5.3 Let $S = \{0, a, b, c\}$ The addition and multiplication on the set S is defined in the following tables:

+	0	a	B	C
0	0	a	B	C
A	a	0	c	B
B	b	c	0	A
C	c	b	a	0

×	0	a	b	c
0	0	0	0	0
a	0	a	b	c
b	0	a	b	c
c	0	0	0	0

Is S a noncommutative ring? Justify your answer.

5.4 Develop a set of tables similar to Table 5.1 for GF(5).

5.5 Demonstrate that the set of polynomials whose coefficients form a field is a ring.

5.6 Let R be the field of real numbers. Let $R[x]$ be the ring of polynomials with coefficients in field R. State whether each of the following statements is true or false.

a. $R[x]$ is a commutative ring with unity, with multiplicative identity being the constant polynomial 1.

b. $f \in R[x]$ has a multiplicative inverse if and only if f is a non-zero constant.

c. $R[x]$ is also a field.

5.7 For polynomial arithmetic with coefficients in Z_{11}, perform the following calculations.

a. $(x^2 + 2x + 9)(x^3 + 11x^2 + x + 7)$

b. $(8x^2 + 3x + 2)(5x^2 + 6)$

5.8 Determine which of the following polynomials are reducible over GF(2).

a. $x^2 + 1$

b. $x^2 + x + 1$

c. $x^4 + x + 1$

5.9 Determine the gcd of the following pairs of polynomials.

a. $(x^3 + 1)$ and $(x^2 + x + 1)$ over GF(2)

b. $(x^3 + x + 1)$ and $(x^2 + 1)$ over GF(3)

c. $(x^3 - 2x + 1)$ and $(x^2 - x - 2)$ over GF(5)

d. $(x^4 + 8x^3 + 7x + 8)$ and $(2x^3 + 9x^2 + 10x + 1)$ over GF(11)

5.10 Develop a set of tables similar to Table 5.3 for GF(3) with $m(x) = x^2 + x + 1$.

5.11 Determine the multiplicative inverse of $x^2 + 1$ in GF(2^3) with $m(x) = x^3 + x - 1$.

5.12 Develop a table similar to Table 5.5 for GF(2^5) with $m(x) = x^5 + x^4 + x^3 + x + 1$.

Programming Problems

5.1 Write a simple four-function calculator in GF(2^4). You may use table lookups for the multiplicative inverses.

5.2 Write a simple four-function calculator in GF(2^8). You should compute the multiplicative inverses on the fly.

CHAPTER 6

ADVANCED ENCRYPTION STANDARD

LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- Present an overview of the general structure of Advanced Encryption Standard (AES).
- Understand the four transformations used in AES.
- Explain the AES key expansion algorithm.
- Understand the use of polynomials with coefficients in GF(2^8).

The **Advanced Encryption Standard (AES)** was published by the National Institute of Standards and Technology (NIST) in 2001. AES is a symmetric block cipher that is intended to replace DES as the approved standard for a wide range of applications.

[NECH01], available from NIST, summarizes the evaluation criteria used by NIST to select from among the candidates for AES, plus the rationale for picking Rijndael, which was the winning candidate. This material is useful in understanding not just the AES design but also the criteria by which to judge any symmetric encryption algorithm. The essence of the criteria was to develop an algorithm with a high level of security and good performance on a range of systems.

It is worth making additional comment about the performance of AES. Because of the popularity of AES, a number of efforts have been made to improve performance through both software and hardware optimization. Most notably, in 2008, Intel introduced the Advanced Encryption Standard New Instructions (AES-NI) as a hardware extension to the x86 instruction set to improve the speed of encryption and decryption. The AES-NI instruction enables x86 processors to achieve a performance of 0.64 cycles/byte for an authenticated encryption mode known as AES-GCM (described in Chapter 12).

In 2018, Intel added vectorized instructions, referred to as VAES*, to the existing AES-NI for its high-end processors [INTE17]. These instructions are intended to push the performance of AES software further down, to a new theoretical throughput of 0.16 cycles/byte [DRUC18].

AES has become the most widely used symmetric cipher. Compared to public-key ciphers such as RSA, the structure of AES and most symmetric ciphers is quite complex and cannot be explained as easily as many other cryptographic algorithms. Accordingly, the reader may wish to begin with a simplified version of AES, which is described in Appendix A. This version allows the reader to perform encryption and decryption by hand and gain a good understanding of the working of the algorithm details. Classroom experience indicates that a study of this simplified version enhances understanding of AES. One possible approach is to read the chapter first, then carefully read Appendix A and then re-read the main body of the chapter.

6.1 FINITE FIELD ARITHMETIC

In AES, all operations are performed on 8-bit bytes. In particular, the arithmetic operations of addition, multiplication, and division are performed over the finite field GF(2^8). Section 5.6 discusses such operations in some detail. For the reader who has not studied Chapter 5, and as a quick review for those who have, this section summarizes the important concepts.

In essence, a **field** is a set in which we can do addition, subtraction, multiplication, and division without leaving the set. Division is defined with the following rule: $a/b = a(b^{-1})$. An example of a **finite field** (one with a finite number of elements) is the set Z_p consisting of all the integers $\{0, 1, \ldots, p - 1\}$, where p is a prime number and in which arithmetic is carried out modulo p.

Virtually all encryption algorithms, both conventional and public-key, involve arithmetic operations on integers. If one of the operations used in the algorithm is division, then we need to work in arithmetic defined over a field; this is because division requires that each nonzero element have a multiplicative inverse. For convenience and for implementation efficiency, we would also like to work with integers that fit exactly into a given number of bits, with no wasted bit patterns. That is, we wish to work with integers in the range 0 through $2^n - 1$, which fit into an n-bit word. Unfortunately, the set of such integers, Z_{2^n}, using modular arithmetic, is not a field. For example, the integer 2 has no multiplicative inverse in Z_{2^n}, that is, there is no integer b, such that $2b \bmod 2^n = 1$.

There is a way of defining a finite field containing 2^n elements; such a field is referred to as GF(2^n). Consider the set, S, of all polynomials of degree $n - 1$ or less with binary coefficients. Thus, each polynomial has the form

$$f(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_1x + a_0 = \sum_{i=0}^{n-1} a_i x^i$$

where each a_i takes on the value 0 or 1. There are a total of 2^n different polynomials in S. For $n = 3$, the $2^3 = 8$ polynomials in the set are

$$\begin{array}{cccc} 0 & x & x^2 & x^2 + x \\ 1 & x + 1 & x^2 + 1 & x^2 + x + 1 \end{array}$$

With the appropriate definition of arithmetic operations, each such set S is a finite field. The definition consists of the following elements.

1. Arithmetic follows the ordinary rules of polynomial arithmetic using the basic rules of algebra with the following two refinements.
2. Arithmetic on the coefficients is performed modulo 2. This is the same as the XOR operation.
3. If multiplication results in a polynomial of degree greater than $n - 1$, then the polynomial is reduced modulo some irreducible polynomial $m(x)$ of degree n. That is, we divide by $m(x)$ and keep the remainder. For a polynomial $f(x)$, the remainder is expressed as $r(x) = f(x) \bmod m(x)$. A polynomial $m(x)$ is called irreducible if and only if $m(x)$ cannot be expressed as a product of two polynomials, both of degree lower than that of $m(x)$.

For example, to construct the finite field GF(2^3), we need to choose an irreducible polynomial of degree 3. There are only two such polynomials: $(x^3 + x^2 + 1)$ and $(x^3 + x + 1)$. Addition is equivalent to taking the XOR of like terms. Thus, $(x + 1) + x = 1$.

A polynomial in GF(2^n) can be uniquely represented by its n binary coefficients $(a_{n-1}a_{n-2} \ldots a_0)$. Therefore, every polynomial in GF(2^n) can be represented by an n-bit number. Addition is performed by taking the bitwise XOR of the two n-bit elements. There is no simple XOR operation that will accomplish multiplication in GF(2^n). However, a reasonably straightforward, easily implemented, technique is available. In essence, it can be shown that multiplication of a number in GF(2^n) by 2 consists of a left shift followed by a conditional XOR with a constant. Multiplication by larger numbers can be achieved by repeated application of this rule.

For example, AES uses arithmetic in the finite field GF(2^8) with the irreducible polynomial $m(x) = x^8 + x^4 + x^3 + x + 1$. Consider two elements $A = (a_7a_6 \ldots a_1a_0)$ and $B = (b_7b_6 \ldots b_1b_0)$. The sum $A + B = (c_7c_6 \ldots c_1c_0)$, where $c_i = a_i \oplus b_i$. The multiplication $\{02\} \cdot A$ equals $(a_6 \ldots a_1a_00)$ if $a_7 = 0$ and equals $(a_6 \ldots a_1a_00) \oplus (00011011)$ if $a_7 = 1$.[1]

To summarize, AES operates on 8-bit bytes. Addition of two bytes is defined as the bitwise XOR operation. Multiplication of two bytes is defined as multiplication in the finite field GF(2^8), with the irreducible polynomial[2] $m(x) = x^8 + x^4 + x^3 + x + 1$. The developers of Rijndael give as their motivation for selecting this one of the 30 possible irreducible polynomials of degree 8 that it is the first one on the list given in [LIDL94].

6.2 AES STRUCTURE

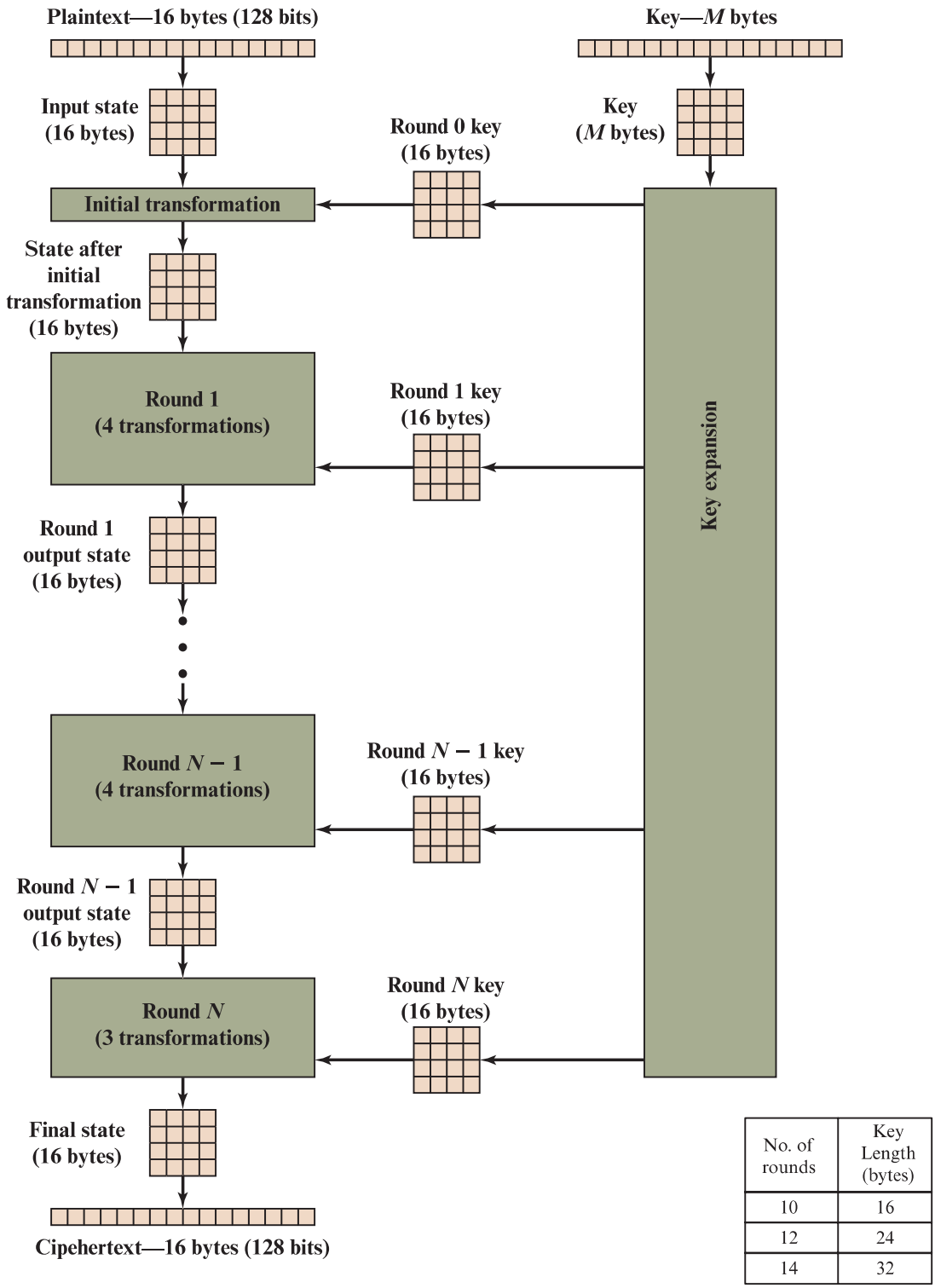
General Structure

Figure 6.1 shows the overall structure of the AES encryption process. The cipher takes a plaintext block size of 128 bits, or 16 bytes. The key length can be 16, 24, or 32 bytes (128, 192, or 256 bits). The algorithm is referred to as AES-128, AES-192, or AES-256, depending on the key length.

The input to the encryption and decryption algorithms is a single 128-bit block. In FIPS PUB 197, this block is depicted as a 4×4 square matrix of bytes. This block is copied into the **State** array, which is modified at each stage of encryption or decryption. After the final stage, **State** is copied to an output matrix. These operations are depicted in Figure 6.2a. Similarly, the key is depicted as a square matrix of bytes. This key is then expanded into an array of key schedule words. Figure 6.2b shows the expansion for the 128-bit key. Each word is four bytes, and the total key schedule is 44 words for the 128-bit key. Note that the ordering of bytes within a matrix is by column. So, for example, the first four bytes of a 128-bit plaintext input to the encryption cipher occupy the first column of the **in** matrix, the second four

[1] In FIPS PUB 197, a hexadecimal number is indicated by enclosing it in curly brackets. We use that convention in this chapter.

[2] In the remainder of this discussion, references to GF(2^8) refer to the finite field defined with this polynomial.

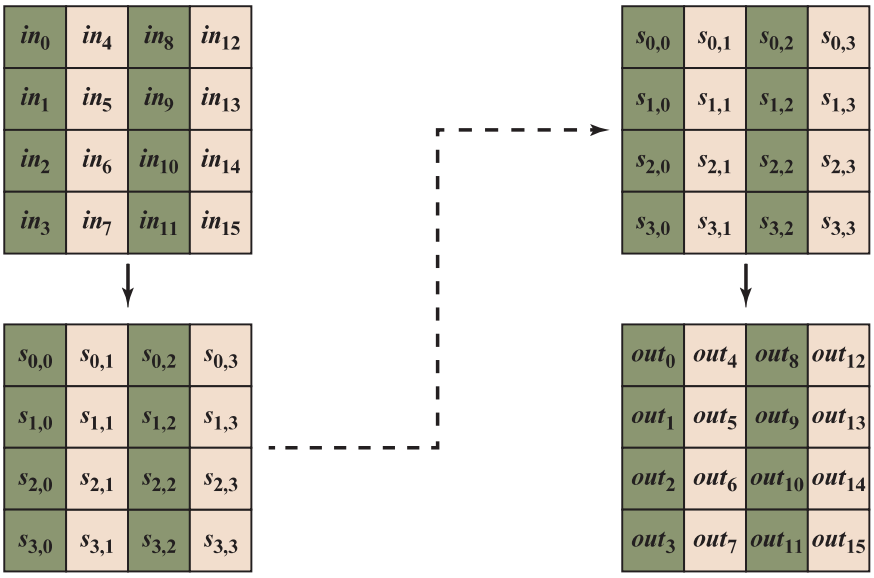


No. of rounds	Key Length (bytes)
10	16
12	24
14	32

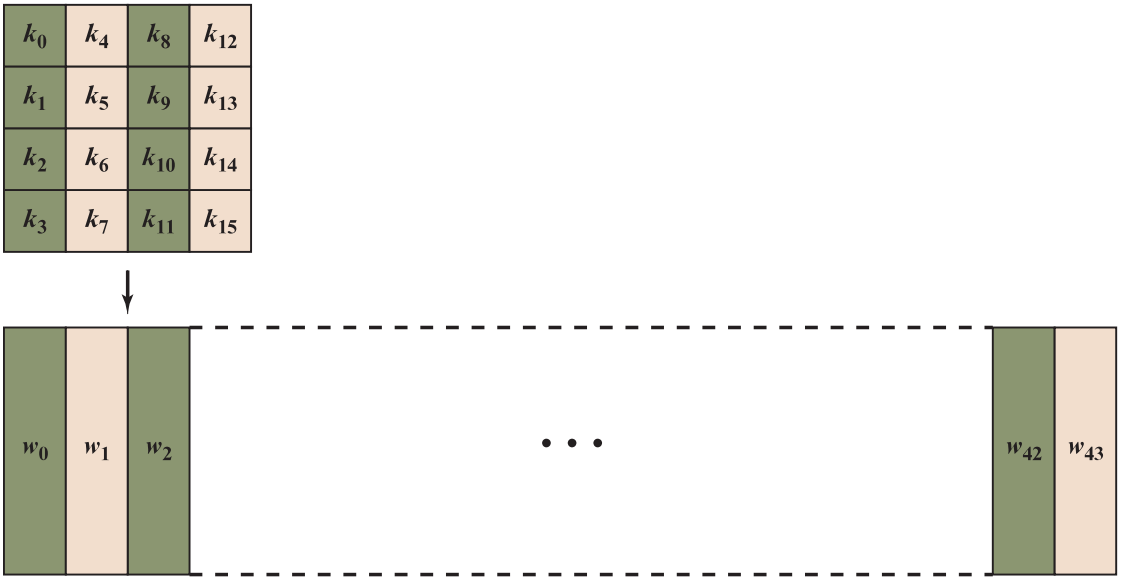
Figure 6.1 AES Encryption Process

bytes occupy the second column, and so on. Similarly, the first four bytes of the expanded key, which form a word, occupy the first column of the **w** matrix.

The cipher consists of N rounds, where the number of rounds depends on the key length: 10 rounds for a 16-byte key, 12 rounds for a 24-byte key, and 14 rounds for a 32-byte key (Table 6.1). The first $N - 1$ rounds consist of four distinct



(a) Input, state array, and output



(b) Key and expanded key

Figure 6.2 AES Data Structures

Table 6.1 AES Parameters

Key Size (words/bytes/bits)	4/16/128	6/24/192	8/32/256
Plaintext Block Size (words/bytes/bits)	4/16/128	4/16/128	4/16/128
Number of Rounds	10	12	14
Round Key Size (words/bytes/bits)	4/16/128	4/16/128	4/16/128
Expanded Key Size (words/bytes)	44/176	52/208	60/240

transformation functions: SubBytes, ShiftRows, MixColumns, and AddRoundKey, which are described subsequently. The final round contains only three transformations, and there is a initial single transformation (AddRoundKey) before the first round, which can be considered Round 0. Each transformation takes one or more 4×4 matrices as input and produces a 4×4 matrix as output. Figure 6.1 shows that the output of each round is a 4×4 matrix, with the output of the final round being the ciphertext. Also, the key expansion function generates $N + 1$ round keys, each of which is a distinct 4×4 matrix. Each round key serves as one of the inputs to the AddRoundKey transformation in each round.

Detailed Structure

Figure 6.3 shows the AES cipher in more detail, indicating the sequence of transformations in each round and showing the corresponding decryption function. As was done in Chapter 4, we show encryption proceeding down the page and decryption proceeding up the page.

Before delving into details, we can make several comments about the overall AES structure.

1. One noteworthy feature of this structure is that it is not a Feistel structure. Recall that, in the classic Feistel structure, half of the data block is used to modify the other half of the data block and then the halves are swapped. AES instead processes the entire data block as a single matrix during each round using substitutions and permutation.
2. The key that is provided as input is expanded into an array of forty-four 32-bit words, $\mathbf{w}[i]$. Four distinct words (128 bits) serve as a round key for each round; these are indicated in Figure 6.3.
3. Four different stages are used, one of permutation and three of substitution:
   - **Substitute bytes:** Uses an S-box to perform a byte-by-byte substitution of the block.
   - **ShiftRows:** A simple permutation.
   - **MixColumns:** A substitution that makes use of arithmetic over GF(2^8).
   - **AddRoundKey:** A simple bitwise XOR of the current block with a portion of the expanded key.
4. The structure is quite simple. For both encryption and decryption, the cipher begins with an AddRoundKey stage, followed by nine rounds that each includes all four stages, followed by a tenth round of three stages. Figure 6.4 depicts the structure of a full encryption round.
5. Only the AddRoundKey stage makes use of the key. For this reason, the cipher begins and ends with an AddRoundKey stage. Any other stage, applied at the beginning or end, is reversible without knowledge of the key and so would add no security.

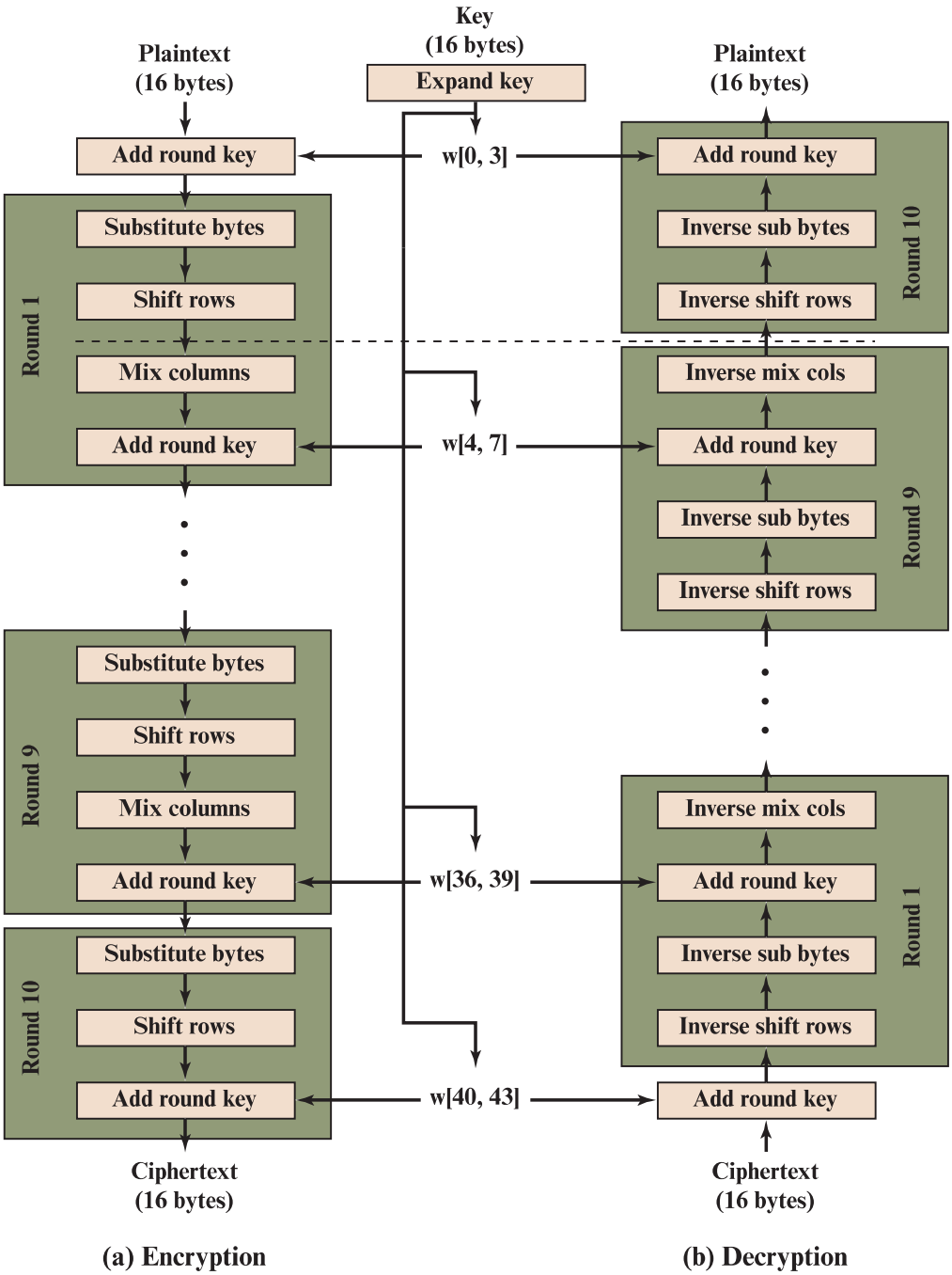


Figure 6.3 AES Encryption and Decryption

6. The AddRoundKey stage is, in effect, a form of Vernam cipher and by itself would not be formidable. The other three stages together provide confusion, diffusion, and nonlinearity, but by themselves would provide no security because they do not use the key. We can view the cipher as alternating operations of XOR encryption (AddRoundKey) of a block, followed by scrambling of the block (the other three stages), followed by XOR encryption, and so on. This scheme is both efficient and highly secure.

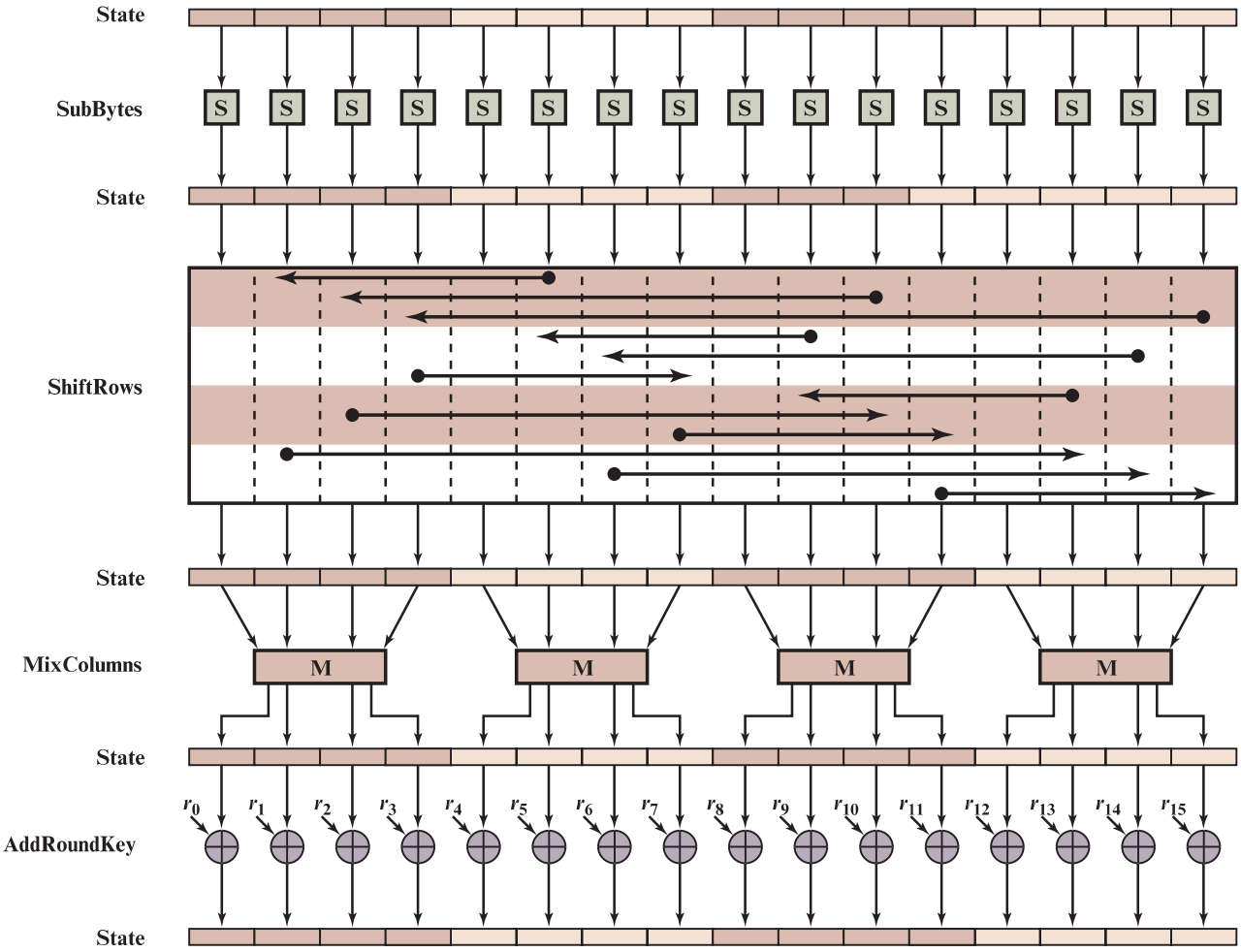


Figure 6.4 AES Encryption Round

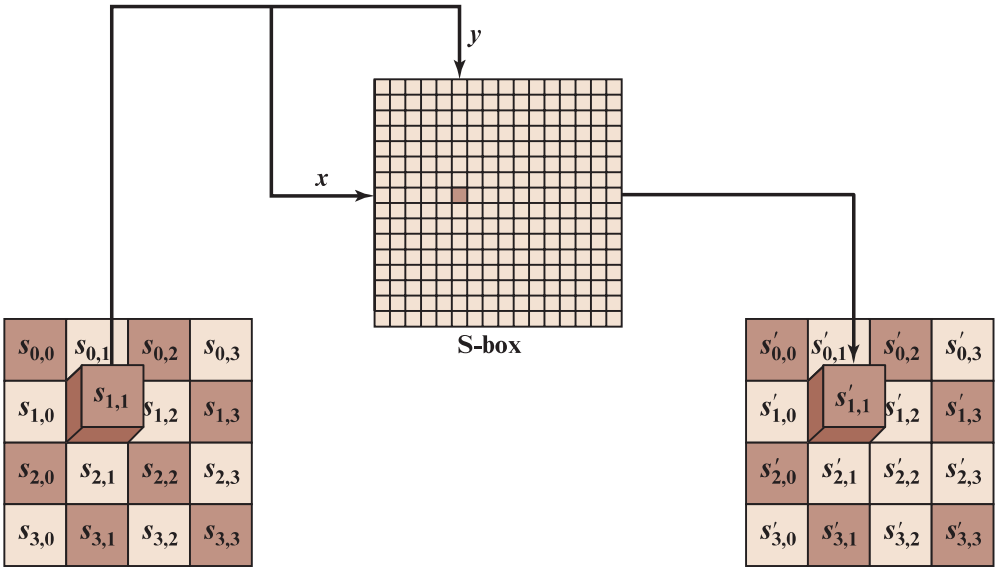
7. Each stage is easily reversible. For the Substitute Byte, ShiftRows, and MixColumns stages, an inverse function is used in the decryption algorithm. For the AddRoundKey stage, the inverse is achieved by XORing the same round key to the block, using the result that $A \oplus B \oplus B = A$.
8. As with most block ciphers, the decryption algorithm makes use of the expanded key in reverse order. However, the decryption algorithm is not identical to the encryption algorithm. This is a consequence of the particular structure of AES.
9. Once it is established that all four stages are reversible, it is easy to verify that decryption does recover the plaintext. Figure 6.3 lays out encryption and decryption going in opposite vertical directions. At each horizontal point (e.g., the dashed line in the figure), **State** is the same for both encryption and decryption.
10. The final round of both encryption and decryption consists of only three stages. Again, this is a consequence of the particular structure of AES and is required to make the cipher reversible.

6.3 AES TRANSFORMATION FUNCTIONS

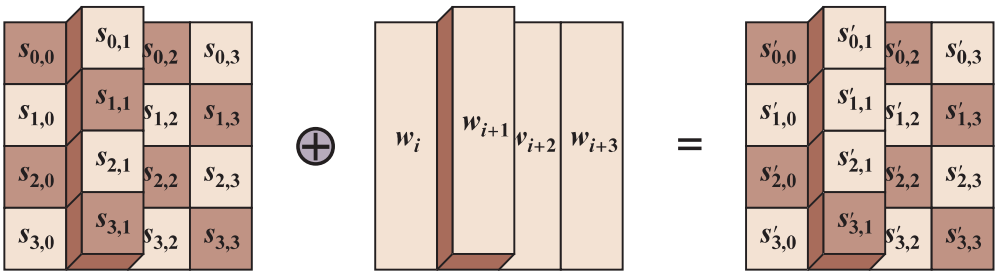
We now turn to a discussion of each of the four transformations used in AES. For each stage, we describe the forward (encryption) algorithm, the inverse (decryption) algorithm, and the rationale for the stage.

Substitute Bytes Transformation

FORWARD AND INVERSE TRANSFORMATIONS The **forward substitute byte transformation**, called SubBytes, is a simple table lookup (Figure 6.5a). AES defines a 16 × 16 matrix of byte values, called an **S-box** (Table 6.2a), that contains a permutation of all possible 256 8-bit values. Each individual byte of **State** is mapped into a new byte in the following way: The leftmost 4 bits of the byte are used as a row value and the rightmost 4 bits are used as a column value. These row and column values serve as indexes into the S-box to select a unique 8-bit output value.



(a) Substitute byte transformation



(b) Add round key transformation

Figure 6.5 AES Byte-Level Operations

Table 6.2 AES S-Boxes

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

(a) S-box

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

(b) Inverse S-box

For example, the hexadecimal value {95} references row 9, column 5 of the S-box, which contains the value {2A}. Accordingly, the value {95} is mapped into the value {2A}.

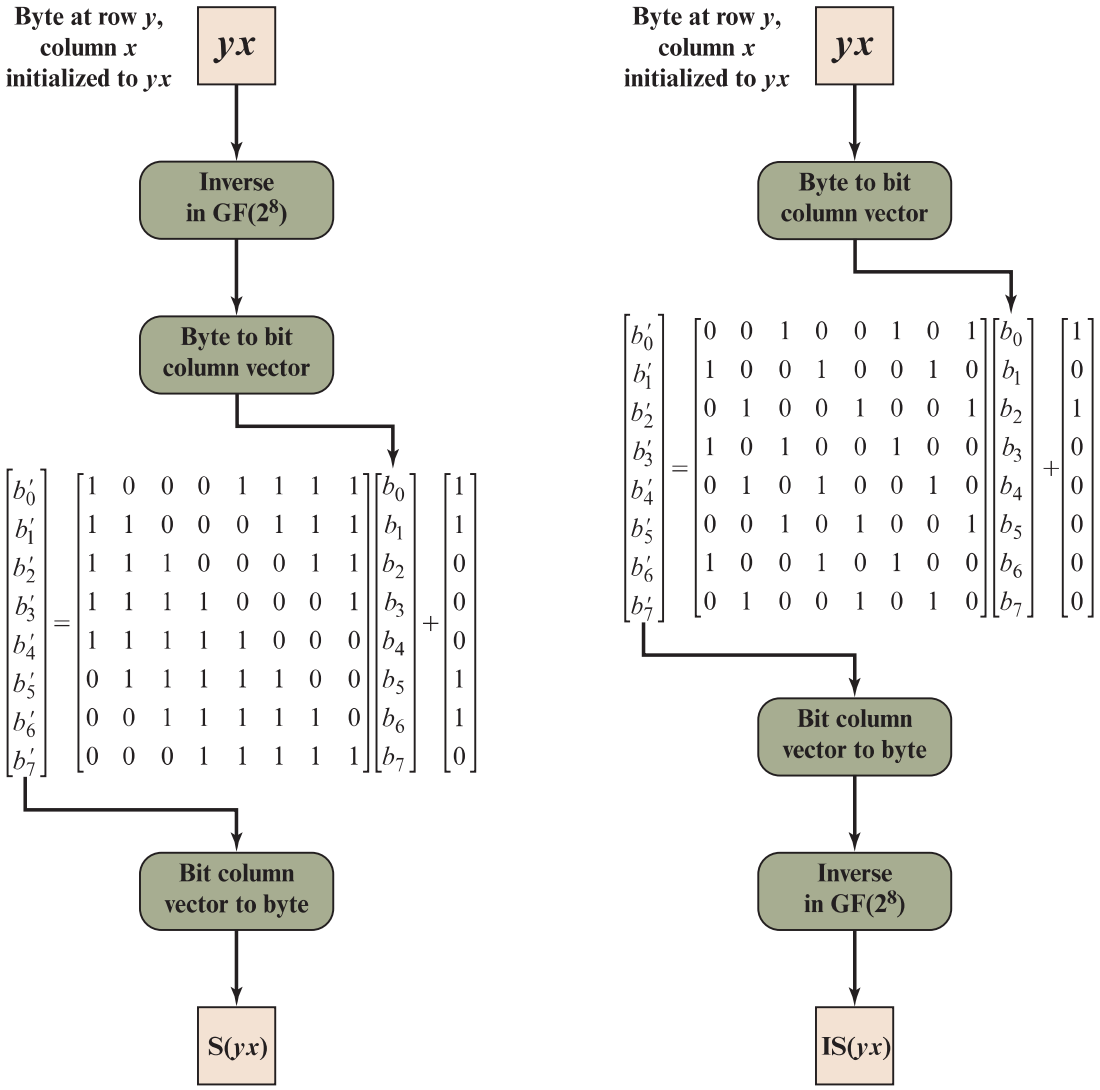
Here is an example of the SubBytes transformation:

EA	04	65	85
83	45	5D	96
5C	33	98	B0
F0	2D	AD	C5

→

87	F2	4D	97
EC	6E	4C	90
4A	C3	46	E7
8C	D8	95	A6

The S-box is constructed in the following fashion (Figure 6.6a).



(a) Calculation of byte at row *y*, column *x* of S-box

(a) Calculation of byte at row *y*, column *x* of IS-box

Figure 6.6 Constuction of S-Box and IS-Box

1. Initialize the S-box with the byte values in ascending sequence row by row. The first row contains {00}, {01}, {02}, . . . , {0F}; the second row contains {10}, {11}, etc.; and so on. Thus, the value of the byte at row y, column x is $\{yx\}$.
2. Map each byte in the S-box to its multiplicative inverse in the finite field GF(2^8); the value {00} is mapped to itself.
3. Consider that each byte in the S-box consists of 8 bits labeled ($b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0$). Apply the following transformation to each bit of each byte in the S-box:

$$b_i' = b_i \oplus b_{(i+4) \bmod 8} \oplus b_{(i+5) \bmod 8} \oplus b_{(i+6) \bmod 8} \oplus b_{(i+7) \bmod 8} \oplus c_i \quad \textbf{(6.1)}$$

where c_i is the ith bit of byte c with the value {63}; that is, $(c_7c_6c_5c_4c_3c_2c_1c_0) = (01100011)$. The prime ($'$) indicates that the variable is to be updated by the value on the right. The AES standard depicts this transformation in matrix form as follows.

$$\begin{bmatrix} b_0' \\ b_1' \\ b_2' \\ b_3' \\ b_4' \\ b_5' \\ b_6' \\ b_7' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \quad \textbf{(6.2)}$$

Equation (6.2) has to be interpreted carefully. In ordinary matrix multiplication,[3] each element in the product matrix is the sum of products of the elements of one row and one column. In this case, each element in the product matrix is the bitwise XOR of products of elements of one row and one column. Furthermore, the final addition shown in Equation (6.2) is a bitwise XOR. Recall from Section 5.6 that the bitwise XOR is addition in GF(2^8).

As an example, consider the input value {95}. The multiplicative inverse in GF(2^8) is $\{95\}^{-1} = \{8A\}$, which is 10001010 in binary. Using Equation (6.2),

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \oplus \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \oplus \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

[3]For a brief review of the rules of matrix and vector multiplication, refer to Appendix B.

The result is {2A}, which should appear in row {09} column {05} of the S-box. This is verified by checking Table 6.2a.

The **inverse substitute byte transformation**, called InvSubBytes, makes use of the inverse S-box shown in Table 6.2b. Note, for example, that the input {2A} produces the output {95}, and the input {95} to the S-box produces {2A}. The inverse S-box is constructed (Figure 6.6b) by applying the inverse of the transformation in Equation (6.1) followed by taking the multiplicative inverse in GF(2^8). The inverse transformation is

$$b'_i = b_{(i+2) \bmod 8} \oplus b_{(i+5) \bmod 8} \oplus b_{(i+7) \bmod 8} \oplus d_i$$

where byte $d = \{05\}$, or 00000101. We can depict this transformation as follows.

$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

To see that InvSubBytes is the inverse of SubBytes, label the matrices in SubBytes and InvSubBytes as **X** and **Y**, respectively, and the vector versions of constants c and d as **C** and **D**, respectively. For some 8-bit vector **B**, Equation (6.2) becomes $\mathbf{B}' = \mathbf{XB} \oplus \mathbf{C}$. We need to show that $\mathbf{Y}(\mathbf{XB} \oplus \mathbf{C}) \oplus \mathbf{D} = \mathbf{B}$. To multiply out, we must show $\mathbf{YXB} \oplus \mathbf{YC} \oplus \mathbf{D} = \mathbf{B}$. This becomes

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} \oplus$$

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \oplus \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} =$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} \oplus \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \oplus \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix}$$

We have demonstrated that **YX** equals the identity matrix, and the **YC** = **D**, so that **YC** $\oplus$ **D** equals the null vector.

Rationale 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 is not a linear mathematical function of the input [DAEM01]. The nonlinearity is due to the use of the multiplicative inverse. In addition, the constant in Equation (6.1) was chosen so that the S-box has no fixed points [S-box(a) = a] and no "opposite fixed points" [S-box(a) = $\overline{a}$], where $\overline{a}$ is the bitwise complement of a.

Of course, the S-box must be invertible, that is, IS-box[S-box(a)] = a. However, the S-box does not self-inverse in the sense that it is not true that S-box(a) = IS-box(a). For example, S-box({95}) = {2A}, but IS-box({95}) = {AD}.

ShiftRows Transformation

Forward and Inverse Transformations The **forward shift row transformation**, called ShiftRows, is depicted in Figure 6.7a. The first row of **State** is not altered. For the second row, a 1-byte circular left shift is performed. For the third row, a 2-byte circular left shift is performed. For the fourth row, a 3-byte circular left shift is performed. The following is an example of ShiftRows.

87	F2	4D	97
EC	6E	4C	90
4A	C3	46	E7
8C	D8	95	A6

→

87	F2	4D	97
6E	4C	90	EC
46	E7	4A	C3
A6	8C	D8	95

The **inverse shift row transformation**, called InvShiftRows, performs the circular shifts in the opposite direction for each of the last three rows, with a 1-byte circular right shift for the second row, and so on.

Rationale The shift row transformation is more substantial than it may first appear. This is because the **State**, as well as the cipher input and output, is treated as an array of four 4-byte columns. Thus, on encryption, the first 4 bytes of the plaintext are copied to the first column of **State**, and so on. Furthermore, as will be seen, the round key is applied to **State** column by column. Thus, a row shift moves an individual byte from one column to another, which is a linear distance

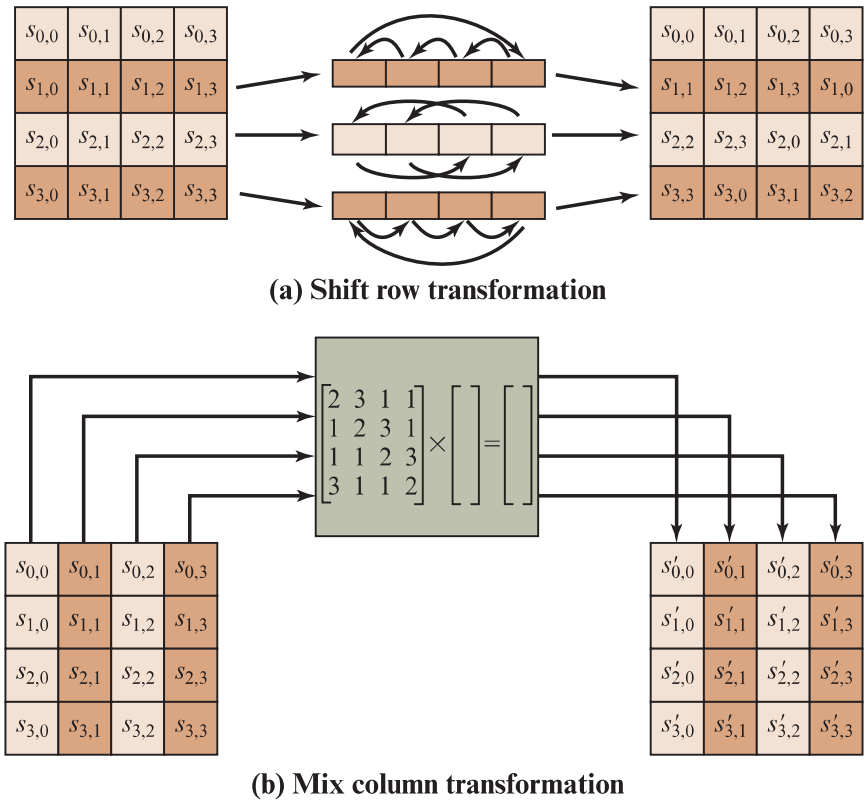


Figure 6.7 AES Row and Column Operations

of a multiple of 4 bytes. Also note that the transformation ensures that the 4 bytes of one column are spread out to four different columns. Figure 6.4 illustrates the effect.

MixColumns Transformation

Forward and Inverse Transformations The **forward mix column transformation**, called MixColumns, operates on each column individually. Each byte of a column is mapped into a new value that is a function of all four bytes in that column. The transformation can be defined by the following matrix multiplication on **State** (Figure 6.7b):

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} = \begin{bmatrix} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix} \quad (6.3)$$

Each element in the product matrix is the sum of products of elements of one row and one column. In this case, the individual additions and multiplications[4] are

[4]We follow the convention of FIPS PUB 197 and use the symbol · to indicate multiplication over the finite field GF(2^8) and $\oplus$ to indicate bitwise XOR, which corresponds to addition in GF(2^8).

performed in GF(2^8). The MixColumns transformation on a single column of **State** can be expressed as

$$\begin{aligned} s'_{0,j} &= (2 \cdot s_{0,j}) \oplus (3 \cdot s_{1,j}) \oplus s_{2,j} \oplus s_{3,j} \\ s'_{1,j} &= s_{0,j} \oplus (2 \cdot s_{1,j}) \oplus (3 \cdot s_{2,j}) \oplus s_{3,j} \\ s'_{2,j} &= s_{0,j} \oplus s_{1,j} \oplus (2 \cdot s_{2,j}) \oplus (3 \cdot s_{3,j}) \\ s'_{3,j} &= (3 \cdot s_{0,j}) \oplus s_{1,j} \oplus s_{2,j} \oplus (2 \cdot s_{3,j}) \end{aligned} \tag{6.4}$$

The following is an example of MixColumns:

87	F2	4D	97
6E	4C	90	EC
46	E7	4A	C3
A6	8C	D8	95

→

47	40	A3	4C
37	D4	70	9F
94	E4	3A	42
ED	A5	A6	BC

Let us verify the first column of this example. Recall from Section 5.6 that, in GF(2^8), addition is the bitwise XOR operation and that multiplication can be performed according to the rule established in Equation (5.6). In particular, multiplication of a value by x (i.e., by {02}) can be implemented as a 1-bit left shift followed by a conditional bitwise XOR with (0001 1011) if the leftmost bit of the original value (prior to the shift) is 1. Thus, to verify the MixColumns transformation on the first column, we need to show that

$$\begin{aligned} (\{02\} \cdot \{87\}) &\oplus (\{03\} \cdot \{6E\}) \oplus \{46\} \oplus \{A6\} = \{47\} \\ \{87\} &\oplus (\{02\} \cdot \{6E\}) \oplus (\{03\} \cdot \{46\}) \oplus \{A6\} = \{37\} \\ \{87\} &\oplus \{6E\} \oplus (\{02\} \cdot \{46\}) \oplus (\{03\} \cdot \{A6\}) = \{94\} \\ (\{03\} \cdot \{87\}) &\oplus \{6E\} \oplus \{46\} \oplus (\{02\} \cdot \{A6\}) = \{ED\} \end{aligned}$$

For the first equation, we have $\{02\} \cdot \{87\} = (0000\ 1110) \oplus (0001\ 1011) = (0001\ 0101)$ and $\{03\} \cdot \{6E\} = \{6E\} \oplus (\{02\} \cdot \{6E\}) = (0110\ 1110) \oplus (1101\ 1100) = (1011\ 0010)$. Then,

$$\begin{aligned} \{02\} \cdot \{87\} &= 0001\ 0101 \\ \{03\} \cdot \{6E\} &= 1011\ 0010 \\ \{46\} &= 0100\ 0110 \\ \{A6\} &= \underline{1010\ 0110} \\ & \quad 0100\ 0111 = \{47\} \end{aligned}$$

The other equations can be similarly verified.

The **inverse mix column transformation**, called InvMixColumns, is defined by the following matrix multiplication:

$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} = \begin{bmatrix} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix} \tag{6.5}$$

It is not immediately clear that Equation (6.5) is the **inverse** of Equation (6.3). We need to show

$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} = \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{0,3} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix}$$

which is equivalent to showing

$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.6}$$

That is, the inverse transformation matrix times the forward transformation matrix equals the identity matrix. To verify the first column of Equation (6.6), we need to show

$$\begin{aligned} (\{0E\} \cdot \{02\}) \oplus \{0B\} \oplus \{0D\} \oplus (\{09\} \cdot \{03\}) &= \{01\} \\ (\{09\} \cdot \{02\}) \oplus \{0E\} \oplus \{0B\} \oplus (\{0D\} \cdot \{03\}) &= \{00\} \\ (\{0D\} \cdot \{02\}) \oplus \{09\} \oplus \{0E\} \oplus (\{0B\} \cdot \{03\}) &= \{00\} \\ (\{0B\} \cdot \{02\}) \oplus \{0D\} \oplus \{09\} \oplus (\{0E\} \cdot \{03\}) &= \{00\} \end{aligned}$$

For the first equation, we have $\{0E\} \cdot \{02\} = 00011100$ and $\{09\} \cdot \{03\} = \{09\} \oplus (\{09\} \cdot \{02\}) = 00001001 \oplus 00010010 = 00011011$. Then

$$\begin{aligned} \{0E\} \cdot \{02\} &= 00011100 \\ \{0B\} &= 00001011 \\ \{0D\} &= 00001101 \\ \{09\} \cdot \{03\} &= \underline{00011011} \\ & \quad 00000001 \end{aligned}$$

The other equations can be similarly verified.

The AES document describes another way of characterizing the MixColumns transformation, which is in terms of polynomial arithmetic. In the standard, MixColumns is defined by considering each column of **State** to be a four-term polynomial with coefficients in GF(2^8). Each column is multiplied modulo ($x^4 + 1$) by the fixed polynomial $a(x)$, given by

$$a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\} \tag{6.7}$$

Appendix 6A demonstrates that multiplication of each column of **State** by $a(x)$ can be written as the matrix multiplication of Equation (6.3). Similarly, it can be seen that the transformation in Equation (6.5) corresponds to treating

each column as a four-term polynomial and multiplying each column by $b(x)$, given by

$$b(x) = \{0B\}x^3 + \{0D\}x^2 + \{09\}x + \{0E\} \quad \textbf{(6.8)}$$

It readily can be shown that $b(x) = a^{-1}(x) \bmod (x^4 + 1)$.

Rationale The coefficients of the matrix in Equation (6.3) are based on a linear code with maximal distance between code words, which ensures a good mixing among the bytes of each column. The mix column transformation combined with the shift row transformation ensures that after a few rounds all output bits depend on all input bits. See [DAEM99] for a discussion.

In addition, the choice of coefficients in MixColumns, which are all {01}, {02}, or {03}, was influenced by implementation considerations. As was discussed, multiplication by these coefficients involves at most a shift and an XOR. The coefficients in InvMixColumns are more formidable to implement. However, encryption was deemed more important than decryption for two reasons:

1. For the CFB and OFB cipher modes (Figures 7.5 and 7.6; described in Chapter 7), only encryption is used.
2. As with any block cipher, AES can be used to construct a message authentication code (Chapter 12), and for this, only encryption is used.

AddRoundKey Transformation

Forward and Inverse Transformations In the **forward add round key transformation**, called AddRoundKey, the 128 bits of **State** are bitwise XORed with the 128 bits of the round key. As shown in Figure 6.5b, the operation is viewed as a columnwise operation between the 4 bytes of a **State** column and one word of the round key; it can also be viewed as a byte-level operation. The following is an example of AddRoundKey:

47	40	A3	4C
37	D4	70	9F
94	E4	3A	42
ED	A5	A6	BC

$\oplus$

AC	19	28	57
77	FA	D1	5C
66	DC	29	00
F3	21	41	6A

$=$

EB	59	8B	1B
40	2E	A1	C3
F2	38	13	42
1E	84	E7	D6

The first matrix is **State**, and the second matrix is the round key.

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

Rationale The add round key transformation is as simple as possible and affects every bit of **State**. The complexity of the round key expansion, plus the complexity of the other stages of AES, ensure security.

Figure 6.8 is another view of a single round of AES, emphasizing the mechanisms and inputs of each transformation.

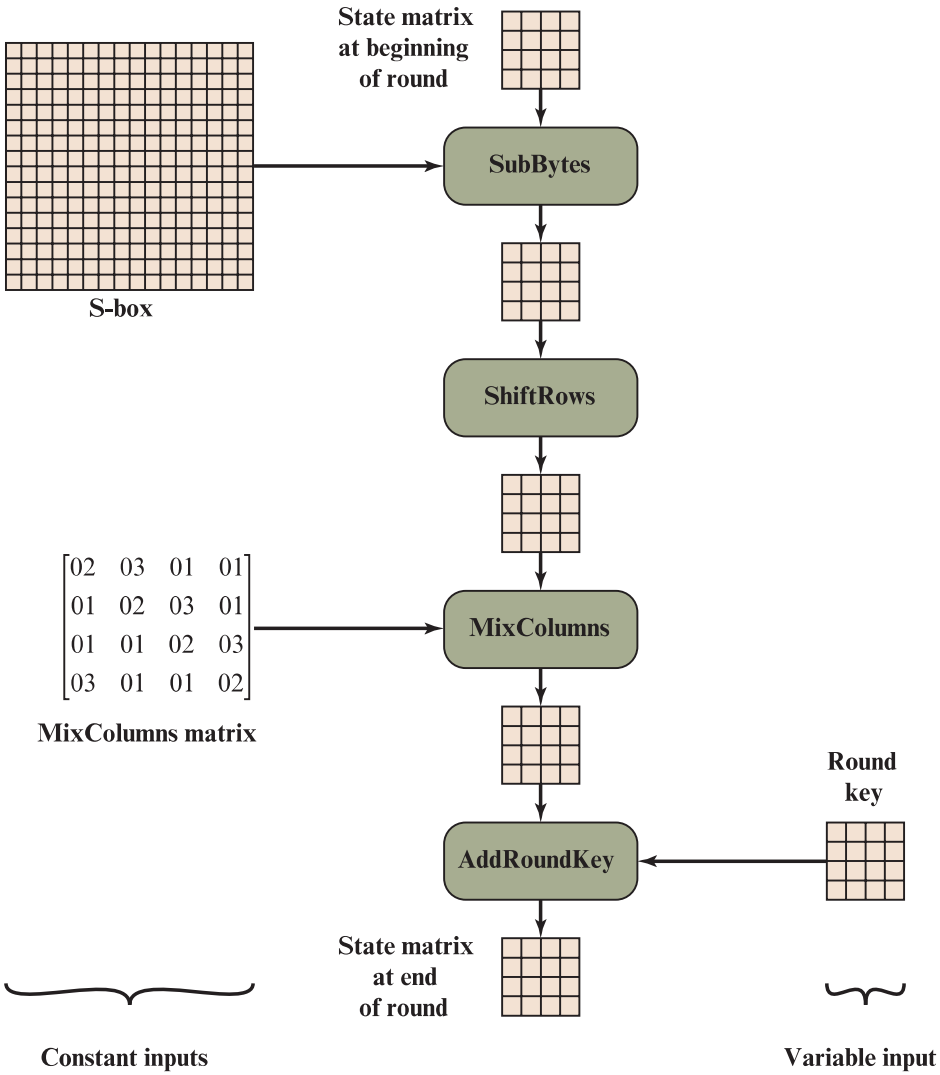


Figure 6.8 Inputs for Single AES Round

6.4 AES KEY EXPANSION

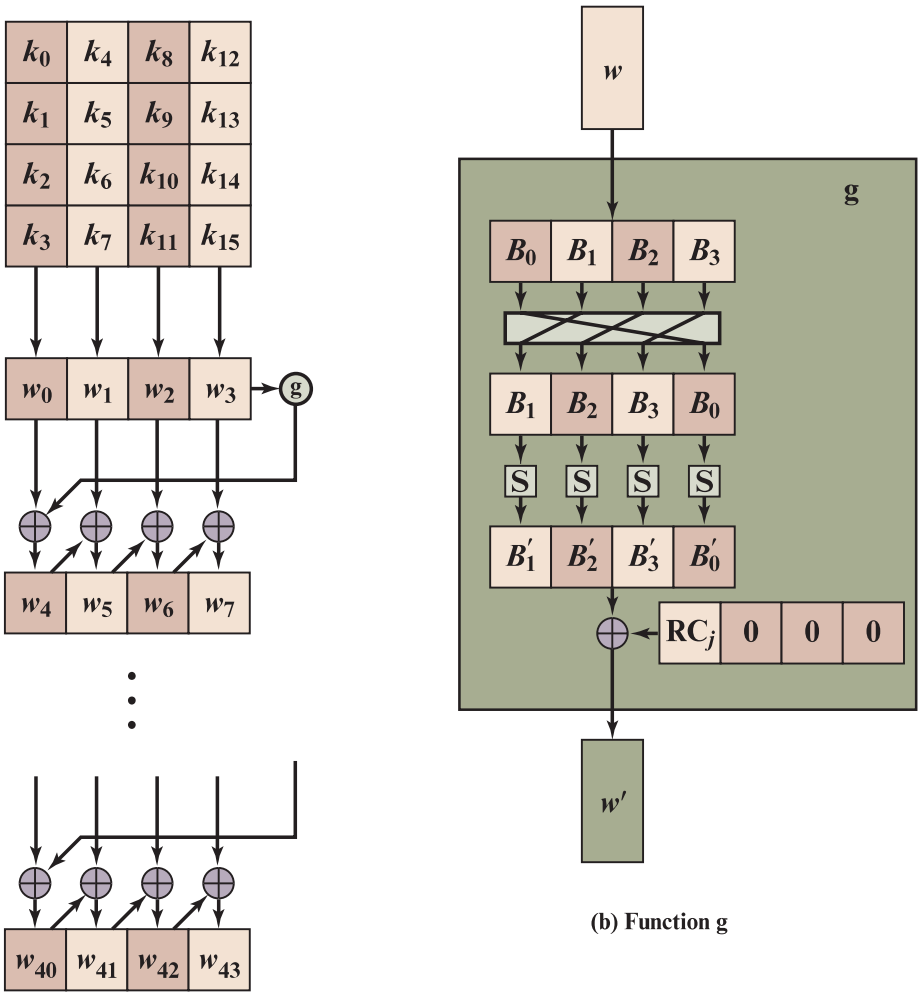
Key Expansion Algorithm

The AES **key expansion** algorithm takes as input a four-word (16-byte) key and produces a linear array of 44 words (176 bytes). This is sufficient to provide a four-word round key for the initial AddRoundKey stage and each of the 10 rounds of the cipher. The pseudocode on the next page describes the expansion.

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 6.9 illustrates the generation of the expanded key, using the symbol g to represent that complex function. The function g consists of the following subfunctions.

```
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 mod 4 = 0) temp = SubWord (RotWord (temp))
                                ⊕ Rcon[i/4];
    w[i] = w[i-4] ⊕ temp
    }
}
```



(a) Overall algorithm

(b) Function g

Figure 6.9 AES Key Expansion

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 (Table 6.2a).
3. The result of steps 1 and 2 is XORed with a 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 · RC[j − 1] and with multiplication defined over the field GF(2^8). The values of RC[j] in hexadecimal are

j	1	2	3	4	5	6	7	8	9	10
RC[j]	01	02	04	08	10	20	40	80	1B	36

For example, suppose that the round key for round 8 is

EA D2 73 21 B5 8D BA D2 31 2B F5 60 7F 8D 29 2F

Then the first 4 bytes (first column) of the round key for round 9 are calculated as shown in Table 6.3

Table 6.3 Example Round Key Calculation

Description	Value
i (decimal)	36
temp = w[i − 1]	7F8D292F
RotWord (temp)	8D292F7F
SubWord (RotWord (temp))	5DA515D2
Rcon (9)	1B000000
SubWord (RotWord (temp)) ⊕ Rcon (9)	46A515D2
w[i − 4]	EAD27321
w[i] = w[i − 4] ⊕ SubWord (RotWord (temp)) ⊕ Rcon (9)	AC7766F3

Rationale

The Rijndael developers designed the expansion key algorithm to be resistant to known cryptanalytic attacks. The inclusion of a round-dependent round constant eliminates the symmetry, or similarity, between the ways in which round keys are generated in different rounds. The specific criteria that were used are [DAEM99]

- Knowledge of a part of the cipher key or round key does not enable calculation of many other round-key bits.
- An invertible transformation [i.e., knowledge of any *Nk* consecutive words of the expanded key enables regeneration of the entire expanded key (*Nk* = key size in words)].
- Speed on a wide range of processors.

- Usage of round constants to eliminate symmetries.
- Diffusion of cipher key differences into the round keys; that is, each key bit affects many round key bits.
- Enough nonlinearity to prohibit the full determination of round key differences from cipher key differences only.
- Simplicity of description.

The authors do not quantify the first point on the preceding list, but the idea is that if you know less than *Nk* consecutive words of either the cipher key or one of the round keys, then it is difficult to reconstruct the remaining unknown bits. The fewer bits one knows, the more difficult it is to do the reconstruction or to determine other bits in the key expansion.

6.5 AN AES EXAMPLE

We now work through an example and consider some of its implications. Although you are not expected to duplicate the example by hand, you will find it informative to study the hex patterns that occur from one step to the next.

For this example, the plaintext is a hexadecimal palindrome. The plaintext, key, and resulting ciphertext are

Plaintext:	`0123456789abcdeffedcba9876543210`
Key:	`0f1571c947d9e8590cb7add6af7f6798`
Ciphertext:	`ff0b844a0853bf7c6934ab4364148fb9`

Results

Table 6.4 shows the expansion of the 16-byte key into 10 round keys. As previously explained, this process is performed word by word, with each four-byte word occupying one column of the word round-key matrix. The left-hand column shows the four round-key words generated for each round. The right-hand column shows

Table 6.4 Key Expansion for AES Example

Key Words	Auxiliary Function
w0 = 0f 15 71 c9 w1 = 47 d9 e8 59 w2 = 0c b7 ad d6 w3 = af 7f 67 98	RotWord (w3) = 7f 67 98 af = x1 SubWord (x1) = d2 85 46 79 = y1 Rcon (1) = 01 00 00 00 y1 ⊕ Rcon (1) = d3 85 46 79 = z1
w4 = w0 ⊕ z1 = dc 90 37 b0 w5 = w4 ⊕ w1 = 9b 49 df e9 w6 = w5 ⊕ w2 = 97 fe 72 3f w7 = w6 ⊕ w3 = 38 81 15 a7	RotWord (w7) = 81 15 a7 38 = x2 SubWord (x2) = 0c 59 5c 07 = y2 Rcon (2) = 02 00 00 00 y2 ⊕ Rcon (2) = 0e 59 5c 07 = z2
w8 = w4 ⊕ z2 = d2 c9 6b b7 w9 = w8 ⊕ w5 = 49 80 b4 5e w10 = w9 ⊕ w6 = de 7e c6 61 w11 = w10 ⊕ w7 = e6 ff d3 c6	RotWord (w11) = ff d3 c6 e6 = x3 SubWord (x3) = 16 66 b4 83 = y3 Rcon (3) = 04 00 00 00 y3 ⊕ Rcon (3) = 12 66 b4 8e = z3

(Continued)

Table 6.4 Continued

Key Words	Auxiliary Function
w12 = w8 ⊕ z3 = c0 af df 39 w13 = w12 ⊕ w9 = 89 2f 6b 67 w14 = w13 ⊕ w10 = 57 51 ad 06 w15 = w14 ⊕ w11 = b1 ae 7e c0	RotWord (w15) = ae 7e c0 b1 = x4 SubWord (x4) = e4 f3 ba c8 = y4 Rcon (4) = 08 00 00 00 y4 ⊕ Rcon (4) = ec f3 ba c8 = 4
w16 = w12 ⊕ z4 = 2c 5c 65 f1 w17 = w16 ⊕ w13 = a5 73 0e 96 w18 = w17 ⊕ w14 = f2 22 a3 90 w19 = w18 ⊕ w15 = 43 8c dd 50	RotWord (w19) = 8c dd 50 43 = x5 SubWord (x5) = 64 c1 53 1a = y5 Rcon(5) = 10 00 00 00 y5 ⊕ Rcon (5) = 74 c1 53 1a = z5
w20 = w16 ⊕ z5 = 58 9d 36 eb w21 = w20 ⊕ w17 = fd ee 38 7d w22 = w21 ⊕ w18 = 0f cc 9b ed w23 = w22 ⊕ w19 = 4c 40 46 bd	RotWord (w23) = 40 46 bd 4c = x6 SubWord (x6) = 09 5a 7a 29 = y6 Rcon(6) = 20 00 00 00 y6 ⊕ Rcon(6) = 29 5a 7a 29 = z6
w24 = w20 ⊕ z6 = 71 c7 4c c2 w25 = w24 ⊕ w21 = 8c 29 74 bf w26 = w25 ⊕ w22 = 83 e5 ef 52 w27 = w26 ⊕ w23 = cf a5 a9 ef	RotWord (w27) = a5 a9 ef cf = x7 SubWord (x7) = 06 d3 bf 8a = y7 Rcon (7) = 40 00 00 00 y7 ⊕ Rcon(7) = 46 d3 df 8a = z7
w28 = w24 ⊕ z7 = 37 14 93 48 w29 = w28 ⊕ w25 = bb 3d e7 f7 w30 = w29 ⊕ w26 = 38 d8 08 a5 w31 = w30 ⊕ w27 = f7 7d a1 4a	RotWord (w31) = 7d a1 4a f7 = x8 SubWord (x8) = ff 32 d6 68 = y8 Rcon (8) = 80 00 00 00 y8 ⊕ Rcon(8) = 7f 32 d6 68 = z8
w32 = w28 ⊕ z8 = 48 26 45 20 w33 = w32 ⊕ w29 = f3 1b a2 d7 w34 = w33 ⊕ w30 = cb c3 aa 72 w35 = w34 ⊕ w32 = 3c be 0b 3	RotWord (w35) = be 0b 38 3c = x9 SubWord (x9) = ae 2b 07 eb = y9 Rcon (9) = 1B 00 00 00 y9 ⊕ Rcon (9) = b5 2b 07 eb = z9
w36 = w32 ⊕ z9 = fd 0d 42 cb w37 = w36 ⊕ w33 = 0e 16 e0 1c w38 = w37 ⊕ w34 = c5 d5 4a 6e w39 = w38 ⊕ w35 = f9 6b 41 56	RotWord (w39) = 6b 41 56 f9 = x10 SubWord (x10) = 7f 83 b1 99 = y10 Rcon (10) = 36 00 00 00 y10 ⊕ Rcon (10) = 49 83 b1 99 = z10
w40 = w36 ⊕ z10 = b4 8e f3 52 w41 = w40 ⊕ w37 = ba 98 13 4e w42 = w41 ⊕ w38 = 7f 4d 59 20 w43 = w42 ⊕ w39 = 86 26 18 76	

the steps used to generate the auxiliary word used in key expansion. We begin, of course, with the key itself serving as the round key for round 0.

Next, Table 6.5 shows the progression of **State** through the AES encryption process. The first column shows the value of **State** at the start of a round. For the first row, **State** is just the matrix arrangement of the plaintext. The second, third, and fourth columns show the value of **State** for that round after the SubBytes, ShiftRows, and MixColumns transformations, respectively. The fifth column shows the round key. You can verify that these round keys equate with those shown in Table 6.4. The first column shows the value of **State** resulting from the bitwise XOR of **State** after the preceding MixColumns with the round key for the preceding round.

Avalanche Effect

If a small change in the key or plaintext were to produce a corresponding small change in the ciphertext, this might be used to effectively reduce the size of the

Table 6.5 AES Example

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

Table 6.6 Avalanche Effect in AES: Change in Plaintext

Round		Number of Bits that Differ
	0123456789abcdeffedcba9876543210 0023456789abcdeffedcba9876543210	1
0	0e3634aece7225b6f26b174ed92b5588 0f3634aece7225b6f26b174ed92b5588	1
1	657470750fc7ff3fc0e8e8ca4dd02a9c c4a9ad090fc7ff3fc0e8e8ca4dd02a9c	20
2	5c7bb49a6b72349b05a2317ff46d1294 fe2ae569f7ee8bb8c1f5a2bb37ef53d5	58
3	7115262448dc747e5cdac7227da9bd9c ec093dfb7c45343d689017507d485e62	59
4	f867aee8b437a5210c24c1974cffeabc 43efdb697244df808e8d9364ee0ae6f5	61
5	721eb200ba06206dcbd4bce704fa654e 7b28a5d5ed643287e006c099bb375302	68
6	0ad9d85689f9f77bc1c5f71185e5fb14 3bc2d8b6798d8ac4fe36a1d891ac181a	64
7	db18a8ffa16d30d5f88b08d777ba4eaa 9fb8b5452023c70280e5c4bb9e555a4b	67
8	f91b4fbfe934c9bf8f2f85812b084989 20264e1126b219aef7feb3f9b2d6de40	65
9	cca104a13e678500ff59025f3bafaa34 b56a0341b2290ba7dfdfbddcd8578205	61
10	ff0b844a0853bf7c6934ab4364148fb9 612b89398d0600cde116227ce72433f0	58

plaintext (or key) space to be searched. What is desired is the **avalanche effect**, in which a small change in plaintext or key produces a large change in the ciphertext.

Using the example from Table 6.5, Table 6.6 shows the result when the eighth bit of the plaintext is changed. The second column of the table shows the value of the **State** matrix at the end of each round for the two plaintexts. Note that after just one round, 20 bits of the **State** vector differ. After two rounds, close to half the bits differ. This magnitude of difference propagates through the remaining rounds. A bit difference in approximately half the positions in the most desirable outcome. Clearly, if almost all the bits are changed, this would be logically equivalent to almost none of the bits being changed. Put another way, if we select two plaintexts at random, we would expect the two plaintexts to differ in about half of the bit positions and the two ciphertexts to also differ in about half the positions.

Table 6.7 shows the change in **State** matrix values when the same plaintext is used and the two keys differ in the eighth bit. That is, for the second case, the key is `0e1571c947d9e8590cb7add6af7f6798`. Again, one round produces a significant change, and the magnitude of change after all subsequent rounds is roughly half the bits. Thus, based on this example, AES exhibits a very strong avalanche effect.

Table 6.7 Avalanche Effect in AES: Change in Key

Round		Number of Bits that Differ
	0123456789abcdeffedcba9876543210 0123456789abcdeffedcba9876543210	0
0	0e3634aece7225b6f26b174ed92b5588 0f3634aece7225b6f26b174ed92b5588	1
1	657470750fc7ff3fc0e8e8ca4dd02a9c c5a9ad090ec7ff3fc1e8e8ca4cd02a9c	22
2	5c7bb49a6b72349b05a2317ff46d1294 90905fa9563356d15f3760f3b8259985	58
3	7115262448dc747e5cdac7227da9bd9c 18aeb7aa794b3b66629448d575c7cebf	67
4	f867aee8b437a5210c24c1974cffeabc f81015f993c978a876ae017cb49e7eec	63
5	721eb200ba06206dcbd4bce704fa654e 5955c91b4e769f3cb4a94768e98d5267	81
6	0ad9d85689f9f77bc1c5f71185e5fb14 dc60a24d137662181e45b8d3726b2920	70
7	db18a8ffa16d30d5f88b08d777ba4eaa fe8343b8f88bef66cab7e977d005a03c	74
8	f91b4fbfe934c9bf8f2f85812b084989 da7dad581d1725c5b72fa0f9d9d1366a	67
9	cca104a13e678500ff59025f3bafaa34 0ccb4c66bbfd912f4b511d72996345e0	59
10	ff0b844a0853bf7c6934ab4364148fb9 fc8923ee501a7d207ab670686839996b	53

Note that this avalanche effect is stronger than that for DES (Table 4.2), which requires three rounds to reach a point at which approximately half the bits are changed, both for a bit change in the plaintext and a bit change in the key.

6.6 AES IMPLEMENTATION

Equivalent Inverse Cipher

As was mentioned, the AES decryption cipher is not identical to the encryption cipher (Figure 6.3). That is, the sequence of transformations for decryption differs from that for encryption, although the form of the key schedules for encryption and decryption is the same. This has the disadvantage that two separate software or firmware modules are needed for applications that require both encryption and decryption. There is, however, an equivalent version of the decryption algorithm that has the same structure as the encryption algorithm. The equivalent version has the same sequence of transformations as the encryption algorithm (with transformations replaced by their inverses). To achieve this equivalence, a change in key schedule is needed.

Two separate changes are needed to bring the decryption structure in line with the encryption structure. As illustrated in Figure 6.3, an encryption round has the structure SubBytes, ShiftRows, MixColumns, AddRoundKey. The standard decryption round has the structure InvShiftRows, InvSubBytes, AddRoundKey, InvMixColumns. Thus, the first two stages of the decryption round need to be interchanged, and the second two stages of the decryption round need to be interchanged.

Interchanging InvShiftRows and InvSubBytes InvShiftRows affects the sequence of bytes in **State** but does not alter byte contents and does not depend on byte contents to perform its transformation. InvSubBytes affects the contents of bytes in **State** but does not alter byte sequence and does not depend on byte sequence to perform its transformation. Thus, these two operations commute and can be interchanged. For a given **State** S_i,

$$\text{InvShiftRows}\,[\text{InvSubBytes}\,(S_i)] = \text{InvSubBytes}\,[\text{InvShiftRows}\,(S_i)]$$

Interchanging AddRoundKey and InvMixColumns The transformations AddRoundKey and InvMixColumns do not alter the sequence of bytes in **State**. If we view the key as a sequence of words, then both AddRoundKey and InvMixColumns operate on **State** one column at a time. These two operations are linear with respect to the column input. That is, for a given **State** S_i and a given round key w_j,

$$\text{InvMixColumns}\,(S_i \oplus w_j) = [\text{InvMixColumns}\,(S_i)] \oplus [\text{InvMixColumns}\,(w_j)]$$

To see this, suppose that the first column of **State** S_i is the sequence (y_0, y_1, y_2, y_3) and the first column of the round key w_j is (k_0, k_1, k_2, k_3). Then we need to show

$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} y_0 \oplus k_0 \\ y_1 \oplus k_1 \\ y_2 \oplus k_2 \\ y_3 \oplus k_3 \end{bmatrix} = \begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} \oplus \begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} k_0 \\ k_1 \\ k_2 \\ k_3 \end{bmatrix}$$

Let us demonstrate that for the first column entry. We need to show

$$\begin{aligned} &[\{0E\} \cdot (y_0 \oplus k_0)] \oplus [\{0B\} \cdot (y_1 \oplus k_1)] \oplus [\{0D\} \cdot (y_2 \oplus k_2)] \oplus [\{09\} \cdot (y_3 \oplus k_3)] \\ &\quad = [\{0E\} \cdot y_0] \oplus [\{0B\} \cdot y_1] \oplus [\{0D\} \cdot y_2] \oplus [\{09\} \cdot y_3] \oplus \\ &\quad\quad [\{0E\} \cdot k_0] \oplus [\{0B\} \cdot k_1] \oplus [\{0D\} \cdot k_2] \oplus [\{09\} \cdot k_3] \end{aligned}$$

This equation is valid by inspection. Thus, we can interchange AddRoundKey and InvMixColumns, provided that we first apply InvMixColumns to the round key. Note that we do not need to apply InvMixColumns to the round key for the input to the first AddRoundKey transformation (preceding the first round) nor to the last AddRoundKey transformation (in round 10). This is because these two AddRoundKey transformations are not interchanged with InvMixColumns to produce the equivalent decryption algorithm.

Figure 6.10 illustrates the equivalent decryption algorithm.

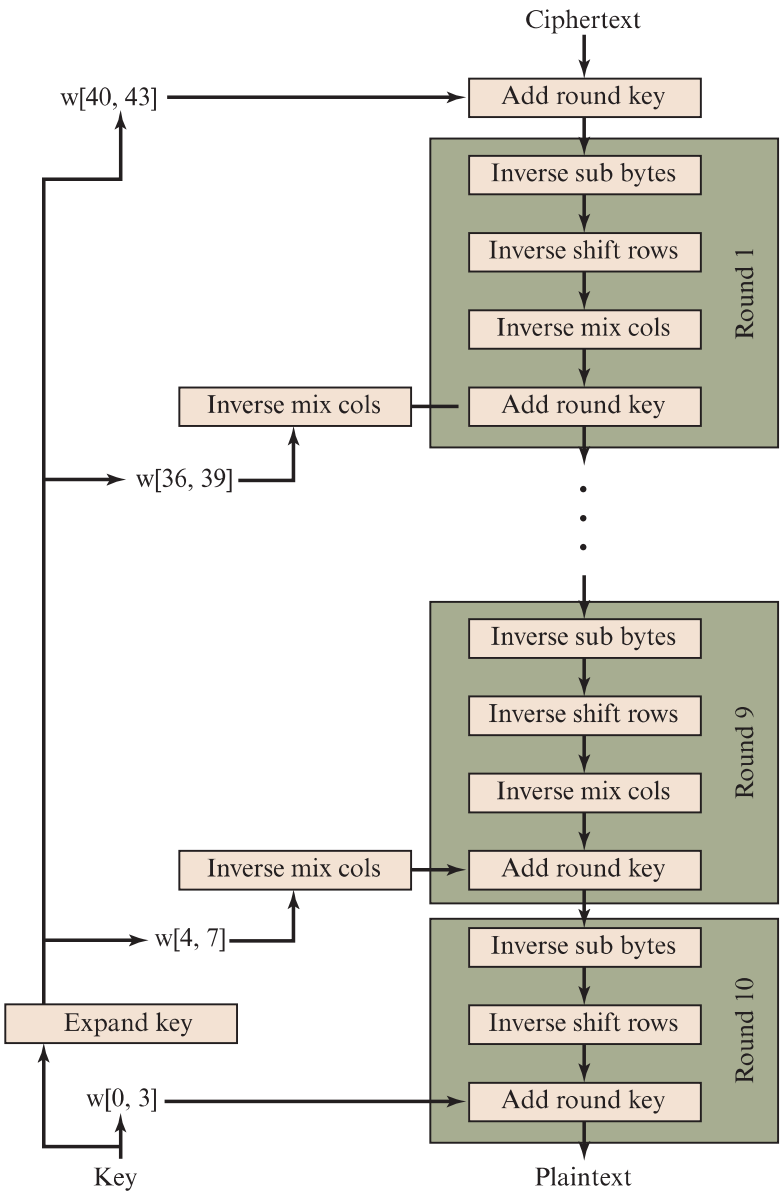


Figure 6.10 Equivalent Inverse Cipher

Implementation Aspects

The Rijndael proposal [DAEM99] provides some suggestions for efficient implementation on 8-bit processors, typical for current smart cards, and on 32-bit processors, typical for PCs.

8-Bit Processor AES can be implemented very efficiently on an 8-bit processor. AddRoundKey is a bytewise XOR operation. ShiftRows is a simple byte-shifting operation. SubBytes operates at the byte level and only requires a table of 256 bytes.

The transformation MixColumns requires matrix multiplication in the field GF(2^8), which means that all operations are carried out on bytes. MixColumns only requires multiplication by {02} and {03}, which, as we have seen, involved simple shifts, conditional XORs, and XORs. This can be implemented in a more efficient

way that eliminates the shifts and conditional XORs. Equation set (6.4) shows the equations for the MixColumns transformation on a single column. Using the identity $\{03\} \cdot x = (\{02\} \cdot x) \oplus x$, we can rewrite Equation set (6.4) as follows.

$$\begin{aligned} Tmp &= s_{0,j} \oplus s_{1,j} \oplus s_{2,j} \oplus s_{3,j} \\ s'_{0,j} &= s_{0,j} \oplus Tmp \oplus [2 \cdot (s_{0,j} \oplus s_{1,j})] \\ s'_{1,j} &= s_{1,j} \oplus Tmp \oplus [2 \cdot (s_{1,j} \oplus s_{2,j})] \\ s'_{2,j} &= s_{2,j} \oplus Tmp \oplus [2 \cdot (s_{2,j} \oplus s_{3,j})] \\ s'_{3,j} &= s_{3,j} \oplus Tmp \oplus [2 \cdot (s_{3,j} \oplus s_{0,j})] \end{aligned} \tag{6.9}$$

Equation set (6.9) is verified by expanding and eliminating terms.

The multiplication by {02} involves a shift and a conditional XOR. Such an implementation may be vulnerable to a timing attack of the sort described in Section 4.4. To counter this attack and to increase processing efficiency at the cost of some storage, the multiplication can be replaced by a table lookup. Define the 256-byte table X2, such that $X2[i] = \{02\} \cdot i$. Then Equation set (6.9) can be rewritten as

$$\begin{aligned} Tmp &= s_{0,j} \oplus s_{1,j} \oplus s_{2,j} \oplus s_{3,j} \\ s'_{0,j} &= s_{0,j} \oplus Tmp \oplus X2[s_{0,j} \oplus s_{1,j}] \\ s'_{1,c} &= s_{1,j} \oplus Tmp \oplus X2[s_{1,j} \oplus s_{2,j}] \\ s'_{2,c} &= s_{2,j} \oplus Tmp \oplus X2[s_{2,j} \oplus s_{3,j}] \\ s'_{3,j} &= s_{3,j} \oplus Tmp \oplus X2[s_{3,j} \oplus s_{0,j}] \end{aligned}$$

32-Bit Processor The implementation described in the preceding subsection uses only 8-bit operations. For a 32-bit processor, a more efficient implementation can be achieved if operations are defined on 32-bit words. To show this, we first define the four transformations of a round in algebraic form. Suppose we begin with a **State** matrix consisting of elements $a_{i,j}$ and a round-key matrix consisting of elements $k_{i,j}$. Then the transformations can be expressed as follows.

SubBytes	$b_{i,j} = \mathrm{S}[a_{i,j}]$
ShiftRows	$\begin{bmatrix} c_{0,j} \\ c_{1,j} \\ c_{2,j} \\ c_{3,j} \end{bmatrix} = \begin{bmatrix} b_{0,j} \\ b_{1,j-1} \\ b_{2,j-2} \\ b_{3,j-3} \end{bmatrix}$
MixColumns	$\begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} c_{0,j} \\ c_{1,j} \\ c_{2,j} \\ c_{3,j} \end{bmatrix}$
AddRoundKey	$\begin{bmatrix} e_{0,j} \\ e_{1,j} \\ e_{2,j} \\ e_{3,j} \end{bmatrix} = \begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix}$

In the ShiftRows equation, the column indices are taken mod 4. We can combine all of these expressions into a single equation:

$$\begin{bmatrix} e_{0,j} \\ e_{1,j} \\ e_{2,j} \\ e_{3,j} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} \mathrm{S}[a_{0,j}] \\ \mathrm{S}[a_{1,j-1}] \\ \mathrm{S}[a_{2,j-2}] \\ \mathrm{S}[a_{3,j-3}] \end{bmatrix} \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix}$$

$$= \left(\begin{bmatrix} 02 \\ 01 \\ 01 \\ 03 \end{bmatrix} \cdot \mathrm{S}[a_{0,j}] \right) \oplus \left(\begin{bmatrix} 03 \\ 02 \\ 01 \\ 01 \end{bmatrix} \cdot \mathrm{S}[a_{1,j-1}] \right) \oplus \left(\begin{bmatrix} 01 \\ 03 \\ 02 \\ 01 \end{bmatrix} \cdot \mathrm{S}[a_{2,j-2}] \right)$$

$$\oplus \left(\begin{bmatrix} 01 \\ 01 \\ 03 \\ 02 \end{bmatrix} \cdot \mathrm{S}[a_{3,j-3}] \right) \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix}$$

In the second equation, we are expressing the matrix multiplication as a linear combination of vectors. We define four 256-word (1024-byte) tables as follows.

$T_0[x] = \left(\begin{bmatrix} 02 \\ 01 \\ 01 \\ 03 \end{bmatrix} \cdot \mathrm{S}[x] \right)$	$T_1[x] = \left(\begin{bmatrix} 03 \\ 02 \\ 01 \\ 01 \end{bmatrix} \cdot \mathrm{S}[x] \right)$	$T_2[x] = \left(\begin{bmatrix} 01 \\ 03 \\ 02 \\ 01 \end{bmatrix} \cdot \mathrm{S}[x] \right)$	$T_3[x] = \left(\begin{bmatrix} 01 \\ 01 \\ 03 \\ 02 \end{bmatrix} \cdot \mathrm{S}[x] \right)$

Thus, each table takes as input a byte value and produces a column vector (a 32-bit word) that is a function of the S-box entry for that byte value. These tables can be calculated in advance.

We can define a round function operating on a column in the following fashion.

$$\begin{bmatrix} s'_{0,j} \\ s'_{1,j} \\ s'_{2,j} \\ s'_{3,j} \end{bmatrix} = T_0[s_{0,j}] \oplus T_1[s_{1,j-1}] \oplus T_2[s_{2,j-2}] \oplus T_3[s_{3,j-3}] \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix}$$

As a result, an implementation based on the preceding equation requires only four table lookups and four XORs per column per round, plus 4 Kbytes to store the table. The developers of Rijndael believe that this compact, efficient implementation was probably one of the most important factors in the selection of Rijndael for AES.

6.7 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

Advanced Encryption Standard (AES) avalanche effect	field finite field	key expansion S-box

Review Questions

6.1 What was the original set of criteria used by NIST to evaluate candidate AES ciphers?
6.2 What was the final set of criteria used by NIST to evaluate candidate AES ciphers?
6.3 How many different key sizes are approved for AES?
6.4 What is the purpose of the **State** array?
6.5 How is the S-box constructed?
6.6 What is the rationale behind the choice of the specific S-box in the AES?
6.7 Briefly describe ShiftRows.
6.8 How many bytes in **State** are affected by ShiftRows?
6.9 Why are different round constants used in the key expansion of AES?
6.10 Briefly describe AddRoundKey.
6.11 Briefly describe the key expansion algorithm.
6.12 What is the difference between SubBytes and SubWord?
6.13 What is the difference between ShiftRows and RotWord?
6.14 How is the avalanche effect different in AES in comparison to DES? Quantify it in terms of number of rounds.

Problems

6.1 In the discussion of MixColumns and InvMixColumns, it was stated that

$$b(x) = a^{-1}(x) \bmod (x^4 + 1)$$

where $a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$ and $b(x) = \{0B\}x^3 + \{0D\}x^2 + \{09\}x + \{0E.\}$ Show that this is true.

6.2 **a.** What is $\{02\}^{-1}$ in GF(2^8)?
b. Verify the entry for {02} in the S-box.

6.3 Show the first eight words of the key expansion for a 128-bit key of all ones.

6.4 Given the plaintext {0F0E0D0C0B0A09080706050403020100} and the key {02020202020202020202020202020202}:
a. Show the original contents of **State**, displayed as a 4 × 4 matrix.
b. Show the value of **State** after initial AddRoundKey.
c. Show the value of **State** after SubBytes.
d. Show the value of **State** after ShiftRows.
e. Show the value of **State** after MixColumns.

6.5 Verify Equation (6.11) in Appendix 6A. That is, show that $x^i \bmod (x^4 + 1) = x^{i \bmod 4}$.

6.6 For each of the following elements of DES, indicate the differences with the comparable element in AES.
- **a.** Key size
- **b.** Block size
- **c.** S-box
- **d.** Key expansion function
- **e.** Initial and final permutation

6.7 How are the coefficients chosen in the specific matrix used in the MixColumns?

6.8 In the subsection on implementation aspects, it is mentioned that AES can be implemented on 32-bit processors by using certain table lookups. Explain this technique and compute the overall cost of implementing one round of AES using the technique.

6.9 Compute the output of the MixColumns transformation for the following sequence of input bytes "A1 B2 C3 D4." Apply the InvMixColumns transformation to the obtained result to verify your calculations. Change the first byte of the input from "A1" to "A3", perform the MixColumns transformation again for the new input, and determine how many bits have changed in the output.

6.10 Use the key 1010 1001 1100 0011 to encrypt the plaintext "hi" as expressed in ASCII as 0110 1000 0110 1001. The designers of S-AES got the ciphertext 0011 1110 1111 1011. Did you?

6.11 Show that the matrix given here, with entries in GF(2^4), is the inverse of the matrix used in the MixColumns step of S-AES.

$$\begin{pmatrix} x^3 + 1 & x \\ x & x^3 + 1 \end{pmatrix}$$

6.12 Carefully write up a complete decryption of the ciphertext 0011 1110 1111 1011 using the key 1010 1001 1100 0011 and the S-AES algorithm. You should get the plaintext we started with in Problem 6.10. Note that the inverse of the S-boxes can be done with a reverse table lookup. The inverse of the MixColumns step is given by the matrix in the previous problem.

6.13 The decryption algorithm in AES uses a sequence of operations that is the reverse of the sequence used in the encryption algorithm. This has the disadvantage that different circuits or codes are required to implement the encryption and decryption functionality. Explain how it is possible to modify the decryption algorithm such that we can bring the decryption structure in line with the encryption structure in AES.

Programming Problems

6.1 Create software that can encrypt and decrypt using S-AES. *Test data*: A binary plaintext of 0110 1111 0110 1011 encrypted with a binary key of 1010 0111 0011 1011 should give a binary ciphertext of 0000 0111 0011 1000. Decryption should work correspondingly.

6.2 Implement a differential cryptanalysis attack on 1-round S-AES.

APPENDIX 6A POLYNOMIALS WITH COEFFICIENTS IN GF(2^8)

In Section 5.5, we discussed polynomial arithmetic in which the coefficients are in Z_p and the polynomials are defined modulo a polynomial $m(x)$ whose highest power is some integer n. In this case, addition and multiplication of coefficients occurred within the field Z_p; that is, addition and multiplication were performed modulo p.

The AES document defines polynomial arithmetic for polynomials of degree 3 or less with coefficients in GF(2^8). The following rules apply.

1. Addition is performed by adding corresponding coefficients in GF(2^8). As was pointed out Section 5.4, if we treat the elements of GF(2^8) as 8-bit strings, then addition is equivalent to the XOR operation. So, if we have

$$a(x) = a_3x^3 + a_2x^2 + a_1x + a_0 \tag{6.10}$$

and

$$b(x) = b_3x^3 + b_2x^2 + b_1x + b_0 \tag{6.11}$$

then

$$a(x) + b(x) = (a_3 \oplus b_3)x^3 + (a_2 \oplus b_2)x^2 + (a_1 \oplus b_1)x + (a_0 \oplus b_0)$$

2. Multiplication is performed as in ordinary polynomial multiplication with two refinements:
   a. Coefficients are multiplied in GF(2^8).
   b. The resulting polynomial is reduced mod $(x^4 + 1)$.

We need to keep straight which polynomial we are talking about. Recall from Section 5.6 that each element of GF(2^8) is a polynomial of degree 7 or less with binary coefficients, and multiplication is carried out modulo a polynomial of degree 8. Equivalently, each element of GF(2^8) can be viewed as an 8-bit byte whose bit values correspond to the binary coefficients of the corresponding polynomial. For the sets defined in this section, we are defining a polynomial ring in which each element of this ring is a polynomial of degree 3 or less with coefficients in GF(2^8), and multiplication is carried out modulo a polynomial of degree 4. Equivalently, each element of this ring can be viewed as a 4-byte word whose byte values are elements of GF(2^8) that correspond to the 8-bit coefficients of the corresponding polynomial.

We denote the modular product of $a(x)$ and $b(x)$ by $a(x) \oplus b(x)$. To compute $d(x) = a(x) \oplus b(x)$, the first step is to perform a multiplication without the modulo operation and to collect coefficients of like powers. Let us express this as $c(x) = a(x) \times b(x)$. Then

$$c(x) = c_6x^6 + c_5x^5 + c_4x^4 + c_3x^3 + c_2x^2 + c_1x + c_0 \tag{6.12}$$

where

$$c_0 = a_0 \cdot b_0$$
$$c_1 = (a_1 \cdot b_0) \oplus (a_0 \cdot b_1)$$
$$c_2 = (a_2 \cdot b_0) \oplus (a_1 \cdot b_1) \oplus (a_0 \cdot b_2)$$
$$c_3 = (a_3 \cdot b_0) \oplus (a_2 \cdot b_1) \oplus (a_1 \cdot b_2) \oplus (a_0 \cdot b_3)$$
$$c_4 = (a_3 \cdot b_1) \oplus (a_2 \cdot b_2) \oplus (a_1 \cdot b_3)$$
$$c_5 = (a_3 \cdot b_2) \oplus (a_2 \cdot b_3)$$
$$c_6 = a_3 \cdot b_3$$

The final step is to perform the modulo operation

$$d(x) = c(x) \bmod (x^4 + 1)$$

That is, $d(x)$ must satisfy the equation

$$c(x) = [(x^4 + 1) \times q(x)] \oplus d(x)$$

such that the degree of $d(x)$ is 3 or less.

A practical technique for performing multiplication over this polynomial ring is based on the observation that

$$x^i \bmod (x^4 + 1) = x^{i \bmod 4} \tag{6.13}$$

If we now combine Equations (6.12) and (6.13), we end up with

$$\begin{aligned} d(x) &= c(x) \bmod (x^4 + 1) \\ &= [c_6x^6 + c_5x^5 + c_4x^4 + c_3x^3 + c_2x^2 + c_1x + c_0] \bmod (x^4 + 1) \\ &= c_3x^3 + (c_2 \oplus c_6)x^2 + (c_1 \oplus c_5)x + (c_0 \oplus c_4) \end{aligned}$$

Expanding the c_i coefficients, we have the following equations for the coefficients of $d(x)$.

$$\begin{aligned} d_0 &= (a_0 \cdot b_0) \oplus (a_3 \cdot b_1) \oplus (a_2 \cdot b_2) \oplus (a_1 \cdot b_3) \\ d_1 &= (a_1 \cdot b_0) \oplus (a_0 \cdot b_1) \oplus (a_3 \cdot b_2) \oplus (a_2 \cdot b_3) \\ d_2 &= (a_2 \cdot b_0) \oplus (a_1 \cdot b_1) \oplus (a_0 \cdot b_2) \oplus (a_3 \cdot b_3) \\ d_3 &= (a_3 \cdot b_0) \oplus (a_2 \cdot b_1) \oplus (a_1 \cdot b_2) \oplus (a_0 \cdot b_3) \end{aligned}$$

This can be written in matrix form:

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{bmatrix} = \begin{bmatrix} a_0 & a_3 & a_2 & a_1 \\ a_1 & a_0 & a_3 & a_2 \\ a_2 & a_1 & a_0 & a_3 \\ a_3 & a_2 & a_1 & a_0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{6.14}$$

MixColumns Transformation

In the discussion of MixColumns, it was stated that there were two equivalent ways of defining the transformation. The first is the matrix multiplication shown in Equation (6.3), which is repeated here:

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

The second method is to treat each column of **State** as a four-term polynomial with coefficients in GF(2^8). Each column is multiplied modulo $(x^4 + 1)$ by the fixed polynomial $a(x)$, given by

$$a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$$

From Equation (6.10), we have $a_3 = \{03\}; a_2 = \{01\}; a_1 = \{01\}$; and $a_0 = \{02\}$. For the jth column of **State**, we have the polynomial $\text{col}_j(x) = s_{3,j}x^3 + s_{2,j}x^2 + s_{1,j}x + s_{0,j}$. Substituting into Equation (6.14), we can express $d(x) = a(x) \times \text{col}_j(x)$ as

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{bmatrix} = \begin{bmatrix} a_0 & a_3 & a_2 & a_1 \\ a_1 & a_0 & a_3 & a_2 \\ a_2 & a_1 & a_0 & a_3 \\ a_3 & a_2 & a_1 & a_0 \end{bmatrix} \begin{bmatrix} s_{0,j} \\ s_{1,j} \\ s_{2,j} \\ s_{3,j} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,j} \\ s_{1,j} \\ s_{2,j} \\ s_{3,j} \end{bmatrix}$$

which is equivalent to Equation (6.3).

Multiplication by *x*

Consider the multiplication of a polynomial in the ring by x: $c(x) = x \oplus b(x)$. We have

$$\begin{aligned} c(x) = x \oplus b(x) &= [x \times (b_3x^3 + b_2x^2 + b_1x + \text{b}_0] \bmod (x^4 + 1) \\ &= (b_3x^4 + b_2x^3 + b_1x^2 + \text{b}_0x) \bmod (x^4 + 1) \\ &= b_2x^3 + b_1x^2 + \text{b}_0x + b_3 \end{aligned}$$

Thus, multiplication by x corresponds to a 1-byte circular left shift of the 4 bytes in the word representing the polynomial. If we represent the polynomial as a 4-byte column vector, then we have

$$\begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} 00 & 00 & 00 & 01 \\ 01 & 00 & 00 & 00 \\ 00 & 01 & 00 & 00 \\ 00 & 00 & 01 & 00 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}$$