
CHAPTER 7

AES

(Solution to Practice Set)

Review Questions

1. The criteria defined by NIST for selecting AES fall into three areas: *security*, *cost*, and *implementation*.
2. The following table lists the parameters:

<i>Version</i>	<i>Block size</i>	<i>Key size</i>	<i>Round-key size</i>	<i>Number of rounds</i>
AES-128	128	128	128	10
AES-192	128	192	128	12
AES-256	128	256	128	14

3. The number of round keys and the number of transformation depend on the number of rounds. The following table list the information. Note that there is one transformation for pre-round. Also note that the last round uses only three transformations.

<i>Version</i>	<i>Number of rounds</i>	<i>Number of round keys</i>	<i>Number of Transformations</i>
AES-128	10	11	$1 + 9 \times 4 + 3 = 40$
AES-192	12	13	$1 + 11 \times 4 + 3 = 48$
AES-256	14	15	$1 + 13 \times 4 + 3 = 56$

4. DES is bit-oriented; AES is byte-oriented. In DES, an S-box takes 6 bits and transforms it to 4 bits; in AES, SubBytes transformation takes a byte and change it to another byte. In DES, P-boxes transpose bits; in AES the ShiftRows transformation transposes bytes. To provide mixing of bits inside a byte, AES uses the Mix-Columns transformation.

5. Before and after each stage, the data block is referred to as a state. States, like blocks, are made of 16 bytes, but they are normally treated as matrices of 4×4 bytes. The following table shows the number of states used in each version. We have used two extra states: one before the pre-round and one after the last round. If you do not consider this, the number of states is reduced by two.

<i>Version</i>	<i>Number of rounds</i>	<i>Number of States</i>
AES-128	10	(1) + 1 + 9×4 + 3 + (1) = 42
AES-192	12	(1) + 1 + 11×4 + 3 + (1) = 50
AES-256	14	(1) + 1 + 13×4 + 3 + (1) = 58

6. The SubBytes, MixColumns, and AddRoundKey transformation change the contents of bytes; the ShiftRows transformation does not.
7. Substitution in DES is done by S-boxes. Each box substitutes a 6-bit value with a 4-bit value. We need eight S-boxes to create a 32-bit half block. Substitution in AES is done through SubBytes transformation that transforms a whole state to another state. However, we can say that SubBytes actually substitutes 16 bytes with new 16 bytes.
8. Permutation in DES is applied in two steps: before substitution and after substitution. The first is an expansion permutation to create a 48-bit half block out of a 32-bit half block. This is needed because the round key is 48 bits long. Permutation in AES is straight permutation of bytes. Since the size of the block and the size of the round key are the same, there is no need for an expansion permutation.
8. Permutation in DES is applied in two steps: before substitution and after substitution. The first is an expansion permutation to create a 48-bit half block out of a 32-bit half block. This is needed because the round key is 48 bits long. Permutation in AES is straight permutation of bytes. Since the size of the block and the size of the round key are the same, there is no need for an expansion permutation.
9. In DES, the size of the block is 64 bits, but the size of the round key is 48 bits. In AES the size of the block and the round key are both 128 bits (for all versions).
10. In DES, permutation is bit-oriented; in AES, permutation is byte oriented. To permute the bits inside a byte, AES uses the MixColumns transformation.

Exercises

11. The disadvantage of using keyed S-boxes is that it makes the design of the cipher more difficult. In particular, it is more difficult to create S-boxes that are inverse of each other in the encryption and decryption cipher. The advantage of using keyed S-boxes is that a keyed S-box is normally non-linear, which protect the cipher against linear cryptanalysis.
12. A larger block size creates more mixing of bits in each block. In a cipher with a block size of 64, each bit, in average, is dependent on only 32 other bits; in a block size of 128 bits, each bit depends, on average, on 64 bits. This is definitely an advantage. One disadvantage is that the large block size creates an overhead of adding more padding to the last block. In particular, if the size of the message is very small and less than the size of a block, most of the plaintext is made of padding.

13. Having different number of rounds has the advantage that new versions of cipher with more number of rounds can be used, without changing the structure of cipher, if the cipher is attacked by differential and linear cryptanalysis (or other attacks that depend on the number of rounds). For example, we can use AES with 10 rounds as long as it is not secured. If it is attacked, we can move to the version with 12 or 14 rounds without changing the structure of the cipher.
14. Having different key size has the advantage that new versions of cipher with larger key size can be used, without changing the structure of cipher, if the cipher is attacked by brute-force cryptanalysis. For example, we can use AES with a 128-bit cipher key as long as it is not attacked. If it is attacked, we can move to the version with 192-bit or 256-bit cipher key.
15. A cipher in which the size of the round is the same as the size of the round key is easier to design because we do not have to use expansion (or compression) permutation to match the size of the block to the size of the round key. This enable us to make the non-Feistel ciphers.

16.

- a. The state has 16 bytes which are permuted by ShiftRows transformation using the following permutation table:

01	02	03	04	06	07	08	05	11	12	09	10	16	13	14	15
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

- b. The state has 16 bytes which are permuted by InvShiftRows transformation using the following permutation table:

01	02	03	04	08	05	06	07	11	12	09	10	14	15	16	13
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

- c. We can prove that two transformation are inverse of each other by applying the the process we learned in Chapter 3. We add indexes, we swap indexes and contents, and then sort the table according to the indexes. We start with the ShiftRows table to get InvShiftRows table. After adding the index, we have

01	02	03	04	06	07	08	05	11	12	09	10	16	13	14	15
01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16

After swapping the contents and the indexes, we have

01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16
01	02	03	04	06	07	08	05	11	12	09	10	16	13	14	15

After sorting the table according to the index, we have the following table which is the same as the InvShiftRows table found in part b.

01	02	03	04	08	05	06	07	11	12	09	10	14	15	16	13
01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16

17. We use two plaintexts that differ only in the first bit:

P₁: (0000 0000 0000 0000 0000 0000 0000 0000)₁₆

P₂: (8000 0000 0000 0000 0000 0000 0000 0000)₁₆

- a. Applying the SubBytes transformation to P₁ and P₂, we get:

T₁: (6363 6363 6363 6363 6363 6363 6363 6363)₁₆

T₂: (CD63 6363 6363 6363 6363 6363 6363 6363)₁₆

T₂ and T₁ differ only in 5 bits.

- b. Applying the ShiftRows transformation to P₁ and P₂, we get:

T₁: (0000 0000 0000 0000 0000 0000 0000 0000)₁₆

T₂: (8000 0000 0000 0000 0000 0000 0000 0000)₁₆

T₂ and T₁ differ only in 1 bit.

Applying the MixColumn transformation to P₁ and P₂, we get:

T₁: (0000 0000 0000 0000 0000 0000 0000 0000)₁₆

T₂: (1B80 809B 0000 0000 0000 0000 0000 0000)₁₆

T₂ and T₁ differ in 9 bits.

- c. Applying the AddRoundKey of 128 0-bit transformation to P₁ and P₂, we get:

T₁: (0000 0000 0000 0000 0000 0000 0000 0000)₁₆

T₂: (8000 0000 0000 0000 0000 0000 0000 0000)₁₆

T₂ and T₁ differ only in 1 bit. Note that we have used a round key with 128 0's, but the result is the same if we use any key.

18. We have

SubBytes (0x57 $\oplus$ 0xA2) = SubBytes (0xF5) = **0xE6**

SubBytes (0x57) $\oplus$ SubBytes (0xA2) = 0x5B $\oplus$ 0x3A = **0x61**

Definitely, the two are different; the SubByte transformation is non-linear

19.

- a. The SubBytes transformation is repeated in each round. So we have $\mathcal{N}_r$ of this transformation.
- b. The ShiftRows transformation is repeated in each round. So we have $\mathcal{N}_r$ of this transformation.
- c. The MixColumns transformation is repeated in each round except the last round. So we have $(\mathcal{N}_r - 1)$ of this transformation.

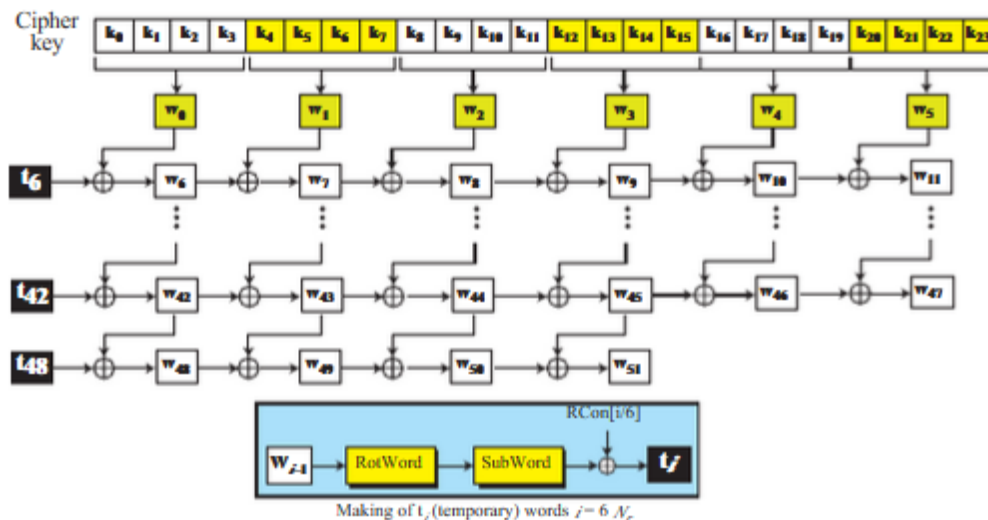
- d. The AddRoundKey transformation is repeated in each round. In addition, we have one of this transformation in the pre-round section. So we have $(N_r + 1)$ of this transformation.
- e. The total number of transformation is

$$\text{Total number of transformation} = N_r + N_r + (N_r - 1) + (N_r + 1) = 4 \times N_r$$

20.

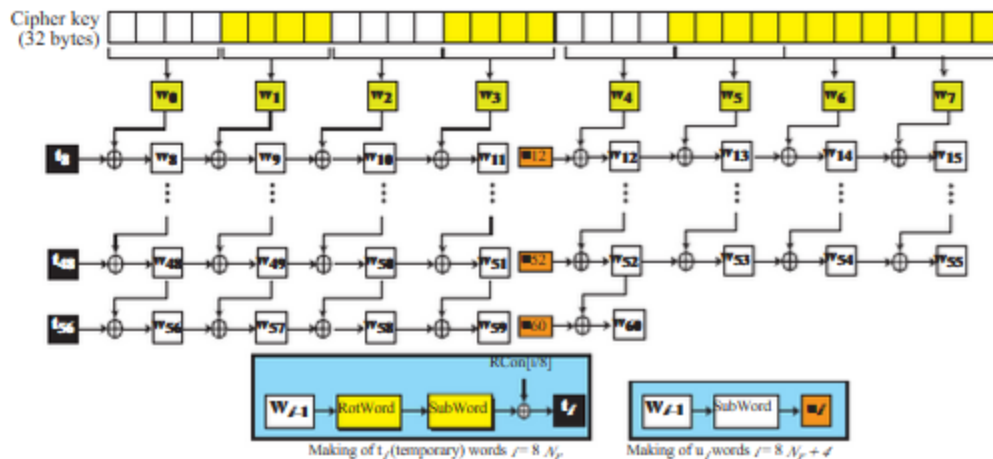
- a. Figure S7.20a shows the key expansion in AES-192.

Figure S7.20a Solution to Exercise 20.a



- b. Figure S7.20b shows the key expansion in AES-256

Figure S7.20b Solution to Exercise 20.b



21.

- a. We can use $(x^{11-1} \bmod \text{prime})$ and $(x^{12-1} \bmod \text{prime})$, in which the prime is the irreducible polynomial $(x^8 + x^4 + x^3 + x + 1)$, to find the first terms of RCon[11] and RCon[12]. The following shows the constants for AES-192.

Round	(RCon)	Round	(RCon)	Round	(RCon)
1	(01 00 00 00) ₁₆	5	(10 00 00 00) ₁₆	9	(1B 00 00 00) ₁₆
2	(02 00 00 00) ₁₆	6	(20 00 00 00) ₁₆	10	(36 00 00 00) ₁₆
3	(04 00 00 00) ₁₆	7	(40 00 00 00) ₁₆	11	(6C 00 00 00) ₁₆
4	(08 00 00 00) ₁₆	8	(80 00 00 00) ₁₆	12	(D8 00 00 00) ₁₆

- b. We can use $(x^{13-1} \bmod \text{prime})$ and $(x^{14-1} \bmod \text{prime})$, in which the prime is the irreducible polynomial $(x^8 + x^4 + x^3 + x + 1)$, to find the first terms of RCon[13] and RCon[14]. The following shows the constants for AES-256.

Round	(RCon)	Round	(RCon)	Round	(RCon)
1	(01 00 00 00) ₁₆	6	(20 00 00 00) ₁₆	11	(6C 00 00 00) ₁₆
2	(02 00 00 00) ₁₆	7	(40 00 00 00) ₁₆	12	(D8 00 00 00) ₁₆
3	(04 00 00 00) ₁₆	8	(80 00 00 00) ₁₆	13	(AB 00 00 00) ₁₆
4	(08 00 00 00) ₁₆	9	(1B 00 00 00) ₁₆	14	(4D 00 00 00) ₁₆
5	(10 00 00 00) ₁₆	10	(36 00 00 00) ₁₆		

22.

- a. The pre-round operation needs a four-word round key. The cipher key in AES-192 is six words; only the first four words of it is used in the pre-round operation.
 - b. The pre-round operation needs a four-word round key. The cipher key in AES-256 is eight words; only the first four words of it is used in the pre-round operation.
23. The result is an identity matrix as shown below. Note that the addition and multiplication of elements are in GF(2).

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

$X \qquad X^{-1} \qquad X \times X^{-1}$

24. The result is an identity matrix as shown below. Note that the addition and multiplication of coefficients are in GF(2).

$$\begin{bmatrix} x & x+1 & 1 & 1 \\ 1 & x & x+1 & 1 \\ 1 & 1 & x & x+1 \\ x+1 & 1 & 1 & x \end{bmatrix} \times \begin{bmatrix} x^3+x^2+x & x^3+x+1 & x^3+x^2+1 & x^3+1 \\ x^3+1 & x^3+x^2+x & x^3+x+1 & x^3+x^2+1 \\ x^3+x^2+1 & x^3+1 & x^3+x^2+x & x^3+x+1 \\ x^3+x+1 & x^3+x^2+1 & x^3+1 & x^3+x^2+x \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$C \qquad C^{-1} \qquad C \times C^{-1}$

The identity matrix can be easily found. For example, if we multiply the first row of C by the first column of C^{-1} , we get

$$(x)(x^3+x^2+x) + (x+1)(x^3+1) + (1)(x^3+x^2+1) + (1)(x^3+x+1) = 1$$

But if we multiply the second row of C by the first column of C^{-1} , we get

$$(1)(x^3+x^2+x) + (x)(x^3+1) + (x+1)(x^3+x^2+1) + (1)(x^3+x+1) = 0$$

25. Most of the code matches, line by line, with the steps in the process. The only section that needs some explanation is the loop. The iterations of the loop gives us

$\mathbf{c}_0 = \mathbf{b}_0 \oplus \mathbf{b}_4 \oplus \mathbf{b}_5 \oplus \mathbf{b}_6 \oplus \mathbf{b}_7$	$\rightarrow$	$\mathbf{c}_0 = \mathbf{b}_0 \oplus \mathbf{b}_4 \oplus \mathbf{b}_5 \oplus \mathbf{b}_6 \oplus \mathbf{b}_7$
$\mathbf{c}_1 = \mathbf{b}_1 \oplus \mathbf{b}_5 \oplus \mathbf{b}_6 \oplus \mathbf{b}_7 \oplus \mathbf{b}_0$	$\rightarrow$	$\mathbf{c}_1 = \mathbf{b}_0 \oplus \mathbf{b}_1 \oplus \mathbf{b}_5 \oplus \mathbf{b}_6 \oplus \mathbf{b}_7$
$\mathbf{c}_2 = \mathbf{b}_2 \oplus \mathbf{b}_6 \oplus \mathbf{b}_7 \oplus \mathbf{b}_0 \oplus \mathbf{b}_1$	$\rightarrow$	$\mathbf{c}_2 = \mathbf{b}_0 \oplus \mathbf{b}_1 \oplus \mathbf{b}_2 \oplus \mathbf{b}_6 \oplus \mathbf{b}_7$
$\mathbf{c}_3 = \mathbf{b}_3 \oplus \mathbf{b}_7 \oplus \mathbf{b}_0 \oplus \mathbf{b}_1 \oplus \mathbf{b}_2$	$\rightarrow$	$\mathbf{c}_3 = \mathbf{b}_0 \oplus \mathbf{b}_1 \oplus \mathbf{b}_2 \oplus \mathbf{b}_3 \oplus \mathbf{b}_7$
$\mathbf{c}_4 = \mathbf{b}_4 \oplus \mathbf{b}_0 \oplus \mathbf{b}_1 \oplus \mathbf{b}_2 \oplus \mathbf{b}_3$	$\rightarrow$	$\mathbf{c}_4 = \mathbf{b}_0 \oplus \mathbf{b}_1 \oplus \mathbf{b}_2 \oplus \mathbf{b}_3 \oplus \mathbf{b}_4$
$\mathbf{c}_5 = \mathbf{b}_5 \oplus \mathbf{b}_1 \oplus \mathbf{b}_2 \oplus \mathbf{b}_3 \oplus \mathbf{b}_4$	$\rightarrow$	$\mathbf{c}_5 = \mathbf{b}_1 \oplus \mathbf{b}_2 \oplus \mathbf{b}_3 \oplus \mathbf{b}_4 \oplus \mathbf{b}_5$
$\mathbf{c}_6 = \mathbf{b}_6 \oplus \mathbf{b}_2 \oplus \mathbf{b}_3 \oplus \mathbf{b}_4 \oplus \mathbf{b}_5$	$\rightarrow$	$\mathbf{c}_6 = \mathbf{b}_2 \oplus \mathbf{b}_3 \oplus \mathbf{b}_4 \oplus \mathbf{b}_5 \oplus \mathbf{b}_6$
$\mathbf{c}_7 = \mathbf{b}_7 \oplus \mathbf{b}_3 \oplus \mathbf{b}_4 \oplus \mathbf{b}_5 \oplus \mathbf{b}_6$	$\rightarrow$	$\mathbf{c}_7 = \mathbf{b}_3 \oplus \mathbf{b}_4 \oplus \mathbf{b}_5 \oplus \mathbf{b}_6 \oplus \mathbf{b}_7$

The rearranged code is the result of matrix multiplication $\mathbf{c} = \mathbf{X} \times \mathbf{b}$ if we ignore the zero terms. After each line is created $\mathbf{d} = \mathbf{c} + \mathbf{y}$ is made.

26.

- a. The **invByte** routine calls two other routine: **multiply** and **quotation**. The first routine multiplies two bytes; the second routine finds the quotient of dividing the first byte by the second. The **quotation** routine calls the **degree** routine which finds the degree of a byte (as a polynomial)

```

invByte(byte)
{
    if (byte = 0)
        return byte
     $r_1 \leftarrow (11\text{B})_{16}$  // modulus
     $r_2 \leftarrow \text{byte}$ 
     $t_1 \leftarrow (00)_{16}$ 
     $t_2 \leftarrow (01)_{16}$ 
    while ( $r_2 > (00)_{16}$ )
    {
         $q \leftarrow \text{quotation}(r_1, r_2)$ 
         $r \leftarrow r_1 \oplus \text{multiply}(q, r_2)$ 
         $t \leftarrow t_1 \oplus \text{multiply}(q, t_2)$ 
         $r_1 \leftarrow r_2$     $r_2 \leftarrow r$     $t_1 \leftarrow t_2$     $t_2 \leftarrow t$ 
    }
    return  $t_1$ 
}

```

multiply(byte₁, byte₂)

```
{  
     $i \leftarrow (00)_{16}$   
     $res \leftarrow (00)_{16}$   
    while (byte2  $\neq$  (00)16)  
    {  
        if [(byte2 AND (01)16) = (01)16]  
             $res \leftarrow res \oplus \text{shiftLeft}(i, \text{byte}_1)$   
        byte2  $\leftarrow \text{shiftRight}(1, \text{byte}_2)$   
    }  
    return  $res$   
}
```

Quotation(byte₁, byte₂)

```
{  
    degreeDiff  $\leftarrow \text{degree}(\text{byte}_1) - \text{degree}(\text{byte}_2)$   
    while (degreeDiff  $\geq 0$ )  
    {  
        temp  $\leftarrow \text{shiftLeft}(\text{degreeDiff}, 1)$ 
```

```

        byte1 ← byte1 ⊕ shiftLeft(degreeDiff, byte2)
        res ← res OR temp
    }
    return res
}

```

```

degree(byte)
{
    deg ← -1
    while(byte > 0)
    {
        deg ← deg + 1
        byte ← shiftLeft(1, byte)
    }
    return deg
}

```

b.

```

ByteToMatrix(byte, matrix)
{
    for (r = 7 downto 0)
    {
        matrix[r] ← byte mod 2
        byte ← byte / 2           // integer division
    }
}

```

c.

```

MatrixToByte(matrix, byte)
{
    byte ← 0
    for (r = 7 downto 0)
    {
        byte ← byte × 2 + matrix[r]
    }
}

```

27.

```
InvSubBytes(S[0 ... 3][0 ... 3])
{
    for ( $r = 0$  to 3)
        for ( $c = 0$  to 3)
             $S[r][c] \leftarrow \text{invsubbyte}(S[r][c])$ 

    return(S)
}
```

```
invsubbyte(byte)
{
     $\mathbf{d} \leftarrow \text{ByteToMatrix}(\text{byte})$ 
     $\mathbf{c} \leftarrow \mathbf{d} \oplus \text{ByteToMatrix}(0x63)$ 
     $\mathbf{b} \leftarrow \mathbf{X}^{-1} \times \mathbf{c}$ 
     $\mathbf{a} \leftarrow \text{MatrixToByte}(\mathbf{b})$ 
    return ( $\mathbf{a}^{-1}$ )
}
```

28. The **ShiftRows** algorithm calls the **shiftrow** routine three times (the first row is not shifted) to shift each row. The following shows how shifting is done:

First call $n = 1$	$c = 0$	$\text{row}_3 \leftarrow t_0$
	$c = 1$	$\text{row}_0 \leftarrow t_1$
	$c = 2$	$\text{row}_1 \leftarrow t_2$
	$c = 3$	$\text{row}_2 \leftarrow t_3$
Second call $n = 2$	$c = 0$	$\text{row}_2 \leftarrow t_0$
	$c = 1$	$\text{row}_3 \leftarrow t_1$
	$c = 2$	$\text{row}_0 \leftarrow t_2$
	$c = 3$	$\text{row}_1 \leftarrow t_3$
Third call $n = 3$	$c = 0$	$\text{row}_3 \leftarrow t_0$
	$c = 1$	$\text{row}_2 \leftarrow t_1$
	$c = 2$	$\text{row}_0 \leftarrow t_2$
	$c = 3$	$\text{row}_1 \leftarrow t_3$

29.

```
CopyRow(row[0 ... 3], t[0 ... 3])
{
    for ( $c = 0$  to 3)
         $t[c] \leftarrow \text{row}[c]$ 
}
```

30.

```

InvShiftRows(S[0 ... 3][0 ... 3])
{
    for (r = 0 to 3)
        invshiftrow (S[r], r)
}

```

```

invshiftrow(row[0 ... 3], r)
{
    CopyRow(row, t)
    for (c = 0 to 3)
        row[(c + n) mod 4] ← t[c]
}

```

31. The **MixColumns** algorithm calls the **mixcolumn** routine four times, once for each column of the old state to create the correspond column of the new state. The **mixcolumn** routine actually performs the following matrix multiplication $\mathbf{col} = \mathbf{C} \times \mathbf{t}$, in which **col** is the new column matrix, **t** is the old column matrix, and the **C** is the square constant matrix. We can write the code in the **mixcolumn** routine as

$$\begin{array}{llll}
 \mathbf{col}_0 & \leftarrow & \mathbf{C}_{00} \bullet \mathbf{t}_0 \oplus & \mathbf{C}_{01} \bullet \mathbf{t}_1 \oplus & \mathbf{C}_{02} \bullet \mathbf{t}_2 \oplus & \mathbf{C}_{03} \bullet \mathbf{t}_3 \\
 \mathbf{col}_1 & \leftarrow & \mathbf{C}_{10} \bullet \mathbf{t}_0 \oplus & \mathbf{C}_{11} \bullet \mathbf{t}_1 \oplus & \mathbf{C}_{12} \bullet \mathbf{t}_2 \oplus & \mathbf{C}_{13} \bullet \mathbf{t}_3 \\
 \mathbf{col}_2 & \leftarrow & \mathbf{C}_{20} \bullet \mathbf{t}_0 \oplus & \mathbf{C}_{21} \bullet \mathbf{t}_1 \oplus & \mathbf{C}_{22} \bullet \mathbf{t}_2 \oplus & \mathbf{C}_{23} \bullet \mathbf{t}_3 \\
 \mathbf{col}_3 & \leftarrow & \mathbf{C}_{30} \bullet \mathbf{t}_0 \oplus & \mathbf{C}_{31} \bullet \mathbf{t}_1 \oplus & \mathbf{C}_{32} \bullet \mathbf{t}_2 \oplus & \mathbf{C}_{33} \bullet \mathbf{t}_3
 \end{array}$$

32.

```

CopyColumn(column[0 ... 3], t[0 ... 3])
{
    for (r = 0 to 3)
        t[r] ← column[r]
}

```

33.

```

MixColumns(S[0 ... 3][0 ... 3])
{
    for (c = 0 to 3)
        mixcolumn(S[c])
}

```

```

mixcolumn(col[0 ... 3])
{
    CopyColumn(col, t)
    col[0] ← MultField(0x02, t[0]) ⊕ MultField(0x03, t[1]) ⊕ t[2] ⊕ t[3]
    col[1] ← t[0] ⊕ MultField(0x02, t[1]) ⊕ MultField(0x03, t[2]) ⊕ t[3]
    col[2] ← t[0] ⊕ t[1] ⊕ MultField(0x02, t[2]) ⊕ MultField(0x03, t[3])
    col[3] ← MultField(0x03, t[0]) ⊕ t[1] ⊕ t[2] ⊕ MultField(0x02, t[3])
}

```

34.

```

InvMixColumns(S[0 ... 3][0 ... 3])
{
    for (c = 0 to 3)
        invmixcolumn(S[c])
}

```

```

invmixcolumn(col[0 ... 3])
{
    CopyColumn(col, t)
    col[0] ← 0x0E • t[0] ⊕ (0x0B • t[1]) ⊕ (0x0D • t[2]) ⊕ (0x09 • t[3])
    col[1] ← 0x09 • t[0] ⊕ (0x0E • t[1]) ⊕ (0x0B • t[2]) ⊕ (0x0D • t[3])
    col[2] ← 0x0D • t[0] ⊕ (0x09 • t[1]) ⊕ (0x0E • t[2]) ⊕ (0x0B • t[3])
    col[3] ← 0x0B • t[0] ⊕ (0x0D • t[1]) ⊕ (0x09 • t[2]) ⊕ (0x0E • t[3])
}

```

35. The algorithm uses a loop that iterates four times, one for each column of the current state. In each iteration, a column of the old state is exclusive-ored with a key word to create a new column.

$$\mathbf{S}[c] \leftarrow \mathbf{S}[c] \oplus \mathbf{W}[4 \times \text{round} + c]$$

For example, in round 5, we have

```

S[0]  ←  S[0] ⊕ W[20]
S[1]  ←  S[1] ⊕ W[21]
S[2]  ←  S[2] ⊕ W[22]
S[3]  ←  S[3] ⊕ W[23]

```

36.

- a. The subbyte routine called by the SubWord is the one defined in the text.

```

SubWord(W[0 ... 3])
{
    for ( $i = 0$  to 3)
         $W[i] \leftarrow \text{subbyte}(W[i])$ 
    return (W)
}

```

b.

```

RotWord(W[0 ... 3])
{
    CopyRow(W, t)
    for ( $i = 0$  to 3)
         $W[(i + 3) \bmod 4] \leftarrow t[i]$ 
    return (W)
}

```

37.

a.

```

KeyExpansion(Key[0 ... 23], W[0 ... 51])
{
    for ( $i = 0$  to 5)
         $W[i] \leftarrow \text{Key}[4i] \mid \text{Key}[4i+1] \mid \text{Key}[4i+2] \mid \text{Key}[4i+3]$ 
    for ( $i = 6$  to 51)
        if ( $i \bmod 6 = 0$ )
        {
             $t \leftarrow \text{subword}(\text{rotWord}(W[i-1]) \oplus \text{Rcon}[i/6])$ 
             $W[i] \leftarrow t \oplus W[i-6]$ 
        }
        else
             $W[i] \leftarrow W[i-1] \oplus W[i-6]$ 
}

```

b.

```

KeyExpansion(Key[0 ... 31], W[0 ... 59])
{
    for ( $i = 0$  to 7)
         $W[i] \leftarrow \text{Key}[4i] \mid \text{Key}[4i+1] \mid \text{Key}[4i+2] \mid \text{Key}[4i+3]$ 
    for ( $i = 8$  to 59)

```

```

    if ( $i \bmod 8 = 0$ )
    {
         $t \leftarrow \text{subword}(\text{rotWord}(W[i-1]) \oplus \text{Rcon}[i/8])$ 
         $W[i] \leftarrow t \oplus W[i-8]$ 
    }
    if ( $i \bmod 8 = 4$ )
    {
         $t \leftarrow \text{subword}(W[i-1])$ 
         $W[i] \leftarrow t \oplus W[i-8]$ 
    }
    else
         $W[i] \leftarrow W[i-1] \oplus W[i-8]$ 
}

```

38. The following algorithm modifies the keys created for the original design to create the round keys for the alternate design.

```

KeyExpansionAlternative(Key[0 ... 15])
{
    KeyExpansion Key[0 ... 15], W[0 ... 43]
    for ( $rnd = 1$  to 9)
    {
        for ( $r = 0$  to 3)
             $t[r] \leftarrow W[rnd \times 4 + r]$ 
             $W[rnd \times 4 + 0] \leftarrow 0x0E \bullet t[0] \oplus 0x0B \bullet t[1] \oplus 0x0D \bullet t[2] \oplus 0x09 \bullet t[3]$ 
             $W[rnd \times 4 + 1] \leftarrow 0x09 \bullet t[0] \oplus 0x0E \bullet t[1] \oplus 0x0B \bullet t[2] \oplus 0x0D \bullet t[3]$ 
             $W[rnd \times 4 + 2] \leftarrow 0x0D \bullet t[0] \oplus 0x09 \bullet t[1] \oplus 0x0E \bullet t[2] \oplus 0x0B \bullet t[3]$ 
             $W[rnd \times 4 + 3] \leftarrow 0x0B \bullet t[0] \oplus 0x0D \bullet t[1] \oplus 0x09 \bullet t[2] \oplus 0x0E \bullet t[3]$ 
    }
    return(W[0 ... 43])
}

```

39.

```

InvCipher(InBlock[0 ... 16], outBlock[0 ... 16], W[0 ... 43])
{
    BlockToState(Inblock S)
     $S \leftarrow \text{AddRoundKey}(S, W[40 \dots 43])$ 
    for ( $r = 1$  to 10)                //  $r$  defines the round

```

```

{
    S ← InvShiftRows(S)
    S ← InvSubBytes(S)
    S ← AddRoundKey(S, W[(10 - r) × 4 ... (10 - r) × 4 + 3])
    if (r ≠ 10)
        S ← InvMixColumns(S)
}
StateToBlock(S, outBlock)
}

```

40.

```

InvCipherAlternate(InBlock[0 ... 16], outBlock[0 ... 16], W[0 ... 43])
{
    BlockToState(InBlock, S)
    S ← AddRoundKey(S, W[40 ... 43])
    for (r = 1 to 9) // r defines the round
    {
        S ← InvSubBytes(S)
        S ← InvShiftRows(S)
        S ← InvMixColumns(S)
        S ← InvAddRoundKey(S, W[(10 - r) × 4 ... (10 - r) × 4 + 3])
    }
    S ← InvSubBytes(S)
    S ← InvShiftRows(S)
    S ← AddRoundKey(S, W[0 ... 3])
    StateToBlock(S, outBlock)
}

```