considerations in the design of the S-box were the following. To make choice of an element from the S-box non-linear, the designers of the AES chose the map $x \mapsto x^{-1}$. Since this map was very simple, it was possibily vulnerable to certain attacks, so it was combined with multiplication by the matrix and adding the vector, as described previously. The matrix was chosen mostly because of its simple form (note how the rows are shifts of each other). The vector was chosen so that no input ever equals its S-box output or the complement of its S-box output (complementation means changing each 1 to 0 and each 0 to 1).

4.4.3 Decryption

Each of the steps SubBytes, ShiftRows, MixColumns, and AddRoundKey is invertible:

1. The inverse of SubBytes is another lookup table, called **InvSubBytes**.
2. The inverse of ShiftRows is obtained by shifting the rows to the right instead of to the left, yielding **InvSubBytes**.
3. The inverse of MixColumns exists because the 4×4 matrix used in MixColumns is invertible. The transformation **InvMixColumns** is given by multiplication by the matrix

$$\begin{pmatrix} 00001110 & 00001011 & 00001101 & 00001001 \\ 00001001 & 00001110 & 00001011 & 00001101 \\ 00001101 & 00001001 & 00001110 & 00001011 \\ 00001011 & 00001101 & 00001001 & 000011101 \end{pmatrix}$$

4. AddRoundKey is its own inverse.

The Rijndael encryption consists of the steps

ARK
BS, SR, MC, ARK
...
BS, SR, MC, ARK
BS, SR, ARK.

Recall that MC is missing in the last round.

To decrypt, we need to run through the inverses of these steps in the reverse order. This yields the following preliminary version of decryption:

ARK, ISR, IBS
ARK, IMC, ISR, IBS
...
ARK, IMC, ISR, IBS
ARK.

We conclude our discussion of AES here.

4.5 SUMMARY

In this unit, we discussed the following:

1) We defined what are block ciphers and stream ciphers;
2) We disussed the general design principles behind block ciphers as enunciated by Claude Shannon;
3) We saw how to encrypt and decrypt using a toy block cipher;
4) We saw how DES encrypts and decrypts messages;
5) We saw various modes of operation of block ciphers;
6) We saw why DES had to be replaced; and
7) We discussed how the AES encrypts and decrypts messages;

4.6 SOLUTIONS/ANSWERS

E1) **Step 1** Split the input into two halves 101011 and 100100.

Step 2 Expand the right half to 10101000.

Step 3 XOR with the key 10101110 to get 00000110. Divide into two parts and use the S-boxes:

$0000 \rightarrow$ First element in the first row of S_1 : 101
$0110 \rightarrow$ Seventh element in the first row of S_2 : 011

Concatenate to get $f(R_1, k_2) = 101011$.

Step 4 To get R_2, XOR $f(R_1, k_2)$ with L_1:

$$R_2 = f(R_1, k_2) \oplus L_1 = 101011 \oplus 101011 = 000000.$$

So, R_2 is 000000 and $L_2 = R_1 = 100100$

E2) The key k_2 for the second round is 11001110. The key for the fourth round is 00111001.

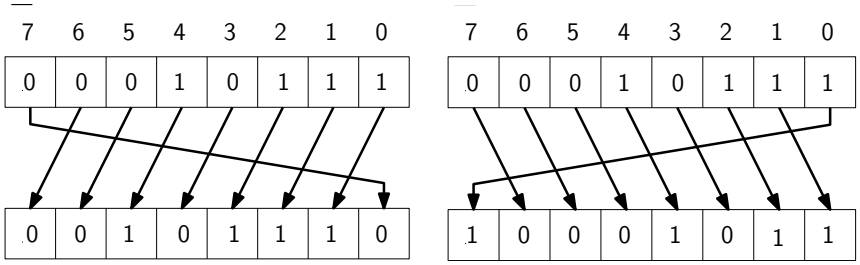
E3) The plain text is 000110001011

E4) a) 10011010.

b) 10011101

APPENDIX: LEFT AND RIGHT CIRCULAR SHIFTS

In this appendix, we will discuss the **right and left circular shifts**. The operator rotates the bits to the left or to the right. The symbol for the left shift operator is $\lll$ and the symbol for the right shift operator is $\ggg$. We write x $\lll$ n(resp. x $\ggg$ n) to denote the shift of the bits by n positions to the left(resp. n positions to the right). For example, x $\lll$ 1 shifts the bits to the left by one place and puts the left most bit at the right most position. Fig. 2(a) shows what happens to the bits in this case. Similarly, Fig. 2(b) shows what happens when we rotate the bits by one place to the right. Note that, x $\lll$ n(resp. x $\ggg$ n) has the same effect as applying x $\lll$ 1(resp. x $\ggg$ 1) n times.



(a) Left circular shift. **(b) Right circular shift.**

Fig. 2: Right and left circular shifts.

```
/* Assumes that the number of bits in an unsigned int is 32 */
/*Left circular shift*/
unsigned int_rotl(unsigned int value, int shift) {
    if ((shift &= 31) == 0)
      return value;
    return (value << shift) | (value >> (32 - shift));
}
/*Right circular shift*/
unsigned int_rotr(unsigned int value, int shift) {
    if ((shift &= 31) == 0)
      return value;
    return (value >> shift) | (value << (32 - shift));
}
```

Listing 4.1: C function for left and right circular shifts. Source: Wikipaedia