Chapter 7

The Advanced Encryption Standard (AES)

All of the cryptographic algorithms we have looked at so far have some problem. The earlier ciphers can be broken with ease on modern computation systems. The DES algorithm was broken in 1998 using a system that cost about $250,000. It was also far too slow in software as it was developed for mid-1970's hardware and does not produce efficient software code. Triple DES on the other hand, has three times as many rounds as DES and is correspondingly slower. As well as this, the 64 bit block size of triple DES and DES is not very efficient and is questionable when it comes to security.

What was required was a brand new encryption algorithm. One that would be resistant to all known attacks. The National Institute of Standards and Technology (NIST) wanted to help in the creation of a new standard. However, because of the controversy that went with the DES algorithm, and the years of some branches of the U.S. government trying everything they could to hinder deployment of secure cryptography this was likely to raise strong skepticism. The problem was that NIST did actually want to help create a new excellent encryption standard but they couldn't get involved directly. Unfortunately they were really the only ones with the technical reputation and resources to the lead the effort.

Instead of designing or helping to design a cipher, what they did instead was to set up a contest in which anyone in the world could take part. The contest was announced on the 2nd of January 1997 and the idea was to develop a new encryption algorithm that would be used for protecting sensitive, non-classified, U.S. government information. The ciphers had to meet a lot of requirements and the whole design had to be fully documented (unlike the DES cipher). Once the candidate algorithms had been submitted, several years of scrutinisation in the form of cryptographic conferences took place. In the first round of the competition 15 algorithms were accepted and this was narrowed to 5 in the second round. The fifteen algorithms are shown in table 7 of which the 5 that were selected are shown in bold. The algorithms were tested for efficiency and security both by some of the worlds best publicly renowned cryptographers and NIST itself.

After all this investigation NIST finally chose an algorithm known as **Rijndael**. Rijndael was named after the two Belgian cryptographers who developed and submitted it - Dr. Joan Daemen of Proton World International and Dr. Vincent Rijmen, a postdoctoral researcher in the Electrical Engineering Department of Katholieke Universisteit Leuven. On the 26 November 2001, AES (which is a standarised version of Rijndael)

ALGORITHM NAME	SUBMITTER
CAST-256	Entrust Technologies, Inc.
CRYPTON	Future Systems, Inc.
DEAL	Richard Outerbridge, Lars Knudsen
DFC	CNRS - Centre National pour la Recherche Scientifique - Ecole Normale Superieure
E2	NTT - Nippon Telegraph and Telephone Corporation
FROG	TecApro Internacional S.A.
HPC	Rich Schroeppel
LOKI97	Lawrie Brown, Josef Pieprzyk, Jennifer Seberry
MAGENTA	Deutsche Telekom AG
MARS	IBM
RC 6	RSA Laboratories
Rijndael	Joaen Daemen, Vincent Rijmen
SAFER+	Cylink Corporation
Serpent	Ross Anderson, Eli Biham, Lars Knudsen
Twofish	Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, Niels Ferguson

became a FIPS standard (FIPS 197).

7.1 The AES cipher

Like DES, AES is a symmetric block cipher. This means that it uses the same key for both encryption and decryption. However, AES is quite different from DES in a number of ways. The algorithm Rijndael allows for a variety of block and key sizes and not just the 64 and 56 bits of DES' block and key size. The block and key can in fact be chosen independently from $128, 160, 192, 224, 256$ bits and need not be the same. However, the AES standard states that the algorithm can only accept a block size of 128 bits and a choice of three keys - $128, 192, 256$ bits. Depending on which version is used, the name of the standard is modified to AES-128, AES-192 or AES-256 respectively. As well as these differences AES differs from DES in that it is not a feistel structure. Recall that in a feistel structure, half of the data block is used to modify the other half of the data block and then the halves are swapped. In this case the entire data block is processed in parallel during each round using substitutions and permutations.

A number of AES parameters depend on the key length. For example, if the key size used is 128 then the number of rounds is 10 whereas it is 12 and 14 for 192 and 256 bits respectively. At present the most common key size likely to be used is the 128 bit key. This description of the AES algorithm therefore describes this particular implementation.

Rijndael was designed to have the following characteristics:

- Resistance against all known attacks.
- Speed and code compactness on a wide range of platforms.
- Design Simplicity.

The overall structure of AES can be seen in 7.1. The input is a single 128 bit block both for decryption and encryption and is known as the **in** matrix. This block is copied into a **state** array which is modified at each stage of the algorithm and then copied to an output matrix (see figure 7.2). Both the plaintext and key are depicted as a 128 bit square matrix of bytes. This key is then expanded into an array of key schedule words (the **w** matrix). It must be noted that the ordering of bytes within the **in** matrix is by column. The same applies to the **w** matrix.

7.2 Inner Workings of a Round

The algorithm begins with an **Add round key** stage followed by 9 rounds of four stages and a tenth round of three stages. This applies for both encryption and decryption with the exception that each stage of a round the decryption algorithm is the inverse of it's counterpart in the encryption algorithm. The four stages are as follows:

1. Substitute bytes
2. Shift rows
3. Mix Columns
4. Add Round Key

The tenth round simply leaves out the **Mix Columns** stage. The first nine rounds of the decryption algorithm consist of the following:

1. Inverse Shift rows
2. Inverse Substitute bytes
3. Inverse Add Round Key
4. Inverse Mix Columns

Again, the tenth round simply leaves out the **Inverse Mix Columns** stage. Each of these stages will now be considered in more detail.

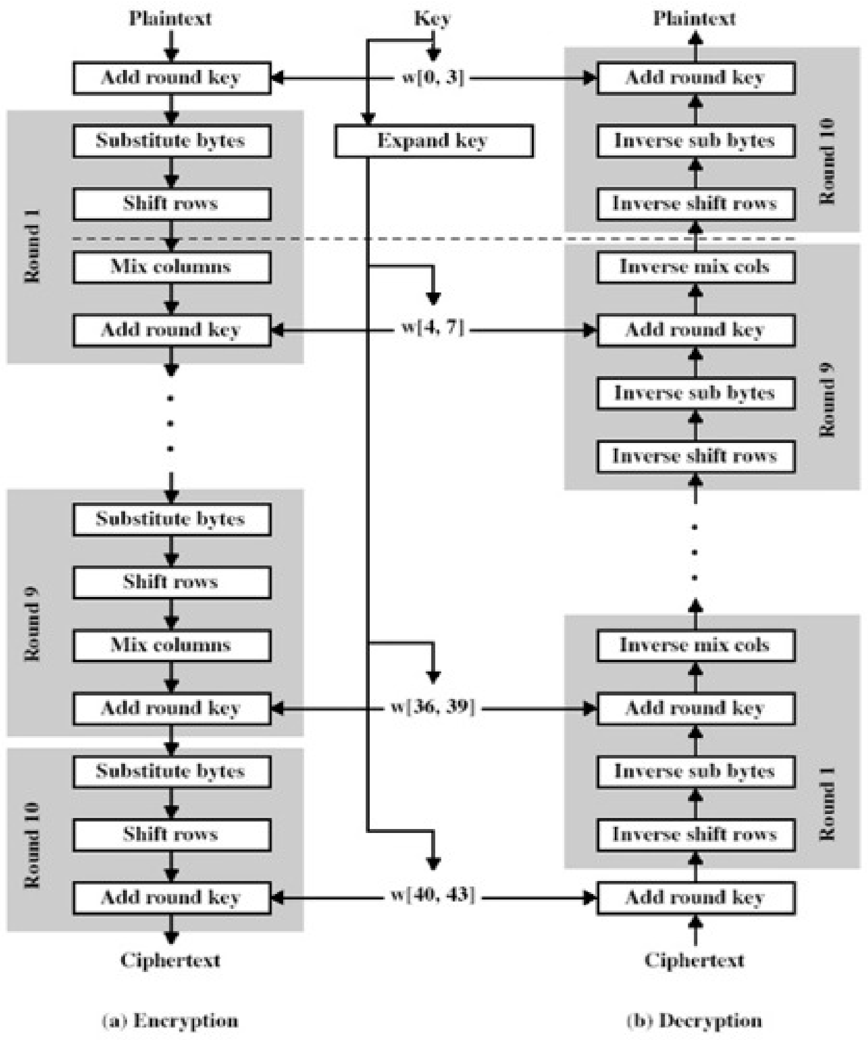


Figure 7.1: Overall structure of the AES algorithm.

7.3 Substitute Bytes

This stage (known as SubBytes) is simply a table lookup using a 16×16 matrix of byte values called an **s-box**. This matrix consists of all the possible combinations of an 8 bit sequence ($2^8 = 16 \times 16 = 256$). However, the s-box is not just a random permutation of these values and there is a well defined method for creating the s-box tables. The designers of Rijndael showed how this was done unlike the s-boxes in DES for which no rationale was given. We will not be too concerned here how the s-boxes are made up and can simply take them as table lookups.

Again the matrix that gets operated upon throughout the encryption is known as **state**. We will be concerned with how this matrix is effected in each round. For this particular

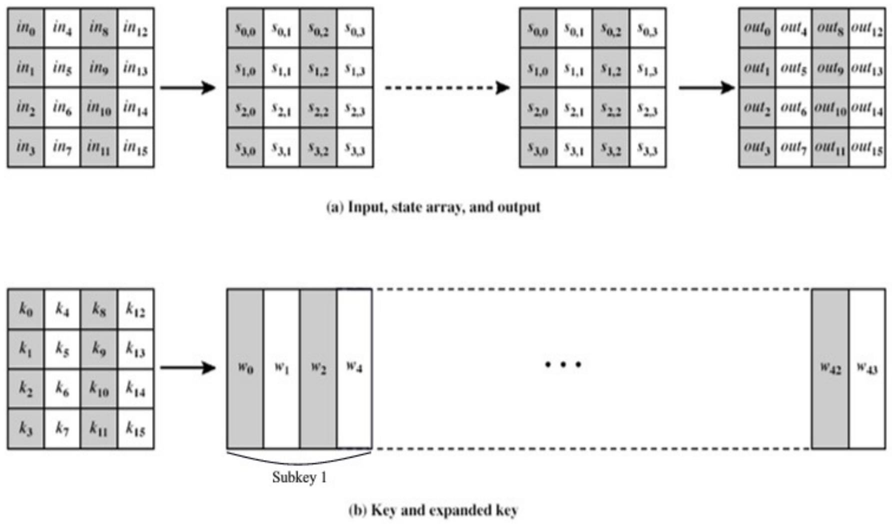


Figure 7.2: Data structures in the AES algorithm.

round each byte is mapped into a new byte in the following way: the leftmost nibble of the byte is used to specify a particular row of the s-box and the rightmost nibble specifies a column. For example, the byte $\{95\}$ (curly brackets represent hex values in FIPS PUB 197) selects row 9 column 5 which turns out to contain the value $\{2A\}$. This is then used to update the **state** matrix. Figure 7.3 depicts this idea.

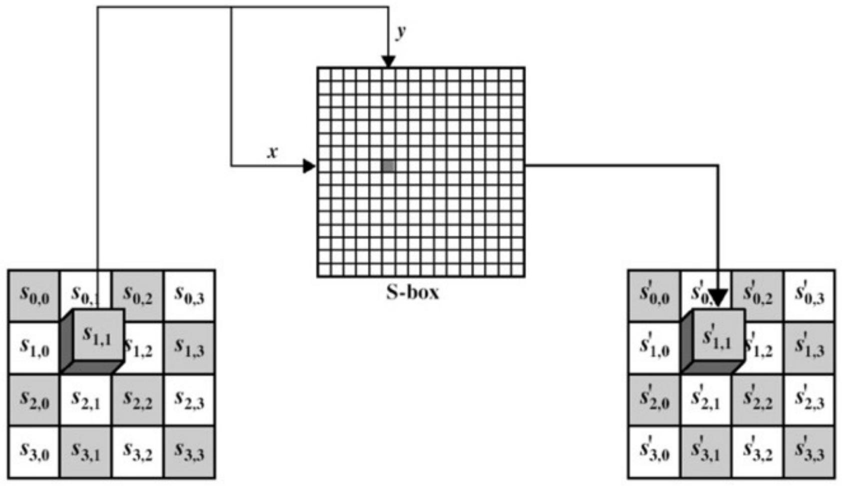
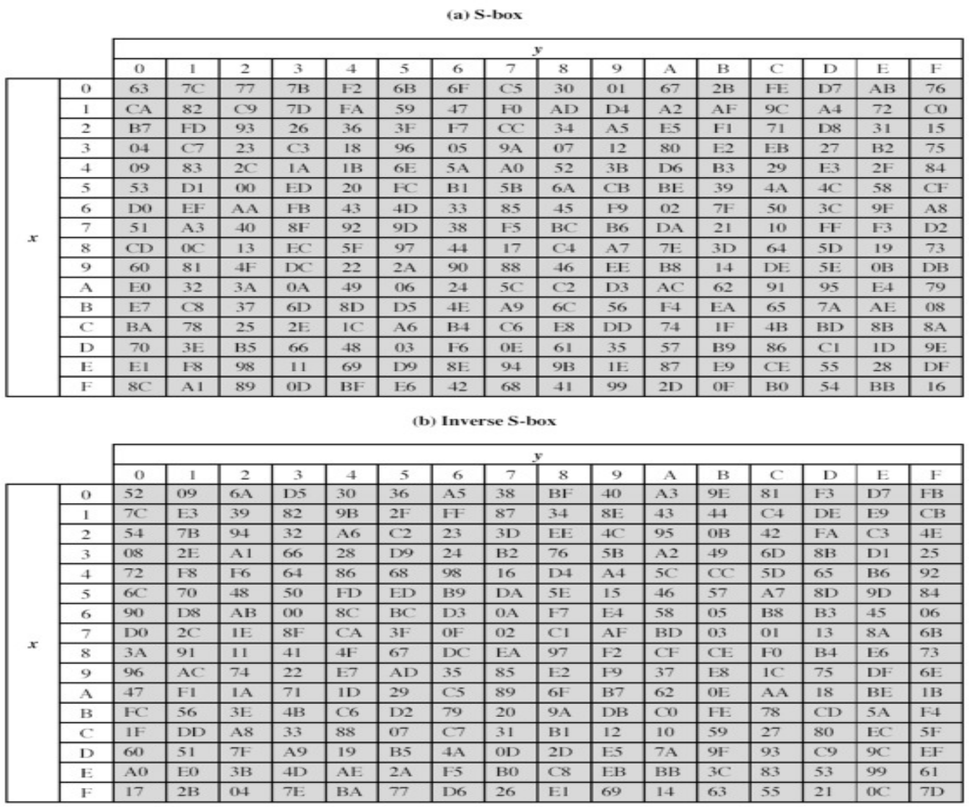


Figure 7.3: Substitute Bytes Stage of the AES algorithm.

The Inverse substitute byte transformation (known as InvSubBytes) makes use of an inverse s-box. In this case what is desired is to select the value $\{2A\}$ and get the value $\{95\}$. Table 7.4 shows the two s-boxes and it can be verified that this is in fact the case.

The s-box is designed to be resistant to known cryptanalytic attacks. Specifically, the Rijndael developers sought a design that has a low correlation between input bits and output bits, and the property that the output cannot be described as a simple mathematical function of the input. In addition, the s-box has no fixed points (s-box$(a) = a$) and no opposite fixed points (s-box$(a) = \bar{a}$) where $\bar{a}$ is the bitwise compliment of a. The s-box must be invertible if decryption is to be possible (Is-box[s-box(a)]$= a$) however it should not be its self inverse i.e. s-box$(a) \neq$ Is-box(a)



(a) S-box

		y															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	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
x	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

(b) Inverse S-box

		y															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	0	52	09	6A	D5	30	36	A5	38	BF	40	A3	9E	81	F3	D7	FB
	1	7C	E3	39	82	9B	2F	FF	87	34	8E	43	44	C4	DE	E9	CB
	2	54	7B	94	32	A6	C2	23	3D	EE	4C	95	0B	42	FA	C3	4E
	3	08	2E	A1	66	28	D9	24	B2	76	5B	A2	49	6D	8B	D1	25
	4	72	F8	F6	64	86	68	98	16	D4	A4	5C	CC	5D	65	B6	92
	5	6C	70	48	50	FD	ED	B9	DA	5E	15	46	57	A7	8D	9D	84
	6	90	D8	AB	00	8C	BC	D3	0A	F7	E4	58	05	B8	B3	45	06
x	7	D0	2C	1E	8F	CA	3F	0F	02	C1	AF	BD	03	01	13	8A	6B
	8	3A	91	11	41	4F	67	DC	EA	97	F2	CF	CE	F0	B4	E6	73
	9	96	AC	74	22	E7	AD	35	85	E2	F9	37	E8	1C	75	DF	6E
	A	47	F1	1A	71	1D	29	C5	89	6F	B7	62	0E	AA	18	BE	1B
	B	FC	56	3E	4B	C6	D2	79	20	9A	DB	C0	FE	78	CD	5A	F4
	C	1F	DD	A8	33	88	07	C7	31	B1	12	10	59	27	80	EC	5F
	D	60	51	7F	A9	19	B5	4A	0D	2D	E5	7A	9F	93	C9	9C	EF
	E	A0	E0	3B	4D	AE	2A	F5	B0	C8	EB	BB	3C	83	53	99	61
	F	17	2B	04	7E	BA	77	D6	26	E1	69	14	63	55	21	0C	7D

Figure 7.4: AES s-boxes both forward and inverse.

7.4 Shift Row Transformation

This stage (known as ShiftRows) is shown in figure 7.5. This is a simple permutation an nothing more. It works as follow:

- The first row of **state** is *not* altered.
- The second row is shifted 1 bytes to the left in a circular manner.
- The third row is shifted 2 bytes to the left in a circular manner.

- The fourth row is shifted 3 bytes to the left in a circular manner.

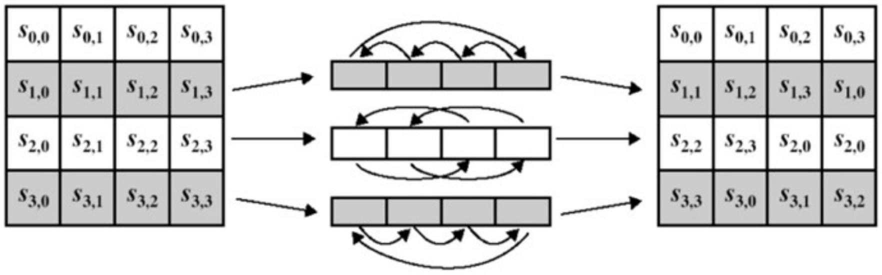

$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,0}$ $s_{1,1}$ $s_{1,2}$ $s_{1,3}$
$s_{2,0}$ $s_{2,1}$ $s_{2,2}$ $s_{2,3}$
$s_{3,0}$ $s_{3,1}$ $s_{3,2}$ $s_{3,3}$
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,1}$ $s_{1,2}$ $s_{1,3}$ $s_{1,0}$
$s_{2,2}$ $s_{2,3}$ $s_{2,0}$ $s_{2,0}$
$s_{3,3}$ $s_{3,0}$ $s_{3,1}$ $s_{3,2}$

Figure 7.5: ShiftRows stage.

The Inverse Shift Rows transformation (known as InvShiftRows) performs these circular shifts in the opposite direction for each of the last three rows (the first row was unaltered to begin with).

This operation may not appear to do much but if you think about how the bytes are ordered within **state** then it can be seen to have far more of an impact. Remember that **state** is treated as an array of four byte columns, i.e. the first column actually represents bytes $1, 2, 3$ and 4. A one byte shift is therefore a linear distance of four bytes. The transformation also ensures that the four bytes of one column are spread out to four different columns.

7.5 Mix Column Transformation

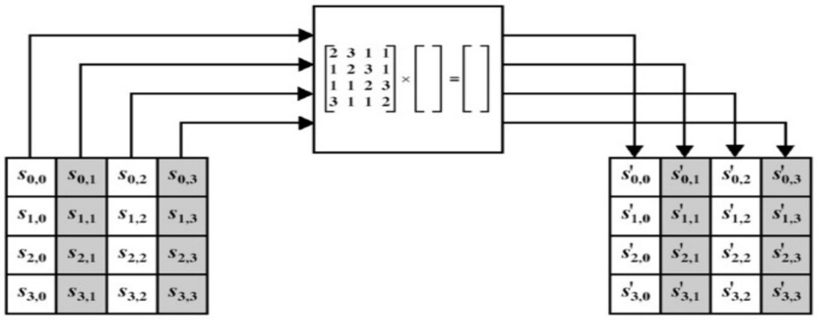
This stage (known as MixColumn) is basically a substitution but it makes use of arithmetic of GF(2^8). Each column is operated on individually. Each byte of a column is mapped into a new value that is a function of all four bytes in the column. The transformation can be determined by the following matrix multiplication on **state** (see figure 7.6):

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} = \begin{bmatrix} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix} \quad (7.1)$$

Each element of the product matrix is the sum of products of elements of one row and one column. In this case the individual additions and multiplications are performed in GF(2^8). The MixColumns transformation of a single column j $(0 \leq j \leq 3)$ of **state** can be expressed as:

$$\begin{aligned}
s'_{0,j} &= (2 \bullet s_{0,j}) \oplus (3 \bullet s_{1,j}) \oplus s_{2,j} \oplus s_{3,j} \\
s'_{1,j} &= s_{0,j} \oplus (2 \bullet s_{1,j}) \oplus (3 \bullet s_{2,j}) \oplus s_{3,j} \\
s'_{2,j} &= s_{0,j} \oplus s_{1,j} \oplus (2 \bullet s_{2,j}) \oplus (3 \bullet s_{3,j}) \\
s'_{3,j} &= (3 \bullet s_{0,j}) \oplus s_{1,j} \oplus s_{2,j} \oplus (2 \bullet s_{3,j})
\end{aligned} \tag{7.2}$$

where $\bullet$ denotes multiplication over the finite field GF(2^8).



2 3 1 1
1 2 3 1
1 1 2 3
3 1 1 2
×
=
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,0}$ $s_{1,1}$ $s_{1,2}$ $s_{1,3}$
$s_{2,0}$ $s_{2,1}$ $s_{2,2}$ $s_{2,3}$
$s_{3,0}$ $s_{3,1}$ $s_{3,2}$ $s_{3,3}$
$s'_{0,0}$ $s'_{0,1}$ $s'_{0,2}$ $s'_{0,3}$
$s'_{1,0}$ $s'_{1,1}$ $s'_{1,2}$ $s'_{1,3}$
$s'_{2,0}$ $s'_{2,1}$ $s'_{2,2}$ $s'_{2,3}$
$s'_{3,0}$ $s'_{3,1}$ $s'_{3,2}$ $s'_{3,3}$

Figure 7.6: MixColumns stage.

As and example, lets take the first column of a matrix to be $s_{0,0} = \{87\}, s_{1,0} = \{6E\}, s_{2,0} = \{46\}, s_{3,0} = \{A6\}$. This would mean that $s_{0,0} = \{87\}$ gets mapped to the value $s'_{0,0} = \{47\}$ which can be seen by working out the first line of equation 7.2 with $j = 0$. Therefore we have:

$$(02 \bullet 87) \oplus (03 \bullet 6E) \oplus 46 \oplus A6 = 47$$

So to show this is the case we can represent each Hex number by a polynomial:

$$\begin{aligned}
\{02\} &= x \\
\{87\} &= x^7 + x^2 + x + 1
\end{aligned}$$

Multiply these two together and we get:

$$x \bullet (x^7 + x^2 + x + 1) = x^8 + x^3 + x^2 + x$$

The degree of this result is greater than 7 so we have to reduce it modulo an irreducible polynomial $m(x)$. The designers of AES chose $m(x) = x^8 + x^4 + x^3 + x + 1$. So it can be seen that

$$(x^8 + x^3 + x^2 + x) \bmod (x^8 + x^4 + x^3 + x + 1) = x^4 + x^2 + 1$$

This is equal to $0001\ 0101$ in binary. This method can be used to work out the other terms. The result is therefore:

$$\begin{array}{rll} & 0001\,0101 & \\ & 1011\,0010 & \\ & 0100\,0110 & \\ \oplus & 1010\,0110 & \\ \hline & 0100\,0111 & = \{47\} \end{array}$$

There is infact an easier way to do multiplication modulo $m(x)$. If we were multiplying by $\{02\}$ then all we have to do is a 1-bit left shift followed by a conditional bitwise XOR with (00011011) if the leftmost bit of the original value (prior to the shift) was 1. Multiplication by other numbers can be seen to be repeated application of this method. Stallings goes into more detail on why this works but we will not be too concerned with it here. What is important to note however is that a multiplication operation has been reduced to a shift and an XOR operation. This is one of the reasons why AES is a very efficient algorithm to implement.

The InvMixColumns is defined by the following matrix multiplication:

$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} = \begin{bmatrix} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix} \tag{7.3}$$

This first matrix of equation 7.1 can be shown to be the inverse of the first matrix in equation 7.3. If we label these **A** and $\mathbf{A}^{-1}$ respectively and we label state before the mix columns operation as **S** and after as **S**′, we can see that:

$$\mathbf{AS} = \mathbf{S}'$$

therefore

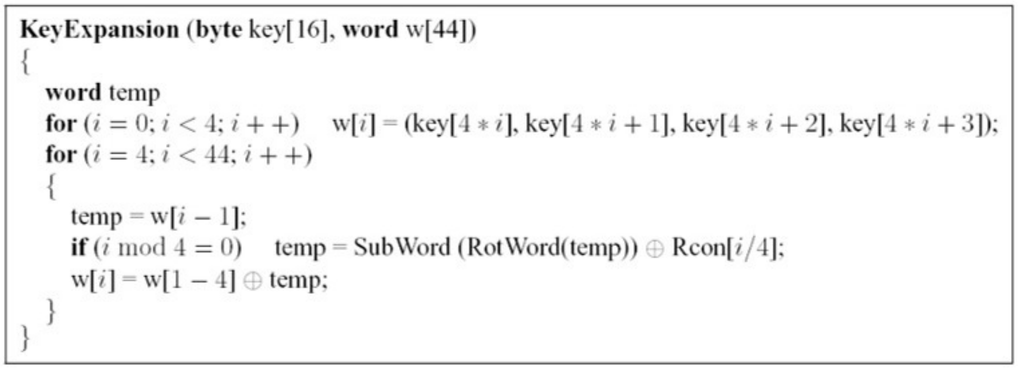
$$\begin{aligned} &\mathbf{A}^{-1}\mathbf{S}' \\ &= \mathbf{A}^{-1}\mathbf{AS} = \mathbf{S} \end{aligned}$$

7.6 Add Round Key Transformation

In this stage (known as AddRoundKey) the 128 bits of **state** are bitwise XORed with the 128 bits of the round key. The operation is viewed as a columnwise operation between the 4 bytes of a **state** column and one word of the round key. This transformation is as simple as possible which helps in efficiency but it also effects every bit of **state**.

7.6.1 AES Key Expansion

The AES key expansion algorithm takes as input a 4-word key and produces a linear array of 44 words. Each round uses 4 of these words as shown in figure 7.2. Each word contains 32 bytes which means each subkey is 128 bits long. Figure 7.7 show pseudocode for generating the expanded key from the actual key.



```
KeyExpansion (byte key[16], word w[44])
{
  word temp
  for ($i = 0; i < 4; i++$)    w[$i$] = (key[$4*i$], key[$4*i+1$], key[$4*i+2$], key[$4*i+3$]);
  for ($i = 4; i < 44; i++$)
  {
    temp = w[$i-1$];
    if ($i \bmod 4 = 0$)    temp = SubWord (RotWord(temp)) $\oplus$ Rcon[$i/4$];
    w[$i$] = w[$1-4$] $\oplus$ temp;
  }
}
```

Figure 7.7: Key expansion pseudocode.

The key is copied into the first four words of the expanded key. The remainder of the expanded key is filled in four words at a time. Each added word **w**[i] depends on the immediately preceding word, **w**[$i-1$], and the word four positions back **w**[$i-4$]. In three out of four cases, a simple XOR is used. For a word whose position in the **w** array is a multiple of 4, a more complex function is used. Figure 7.8 illustrates the generation of the first eight words of the expanded key using the symbol g to represent that complex function. The function g consists of the following subfunctions:

1. **RotWord** performs a one-byte circular left shift on a word. This means that an input word [b_0, b_1, b_2, b_3] is transformed into [b_1, b_2, b_3, b_0].

2. **SubWord** performs a byte substitution on each byte of its input word, using the s-box described earlier.

3. The result of steps 1 and 2 is XORed with round constant, Rcon[j].

The round constant is a word in which the three rightmost bytes are always 0. Thus the effect of an XOR of a word with Rcon is to only perform an XOR on the leftmost byte of the word. The round constant is different for each round and is defined as Rcon[j] = (RC[J], 0,0,0), with RC[1]= 1, RC[j]= 2$\bullet$ RC[$j-1$] and with multiplication defined over the field GF(2^8).

The key expansion was designed to be resistant to known cryptanalytic attacks. The inclusion of a round-dependent round constant eliminates the symmetry, or similarity, between the way in which round keys are generated in different rounds.

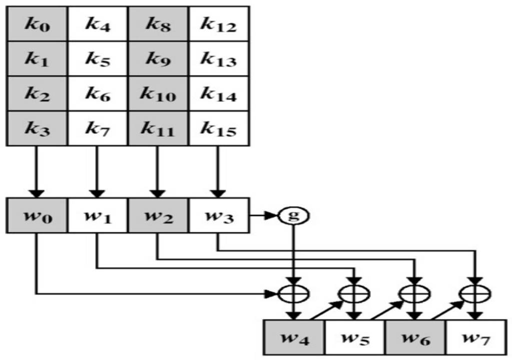

k_0 k_4 k_8 k_{12}
k_1 k_5 k_9 k_{13}
k_2 k_6 k_{10} k_{14}
k_3 k_7 k_{11} k_{15}
w_0 w_1 w_2 w_3
g
w_4 w_5 w_6 w_7

Figure 7.8: AES key expansion.

Figure 7.9 give a summary of each of the rounds. The ShiftRows column is depicted here as a linear shift which gives a better idea how this section helps in the encryption.

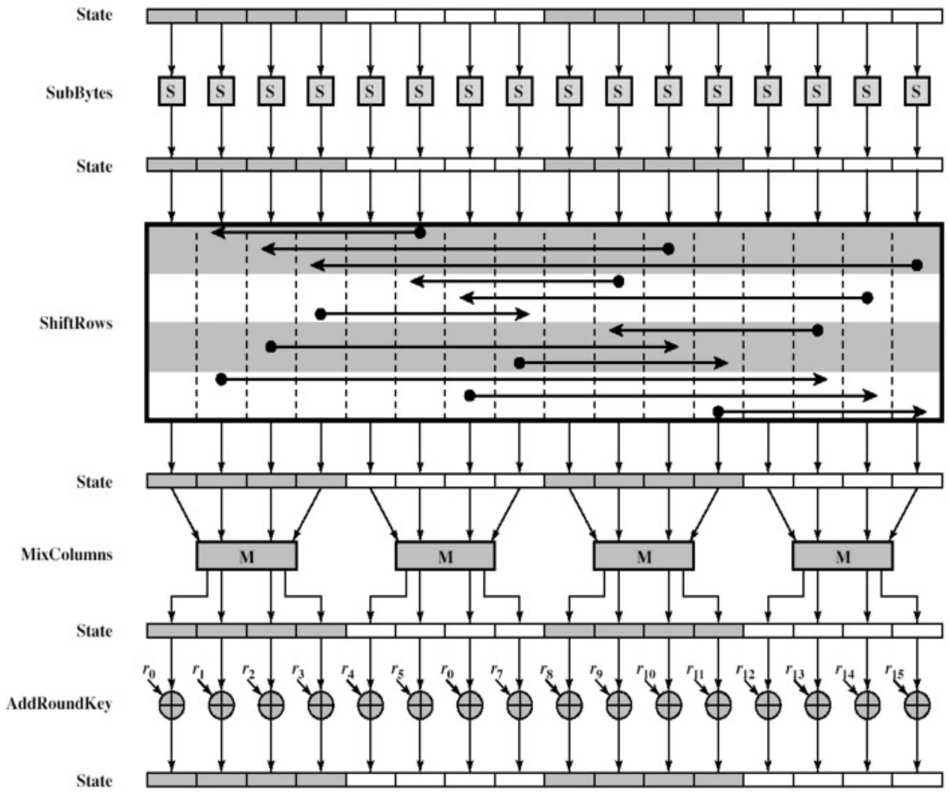

State
SubBytes
S
State
ShiftRows
State
MixColumns
M
State
AddRoundKey
r_0 r_1 r_2 r_3 r_4 r_5 r_0 r_7 r_8 r_9 r_{10} r_{11} r_{12} r_{13} r_{14} r_{15}
State

Figure 7.9: AES encryption round.

7.6.2 Equivalent Inverse Cipher

As can be seen from figure 7.1 the decryption ciphers is not identical to the encryption ciphers. However the form of the key schedules is the same for both. This has the disadvantage that two separate software or firmware modules are needed for applications that require both encryption and decryption. As well as that, decryption is slightly less efficient to implement. However, encryption was deemed more important than decryption for two reasons:

1. For the CFB and OFB cipher mode (which we have seen before but will study in more detail next) only encryption is used.
2. As with any block cipher, AES can be used to construct a message authentication code (to be described later), and for this only encryption is used.

However, if desired it is possible to create an **equivalent inverse cipher**. This means that decryption has the same structure as the encryption algorithms. However, to achieve this, a change of key schedule is needed. We will not be concerned with this alternate form but you should be aware that is exists.

7.7 Block Cipher Modes of Operation

We have seen previously that five modes of operation are used when applying block ciphers in a variety of applications. This section will give a more detailed view of how these modes operate.

7.7.1 Electronic Codebook Mode (ECB)

This first mode is the simplest of all five modes. Figure 7.10 shows the scheme where it can be seen that a block of plaintext (which is the same size in each case) is encrypted with the same key K. The term *codebook* is used because, for a given key, there is a unique ciphertext for every block of plaintext. Therefore we can imagine a gigantic codebook in which there is an entry for every possible plaintext pattern showing its corresponding ciphertext. If the message is longer than the block length then the procedure is to break the message into blocks of the required length padding the last block if necessary. As with encryption, decryption is performed one block at a time, always using the same key.

The ECB method is ideal for small amounts of data such as an encryption key however for larger messages if the same plaintext block appears more than once then the same ciphertext is produced. This may assist an attacker.

7.7.2 Cipher Block Chaining (CBC) Mode

We would like that same plaintext blocks produce different ciphertext blocks. Cipher Block Chaining (see figure 7.11) allows this by XORing each plaintext with the ci-

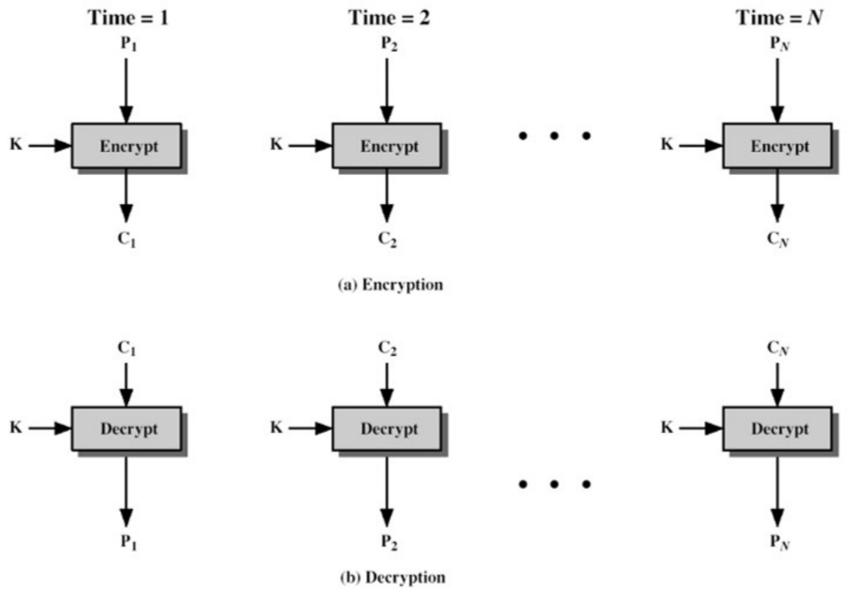


Figure 7.10: Electronic Codebook Mode (ECB)

phertext from the previous round (the first round using an Initialisation Vector (IV)). As before, the same key is used for each block. Decryption works as shown in the figure because of the properties of the XOR operation, i.e. IV $\oplus$ IV $\oplus P = P$ where IV is the Initialisation Vector and P is the plaintext. Obviously the IV needs to be known by both sender and received and it should be kept secret along with the key for maximum security.

7.7.3 Cipher Feedback (CFB) Mode

The Cipher Feedback and Output Feedback allows a block cipher to be converted into a stream cipher. This eliminates the need to pad a message to be an integral number of blocks. It also can operate in real time.

Figure 7.12 shows the CFB scheme. In this figure it assumed that the unit of transmission is s bits; a common value is $s = 8$. As with CBC, the units of plaintext are chained together, so that the ciphertext of any plaintext unit is a function of all the preceding plaintext (which is split into s bit segments). The input to the encryption function is a shift register equal in length to the block cipher of the algorithm (although the diagram shows 64 bits, which is block size used by DES, this can be extended to other block sizes such as the 128 bits of AES). This is initially set to some Initialisation Vector (IV). The leftmost s bits of the output of the encryption function are XORed with the first segment of plaintext P_1 (also s bits) to produce the first unit of ciphertext C_1 which is then transmitted. In addition, the contents of the shift register are shifted left by s bits and C_1 is placed in the rightmost (least significant) s bits of the shift register. This process continues until all plaintext units have been encrypted. Decryption is similar.

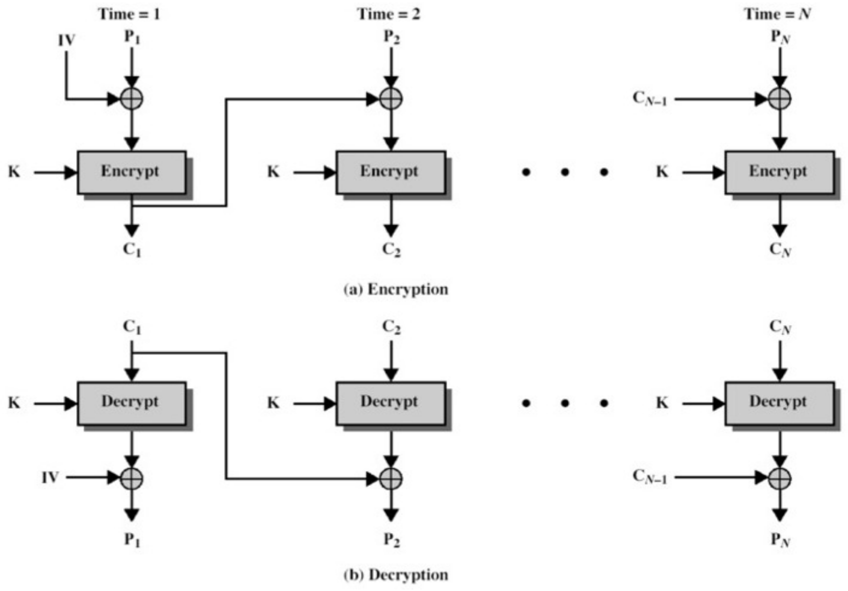


Figure 7.11: Cipher Block Chaining (CBC) Mode

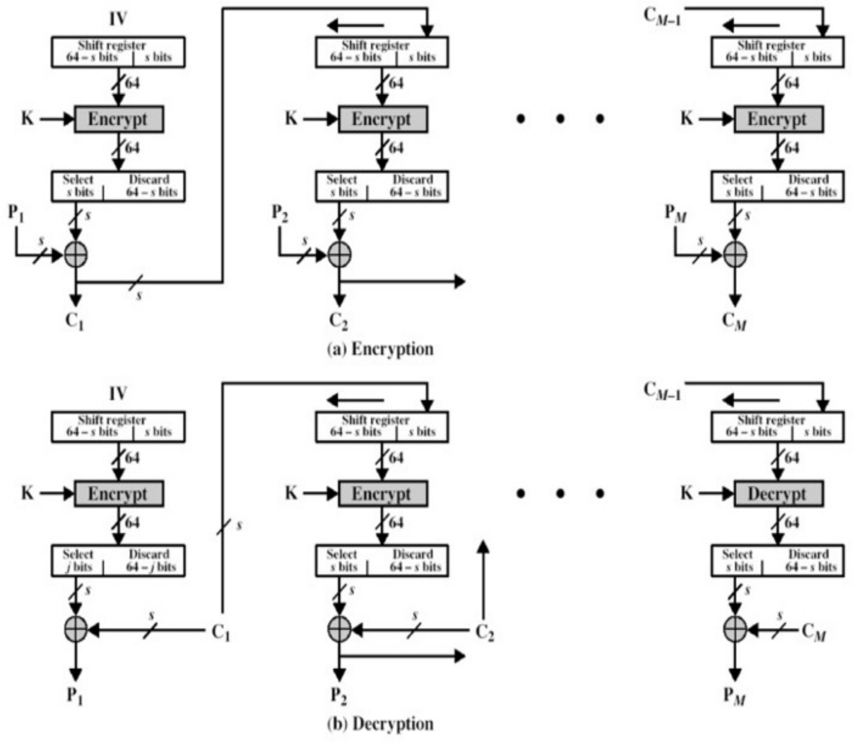


Figure 7.12: Cipher Feedback (CFB)

7.7.4 Output Feedback (OFB) Mode

The Output Feedback Mode is similar in structure to that of CFB, as seen in figure 7.13. As can be seen, it is the output of the encryption function that is fed back to the shift register in OFB, whereas in CFB the ciphertext unit is fed back to the shift register. One advantage of the OFB method is that bit errors in transmission do not propagate. For example, if a bit error occurs in C_1 only the recovered value of P_1 is affected; subsequent plaintext units are not corrupted. With CFB, C_1 also serves as input to the shift register and therefore causes additional corruption downstream.

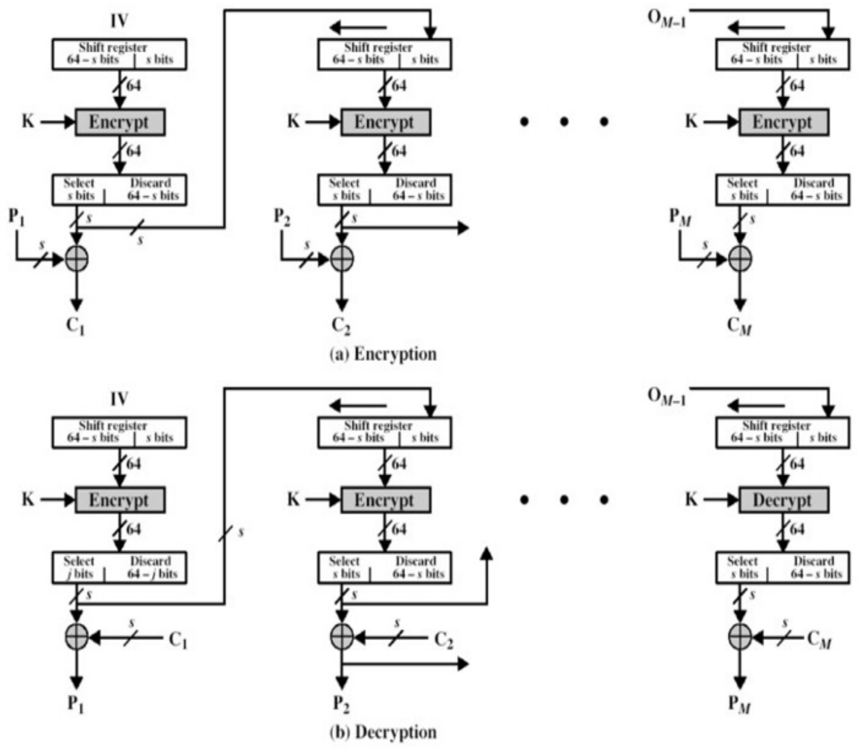


Figure 7.13: Output Feedback (OFB)

7.7.5 Counter (CTR)

This is a newer mode that was not listed initially with the above four. Interest in this mode has increased a good deal lately. A counter, equal to the plaintext block size is used. The only requirement stated in the standard is that the counter value must be different for each plaintext block that is encrypted. Typically, this counter is initialised to some value and then incremented by 1 for each subsequent block (modulo 2^b where b is the block size). For encryption, the counter is encrypted and then XORed with the plaintext to produce the ciphertext block; there is no chaining. For decryption, the same sequence of counter values is used, with each encrypted counter XORed with a ciphertext block to recover the corresponding plaintext block. This mode contains a number of advantages including hardware efficiency, software efficiency, provable security (in the sense that it is at least as secure as the other modes discussed) and simplicity.

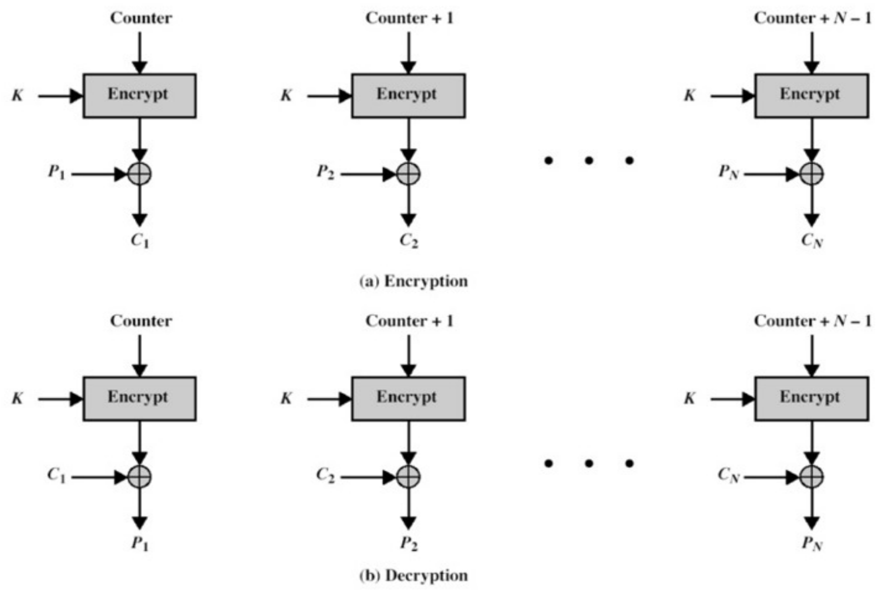


Figure 7.14: Counter (CTR) Mode