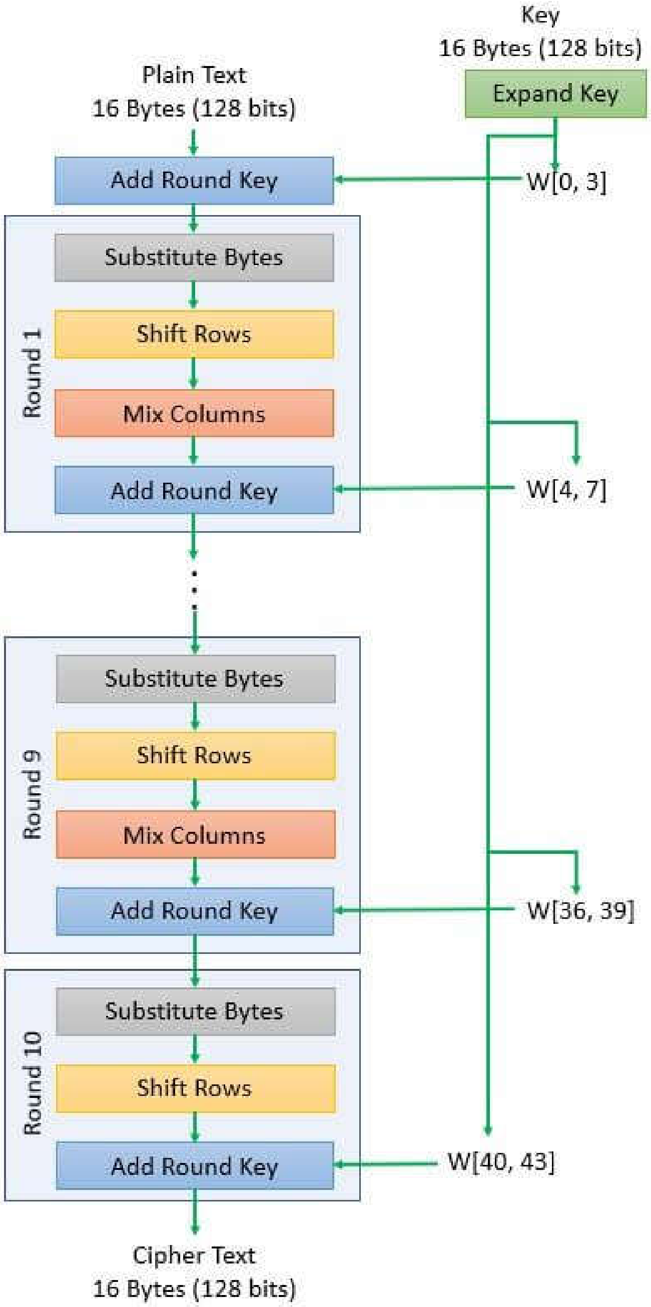
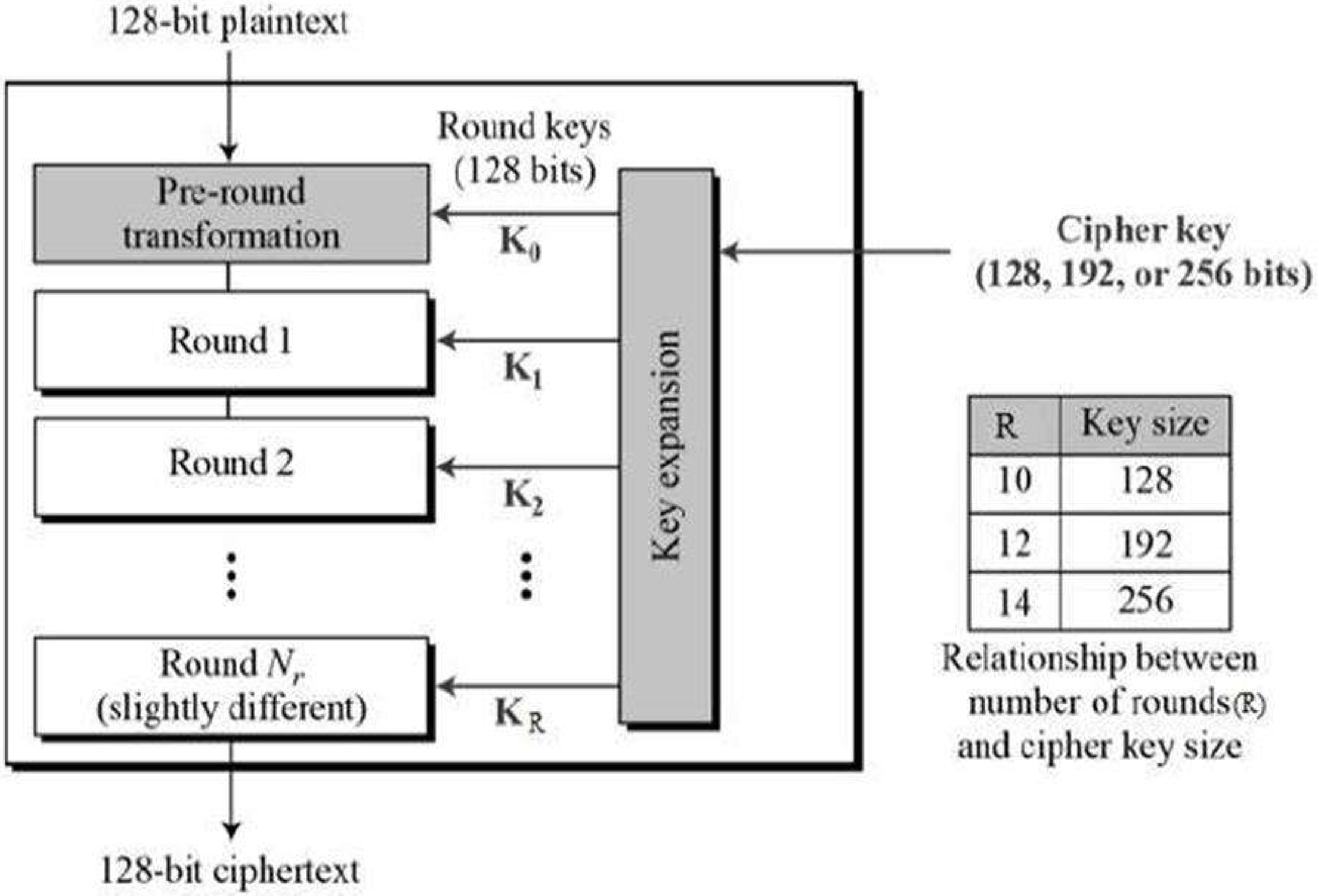
Outline

- AES Key Schedule
- Block Cipher Modes of Operations
- Stream Cipher

AES Encryption



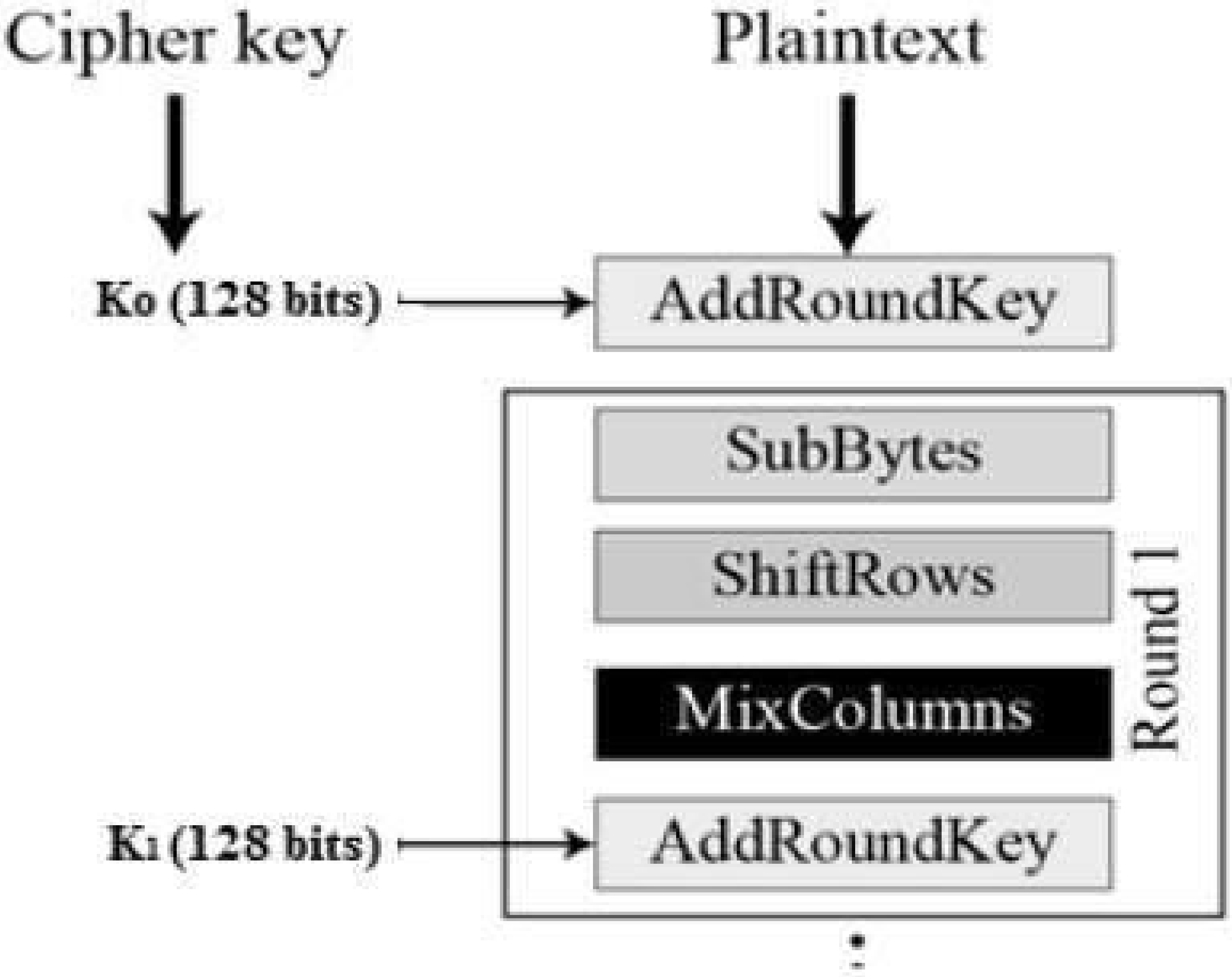
AES Encryption



R	Key size
10	128
12	192
14	256

Relationship between number of rounds(R) and cipher key size

AES Encryption



AES Encryption

The AES algorithm can be broken into three phases: the initial round, the main rounds, and the final round. All of the phases use the same sub-operations in different combinations as follows:

Initial Round
AddRoundKey
Main Rounds (1,2…Nr-1)
SubBytes
ShiftRows
MixColumns
AddRoundKey
Final Round (Nr)
SubBytes
ShiftRows
AddRoundKey

AES Encryption

Plaintext in English: Two One Nine Two (16 ASCII characters, 1 byte each)

Translation into Hex:

T	w	o		O	n	e		N	i	n	e		T	w	o
54	77	6F	20	4F	6E	65	20	4E	69	6E	65	20	54	77	6F

Plaintext in Hex (128 bits): 54 77 6F 20 4F 6E 65 20 4E 69 6E 65 20 54 77 6F

Key in English: Thats my Kung Fu (16 ASCII characters, 1 byte each)

Translation into Hex:

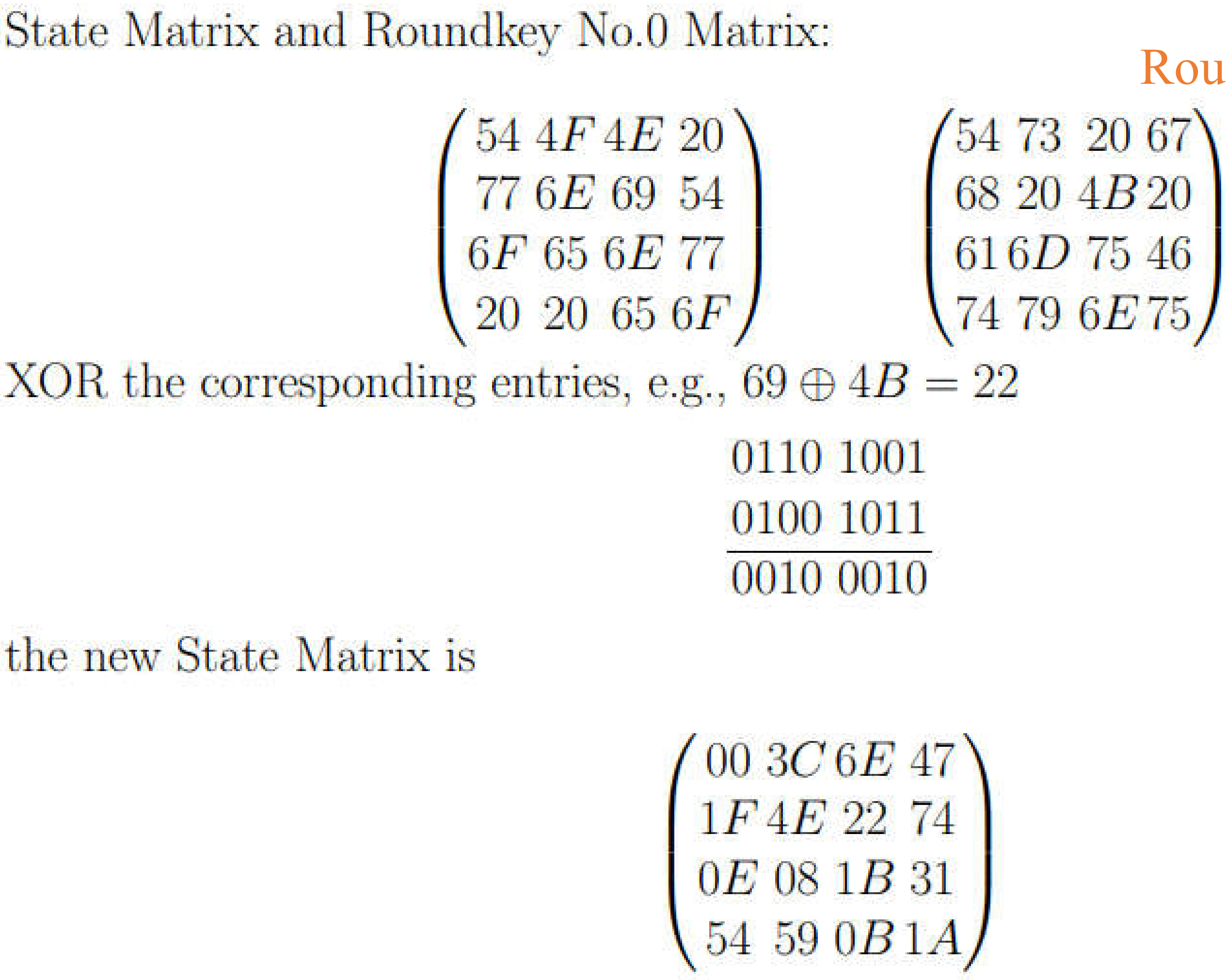
T	h	a	t	s		m	y		K	u	n	g		F	u
54	68	61	74	73	20	6D	79	20	4B	75	6E	67	20	46	75

Key in Hex (128 bits): 54 68 61 74 73 20 6D 79 20 4B 75 6E 67 20 46 75

AES Encryption

Initial Round

AddRoundKey



State Matrix and Roundkey No.0 Matrix:

Round 0 Key

$$\begin{pmatrix} 54 & 4F & 4E & 20 \\ 77 & 6E & 69 & 54 \\ 6F & 65 & 6E & 77 \\ 20 & 20 & 65 & 6F \end{pmatrix} \qquad \begin{pmatrix} 54 & 73 & 20 & 67 \\ 68 & 20 & 4B & 20 \\ 61 & 6D & 75 & 46 \\ 74 & 79 & 6E & 75 \end{pmatrix}$$

XOR the corresponding entries, e.g., $69 \oplus 4B = 22$

$$\begin{array}{r} 0110\ 1001 \\ 0100\ 1011 \\ \hline 0010\ 0010 \end{array}$$

the new State Matrix is

$$\begin{pmatrix} 00 & 3C & 6E & 47 \\ 1F & 4E & 22 & 74 \\ 0E & 08 & 1B & 31 \\ 54 & 59 & 0B & 1A \end{pmatrix}$$

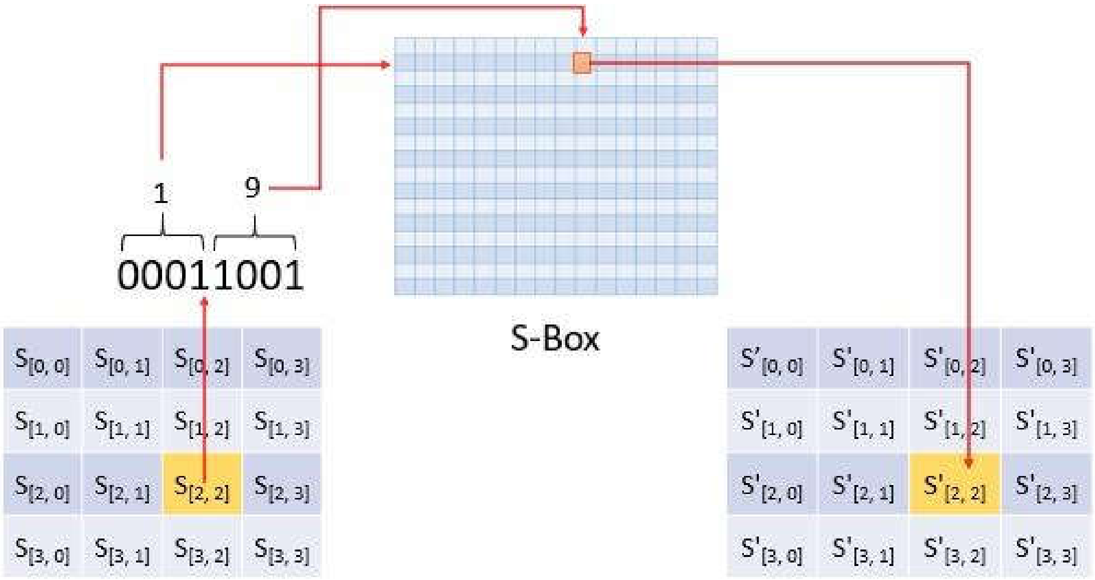
AES Encryption

Main Rounds (1,2…Nr-1)

SubBytes
ShiftRows
MixColumns
AddRoundKey

AES Encryption: Substitute Bytes

- The input to Substitute Byte is a 4X4 state matrix of 16 bytes where each element of the matrix is of 1 byte.
- Now AES has defined a <u>16X16</u> matrix namely S-box which contains a permutation of 256 8-bit values.



$S_{[0,0]}$	$S_{[0,1]}$	$S_{[0,2]}$	$S_{[0,3]}$
$S_{[1,0]}$	$S_{[1,1]}$	$S_{[1,2]}$	$S_{[1,3]}$
$S_{[2,0]}$	$S_{[2,1]}$	$S_{[2,2]}$	$S_{[2,3]}$
$S_{[3,0]}$	$S_{[3,1]}$	$S_{[3,2]}$	$S_{[3,3]}$

$S'_{[0,0]}$	$S'_{[0,1]}$	$S'_{[0,2]}$	$S'_{[0,3]}$
$S'_{[1,0]}$	$S'_{[1,1]}$	$S'_{[1,2]}$	$S'_{[1,3]}$
$S'_{[2,0]}$	$S'_{[2,1]}$	$S'_{[2,2]}$	$S'_{[2,3]}$
$S'_{[3,0]}$	$S'_{[3,1]}$	$S'_{[3,2]}$	$S'_{[3,3]}$

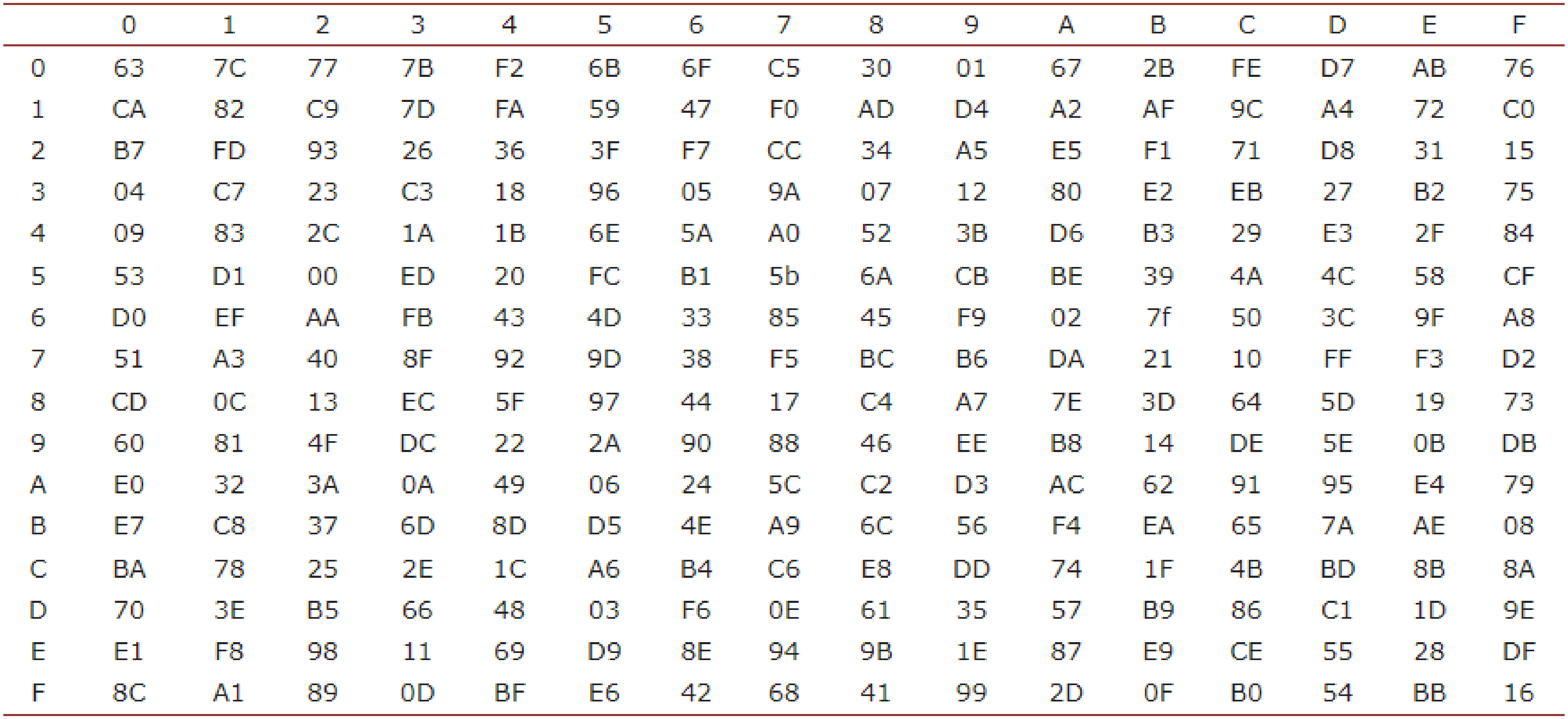
Substitute Bytes

AES Encryption: SubBytes

- A nonlinear substitution step where each entry (byte) of the current state matrix is substituted by a corresponding entry in the AES S-Box. For instance: byte (6E) is substituted by the entry of the S-Box in row 6 and column E, i.e., by (9F). (The byte input is broken into two 4-bit halves. The first half determines the row and the second half determines the column).

$$\text{state} = \begin{pmatrix} 00 & 3C & 6E & 47 \\ 1F & 4E & 22 & 74 \\ 0E & 08 & 1B & 31 \\ 54 & 59 & 0B & 1A \end{pmatrix} \Rightarrow \text{S\_box(State)} = \begin{pmatrix} 63 & EB & 9F & A0 \\ C0 & 2F & 93 & 92 \\ AB & 30 & AF & C7 \\ 20 & CB & 2B & A2 \end{pmatrix}$$

AES Encryption: SubBytes, 16x16 S-Box

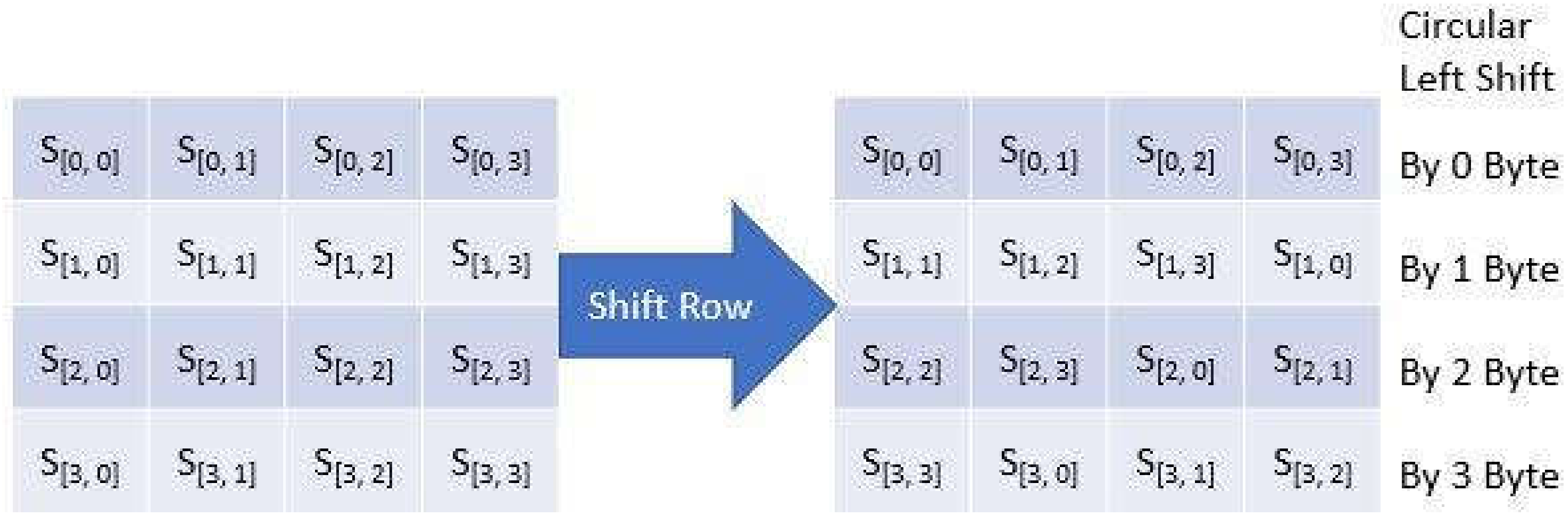


	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	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
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

AES Encryption: ShiftRows

- The input to Shift Row function is a 4X4 state matrix forwarded from the Substitute Bytes function. The Shift row performs the **circular left shift** on the rows of the matrix.
- On the first row, the circular left shift is performed by 0 bytes.
- On the second row, a circular left shift is performed by 1 byte.
- On the third row, the circular left shift is performed by 2 bytes.
- On the fourth row of the input state matrix, the circular let shift is performed by 3 bytes.

AES Encryption: ShiftRows



$S_{[0,0]}$	$S_{[0,1]}$	$S_{[0,2]}$	$S_{[0,3]}$
$S_{[1,0]}$	$S_{[1,1]}$	$S_{[1,2]}$	$S_{[1,3]}$
$S_{[2,0]}$	$S_{[2,1]}$	$S_{[2,2]}$	$S_{[2,3]}$
$S_{[3,0]}$	$S_{[3,1]}$	$S_{[3,2]}$	$S_{[3,3]}$

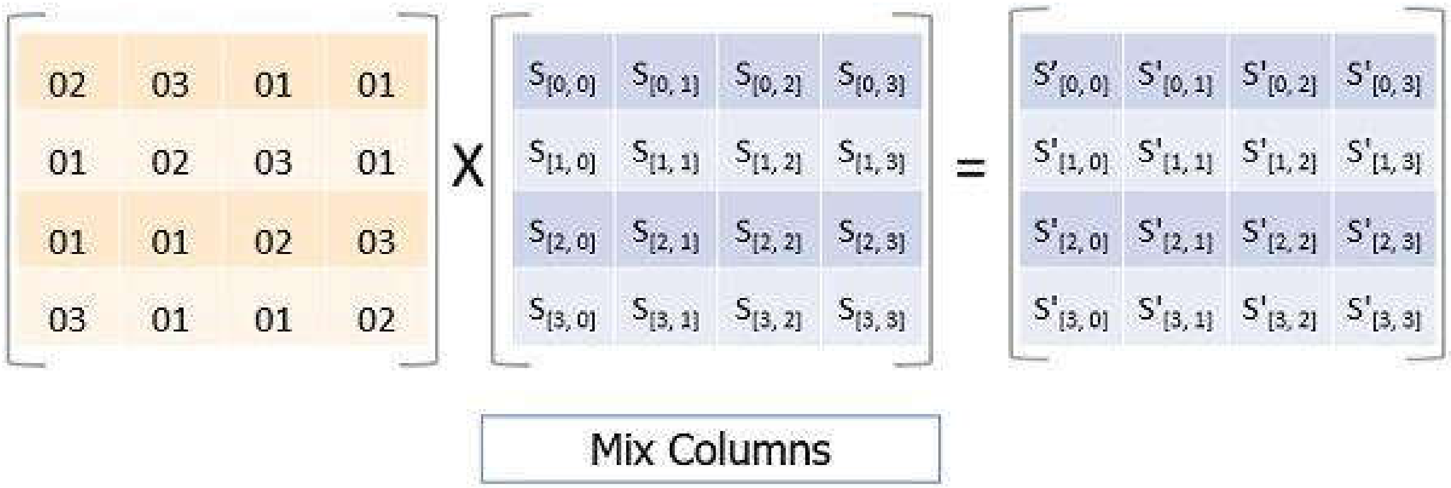
				Circular Left Shift
$S_{[0,0]}$	$S_{[0,1]}$	$S_{[0,2]}$	$S_{[0,3]}$	By 0 Byte
$S_{[1,1]}$	$S_{[1,2]}$	$S_{[1,3]}$	$S_{[1,0]}$	By 1 Byte
$S_{[2,2]}$	$S_{[2,3]}$	$S_{[2,0]}$	$S_{[2,1]}$	By 2 Byte
$S_{[3,3]}$	$S_{[3,0]}$	$S_{[3,1]}$	$S_{[3,2]}$	By 3 Byte

Shift Rows

$$\begin{pmatrix} 63 & EB & 9F & A0 \\ C0 & 2F & 93 & 92 \\ AB & 30 & AF & C7 \\ 20 & CB & 2B & A2 \end{pmatrix} \Rightarrow \begin{pmatrix} 63 & EB & 9F & A0 \\ 2F & 93 & 92 & C0 \\ AF & C7 & AB & 30 \\ A2 & 20 & CB & 2B \end{pmatrix}$$

AES Encryption: MixColumns

- The input 4X4 state matrix is multiplied with **a constant predefined matrix**.



$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} X \begin{bmatrix} S_{[0,0]} & S_{[0,1]} & S_{[0,2]} & S_{[0,3]} \\ S_{[1,0]} & S_{[1,1]} & S_{[1,2]} & S_{[1,3]} \\ S_{[2,0]} & S_{[2,1]} & S_{[2,2]} & S_{[2,3]} \\ S_{[3,0]} & S_{[3,1]} & S_{[3,2]} & S_{[3,3]} \end{bmatrix} = \begin{bmatrix} S'_{[0,0]} & S'_{[0,1]} & S'_{[0,2]} & S'_{[0,3]} \\ S'_{[1,0]} & S'_{[1,1]} & S'_{[1,2]} & S'_{[1,3]} \\ S'_{[2,0]} & S'_{[2,1]} & S'_{[2,2]} & S'_{[2,3]} \\ S'_{[3,0]} & S'_{[3,1]} & S'_{[3,2]} & S'_{[3,3]} \end{bmatrix}$$

Mix Columns

AES Encryption: MixColumns

- Unlike standard matrix multiplication, MixColumns performs matrix multiplication as per Galois Field (2^8).

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

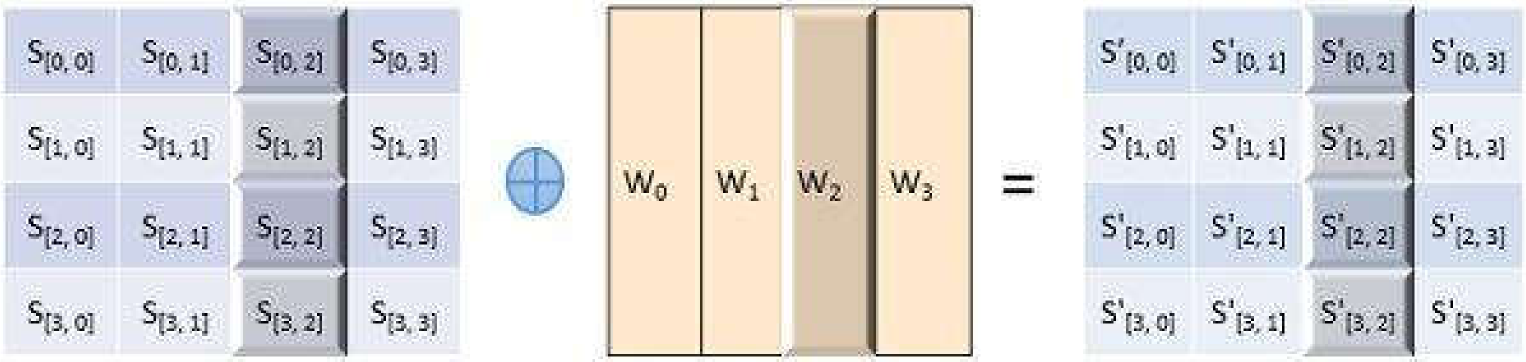
$$\begin{aligned} s'_{0,j} &= (2 \bullet s_{0,j}) \oplus (3 \bullet s_{1,j}) \oplus s_{2,j} \oplus s_{3,j} \\ s'_{1,j} &= s_{0,j} \oplus (2 \bullet s_{1,j}) \oplus (3 \bullet s_{2,j}) \oplus s_{3,j} \\ s'_{2,j} &= s_{0,j} \oplus s_{1,j} \oplus (2 \bullet s_{2,j}) \oplus (3 \bullet s_{3,j}) \\ s'_{3,j}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

AES Encryption: MixColumns

$$\begin{pmatrix} 02\,03\,01\,01 \\ 01\,02\,03\,01 \\ 01\,01\,02\,03 \\ 03\,01\,01\,02 \end{pmatrix} \begin{pmatrix} 63 & EB & 9F & A0 \\ 2F & 93 & 92 & C0 \\ AF & C7 & AB & 30 \\ A2 & 20 & CB & 2B \end{pmatrix} = \begin{pmatrix} BA & 84 & E8 & 1B \\ 75 & A4 & 8D & 40 \\ F4 & 8D & 06 & 7D \\ 7A & 32 & 0E & 5D \end{pmatrix}$$

AES Encryption: AddRoundKey

- In the Add Round Key function, the input state matrix is XORed with the 4-words unique key. In each round the key used to XOR with state matrix is distinct.
- Add Round Key function is a column-wise function, a 4-byte state matrix column is XORed with a 4-byte word of a key. It can also be taken as byte-level function.


$S_{[0,0]}$ $S_{[0,1]}$ $S_{[0,2]}$ $S_{[0,3]}$
$S_{[1,0]}$ $S_{[1,1]}$ $S_{[1,2]}$ $S_{[1,3]}$
$S_{[2,0]}$ $S_{[2,1]}$ $S_{[2,2]}$ $S_{[2,3]}$
$S_{[3,0]}$ $S_{[3,1]}$ $S_{[3,2]}$ $S_{[3,3]}$
$\oplus$
W_0 W_1 W_2 W_3
=
$S'_{[0,0]}$ $S'_{[0,1]}$ $S'_{[0,2]}$ $S'_{[0,3]}$
$S'_{[1,0]}$ $S'_{[1,1]}$ $S'_{[1,2]}$ $S'_{[1,3]}$
$S'_{[2,0]}$ $S'_{[2,1]}$ $S'_{[2,2]}$ $S'_{[2,3]}$
$S'_{[3,0]}$ $S'_{[3,1]}$ $S'_{[3,2]}$ $S'_{[3,3]}$

Add Round Key

AES Encryption: AddRoundKey

State Matrix and Roundkey No.1 Matrix:

$$\begin{pmatrix} BA & 84 & E8 & 1B \\ 75 & A4 & 8D & 40 \\ F4 & 8D & 06 & 7D \\ 7A & 32 & 0E & 5D \end{pmatrix} \qquad \begin{pmatrix} E2 & 91 & B1 & D6 \\ 32 & 12 & 59 & 79 \\ FC & 91 & E4 & A2 \\ F1 & 88 & E6 & 93 \end{pmatrix}$$

XOR yields new State Matrix

$$\begin{pmatrix} 58 & 15 & 59 & CD \\ 47 & B6 & D4 & 39 \\ 08 & 1C & E2 & DF \\ 8B & BA & E8 & CE \end{pmatrix}$$

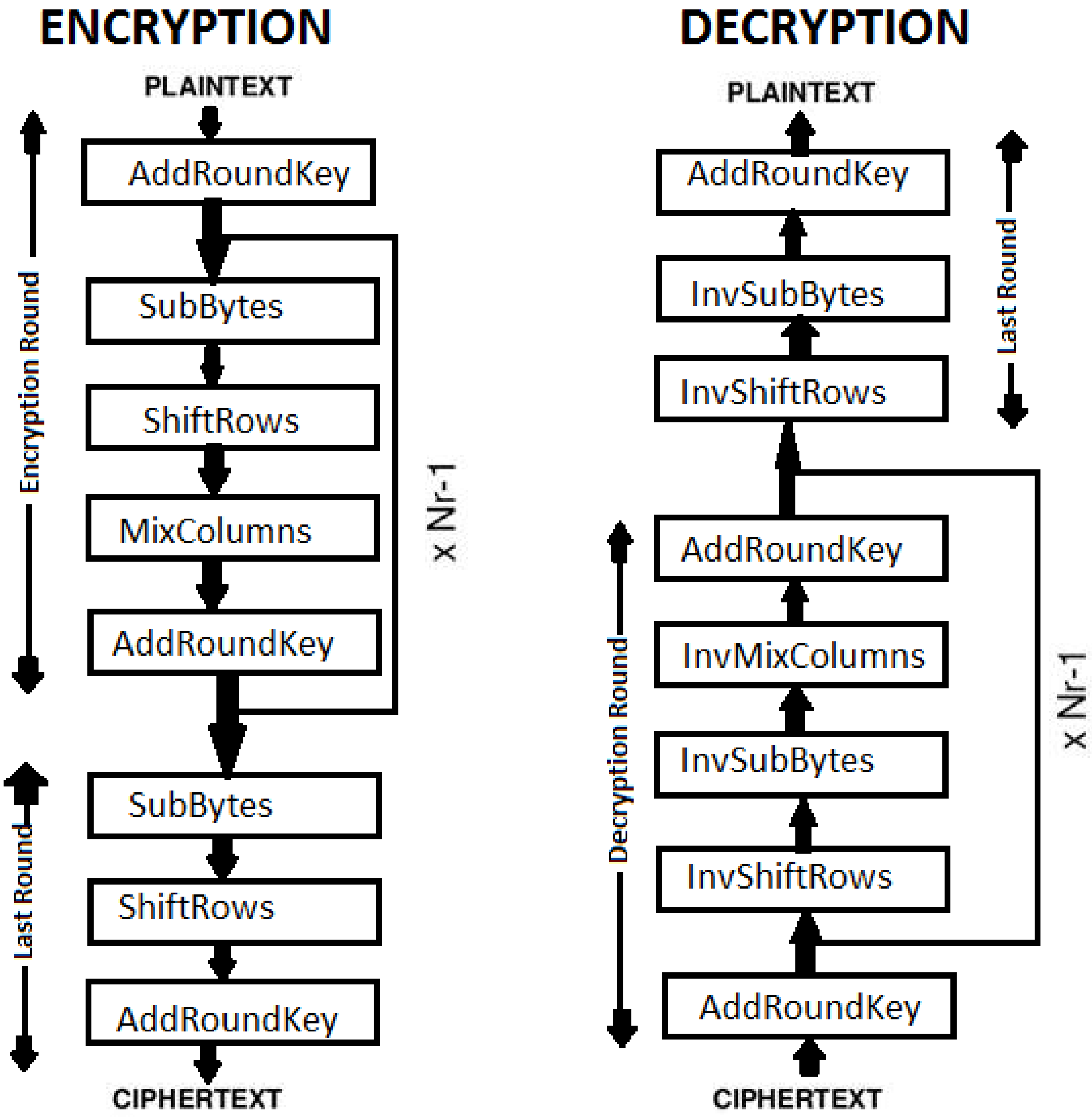
AES output after Round 1: 58 47 08 8B 15 B6 1C BA 59 D4 E2 E8 CD 39 DF CE

AES Decryption

- AES Encryption and AES Decryption process are the same and it also starts with the Add Round Key Function. The 16-byte cipher text in the form of 4X4 state matrix is XORed with the unique 4-word key.

- The key sequence in encryption is **reversed** during the decryption.
- And all the other round functions are also **inversed** in the decryption process to retrieve the original 16-byte plain text block.

AES Decryption



ENCRYPTION

PLAINTEXT

AddRoundKey

SubBytes

ShiftRows

MixColumns

AddRoundKey

SubBytes

ShiftRows

AddRoundKey

CIPHERTEXT

Encryption Round

Last Round

x Nr-1

DECRYPTION

PLAINTEXT

AddRoundKey

InvSubBytes

InvShiftRows

AddRoundKey

InvMixColumns

InvSubBytes

InvShiftRows

AddRoundKey

CIPHERTEXT

Last Round

Decryption Round

x Nr-1

AES Decryption

AddRoundKey:

Add Roundkey transformation is identical to the forward add round key transformation, because the XOR operation is its own inverse.

Inverse SubBytes:

This operation can be performed using the inverse S-Box. It is read identically to the S-Box matrix.

InvShiftRows:

Inverse Shift Rows performs the circular shifts in the opposite direction for each of the last three rows, with a one-byte **circular right shift** for the second row, and so on.

InvMixColumns:

The inverse mix column transformation is defined by the following matrix multiplication in Galois Field (2^8).

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

AES Key Schedule / Expansion

- The AES Key Expansion algorithm is used to derive the 128-bit round key for each round from the original 128-bit encryption key.
- In the same manner as the 128-bit input block is arranged in the form of a state array, the algorithm first arranges the 16 bytes of the encryption key in the form of a 4×4 array of bytes.

$$\begin{bmatrix} k_0 & k_4 & k_8 & k_{12} \\ k_1 & k_5 & k_9 & k_{13} \\ k_2 & k_6 & k_{10} & k_{14} \\ k_3 & k_7 & k_{11} & k_{15} \end{bmatrix}$$

$$\Downarrow$$

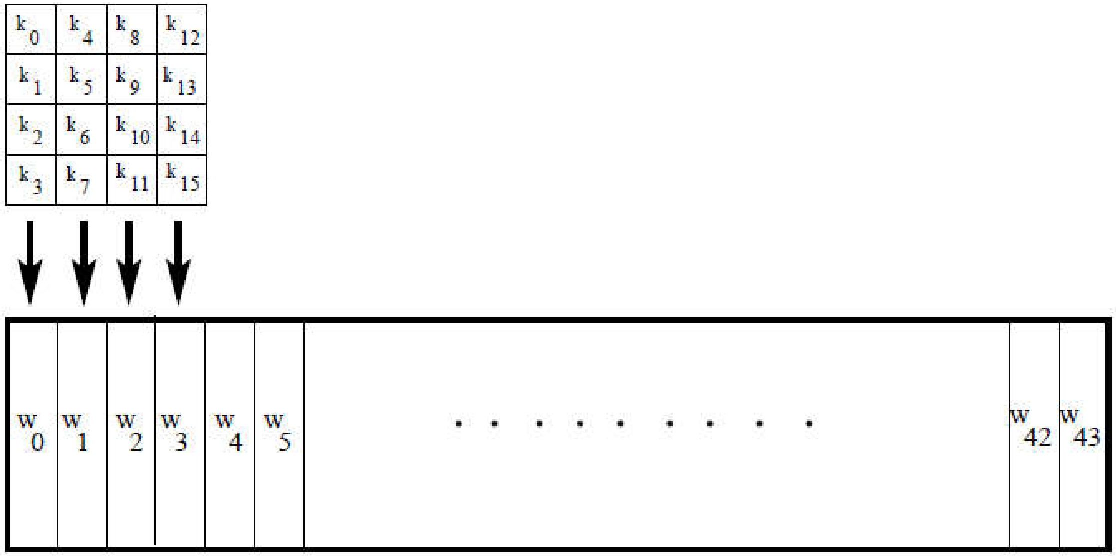
$$\begin{bmatrix} w_0 & w_1 & w_2 & w_3 \end{bmatrix}$$

AES Key Schedule / Expansion

- The first four bytes of the encryption key constitute the word w_0, the next four bytes the word w_1, and so on.

- The algorithm subsequently expands the words $[w_0, w_1, w_2, w_3]$ into a 44-word **key schedule** that can be labeled

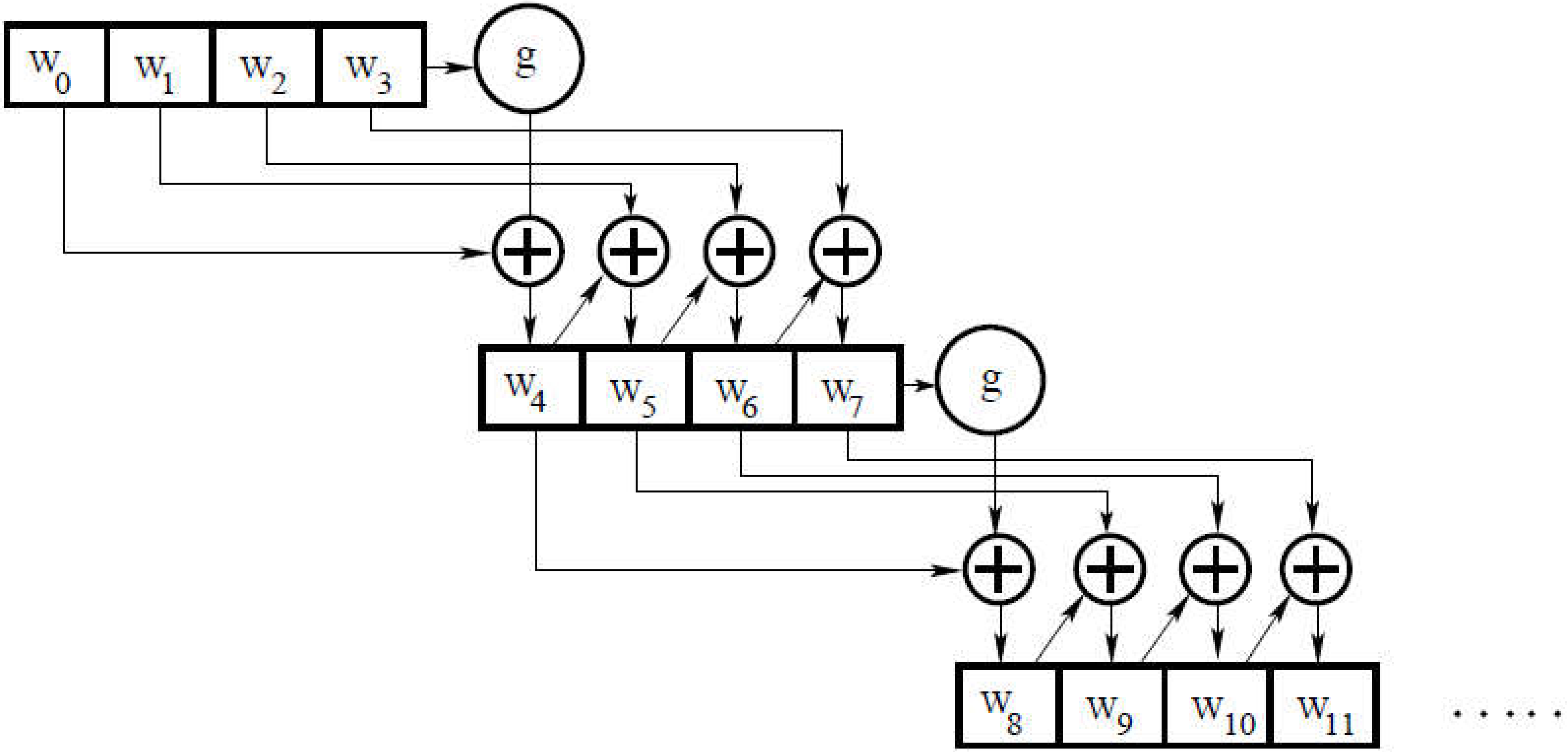
$$w_0,\ w_1,\ w_2,\ w_3, \ldots\ldots\ldots\ldots\ldots,\ w_{43}$$



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}

AES Key Schedule / Expansion

Now comes the difficult part: How does the **Key Expansion Algorithm** expand four words w_0, w_1, w_2, w_3 into the 44 words $w_0, w_1, w_2, w_3, w_4, w_5, \ldots\ldots\ldots, w_{43}$?



AES Key Schedule / Expansion

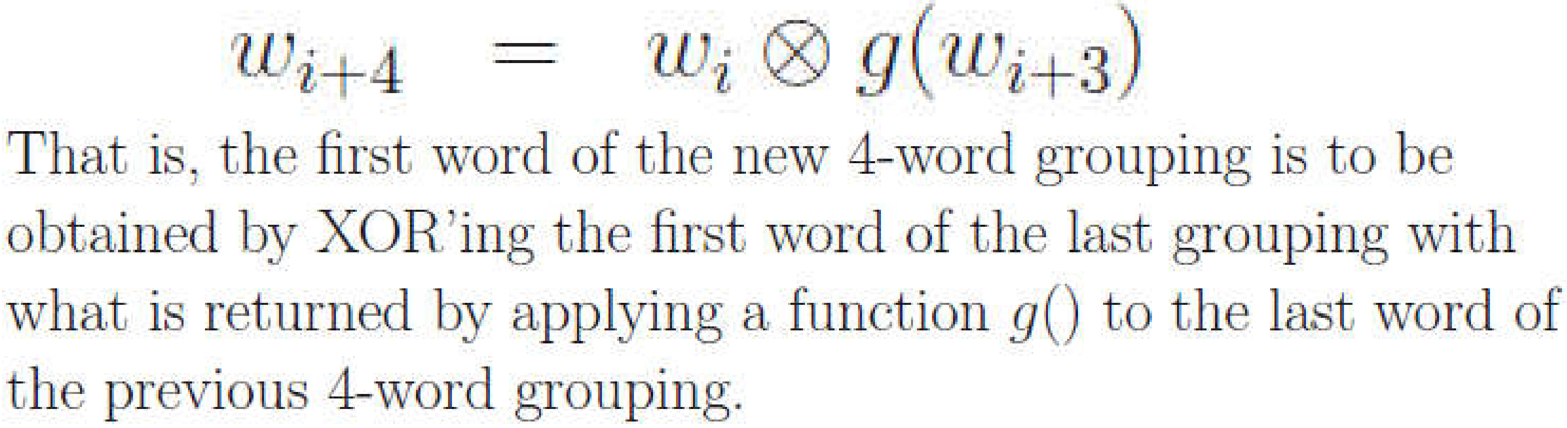
Now we need to determine the words

$$w_{i+4} \quad w_{i+5} \quad w_{i+6} \quad w_{i+7}$$

from the words $w_i \quad w_{i+1} \quad w_{i+2} \quad w_{i+3}$.

$$\begin{aligned}
w_{i+5} &= w_{i+4} \otimes w_{i+1} \\
w_{i+6} &= w_{i+5} \otimes w_{i+2} \\
w_{i+7} &= w_{i+6} \otimes w_{i+3}
\end{aligned}$$

AES Key Schedule / Expansion



$$w_{i+4} \quad = \quad w_i \otimes g(w_{i+3})$$

That is, the first word of the new 4-word grouping is to be obtained by XOR'ing the first word of the last grouping with what is returned by applying a function $g()$ to the last word of the previous 4-word grouping.

AES Key Schedule / Expansion

- The function $g()$ consists of the following three steps:
  - Perform a one-byte left circular rotation on the argument 4-byte word.
  - Perform a byte substitution for each byte of the word returned by the previous step by using the same 16 × 16 lookup table as used in the **SubBytes** step of the encryption rounds.
  - XOR the bytes obtained from the previous step with what is known as a **round constant**. The **round constant** is a word whose three rightmost bytes are always zero. Therefore, XOR'ing with the round constant amounts to XOR'ing with just its leftmost byte.

AES Key Schedule / Expansion

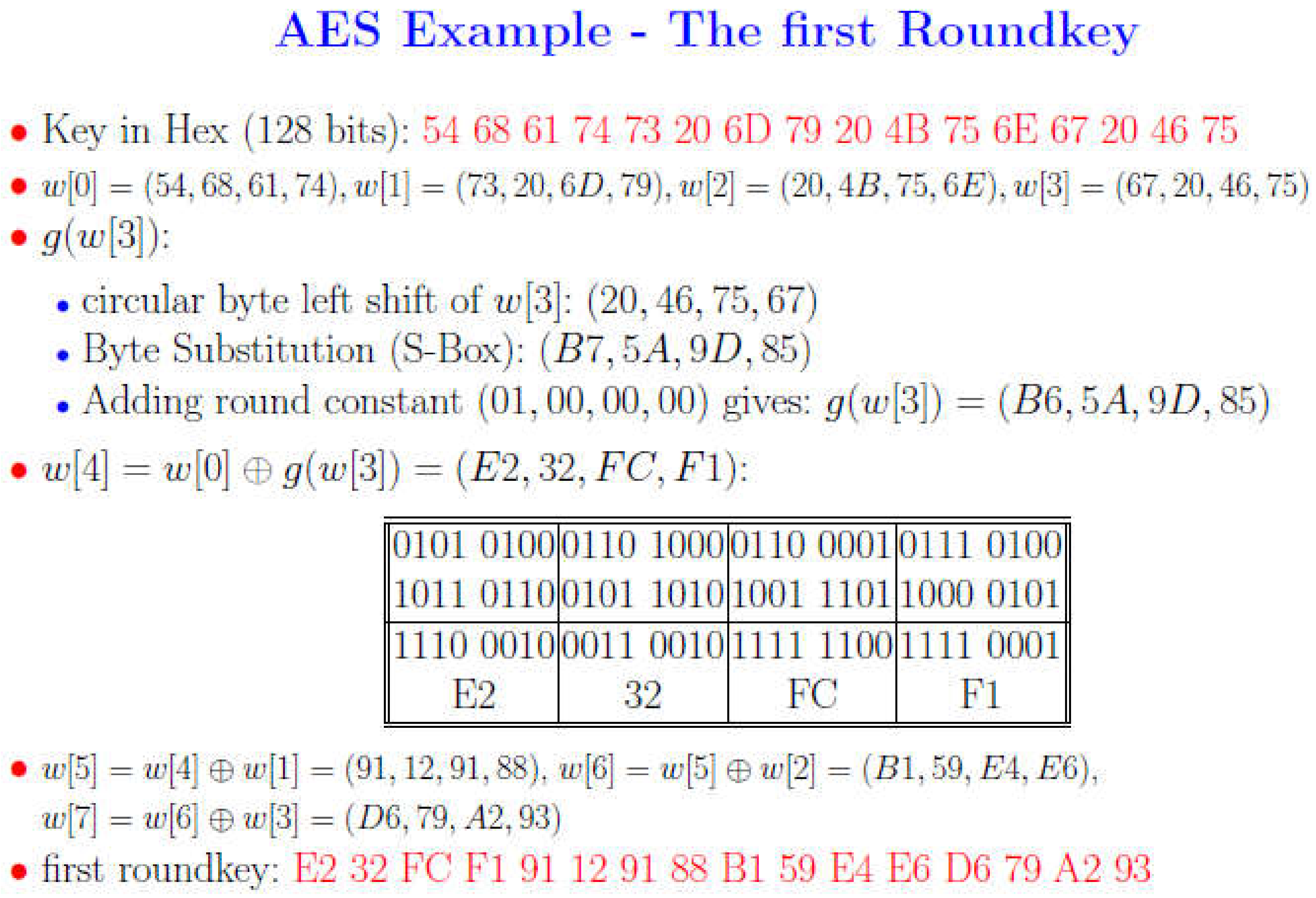
- The **round constant** for the i^{th} round is denoted $Rcon[i]$. Since, by specification, the three rightmost bytes of the round constant are zero, we can write it as shown below. The left hand side of the equation below stands for the round constant to be used in the i^{th} round. The right hand side of the equation says that the rightmost three bytes of the round constant are zero.

$$Rcon[i] \quad = \quad (RC[i],\ 0\text{x}00,\ 0\text{x}00,\ 0\text{x}00)$$

The only non-zero byte in the round constants, $RC[i]$, obeys the following recursion:

$$\begin{aligned} RC[1] &\quad = \quad 0\text{x}01 \\ RC[j] &\quad = \quad 0\text{x}02 \times RC[j-1] \end{aligned}$$

AES Key Schedule / Expansion



AES Example - The first Roundkey

- Key in Hex (128 bits): 54 68 61 74 73 20 6D 79 20 4B 75 6E 67 20 46 75
- $w[0] = (54, 68, 61, 74), w[1] = (73, 20, 6D, 79), w[2] = (20, 4B, 75, 6E), w[3] = (67, 20, 46, 75)$
- $g(w[3])$:
  - circular byte left shift of $w[3]$: $(20, 46, 75, 67)$
  - Byte Substitution (S-Box): $(B7, 5A, 9D, 85)$
  - Adding round constant $(01, 00, 00, 00)$ gives: $g(w[3]) = (B6, 5A, 9D, 85)$
- $w[4] = w[0] \oplus g(w[3]) = (E2, 32, FC, F1)$:

0101 0100	0110 1000	0110 0001	0111 0100
1011 0110	0101 1010	1001 1101	1000 0101
1110 0010	0011 0010	1111 1100	1111 0001
E2	32	FC	F1

- $w[5] = w[4] \oplus w[1] = (91, 12, 91, 88)$, $w[6] = w[5] \oplus w[2] = (B1, 59, E4, E6)$, $w[7] = w[6] \oplus w[3] = (D6, 79, A2, 93)$
- first roundkey: E2 32 FC F1 91 12 91 88 B1 59 E4 E6 D6 79 A2 93

Block Cipher Modes of Operations

- A block cipher takes a fixed-length block of text of length **b bits** and a key as input and produces a **b-bit** block of ciphertext.

- If the amount of plaintext to be encrypted is greater than b bits, then the block cipher can still be used by breaking the plaintext up into b-bit blocks.

Block Cipher Modes of Operations

- Block ciphers process blocks of fixed sizes (say 64 bits).
- The length of plaintexts is mostly not a multiple of the block size.
- The last block of bits needs to be padded up with redundant information so that the length of the final block equal to block size of the scheme.
- The process of adding bits to the last block is referred to as padding.

Block Cipher Modes of Operations

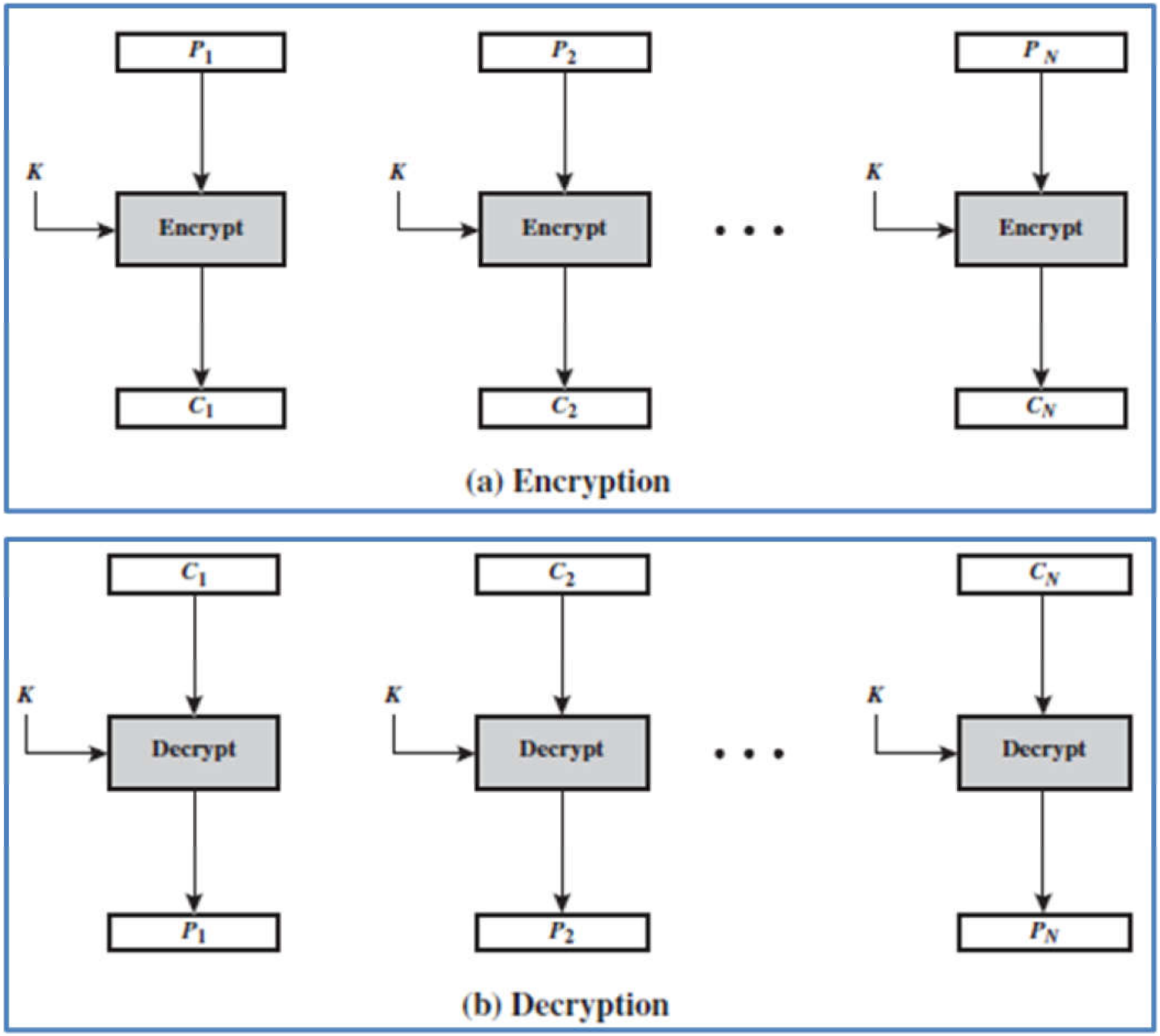
- A mode of operation is a technique for enhancing the effect of a cryptographic algorithm or adapting the algorithm for an application, such as applying a block cipher to a sequence of data blocks or a data stream.
- Modes of operations also avoid mapping the same plaintext to the same ciphertext.

- There are 5 modes of operation.

Electronic Code Block Mode (ECB)

- It is one of the simplest modes of operation.
- In this mode, the plain text is divided into a block where each block is 64 bits.
- Then each block is encrypted separately.
- The same key is used for the encryption of all blocks. Each block is encrypted using the key and makes the block of ciphertext.
- The term codebook is used because, for a given key, there is a unique ciphertext for every **b-bit** block of plaintext. The ECB method is ideal for a short amount of data

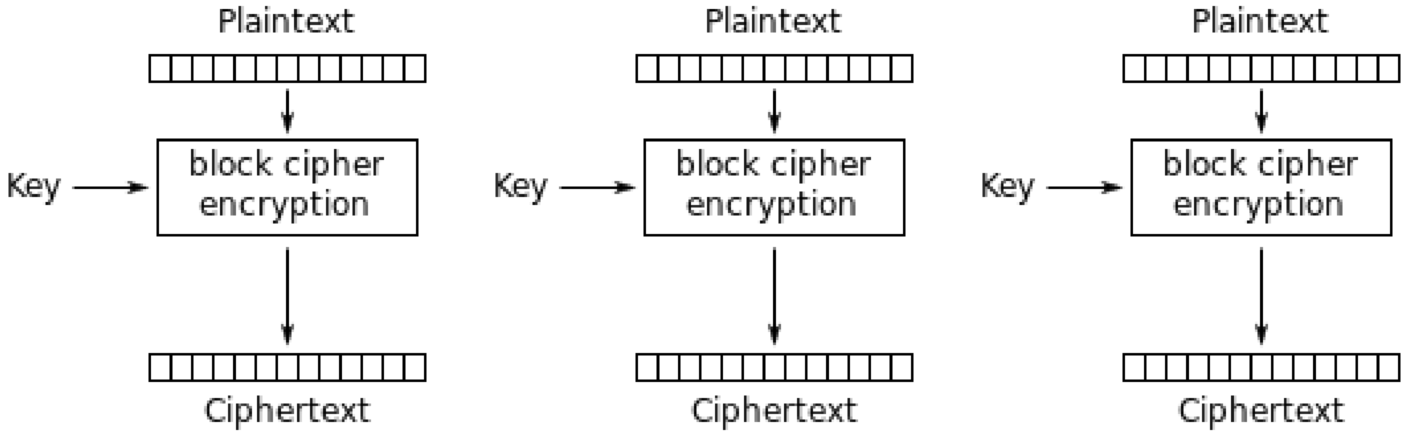
Electronic Code Block Mode (ECB)



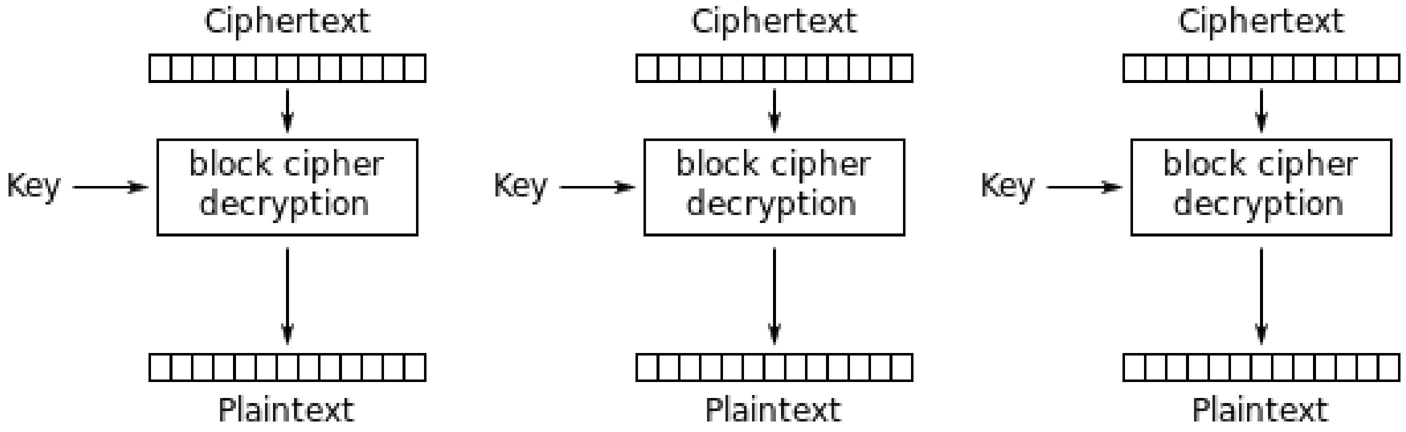
(a) Encryption

(b) Decryption

Electronic Code Block Mode (ECB)



Electronic Codebook (ECB) mode encryption



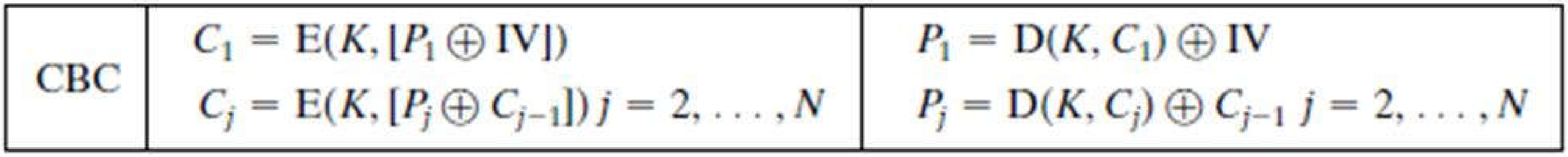
Electronic Codebook (ECB) mode decryption

Cipher Block Chaining Mode (CBC)

- In this scheme, the input to the encryption algorithm is the XOR of the current plaintext block and the preceding ciphertext block; the same key is used for each block.
- Therefore, if the same plaintext block is repeated, different ciphertext blocks are produced.
- For decryption, each cipher block is passed through the decryption algorithm. The result is XORed with the preceding ciphertext block to produce the plaintext block.

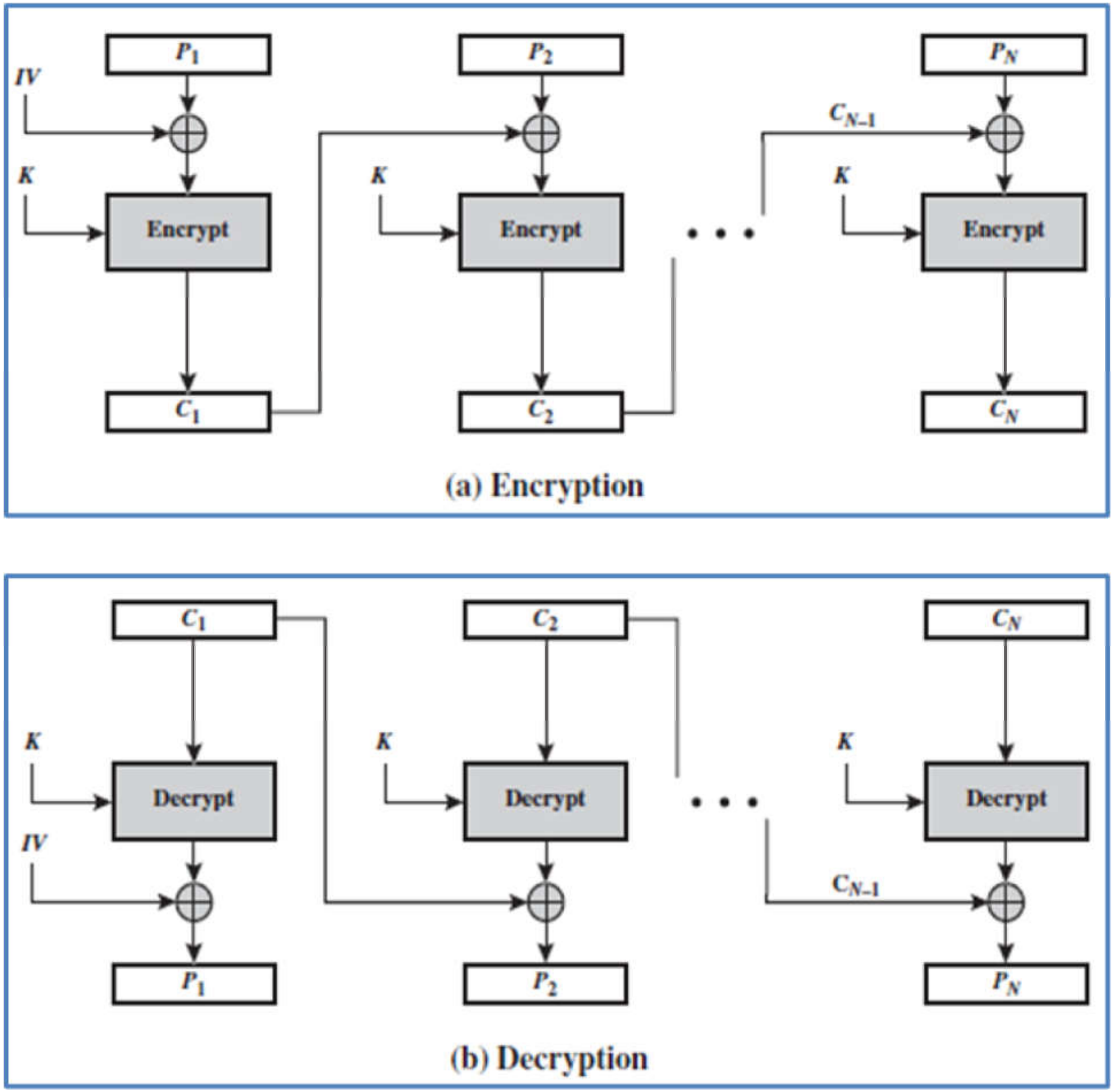
Cipher Block Chaining Mode (CBC)

- The IV is an initialization block, which is produced using random number generator and it should be the same size as the cipher block.
- This must be known to both the sender and receiver but it should be unpredictable by a third party.



CBC	$C_1 = \mathrm{E}(K, [P_1 \oplus \mathrm{IV}])$	$P_1 = \mathrm{D}(K, C_1) \oplus \mathrm{IV}$
	$C_j = \mathrm{E}(K, [P_j \oplus C_{j-1}])\ j = 2, \ldots, N$	$P_j = \mathrm{D}(K, C_j) \oplus C_{j-1}\ j = 2, \ldots, N$

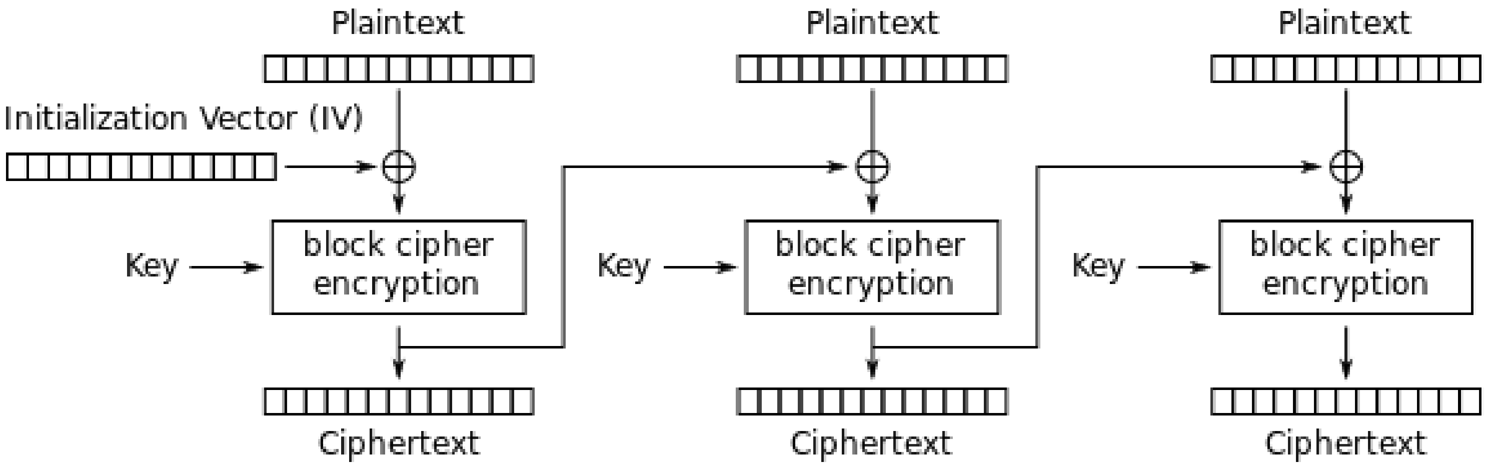
Cipher Block Chaining Mode (CBC)



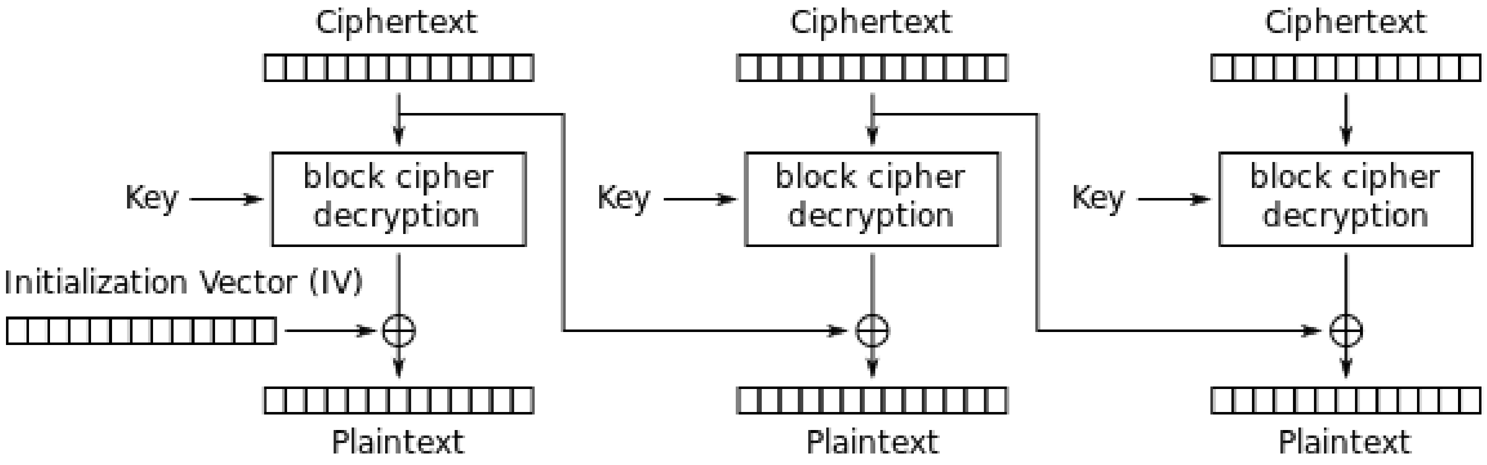
(a) Encryption

(b) Decryption

Cipher Block Chaining Mode (CBC)



Cipher Block Chaining (CBC) mode encryption



Cipher Block Chaining (CBC) mode decryption

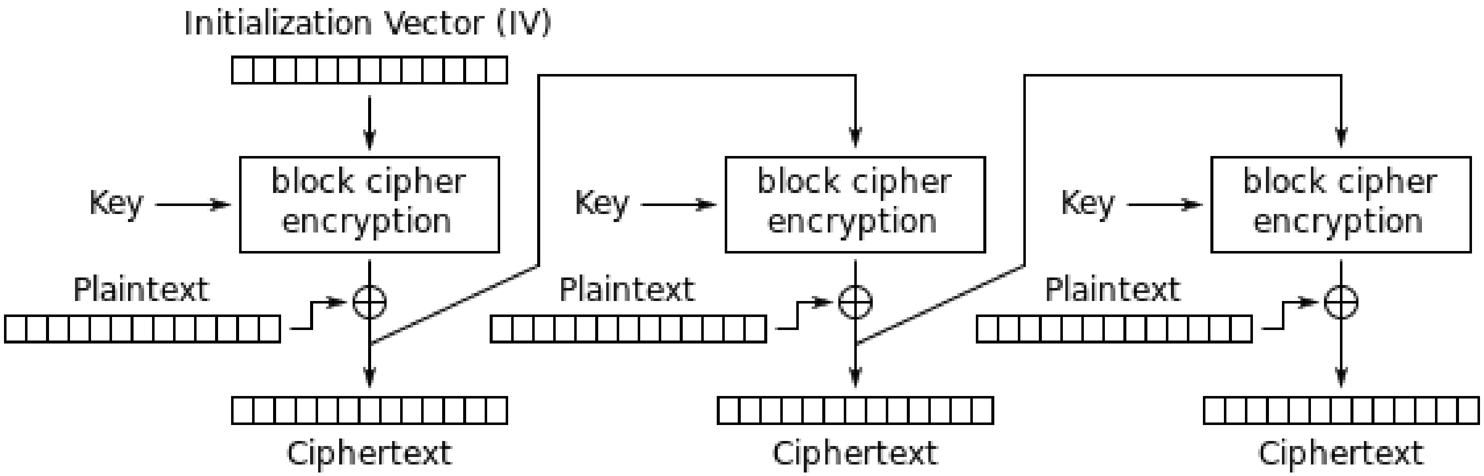
Cipher Feedback Mode (CFB) [full-CFB]

- The cipher feedback (CFB) mode, in its simplest form uses the entire output of the block cipher.
- In this variation, it is very similar to CBC, makes a block cipher into a self-synchronizing stream cipher.

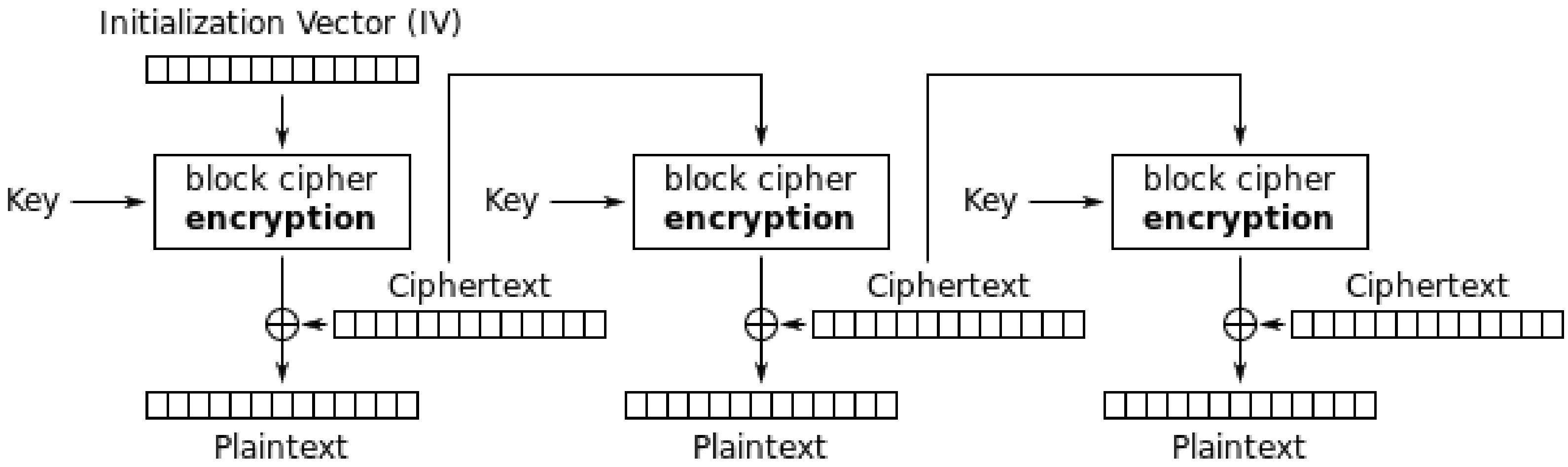
$$C_i = \begin{cases} \mathrm{IV}, & i = 0 \\ E_K(C_{i-1}) \oplus P_i, & \text{otherwise} \end{cases}$$

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

Cipher Feedback Mode (CFB) [full-CFB]

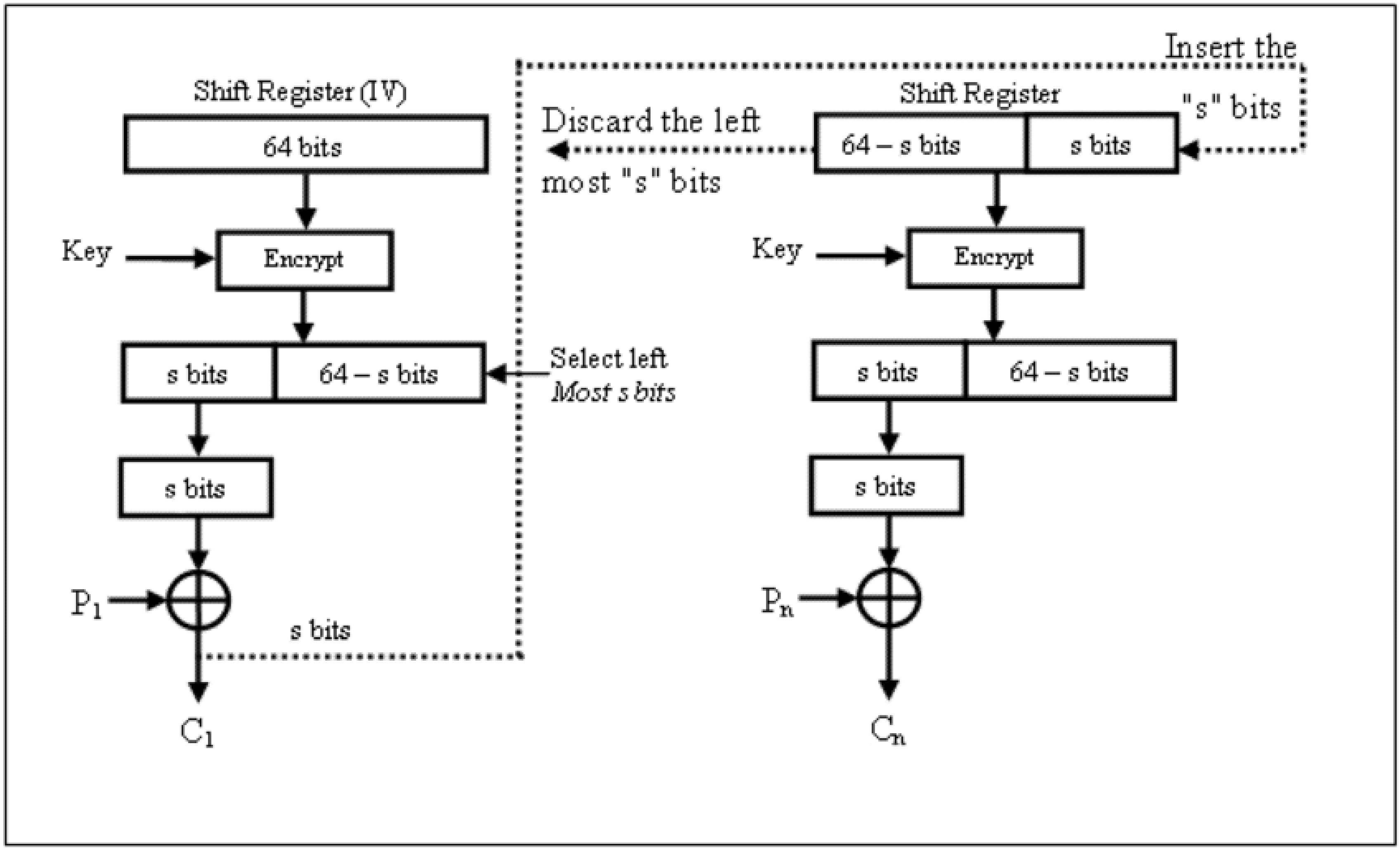


Cipher Feedback (CFB) mode encryption



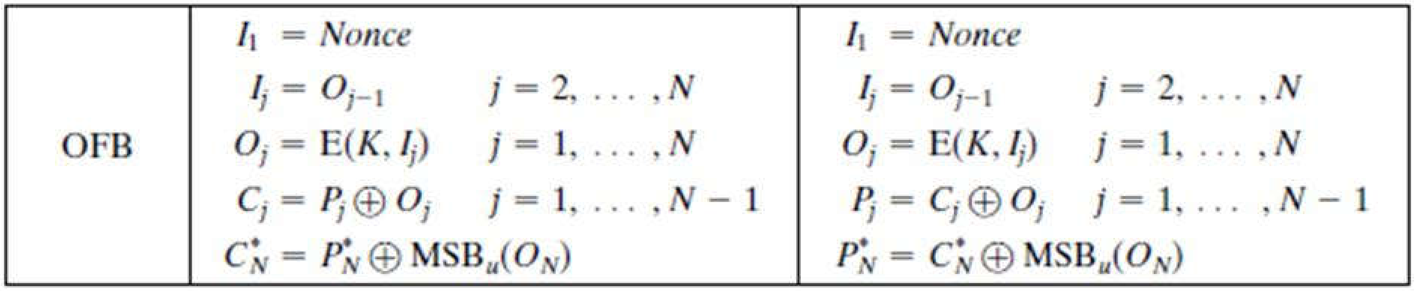
Cipher Feedback (CFB) mode decryption

Cipher Feedback Mode (CFB) [CFB-s]



Output Feedback Mode (OFB)

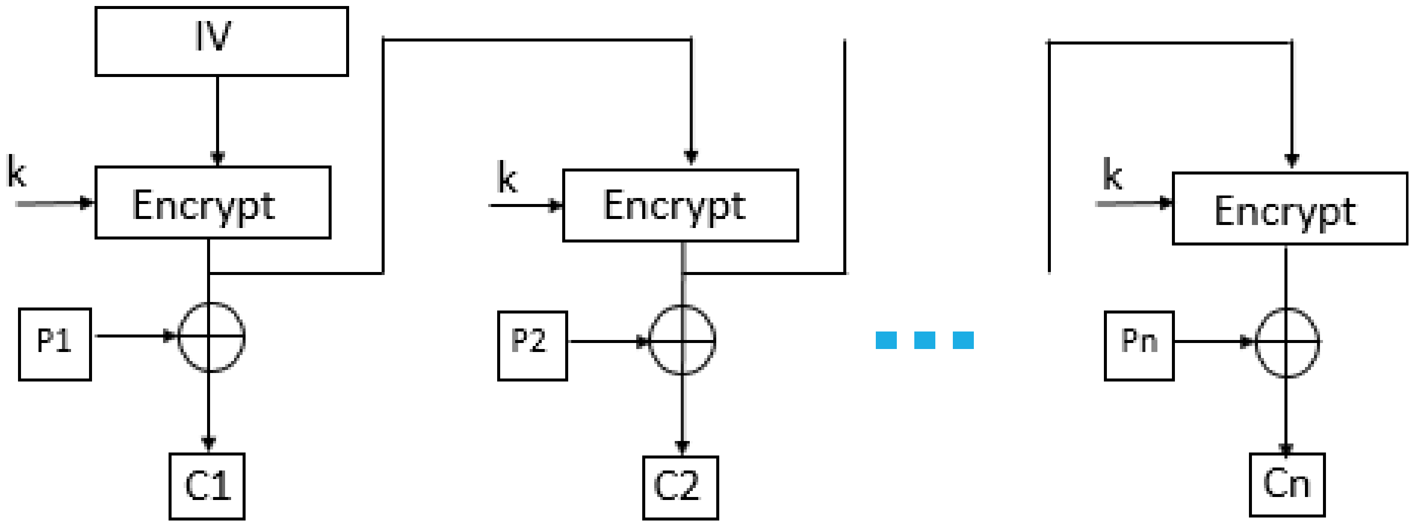
- This scheme operates on full blocks of plaintext and ciphertext where the output of the encryption function is fed back to become the input for encrypting the next block of plaintext.
- In the case of OFB, the IV must be a nonce; that is, the IV must be unique to each execution of the encryption operation.



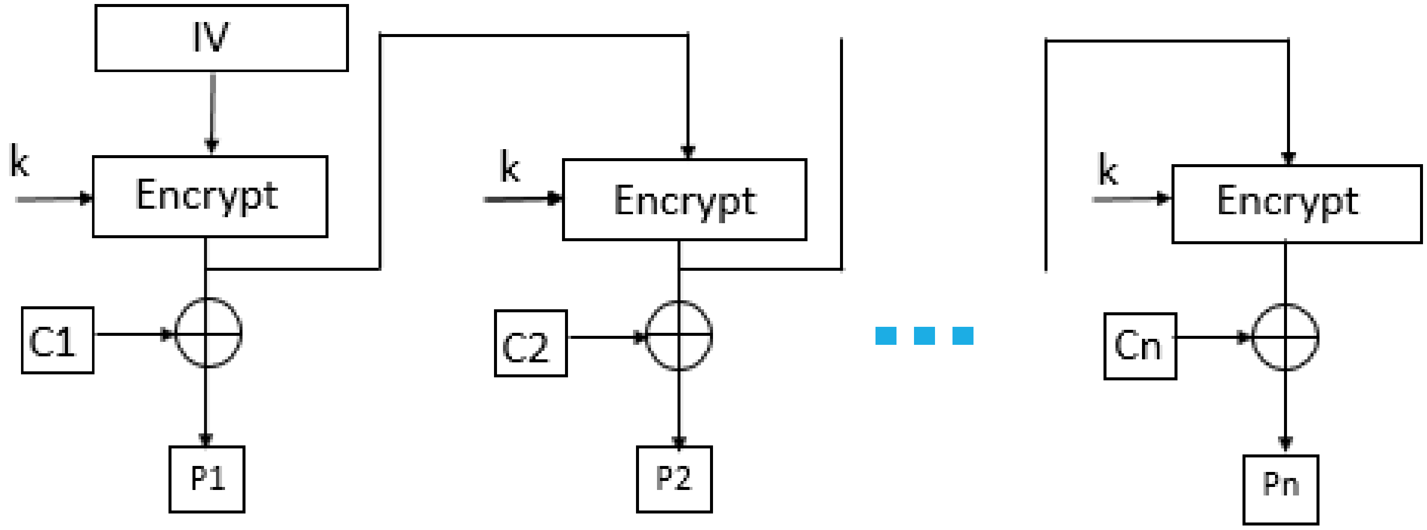
OFB	$I_1 = Nonce$	$I_1 = Nonce$
	$I_j = O_{j-1} \qquad j = 2, \ldots, N$	$I_j = O_{j-1} \qquad j = 2, \ldots, N$
	$O_j = \mathrm{E}(K, I_j) \qquad j = 1, \ldots, N$	$O_j = \mathrm{E}(K, I_j) \qquad j = 1, \ldots, N$
	$C_j = P_j \oplus O_j \qquad j = 1, \ldots, N - 1$	$P_j = C_j \oplus O_j \qquad j = 1, \ldots, N - 1$
	$C^*_N = P^*_N \oplus \mathrm{MSB}_u(O_N)$	$P^*_N = C^*_N \oplus \mathrm{MSB}_u(O_N)$

Output Feedback Mode (OFB)

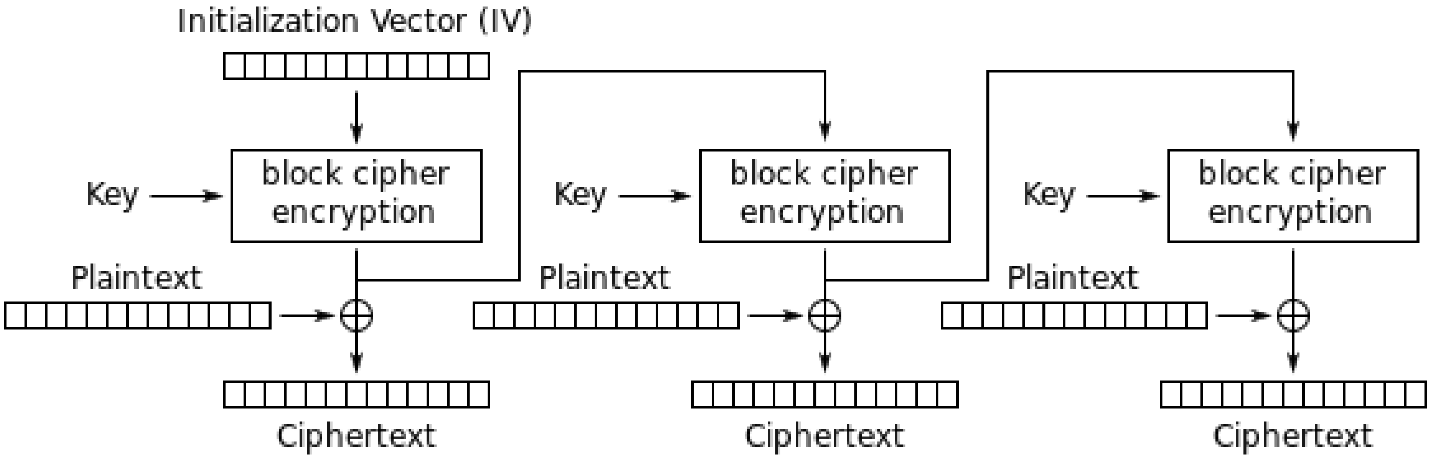
Encryption



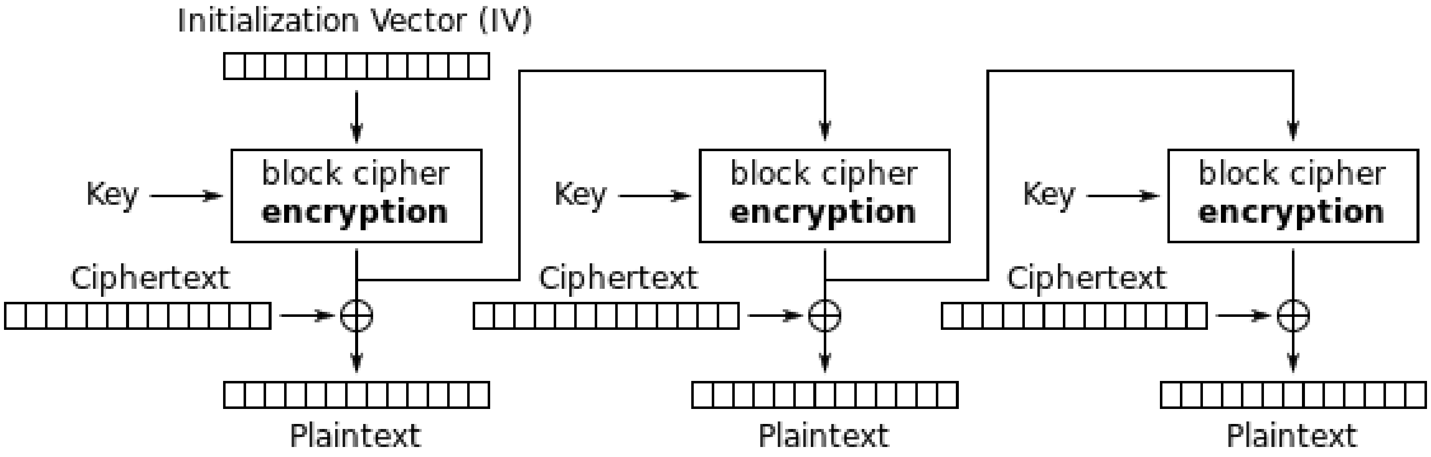
Decryption



Output Feedback Mode (OFB)



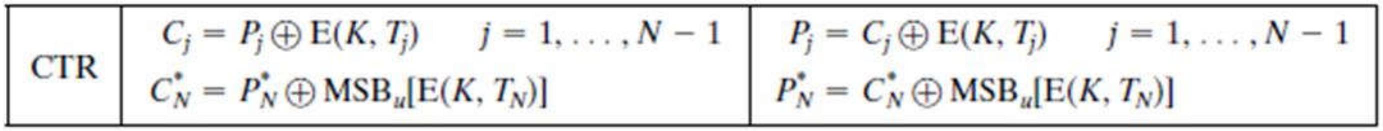
Output Feedback (OFB) mode encryption


Initialization Vector (IV)
Key
block cipher **encryption**
Ciphertext
Plaintext
Key
block cipher **encryption**
Ciphertext
Plaintext
Key
block cipher **encryption**
Ciphertext
Plaintext

Output Feedback (OFB) mode decryption

Counter Mode (CTR)

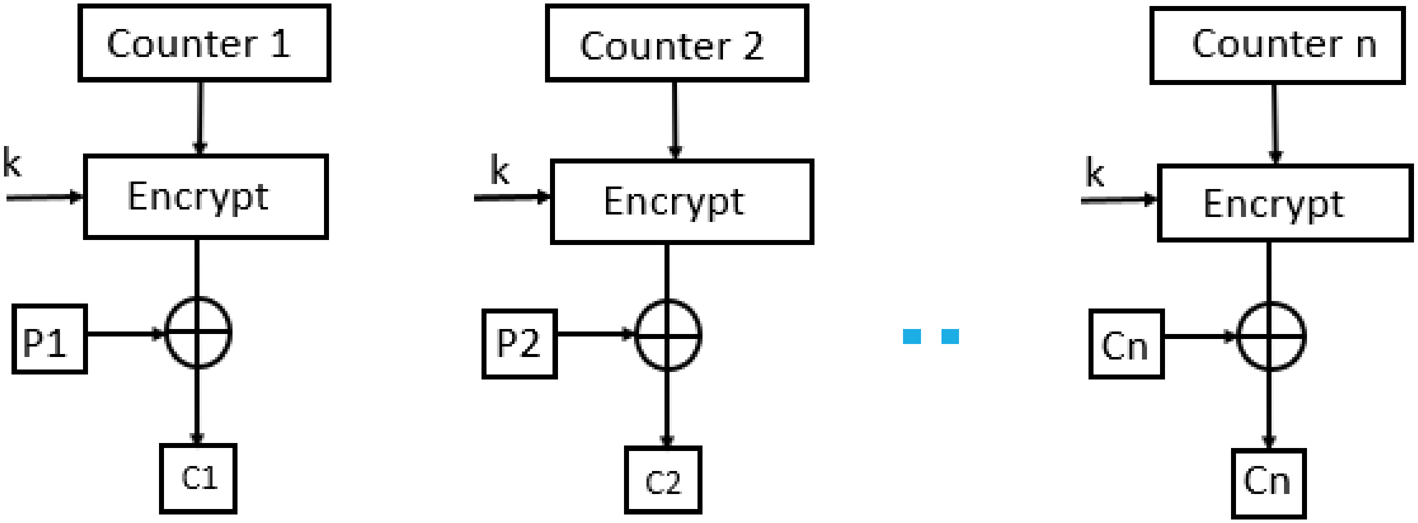
In this mode, each block of plaintext is XORed with an encrypted counter. Typically, the counter is initialized to some value and then incremented by 1 for each subsequent block being encrypted using the same key. Given a sequence of counters $\boldsymbol{T_1}$, $\boldsymbol{T_2}$, …, $\boldsymbol{T_N}$, we can define CTR mode as follows:



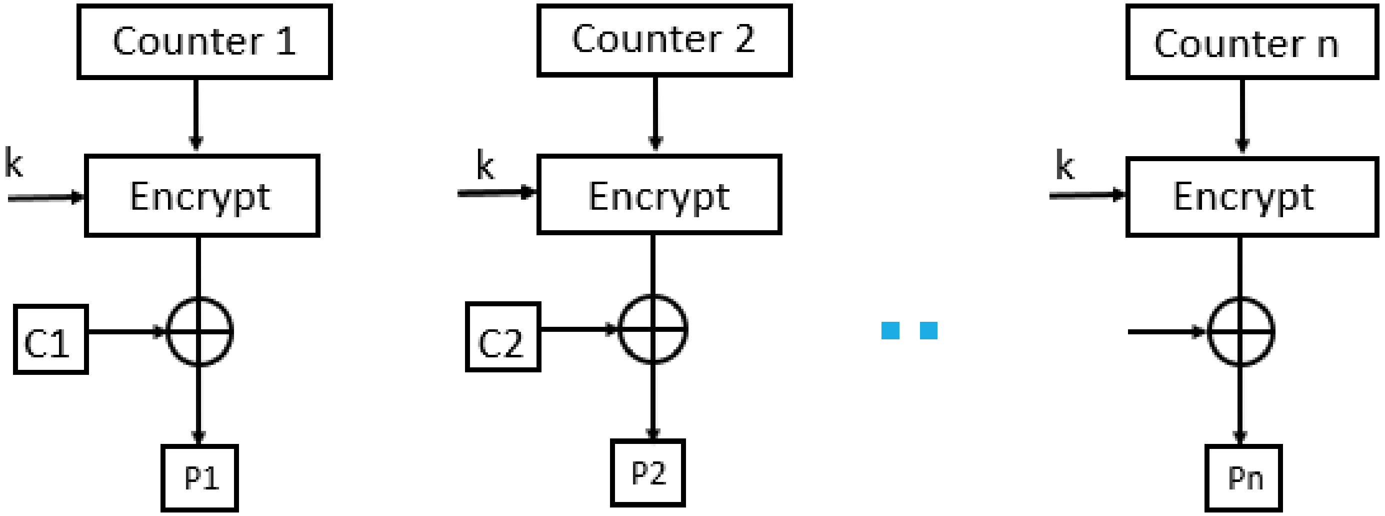
CTR	$C_j = P_j \oplus \mathrm{E}(K, T_j) \qquad j = 1, \ldots, N-1$ $C^*_N = P^*_N \oplus \mathrm{MSB}_u[\mathrm{E}(K, T_N)]$	$P_j = C_j \oplus \mathrm{E}(K, T_j) \qquad j = 1, \ldots, N-1$ $P^*_N = C^*_N \oplus \mathrm{MSB}_u[\mathrm{E}(K, T_N)]$

Counter Mode (CTR)

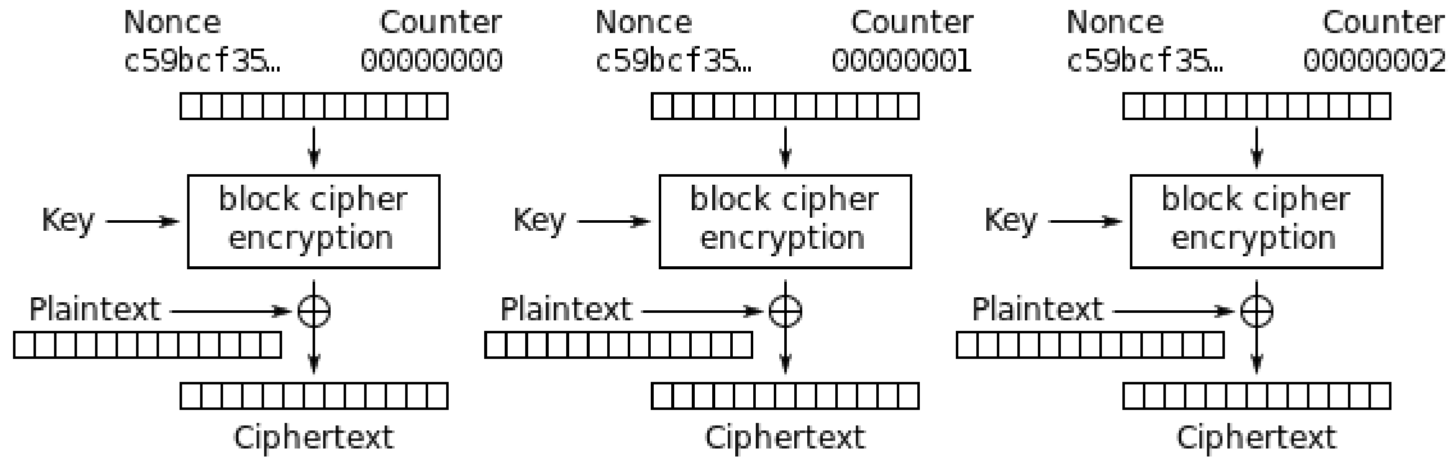
Encryption



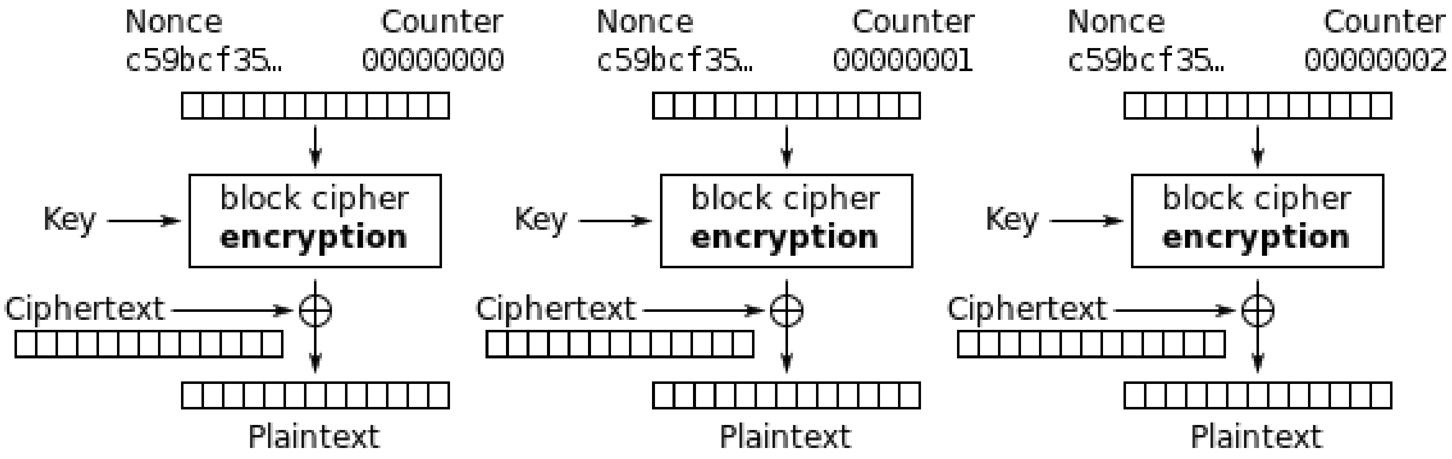
Decryption



Counter Mode (CTR)

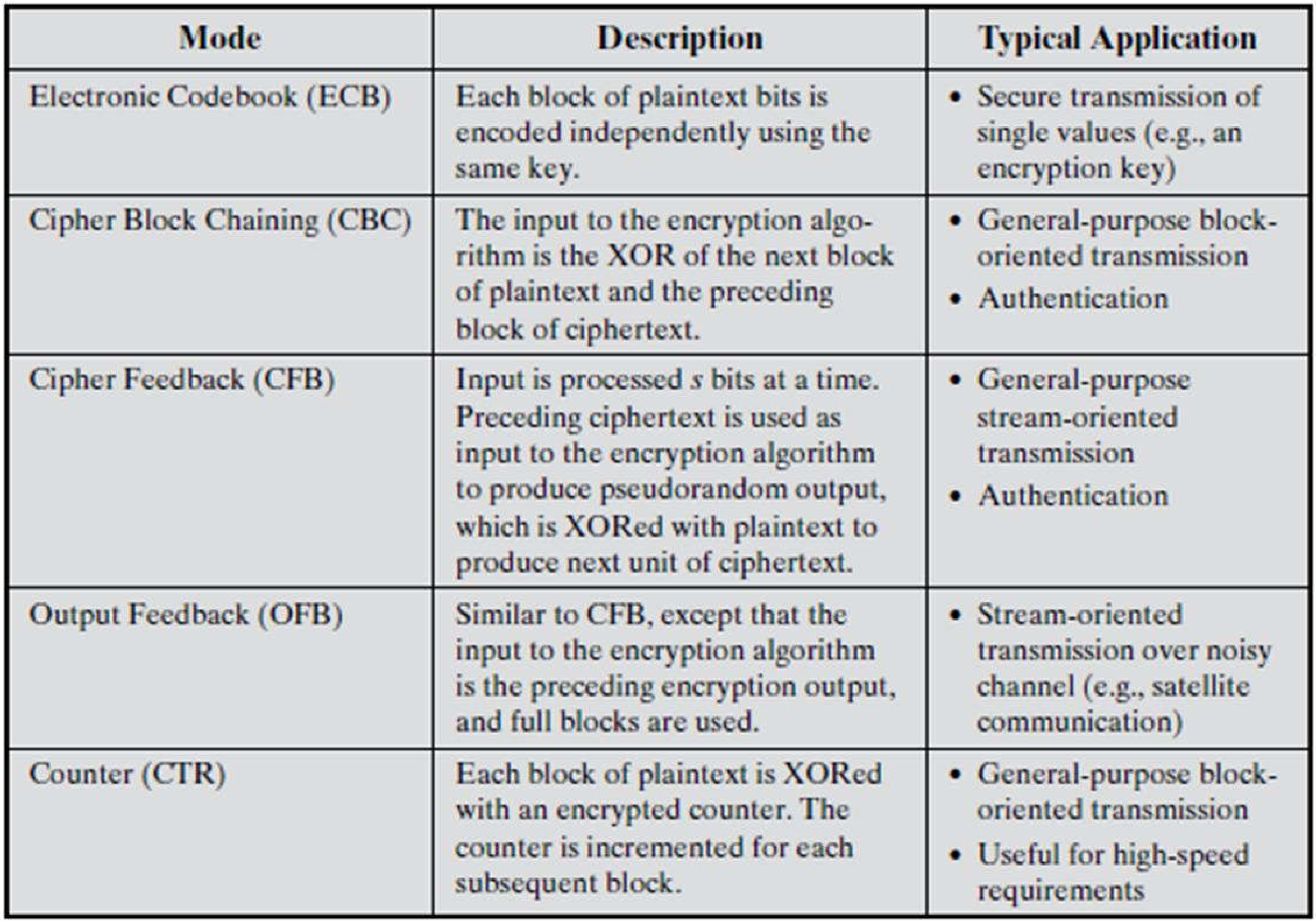


Counter (CTR) mode encryption



Counter (CTR) mode decryption

Block Cipher Modes of Operations



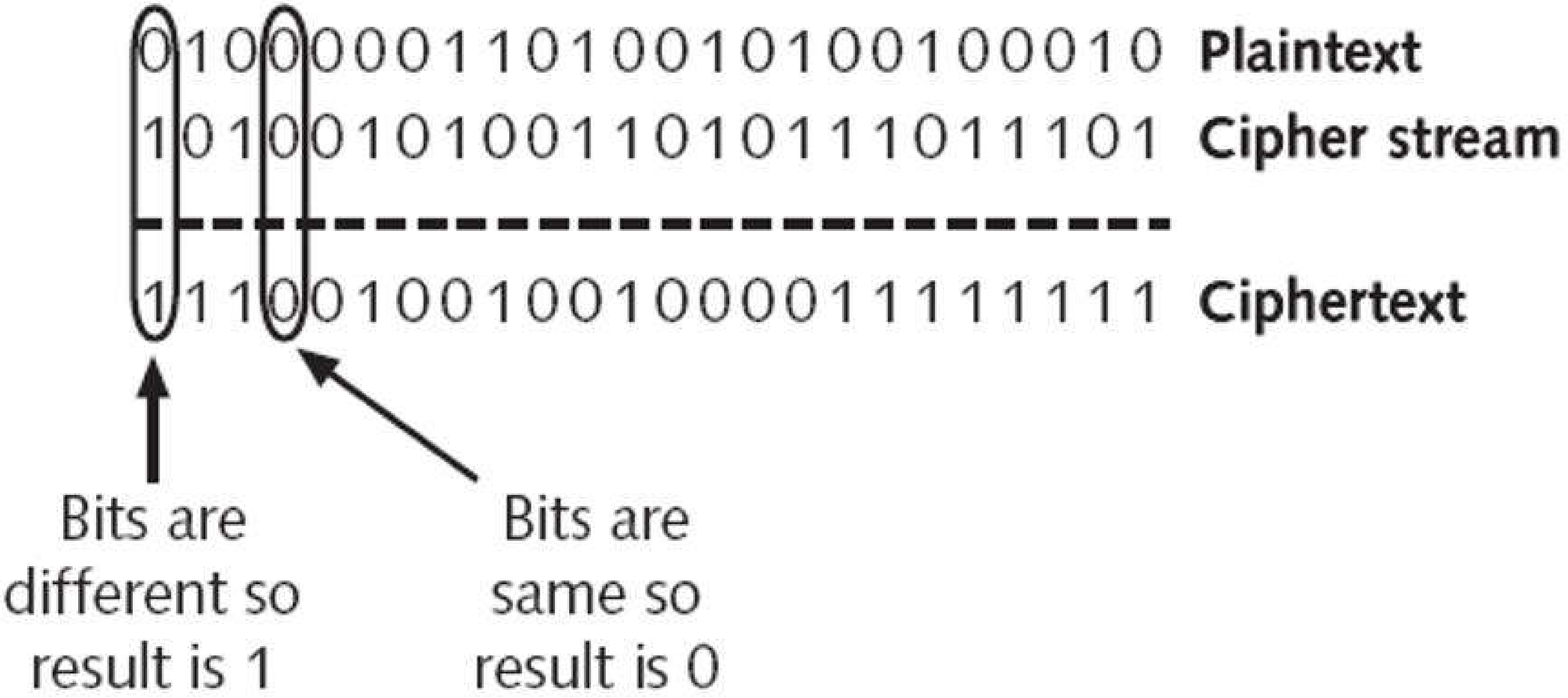
Mode	Description	Typical Application
Electronic Codebook (ECB)	Each block of plaintext bits is encoded independently using the same key.	• Secure transmission of single values (e.g., an encryption key)
Cipher Block Chaining (CBC)	The input to the encryption algorithm is the XOR of the next block of plaintext and the preceding block of ciphertext.	• General-purpose block-oriented transmission • Authentication
Cipher Feedback (CFB)	Input is processed *s* bits at a time. Preceding ciphertext is used as input to the encryption algorithm to produce pseudorandom output, which is XORed with plaintext to produce next unit of ciphertext.	• General-purpose stream-oriented transmission • Authentication
Output Feedback (OFB)	Similar to CFB, except that the input to the encryption algorithm is the preceding encryption output, and full blocks are used.	• Stream-oriented transmission over noisy channel (e.g., satellite communication)
Counter (CTR)	Each block of plaintext is XORed with an encrypted counter. The counter is incremented for each subsequent block.	• General-purpose block-oriented transmission • Useful for high-speed requirements

Stream Cipher

- A stream cipher is a method of encrypting text (to produce ciphertext) in which a cryptographic key and algorithm are applied to each binary digit in a data stream, one bit at a time.

- The main alternative method to stream cipher is the block cipher, where a key and algorithm are applied to blocks of data rather than individual bits in a stream.

Stream Cipher



010000011010010100100010 Plaintext

101001010011010111011101 Cipher stream

111001001001000001111111 Ciphertext

Bits are different so result is 1

Bits are same so result is 0

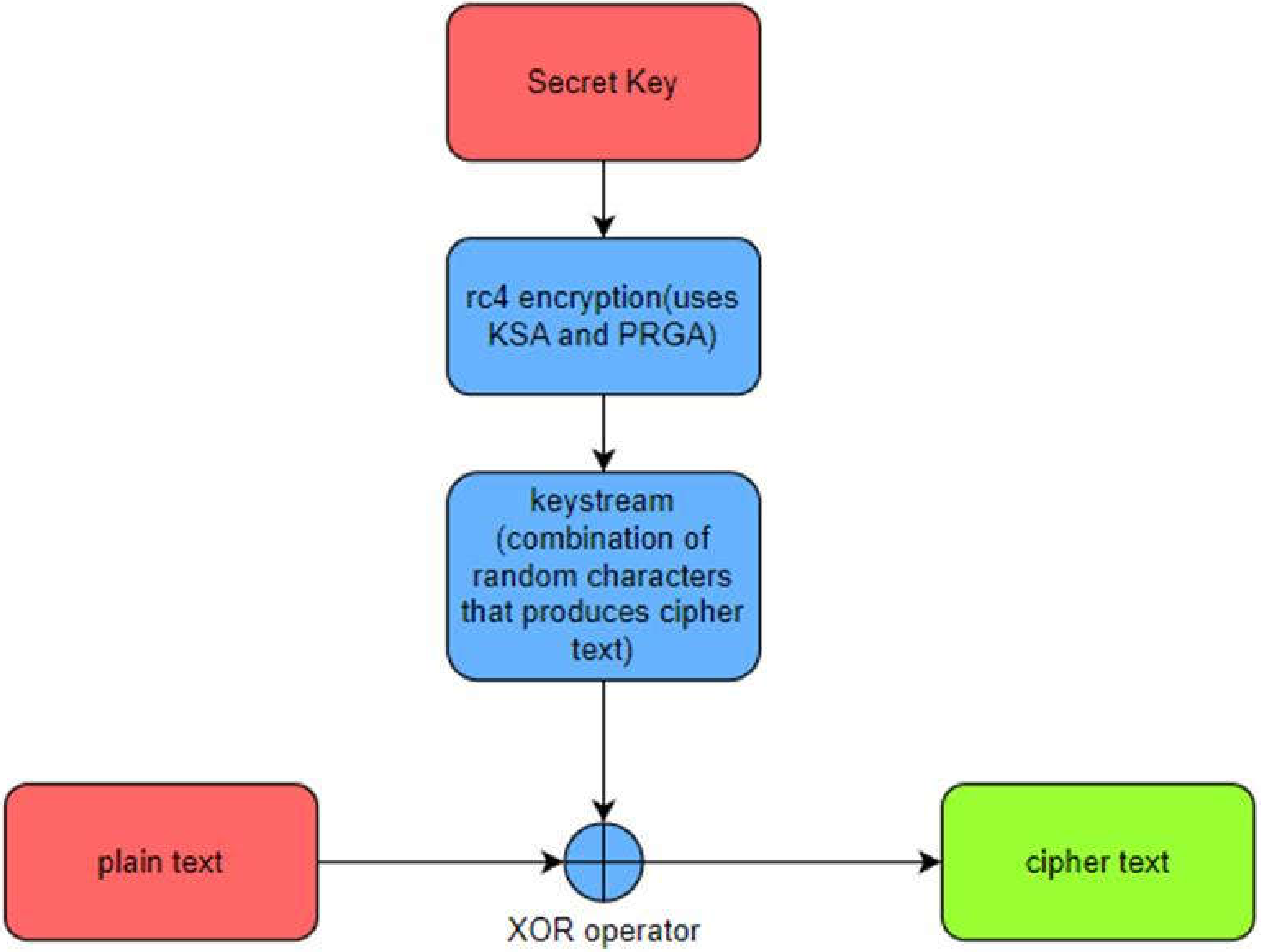
RC4 Algorithm

- RC4 (also known as Rivest Cipher 4) is a form of stream cipher. It encrypts messages one byte at a time via an algorithm.
- Plenty of stream ciphers exist, but RC4 is among the most popular.
- It's simple to apply, and it works quickly, even on very large pieces of data.
- If you've ever used an application like TSL (transport layer security) or SSL (secure socket layer), you've probably encountered RC4 encryption.

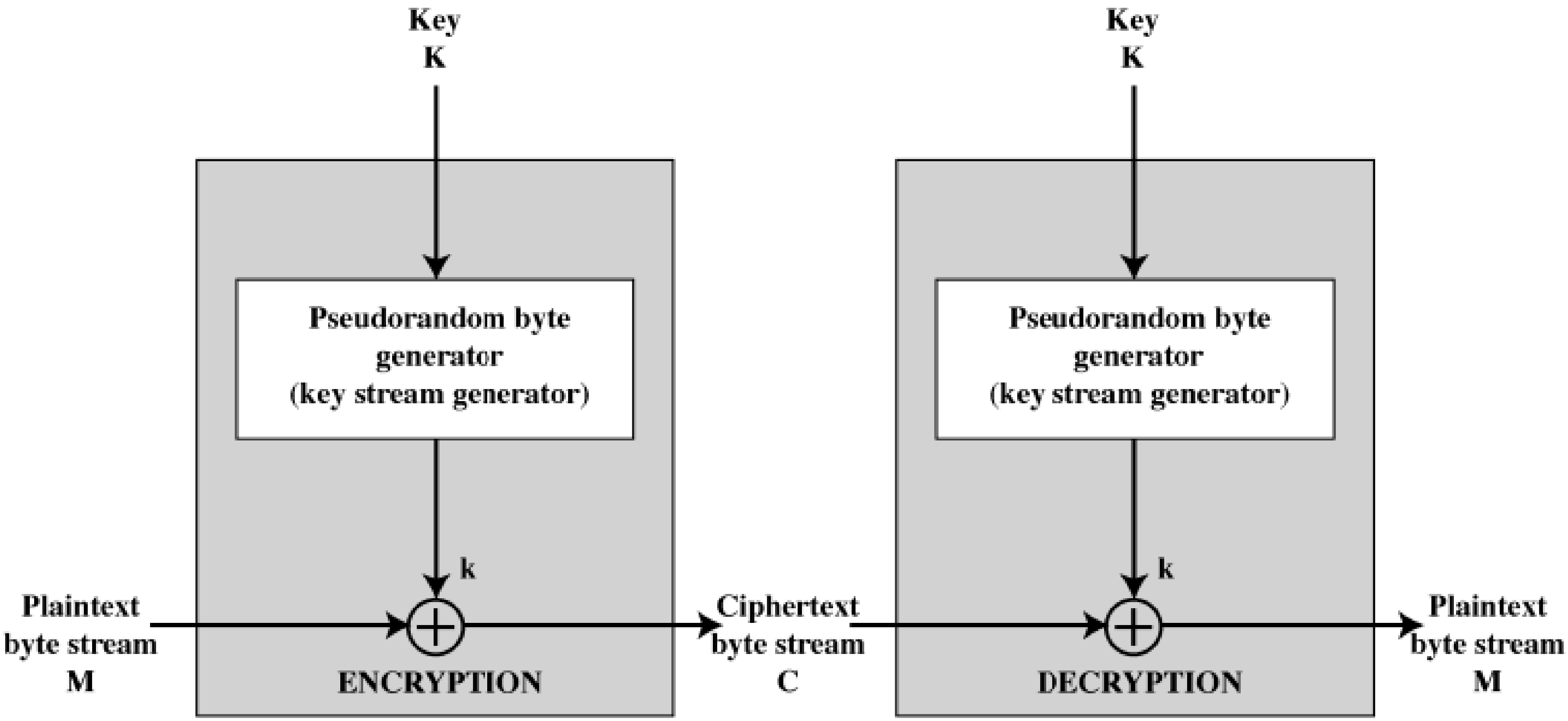
RC4 Algorithm

- RC4 encryption algorithm comprises two components—KSA (Key scheduling algorithm) and PRGA (Pseudo random generation algorithm).
- These two algorithms together help the rc4 algorithm to produce the stream cipher.

RC4 Algorithm


Secret Key
rc4 encryption(uses KSA and PRGA)
keystream (combination of random characters that produces cipher text)
plain text
XOR operator
cipher text

RC4 Algorithm

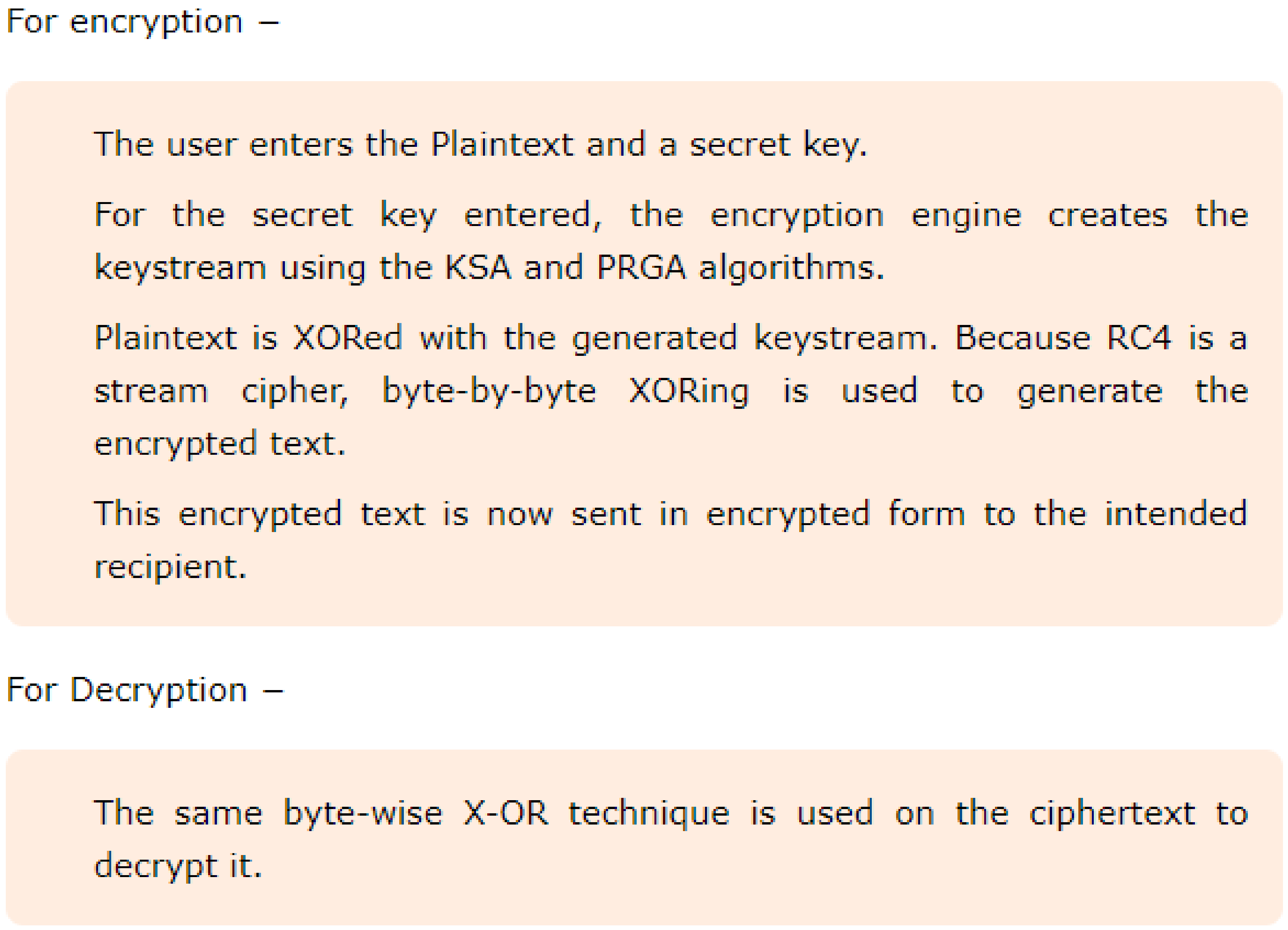

Key
K
Pseudorandom byte generator (key stream generator)
k
Plaintext byte stream M
ENCRYPTION
Ciphertext byte stream C
Key
K
Pseudorandom byte generator (key stream generator)
k
DECRYPTION
Plaintext byte stream M

RC4 Algorithm

- **Consists of 2 parts:**
  - Key Scheduling Algorithm (KSA)
  - Pseudo-Random Generation Algorithm (PRGA)

- **KSA**
  - Generate State array
- **PRGA on the KSA**
  - Generate keystream
  - XOR keystream with the data to generated encrypted stream

RC4 Algorithm



For encryption –

The user enters the Plaintext and a secret key.

For the secret key entered, the encryption engine creates the keystream using the KSA and PRGA algorithms.

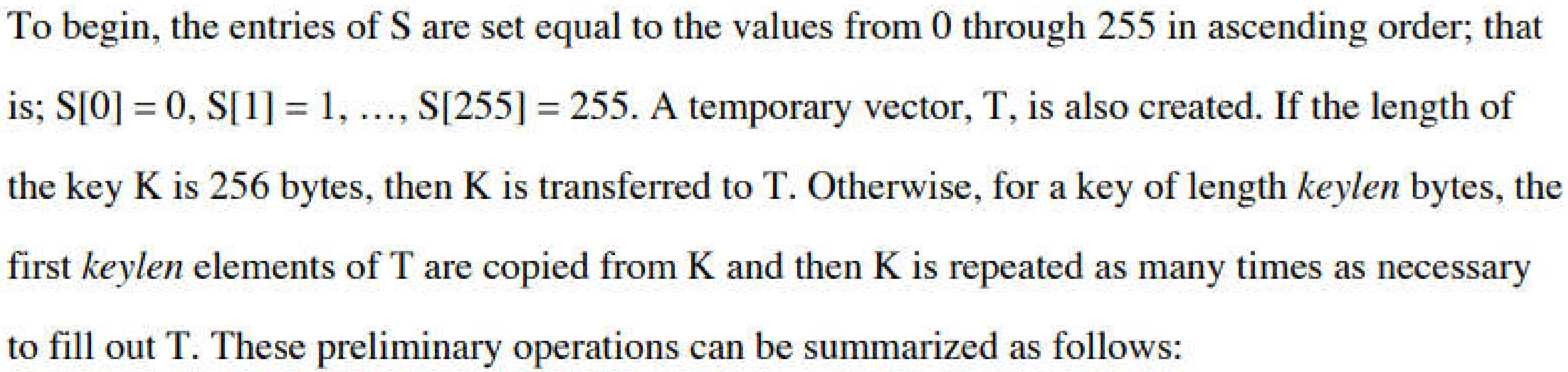
Plaintext is XORed with the generated keystream. Because RC4 is a stream cipher, byte-by-byte XORing is used to generate the encrypted text.

This encrypted text is now sent in encrypted form to the intended recipient.

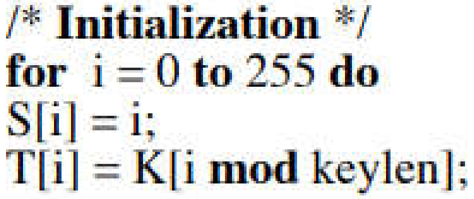
For Decryption –

The same byte-wise X-OR technique is used on the ciphertext to decrypt it.

RC4 Algorithm: KSA

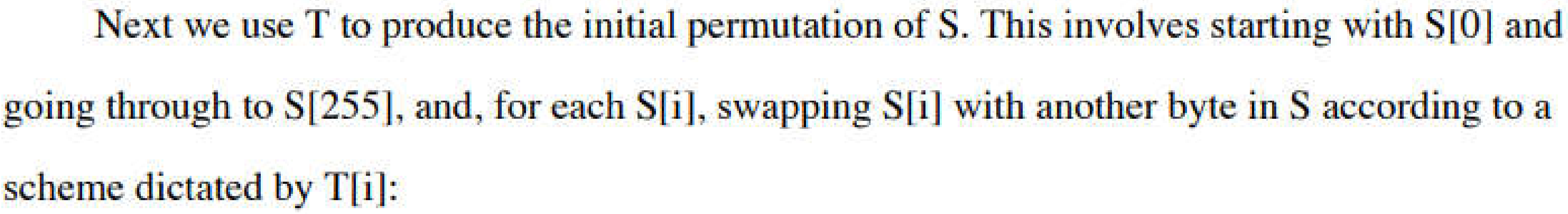


To begin, the entries of S are set equal to the values from 0 through 255 in ascending order; that is; S[0] = 0, S[1] = 1, …, S[255] = 255. A temporary vector, T, is also created. If the length of the key K is 256 bytes, then K is transferred to T. Otherwise, for a key of length *keylen* bytes, the first *keylen* elements of T are copied from K and then K is repeated as many times as necessary to fill out T. These preliminary operations can be summarized as follows:

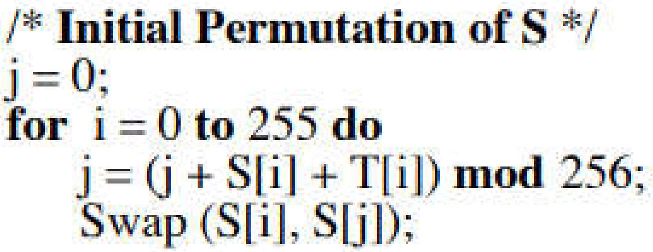


```
/* Initialization */
for  i = 0 to 255 do
S[i] = i;
T[i] = K[i mod keylen];
```

RC4 Algorithm: KSA

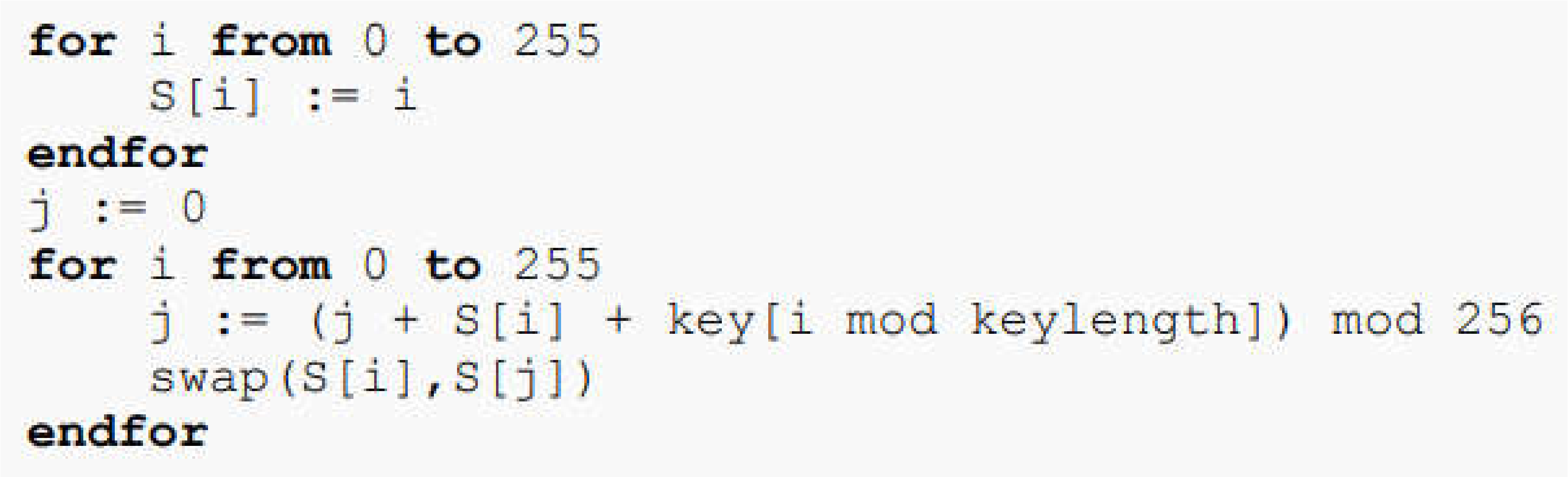


Next we use T to produce the initial permutation of S. This involves starting with S[0] and going through to S[255], and, for each S[i], swapping S[i] with another byte in S according to a scheme dictated by T[i]:

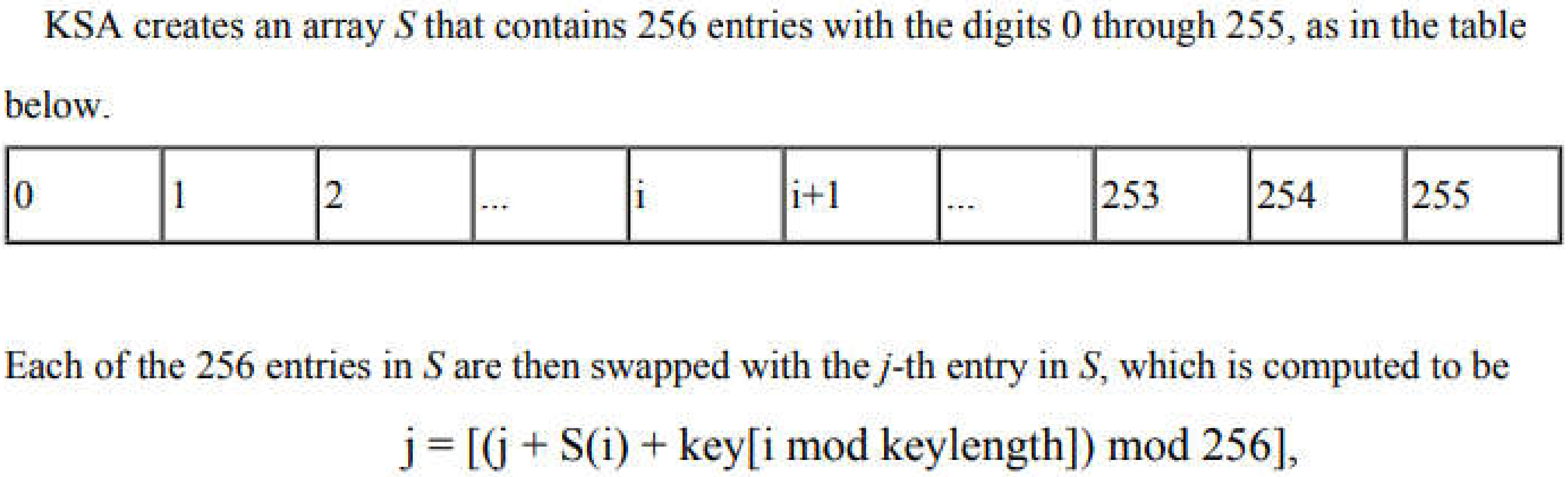


```
/* **Initial Permutation of S** */
j = 0;
**for** i = 0 **to** 255 **do**
    j = (j + S[i] + T[i]) **mod** 256;
    Swap (S[i], S[j]);
```

RC4 Algorithm: KSA



```
for i from 0 to 255
    S[i] := i
endfor
j := 0
for i from 0 to 255
    j := (j + S[i] + key[i mod keylength]) mod 256
    swap(S[i],S[j])
endfor
```



KSA creates an array *S* that contains 256 entries with the digits 0 through 255, as in the table below.

0	1	2	...	i	i+1	...	253	254	255

Each of the 256 entries in *S* are then swapped with the *j*-th entry in *S*, which is computed to be

$$j = [(j + S(i) + key[i \bmod keylength]) \bmod 256],$$

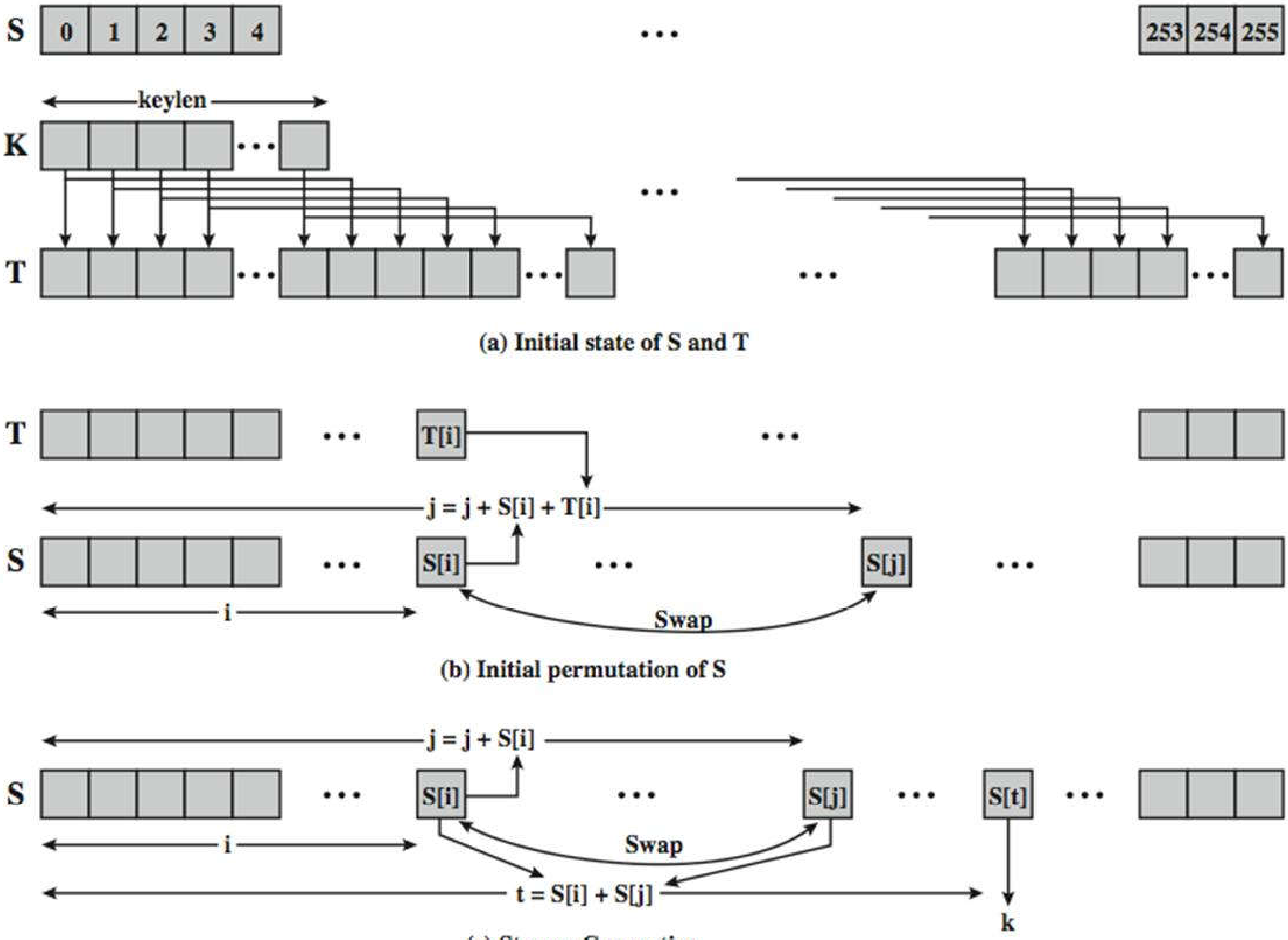
RC4 Algorithm: PRGA

Once the S vector is initialized, the input key is no longer used. Stream generation involves starting with S[0] and going through to S[255], and, for each S[i], swapping S[i] with another byte in S according to a scheme dictated by the current configuration of S. After S[255] is reached, the process continues, starting over again at S[0]:

```
/* Stream Generation */
i, j = 0;
while (true)
    i = (i + 1) mod 256;
    j = (j + S[i]) mod 256;
    Swap (S[i], S[j]);
    t = (S[i] + S[j]) mod 256;
    k = S[t];
```

To encrypt, XOR the value *k* with the next byte of plaintext. To decrypt, XOR the value *k* with the next byte of ciphertext.

RC4 Algorithm



(a) Initial state of S and T

(b) Initial permutation of S

(c) Stream Generation

Decryption in RC4 Algorithm

- Use the same secret key as during the encryption phase.
- Generate keystream by running the KSA and PRGA.
- XOR keystream with the encrypted text to generate the plain text.
- Logic is simple :

$$(A \text{ xor } B) \text{ xor } B = A$$

A = Plain Text or Data
B = KeyStream

RC4 Encryption Example

Lets consider the stream cipher RC4, but instead of the full 256 bytes, we will use 8 x 3-bits. That is, the state vector **S** is 8 x 3-bits. We will operate on 3-bits of plaintext at a time since S can take the values 0 to 7, which can be represented as 3 bits.

Assume we use a 4 x 3-bit key of **K** = [1 2 3 6]. And a plaintext **P** = [1 2 2 2]

Initialise the state vector **S** and temporary vector **T**. **S** is initialised so the S[i] = i, and **T** is initialised so it is the key **K** (repeated as necessary).

S = [0 1 2 3 4 5 6 7]
T = [1 2 3 6 1 2 3 6]

RC4 Encryption Example

Now perform the initial permutation on S.

```
j = 0;
for i = 0 to 7 do
            j = (j + S[i] + T[i]) mod 8
            Swap(S[i],S[j]);
end
```

For i = 0:

$$\begin{aligned} j &= (0 + 0 + 1) \bmod 8 \\ &= 1 \end{aligned}$$

Swap(S[0],S[1]);

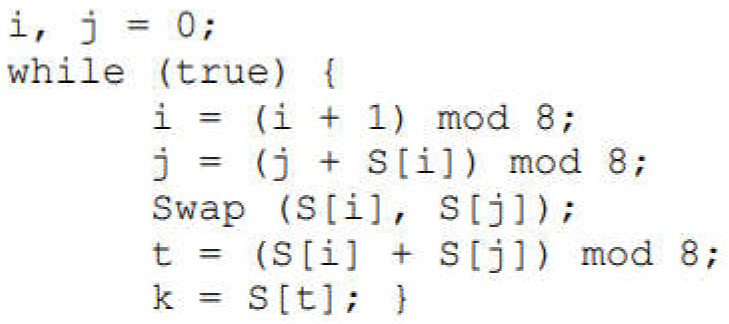
S = [1 0 2 3 4 5 6 7]

our initial permutation of **S** = [2 3 7 4 0 1 6 5]:

For i = 1:
j = 3
Swap(S[1],S[3])
S = [1 3 2 0 4 5 6 7];

RC4 Encryption Example

Now we generate 3-bits at a time, k, that we XOR with each 3-bits of plaintext to produce the ciphertext. The 3-bits k is generated by:



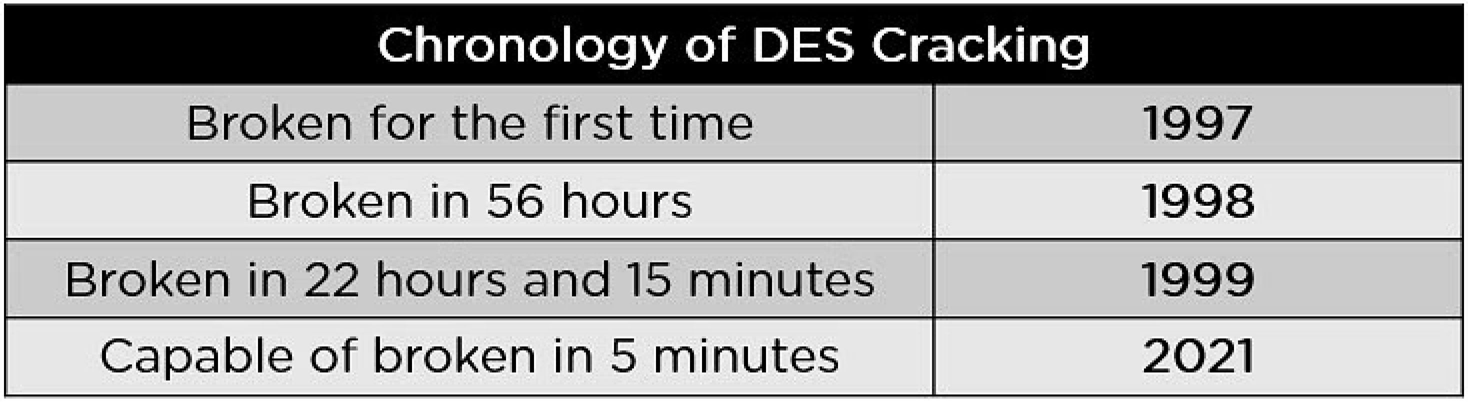
```
i, j = 0;
while (true) {
        i = (i + 1) mod 8;
        j = (j + S[i]) mod 8;
        Swap (S[i], S[j]);
        t = (S[i] + S[j]) mod 8;
        k = S[t]; }
```

The first iteration:
S = [2 3 7 4 0 1 6 5]
i = (0 + 1) mod 8 = 1
j = (0 + S[1]) mod 8 = 3
Swap(S[1],S[3])
S = [2 4 7 3 0 1 6 5]
t = (S[1] + S[3]) mod 8 = 7
k = S[7] = 5

Remember, **P** = [1 2 2 2]

So our first 3-bits of ciphertext is obtained by: k XOR P
5 XOR 1 = 101 XOR 001 = 100 = 4

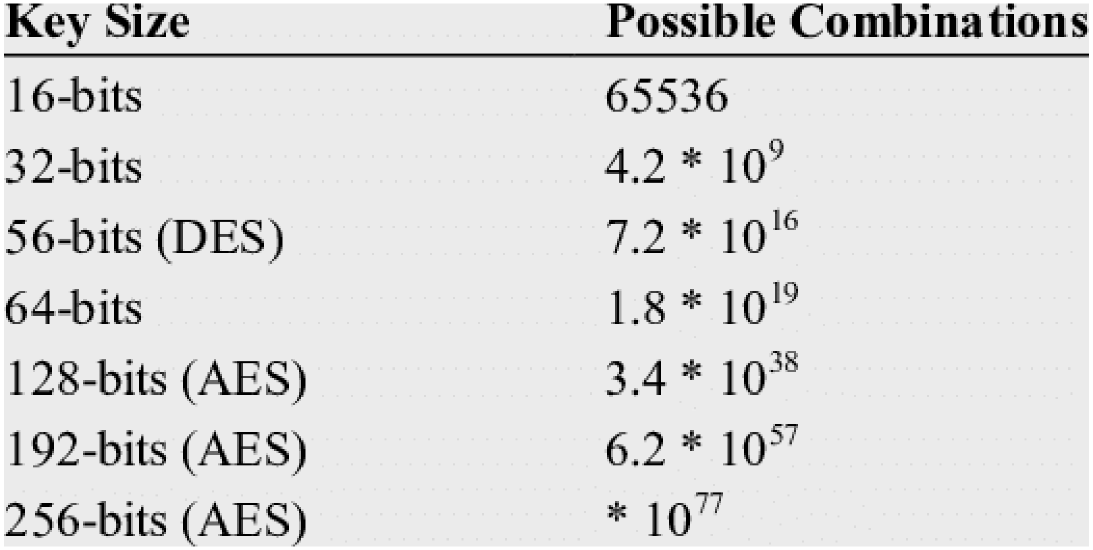
Time to Crack?



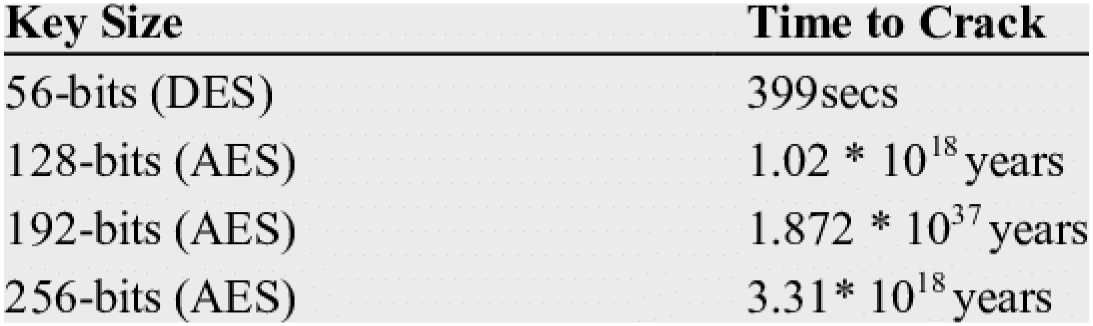
Chronology of DES Cracking	
Broken for the first time	1997
Broken in 56 hours	1998
Broken in 22 hours and 15 minutes	1999
Capable of broken in 5 minutes	2021

Source: Wikipedia

Time to Crack?



Key Size	Possible Combinations
16-bits	65536
32-bits	$4.2 * 10^{9}$
56-bits (DES)	$7.2 * 10^{16}$
64-bits	$1.8 * 10^{19}$
128-bits (AES)	$3.4 * 10^{38}$
192-bits (AES)	$6.2 * 10^{57}$
256-bits (AES)	$* 10^{77}$



Key Size	Time to Crack
56-bits (DES)	399secs
128-bits (AES)	$1.02 * 10^{18}$ years
192-bits (AES)	$1.872 * 10^{37}$ years
256-bits (AES)	$3.31* 10^{18}$ years

Time to Crack?

- AES successor to DES selected in 2001.
- 128-bit keys, encrypt 128-bit blocks
- Brute force attack
  - Try 1 Trillion keys per second
  - Would take 10790283070806000000 years to try all keys!
- A machine that can crack a DES key in one second would take 149 trillion years to crack a 128-bit AES key.

Summary

- Block Cipher Modes of Operations
- RC4